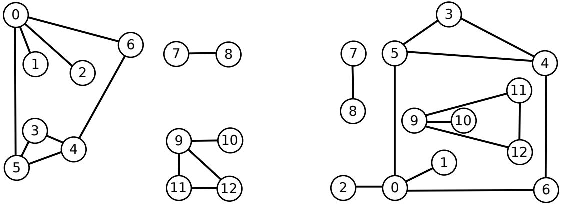
Undirected Graphs

Outline

What are Graphs?

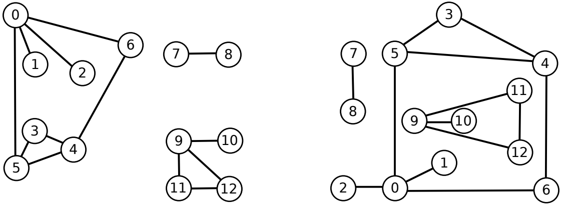
What are Graphs?

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



What are Graphs?

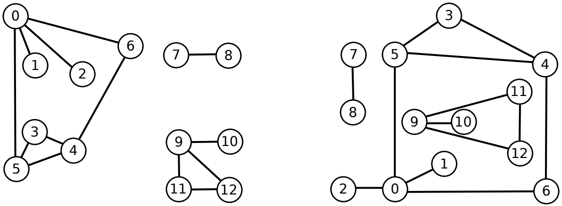
A graph is a set of V vertices connected pairwise by E edges



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

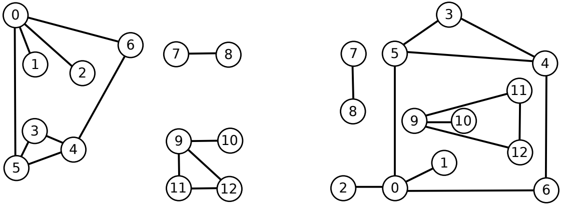


We use the names 0 through $V - 1$ for the vertices in a V-vertex graph

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



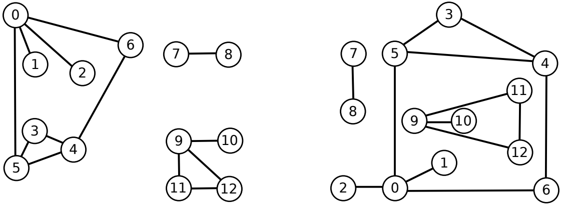
We use the names 0 through $V - 1$ for the vertices in a V-vertex graph

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-w to refer to an edge that connects vertices v and w

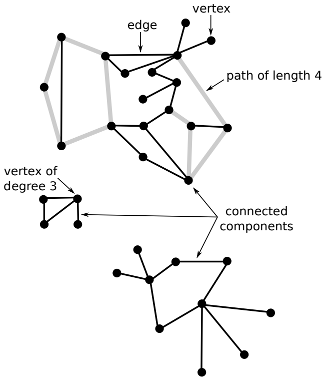
A self-loop is an edge that connects a vertex to itself

Parallel edges are edges that connect the same pair of vertices

What are Graphs?

What are Graphs?

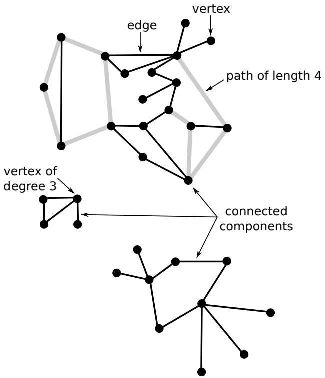
The degree of a vertex is the number of vertices connected to it



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

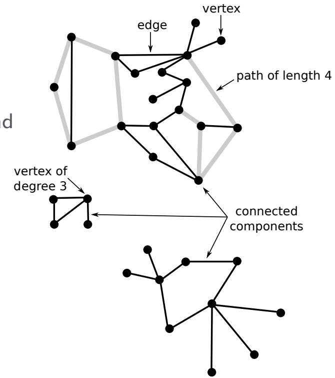


What are Graphs?

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



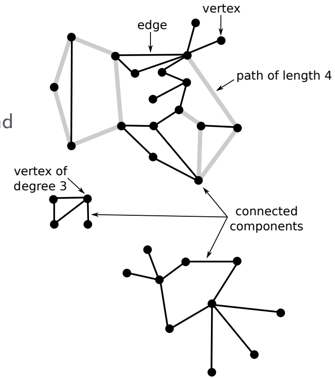
What are Graphs?

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?

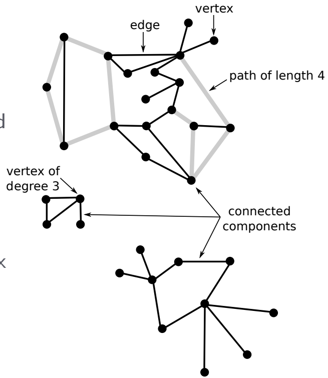
The degree of a vertex is the number of vertices connected to it

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

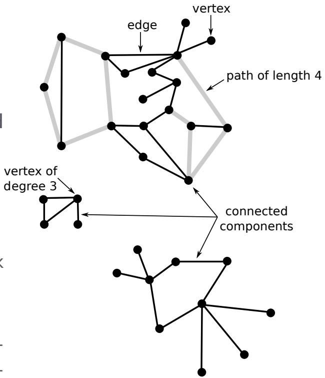
A path is a sequence of vertices connected by edges

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

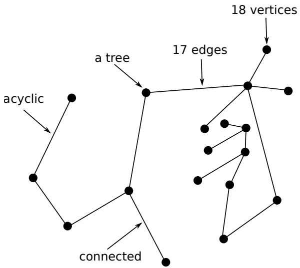
A graph that is not connected consists of a set of connected components, which are maximal connected subgraphs

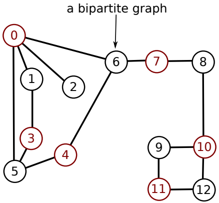


What are Graphs?

What are Graphs?

An acyclic graph is a graph with no cycles

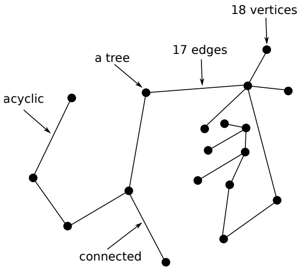


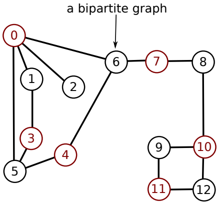


What are Graphs?

An acyclic graph is a graph with no cycles

A tree is an acyclic connected graph



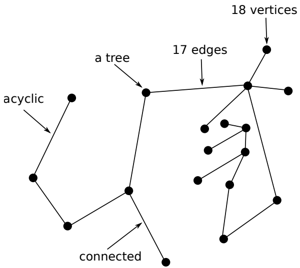


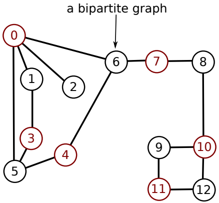
What are Graphs?

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?

What are Graphs?

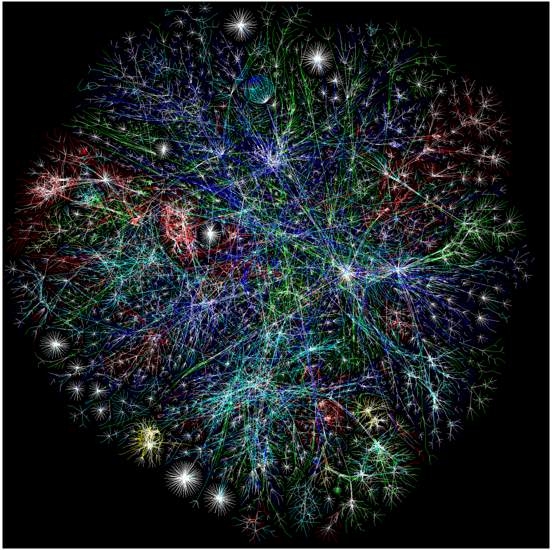
Graph applications

Graph	Vertex	Edge
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?

What are Graphs?

Example: Internet graph



What are Graphs?

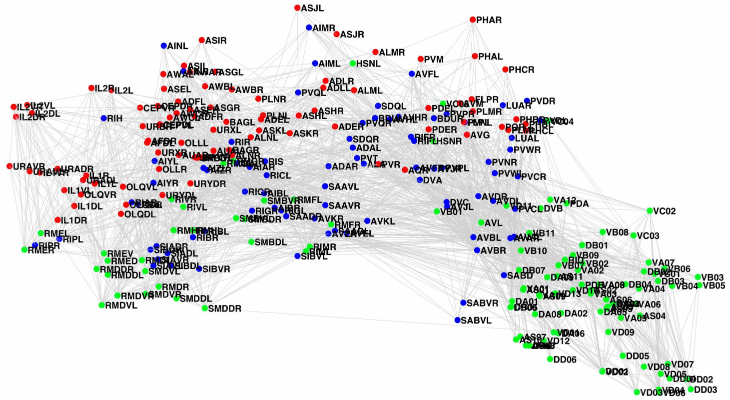
What are Graphs?

Example: facebook graph

What are Graphs?

What are Graphs?

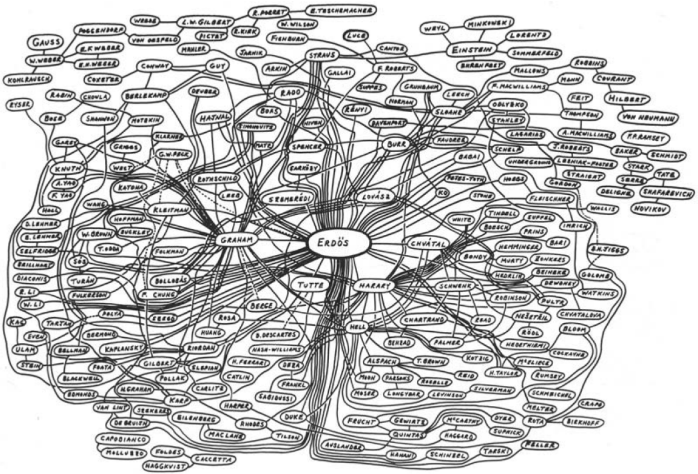
Example: c.elegans connectome graph



What are Graphs?

What are Graphs?

Example: coauthorship graph



What are Graphs?

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?

Undirected Graphs

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

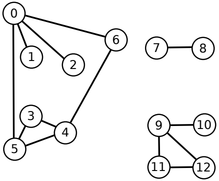
Undirected Graphs

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
```



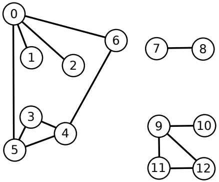
Undirected Graphs

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

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

Undirected Graphs

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

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

Representation	Space	Add edge	Is v-w an edge?	Enumerate `adj(v)`
edge list	E	1	E	E
adjacency matrix	V^2	1	1	V
adjacency list	$E + V$	1	$degree(v)$	$degree(v)$

Undirected Graphs

Undirected Graphs

Graph.java

```
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;
```

Undirected Graphs

Graph.java

```java
    }

    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

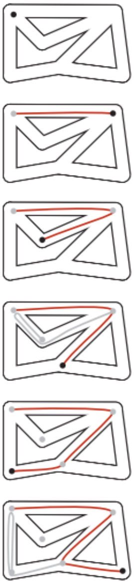
Graph.java

```
}
```

Depth-First Search (DFS)

Depth-First Search (DFS)

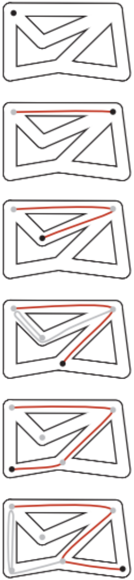
Goal: systematically traverse a graph



Depth-First Search (DFS)

Goal: systematically traverse a graph

Idea: mimic maze exploration



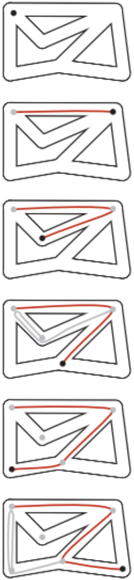
Depth-First Search (DFS)

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

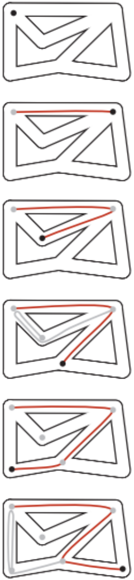
Idea: mimic maze exploration

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

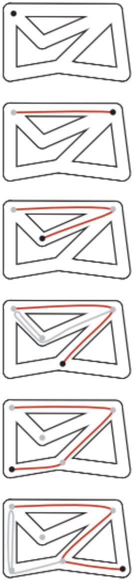
- Find all vertices connected to a given source vertex
- Find a path between two vertices

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)

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

`boolean hasPathTo(int v)`	is there a path from s to v?
`Iterable<Integer> pathTo(int v)`	path from s to v, or `null`

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

`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)

Depth-First Search (DFS)

DFSPaths.java

```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;
```

Depth-First Search (DFS)

DFSPaths.java

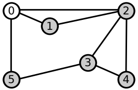
```
        for (int w : G.adj(v)) {
            if (!marked[w]) {
                edgeTo[w] = v;
                dfs(G, w);
            }
        }
    }

    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)

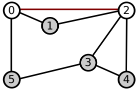
Depth-First Search (DFS)

Trace

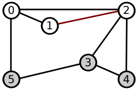
```
dfs(0)
  dfs(2)
    check 0
    dfs(1)
      check 0
      check 2
    1 done
    dfs(3)
      dfs(5)
        check 3
        check 0
      5 done
      dfs(4)
        check 3
        check 2
      4 done
      check 2
    3 done
    check 4
  2 done
```



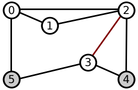
	marked[]	adj[]
0	T	2 1 5
1		0 2
2		0 1 3 4
3		5 4 2
4		3 2
5		3 0



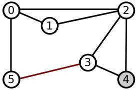
	marked[]	adj[]
0	T	2 1 5
1		0 2
2	T	0 1 3 4
3		5 4 2
4		3 2
5		3 0



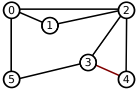
	marked[]	adj[]
0	T	2 1 5
1	T	0 2
2	T	0 1 3 4
3		5 4 2
4		3 2
5		3 0



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



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



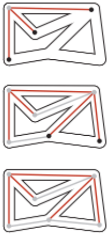
	marked[]	adj[]
0	T	2 1 5
1	T	0 2
2	T	0 1 3 4
3	T	5 4 2
4	T	3 2
5	T	3 0

Breadth-First Search (BFS)

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)



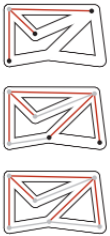
Breadth-First Search (BFS)

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)

Breadth-First Search (BFS)

BreadthFirstPaths.java

```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);
```

Breadth-First Search (BFS)

BreadthFirstPaths.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) {
```

Breadth-First Search (BFS)

BreadthFirstPaths.java

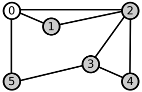
```
                    StdOut.print(x);
                } else {
                    StdOut.print("-" + x);
                }
            }
            StdOut.println();
        } else {
            StdOut.printf("%d to %d (-):  not connected\n", s, v);
        }
    }
  }
}
```

Breadth-First Search (BFS)

Breadth-First Search (BFS)

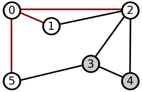
Trace

q: 0



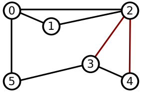
	marked[]	edgeTo[]	adj[]
0	T		2 1 5
1			0 2
2			0 1 3 4
3			5 4 2
4			3 2
5			3 0

q: 2 1 5



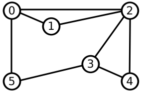
	marked[]	edgeTo[]	adj[]
0	T		2 1 5
1	T	0	0 2
2	T	0	0 1 3 4
3			5 4 2
4			3 2
5	T	0	3 0

q: 1 5 3 4



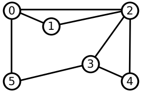
	marked[]	edgeTo[]	adj[]
0	T		2 1 5
1	T	0	0 2
2	T	0	0 1 3 4
3	T	2	5 4 2
4	T	2	3 2
5	T	0	3 0

q: 5 3 4



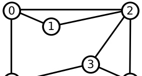
	marked[]	edgeTo[]	adj[]
0	T		2 1 5
1	T	0	0 2
2	T	0	0 1 3 4
3	T	2	5 4 2
4	T	2	3 2
5	T	0	3 0

q: 3 4



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

q: 4



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

Symbol Graphs

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`)

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

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

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

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

Symbol Graphs

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

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

Symbol Graphs

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

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

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

$ java dsa.SymbolGraph ../data/movies.txt "/"
Done reading movies.txt
Tin Men (1987)
   Hershey, Barbara
   Geppi, Cindy
   ...
   Blumenfeld, Alan
   DeBoy, David
Bacon, Kevin
   Woodsman, The (2004)
   Wild Things (1998)
   ...
   Apollo 13 (1995)
   Animal House (1978)
<ctrl-d>
```

Symbol Graphs

Symbol Graphs

SymbolGraph.java

```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);
            }
        }
```

Symbol Graphs

SymbolGraph.java

```java
    }

    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

SymbolGraph.java

```
}
```

Symbol Graphs

Symbol Graphs

DegreesOfSeparation.java

```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");
            }
```

Symbol Graphs

DegreesOfSeparation.java

```
            }
        }
}
```

Symbol Graphs

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>
```

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
   Woodsman, The (2004)
   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>
```