Directed Graphs

> set if 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

```
import java.util.NoSuchElementException;

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

    private final int V;           // number of vertices in this digraph
    private int E;                 // number of edges in this digraph
    private Bag<Integer>[] adj;    // adj[v] = adjacency list for vertex v
    private int[] indegree;        // indegree[v] = indegree of vertex v

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

    public Digraph(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
in a Digraph must be non-negative");
            indegree = new int[V];
            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 in
a Digraph must be non-negative");
            for (int i = 0; i < E; i++) {
                int v = in.readInt();
                int w = in.readInt();
                addEdge(v, w);
            }
        }
        catch (NoSuchElementException e) {
            throw new IllegalArgumentException("invalid input format in
Digraph constructor", e);
```

```
        }
    }

    public Digraph(Digraph G) {
        if (G == null) throw new IllegalArgumentException("argument is null");

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

        // update indegrees
        indegree = new int[V];
        for (int v = 0; v < V; v++)
            this.indegree[v] = G.indegree(v);

        // 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);
        adj[v].add(w);
        indegree[w]++;
        E++;
    }

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

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

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

    public Digraph reverse() {
        Digraph reverse = new Digraph(V);
        for (int v = 0; v < V; v++) {
            for (int w : adj(v)) {
                reverse.addEdge(w, v);
            }
        }
        return reverse;
    }

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

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

adjacency-matrix

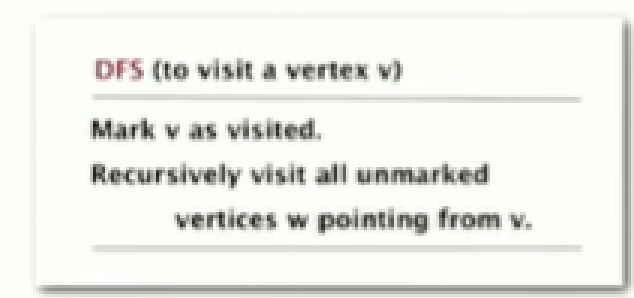
```
import java.util.Iterator;
import java.util.NoSuchElementException;


public class AdjMatrixDigraph {
    private int V;
    private int E;
    private boolean[][] adj;

    // empty graph with V vertices
    public AdjMatrixDigraph(int V) {
        if (V < 0) throw new RuntimeException("Number of vertices must be non-
negative");
        this.V = V;
        this.E = 0;
        this.adj = new boolean[V][V];
    }

    // random graph with V vertices and E edges
    public AdjMatrixDigraph(int V, int E) {
        this(V);
        if (E < 0) throw new RuntimeException("Number of edges must be non-
negative");
        if (E > V*V) throw new RuntimeException("Too many 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 directed edge v->w
    public void addEdge(int v, int w) {
        if (!adj[v][w]) E++;
        adj[v][w] = true;
    }

    // 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()) return w++;
            else           throw new NoSuchElementException();
        }

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

    // string representation of Graph - takes quadratic time
    public String toString() {
        String NEWLINE = System.getProperty("line.separator");
        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]);
        AdjMatrixDigraph G = new AdjMatrixDigraph(V, E);
```

```
        StdOut.println(G);
    }

}
```

Digraph Search

DFS

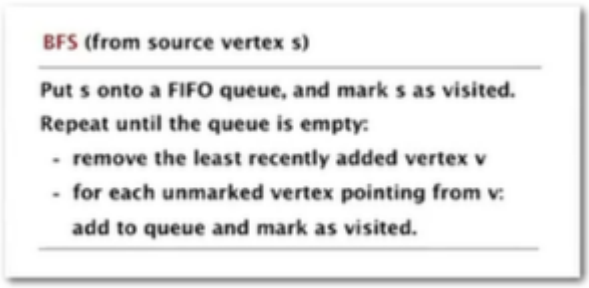


Application

- program detection
- mark-sweeo garbage detection

BFS

> $E+V$



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

Application

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

Implementation

```
public class DepthFirstOrder {
    private boolean[] marked;          // marked[v] = has v been marked in
dfs?
    private int[] pre;                 // pre[v]    = preorder  number of v
    private int[] post;                // post[v]   = postorder number of v
    private Queue<Integer> preorder;   // vertices in preorder
    private Queue<Integer> postorder;  // vertices in postorder
    private int preCounter;            // counter or preorder numbering
    private int postCounter;           // counter for postorder numbering

    public DepthFirstOrder(Digraph G) {
        pre    = new int[G.V()];
        post   = new int[G.V()];
        postorder = new Queue<Integer>();
        preorder  = new Queue<Integer>();
        marked    = new boolean[G.V()];
        for (int v = 0; v < G.V(); v++)
            if (!marked[v]) dfs(G, v);

        assert check();
    }

    public DepthFirstOrder(EdgeWeightedDigraph G) {
        pre    = new int[G.V()];
        post   = new int[G.V()];
        postorder = new Queue<Integer>();
        preorder  = new Queue<Integer>();
        marked    = new boolean[G.V()];
        for (int v = 0; v < G.V(); v++)
            if (!marked[v]) dfs(G, v);
    }

```

```
    // run DFS in digraph G from vertex v and compute preorder/postorder
    private void dfs(Digraph G, int v) {
        marked[v] = true;
        pre[v] = preCounter++;
        preorder.enqueue(v);
        for (int w : G.adj(v)) {
            if (!marked[w]) {
                dfs(G, w);
            }
        }
        postorder.enqueue(v);
        post[v] = postCounter++;
    }

    // run DFS in edge-weighted digraph G from vertex v and compute
preorder/postorder
    private void dfs(EdgeWeightedDigraph G, int v) {
        marked[v] = true;
        pre[v] = preCounter++;
        preorder.enqueue(v);
        for (DirectedEdge e : G.adj(v)) {
            int w = e.to();
            if (!marked[w]) {
                dfs(G, w);
            }
        }
        postorder.enqueue(v);
        post[v] = postCounter++;
    }

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

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

    public Iterable<Integer> post() {
        return postorder;
    }

    public Iterable<Integer> pre() {
        return preorder;
    }

    public Iterable<Integer> reversePost() {
        Stack<Integer> reverse = new Stack<Integer>();
        for (int v : postorder)
            reverse.push(v);
```

```
        return reverse;
    }


    // check that pre() and post() are consistent with pre(v) and post(v)
    private boolean check() {

        // check that post(v) is consistent with post()
        int r = 0;
        for (int v : post()) {
            if (post(v) != r) {
                StdOut.println("post(v) and post() inconsistent");
                return false;
            }
            r++;
        }

        // check that pre(v) is consistent with pre()
        r = 0;
        for (int v : pre()) {
            if (pre(v) != r) {
                StdOut.println("pre(v) and pre() inconsistent");
                return false;
            }
            r++;
        }

        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));
    }

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

        DepthFirstOrder dfs = new DepthFirstOrder(G);
        StdOut.println("   v  pre post");
        StdOut.println("--------------");
        for (int v = 0; v < G.V(); v++) {
            StdOut.printf("%4d %4d %4d\n", v, dfs.pre(v), dfs.post(v));
        }

        StdOut.print("Preorder:  ");
        for (int v : dfs.pre()) {
            StdOut.print(v + " ");
```

```
        }
        StdOut.println();

        StdOut.print("Postorder: ");
        for (int v : dfs.post()) {
            StdOut.print(v + " ");
        }
        StdOut.println();

        StdOut.print("Reverse postorder: ");
        for (int v : dfs.reversePost()) {
            StdOut.print(v + " ");
        }
        StdOut.println();
    }
}
```

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

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

```
    private int count;            // number of strongly-connected components

    public KosarajuSharirSCC(Digraph G) {

        // compute reverse postorder of reverse graph
        DepthFirstOrder dfs = new DepthFirstOrder(G.reverse());

        // run DFS on G, using reverse postorder to guide calculation
        marked = new boolean[G.V()];
        id = new int[G.V()];
        for (int v : dfs.reversePost()) {
            if (!marked[v]) {
                dfs(G, v);
                count++;
            }
        }

        // check that id[] gives strong components
        assert check(G);
    }

    // DFS on graph G
    private void dfs(Digraph G, int v) {
        marked[v] = true;
        id[v] = count;
        for (int w : G.adj(v)) {
            if (!marked[w]) dfs(G, w);
        }
    }

    public int count() {
        return count;
    }

    public boolean stronglyConnected(int v, int w) {
        validateVertex(v);
        validateVertex(w);
        return id[v] == id[w];
    }

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

    // does the id[] array contain the strongly connected components?
    private boolean check(Digraph G) {
        TransitiveClosure tc = new TransitiveClosure(G);
        for (int v = 0; v < G.V(); v++) {
            for (int w = 0; w < G.V(); w++) {
                if (stronglyConnected(v, w) != (tc.reachable(v, w) &&
tc.reachable(w, v)))
```

```
                    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));
        }

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

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

            // compute list of vertices in each strong 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[scc.id(v)].enqueue(v);
            }

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