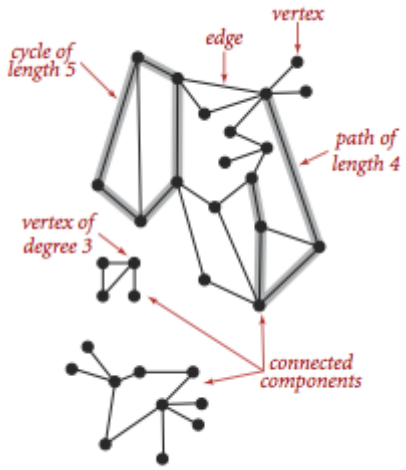
Undirected Graphs

Introduction

> set of vertices connected pairwise by edges

Terminology



Anatomy of a graph

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	street intersection, airport	highway, airway route
internet	class C network	connection
game	board position	legal move
social relationship	person, actor	friendship, movie cast
neural network	neuron	synapse
protein network	protein	protein protein interaction
molecule	atom	bond

Graph API

Undirected Graph API

public class	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
String	toString()	*string representation*

API for an undirected graph

```
/******************************************************************************
 *  13 13
 *  0: 6 2 1 5
 *  1: 0
 *  2: 0
 *  3: 5 4
 *  4: 5 6 3
 *  5: 3 4 0
 *  6: 0 4
 *  7: 8
 *  8: 7
 *  9: 11 10 12
 *  10: 9
 *  11: 9 12
 *  12: 11 9
 *
 *  vertex of maximum degree = 4
 *  average degree           = 2
 *  number of self loops     = 0
 *
 ******************************************************************************/

public class GraphClient {

    // maximum degree
    public static int maxDegree(Graph G) {
        int max = 0;
        for (int v = 0; v < G.V(); v++)
            if (G.degree(v) > max)
                max = G.degree(v);
        return max;
    }

    // average degree
    public static int avgDegree(Graph G) {
        // each edge incident on two vertices
        return 2 * G.E() / G.V();
    }

    // number of self-loops
```

```
    public static int numberOfSelfLoops(Graph G) {
        int count = 0;
        for (int v = 0; v < G.V(); v++)
            for (int w : G.adj(v))
                if (v == w) count++;
        return count/2;   // self loop appears in adjacency list twice
    }

    public static void main(String[] args) {
        In in = new In(args[0]);
        Graph G = new Graph(in);
        StdOut.println(G);

        StdOut.println("vertex of maximum degree = " + maxDegree(G));
        StdOut.println("average degree           = " + avgDegree(G));
        StdOut.println("number of self loops     = " + numberOfSelfLoops(G));
    }
}
```

Representation

adjacency-lists

> widely used

```
import java.util.NoSuchElementException;

public class Graph {
    private static final String NEWLINE =
System.getProperty("line.separator");

    private final int V;
    private int E;
    private Bag<Integer>[] adj;

    public Graph(int V) {
        if (V < 0) throw new IllegalArgumentException("Number of vertices must
be non-negative");
        this.V = V;
        this.E = 0;
        adj = (Bag<Integer>[]) new Bag[V];
        for (int v = 0; v < V; v++) {
            adj[v] = new Bag<Integer>();
        }
    }

    public Graph(In in) {
        if (in == null) throw new IllegalArgumentException("argument is
null");
        try {
            this.V = in.readInt();
```

```
            if (V < 0) throw new IllegalArgumentException("number of vertices must be non-negative");
            adj = (Bag<Integer>[]) new Bag[V];
            for (int v = 0; v < V; v++) {
                adj[v] = new Bag<Integer>();
            }
            int E = in.readInt();
            if (E < 0) throw new IllegalArgumentException("number of edges must be non-negative");
            for (int i = 0; i < E; i++) {
                int v = in.readInt();
                int w = in.readInt();
                validateVertex(v);
                validateVertex(w);
                addEdge(v, w);
            }
        }
        catch (NoSuchElementException e) {
            throw new IllegalArgumentException("invalid input format in Graph constructor", e);
        }
    }

    public Graph(Graph G) {
        this.V = G.V();
        this.E = G.E();
        if (V < 0) throw new IllegalArgumentException("Number of vertices must be non-negative");

        // update adjacency lists
        adj = (Bag<Integer>[]) new Bag[V];
        for (int v = 0; v < V; v++) {
            adj[v] = new Bag<Integer>();
        }

        for (int v = 0; v < G.V(); v++) {
            // reverse so that adjacency list is in same order as original
            Stack<Integer> reverse = new Stack<Integer>();
            for (int w : G.adj[v]) {
                reverse.push(w);
            }
            for (int w : reverse) {
                adj[v].add(w);
            }
        }
    }

    public int V() {
        return V;
    }

    public int E() {
```

```
        return E;
    }

    // throw an IllegalArgumentException unless {@code 0 <= v < V}
    private void validateVertex(int v) {
        if (v < 0 || v >= V)
            throw new IllegalArgumentException("vertex " + v + " is not
between 0 and " + (V-1));
    }

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

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

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

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

    public static void main(String[] args) {
        In in = new In(args[0]);
        Graph G = new Graph(in);
        StdOut.println(G);
    }
}
```

adjacency-matrix

```
import java.util.Iterator;
import java.util.NoSuchElementException;


public class AdjMatrixGraph {
    private static final String NEWLINE =
System.getProperty("line.separator");

    private final int V;
    private int E;
    private boolean[][] adj;

    // empty graph with V vertices
    public AdjMatrixGraph(int V) {
        if (V < 0) throw new IllegalArgumentException("Too few vertices");
        this.V = V;
        this.E = 0;
        this.adj = new boolean[V][V];
    }

    // random graph with V vertices and E edges
    public AdjMatrixGraph(int V, int E) {
        this(V);
        if (E > (long) V*(V-1)/2 + V) throw new IllegalArgumentException("Too
many edges");
        if (E < 0)                    throw new IllegalArgumentException("Too
few edges");

        // can be inefficient
        while (this.E != E) {
            int v = StdRandom.uniformInt(V);
            int w = StdRandom.uniformInt(V);
            addEdge(v, w);
        }
    }

    // number of vertices and edges
    public int V() { return V; }
    public int E() { return E; }


    // add undirected edge v-w
    public void addEdge(int v, int w) {
        if (!adj[v][w]) E++;
        adj[v][w] = true;
        adj[w][v] = true;
    }

    // does the graph contain the edge v-w?
```

```
    public boolean contains(int v, int w) {
        return adj[v][w];
    }

    // return list of neighbors of v
    public Iterable<Integer> adj(int v) {
        return new AdjIterator(v);
    }

    // support iteration over graph vertices
    private class AdjIterator implements Iterator<Integer>, Iterable<Integer>
{
        private int v;
        private int w = 0;

        AdjIterator(int v) {
            this.v = v;
        }

        public Iterator<Integer> iterator() {
            return this;
        }

        public boolean hasNext() {
            while (w < V) {
                if (adj[v][w]) return true;
                w++;
            }
            return false;
        }

        public Integer next() {
            if (!hasNext()) {
                throw new NoSuchElementException();
            }
            return w++;
        }

        public void remove()  {
            throw new UnsupportedOperationException();
        }
    }

    // string representation of Graph - takes quadratic time
    public String toString() {
        StringBuilder s = new StringBuilder();
        s.append(V + " " + E + NEWLINE);
        for (int v = 0; v < V; v++) {
            s.append(v + ": ");
            for (int w : adj(v)) {
                s.append(w + " ");
            }
```

```
            s.append(NEWLINE);
        }
        return s.toString();
    }

    // test client
    public static void main(String[] args) {
        int V = Integer.parseInt(args[0]);
        int E = Integer.parseInt(args[1]);
        AdjMatrixGraph G = new AdjMatrixGraph(V, E);
        StdOut.println(G);
    }
}
```

Comparison

representation	space	add edge	edge between v and w?	iterate over vertices adjacent to v?
list of edges	E	1	E	E
adjacency matrix	V^2	1 *	1	V
adjacency lists	E + V	1	degree(v)	degree(v)

Depth-first Search

> recursive

Basic

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

Properties

- DFS marks all vertices connected to s in time proportional to the sum of their degrees
- After DFS, can find vertces connected to s in constant time and can find a path to s (if one exists) in time proportional to its length

Implementation

```
public class DepthFirstSearch {
    private boolean[] marked;    // marked[v] = is there an s-v path?
    private int count;           // number of vertices connected to s

    public DepthFirstSearch(Graph G, int s) {
        marked = new boolean[G.V()];
```

```
        validateVertex(s);
        dfs(G, s);
    }

    // depth first search from v
    private void dfs(Graph G, int v) {
        count++;
        marked[v] = true;
        for (int w : G.adj(v)) {
            if (!marked[w]) {
                dfs(G, w);
            }
        }
    }

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

    public int count() {
        return count;
    }

    // throw an IllegalArgumentException unless {@code 0 <= v < V}
    private void validateVertex(int v) {
        int V = marked.length;
        if (v < 0 || v >= V)
            throw new IllegalArgumentException("vertex " + v + " is not between
0 and " + (V-1));
    }

    public static void main(String[] args) {
        In in = new In(args[0]);
        Graph G = new Graph(in);
        int s = Integer.parseInt(args[1]);
        DepthFirstSearch search = new DepthFirstSearch(G, s);
        for (int v = 0; v < G.V(); v++) {
            if (search.marked(v))
                StdOut.print(v + " ");
        }

        StdOut.println();
        if (search.count() != G.V()) StdOut.println("NOT connected");
        else                         StdOut.println("connected");
    }

}
```

Breadth-firts Search

Properties

- compute shortest paths from s to all other vertices in a graph in time proportional to $E + V$
- Kevin Bacon graph (6 degree)

Implementation

```
public class BreadthFirstPaths {
    private static final int INFINITY = Integer.MAX_VALUE;
    private boolean[] marked;  // marked[v] = is there an s-v path
    private int[] edgeTo;      // edgeTo[v] = previous edge on shortest s-v path
    private int[] distTo;      // distTo[v] = number of edges shortest s-v path

    public BreadthFirstPaths(Graph G, int s) {
        marked = new boolean[G.V()];
        distTo = new int[G.V()];
        edgeTo = new int[G.V()];
        validateVertex(s);
        bfs(G, s);
        assert check(G, s);
    }

    public BreadthFirstPaths(Graph G, Iterable<Integer> sources) {
        marked = new boolean[G.V()];
        distTo = new int[G.V()];
        edgeTo = new int[G.V()];
        for (int v = 0; v < G.V(); v++)
            distTo[v] = INFINITY;
        validateVertices(sources);
        bfs(G, sources);
    }

    // breadth-first search from a single source
    private void bfs(Graph G, int s) {
        Queue<Integer> q = new Queue<Integer>();
        for (int v = 0; v < G.V(); v++)
            distTo[v] = INFINITY;
        distTo[s] = 0;
        marked[s] = true;
        q.enqueue(s);

        while (!q.isEmpty()) {
            int v = q.dequeue();
            for (int w : G.adj(v)) {
                if (!marked[w]) {
                    edgeTo[w] = v;
                    distTo[w] = distTo[v] + 1;
                    marked[w] = true;
                    q.enqueue(w);
```

```
                }
            }
        }
    }

    // breadth-first search from multiple sources
    private void bfs(Graph G, Iterable<Integer> sources) {
        Queue<Integer> q = new Queue<Integer>();
        for (int s : sources) {
            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]) {
                    edgeTo[w] = v;
                    distTo[w] = distTo[v] + 1;
                    marked[w] = true;
                    q.enqueue(w);
                }
            }
        }
    }

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

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

    public Iterable<Integer> pathTo(int v) {
        validateVertex(v);
        if (!hasPathTo(v)) return null;
        Stack<Integer> path = new Stack<Integer>();
        int x;
        for (x = v; distTo[x] != 0; x = edgeTo[x])
            path.push(x);
        path.push(x);
        return path;
    }


    // check optimality conditions for single source
    private boolean check(Graph G, int s) {

        // check that the distance of s = 0
```

```
        if (distTo[s] != 0) {
            StdOut.println("distance of source " + s + " to itself = " +
distTo[s]);
            return false;
        }

        // check that for each edge v-w dist[w] <= dist[v] + 1
        // provided v is reachable from s
        for (int v = 0; v < G.V(); v++) {
            for (int w : G.adj(v)) {
                if (hasPathTo(v) != hasPathTo(w)) {
                    StdOut.println("edge " + v + "-" + w);
                    StdOut.println("hasPathTo(" + v + ") = " + hasPathTo(v));
                    StdOut.println("hasPathTo(" + w + ") = " + hasPathTo(w));
                    return false;
                }
                if (hasPathTo(v) && (distTo[w] > distTo[v] + 1)) {
                    StdOut.println("edge " + v + "-" + w);
                    StdOut.println("distTo[" + v + "] = " + distTo[v]);
                    StdOut.println("distTo[" + w + "] = " + distTo[w]);
                    return false;
                }
            }
        }

        // check that v = edgeTo[w] satisfies distTo[w] = distTo[v] + 1
        // provided v is reachable from s
        for (int w = 0; w < G.V(); w++) {
            if (!hasPathTo(w) || w == s) continue;
            int v = edgeTo[w];
            if (distTo[w] != distTo[v] + 1) {
                StdOut.println("shortest path edge " + v + "-" + w);
                StdOut.println("distTo[" + v + "] = " + distTo[v]);
                StdOut.println("distTo[" + w + "] = " + distTo[w]);
                return false;
            }
        }
        return true;
    }

    // throw an IllegalArgumentException unless {@code 0 <= v < V}
    private void validateVertex(int v) {
        int V = marked.length;
        if (v < 0 || v >= V)
            throw new IllegalArgumentException("vertex " + v + " is not
between 0 and " + (V-1));
    }

    // throw an IllegalArgumentException if vertices is null, has zero
vertices,
    // or has a vertex not between 0 and V-1
    private void validateVertices(Iterable<Integer> vertices) {
```

```
        if (vertices == null) {
            throw new IllegalArgumentException("argument is null");
        }
        int vertexCount = 0;
        for (Integer v : vertices) {
            vertexCount++;
            if (v == null) {
                throw new IllegalArgumentException("vertex is null");
            }
            validateVertex(v);
        }
        if (vertexCount == 0) {
            throw new IllegalArgumentException("zero vertices");
        }
    }

    public static void main(String[] args) {
        In in = new In(args[0]);
        Graph G = new Graph(in);

        int s = Integer.parseInt(args[1]);
        BreadthFirstPaths bfs = new BreadthFirstPaths(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) StdOut.print(x);
                    else        StdOut.print("-" + x);
                }
                StdOut.println();
            }

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

Connected Components

> A maximal set of connected components

> Is v connected to w in constant time?

Properties

- equivalence equation
  - reflexive
  - symmetric
  - transitive

Implementation

```
public class CC {
    private boolean[] marked;   // marked[v] = has vertex v been marked?
    private int[] id;           // id[v] = id of connected component containing v
    private int[] size;         // size[id] = number of vertices in given component
    private int count;          // number of connected components

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

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

    // depth-first search for a Graph
    private void dfs(Graph G, int v) {
        marked[v] = true;
        id[v] = count;
        size[count]++;
        for (int w : G.adj(v)) {
            if (!marked[w]) {
                dfs(G, w);
            }
```

```
        }
    }

    // depth-first search for an EdgeWeightedGraph
    private void dfs(EdgeWeightedGraph G, int v) {
        marked[v] = true;
        id[v] = count;
        size[count]++;
        for (Edge e : G.adj(v)) {
            int w = e.other(v);
            if (!marked[w]) {
                dfs(G, w);
            }
        }
    }

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

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

    public int count() {
        return count;
    }

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

    @Deprecated
    public boolean areConnected(int v, int w) {
        validateVertex(v);
        validateVertex(w);
        return id(v) == id(w);
    }

    private void validateVertex(int v) {
        int V = marked.length;
        if (v < 0 || v >= V)
            throw new IllegalArgumentException("vertex " + v + " is not
between 0 and " + (V-1));
    }

    public static void main(String[] args) {
        In in = new In(args[0]);
```

```
        Graph G = new Graph(in);
        CC cc = new CC(G);

        // number of connected components
        int m = cc.count();
        StdOut.println(m + " components");

        // compute list of vertices in each connected component
        Queue<Integer>[] components = (Queue<Integer>[]) new Queue[m];
        for (int i = 0; i < m; i++) {
            components[i] = new Queue<Integer>();
        }
        for (int v = 0; v < G.V(); v++) {
            components[cc.id(v)].enqueue(v);
        }

        // print results
        for (int i = 0; i < m; i++) {
            for (int v : components[i]) {
                StdOut.print(v + " ");
            }
            StdOut.println();
        }
    }
}
```

Graph Challenges

- Path---Is there a path between?
- Shortest path---what is the shortest path between?

- Cycle---Is there a cycle between?
- Euler tour---Is there a cycle that uses each edge exactly once?
- Hamilton tour---Is there a cycle that uses each vertex exactly once?
    - Intractable

- Connectivity---Is there a away to connect all of the vertices?
- MST---What is the best way to connect all the vertices?
- Biconnectivity---Is there a vertex whose removal disconnects the graph?

- Planarity---Can you draw the graph in the plane with no crossing edges?
    - Hire an ex
- Graph isomorphism---Do two adjacency lists represent the same graph?

- No one knows