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
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.

![Two drawings of the same graph](image1)

**Caveat.** Intuition can be misleading.

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**

```java
public class Graph
{
    public Graph(int V)
    {
        create an empty graph with V vertices
    }

    public 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)
    { vertices adjacent to v }

    int V()
    { number of vertices }

    int E()
    { number of edges }

    String toString()
    { string representation }
}
```

**Graph API: sample client**

```java
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);
```

**Graph input format.**

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

In `java Test tinyG.txt`

```
0-6
0-2
0-1
0-5
1-0
2-0
3-5
3-4
```

### Graph API: sample client

```java
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);
```

**Graph input format.**

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

In `java Test tinyG.txt`

```
0-6
0-2
0-1
0-5
1-0
2-0
3-5
3-4
```

**Graph input format.**

```
<table>
<thead>
<tr>
<th>tinyG.txt</th>
<th>mediumG.txt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 5</td>
<td>0 1</td>
</tr>
<tr>
<td>4 3</td>
<td>0 2</td>
</tr>
<tr>
<td>0 1</td>
<td>1 0</td>
</tr>
<tr>
<td>9 12</td>
<td>6 4</td>
</tr>
<tr>
<td>5 4</td>
<td>3 5</td>
</tr>
<tr>
<td>0 2</td>
<td>11 12</td>
</tr>
<tr>
<td>9 10</td>
<td>0 6</td>
</tr>
<tr>
<td>9 11</td>
<td>7 8</td>
</tr>
<tr>
<td>5 3</td>
<td>9 12</td>
</tr>
</tbody>
</table>
```
### Typical graph-processing code

- **compute the degree of** \( v \)
  ```java
  public static int degree(Graph G, int v) {
      int degree = 0;
      for (int w : G.adj(v)) degree++;
      return degree;
  }
  ```

- **compute maximum degree**
  ```java
  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;
  }
  ```

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

- **count self-loops**
  ```java
  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; // each edge counted twice
  }
  ```

### Adjacency-matrix graph representation

Maintain a two-dimensional \( V \)-by-\( V \) boolean array; for each edge \( v \rightarrow w \) 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];
    }
}
```

In practice. Use adjacency-lists representation.
- Algorithms based on iterating over vertices adjacent to v.
- Real-world graphs tend to be sparse.

Huge number of vertices, small average vertex degree.

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 x V</td>
<td>1</td>
<td>1</td>
<td>V</td>
</tr>
<tr>
<td>adjacency lists</td>
<td>E + V</td>
<td>1</td>
<td>degree(v)</td>
<td>degree(v)</td>
</tr>
</tbody>
</table>

* disallows parallel edges

Undirected Graphs

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


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

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);

Design pattern for graph processing

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

print all vertices connected to s

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

V E V E
250 1273 244 246
239 240 238 245
235 238 233 240
232 248 231 248
229 249 228 241
... (1261 additional lines)

mediumG.txt

V E
250 1273

Depth-first search

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 $v$.

**Depth-first search**

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

**Visit 0**

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

**Visit 6**

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

**Visit 6**

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

**Visit 4**

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

### Depth-first search

**visit 5**

### Depth-first search

**visit 3**

**visit 3**

**visit 3**

3 done
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 \( v \).
To visit a vertex $v$:
- Mark vertex $v$ as visited.
- Recursively visit all unmarked vertices adjacent to $v$.
Depth-first search

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

    public DepthFirstPaths(Graph G, int s) {
        ...  
        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);
        edgeTo[w] = 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: preparing for a date

http://xkcd.com/761/
**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.

---

**UNDIRECTED GRAPHS**

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

Queue

\( v \)  edgeTo[\( v \)]

\( 0 \)  –

\( 1 \)  0

\( 2 \)  0

\( 3 \)  2

\( 4 \)  –

\( 5 \)  0

 dequeue 2

Breadth-first search

Queue

\( v \)  edgeTo[\( v \)]

\( 0 \)  –

\( 1 \)  0

\( 2 \)  0

\( 3 \)  2

\( 4 \)  2

\( 5 \)  0

 dequeue 2

2 done

Breadth-first search

Queue

\( v \)  edgeTo[\( v \)]

\( 0 \)  –

\( 1 \)  0

\( 2 \)  0

\( 3 \)  2

\( 4 \)  2

\( 5 \)  0

 dequeue 1

Breadth-first search

Queue

\( v \)  edgeTo[\( v \)]

\( 0 \)  –

\( 1 \)  0

\( 2 \)  0

\( 3 \)  2

\( 4 \)  2

\( 5 \)  0

 dequeue 1
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**

5 done

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**

dequeue 3

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**

dequeue 5

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**
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.
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>0</td>
<td>0</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>0</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

4 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.

4 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.

**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.

**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.

**ARPANET, July 1977**

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 $s = \text{Kevin Bacon}$.

---

Breadth-first search application: Kevin Bacon numbers

Kevin Bacon numbers.

![Kevin Bacon numbers](http://oracleofbacon.org)

Endless Games board game

**The Oracle of Bacon**

Endless Games board game

SixDegrees iPhone App

http://oracleofbacon.org

---

Breadth-first search application: Erdös numbers

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

```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
}
```

**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.
**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 $v$. 

**Input format for Graph constructor (two examples)**

```
250 1273
```

```
244 246
239 240
238 245
235 238
233 240
232 248
231 248
229 249
228 241
226 231
... 
```

(1261 additional lines)

```
250 1273
```

```
244 246
239 240
238 245
235 238
233 240
232 248
231 248
229 249
228 241
226 231
... 
```

(1261 additional lines)
To visit a vertex $v$:
- Mark vertex $v$ as visited.
- Recursively visit all unmarked vertices adjacent to $v$.  

**Connected components**

<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>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 0</td>
<td></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 6*

**Connected components**

<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>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 0</td>
<td></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 4*

**Connected components**

<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>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 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>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*
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 $v$.

5 done

Connected components

To visit a vertex $v$:

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

visit 4

Connected components

To visit a vertex $v$:

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

visit 4

Connected components

To visit a vertex $v$:

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

4 done

Connected components

To visit a vertex $v$:

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

visit 4

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 $v$. 

**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 $v$. 

**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**

Visit 0

Visit 1

Visit 2

Visit 3
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</td>
<td>1</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>

8 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</td>
<td>1</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>

7 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</td>
<td>1</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>

check 8 

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>1</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>

check 8 

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

**Connected components**

<table>
<thead>
<tr>
<th>vertex</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>1</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>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 9

**Connected components**

<table>
<thead>
<tr>
<th>vertex</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>T</td>
<td>2</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 10

**Connected components**

<table>
<thead>
<tr>
<th>vertex</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>T</td>
<td>2</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 11

**Connected components**

<table>
<thead>
<tr>
<th>vertex</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>T</td>
<td>2</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 12
To visit a vertex $v$:
- Mark vertex $v$ as visited.
- Recursively visit all unmarked vertices adjacent to $v$. 

Connected components

visit 12

visit 9

visit 12 done

visit 11 done
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 $v$.
**Undirected Graphs**

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

**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 an open problem](link)

**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)