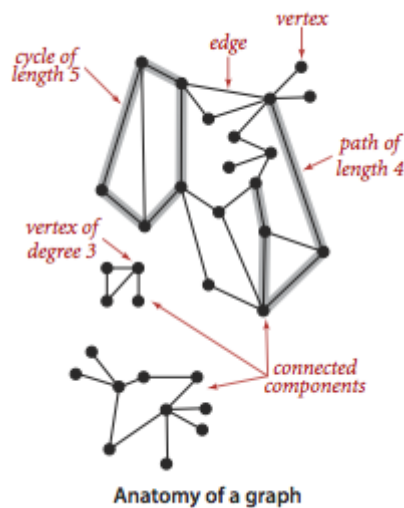


Undirected Graphs

Introduction

set of vertices connected pairwise by edges

Terminology



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
```

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

API for an undirected graph

```
1  /*****
2  *   13 13
3  *   0: 6 2 1 5
4  *   1: 0
5  *   2: 0
6  *   3: 5 4
7  *   4: 5 6 3
8  *   5: 3 4 0
9  *   6: 0 4
10 *   7: 8
11 *   8: 7
12 *   9: 11 10 12
13 *  10: 9
14 *  11: 9 12
15 *  12: 11 9
16 *
17 *   vertex of maximum degree = 4
18 *   average degree           = 2
19 *   number of self loops     = 0
20 *
21 *****/
22
23 public class GraphClient {
24
25     // maximum degree
26     public static int maxDegree(Graph G) {
27         int max = 0;
28         for (int v = 0; v < G.V(); v++)
29             if (G.degree(v) > max)
30                 max = G.degree(v);
31         return max;
32     }
33
34     // average degree
35     public static int avgDegree(Graph G) {
36         // each edge incident on two vertices
37         return 2 * G.E() / G.V();
38     }
39
40     // number of self-loops
```

```

41     public static int numberOfSelfLoops(Graph G) {
42         int count = 0;
43         for (int v = 0; v < G.V(); v++)
44             for (int w : G.adj(v))
45                 if (v == w) count++;
46         return count/2;    // self loop appears in adjacency list twice
47     }
48
49     public static void main(String[] args) {
50         In in = new In(args[0]);
51         Graph G = new Graph(in);
52         StdOut.println(G);
53
54         StdOut.println("vertex of maximum degree = " + maxDegree(G));
55         StdOut.println("average degree          = " + avgDegree(G));
56         StdOut.println("number of self loops    = " + numberOfSelfLoops(G));
57     }
58 }

```

Representation

adjacency-lists

widely used

```

1  import java.util.NoSuchElementException;
2
3  public class Graph {
4      private static final String NEWLINE =
5      System.getProperty("line.separator");
6
7      private final int V;
8      private int E;
9      private Bag<Integer>[] adj;
10
11     public Graph(int V) {
12         if (V < 0) throw new IllegalArgumentException("Number of vertices must
13         be non-negative");
14         this.V = V;
15         this.E = 0;
16         adj = (Bag<Integer>[]) new Bag[V];
17         for (int v = 0; v < V; v++) {
18             adj[v] = new Bag<Integer>();
19         }
20     }
21
22     public Graph(In in) {
23         if (in == null) throw new IllegalArgumentException("argument is
24         null");
25         try {
26             this.V = in.readInt();

```

```

24         if (V < 0) throw new IllegalArgumentException("number of vertices
must be non-negative");
25         adj = (Bag<Integer>[]) new Bag[V];
26         for (int v = 0; v < V; v++) {
27             adj[v] = new Bag<Integer>();
28         }
29         int E = in.readInt();
30         if (E < 0) throw new IllegalArgumentException("number of edges
must be non-negative");
31         for (int i = 0; i < E; i++) {
32             int v = in.readInt();
33             int w = in.readInt();
34             validateVertex(v);
35             validateVertex(w);
36             addEdge(v, w);
37         }
38     }
39     catch (NoSuchElementException e) {
40         throw new IllegalArgumentException("invalid input format in Graph
constructor", e);
41     }
42 }
43
44 public Graph(Graph G) {
45     this.V = G.V();
46     this.E = G.E();
47     if (V < 0) throw new IllegalArgumentException("Number of vertices must
be non-negative");
48
49     // update adjacency lists
50     adj = (Bag<Integer>[]) new Bag[V];
51     for (int v = 0; v < V; v++) {
52         adj[v] = new Bag<Integer>();
53     }
54
55     for (int v = 0; v < G.V(); v++) {
56         // reverse so that adjacency list is in same order as original
57         Stack<Integer> reverse = new Stack<Integer>();
58         for (int w : G.adj[v]) {
59             reverse.push(w);
60         }
61         for (int w : reverse) {
62             adj[v].add(w);
63         }
64     }
65 }
66
67 public int V() {
68     return V;
69 }
70
71 public int E() {

```

```

72         return E;
73     }
74
75     // throw an IllegalArgumentException unless {@code 0 <= v < V}
76     private void validateVertex(int v) {
77         if (v < 0 || v >= V)
78             throw new IllegalArgumentException("vertex " + v + " is not
between 0 and " + (V-1));
79     }
80
81     public void addEdge(int v, int w) {
82         validateVertex(v);
83         validateVertex(w);
84         E++;
85         adj[v].add(w);
86         adj[w].add(v);
87     }
88
89     public Iterable<Integer> adj(int v) {
90         validateVertex(v);
91         return adj[v];
92     }
93
94     public int degree(int v) {
95         validateVertex(v);
96         return adj[v].size();
97     }
98
99     public String toString() {
100         StringBuilder s = new StringBuilder();
101         s.append(V + " vertices, " + E + " edges " + NEWLINE);
102         for (int v = 0; v < V; v++) {
103             s.append(v + ": ");
104             for (int w : adj[v]) {
105                 s.append(w + " ");
106             }
107             s.append(NEWLINE);
108         }
109         return s.toString();
110     }
111
112     public static void main(String[] args) {
113         In in = new In(args[0]);
114         Graph G = new Graph(in);
115         StdOut.println(G);
116     }
117 }

```

adjacency-matrix

```
1  import java.util.Iterator;
2  import java.util.NoSuchElementException;
3
4
5  public class AdjMatrixGraph {
6      private static final String NEWLINE =
7          System.getProperty("line.separator");
8
9      private final int V;
10     private int E;
11     private boolean[][] adj;
12
13     // empty graph with V vertices
14     public AdjMatrixGraph(int V) {
15         if (V < 0) throw new IllegalArgumentException("Too few vertices");
16         this.V = V;
17         this.E = 0;
18         this.adj = new boolean[V][V];
19     }
20
21     // random graph with V vertices and E edges
22     public AdjMatrixGraph(int V, int E) {
23         this(V);
24         if (E > (long) V*(V-1)/2 + V) throw new IllegalArgumentException("Too
25 many edges");
26         if (E < 0) throw new IllegalArgumentException("Too
27 few edges");
28
29         // can be inefficient
30         while (this.E != E) {
31             int v = StdRandom.uniformInt(V);
32             int w = StdRandom.uniformInt(V);
33             addEdge(v, w);
34         }
35
36         // number of vertices and edges
37         public int V() { return V; }
38         public int E() { return E; }
39
40         // add undirected edge v-w
41         public void addEdge(int v, int w) {
42             if (!adj[v][w]) E++;
43             adj[v][w] = true;
44             adj[w][v] = true;
45         }
46
47         // does the graph contain the edge v-w?
```

```

47     public boolean contains(int v, int w) {
48         return adj[v][w];
49     }
50
51     // return list of neighbors of v
52     public Iterable<Integer> adj(int v) {
53         return new AdjIterator(v);
54     }
55
56     // support iteration over graph vertices
57     private class AdjIterator implements Iterator<Integer>, Iterable<Integer>
58     {
59         private int v;
60         private int w = 0;
61
62         AdjIterator(int v) {
63             this.v = v;
64         }
65
66         public Iterator<Integer> iterator() {
67             return this;
68         }
69
70         public boolean hasNext() {
71             while (w < V) {
72                 if (adj[v][w]) return true;
73                 w++;
74             }
75             return false;
76         }
77
78         public Integer next() {
79             if (!hasNext()) {
80                 throw new NoSuchElementException();
81             }
82             return w++;
83         }
84
85         public void remove() {
86             throw new UnsupportedOperationException();
87         }
88     }
89
90     // string representation of Graph - takes quadratic time
91     public String toString() {
92         StringBuilder s = new StringBuilder();
93         s.append(V + " " + E + NEWLINE);
94         for (int v = 0; v < V; v++) {
95             s.append(v + ": ");
96             for (int w : adj(v)) {
97                 s.append(w + " ");

```

```

98         s.append(NEWLINE);
99     }
100     return s.toString();
101 }
102
103 // test client
104 public static void main(String[] args) {
105     int V = Integer.parseInt(args[0]);
106     int E = Integer.parseInt(args[1]);
107     AdjMatrixGraph G = new AdjMatrixGraph(V, E);
108     Stdout.println(G);
109 }
110 }

```

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 vertex 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 vertices connected to s in constant time and can find a path to s (if one exists) in time proportional to its length

Implementation

```

1 public class DepthFirstSearch {
2     private boolean[] marked; // marked[v] = is there an s-v path?
3     private int count;       // number of vertices connected to s
4
5     public DepthFirstSearch(Graph G, int s) {
6         marked = new boolean[G.V()];

```



```

7         validateVertex(s);
8         dfs(G, s);
9     }
10
11     // depth first search from v
12     private void dfs(Graph G, int v) {
13         count++;
14         marked[v] = true;
15         for (int w : G.adj(v)) {
16             if (!marked[w]) {
17                 dfs(G, w);
18             }
19         }
20     }
21
22     public boolean marked(int v) {
23         validateVertex(v);
24         return marked[v];
25     }
26
27     public int count() {
28         return count;
29     }
30
31     // throw an IllegalArgumentException unless {0 <= v < V}
32     private void validateVertex(int v) {
33         int V = marked.length;
34         if (v < 0 || v >= V)
35             throw new IllegalArgumentException("vertex " + v + " is not between
0 and " + (V-1));
36     }
37
38     public static void main(String[] args) {
39         In in = new In(args[0]);
40         Graph G = new Graph(in);
41         int s = Integer.parseInt(args[1]);
42         DepthFirstSearch search = new DepthFirstSearch(G, s);
43         for (int v = 0; v < G.V(); v++) {
44             if (search.marked(v))
45                 StdOut.print(v + " ");
46         }
47
48         StdOut.println();
49         if (search.count() != G.V()) StdOut.println("NOT connected");
50         else StdOut.println("connected");
51     }
52
53 }

```

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

```
1 public class BreadthFirstPaths {
2     private static final int INFINITY = Integer.MAX_VALUE;
3     private boolean[] marked; // marked[v] = is there an s-v path
4     private int[] edgeTo;     // edgeTo[v] = previous edge on shortest s-v
    path
5     private int[] distTo;     // distTo[v] = number of edges shortest s-v
    path
6
7     public BreadthFirstPaths(Graph G, int s) {
8         marked = new boolean[G.V()];
9         distTo = new int[G.V()];
10        edgeTo = new int[G.V()];
11        validateVertex(s);
12        bfs(G, s);
13        assert check(G, s);
14    }
15
16    public BreadthFirstPaths(Graph G, Iterable<Integer> sources) {
17        marked = new boolean[G.V()];
18        distTo = new int[G.V()];
19        edgeTo = new int[G.V()];
20        for (int v = 0; v < G.V(); v++)
21            distTo[v] = INFINITY;
22        validateVertices(sources);
23        bfs(G, sources);
24    }
25
26    // breadth-first search from a single source
27    private void bfs(Graph G, int s) {
28        Queue<Integer> q = new Queue<Integer>();
29        for (int v = 0; v < G.V(); v++)
30            distTo[v] = INFINITY;
31        distTo[s] = 0;
32        marked[s] = true;
33        q.enqueue(s);
34
35        while (!q.isEmpty()) {
36            int v = q.dequeue();
37            for (int w : G.adj(v)) {
38                if (!marked[w]) {
39                    edgeTo[w] = v;
40                    distTo[w] = distTo[v] + 1;
41                    marked[w] = true;
42                    q.enqueue(w);
```

```

43         }
44     }
45 }
46
47
48 // breadth-first search from multiple sources
49 private void bfs(Graph G, Iterable<Integer> sources) {
50     Queue<Integer> q = new Queue<Integer>();
51     for (int s : sources) {
52         marked[s] = true;
53         distTo[s] = 0;
54         q.enqueue(s);
55     }
56     while (!q.isEmpty()) {
57         int v = q.dequeue();
58         for (int w : G.adj(v)) {
59             if (!marked[w]) {
60                 edgeTo[w] = v;
61                 distTo[w] = distTo[v] + 1;
62                 marked[w] = true;
63                 q.enqueue(w);
64             }
65         }
66     }
67 }
68
69 public boolean hasPathTo(int v) {
70     validateVertex(v);
71     return marked[v];
72 }
73
74 public int distTo(int v) {
75     validateVertex(v);
76     return distTo[v];
77 }
78
79 public Iterable<Integer> pathTo(int v) {
80     validateVertex(v);
81     if (!hasPathTo(v)) return null;
82     Stack<Integer> path = new Stack<Integer>();
83     int x;
84     for (x = v; distTo[x] != 0; x = edgeTo[x])
85         path.push(x);
86     path.push(x);
87     return path;
88 }
89
90
91 // check optimality conditions for single source
92 private boolean check(Graph G, int s) {
93
94     // check that the distance of s = 0

```

```

95         if (distTo[s] != 0) {
96             StdOut.println("distance of source " + s + " to itself = " +
distTo[s]);
97             return false;
98         }
99
100         // check that for each edge v-w dist[w] <= dist[v] + 1
101         // provided v is reachable from s
102         for (int v = 0; v < G.V(); v++) {
103             for (int w : G.adj(v)) {
104                 if (hasPathTo(v) != hasPathTo(w)) {
105                     StdOut.println("edge " + v + "-" + w);
106                     StdOut.println("hasPathTo(" + v + ") = " + hasPathTo(v));
107                     StdOut.println("hasPathTo(" + w + ") = " + hasPathTo(w));
108                     return false;
109                 }
110                 if (hasPathTo(v) && (distTo[w] > distTo[v] + 1)) {
111                     StdOut.println("edge " + v + "-" + w);
112                     StdOut.println("distTo[" + v + "] = " + distTo[v]);
113                     StdOut.println("distTo[" + w + "] = " + distTo[w]);
114                     return false;
115                 }
116             }
117         }
118
119         // check that v = edgeTo[w] satisfies distTo[w] = distTo[v] + 1
120         // provided v is reachable from s
121         for (int w = 0; w < G.V(); w++) {
122             if (!hasPathTo(w) || w == s) continue;
123             int v = edgeTo[w];
124             if (distTo[w] != distTo[v] + 1) {
125                 StdOut.println("shortest path edge " + v + "-" + w);
126                 StdOut.println("distTo[" + v + "] = " + distTo[v]);
127                 StdOut.println("distTo[" + w + "] = " + distTo[w]);
128                 return false;
129             }
130         }
131         return true;
132     }
133
134     // throw an IllegalArgumentException unless {@code 0 <= v < V}
135     private void validateVertex(int v) {
136         int V = marked.length;
137         if (v < 0 || v >= V)
138             throw new IllegalArgumentException("vertex " + v + " is not
between 0 and " + (V-1));
139     }
140
141     // throw an IllegalArgumentException if vertices is null, has zero
vertices,
142     // or has a vertex not between 0 and V-1
143     private void validateVertices(Iterable<Integer> vertices) {

```

```

144         if (vertices == null) {
145             throw new IllegalArgumentException("argument is null");
146         }
147         int vertexCount = 0;
148         for (Integer v : vertices) {
149             vertexCount++;
150             if (v == null) {
151                 throw new IllegalArgumentException("vertex is null");
152             }
153             validateVertex(v);
154         }
155         if (vertexCount == 0) {
156             throw new IllegalArgumentException("zero vertices");
157         }
158     }
159
160     public static void main(String[] args) {
161         In in = new In(args[0]);
162         Graph G = new Graph(in);
163
164         int s = Integer.parseInt(args[1]);
165         BreadthFirstPaths bfs = new BreadthFirstPaths(G, s);
166
167         for (int v = 0; v < G.V(); v++) {
168             if (bfs.hasPathTo(v)) {
169                 StdOut.printf("%d to %d (%d): ", s, v, bfs.distTo(v));
170                 for (int x : bfs.pathTo(v)) {
171                     if (x == s) StdOut.print(x);
172                     else StdOut.print("-" + x);
173                 }
174                 StdOut.println();
175             }
176
177             else {
178                 StdOut.printf("%d to %d (-): not connected\n", s, v);
179             }
180         }
181     }
182 }

```

Connected Components

A maximal set of connected components

Is v connected to w in constant time?

Properties

- equivalence equation
 - reflexive
 - symmetric
 - transitive

Implementation

```
1 public class CC {
2     private boolean[] marked; // marked[v] = has vertex v been marked?
3     private int[] id; // id[v] = id of connected component
4     containing v
5     private int[] size; // size[id] = number of vertices in given
6     component
7     private int count; // number of connected components
8
9     public CC(Graph G) {
10         marked = new boolean[G.V()];
11         id = new int[G.V()];
12         size = new int[G.V()];
13         for (int v = 0; v < G.V(); v++) {
14             if (!marked[v]) {
15                 dfs(G, v);
16                 count++;
17             }
18         }
19
20     public CC(EdgeweightedGraph G) {
21         marked = new boolean[G.V()];
22         id = new int[G.V()];
23         size = new int[G.V()];
24         for (int v = 0; v < G.V(); v++) {
25             if (!marked[v]) {
26                 dfs(G, v);
27                 count++;
28             }
29         }
30
31     // depth-first search for a Graph
32     private void dfs(Graph G, int v) {
33         marked[v] = true;
34         id[v] = count;
35         size[count]++;
36         for (int w : G.adj(v)) {
37             if (!marked[w]) {
38                 dfs(G, w);
39             }
40         }
41     }
42 }
```

```

40     }
41 }
42
43 // depth-first search for an EdgeweightedGraph
44 private void dfs(EdgeweightedGraph G, int v) {
45     marked[v] = true;
46     id[v] = count;
47     size[count]++;
48     for (Edge e : G.adj(v)) {
49         int w = e.other(v);
50         if (!marked[w]) {
51             dfs(G, w);
52         }
53     }
54 }
55
56 public int id(int v) {
57     validateVertex(v);
58     return id[v];
59 }
60
61 public int size(int v) {
62     validateVertex(v);
63     return size[id[v]];
64 }
65
66 public int count() {
67     return count;
68 }
69
70 public boolean connected(int v, int w) {
71     validateVertex(v);
72     validateVertex(w);
73     return id(v) == id(w);
74 }
75
76 @Deprecated
77 public boolean areConnected(int v, int w) {
78     validateVertex(v);
79     validateVertex(w);
80     return id(v) == id(w);
81 }
82
83 private void validateVertex(int v) {
84     int V = marked.length;
85     if (v < 0 || v >= V)
86         throw new IllegalArgumentException("vertex " + v + " is not
87 between 0 and " + (V-1));
88 }
89
90 public static void main(String[] args) {
91     In in = new In(args[0]);

```

```

91     Graph G = new Graph(in);
92     CC cc = new CC(G);
93
94     // number of connected components
95     int m = cc.count();
96     StdOut.println(m + " components");
97
98     // compute list of vertices in each connected component
99     Queue<Integer>[] components = (Queue<Integer>[]) new Queue[m];
100    for (int i = 0; i < m; i++) {
101        components[i] = new Queue<Integer>();
102    }
103    for (int v = 0; v < G.V(); v++) {
104        components[cc.id(v)].enqueue(v);
105    }
106
107    // print results
108    for (int i = 0; i < m; i++) {
109        for (int v : components[i]) {
110            StdOut.print(v + " ");
111        }
112        StdOut.println();
113    }
114 }
115 }

```

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 way 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