

Directed Graphs

set of vertices connected pairwise by directed edges

Introduction

Application

digraph	vertex	directed edge
transportation	street intersection	one-way street
web	web page	hyperlink
food web	species	predator-prey relationship
WordNet	synset	hypernym
scheduling	task	precedence constraint
financial	bank	transaction
cell phone	person	placed call
infectious disease	person	infection
game	board position	legal move
citation	journal article	citation
object graph	object	pointer
inheritance hierarchy	class	inherits from
control flow	code block	jump

Problems

- Path---Is there a path from s to t ?
- Shortest path---what is the shortest path from s to t ?
- Topological sort---Can you draw a digraph so that all edges point upwards?
- Strong connectivity---Is there a directed path between all pairs of vertices?
- Transitive closure---For which vertices v and w is there a path from v to w ?
- PageRank---What is the importance of webpages?

Digraph API

Digraph API

```
public class Digraph
{
    Digraph(int V)           create a V-vertex digraph with no edges
    Digraph(In in)           read a digraph 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 digraph
    Iterable<Integer> adj(int v) vertices connected to v by edges pointing from v
    Digraph reverse()        reverse of this digraph
    String toString()        string representation
}
```

Representation

adjacency-lists

```
1  import java.util.NoSuchElementException;
2
3  public class Digraph {
4      private static final String NEWLINE =
5      System.getProperty("line.separator");
6
7      private final int V;          // number of vertices in this digraph
8      private int E;               // number of edges in this digraph
9      private Bag<Integer>[] adj;   // adj[v] = adjacency list for vertex v
10     private int[] indegree;       // indegree[v] = indegree of vertex v
11
12     public Digraph(int V) {
13         if (V < 0) throw new IllegalArgumentException("Number of vertices in a
14         Digraph must be non-negative");
15         this.V = V;
16         this.E = 0;
17         indegree = new int[V];
18         adj = (Bag<Integer>[]) new Bag[V];
19         for (int v = 0; v < V; v++) {
20             adj[v] = new Bag<Integer>();
21         }
22     }
23
24     public Digraph(In in) {
25         if (in == null) throw new IllegalArgumentException("argument is
26         null");
27         try {
28             this.V = in.readInt();
29             if (V < 0) throw new IllegalArgumentException("number of vertices
30             in a Digraph must be non-negative");
31             indegree = new int[V];
32             adj = (Bag<Integer>[]) new Bag[V];
33             for (int v = 0; v < V; v++) {
34                 adj[v] = new Bag<Integer>();
35             }
36             int E = in.readInt();
37             if (E < 0) throw new IllegalArgumentException("number of edges in
38             a Digraph must be non-negative");
39             for (int i = 0; i < E; i++) {
40                 int v = in.readInt();
41                 int w = in.readInt();
42                 addEdge(v, w);
43             }
44         }
45         catch (NoSuchElementException e) {
46             throw new IllegalArgumentException("invalid input format in
47             Digraph constructor", e);
48         }
49     }
50 }
```

```

42     }
43 }
44
45 public Digraph(Digraph G) {
46     if (G == null) throw new IllegalArgumentException("argument is null");
47
48     this.V = G.V();
49     this.E = G.E();
50     if (V < 0) throw new IllegalArgumentException("Number of vertices in a
Digraph must be non-negative");
51
52     // update indegrees
53     indegree = new int[V];
54     for (int v = 0; v < V; v++)
55         this.indegree[v] = G.indegree(v);
56
57     // update adjacency lists
58     adj = (Bag<Integer>[]) new Bag[V];
59     for (int v = 0; v < V; v++) {
60         adj[v] = new Bag<Integer>();
61     }
62
63     for (int v = 0; v < G.V(); v++) {
64         // reverse so that adjacency list is in same order as original
65         Stack<Integer> reverse = new Stack<Integer>();
66         for (int w : G.adj[v]) {
67             reverse.push(w);
68         }
69         for (int w : reverse) {
70             adj[v].add(w);
71         }
72     }
73 }
74
75 public int V() {
76     return V;
77 }
78
79 public int E() {
80     return E;
81 }
82
83
84 // throw an IllegalArgumentException unless {0 <= v < V}
85 private void validateVertex(int v) {
86     if (v < 0 || v >= V)
87         throw new IllegalArgumentException("vertex " + v + " is not
between 0 and " + (V-1));
88 }
89
90 public void addEdge(int v, int w) {
91     validateVertex(v);

```

```

92     validateVertex(w);
93     adj[v].add(w);
94     indegree[w]++;
95     E++;
96 }
97
98 public Iterable<Integer> adj(int v) {
99     validateVertex(v);
100     return adj[v];
101 }
102
103 public int outdegree(int v) {
104     validateVertex(v);
105     return adj[v].size();
106 }
107
108 public int indegree(int v) {
109     validateVertex(v);
110     return indegree[v];
111 }
112
113 public Digraph reverse() {
114     Digraph reverse = new Digraph(V);
115     for (int v = 0; v < V; v++) {
116         for (int w : adj(v)) {
117             reverse.addEdge(w, v);
118         }
119     }
120     return reverse;
121 }
122
123 public String toString() {
124     StringBuilder s = new StringBuilder();
125     s.append(V + " vertices, " + E + " edges " + NEWLINE);
126     for (int v = 0; v < V; v++) {
127         s.append(String.format("%d: ", v));
128         for (int w : adj[v]) {
129             s.append(String.format("%d ", w));
130         }
131         s.append(NEWLINE);
132     }
133     return s.toString();
134 }
135
136 public static void main(String[] args) {
137     In in = new In(args[0]);
138     Digraph G = new Digraph(in);
139     StdOut.println(G);
140 }
141 }

```

adjacency-matrix

```
1  import java.util.Iterator;
2  import java.util.NoSuchElementException;
3
4
5  public class AdjMatrixDigraph {
6      private int V;
7      private int E;
8      private boolean[][] adj;
9
10     // empty graph with v vertices
11     public AdjMatrixDigraph(int V) {
12         if (V < 0) throw new RuntimeException("Number of vertices must be non-
negative");
13         this.V = V;
14         this.E = 0;
15         this.adj = new boolean[V][V];
16     }
17
18     // random graph with v vertices and E edges
19     public AdjMatrixDigraph(int V, int E) {
20         this(V);
21         if (E < 0) throw new RuntimeException("Number of edges must be non-
negative");
22         if (E > V*V) throw new RuntimeException("Too many edges");
23
24         // can be inefficient
25         while (this.E != E) {
26             int v = StdRandom.uniformInt(V);
27             int w = StdRandom.uniformInt(V);
28             addEdge(v, w);
29         }
30     }
31
32     // number of vertices and edges
33     public int V() { return V; }
34     public int E() { return E; }
35
36
37     // add directed edge v->w
38     public void addEdge(int v, int w) {
39         if (!adj[v][w]) E++;
40         adj[v][w] = true;
41     }
42
43     // return list of neighbors of v
44     public Iterable<Integer> adj(int v) {
45         return new AdjIterator(v);
46     }
47 }
```

```

48 // support iteration over graph vertices
49 private class AdjIterator implements Iterator<Integer>, Iterable<Integer>
{
50     private int v;
51     private int w = 0;
52
53     AdjIterator(int v) {
54         this.v = v;
55     }
56
57     public Iterator<Integer> iterator() {
58         return this;
59     }
60
61     public boolean hasNext() {
62         while (w < V) {
63             if (adj[v][w]) return true;
64             w++;
65         }
66         return false;
67     }
68
69     public Integer next() {
70         if (hasNext()) return w++;
71         else throw new NoSuchElementException();
72     }
73
74     public void remove() {
75         throw new UnsupportedOperationException();
76     }
77 }
78
79 // string representation of Graph - takes quadratic time
80 public String toString() {
81     String NEWLINE = System.getProperty("line.separator");
82     StringBuilder s = new StringBuilder();
83     s.append(V + " " + E + NEWLINE);
84     for (int v = 0; v < V; v++) {
85         s.append(v + ": ");
86         for (int w : adj(v)) {
87             s.append(w + " ");
88         }
89         s.append(NEWLINE);
90     }
91     return s.toString();
92 }
93
94 // test client
95 public static void main(String[] args) {
96     int V = Integer.parseInt(args[0]);
97     int E = Integer.parseInt(args[1]);
98     AdjMatrixDigraph G = new AdjMatrixDigraph(V, E);

```

```

99         StdOut.println(G);
100     }
101
102 }
```

Digraph Search

DFS

DFS (to visit a vertex v)

Mark v as visited.

Recursively visit all unmarked
vertices w pointing from v .

Application

- program detection
- mark-sweeo garbage detection

BFS

$E + V$

BFS (from source vertex s)

Put s onto a FIFO queue, and mark s as visited.

Repeat until the queue is empty:

- remove the least recently added vertex v
- for each unmarked vertex pointing from v :
add to queue and mark as visited.

Application

- Multi-source Shortest Path
- Web Crawler

Topological Sort

draw DAG so all edges point upwards

- DAG---directed acyclic graph

DFS

Basic

- run DFS
- return vertices in reverse order

Properties

- If directed cycle, topological order impossible
- If no directed cycle, DFS-based algorithm finds a topological order

Application

- java compiler---cyclic inheritance
- microsoft excel---spreadsheet recalculation

Implementation

```
1 public class DepthFirstOrder {
2     private boolean[] marked;           // marked[v] = has v been marked in
    dfs?
3     private int[] pre;                  // pre[v]    = preorder  number of v
4     private int[] post;                 // post[v]   = postorder number of v
5     private Queue<Integer> preorder;    // vertices in preorder
6     private Queue<Integer> postorder;   // vertices in postorder
7     private int preCounter;             // counter for preorder numbering
8     private int postCounter;            // counter for postorder numbering
9
10    public DepthFirstOrder(Digraph G) {
11        pre    = new int[G.V()];
12        post   = new int[G.V()];
13        postorder = new Queue<Integer>();
14        preorder  = new Queue<Integer>();
15        marked    = new boolean[G.V()];
16        for (int v = 0; v < G.V(); v++)
17            if (!marked[v]) dfs(G, v);
18
19        assert check();
20    }
21
22    public DepthFirstOrder(EdgeWeightedDigraph G) {
23        pre    = new int[G.V()];
24        post   = new int[G.V()];
25        postorder = new Queue<Integer>();
26        preorder  = new Queue<Integer>();
27        marked    = new boolean[G.V()];
28        for (int v = 0; v < G.V(); v++)
29            if (!marked[v]) dfs(G, v);
30    }
31 }
```



```

32 // run DFS in digraph G from vertex v and compute preorder/postorder
33 private void dfs(Digraph G, int v) {
34     marked[v] = true;
35     pre[v] = preCounter++;
36     preorder.enqueue(v);
37     for (int w : G.adj(v)) {
38         if (!marked[w]) {
39             dfs(G, w);
40         }
41     }
42     postorder.enqueue(v);
43     post[v] = postCounter++;
44 }
45
46 // run DFS in edge-weighted digraph G from vertex v and compute
preorder/postorder
47 private void dfs(EdgeweightedDigraph G, int v) {
48     marked[v] = true;
49     pre[v] = preCounter++;
50     preorder.enqueue(v);
51     for (DirectedEdge e : G.adj(v)) {
52         int w = e.to();
53         if (!marked[w]) {
54             dfs(G, w);
55         }
56     }
57     postorder.enqueue(v);
58     post[v] = postCounter++;
59 }
60
61 public int pre(int v) {
62     validateVertex(v);
63     return pre[v];
64 }
65
66 public int post(int v) {
67     validateVertex(v);
68     return post[v];
69 }
70
71 public Iterable<Integer> post() {
72     return postorder;
73 }
74
75 public Iterable<Integer> pre() {
76     return preorder;
77 }
78
79 public Iterable<Integer> reversePost() {
80     Stack<Integer> reverse = new Stack<Integer>();
81     for (int v : postorder)
82         reverse.push(v);

```

```

83         return reverse;
84     }
85
86
87     // check that pre() and post() are consistent with pre(v) and post(v)
88     private boolean check() {
89
90         // check that post(v) is consistent with post()
91         int r = 0;
92         for (int v : post()) {
93             if (post(v) != r) {
94                 StdOut.println("post(v) and post() inconsistent");
95                 return false;
96             }
97             r++;
98         }
99
100        // check that pre(v) is consistent with pre()
101        r = 0;
102        for (int v : pre()) {
103            if (pre(v) != r) {
104                StdOut.println("pre(v) and pre() inconsistent");
105                return false;
106            }
107            r++;
108        }
109
110        return true;
111    }
112
113    // throw an IllegalArgumentException unless {0 <= v < V}
114    private void validateVertex(int v) {
115        int V = marked.length;
116        if (v < 0 || v >= V)
117            throw new IllegalArgumentException("vertex " + v + " is not
between 0 and " + (V-1));
118    }
119
120    public static void main(String[] args) {
121        In in = new In(args[0]);
122        Digraph G = new Digraph(in);
123
124        DepthFirstOrder dfs = new DepthFirstOrder(G);
125        StdOut.println("    v  pre post");
126        StdOut.println("-----");
127        for (int v = 0; v < G.V(); v++) {
128            StdOut.printf("%4d %4d %4d\n", v, dfs.pre(v), dfs.post(v));
129        }
130
131        StdOut.print("Preorder: ");
132        for (int v : dfs.pre()) {
133            StdOut.print(v + " ");

```

```

134     }
135     stdout.println();
136
137     stdout.print("Postorder: ");
138     for (int v : dfs.post()) {
139         stdout.print(v + " ");
140     }
141     stdout.println();
142
143     stdout.print("Reverse postorder: ");
144     for (int v : dfs.reversePost()) {
145         stdout.print(v + " ");
146     }
147     stdout.println();
148 }
149 }

```

Strong Connected Components

If there is a directed path from v to w and a directed path from w to v

A strong component is a maximal subset of strongly-connected vertices

Properties

- equivalence relation

Applications

- ecological cycle

Kosaraju-Sharir Algorithm

- Reverse graph---strong components in G are the same as in G^R
- Kernel DAG---Contract each strong component into a single vertex
- Idea
 - compute topological order (reverse postorder) in kernel DAG
 - Run DFS, considering vertices in reverse topological order
- Phase
 - Compute reverse postorder in G^R
 - Run DFS in G , visiting unmarked vertices in reverse postorder of G^R

Implementation

```

1 public class KosarajuSharirSCC {
2     private boolean[] marked; // marked[v] = has vertex v been visited?
3     private int[] id; // id[v] = id of strong component containing
    v

```

```

4     private int count;                // number of strongly-connected components
5
6     public KosarajuSharirSCC(Digraph G) {
7
8         // compute reverse postorder of reverse graph
9         DepthFirstOrder dfs = new DepthFirstOrder(G.reverse());
10
11        // run DFS on G, using reverse postorder to guide calculation
12        marked = new boolean[G.V()];
13        id = new int[G.V()];
14        for (int v : dfs.reversePost()) {
15            if (!marked[v]) {
16                dfs(G, v);
17                count++;
18            }
19        }
20
21        // check that id[] gives strong components
22        assert check(G);
23    }
24
25    // DFS on graph G
26    private void dfs(Digraph G, int v) {
27        marked[v] = true;
28        id[v] = count;
29        for (int w : G.adj(v)) {
30            if (!marked[w]) dfs(G, w);
31        }
32    }
33
34    public int count() {
35        return count;
36    }
37
38    public boolean stronglyConnected(int v, int w) {
39        validateVertex(v);
40        validateVertex(w);
41        return id[v] == id[w];
42    }
43
44    public int id(int v) {
45        validateVertex(v);
46        return id[v];
47    }
48
49    // does the id[] array contain the strongly connected components?
50    private boolean check(Digraph G) {
51        TransitiveClosure tc = new TransitiveClosure(G);
52        for (int v = 0; v < G.V(); v++) {
53            for (int w = 0; w < G.V(); w++) {
54                if (stronglyConnected(v, w) != (tc.reachable(v, w) &&
tc.reachable(w, v)))

```

```

55         return false;
56     }
57 }
58     return true;
59 }
60
61 // throw an IllegalArgumentException unless {@code 0 <= v < V}
62 private void validateVertex(int v) {
63     int V = marked.length;
64     if (v < 0 || v >= V)
65         throw new IllegalArgumentException("vertex " + v + " is not between
0 and " + (V-1));
66 }
67
68 public static void main(String[] args) {
69     In in = new In(args[0]);
70     Digraph G = new Digraph(in);
71     KosarajuSharirSCC scc = new KosarajuSharirSCC(G);
72
73     // number of connected components
74     int m = scc.count();
75     StdOut.println(m + " strong components");
76
77     // compute list of vertices in each strong component
78     Queue<Integer>[] components = (Queue<Integer>[]) new Queue[m];
79     for (int i = 0; i < m; i++) {
80         components[i] = new Queue<Integer>();
81     }
82     for (int v = 0; v < G.V(); v++) {
83         components[scc.id(v)].enqueue(v);
84     }
85
86     // print results
87     for (int i = 0; i < m; i++) {
88         for (int v : components[i]) {
89             StdOut.print(v + " ");
90         }
91         StdOut.println();
92     }
93 }
94 }

```