```java
//******************************************************************************
//
// File:    UndirectedGraph.java
// Package: ---
// Unit:    Class UndirectedGraph
//
//******************************************************************************

import java.util.ArrayList;
import java.util.LinkedList;
import edu.rit.pj2.vbl.DoubleVbl;
import edu.rit.util.Random;
import edu.rit.util.Searching;

/**
 * Class UndirectedGraph represents an undirected graph meaning that if
 * there exists an edge connecting some vertex A to some vertex B, then
 * that same edge connects vertex B to vertex A.
 *
 * @author Jimi Ford
 * @version 2-15-2015
 */
public class UndirectedGraph {

    // private data members
    private ArrayList<UndirectedEdge> edges;
    public ArrayList<Cricket> vertices;
    private int v;


    /**
     * Private constructor used internally by the static random graph
     * method
     * @param v the number of vertices in the graph
     */
    private UndirectedGraph(int v, CricketObserver o) {
        this.v = v;
        vertices = new ArrayList<Cricket>(v);
        edges = new ArrayList<UndirectedEdge>();
        for(int i = 0; i < v; i++) {
            vertices.add(new Cricket(i,o));
        }
    }

    /**
     * Perform a BFS to get the distance from one vertex to another
     *
     * @param start the id of the start vertex
     * @param goal the id of the goal vertex
     * @return the minimum distance between the two vertices
     */
    private int BFS(int start, int goal) {
        return BFS(vertices.get(start), vertices.get(goal));
    }

    /**
     * Perform a BFS to get the distance from one vertex to another
     *
     * @param start the reference to the start vertex
     * @param goal the reference to the goal vertex
     * @return the minimum distance between the two vertices
     */
    private int BFS(Cricket start, Cricket goal) {
        int distance = 0, verticesToProcess = 1, uniqueNeighbors = 0;
        LinkedList<Cricket> queue = new LinkedList<Cricket>();
        boolean[] visited = new boolean[v];
        visited[start.n] = true;
        Cricket current, t2;
        queue.add(start);
        while(!queue.isEmpty()) {
            current = queue.removeFirst();
            if(current.equals(goal)) {
                return distance;
            }
            for(int i = 0; i < current.degree(); i++) {
                t2 = current.getEdges().get(i).other(current);
                if(!visited[t2.n]) {
                    visited[t2.n] = true;
                    queue.add(t2);
                    uniqueNeighbors++;
                }
            }
            verticesToProcess--;
            if(verticesToProcess <= 0) {
                verticesToProcess = uniqueNeighbors;
                uniqueNeighbors = 0;
                distance++;
            }

        }
        return 0;
    }

    /**
     * Accumulate the distances of each pair of vertices into
     * a "running total" to be averaged
     *
     * @param thrLocal the reference to the "running total"
     * Prof. Alan Kaminsky's library handles averaging this
     * accumulated value.
     */
    public void accumulateDistances(DoubleVbl.Mean thrLocal) {
        for(int i = 0; i < v; i++) {
            for(int j = i + 1; j < v; j++) {
                int distance = BFS(i, j);
                // only accumulate the distance if the two vertices
                // are actually connected
                if(distance > 0) {
                    thrLocal.accumulate(distance);
                }
            }
        }
    }

    /**
     * simulate time passing
```

```java
     * @param tick the current time tick
     */
    public void tick(int tick) {
        Cricket c;
        for(int i = 0; i < v; i++) {
            c = vertices.get(i);
            c.timeTick(tick);
        }
        for(int i = 0; i < v; i++) {
            c = vertices.get(i);
            c.emitChirp();
        }
    }

    /**
     * Generate a random graph with a PRNG, a specified vertex count and
     * an edge probability
     *
     * @param prng Prof. Alan Kaminsky's Perfect Random Number Generator
     * @param v number of vertices to use
     * @param p edge probability between vertices
     * @return the randomly generated graph
     */
    public static UndirectedGraph randomGraph(Random prng, int v, double p,
            CricketObserver o) {
        UndirectedGraph g = new UndirectedGraph(v, o);
        UndirectedEdge edge;
        Cricket a, b;
        int edgeCount = 0;
        for (int i = 0; i < v; i++) {
            for (int j = i + 1; j < v; j++) {
                // connect edges
                // always order it `i` then `j`
                if(prng.nextDouble() <= p) {
                    a = g.vertices.get(i);
                    b = g.vertices.get(j);
                    edge = new UndirectedEdge(edgeCount++, a, b);
                    g.edges.add(edge);
                }
            }
        }
        return g;
    }

    /**
     * create a cycle graph
     * @param v number of vertices
     * @param o cricket observer crickets should report to
     * @return constructed cycle graph
     */
    public static UndirectedGraph cycleGraph(int v, CricketObserver o) {
        return kregularGraph(v, 1, o);
    }

    /**
     * create a k-regular graph
     * @param v number of vertices
     * @param k number of adjacent vertices left and right of given vertex to
     * connect to
     * @param o cricket observer the crickets should report to
     * @return the constructed k-regular graph
     */
    public static UndirectedGraph kregularGraph(int v, int k,
            CricketObserver o) {
        return smallWorldGraph(null, v, k, 0, o);
    }

    /**
     * create a small-world graph
     * @param prng pseudorandom number generator
     * @param v number of vertices
     * @param k the initial k-regular graph to modify
     * @param p edge rewire probability
     * @param o cricket observer the crickets should report to
     * @return the constructed small-world graph
     */
    public static UndirectedGraph smallWorldGraph(Random prng, final int v,
            int k, double p, CricketObserver o) {
        UndirectedGraph g = new UndirectedGraph(v, o);
        UndirectedEdge edge;
        Cricket a, b, c;
        int edgeCount = 0;
        for(int i = 0; i < v; i++) {
            a = g.vertices.get(i);
            for(int j = 1; j <= k; j++) {
                b = g.vertices.get((i + j) % v);
                if(prng != null && prng.nextDouble() < p) {
                    do {
                        c = g.vertices.get(prng.nextInt(v));
                    } while(c.n == a.n || c.n == b.n || a.directFlight(c));
                    b = c;
                }
                edge = new UndirectedEdge(edgeCount++, a, b);
                g.edges.add(edge);
            }
        }
        return g;
    }

    /**
     * create a scale-free graph
     * @param prng psuedorandom number generator to use
     * @param v number of vertices
     * @param dE number of edges to add to each additional vertex
     * @param o cricket observer the crickets should report to
     * @return the scale-free graph generated
     */
    public static UndirectedGraph scaleFreeGraph(Random prng, final int v,
            final int dE, CricketObserver o) {
        UndirectedGraph g = new UndirectedGraph(v, o);
//      boolean[]
        int edgeCount = 0;
        int c0 = prng.nextInt(v);
        int c1 = (c0 + 1) % v;
        int c2 = (c1 + 1) % v;
        Cricket a = g.vertices.get(c0), b = g.vertices.get(c1),
                c = g.vertices.get(c2);
        UndirectedEdge edge = new UndirectedEdge(edgeCount++, a, b);
        g.edges.add(edge);
        edge = new UndirectedEdge(edgeCount++, b, c);
        g.edges.add(edge);
        edge = new UndirectedEdge(edgeCount++, a, c);
        g.edges.add(edge);
        // we have 3 fully connected vertices now
        Cricket[] others = new Cricket[v-3];
        for(int other = 0, i = 0; i < v; i++) {
            if(i != c0 && i != c1 && i != c2) {
                others[other++] = g.vertices.get(i);
            }
        }
        // the rest are contained in others
        double[] cum = new double[v];
        double[] deg = new double[v];
        Cricket next, temp;
        ArrayList<Cricket> existing = new ArrayList<Cricket>();
        existing.add(a); existing.add(b); existing.add(c);
        Searching.Double h = new Searching.Double();
        for(int i = 0; i < others.length; i++) {
            next = others[i];
            existing.add(next);
            if(existing.size() <= dE) {
                for(int e = 0; e < existing.size(); e++) {
                    temp = existing.get(e);
                    if(next.equals(temp)) continue;
                    edge = new UndirectedEdge(edgeCount++, temp, next);
                    g.edges.add(edge);
                }
            } else {
                setDegreeDistribution(g, deg);
                for(int e = 0; e < dE; e++) {
                    setProbabilityDistribution(deg, cum);
                    double nr = prng.nextDouble();
                    double ch = cum[cum.length-1]*nr;
                    int vertex = Searching.searchInterval(cum, ch, h);
                    deg[vertex] = 0;
                    temp = g.vertices.get(vertex);
                    edge = new UndirectedEdge(edgeCount++, next, temp);
                    g.edges.add(edge);
                }
            }
        }

        return g;
    }

    /**
     * set the degree distribution of a given graph
     * @param g the given graph
     * @param deg the degree distribution of the graph
     */
    private static void setDegreeDistribution(UndirectedGraph g, double[] deg) {
        for(int i = 0; i < g.v; i++) {
            deg[i] = g.vertices.get(i).degree();
        }
    }

    /**
     * set the cumulative sum of the degree array
     * @param deg degrees of a graph
     * @param cum cumulative sum of the degree
     */
    private static void setProbabilityDistribution(double deg[], double[] cum) {
        double cumulative = 0;
        for(int i = 0; i < deg.length; i++) {
            cum[i] = cumulative += deg[i];
        }
    }
}

```