Undirected Graphs
Outline

1 What are Graphs?

2 Undirected Graphs

3 Depth-First Search (DFS)

4 Breadth-First Search (BFS)

5 Symbol Graphs
What are Graphs?

A graph is a set of vertices connected pairwise by edges. We use the names 0 through $V-1$ for the vertices in a $V$-vertex graph. We use the notation $v-w$ to refer to an edge that connects vertices $v$ and $w$. A self-loop is an edge that connects a vertex to itself. Parallel edges are edges that connect the same pair of vertices.
What are Graphs?

A graph is a set of \( V \) vertices connected pairwise by \( E \) edges.
**What are Graphs?**

A graph is a set of $V$ vertices connected pairwise by $E$ edges.

We use the names 0 through $V - 1$ for the vertices in a $V$-vertex graph.
What are Graphs?

A graph is a set of $V$ vertices connected pairwise by $E$ edges.

We use the names 0 through $V - 1$ for the vertices in a $V$-vertex graph.

We use the notation $v-w$ to refer to an edge that connects vertices $v$ and $w.$
**What are Graphs?**

A graph is a set of $V$ vertices connected pairwise by $E$ edges.

We use the names 0 through $V - 1$ for the vertices in a $V$-vertex graph.

We use the notation $v-w$ to refer to an edge that connects vertices $v$ and $w$.

A self-loop is an edge that connects a vertex to itself.
What are Graphs?

A graph is a set of $V$ vertices connected pairwise by $E$ edges

We use the names 0 through $V - 1$ for the vertices in a $V$-vertex graph.

We use the notation $v \rightarrow w$ to refer to an edge that connects vertices $v$ and $w$.

A self-loop is an edge that connects a vertex to itself.

Parallel edges are edges that connect the same pair of vertices.
What are Graphs?

- The degree of a vertex is the number of vertices connected to it.
- A path is a sequence of vertices connected by edges.
- A cycle is a path with at least one edge whose first and last vertices are the same.
- The length of a path or a cycle is its number of edges.
- A graph is connected if there is a path from every vertex to every other vertex in the graph.
- A graph that is not connected consists of a set of connected components, which are maximal connected subgraphs.
What are Graphs?

The degree of a vertex is the number of vertices connected to it.
What are Graphs?

The degree of a vertex is the number of vertices connected to it.

A path is a sequence of vertices connected by edges.
What are Graphs?

The degree of a vertex is the number of vertices connected to it.

A path is a sequence of vertices connected by edges.

A cycle is a path with at least one edge whose first and last vertices are the same.
What are Graphs?

The degree of a vertex is the number of vertices connected to it.

A path is a sequence of vertices connected by edges.

A cycle is a path with at least one edge whose first and last vertices are the same.

The length of a path or a cycle is its number of edges.
What are Graphs?

The degree of a vertex is the number of vertices connected to it.

A path is a sequence of vertices connected by edges.

A cycle is a path with at least one edge whose first and last vertices are the same.

The length of a path or a cycle is its number of edges.

A graph is connected if there is a path from every vertex to every other vertex in the graph.
What are Graphs?

The degree of a vertex is the number of vertices connected to it

A path is a sequence of vertices connected by edges

A cycle is a path with at least one edge whose first and last vertices are the same

The length of a path or a cycle is its number of edges

A graph is connected if there is a path from every vertex to every other vertex in the graph

A graph that is not connected consists of a set of connected components, which are maximal connected subgraphs
What are Graphs?

An acyclic graph is a graph with no cycles.

A tree is an acyclic connected graph.

A bipartite graph is a graph whose vertices can be divided into two sets such that all edges connect a vertex in one set with a vertex in the other set.
An acyclic graph is a graph with no cycles
What are Graphs?

An acyclic graph is a graph with no cycles.

A tree is an acyclic connected graph.
What are Graphs?

An acyclic graph is a graph with no cycles.

A tree is an acyclic connected graph.

A bipartite graph is a graph whose vertices can be divided into two sets such that all edges connect a vertex in one set with a vertex in the other set.
What are Graphs?

Graph applications

- Communication: telephone, computer, fiber optic cable, circuit gate, register, processor, wire
- Mechanical: joint, rod, beam, spring
- Financial: stock, currency transactions
- Transportation: intersection, street
- Internet: class C network connection
- Game: board position, legal move
- Social: relationship, person, friendship
- Neural network: neuron, synapse
- Protein: network, protein, protein-protein interaction
- Molecule: atom, bond
What are Graphs?

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>
What are Graphs?

Example: Internet graph
What are Graphs?

Example: Internet graph
What are Graphs?

Example: facebook graph
What are Graphs?

Example: facebook graph
What are Graphs?

Example: c.elegans connectome graph
What are Graphs?

Example: c.elegans connectome graph
What are Graphs?

Example: coauthorship graph
What are Graphs?

Example: coauthorship graph
What are Graphs?

Some graph-processing problems

Problem Description

- $s-t$ path is there a path between $s$ and $t$?
- Shortest $s-t$ path what is the shortest path between $s$ and $t$?
What are Graphs?

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>
</tbody>
</table>
Undirected Graphs

Graph(int V)
create a V-vertex graph with no edges

Graph(In in)
read a graph from input stream

int V()
number of vertices

int E()
number of edges

void addEdge(int v, int w)
add edge v-w to this graph

Iterable<Integer> adj(int v)
vertices adjacent to v

int degree(int v)
degree of v

Graph input format

terminal

~/workspace/dsa/programs

$ more ../ data / tinyG . txt

13 13
0 5 4 3 0 1 9 12 6 4 5 4 0 2
11 12 9 10 0 6 7 8 9 11 5 3

Typical graph-processing code

public static int degree ( Graph G, int v) {
    int degree = 0;
    for ( int w : G.adj(v)) {
        degree ++;
    }
    return degree;
}
### Undirected Graphs

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Graph(int V)</code></td>
<td>create a ( V )-vertex graph with no edges</td>
</tr>
<tr>
<td><code>Graph(In in)</code></td>
<td>read a graph from input stream ( in )</td>
</tr>
<tr>
<td><code>int V()</code></td>
<td>number of vertices</td>
</tr>
<tr>
<td><code>int E()</code></td>
<td>number of edges</td>
</tr>
<tr>
<td><code>void addEdge(int v, int w)</code></td>
<td>add edge ( v-w ) to this graph</td>
</tr>
<tr>
<td><code>Iterable&lt;Integer&gt; adj(int v)</code></td>
<td>vertices adjacent to ( v )</td>
</tr>
<tr>
<td><code>int degree(int v)</code></td>
<td>degree of ( v )</td>
</tr>
</tbody>
</table>

**Graph input format**

```
~/workspace/dsa/programs
$ more ../ data / tinyG . txt
13 13
0 5 4 3 0 1 9 12 6 4 5 4 0 2
11 12 9 10 0 6 7 8 9 11 5 3
```

**Typical graph-processing code**

```java
public static int degree ( Graph G, int v) {
    int degree = 0;
    for ( int w : G. adj (v)) {
        degree ++;
    }
    return degree ;
}
```
### Undirected Graphs

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Graph(int V)</code></td>
<td>create a $V$-vertex graph with no edges</td>
</tr>
<tr>
<td><code>Graph(In in)</code></td>
<td>read a graph from input stream <code>in</code></td>
</tr>
<tr>
<td><code>int V()</code></td>
<td>number of vertices</td>
</tr>
<tr>
<td><code>int E()</code></td>
<td>number of edges</td>
</tr>
<tr>
<td><code>void addEdge(int v, int w)</code></td>
<td>add edge $v$-$w$ to this graph</td>
</tr>
<tr>
<td><code>Iterable&lt;Integer&gt; adj(int v)</code></td>
<td>vertices adjacent to $v$</td>
</tr>
<tr>
<td><code>int degree(int v)</code></td>
<td>degree of $v$</td>
</tr>
</tbody>
</table>

#### Graph input format

```
> ~/workspace/dsa/programs
$ more ../ data / tinyG . txt
13 13
0 5 4 3 0 1 9 12 6 4 5 4 0 2
11 12 9 10 0 6 7 8 9 11 5 3
```

```
0
1 2
3
4
5
6 7 8
9 10
11 12
```

```java
typical graph-processing code
public static int degree ( Graph G, int v) {
    int degree = 0;
    for ( int w : G. adj (v)) {
        degree ++;
    }
    return degree ;
}
```
Undirected Graphs

Graph

Graph(int V) create a V-vertex graph with no edges
Graph(In in) read a graph from input stream in
int V() number of vertices
int E() number of edges
void addEdge(int v, int w) add edge v-w to this graph
Iterable<Integer> adj(int v) vertices adjacent to v
int degree(int v) degree of v

Graph input format

> ~/workspace/dsa/programs
$ more ../data/tinyG.txt
13 13
0 5 4 3 0 1 9 12 6 4 5 4 0 2
11 12 9 10 0 6 7 8 9 11 5 3

Typical graph-processing code

```java
public static int degree(Graph G, int v) {
    int degree = 0;
    for (int w : G.adj(v)) {
        degree++;
    }
    return degree;
}
```
Undirected Graphs

Graph representations

- Edge list: maintain a list of the edges (linked list or array)
- Adjacency matrix: maintain a $V \times V$ matrix $M$, such that $M[v][w]$ is 1 if there is an edge from $v$ to $w$, and 0 otherwise
- Adjacency list: maintain a vertex-indexed array of lists

Performance characteristics

<table>
<thead>
<tr>
<th>Representation</th>
<th>Space</th>
<th>Add edge</th>
<th>Is $v$–$w$ an edge?</th>
<th>Enumerate $\text{adj}(v)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>edge list</td>
<td>$E$</td>
<td>$E$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>adjacency matrix</td>
<td>$V$</td>
<td>1</td>
<td>1</td>
<td>$\text{degree}(v)$</td>
</tr>
<tr>
<td>adjacency list</td>
<td>$E + V$</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Undirected Graphs

Graph representations

- Edge list: maintain a list of the edges (linked list or array)
- Adjacency matrix: maintain a $V$-by-$V$ matrix $M$, such that $M[v][w]$ is 1 if there is an edge from $v$ to $w$, and 0 otherwise
- Adjacency list: maintain a vertex-indexed array of lists
Undirected Graphs

Graph representations

- Edge list: maintain a list of the edges (linked list or array)
- Adjacency matrix: maintain a $V$-by-$V$ matrix $M$, such that $M[v][w]$ is 1 if there is an edge from $v$ to $w$, and 0 otherwise
- Adjacency list: maintain a vertex-indexed array of lists

Performance characteristics

<table>
<thead>
<tr>
<th>Representation</th>
<th>Space</th>
<th>Add edge</th>
<th>Is $v$-$w$ an edge?</th>
<th>Enumerate $\text{adj}(v)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>edge list</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 list</td>
<td>$E + V$</td>
<td>1</td>
<td>$\text{degree}(v)$</td>
<td>$\text{degree}(v)$</td>
</tr>
</tbody>
</table>
package dsa;
import stdlib.In;
import stdlib.StdOut;
public class Graph {
    private LinkedBag<Integer>[] adj;
    private int V;
    private int E;
    public Graph(int V) {
        adj = (LinkedBag<Integer>[]) new LinkedBag[V];
        for (int v = 0; v < V; v++) {
            adj[v] = new LinkedBag<Integer>();
        }
        this.V = V;
        this.E = 0;
    }
    public Graph(In in) {
        this(in.readInt());
        int E = in.readInt();
        for (int i = 0; i < E; i++) {
            int v = in.readInt();
            int w = in.readInt();
            addEdge(v, w);
        }
    }
    public int V() {
        return V;
    }
    public int E() {
        return E;
    }
}
package dsa;

import stdlib.In;
import stdlib.StdOut;

public class Graph {
    private LinkedBag<Integer>[] adj;
    private int V;
    private int E;

    public Graph(int V) {
        adj = (LinkedBag<Integer>[]) new LinkedBag[V];
        for (int v = 0; v < V; v++) {
            adj[v] = new LinkedBag<Integer>();
        }
        this.V = V;
        this.E = 0;
    }

    public Graph(In in) {
        this(in.readInt());
        int E = in.readInt();
        for (int i = 0; i < E; i++) {
            int v = in.readInt();
            int w = in.readInt();
            addEdge(v, w);
        }
    }

    public int V() {
        return V;
    }

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

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

public int degree(int v) {
    return adj[v].size();
}

public String toString() {
    StringBuilder sb = new StringBuilder();
    sb.append(V + " vertices, " + E + " edges\n");
    for (int v = 0; v < V; v++) {
        sb.append(v + " : ");
        for (int w : adj[v]) {
            sb.append(w + " ");
        }
        sb.append("\n");
    }
    return sb.toString().strip();
}

public static void main(String[] args) {
    String filename = args[0];
    In in = new In(filename);
    Graph G = new Graph(in);
    StdOut.println(G);
}
Undirected Graphs
Depth-First Search (DFS)

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

To visit a vertex $v$

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

Data structures

• Boolean array $marked[\cdot]$ to mark visited vertices
• Integer array $edgeTo[\cdot]$ to keep track of paths; $edgeTo[w] = v$ means that edge $v - w$ taken to visit $w$ for first time
Depth-First Search (DFS)

Goal: systematically traverse a graph
Depth-First Search (DFS)

Goal: systematically traverse a graph

Idea: mimic maze exploration
Depth-First Search (DFS)

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
Depth-First Search (DFS)

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

To visit a vertex \( v \)
- Mark vertex \( v \) as visited
- Recursively visit all unmarked vertices adjacent to \( v \)
Depth-First Search (DFS)

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

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

Data structures
- Boolean array `marked[]` to mark visited vertices
- Integer array `edgeTo[]` to keep track of paths; `edgeTo[w] = v` means that edge $v$-$w$ taken to visit $w$ for first time
Depth-First Search (DFS)

Design pattern for graph processing: decouple graph data type from graph processing

- Create a `Graph` object
- Pass the `Graph` object to a graph-processing routine
- Query the graph-processing routine for information

/\* list Paths boolean hasPathTo(int v) is there a path from s to v? \\
Iterable<Integer> pathTo(int v) path from s to v, or null \\
Typical graph-processing code

DFSPaths paths = new DFSPaths (G, s);
for ( int v = 0; v < G.V (); v ++) {
    if ( paths . hasPathTo (v)) {
        StdOut . println (v);
    }
}
Depth-First Search (DFS)

Design pattern for graph processing: decouple graph data type from graph processing

- Create a `Graph` object
- Pass the `Graph` object to a graph-processing routine
- Query the graph-processing routine for information

| Paths
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean hasPathTo(int v)</td>
</tr>
<tr>
<td>Iterable&lt;Integer&gt; pathTo(int v)</td>
</tr>
</tbody>
</table>
**Depth-First Search (DFS)**

Design pattern for graph processing: decouple graph data type from graph processing

- Create a `Graph` object
- Pass the `Graph` object to a graph-processing routine
- Query the graph-processing routine for information

<table>
<thead>
<tr>
<th>Paths</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean hasPathTo(int v)</td>
<td>is there a path from s to v?</td>
</tr>
<tr>
<td>Iterable&lt;Integer&gt; pathTo(int v)</td>
<td>path from s to v, or null</td>
</tr>
</tbody>
</table>

**Typical graph-processing code**

```java
DFSPaths paths = new DFSPaths(G, s);
for (int v = 0; v < G.V(); v++) {
    if (paths.hasPathTo(v)) {
        StdOut.println(v);
    }
}
```
Depth-First Search (DFS)

```java
package dsa;
import stdlib.In;
import stdlib.StdOut;
public class DFSPaths {
    private int s;
    private boolean[] marked;
    private int[] edgeTo;
    public DFSPaths(Graph G, int s) {
        this.s = s;
        marked = new boolean[G.V()];
        edgeTo = new int[G.V()];
        dfs(G, s);
    }

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

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

    private void dfs(Graph G, int v) {
        marked[v] = true;
        // Additional code for DFS traversal
    }
```
package dsa;
import stdlib.In;
import stdlib.StdOut;

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

    public DFSPaths(Graph G, int s) {
        this.s = s;
        marked = new boolean [G.V()];
        edgeTo = new int [G.V()];
        dfs(G, s);
    }

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

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

    private void dfs(Graph G, int v) {
        marked[v] = true;
    }
}

Depth-First Search (DFS)
public static void main(String[] args) {
    In in = new In(args[0]);
    int s = Integer.parseInt(args[1]);
    Graph G = new Graph(in);
    DFSPaths dfs = new DFSPaths(G, s);
    for (int v = 0; v < G.V(); v++) {
        if (dfs.hasPathTo(v)) {
            StdOut.printf("%d to %d: ", s, v);
            for (int x : dfs.pathTo(v)) {
                if (x == s) {
                    StdOut.print(x);
                } else {
                    StdOut.print("-"+ x);
                }
            }
            StdOut.println();
        } else {
            StdOut.printf("%d to %d: not connected\n", s, v);
        }
    }
}
Depth-First Search (DFS)
Depth-First Search (DFS)

Trace

dfs(0)

marked[]
adj[]
0 T
1
2 T
3
4
5

dfs(2)
check 0

dfs(1)
check 0
check 2
1 done

dfs(3)

dfs(5)
check 3
check 0
5 done

dfs(4)
check 3
check 2
4 done
check 2
3 done
check 4
2 done
Breadth-First Search (BFS)

Goal: given a graph and a source vertex $s$, support queries of the form

- Is there a path from $s$ to a given target vertex $v$?
- If so, find a shortest such path (one with minimal number of edges)

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 (BFS)

Goal: given a graph and a source vertex $s$, support queries of the form

- Is there a path from $s$ to a given target vertex $v$?
- If so, find a shortest such path (one with minimal number of edges)
Breadth-First Search (BFS)

Goal: given a graph and a source vertex $s$, support queries of the form

- Is there a path from $s$ to a given target vertex $v$?
- If so, find a shortest such path (one with minimal number of edges)

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 (BFS)

```java
package dsa;

import stdlib.In;
import stdlib.StdOut;

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

    public BFSPaths(Graph G, int s) {
        this.s = s;
        marked = new boolean[G.V()];
        distTo = new int[G.V()];
        for (int v = 0; v < G.V(); v++) {
            distTo[v] = Integer.MAX_VALUE;
        }
        edgeTo = new int[G.V()];
        bfs(G, s);
    }

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

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

import stdlib.In;
import stdlib.StdOut;

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

    public BFSPaths(Graph G, int s) {
        this.s = s;
        marked = new boolean[G.V()];
        distTo = new int[G.V()];
        for (int v = 0; v < G.V(); v++) {
            distTo[v] = Integer.MAX_VALUE;
        }
        edgeTo = new int[G.V()];
        bfs(G, s);
    }

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

    public Iterable<Integer> pathTo(int v) {
        if (!hasPathTo(v)) {
            return null;
        }
        LinkedStack<Integer> path = new LinkedStack<Integer>();
        for (int x = v; x != s; x = edgeTo[x]) {
            path.push(x);
        }
        path.push(s);
    }
}
Breadth-First Search (BFS)

```java
return path;
}

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

private void bfs(Graph G, int s) {
    LinkedQueue<Integer> q = new LinkedQueue<Integer>();
    marked[s] = true;
    distTo[s] = 0;
    q.enqueue(s);
    while (!q.isEmpty()) {
        int v = q.dequeue();
        for (int w : G.adj(v)) {
            if (!marked[w]) {
                marked[w] = true;
                edgeTo[w] = v;
                distTo[w] = distTo[v] + 1;
                q.enqueue(w);
            }
        }
    }
}

public static void main(String[] args) {
    In in = new In(args[0]);
    int s = Integer.parseInt(args[1]);
    Graph G = new Graph(in);
    BFSPaths bfs = new BFSPaths(G, s);
    for (int v = 0; v < G.V(); v++) {
        if (bfs.hasPathTo(v)) {
            StdOut.printf("%d to %d (%d): ", s, v, bfs.distTo(v));
            for (int x : bfs.pathTo(v)) {
                if (x == s) {
                    break;
                }
                StdOut.print(x + " ");
            }
            StdOut.println();
        }
    }
```
Breadth-First Search (BFS)

```java
StdOut.print(x);
} else {
    StdOut.print("-" + x);
}
}
StdOut.println();
} else {
    StdOut.printf("%d to %d (-): not connected\n", s, v);
}
}
```
Breadth-First Search (BFS)
**Breadth-First Search (BFS)**

**Trace**

<table>
<thead>
<tr>
<th>q</th>
<th>marked[]</th>
<th>edgeTo[]</th>
<th>adj[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>0</td>
<td>2 1 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1 0 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>2 0 1 3 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>3 5 4 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>4 3 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>5 3 0</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>0</td>
<td>2 1 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1 0 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>2 0 1 3 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>3 5 4 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>4 3 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>5 3 0</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>0</td>
<td>2 1 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1 0 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>2 0 1 3 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>3 5 4 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>4 3 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>5 3 0</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>0</td>
<td>2 1 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1 0 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>2 0 1 3 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>3 5 4 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>4 3 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>5 3 0</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>0</td>
<td>2 1 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1 0 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>2 0 1 3 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>3 5 4 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>4 3 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>5 3 0</td>
</tr>
</tbody>
</table>
Symbol Graphs

Typical applications involve processing graphs defined in files or on web pages, using strings, not integer indices, to define and refer to vertices. To accommodate such applications, we define an input format with these properties:

- Vertex names are strings.
- A specified delimiter separates vertex names (to allow for the possibility of spaces in names).
- Each line represents a set of edges, connecting the first vertex name on the line to each of the other vertices named on the line.
- The number of vertices \( V \) and the number of edges \( E \) are both implicitly defined.

Example (routes.txt):
```
/terminal
~/workspace/dsa/programs
JFK MCO
ORD DEN
ORD HOU
DFW PHX
JFK ATL
...
```

Example (movies.txt):
```
/terminal
~/workspace/dsa/programs
'Breaker' Morant (1980)/ Brown, Bryan (I)/ Henderson, Dick (II)/...
'burbs, The (1989)/ Jayne, Billy / Howard, Rance / Ducommun, Rick /...
'Crocodile Dundee II (1988)/ Jbara, Gregory /Holt, Jim (I )/...
*batteries not included (1987)/ Aldredge, Tom / Boutsikaris, Dennis /...
... And Justice for All (1979)/ Williams, Jonathan (XI )/...
```
Symbol Graphs

Typical applications involve processing graphs defined in files or on web pages, using strings, not integer indices, to define and refer to vertices.
Symbol Graphs

Typical applications involve processing graphs defined in files or on web pages, using strings, not integer indices, to define and refer to vertices.

To accommodate such applications, we define an input format with these properties:

- Vertex names are strings.
- A specified delimiter separates vertex names (to allow for the possibility of spaces in names).
- Each line represents a set of edges, connecting the first vertex name on the line to each of the other vertices named on the line.
- The number of vertices \( V \) and the number of edges \( E \) are both implicitly defined.
Symbol Graphs

Typical applications involve processing graphs defined in files or on web pages, using strings, not integer indices, to define and refer to vertices.

To accommodate such applications, we define an input format with these properties:

- Vertex names are strings.
- A specified delimiter separates vertex names (to allow for the possibility of spaces in names).
- Each line represents a set of edges, connecting the first vertex name on the line to each of the other vertices named on the line.
- The number of vertices $V$ and the number of edges $E$ are both implicitly defined.

Example (routes.txt)

```plaintext
> _ ~/workspace/dsa/programs

JFK MCO
ORD DEN
ORD HOU
DFW PHX
JFK ATL
...
```

Example (movies.txt)

```plaintext
'Breaker' Morant (1980)/ Brown, Bryan (I)/ Henderson, Dick (II)/...
'burbs, The (1989)/ Jayne, Billy / Howard, Rance / Ducommun, Rick /...
'Crocodile Dundee II (1988)/ Jbara, Gregory / Holt, Jim (I)/...
*batteries not included (1987)/ Aldredge, Tom / Boutsikaris, Dennis /...
... And Justice for All (1979)/ Williams, Jonathan (XI)/...
...
Symbol Graphs

Typical applications involve processing graphs defined in files or on web pages, using strings, not integer indices, to define and refer to vertices.

To accommodate such applications, we define an input format with these properties:

- Vertex names are strings.
- A specified delimiter separates vertex names (to allow for the possibility of spaces in names).
- Each line represents a set of edges, connecting the first vertex name on the line to each of the other vertices named on the line.
- The number of vertices $V$ and the number of edges $E$ are both implicitly defined.

Example (routes.txt)

```
> ~/workspace/dsa/programs

JFK MCO
ORD DEN
ORD HOU
DFW PHX
JFK ATL
...
```

Example (movies.txt)

```
> ~/workspace/dsa/programs

'Breaker' Morant (1980)/Brown, Bryan (I)/Henderson, Dick (II)/...
'burbs, The (1989)/Jayne, Billy/Howard, Rance/Ducommun, Rick/...
'Crocodile' Dundee II (1988)/Jbara, Gregory/Holt, Jim (I)/...
*batteries not included (1987)/Aldredge, Tom/Boutsikaris, Dennis/...
... And Justice for All (1979)/Williams, Jonathan (XI)/...
...
Symbol Graphs

API for graphs with symbolic vertex names

SymbolGraph

SymbolGraph(String filename, String delim)
build graph specified in filename using delim to separate vertex names

boolean contains(String key)
is key a vertex?

int indexOf(String key)
index associated with key

String nameOf(int v)
key associated with index v

Graph G()
underlying graph as a Graph object
### Symbol Graphs

API for graphs with symbolic vertex names

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SymbolGraph(String filename, String delim)</td>
<td>build graph specified in <em>filename</em> using <em>delim</em> to separate vertex names</td>
</tr>
<tr>
<td>boolean contains(String key)</td>
<td>is <em>key</em> a vertex?</td>
</tr>
<tr>
<td>int indexOf(String key)</td>
<td>index associated with <em>key</em></td>
</tr>
<tr>
<td>String nameOf(int v)</td>
<td>key associated with index <em>v</em></td>
</tr>
<tr>
<td>Graph G()</td>
<td>underlying graph as a <em>Graph</em> object</td>
</tr>
</tbody>
</table>
Symbol Graphs

```
$ java dsa.SymbolGraph ../data/routes.txt ""
Done reading routes.txt
JFK
ORD
ATL
MCO
LAX
LAS
PHX
<ctrl-d>
```

```
$ java dsa.SymbolGraph ../data/movies.txt "/"
Done reading movies.txt
Tin Men (1987)
Hershey , Barbara
Geppi , Cindy
...
Blumenfeld , Alan
DeBoy , David
Bacon , Kevin
Wild Things (1998)
...
Apollo 13 (1995)
Animal House (1978)
<ctrl-d>
```
Symbol Graphs

$ java dsa.SymbolGraph ../data/routes.txt " "
Done reading routes.txt
JFK
   ORD
   ATL
   MCO
LAX
   LAS
   PHX
<ctrl-d>

$ java dsa.SymbolGraph ../data/movies.txt "/"
Done reading movies.txt
Tin Men (1987)
   Hershey, Barbara
   Geppi, Cindy
   ... 
   Blumenfeld, Alan
   DeBoy, David
Bacon, Kevin
   Wild Things (1998)
   ... 
   Apollo 13 (1995)
   Animal House (1978)
<ctrl-d>
Symbol Graphs

$ java dsa.SymbolGraph ..data/routes.txt " "
Done reading routes.txt
JFK
   ORD
   ATL
   MCO
LAX
   LAS
   PHX
<ctrl-d>

$ java dsa.SymbolGraph ..data/movies.txt "/"
Done reading movies.txt
Tin Men (1987)
   Hershey, Barbara
   Geppi, Cindy
   ...
   Blumenfeld, Alan
   DeBoy, David
Bacon, Kevin
   Wild Things (1998)
   ...
   Apollo 13 (1995)
   Animal House (1978)
<ctrl-d>
Symbol Graphs

```
package dsa;
import stdlib.In;
import stdlib.StdIn;
import stdlib.StdOut;

public class SymbolGraph {
    private SeparateChainingHashST<String, Integer> st;
    private String[] keys;
    private Graph G;
    public SymbolGraph(In in, String delim) {
        st = new SeparateChainingHashST<>();
        String[] lines = in.readAllLines();
        for (String line : lines) {
            String[] a = line.split(delim);
            for (int i = 0; i < a.length; i++) {
                if (!st.contains(a[i])) {
                    st.put(a[i], st.size());
                }
            }
        }
        keys = new String[st.size()];
        for (String name : st.keys()) {
            keys[st.get(name)] = name;
        }
        G = new Graph(st.size());
        for (String line : lines) {
            String[] a = line.split(delim);
            int v = st.get(a[0]);
            for (int i = 1; i < a.length; i++) {
                int w = st.get(a[i]);
                G.addEdge(v, w);
            }
        }
    }
}
```
Symbol Graphs

```java
package dsa;

import stdlib.In;
import stdlib.StdIn;
import stdlib.StdOut;

public class SymbolGraph {
    private SeparateChainingHashST<String, Integer> st;
    private String[] keys;
    private Graph G;

    public SymbolGraph(In in, String delim) {
        st = new SeparateChainingHashST<>();
        String[] lines = in.readAllLines();
        for (String line : lines) {
            String[] a = line.split(delim);
            for (int i = 0; i < a.length; i++) {
                if (!st.contains(a[i])) {
                    st.put(a[i], st.size());
                }
            }
        }
        keys = new String[st.size()];
        for (String name : st.keys()) {
            keys[st.get(name)] = name;
        }
        G = new Graph(st.size());
        for (String line : lines) {
            String[] a = line.split(delim);
            int v = st.get(a[0]);
            for (int i = 1; i < a.length; i++) {
                int w = st.get(a[i]);
                G.addEdge(v, w);
            }
        }
    }
}
```
public boolean contains(String s) {
    return st.contains(s);
}

public int indexOf(String s) {
    return st.get(s);
}

public String nameOf(int v) {
    return keys[v];
}

public Graph graph() {
    return G;
}

public static void main(String[] args) {
    In in = new In(args[0]);
    String delim = args[1];
    SymbolGraph sg = new SymbolGraph(in, delim);
    Graph graph = sg.graph();
    while (!StdIn.isEmpty()) {
        String source = StdIn.readLine();
        if (sg.contains(source)) {
            int s = sg.indexOf(source);
            for (int v : graph.adj(s)) {
                StdOut.println(sg.nameOf(v));
            }
        } else {
            StdOut.println(source + " not in database");
        }
    }
}
Symbol Graphs

```java
}
```
import dsa.BFSPaths;
import dsa.Graph;
import dsa.SymbolGraph;
import stdlib.In;
import stdlib.StdIn;
import stdlib.StdOut;

public class DegreesOfSeparation {
    public static void main(String[] args) {
        String filename = args[0];
        String delim = args[1];
        String source = args[2];
        In in = new In(filename);
        SymbolGraph sg = new SymbolGraph(in, delim);
        Graph G = sg.graph();
        if (!sg.contains(source)) {
            StdOut.println(source + " not in database");
            return;
        }
        int s = sg.indexOf(source);
        BFSPaths bfs = new BFSPaths(G, s);
        while (!StdIn.isEmpty()) {
            String sink = StdIn.readLine();
            if (sg.contains(sink)) {
                int t = sg.indexOf(sink);
                if (bfs.hasPathTo(t)) {
                    for (int v : bfs.pathTo(t)) {
                        StdOut.println(" " + sg.nameOf(v));
                    }
                } else {
                    StdOut.println(source + " and " + sink + " are not connected");
                }
            } else {
                StdOut.println(sink + " not in database");
            }
        }
    }
}
import dsa.BFSPaths;
import dsa.Graph;
import dsa.SymbolGraph;
import stdlib.In;
import stdlib.StdIn;
import stdlib.StdOut;

public class DegreesOfSeparation {

    public static void main(String[] args) {
        String filename = args[0];
        String delim = args[1];
        String source = args[2];
        In in = new In(filename);
        SymbolGraph sg = new SymbolGraph(in, delim);
        Graph G = sg.graph();
        if (!sg.contains(source)) {
            StdOut.println(source + " not in database");
            return;
        }
        int s = sg.indexOf(source);
        BFSPaths bfs = new BFSPaths(G, s);
        while (!StdIn.isEmpty()) {
            String sink = StdIn.readLine();
            if (sg.contains(sink)) {
                int t = sg.indexOf(sink);
                if (bfs.hasPathTo(t)) {
                    for (int v : bfs.pathTo(t)) {
                        StdOut.println(sg.nameOf(v));
                    }
                } else {
                    StdOut.println(source + " and " + sink + " are not connected");
                }
            } else {
                StdOut.println(sink + " not in database");
            }
        }
    }
}
```java
}
}
```
Symbol Graphs

$java$ DegreesOfSeparation ../data/routes.txt " " JFK

Done reading routes.txt

LAS
JFK
ORD
PHX
LAS
DFW
JFK
ORD
DFW
<ctrl-d>

$java$ DegreesOfSeparation ../data/movies.txt "/" " Bacon, Kevin ">

Done reading movies.txt

Kidman, Nicole
Bacon, Kevin
Grier, David Alan
Bewitched (2005)
Kidman, Nicole
Grant, Cary
Bacon, Kevin
Planes, Trains & Automobiles (1987)
Martin, Steve (I)
Dead Men Don't Wear Plaid (1982)
Grant, Cary

<ctrl-d>
Symbol Graphs

$ java DegreesOfSeparation ..../data/routes.txt " " JFK
Done reading routes.txt
LAS
    JFK
    ORD
    PHX
    LAS
DFW
    JFK
    ORD
    DFW
<ctrl-d>
Symbol Graphs

> `~/workspace/dsa/programs`

```
$ java DegreesOfSeparation ../data/routes.txt " " JFK
Done reading routes.txt
LAS
    JFK
    ORD
    PHX
    LAS
DFW
    JFK
    ORD
    DFW
<ctrl-d>
```

> `~/workspace/dsa/programs`

```
$ java DegreesOfSeparation ../data/movies.txt "/" "Bacon, Kevin"
Done reading movies.txt
Kidman, Nicole
    Bacon, Kevin
    Grier, David Alan
    Bewitched (2005)
    Kidman, Nicole
Grant, Cary
    Bacon, Kevin
    Planes, Trains & Automobiles (1987)
    Martin, Steve (I)
    Dead Men Don't Wear Plaid (1982)
    Grant, Cary
<ctrl-d>
```