Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.

Undirected graphs

Graph. Set of vertices connected pairwise by edges.

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

Protein-protein interaction network

Reference: Jeong et al, Nature Review | Genetics
The Internet as mapped by the Opte Project

Map of science clickstreams

10 million Facebook friends

Framingham heart study

Educational level; the ego's obesity status at the previous time point (t); and most pertinent, the alter's obesity status at times t and t + 1.

We used generalized estimating equations to account for multiple observations of the same ego across examinations and across ego–alter pairs.

We assumed an independent working correlation structure for the clusters.

The use of a time-lagged dependent variable (lagged to the previous examination) eliminated serial correlation in the errors (evaluated with a Lagrange multiplier test) and also substantially controlled for the ego's genetic endowment and any intrinsic, stable predisposition to obesity. The use of a lagged independent variable for an alter's weight status controlled for homophily.

The key variable of interest was an alter's obesity at time t + 1. A significant coefficient for this variable would suggest either that an alter's weight affected an ego's weight or that an ego and an alter experienced contemporaneous events affecting both their weights. We estimated these models in varied ego–alter pair types.

To evaluate the possibility that omitted variables or unobserved events might explain the associations, we examined how the type or direction of the social relationship between the ego and the alter affected the association between the ego's obesity and the alter's obesity. For example, if unobserved factors drove the association between the ego's obesity and the alter's obesity, then the directionality of friendship should not have been relevant.

We evaluated the role of a possible spread in smoking-cessation behavior as a contributor to the spread of obesity by adding variables for the smoking status of egos and alters at times t and t + 1 to the foregoing models. We also analyzed the role of geographic distance between egos and alters by adding such a variable.

We calculated 95% confidence intervals by simulating the first difference in the alter's contemporaneous weight statuses.
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.

Graph representation

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

Anomalies.
- Parallel edges
- Self-loop

Graph representation

Graph API: sample client

Graph input format.

public class Graph
{
    Graph(int V)
    Graph(In in)
    void addEdge(int v, int w)
    Iterable<Integer> adj(int v)
    int V()
    int E()
    String toString()
}

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

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

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

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

```java
public static int degree(Graph 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 double averageDegree(Graph G) {
    return 2.0 * G.E() / G.V();
}

public static int numberOfSelfLoops(Graph G) {
    int count = 0;
    for (int v = 0; v < G.V(); v++)
        for (int w : G.adj(v))
            if (v == w) count++;
    return count / 2;
}
```

Set-of-edges graph representation

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

Adjacency-matrix graph representation

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

Adjacency-list graph representation

Maintain vertex-indexed array of lists.
Adjacency-list graph representation: Java implementation

```java
public class Graph {
    private final int V;
    private 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-lists representation.
- Algorithms based on iterating over vertices adjacent to \( v \).
- Real-world graphs tend to be sparse.

Graph representations

<table>
<thead>
<tr>
<th>representation</th>
<th>space</th>
<th>add edge</th>
<th>edge between ( v ) and ( w )</th>
<th>iterate over vertices adjacent to ( v )</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>( \text{degree}(v) )</td>
<td>( \text{degree}(v) )</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.
**Depth-first search**

**Goal.** Systematically search through a graph.

**Idea.** Mimic maze exploration.

**DFS (to visit a vertex v)**

1. Mark v as visited.
2. Recursively visit all unmarked vertices adjacent to v.

**Typical applications.**
- Find all vertices connected to a given source vertex.
- Find a path between two vertices.

**Design challenge.** How to implement?

**Depth-first search**

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

**Design pattern for graph processing**

**Design pattern.** Decouple graph data type from graph processing.
- Create a Graph object.
- Pass the Graph to a graph-processing routine, e.g., Paths.
- Query the graph-processing routine for information.

```java
public class Paths

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
            Paths paths = new Paths(G, s);
            for (int v = 0; v < G.V(); v++)
                if (paths.hasPathTo(v))
                    StdOut.println(v);
            return null;
        }
    }
}
```

**Input format for Graph constructor (two examples)**
- tinyG.txt
- mediumG.txt

**Depth-first search**

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

**Depth-first search**

- **Visit 0**
- **Visit 6**
- **Visit 6**
- **Visit 4**
**Depth-first search**

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

<table>
<thead>
<tr>
<th>$v$</th>
<th>marked[]</th>
<th>edgeTo[$v$]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>6</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>F</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>F</td>
<td>-</td>
</tr>
</tbody>
</table>

visit 5

<table>
<thead>
<tr>
<th>$v$</th>
<th>marked[]</th>
<th>edgeTo[$v$]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>F</td>
<td>-</td>
</tr>
</tbody>
</table>

visit 3

<table>
<thead>
<tr>
<th>$v$</th>
<th>marked[]</th>
<th>edgeTo[$v$]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>F</td>
<td>-</td>
</tr>
</tbody>
</table>

visit 3

3 done
To visit a vertex $v$:
- Mark vertex $v$ as visited.
- Recursively visit all unmarked vertices adjacent to $v$. 

### Depth-first search

#### Visit 5

- $v$ marked[] edgeTo[$v$]
  - 0: T, edgeTo[5]
  - 1: F, edgeTo[0]
  - 2: F, edgeTo[1]
  - 3: T, edgeTo[6]
  - 4: T, edgeTo[5]
  - 5: T, edgeTo[6]
  - 6: T, edgeTo[0]
  - 7: F, edgeTo[0]
  - 8: F, edgeTo[7]
  - 9: F, edgeTo[0]
  - 10: F, edgeTo[0]
  - 11: F, edgeTo[0]
  - 12: F, edgeTo[0]

#### Visit 4

- $v$ marked[] edgeTo[$v$]
  - 0: T, edgeTo[5]
  - 1: T, edgeTo[5]
  - 2: T, edgeTo[6]
  - 3: T, edgeTo[5]
  - 4: T, edgeTo[6]
  - 5: T, edgeTo[6]
  - 6: T, edgeTo[0]
  - 7: F, edgeTo[0]
  - 8: F, edgeTo[7]
  - 9: F, edgeTo[0]
  - 10: F, edgeTo[0]
  - 11: F, edgeTo[0]
  - 12: F, edgeTo[0]
To visit a vertex $v$:
- Mark vertex $v$ as visited.
- Recursively visit all unmarked vertices adjacent to $v$.

### Depth-first search

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

---

6 done

---

visit 4

---

visit 0

---

4 done

---

visit 4

---

6 done

---

visit 0

---
**Depth-first search**

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

```
visit 0
```

```
v  marked[]  edgeTo[v]
0  T       -
1  F       -
2  T       0
3  T       5
4  T       6
5  T       4
6  T       0
7  F       -
8  F       -
9  F       -
10  F      -
11  F      -
12  F      -
```

```
visit 1
```

```
v  marked[]  edgeTo[v]
0  T       -
1  T       0
2  T       0
3  T       5
4  T       6
5  T       4
6  T       0
7  F       -
8  F       -
9  F       -
10  F      -
11  F      -
12  F      -
```

```
visit 2
```

```
v  marked[]  edgeTo[v]
0  T       -
1  F       -
2  T       0
3  T       5
4  T       6
5  T       4
6  T       0
7  F       -
8  F       -
9  F       -
10  F      -
11  F      -
12  F      -
```

```
2 done
```

```
v  marked[]  edgeTo[v]
0  T       -
1  F       -
2  T       0
3  T       5
4  T       6
5  T       4
6  T       0
7  F       -
8  F       -
9  F       -
10  F      -
11  F      -
12  F      -
```

```
visit 0
```

```
v  marked[]  edgeTo[v]
0  T       -
1  F       -
2  T       0
3  T       5
4  T       6
5  T       4
6  T       0
7  F       -
8  F       -
9  F       -
10  F      -
11  F      -
12  F      -
```

```
visit 1
```

```
v  marked[]  edgeTo[v]
0  T       -
1  T       0
2  T       0
3  T       5
4  T       6
5  T       4
6  T       0
7  F       -
8  F       -
9  F       -
10  F      -
11  F      -
12  F      -
```

```
visit 2
```
To visit a vertex $v$:
• Mark vertex $v$ as visited.
• Recursively visit all unmarked vertices adjacent to $v$.

### Depth-first search

![Diagram showing depth-first search](image1)

- **Goal.** Find all vertices connected to $s$ (and a path).
- **Idea.** Mimic maze exploration.

- **Algorithm.**
  - Use recursion (ball of string).
  - Mark each visited vertex (and keep 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.

  $(\text{edgeTo}[w] == v)$ means that edge $v$-$w$ taken to visit $w$ for first time
**Depth-first search**

```java
public class DepthFirstPaths {
    private boolean[] marked;
    private int[] edgeTo;
    private int s;
    public DepthFirstPaths(Graph G, int s) {
        ...  // Initialize data structures
        dfs(G, s);  // find vertices connected to s
    }
    private void dfs(Graph G, int v) {
        marked[v] = true;  // recursive DFS does the work
        for (int w : G.adj(v))
            if (!marked[w])
                dfs(G, w);  // if w unmarked, then consider last edge on a path from s to w that goes from a marked vertex to an unmarked one
        edgeTo[v] = v;  // edgeTo[w] = previous vertex on path from s to v
    }  // Depth-first search properties
}
```

**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: preparing for a date**

http://xkcd.com/761/

```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 application: flood fill**

**Challenge.** Flood fill (Photoshop magic wand).

**Assumptions.** Picture has millions to billions of pixels.

**Solution.** Build a grid graph.
- **Vertex:** pixel.
- **Edge:** between two adjacent gray pixels.
- **Blob:** all pixels connected to given pixel.

---

**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.

**Graph API**
- **Depth-first search**
- **Breadth-first search**
- **Connected components**
- **Challenges**
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.

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

deploy \( 0 \)

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.

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

deploy \( 0 \)

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.

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

deploy \( 2 \)

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.

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

deploy \( 0 \)
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.

<table>
<thead>
<tr>
<th>queue</th>
<th>$v$</th>
<th>edgeTo[$v$]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

0 done

dequeue 2

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.

<table>
<thead>
<tr>
<th>queue</th>
<th>$v$</th>
<th>edgeTo[$v$]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

dequeue 2

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.

<table>
<thead>
<tr>
<th>queue</th>
<th>$v$</th>
<th>edgeTo[$v$]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

dequeue 2

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.

<table>
<thead>
<tr>
<th>queue</th>
<th>$v$</th>
<th>edgeTo[$v$]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

dequeue 2
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**

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**

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**

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**

Repeat until queue is empty:
• Remove vertex \( v \) from queue.
• Add to queue all unmarked vertices adjacent to \( v \) and mark them.
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

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

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

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

Repeat until queue is empty:
• Remove vertex \( v \) from queue.
• Add to queue all unmarked vertices adjacent to \( v \) and mark them.

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

 dequeue 5

5 done
**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.

```
<table>
<thead>
<tr>
<th>queue</th>
<th>v</th>
<th>edgeTo[v]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

```
0
1 0
2 0
3 2
4 2
5 0
```

*dequeue 3*

```
<table>
<thead>
<tr>
<th>queue</th>
<th>v</th>
<th>edgeTo[v]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

```
0
1 0
2 0
3 2
4 2
5 0
```

*dequeue 3*

```
<table>
<thead>
<tr>
<th>queue</th>
<th>v</th>
<th>edgeTo[v]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

```
0
1 0
2 0
3 2
4 2
5 0
```

*dequeue 3*

```
<table>
<thead>
<tr>
<th>queue</th>
<th>v</th>
<th>edgeTo[v]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

```
0
1 0
2 0
3 2
4 2
5 0
```

*3 done*
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.

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

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

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**

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

<table>
<thead>
<tr>
<th>$v$</th>
<th>edgeTo[$v$]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

**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 $k+1$.

**Pf. [running time]** Each vertex connected to $s$ is visited once.

**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.

**Intuition.** BFS examines vertices in increasing distance from $s$.

**Breadth-first search**

```java
public class BreadthFirstPaths {
    private boolean[] marked;
    private boolean[] edgeTo[];
    private final int s;
    ...
    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 application: routing

Fewest number of hops in a communication network.

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 $s = \text{Kevin Bacon}$.

Kevin Bacon numbers.

Breadth-first search application: Erdös numbers

hand-drawing of part of the Erdös graph by Ron Graham

Endless Games board game

SixDegrees iPhone App
**Undirected Graphs**

- 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: 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
```

**Depth-first search.** [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.

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

---

**Connected components**

```
\begin{array}{ll}
\hline
v & id[v] \\
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 \\
\hline
\end{array}
```

3 connected components

---

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

- 63 connected components

---

**Connected components**

```
\begin{array}{ll}
\hline
v & id[v] \\
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 \\
\hline
\end{array}
```

63 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.

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

---

**Graph $G$**
To visit a vertex $v$:
- Mark vertex $v$ as visited.
- Recursively visit all unmarked vertices adjacent to $v$. 

<table>
<thead>
<tr>
<th>v</th>
<th>marked[]</th>
<th>cc[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>F</td>
<td>-</td>
</tr>
</tbody>
</table>

---

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

<table>
<thead>
<tr>
<th>v</th>
<th>marked[]</th>
<th>cc[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>F</td>
<td>-</td>
</tr>
</tbody>
</table>

---

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

<table>
<thead>
<tr>
<th>v</th>
<th>marked[]</th>
<th>cc[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>F</td>
<td>-</td>
</tr>
</tbody>
</table>

---

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

<table>
<thead>
<tr>
<th>v</th>
<th>marked[]</th>
<th>cc[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>F</td>
<td>-</td>
</tr>
</tbody>
</table>
To visit a vertex $v$:
- Mark vertex $v$ as visited.
- Recursively visit all unmarked vertices adjacent to $v$.

**Connected components**

![Diagram showing a graph with vertices marked and unmarked, along with a table tracking marked vertices and connected components.](image)

**Table:**

<table>
<thead>
<tr>
<th>$v$</th>
<th>marked[]</th>
<th>cc[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>-</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>F</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>F</td>
<td>-</td>
</tr>
</tbody>
</table>

Visit 3

**Marked vertices: $v$**

**Connected components**

![Diagram showing a graph with vertices marked and unmarked, along with a table tracking marked vertices and connected components.](image)

**Table:**

<table>
<thead>
<tr>
<th>$v$</th>
<th>marked[]</th>
<th>cc[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>-</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>F</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>F</td>
<td>-</td>
</tr>
</tbody>
</table>

Visit 5

**Marked vertices: $v$**

**Connected components**

![Diagram showing a graph with vertices marked and unmarked, along with a table tracking marked vertices and connected components.](image)

**Table:**

<table>
<thead>
<tr>
<th>$v$</th>
<th>marked[]</th>
<th>cc[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>-</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>F</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>F</td>
<td>-</td>
</tr>
</tbody>
</table>

Visit 5

**Marked vertices: $v$**

**Connected components**

![Diagram showing a graph with vertices marked and unmarked, along with a table tracking marked vertices and connected components.](image)

**Table:**

<table>
<thead>
<tr>
<th>$v$</th>
<th>marked[]</th>
<th>cc[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>-</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>F</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>F</td>
<td>-</td>
</tr>
</tbody>
</table>

Visit 5

**Marked vertices: $v$**
To visit a vertex $v$:
- Mark vertex $v$ as visited.
- Recursively visit all unmarked vertices adjacent to $v$.

Connected components

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

Connected components

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

Connected components

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

Connected components
To visit a vertex \( v \):

- Mark vertex \( v \) as visited.
- Recursively visit all unmarked vertices adjacent to \( v \).

<table>
<thead>
<tr>
<th>( v )</th>
<th>marked</th>
<th>cc[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>-</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>F</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>F</td>
<td>-</td>
</tr>
</tbody>
</table>

Connected components

To visit a vertex \( v \):

- Mark vertex \( v \) as visited.
- Recursively visit all unmarked vertices adjacent to \( v \).

<table>
<thead>
<tr>
<th>( v )</th>
<th>marked</th>
<th>cc[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>-</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>
</tbody>
</table>

Connected components

To visit a vertex \( v \):

- Mark vertex \( v \) as visited.
- Recursively visit all unmarked vertices adjacent to \( v \).

<table>
<thead>
<tr>
<th>( v )</th>
<th>marked</th>
<th>cc[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>0</td>
</tr>
</tbody>
</table>

Connected components

To visit a vertex \( v \):

- Mark vertex \( v \) as visited.
- Recursively visit all unmarked vertices adjacent to \( v \).

<table>
<thead>
<tr>
<th>( v )</th>
<th>marked</th>
<th>cc[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>0</td>
</tr>
</tbody>
</table>
To visit a vertex \( v \):
- Mark vertex \( v \) as visited.
- Recursively visit all unmarked vertices adjacent to \( v \).
To visit a vertex $v$:
• Mark vertex $v$ as visited.
• Recursively visit all unmarked vertices adjacent to $v$. 

**Connected components**

To visit a vertex $v$:
• Mark vertex $v$ as visited.
• Recursively visit all unmarked vertices adjacent to $v$. 

**Connected components**
To visit a vertex $v$:
- Mark vertex $v$ as visited.
- Recursively visit all unmarked vertices adjacent to $v$. 

**Connected components**

- 8 done
- connected component: 7 8

**Marked vertices**

<table>
<thead>
<tr>
<th>v</th>
<th>marked[]</th>
<th>cc[]</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>F</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>F</td>
<td>-</td>
</tr>
</tbody>
</table>

**Connected components**

- 7 done
- check 8
To visit a vertex $v$:
• Mark vertex $v$ as visited.
• Recursively visit all unmarked vertices adjacent to $v$.  

---

Connected components

To visit a vertex $v$:
• Mark vertex $v$ as visited.
• Recursively visit all unmarked vertices adjacent to $v$.  

---

Connected components

To visit a vertex $v$:
• Mark vertex $v$ as visited.
• Recursively visit all unmarked vertices adjacent to $v$.  

---

Connected components

To visit a vertex $v$:
• Mark vertex $v$ as visited.
• Recursively visit all unmarked vertices adjacent to $v$.  

---

Connected components

To visit a vertex $v$:
• Mark vertex $v$ as visited.
• Recursively visit all unmarked vertices adjacent to $v$.  

---
To visit a vertex \( v \):
- Mark vertex \( v \) as visited.
- Recursively visit all unmarked vertices adjacent to \( v \).
To visit a vertex \( v \):
- Mark vertex \( v \) as visited.
- Recursively visit all unmarked vertices adjacent to \( v \).

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

visit 10

Connected components

To visit a vertex \( v \):
- Mark vertex \( v \) as visited.
- Recursively visit all unmarked vertices adjacent to \( v \).

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

10 done

Connected components

To visit a vertex \( v \):
- Mark vertex \( v \) as visited.
- Recursively visit all unmarked vertices adjacent to \( v \).

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

connected component: 9 10 11 12
To visit a vertex $v$:
- 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];  }

  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);
  }
}
```

- 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)

- number of components
- id of component containing $v$
- all vertices discovered in same call of dfs have same id
Graph-processing challenge 1

Problem. Is a graph bipartite?

How difficult?
- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

High-school dating graph

Problem. Is a 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.

---

**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 3**

**Problem.** Find a cycle that uses every edge.

**Assumption.** Need to use each edge exactly once.

**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 3**

Problem. Find a cycle that uses every edge.

Assumption. Need to use each edge exactly once.

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 4**

Problem. Find a cycle that visits every vertex exactly once.

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 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-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 isomorphism is a 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 practitioners).