```java
import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.List;

/**
 *
 * @author jimiford
 *
 */
public class Automator {

    public static void main(String[] args) {
        if(args.length != 1) {
            usage();
        }
        try {
            List<String> lines =
Files.readAllLines(Paths.get(args[0]),
                    Charset.defaultCharset());
            String[] lineArr;
            int lineCount = 0;
            boolean skip, comment;
            for (String line : lines) {
                ++lineCount;
                line = line.trim();
                lineArr = line.split(" ");
                skip = lineArr[0].equals(line);
                comment = lineArr[0].startsWith("#");
                if(skip || comment) {
                    if(comment) {
                        if(line.equals("#")) {
                            System.out.println();
                        } else {
                            System.out.println(line);
                        }
                    }
                    continue;
                }
                Chirp.main(lineArr);
```

```
            }
        } catch (IOException e) {
            error("Error reading automation file");
        }
    }

    /**
     * display usage message and exit
     */
    private static void usage() {
        System.err.println("usage: java Automator <automation file>");
        System.exit(1);
    }

    private static void error(String msg) {
        System.err.println(msg);
        usage();
    }
}

import java.io.IOException;

import edu.rit.util.Random;

/**
 *
 * @author jimiford
 *
 */
public class Chirp {

    private static final int GRAPH_TYPE_INDEX = 0,
                             NUM_VERTICES_INDEX = 1,
                             NUM_TICKS_INDEX = 2,
                             OUTPUT_IMAGE_INDEX = 3,
                             SEED_INDEX = 4,
                             K_INDEX = 4,
                             DE_INDEX = 4,
                             DE_SEED_INDEX = 5,
                             EDGE_PROBABILITY_INDEX = 5,
```

```
81                                      K_SEED_INDEX = 5,
82                                      REWIRE_PROBABILITY_INDEX = 6;
83
84      public static void main(String[] args) {
85          if(args.length != 4 && args.length != 5 &&
86                  args.length != 6 && args.length != 7) usage();
87          int crickets = 0, ticks = 0, k = 0, dE = 0;
88          long seed = 0;
89          double prob = 0;
90          char mode;
91          String outputImage = args[OUTPUT_IMAGE_INDEX];
92
93          try {
94              crickets = Integer.parseInt(args[NUM_VERTICES_INDEX]);
95          } catch (NumberFormatException e) {
96              error("<num vertices> must be a number");
97          }
98          try {
99              ticks = Integer.parseInt(args[NUM_TICKS_INDEX]) + 1;
100         } catch (NumberFormatException e) {
101             error("<num ticks> must be numeric");
102         }
103         mode = args[GRAPH_TYPE_INDEX].toLowerCase().charAt(0);
104         if(!(mode == 'c' || mode == 'r' || mode == 'k' ||
105                 mode == 's' || mode == 'f')) {
106             error("<graph type> must be either 'c' for cycle, "
107                     + "'r' for random, "
108                     + "'k' for k-regular, "
109                     + "'s' for small-world, "
110                     + "'f' for scale-free");
111         }
112         UndirectedGraph g = null;
113         CricketObserver o = new CricketObserver(crickets, ticks);
114         switch(mode) {
115         case 'r': // RANDOM GRAPH
116             try {
117                 seed = Long.parseLong(args[SEED_INDEX]);
118                 prob =
    Double.parseDouble(args[EDGE_PROBABILITY_INDEX]);
119                 g = UndirectedGraph.randomGraph(new Random(seed),
    crickets, prob, o);
```

```
120                 } catch(NumberFormatException e) {
121                     error("<seed> and <edge probability> must be
    numeric");
122                 } catch(IndexOutOfBoundsException e) {
123                     error("<seed> and <edge probability> must be
    included with random graph mode");
124                 }
125                 break;
126             case 'c': // CYCLE GRAPH
127                 g = UndirectedGraph.cycleGraph(crickets, o);
128                 break;
129             case 'k': // K-REGULAR GRAPH
130                 try {
131                     k = Integer.parseInt(args[K_INDEX]);
132                     g = UndirectedGraph.kregularGraph(crickets, k, o);
133                 } catch (NumberFormatException e) {
134                     error("<k> must be an integer");
135                 } catch (IllegalArgumentException e) {
136                     error("<k> must be < the number of crickets");
137                 }
138                 break;
139             case 's': // SMALL WORLD GRAPH
140                 try {
141                     k = Integer.parseInt(args[K_INDEX]);
142                     prob =
    Double.parseDouble(args[REWIRE_PROBABILITY_INDEX]);
143                     seed = Long.parseLong(args[K_SEED_INDEX]);
144                     g = UndirectedGraph.smallWorldGraph(new
    Random(seed), crickets, k, prob, o);
145                 } catch (NumberFormatException e) {
146                     error("<k> must be an integer < V, <rewire
    probability> must be a number "
147                             + "between 0 and 1, and <seed> must be
    numeric");
148                 } catch (IllegalArgumentException e) {
149                     error("<k> must be < the number of crickets");
150                 }
151                 break;
152             case 'f':
153                 try {
154                     dE = Integer.parseInt(args[DE_INDEX]);
```

```
155                     seed = Long.parseLong(args[DE_SEED_INDEX]);
156                     g = UndirectedGraph.scaleFreeGraph(new Random(seed),
    crickets, dE, o);
157                 } catch (NumberFormatException e) {
158                     error("<dE> and <seed> must be numeric");
159                 } catch (IndexOutOfBoundsException e) {
160                     error("<dE> and <seed> must be supplied");
161                 }
162             }
163 
164             g.vertices.get(0).forceChirp();
165             Ticker.tick(g, ticks);
166 
167 
168 
169             try {
170                 ImageHandler.handle(o, outputImage);
171             } catch (IOException e) {
172                 error("Problem writing image");
173             }
174             int sync = o.sync();
175             String description;
176             switch(mode) {
177             case 'c': // CYCLE GRAPH
178                 description = "Cycle V = " + crickets +":";
179                 handleOutput(description,sync);
180                 break;
181             case 'r': // RANDOM GRAPH
182                 description = "Random V = " + crickets +", p = " + prob
    + ":";
183                 handleOutput(description,sync);
184                 break;
185             case 'k': // K-REGULAR GRAPH
186                 description = "K-regular V = " + crickets +", k = " + k
    + ":";
187                 handleOutput(description,sync);
188                 break;
189             case 's': // SMALL-WORLD GRAPH
190                 description = "Small-world V = " + crickets + ", k = " +
    k +
191                     ", p = " + prob + ":";
```

```
192                 handleOutput(description,sync);
193                 break;
194             case 'f': // SCALE-FREE GRAPH
195                 description = "Scale-free V = " + crickets +", dE = " +
    dE + ":";
196                 handleOutput(description,sync);
197                 break;
198             }
199 
200         }
201 
202         private static void handleOutput(String description, int sync) {
203             System.out.print(description);
204             if(sync >= 0) {
205                 System.out.println("\t"+" synchronized at t="+sync+".");
206             } else {
207                 System.out.println("\t "+(char)27+"[31m"+  "did not
    synchronize." +
208                         (char)27 + "[0m");
209             }
210         }
211 
212         private static void error(String msg) {
213             System.err.println(msg);
214             usage();
215         }
216 
217         private static void usage() {
218             System.err.println("usage: java Chirp <graph type> <num
    vertices> <num ticks> "
219                     + "<output image> {(<seed> <edge probability>), or "
220                     + "(<k>), or "
221                     + "(<k> <seed> <rewire probability>), or "
222                     + "(<dE> <seed>)}");
223             System.exit(1);
224         }
225 }
226 
227 
228 public class Cricket extends Vertex {
229 
```

```
230 //  private boolean[] chirp = new boolean[3];
231     private boolean[] chirp = new boolean[2];
232     private boolean willChirp;
233     private int currentTick = 0;
234     private final CricketObserver observer;
235 
236     public Cricket(int n, CricketObserver o) {
237         super(n);
238         this.observer = o;
239     }
240 
241     public void forceChirp() {
242         willChirp = chirp[0] = true;
243     }
244 
245     public void emitChirp() {
246         if(willChirp) {
247             willChirp = false;
248             int n = super.degree();
249             for(int i = 0; i < n; i++) {
250                 edges.get(i).other(this).hearChirp();
251             }
252             observer.reportChirp(currentTick, super.n);
253         }
254     }
255 
256     private void hearChirp() {
257         chirp[1] = true;
258     }
259 
260     public void timeTick(int tick) {
261         currentTick = tick;
262         willChirp = chirp[0];
263         chirp[0] = chirp[1];
264 //        chirp[1] = chirp[2];
265 //        chirp[2] = false;
266         chirp[1] = false;
267     }
268 
269     public boolean directFlight(Cricket other) {
270         boolean retval = false;
```

```
271             if(equals(other)) return true;
272             int e = super.degree();
273             Cricket o;
274             for(int i = 0; i < e && !retval; i++) {
275                 o = super.edges.get(i).other(this);
276                 retval = o.equals(other);
277             }
278             return retval;
279         }
280 
281         public boolean equals(Object o) {
282             if( !(o instanceof Cricket)) {
283                 return false;
284             }
285             if(o == this) {
286                 return true;
287             }
288             Cricket casted = (Cricket) o;
289 
290             return casted.n == this.n;
291         }
292 }
293 
294 
295 public class CricketObserver {
296 
297     public final int crickets, ticks;
298     private boolean[][] chirps;
299 
300     public CricketObserver(int crickets, int ticks) {
301         this.crickets = crickets;
302         this.ticks = ticks;
303         chirps = new boolean[ticks][crickets];
304     }
305 
306     public void reportChirp(int tick, int n) {
307         chirps[tick][n] = true;
308     }
309 
310     public boolean chirped(int tick, int cricket) {
311         return chirps[tick][cricket];
```

```
312     }
313 
314     public int sync() {
315         int row = 0;
316         while(row < ticks) {
317             if(sync(row)) return row;
318             row++;
319         }
320         return -1;
321     }
322 
323     private boolean sync(int tick) {
324         boolean retval = true;
325         for(int i = 0; i < crickets && retval; i++) {
326             retval = chirps[tick][i];
327         }
328         return retval;
329     }
330 
331 //  private boolean equal(boolean[] a, boolean[] b) {
332 //      boolean retval = true;
333 //      if(a.length == b.length) {
334 //          for(int i = 0; i < a.length && retval; i++) {
335 //              retval = a[i] == b[i];
336 //          }
337 //      }
338 //      return retval;
339 //  }
340 }
341 
342 
343 
344 import java.io.BufferedOutputStream;
345 import java.io.File;
346 import java.io.FileNotFoundException;
347 import java.io.FileOutputStream;
348 import java.io.IOException;
349 import java.io.OutputStream;
350 
351 import edu.rit.image.ByteImageQueue;
352 import edu.rit.image.Color;
```

```
353 import edu.rit.image.IndexPngWriter;
354 import edu.rit.util.AList;
355
356
357 public class ImageHandler {
358
359     public static final byte SILENT = 0,
360                              CHIRPED = 1,
361                              SYNC = 2;
362
363     public static void handle(CricketObserver o, String out) throws
    FileNotFoundException {
364         AList<Color> palette = new AList<Color>(); // green
365         Color green = new Color().rgb(0, 255, 0);
366         Color red = new Color().rgb(255, 0, 0); // red
367         Color blue = new Color().rgb(0,0,255); // blue
368         palette.addLast (green);
369         palette.addLast (red);
370         palette.addLast (blue);
371
372
373         OutputStream imageout =
374                 new BufferedOutputStream (new FileOutputStream (new
    File(out)));
375         IndexPngWriter imageWriter = new IndexPngWriter
376                 (o.ticks, o.crickets, imageout, palette);
377         ByteImageQueue imageQueue = imageWriter.getImageQueue();
378         byte[] bytes;
379         boolean chirped;
380         int sync = o.sync();
381         for(int i = 0; i < o.ticks; i++) {
382             bytes = new byte[o.crickets];
383             for(int j = 0, cricket = 0; j < bytes.length; j++,
    cricket++) {
384                 if(i != sync) {
385                     chirped = o.chirped(i, cricket);
386                     bytes[j] = chirped ? CHIRPED : SILENT;
387                 } else {
388                     bytes[j] = SYNC;
389                 }
390             }
```

```
391                 try {
392                     imageQueue.put(i, bytes);
393                 } catch (InterruptedException e) {
394                     // TODO Auto-generated catch block
395                     e.printStackTrace();
396                 }
397             }
398             try {
399                 imageWriter.write();
400             } catch (IOException e) {
401                 // TODO Auto-generated catch block
402                 e.printStackTrace();
403             } catch (InterruptedException e) {
404                 // TODO Auto-generated catch block
405                 e.printStackTrace();
406             }
407         }
408 }
409
410
411 public class Ticker {
412
413     public static void tick(UndirectedGraph g, int ticks) {
414         for(int i = 0; i < ticks; i++) {
415             g.tick(i);
416         }
417     }
418 }
419
420 //
    ************************************************************************
    **********
421 //
422 //File:    UndirectedEdge.java
423 //Package: ---
424 //Unit:    Class UndirectedEdge
425 //
426 //
    ************************************************************************
    **********
427
```

```java
/**
* Class UndirectedEdge represents an edge in a graph that connects two
* vertices. It's important to note that the edge does not have a direction nor
* weight.
*
* @author Jimi Ford
* @version 2-15-2015
*/
public class UndirectedEdge {

    // private data members
    private Cricket a, b;

    // future projects may rely on a unique identifier for an edge
    private final int id;

    /**
     * Construct an undirected edge
     * @param id a unique identifier to distinguish between other edges
     * @param a one vertex in the graph
     * @param b another vertex in the graph not equal to <I>a</I>
     */
    public UndirectedEdge(int id, Cricket a, Cricket b) {
        this.id = id;
        // enforce that a.n is always less than b.n
        if(a.n < b.n) {
            this.a = a;
            this.b = b;
        } else if(b.n < a.n) {
            this.a = b;
            this.b = a;
        } else {
//            System.out.println(a.n + ", " + b.n +", "+ (a==b));
            throw new IllegalArgumentException("Cannot have self loop");
        }
        this.a.addEdge(this);
        this.b.addEdge(this);
```

```
465     }
466 
467     /**
468      * Get the <I>other</I> vertex given a certain vertex connected to
469      * this edge
470      *
471      * @param current the current vertex
472      * @return the other vertex connected to this edge
473      */
474     public Cricket other(Cricket current) {
475         if(current == null) return null;
476         return current.n == a.n ? b : a;
477     }
478 }
479 
480 //**********************************************************************
481 //
482 //File:    UndirectedGraph.java
483 //Package: ---
484 //Unit:    Class UndirectedGraph
485 //
486 //**********************************************************************
487 
488 import java.util.ArrayList;
489 import java.util.LinkedList;
490 import edu.rit.pj2.vbl.DoubleVbl;
491 import edu.rit.util.Random;
492 
493 /**
494 * Class UndirectedGraph represents an undirected graph meaning that if
495 * there exists an edge connecting some vertex A to some vertex B, then
496 * that same edge connects vertex B to vertex A.
497 *
498 * @author Jimi Ford
```

```
499 * @version 2-15-2015
500 */
501 public class UndirectedGraph {
502 
503     // private data members
504     private ArrayList<UndirectedEdge> edges;
505     public ArrayList<Cricket> vertices;
506     private int v;
507 
508     // Prevent construction
509     private UndirectedGraph() {
510 
511     }
512 
513     /**
514      * Private constructor used internally by the static random
    graph
515      * method
516      * @param v the number of vertices in the graph
517      */
518     private UndirectedGraph(int v, CricketObserver o) {
519         this.v = v;
520         vertices = new ArrayList<Cricket>(v);
521         edges = new ArrayList<UndirectedEdge>();
522         for(int i = 0; i < v; i++) {
523             vertices.add(new Cricket(i,o));
524         }
525     }
526 
527     /**
528      * Perform a BFS to get the distance from one vertex to another
529      *
530      * @param start the id of the start vertex
531      * @param goal the id of the goal vertex
532      * @return the minimum distance between the two vertices
533      */
534     private int BFS(int start, int goal) {
535         return BFS(vertices.get(start), vertices.get(goal));
536     }
537 
538     /**
```

```
539         * Perform a BFS to get the distance from one vertex to another
540         *
541         * @param start the reference to the start vertex
542         * @param goal the reference to the goal vertex
543         * @return the minimum distance between the two vertices
544         */
545        private int BFS(Cricket start, Cricket goal) {
546            int distance = 0, verticesToProcess = 1, uniqueNeighbors =
    0;
547            LinkedList<Cricket> queue = new LinkedList<Cricket>();
548            boolean[] visited = new boolean[v];
549            visited[start.n] = true;
550            Cricket current, t2;
551            queue.add(start);
552            while(!queue.isEmpty()) {
553                current = queue.removeFirst();
554                if(current.equals(goal)) {
555                    return distance;
556                }
557                for(int i = 0; i < current.degree(); i++) {
558                    t2 = current.getEdges().get(i).other(current);
559                    if(!visited[t2.n]) {
560                        visited[t2.n] = true;
561                        queue.add(t2);
562                        uniqueNeighbors++;
563                    }
564                }
565                verticesToProcess--;
566                if(verticesToProcess <= 0) {
567                    verticesToProcess = uniqueNeighbors;
568                    uniqueNeighbors = 0;
569                    distance++;
570                }
571
572            }
573            return 0;
574        }
575
576        /**
577         * Accumulate the distances of each pair of vertices into
578         * a "running total" to be averaged
```

```
579         *
580         * @param thrLocal the reference to the "running total"
581         * Prof. Alan Kaminsky's library handles averaging this
582         * accumulated value.
583         */
584        public void accumulateDistances(DoubleVbl.Mean thrLocal) {
585            for(int i = 0; i < v; i++) {
586                for(int j = i + 1; j < v; j++) {
587                    int distance = BFS(i, j);
588                    // only accumulate the distance if the two vertices
589                    // are actually connected
590                    if(distance > 0) {
591                        thrLocal.accumulate(distance);
592                    }
593                }
594            }
595        }
596
597        public void tick(int tick) {
598            Cricket c;
599            for(int i = 0; i < v; i++) {
600                c = vertices.get(i);
601                c.timeTick(tick);
602            }
603            for(int i = 0; i < v; i++) {
604                c = vertices.get(i);
605                c.emitChirp();
606            }
607        }
608
609        /**
610         * Generate a random graph with a PRNG, a specified vertex count
    and
611         * an edge probability
612         *
613         * @param prng Prof. Alan Kaminsky's Perfect Random Number
    Generator
614         * @param v number of vertices to use
615         * @param p edge probability between vertices
616         * @return the randomly generated graph
617         */
```

```
618     public static UndirectedGraph randomGraph(Random prng, int v,
    double p, CricketObserver o) {
619         UndirectedGraph g = new UndirectedGraph(v, o);
620         UndirectedEdge edge;
621         Cricket a, b;
622         int edgeCount = 0;
623         for (int i = 0; i < v; i++) {
624             for (int j = i + 1; j < v; j++) {
625                 // connect edges
626                 // always order it `i` then `j`
627                 if(prng.nextDouble() <= p) {
628                     a = g.vertices.get(i);
629                     b = g.vertices.get(j);
630                     edge = new UndirectedEdge(edgeCount++, a, b);
631                     g.edges.add(edge);
632                 }
633             }
634         }
635         return g;
636     }
637 
638     public static UndirectedGraph cycleGraph(int v, CricketObserver
    o) {
639         return kregularGraph(v, 1, o);
640     }
641 
642     public static UndirectedGraph kregularGraph(int v, int k,
    CricketObserver o) {
643         return smallWorldGraph(null, v, k, 0, o);
644     }
645 
646     public static UndirectedGraph smallWorldGraph(Random prng, final
    int v, int k, double p, CricketObserver o) {
647         UndirectedGraph g = new UndirectedGraph(v, o);
648         UndirectedEdge edge;
649         Cricket a, b, c;
650         int edgeCount = 0;
651         for(int i = 0; i < v; i++) {
652             a = g.vertices.get(i);
653             for(int j = 1; j <= k; j++) {
654                 b = g.vertices.get((i + j) % v);
```

```
655                     if(prng != null && prng.nextDouble() < p) {
656                         do {
657                             c = g.vertices.get(prng.nextInt(v));
658                         } while(c.n == a.n || c.n == b.n ||
    a.directFlight(c));
659                         b = c;
660                     }
661                     edge = new UndirectedEdge(edgeCount++, a, b);
662                     g.edges.add(edge);
663                 }
664             }
665             return g;
666         }
667 
668         public static UndirectedGraph scaleFreeGraph(Random prng, final
    int v,
669                 final int dE, CricketObserver o) {
670             UndirectedGraph g = new UndirectedGraph(v, o);
671 //          boolean[]
672             int edgeCount = 0;
673             int c0 = prng.nextInt(v);
674             int c1 = (c0 + 1) % v;
675             int c2 = (c1 + 1) % v;
676             Cricket a = g.vertices.get(c0), b = g.vertices.get(c1), c =
    g.vertices.get(c2);
677             UndirectedEdge edge = new UndirectedEdge(edgeCount++, a, b);
678             g.edges.add(edge);
679             edge = new UndirectedEdge(edgeCount++, b, c);
680             g.edges.add(edge);
681             edge = new UndirectedEdge(edgeCount++, a, c);
682             g.edges.add(edge);
683             // we have 3 fully connected vertices now
684             Cricket[] others = new Cricket[v-3];
685             for(int other = 0, i = 0; i < v; i++) {
686                 if(i != c0 && i != c1 && i != c2) {
687                     others[other++] = g.vertices.get(i);
688                 }
689             }
690             // the rest are contained in others
691             int[] prob;
692             Cricket next, temp;
```

```
693         ArrayList<Cricket> existing = new ArrayList<Cricket>();
694         existing.add(a); existing.add(b); existing.add(c);
695         for(int i = 0; i < others.length; i++) {
696             next = others[i];
697             existing.add(next);
698             if(existing.size() <= dE) {
699                 for(int e = 0; e < existing.size(); e++) {
700                     temp = existing.get(e);
701                     if(next.equals(temp)) continue;
702                     edge = new UndirectedEdge(edgeCount++, temp,
    next);
703                     g.edges.add(edge);
704                 }
705             } else {
706                 // potential bug - when do i add in the current
    vertex to the
707                 // probability distribution?
708                 int sumD = sumDeg(g);
709                 prob = new int[sumD];
710                 setProbabilityDistribution(g, prob);
711                 for(int e = 0; e < dE; e++) {
712                     do {
713                         int chosen = (int)
    Math.floor(prng.nextDouble() * prob.length);
714                         temp = g.vertices.get(prob[chosen]);
715                     } while(next.directFlight(temp));
716                     edge = new UndirectedEdge(edgeCount++, next,
    temp);
717                     g.edges.add(edge);
718                 }
719             }
720         }
721 
722         return g;
723     }
724 
725     private static void setProbabilityDistribution(UndirectedGraph
    g, int[] prob) {
726         Vertex v;
727         int degree = 0;
728         int counter = 0;
```

```
729             for(int i = 0; i < g.v; i++) {
730                 v = g.vertices.get(i);
731                 degree = v.degree();
732                 for(int j = counter; j < degree + counter; j++) {
733                     prob[j] = v.n;
734                 }
735                 counter += degree;
736             }
737         }
738 
739         private static int sumDeg(UndirectedGraph g) {
740             int retval = 0;
741             Vertex v;
742             for(int i = 0; i < g.v; i++) {
743                 v = g.vertices.get(i);
744                 retval += v.degree();
745             }
746             return retval;
747         }
748 }
749 
750 //
    ************************************************************************
    **********
751 //
752 //File:    Vertex.java
753 //Package: ---
754 //Unit:    Class Vertex
755 //
756 //
    ************************************************************************
    **********
757 
758 import java.util.ArrayList;
759 
760 /**
761 * Class Vertex represents a single vertex in a graph. Vertices can
    be connected
762 * to other vertices through undirected edges.
763 *
764 * @author Jimi Ford
```

```
765 * @version 2-15-2015
766 */
767 public class Vertex {
768 
769     // private data members
770     protected ArrayList<UndirectedEdge> edges = new
    ArrayList<UndirectedEdge>();
771 
772     /**
773      * The unique identifier for this vertex
774      */
775     public final int n;
776 
777     /**
778      * Construct a vertex with a unique identifier <I>n</I>
779      *
780      * @param n the unique identifier to distinguish this vertex
    from
781      *          all other vertices in the graph
782      */
783     public Vertex(int n) {
784         this.n = n;
785     }
786 
787     /**
788      * Get the number of edges connected to this vertex
789      *
790      * @return the number of edges connected to this vertex
791      */
792     public int degree() {
793         return edges.size();
794     }
795 
796     /**
797      * Get the reference to the collection of edges connected to
798      * this vertex.
799      *
800      * @return the reference to the collection of edges
801      */
802     public ArrayList<UndirectedEdge> getEdges() {
803         return this.edges;
```

```
804     }
805
806     /**
807      * Add an edge to this vertex
808      *
809      * @param e the edge to add
810      */
811     public void addEdge(UndirectedEdge e) {
812         this.edges.add(e);
813     }
814
815     /**
816      * Compare another object to this one
817      *
818      * @param o the other object to compare to this one
819      * @return true if the other object is equivalent to this one
820      */
821     public boolean equals(Object o) {
822         if( !(o instanceof Vertex)) {
823             return false;
824         }
825         if(o == this) {
826             return true;
827         }
828         Vertex casted = (Vertex) o;
829
830         return casted.n == this.n;
831     }
832 }
833
```