4.1 Undirected Graphs

- introduction
- graph API
- depth-first search
- breadth-first search
- applications of DFS and BFS
4.1 Undirected Graphs

- introduction
- graph API
- depth-first search
- breadth-first search
- applications of DFS and BFS
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.
Rail network

Vertex = station; edge = route
Social networks

Vertex = person; edge = social relationship.

“Visualizing Friendships” by Paul Butler
Protein-protein interaction network

Vertex = protein; edge = interaction.

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

Vertex = IP address.
Edge = connection.

http://en.wikipedia.org/wiki/Internet
Romantic and sexual relationships in a high school

Relationship graph at "Jefferson High"

## 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**

**Graph:** set of *vertices* connected pairwise by *edges*.

**Path:** sequence of vertices connected by edges, with no repeated edges.

Two vertices are **connected** if there is a path between them.

**Cycle:** Path (with at least 1 edge) whose first and last vertices are the same.
## 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
- applications of DFS and BFS
Graph representation

Graph drawing. Provides intuition about the structure of the graph.

different drawings of the same graph

Caveat. Intuition can be misleading.
Graph representation

Vertex representation.

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

Anomalies.

self-loop
parallel edges
public class Graph

Graph(int V)                       // create an empty graph with V vertices
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

All graph processing can be done using above API. Example:

// degree of vertex v in graph G
public static int degree(Graph G, int v) {
    int count = 0;
    for (int w : G.adj(v))
        count++;
    return count;
}

Arvind’s view:
This API is oversimplified. Any competent graph API must provide degree() and other methods.
Graph representation: adjacency matrix

Maintain a $V$-by-$V$ boolean array; for each edge $v$–$w$ in graph:

$\text{adj}[v][w] = \text{adj}[w][v] = \text{true}$.
Graph representation: adjacency matrix

Maintain a $V$-by-$V$ boolean array; for each edge $v$–$w$ in graph:

$$\text{adj}[v][w] = \text{adj}[w][v] = \text{true}.$$
Which is the order of growth of running time of the following code fragment if the graph uses the **adjacency-matrix** representation, where $V$ is the number of vertices and $E$ is the number of edges?

for (int v = 0; v < G.V(); v++)
    for (int w : G.adj(v))
        StdOut.println(v + "-" + w);

**print each edge twice**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**adjacency-matrix representation**

**A.** $V$

**B.** $E + V$

**C.** $V^2$

**D.** $VE$
Graph representation: adjacency lists

Maintain vertex-indexed array of lists.
Which is the order of growth of running time of the following code fragment if the graph uses the **adjacency-lists** representation, where $V$ is the number of vertices and $E$ is the number of edges?

A. $V$
B. $E + V$
C. $V^2$
D. $VE$

```
for (int v = 0; v < G.V(); v++)
    for (int w : G.adj(v))
        StdOut.println(v + "-" + w);
```

print each edge twice
Graph representations

**In practice.** Use adjacency-lists representation.
- Algorithms based on iterating over vertices adjacent to $v$.
- Real-world graphs tend to be **sparse** (not **dense**).
Graph representations

In practice. Use adjacency-lists representation.

- Algorithms based on iterating over vertices adjacent to v.
- Real-world graphs tend to be sparse (not dense).

<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>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>

† 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>();
    }

    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**

https://algs4.cs.princeton.edu/41undirected/Graph.java.html
4.1 UNDIRECTED GRAPHS

- introduction
- graph API
- depth-first search
- breadth-first search
- applications of DFS and BFS
Warmup: maze exploration

Maze graph.

- Vertex = intersection.
- Edge = passage.

**Goal.** Explore every intersection in the maze.
Maze exploration algorithm in Greek myth

How Theseus escaped from the labyrinth after killing the Minotaur:

- Unroll a ball of string behind you.
- Mark each newly discovered intersection.
- Retrace steps when no unmarked options.
**Depth-first search**

**Goal.** Systematically traverse a graph.

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

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

**Typical applications.**

- Find all vertices connected to a given vertex.
- Find a path between two vertices.
Depth-first search demo

To visit a vertex \( v \):

- Mark vertex \( v \).
- Recursively visit all unmarked vertices adjacent to \( v \).
Depth-first search demo

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

vertices connected to 0
(and associated paths)
Run DFS using the following adjacency-lists representation of graph G, starting at vertex 0. In which order is dfs(G, v) called?

A. 0 1 2 4 5 3 6
B. 0 1 2 4 5 6 3
C. 0 1 4 2 5 3 6
D. 0 1 2 6 4 5 3
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])
                { 
                    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 $s$ (and no others).

**Proof.**

- If $w$ marked, then $w$ connected to $s$ (why?)
- If $w$ connected to $s$, then $w$ marked.

(if $w$ unmarked, then consider the last edge on a path from $s$ to $w$ that goes from a marked vertex to an unmarked one).

*Skipped in class*
Depth-first search: properties

Proposition. DFS marks all vertices connected to $s$ in time proportional to $V + E$ in the worst case.

Proof.

- Initialize two arrays of length $V$.
- Each vertex is visited at most once.
  (visiting a vertex takes time proportional to its degree)

\[
\text{degree}(v_0) + \text{degree}(v_1) + \text{degree}(v_2) + \ldots = 2E
\]
Depth-first search: properties

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

**Proof.** `edgeTo[]` is parent-link representation of a tree rooted at vertex $s$. 
4.1 UNDIRECTED GRAPHS

- introduction
- graph API
- depth-first search
- breadth-first search
- applications of DFS and BFS
**Graph search.** Many ways to explore a graph.

- *Preorder:* vertices in order of calls to `dfs(G, v)`.
- *Postorder:* vertices in order of returns from `dfs(G, v)`.
- *Level-order:* vertices in increasing order of distance from \( s \).

**Tree traversal.** Many ways to explore a binary tree.

- *Inorder:* \( A \ C \ E \ H \ M \ R \ S \ X \)
- *Preorder:* \( S \ E \ A \ C \ R \ H \ M \ X \)
- *Postorder:* \( C \ A \ M \ H \ R \ E \ X \ S \)
- *Level-order:* \( S \ E \ X \ A \ R \ C \ H \ M \)
Breadth-First Search (BFS)

**BFS (from source vertex s)**

Put s on a queue, and mark s as visited.
Repeat until the queue is empty:
- dequeue vertex v
- enqueue each of v’s unmarked neighbors, and mark them.

**Intuition.** BFS traverses vertices in order of distance from s.

![Graph G](image)

**graph G**

![Dijkstra](image)

**dist = 0**  **dist = 1**  **dist = 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.
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.
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

<table>
<thead>
<tr>
<th>queue</th>
<th>$v$</th>
<th>edgeTo[]</th>
<th>marked[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>–</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>–</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>–</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>–</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>–</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>–</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>–</td>
<td>F</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.

**Breadth-first search demo**

<table>
<thead>
<tr>
<th>queue</th>
<th>$v$</th>
<th>edgeTo[]</th>
<th>marked[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>−</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>−</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>−</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>−</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>−</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>−</td>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>

dequeue 0: check 2, check 1, check 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.

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

dqueue 0: check 2, check 1, check 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 demo**

**queue** | **v** | **edgeTo[]** | **marked[]**
--- | --- | --- | ---
0 | – | T | 0
1 | 0 | T | 1
2 | 0 | T | 2
3 | – | F | 3
4 | – | F | 4
5 | 0 | T | 5
6 | – | F | 6

**dequeue 0: check 2, check 1, check 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.

<table>
<thead>
<tr>
<th>queue</th>
<th>$v$</th>
<th>edgeTo[]</th>
<th>marked[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>–</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>–</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>–</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>–</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>–</td>
<td>F</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.

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.

dequeue 2: check 0, check 1, check 3, check 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.

** dequeuе 2:** check 0, check 1, check 3, check 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 2:** check 0, check 1, check 3, check 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 2: check 0, check 1, check 3, check 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

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.

** dequeue 1: check 0, check 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.

dequeue 1: check 0, check 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>marked[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>–</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>–</td>
<td>F</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.

### Breadth-first search demo

![Graph](https://via.placeholder.com/150)

- **queue** | **v** | **edgeTo[]** | **marked[]**
- 0     |   –   |   T       |
- 1     |   0   |   T       |
- 2     |   0   |   T       |
- 3     |   2   |   T       |
- 4     |   2   |   T       |
- 5     |   0   |   T       |
- 6     |   –   |   F       |

**enqueue 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.

**Breadth-first search demo**

depqueue 5: check 3, check 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.

**Breadth-first search demo**

dqueue 5: check 3, check 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.

<table>
<thead>
<tr>
<th>queue</th>
<th>v</th>
<th>edgeTo[]</th>
<th>marked[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>–</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>–</td>
<td>F</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.

```plaintext
queue  v  edgeTo[]  marked[]
0      -    T
1      0    T
2      0    T
3      2    F
4      2    T
5      0    T
6      -    F
```

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.

**deque 3:** check 5, check 4, check 2, check 6
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 3:** check 5, check 4, check 2, check 6
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: check 5, check 4, check 2, check 6
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 3:** check 5, check 4, check 2, **check 6**
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.

3 done
Repeat until queue is empty:

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

```plaintext
|
+-----+-----+-----+-----+-----+-----+-----+
|0    | 1    | 2    | 3    | 4    | 5    | 6    |
+-----+-----+-----+-----+-----+-----+-----+
|
|
|
|
|
|
|
+-----+-----+-----+-----+-----+-----+-----+
|
|
|
|
|
|
|
|
+-----+-----+-----+-----+-----+-----+-----+
|
|
|
|
|
|
|
|
+-----+-----+-----+-----+-----+-----+-----+
|
|
|
|
|
|
|
|
+-----+-----+-----+-----+-----+-----+-----+
|
|
|
|
|
|
|
|
+-----+-----+-----+-----+-----+-----+-----+
|
|
|
|
|
|
|
|
+-----+-----+-----+-----+-----+-----+-----+
```
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 4: check 3, check 2
Repeat until queue is empty:
  • Remove vertex $v$ from queue.
  • Add to queue all unmarked vertices adjacent to $v$ and mark them.

dequeue 4: check 3, check 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>marked[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4 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 6**

<table>
<thead>
<tr>
<th>queue</th>
<th>v</th>
<th>edgeTo[]</th>
<th>marked[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>–</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>T</td>
<td></td>
</tr>
</tbody>
</table>

**6**
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 6: check 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 demo**

<table>
<thead>
<tr>
<th>queue</th>
<th>$v$</th>
<th>edgeTo[]</th>
<th>marked[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>–</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>T</td>
<td></td>
</tr>
</tbody>
</table>

6 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.

```
<table>
<thead>
<tr>
<th>v</th>
<th>edgeTo[]</th>
<th>marked[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>–</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>T</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>T</td>
</tr>
</tbody>
</table>
```

all 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 demo**

<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>
<tr>
<td>6</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
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] = ?; // Set marked[w]
                    edgeTo[w] = v; // Set edgeTo[w]
                    distTo[w] = distTo[v] + 1; // Update distTo[w]
                }
            }
        }
    }
}

https://algs4.cs.princeton.edu/41undirected/BreadthFirstPaths.java.html
```
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

https://algs4.cs.princeton.edu/41undirected/BreadthFirstPaths.java.html
Breadth-first search properties

BFS examines vertices in order of increasing distance (\# 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 \( E + V \).
4.1 Undirected Graphs

- Introduction
- Graph API
- Depth-first search
- Breadth-first search
- Applications of DFS and BFS
Breadth-first search application: routing

Fewest number of hops in a communication network.

ARPANET, July 1977
Breadth-first search application: Kevin Bacon numbers

http://oracleofbacon.org
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 $s = \text{Kevin Bacon}$. 
Exercise: applications of DFS and BFS

Recall: a **connected component** is a maximal set of connected vertices. Given a graph, partition vertices into connected components using DFS or BFS.

i.e. create an `id[]` array such that `id[u] == id[v]` iff `u` & `v` are in same CC.

**Euler cycle**: given a graph, find a general cycle that traverses each edge exactly once, or determine that none exists.

<table>
<thead>
<tr>
<th>Challenge</th>
</tr>
</thead>
</table>

May traverse a node more than once
**Connected components**

**Goal.** Partition vertices into 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.
Finding connected components with DFS

```java
public class CC {
    private boolean[] marked;
    private int[] id;
    private int count;

    public CC(Graph G) {
        marked = new boolean[G.V()];
        id = new int[G.V()];
        for (int v = 0; v < G.V(); v++) {
            if (!marked[v]) {
                dfs(G, v);
                count++;
            }
        }
    }

    public int count() {
        return count;
    }

    public int id(int v) {
        return id[v];
    }

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

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

- `id[v] = id of component containing v`
- `count` number of components
- Run DFS from one vertex in each component
- See next slide
Finding connected components with DFS (continued)

```java
public int count()
{
    return count;
}

public int id(int v)
{
    return id[v];
}

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

private void dfs(Graph G, int v)
{
    marked[v] = true;
    id[v] = count;
    for (int w : G.adj(v))
        if (!marked[w])
            dfs(G, w);
}
```

- **number of components**
- **id of component containing v**
- **v and w connected iff same id**
- **all vertices discovered in same call of dfs have same id**
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>$V$</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>$2^{c \ln^3 V}$</td>
</tr>
</tbody>
</table>