4.1 **Undirected Graphs**

- introduction
- graph API
- depth-first search
- breadth-first search
- challenges
4.1 Undirected Graphs

- introduction
- graph API
- depth-first search
- breadth-first search
- challenges
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 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.
The evolution of FCC lobbying coalitions

“The Evolution of FCC Lobbying Coalitions” by Pierre de Vries in JoSS Visualization Symposium 2010
Map of science clickstreams

http://www.plosone.org/article/info:doi/10.1371/journal.pone.0004803
10 million Facebook friends

"Visualizing Friendships" by Paul Butler
The Internet as mapped by the Opte Project

http://en.wikipedia.org/wiki/Internet
# 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>intersection</td>
<td>street</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</td>
<td>friendship</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>molecule</td>
<td>atom</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

<table>
<thead>
<tr>
<th>problem</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s–t path</td>
<td>Is there a path between s and t?</td>
</tr>
<tr>
<td>shortest s–t path</td>
<td>What is the shortest path between s and t?</td>
</tr>
<tr>
<td>cycle</td>
<td>Is there a cycle in the graph?</td>
</tr>
<tr>
<td>Euler cycle</td>
<td>Is there a cycle that uses each edge exactly once?</td>
</tr>
<tr>
<td>Hamilton cycle</td>
<td>Is there a cycle that uses each vertex exactly once?</td>
</tr>
<tr>
<td>connectivity</td>
<td>Is there a path between every pair of vertices?</td>
</tr>
<tr>
<td>biconnectivity</td>
<td>Is there a vertex whose removal disconnects the graph?</td>
</tr>
<tr>
<td>planarity</td>
<td>Can the graph be drawn in the plane with no crossing edges?</td>
</tr>
<tr>
<td>graph isomorphism</td>
<td>Are two graphs isomorphic?</td>
</tr>
</tbody>
</table>

**Challenge.** Which graph problems are easy? difficult? intractable?
4.1 **Undirected Graphs**

- introduction
- graph API
- depth-first search
- breadth-first search
- challenges
Graph representation

**Vertex representation.**

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

**Anomalies.**

- Self-loop
- Parallel edges
## Graph API

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

    // Constructor to create a graph from input stream
    Graph(In in) {
    }

    // Method to add an edge v-w
    void addEdge(int v, int w) {
    }

    // Method to get the vertices adjacent to v
    Iterable<Integer> adj(int v) {
    }

    // Method to get the number of vertices
    int V() {
    }

    // Method to get the number of edges
    int E() {
    }

    // Method to calculate the degree of vertex v in graph G
    public static int degree(Graph G, int v) {
        int degree = 0;
        for (int w : G.adj(v)) {
            degree++;
        }
        return degree;
    }
}
```
Graph representation: adjacency matrix

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

![Graph representation with adjacency matrix](image-url)
Graph representation: adjacency lists

Maintain vertex-indexed array of lists.
Graph representations

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

Two graphs (\( V = 50 \))

- **Sparse** (\( E = 200 \))
- **Dense** (\( E = 1000 \))

huge number of vertices, small average vertex degree
# Graph representations

**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

| representation       | space  | add edge | edge between \( v \) and \( w \)? | iterate over vertices adjacent to \( v \)?
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></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 ( \dagger )</td>
<td>1</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>

\( \dagger \) disallows parallel edges
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>(v);
    }

    public void addEdge(int v, int w) {
        adj[v].add(w);
        adj[w].add(v);
    }

    public Iterable<Integer> adj(int v) {
        return adj[v];
    }
}
```

- **adjacency lists** (using Bag data type)
- create empty graph with V vertices
- add edge v–w (parallel edges and self-loops allowed)
- iterator for vertices adjacent to v
4.1 Undirected Graphs

- introduction
- graph API
- depth-first search
- breadth-first search
- challenges
Maze exploration

Maze graph.

- Vertex = intersection.
- Edge = passage.

Goal. Explore every intersection in the maze.
Trémaux maze exploration

Algorithm.

- Unroll a ball of string behind you.
- Mark each newly discovered intersection and passage.
- Retrace steps when no unmarked options.
Trémaux maze exploration

Algorithm.

- Unroll a ball of string behind you.
- Mark each newly discovered intersection and passage.
- Retrace steps when no unmarked options.

First use? Theseus entered Labyrinth to kill the monstrous Minotaur; Ariadne instructed Theseus to use a ball of string to find his way back out.
Maze exploration: easy
Maze exploration: medium
Maze exploration: challenge for the bored
Depth-first search

**Goal.** Systematically traverse a graph.

**Idea.** Mimic maze exploration.  

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 demo

To visit a vertex $v$:

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

**Graph G**
Depth-first search demo

To visit a vertex \( v \):

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

---

**graph G**
Depth-first search demo

To visit a vertex \( v \):

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

<table>
<thead>
<tr>
<th>( v )</th>
<th>marked[]</th>
<th>edgeTo[]</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>F</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>F</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 0: check 6, check 5, check 2, check 1, done
Depth-first search demo

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

Visit 6: check 0, check 4, done
Depth-first search demo

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

<table>
<thead>
<tr>
<th>( v )</th>
<th>marked[]</th>
<th>edgeTo[]</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>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>

visit 6: check 0, check 4, done
Depth-first search demo

To visit a vertex \( v \):

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

**visit 4: check 5, check 6, check 3, done**

**Table:**

<table>
<thead>
<tr>
<th>( v )</th>
<th>marked[]</th>
<th>edgeTo[]</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>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>
Depth-first search demo

To visit a vertex \( v \):

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

\[
\begin{array}{c|c|c}
\text{v} & \text{marked[]} & \text{edgeTo[]} \\
\hline
0 & T & - \\
1 & F & - \\
2 & F & - \\
3 & F & - \\
4 & T & 6 \\
5 & T & 4 \\
6 & T & 0 \\
7 & F & - \\
8 & F & - \\
9 & F & - \\
10 & F & - \\
11 & F & - \\
12 & F & - \\
\end{array}
\]

**visit 5:** check 3, check 4, check 0, done
Depth-first search demo

To visit a vertex $v$:

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

Visit 3: check 5, check 4, done
Depth-first search demo

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

Visit 3: check 5, check 4, done
Depth-first search demo

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

visit 3: check 5, check 4, done
Depth-first search demo

To visit a vertex $v$:

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

**visit 5:** check 3, check 4, check 0, done
Depth-first search demo

To visit a vertex $v$:

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

Visit 5: check 3, check 4, check 0, done
Depth-first search demo

To visit a vertex $v$:

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

Visit 5: check 3, check 4, check 0, done
Depth-first search demo

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

visit 4: check 5, check 6, check 3, done
Depth-first search demo

To visit a vertex $v$:

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

**visit 4:** check 5, check 6, **check 3**, done
Depth-first search demo

To visit a vertex $v$:

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

<table>
<thead>
<tr>
<th>v</th>
<th>marked[]</th>
<th>edgeTo[]</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 4: check 5, check 6, check 3, done
Depth-first search demo

To visit a vertex $v$:

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

**Visit 6:** check 0, check 4, **done**
Depth-first search demo

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

**visit 0:** check 6, check 5, check 2, check 1, done
To visit a vertex $v$:

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

**visit 0:** check 6, check 5, **check 2**, check 1, done
Depth-first search demo

To visit a vertex \( v \):

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

<table>
<thead>
<tr>
<th>( v )</th>
<th>marked[]</th>
<th>edgeTo[]</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>T</td>
<td>0</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 2: check 0, done
To visit a vertex $v$:
- Mark vertex $v$.
- Recursively visit all unmarked vertices adjacent to $v$. 

visit 2: check 0, done
Depth-first search demo

To visit a vertex \( v \):

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

Visit 0: check 6, check 5, check 2, check 1, done
Depth-first search demo

To visit a vertex $v$:

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

visit 1: check 0, done
Depth-first search demo

To visit a vertex $v$:

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

**Visit 1:** check 0, done
To visit a vertex $v$:

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

**Depth-first search demo**

Visit 0: check 6, check 5, check 2, check 1, done
Depth-first search demo

To visit a vertex $v$:

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

### Vertices reachable from 0

<table>
<thead>
<tr>
<th>$v$</th>
<th>marked</th>
<th>edgeTo</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>–</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>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>
Depth-first search demo

To visit a vertex \( v \):

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

vertices reachable from 0
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.
- Query the graph-processing routine for information.

### public class Paths

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Paths(Graph G, int s)</code></td>
<td>find paths in <code>G</code> from source <code>s</code></td>
</tr>
<tr>
<td><code>boolean hasPathTo(int v)</code></td>
<td>is there a path from <code>s</code> to <code>v</code>?</td>
</tr>
<tr>
<td><code>Iterable&lt;Integer&gt; pathTo(int v)</code></td>
<td>path from <code>s</code> to <code>v</code>; null if no such path</td>
</tr>
</tbody>
</table>

```java
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`
Depth-first search: data structures

To visit a vertex $v$:

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

Data structures.

- Boolean array `marked[]` to mark vertices.
- Integer array `edgeTo[]` to keep track of paths.
  
  $(\text{edgeTo}[w] == v)$ means that edge $v$-$w$ taken to discover vertex $w$
- Function-call stack for recursion.
**Depth-first search: Java implementation**

```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);
    }

    private void dfs(Graph G, int v) {
        // recursive DFS does the work
        marked[v] = true;
        for (int w : G.adj(v)) {
            if (!marked[w]) {
                edgeTo[w] = v;
                dfs(G, w);
            }
        }
    }
}
```

- `marked[v] = true` if v connected to s
- `edgeTo[v] = previous vertex on path from s to v`
- Initialize data structures
- Find vertices connected to s
- Recursive DFS does the work
Depth-first search: properties

**Proposition.** DFS marks all vertices connected to $s$ in time proportional to the sum of their degrees (plus time to initialize the marked[] array).

**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).

**Pf.** [running time]
Each vertex connected to $s$ is visited once.
**Depth-first search: properties**

**Proposition.** After DFS, a client can check if vertex \( v \) is connected to \( s \) in constant time, and the client can find \( v-s \) path (if one exists) in time proportional to its length.

**Pf.** `edgeTo[]` is parent-link representation of a tree rooted at vertex \( s \).

```java
public boolean hasPathTo(int v)
{
    return marked[v];
}

public Iterable<Integer> pathTo(int v)
{
    if (!hasPathTo(v)) return null;
    Stack<Integer> path = new Stack<Integer>();
    for (int x = v; x != s; x = edgeTo[x])
    {
        path.push(x);
        path.push(s);
    }
    return path;
}
```

Trace of `pathTo()` computation:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Depth-first search application: preparing for a date

[Image of a comic strip from xkcd.com/761/]

PREPARING FOR A DATE:
WHAT SITUATIONS MIGHT I PREPARE FOR?
1) MEDICAL EMERGENCY
2) DANCING
3) FOOD TOO EXPENSIVE

OKAY, WHAT KINDS OF EMERGENCIES CAN HAPPEN?
1) SNAKEBITE
2) LIGHTNING STRIKE
3) FALL FROM CHAIR

HMM, WHICH SNAKES ARE DANGEROUS? LET'S SEE...
1) A) CORN SNAKE
2) GARTER SNAKE
3) COPPERHEAD

THE RESEARCH COMPARING SNAKE VENOMS IS SCATTERED AND INCONSISTENT. I'LL MAKE A SPREADSHEET TO ORGANIZE IT.

I'M HERE TO PICK YOU UP! YOU'RE NOT DRESSED?

BY LD50, THE INLAND TAIPAN HAS THE DEADLIEST VENOM OF ANY SNAKE!

I REALLY NEED TO STOP USING DEPTH-FIRST SEARCHES.
4.1 **Undirected Graphs**

- introduction
- graph API
- depth-first search
- breadth-first search
- challenges
Breadth-first search demo

Repeat until queue is empty:

- Remove vertex $v$ from queue.
- Add to queue all unmarked vertices adjacent to $v$ and mark them.

Graph $G$
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 demo

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

```
dequeue 0
```
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 demo**

dequeue 0
Breadth-first search demo

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

```
dehqueue 0
```
Breadth-first search demo

Repeat until queue is empty:

- Remove vertex $v$ from queue.
- Add to queue all unmarked vertices adjacent to $v$ and mark them.

dequeue 0
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 demo**

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

0 done
Breadth-first search demo

Repeat until queue is empty:

- Remove vertex $v$ from queue.
- Add to queue all unmarked vertices adjacent to $v$ and mark them.

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 demo**

dequeue 2
Breadth-first search demo

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[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>–</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>4</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Breadth-first search demo

Repeat until queue is empty:

• Remove vertex \( v \) from queue.
• Add to queue all unmarked vertices adjacent to \( v \) and mark them.

dequeue 2
Breadth-first search demo

Repeat until queue is empty:

- Remove vertex $v$ from queue.
- Add to queue all unmarked vertices adjacent to $v$ and mark them.

**deque 2**
Breadth-first search demo

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

2 done
Breadth-first search demo

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

deque 1
Breadth-first search demo

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

 dequeue 1
Breadth-first search demo

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[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

dqueue 1
Breadth-first search demo

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[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>–</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

1 done
Breadth-first search demo

Repeat until queue is empty:

- Remove vertex $v$ from queue.
- Add to queue all unmarked vertices adjacent to $v$ and mark them.

deque 5
Breadth-first search demo

Repeat until queue is empty:

- Remove vertex $v$ from queue.
- Add to queue all unmarked vertices adjacent to $v$ and mark them.

dequeue 5
Breadth-first search demo

Repeat until queue is empty:

- Remove vertex \( v \) from queue.
- Add to queue all unmarked vertices adjacent to \( v \) and mark them.

dequeue 5

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

`edgeTo[]` and `distTo[]` are used to track the edges and distances in the graph.
Breadth-first search demo

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[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>–</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

5 done
Breadth-first search demo

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

deque 3
Breadth-first search demo

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

dqueue 3
Breadth-first search demo

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[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>–</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**dequeue 3**
Breadth-first search demo

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 demo

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[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>–</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Breadth-first search demo

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

dequeue 4
Breadth-first search demo

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 demo

dequeue 4
Breadth-first search demo

Repeat until queue is empty:

- Remove vertex $v$ from queue.
- Add to queue all unmarked vertices adjacent to $v$ and mark them.

 dequeue 4
Breadth-first search demo

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[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>–</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Breadth-first search demo

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[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</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>1</td>
</tr>
</tbody>
</table>

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.

\[
\text{BFS (from source vertex } s) \]

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 unmarked neighbors to the queue, and mark them.
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 \( s \).

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

Endless Games board game

SixDegrees iPhone App
Kevin Bacon graph

- 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 $s = \text{Kevin Bacon}$. 
Breadth-first search application: Erdös numbers

hand-drawing of part of the Erdös graph by Ron Graham
4.1 **Undirected Graphs**

- introduction
- graph API
- depth-first search
- breadth-first search
- challenges
Graph traversal summary

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

<table>
<thead>
<tr>
<th>graph problem</th>
<th>BFS</th>
<th>DFS</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>s–t path</td>
<td>✔</td>
<td>✔</td>
<td>$E + V$</td>
</tr>
<tr>
<td>shortest s–t path</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>$2^{1.657V}$</td>
</tr>
<tr>
<td>bipartiteness (odd cycle)</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>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>