Jimi Ford
CSCI 351-01 Data Communications and Networks
Programming Project 2 Report

1. Program Inputs, Program Objective, Program Outputs

<u>Chirp.java</u>

This program takes in a character for the type of graph to generate, the number of vertices (or crickets) and the number of time ticks to step through. Then depending on what graph type is used, the program takes in different parameters. When generating a single cycle graph, no further arguments are required to run the program. When generating a k-regular graph, an additional argument ***k***, must be supplied. When generating a scale-free graph, the change in entropy ***ΔE***, must be supplied. When generating a random graph, the edge probability ***p***, and seed value ***seed***, must be supplied. When generating a small-world graph, edge probability ***p***, a seed value ***seed***, and ***k*** must be included. Regardless of what graph type is used, the program simulates the cricket network as a graph of vertices connected to other vertices through unweighted, undirected edges. Each vertex represents a cricket, and each edge (connecting 2 vertices) represents the fact that 2 crickets are near each other. The program's objective is to explore how the topology of a network of chirping crickets effects chirp-synchronization. After the program ensures proper usage, it generates a single graph (according to the type specified), tells cricket number 0 to chirp at ***t=0***, simulates the number of time ticks and generates an image describing the result of the simulation.

2. Exact Command Line

<u>Chirp.java</u>

note: This program accepts numerous amounts of supplied command line arguments due to the differing number of necessary "knobs" needed for each kind of graph.

Cycle Graph Mode

```
usage: java Chirp c <num crickets> <num ticks> <output image>
```

c	- denotes "cycle graph" mode
<num crickets>	- number of crickets (vertices) in graph
<num ticks>	- number of time steps to simulate the network for
<output image>	- name of output image file describing the results

K-Regular Graph Mode

```
usage: java Chirp k <num crickets> <num ticks> <output image> <k>
```

k	- denotes "k-regular graph" mode

<num crickets> - number of crickets (vertices) in graph
<num ticks> - number of time steps to simulate the network for
<output image> - name of output image file describing the results
<k> - number of vertices to connect a given vertex to (on its left and right)

Scale-free Graph Mode

usage: java Chirp k <num crickets> <num ticks> <output image> <E>

f - denotes “scale-free graph” mode
<num crickets> - number of crickets (vertices) in graph
<num ticks> - number of time steps to simulate the network for
<output image> - name of output image file describing the results
<E> - entropy cutoff in graph generation

Random Graph Mode

usage: java Chirp r <num crickets> <num ticks> <output image> <seed> <p>

r - denotes “random graph” mode
<num crickets> - number of crickets (vertices) in graph
<num ticks> - number of time steps to simulate the network for
<output image> - name of output image file describing the results
<seed> - seed value for the Pseudorandom Number Generator
<p> - edge probability of every pair of vertices

Small World Graph Mode

usage: java Chirp s <num crickets> <num ticks> <output image> <k> <seed> <p>

s - denotes “small world graph” mode
<num crickets> - number of crickets (vertices) in graph
<num ticks> - number of time steps to simulate the network for
<output image> - name of output image file describing the results
<k> - the type of k-regular graph to mimic before edges are rewired
<seed> - seed value for the Pseudorandom Number Generator
<p> - rewiring probability for every edge in the k regular graph

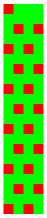
3. Source Code ***(See Appendix A for project’s source code)***

4. (Even number cycle graph) Do the networks synchronize? If so, how long do the networks take to synchronize? Why are the networks behaving in this fashion?

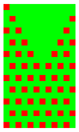
---

note: Some of the images are very small. When enlarged with Google Drive's image editor, the square pixels are smoothened out. The actual images generated by the program will contain squared off pixels. A green pixel represents a silent cricket, and a red pixel represents a cricket that chirped at that moment in time. Time starts at the top of each image and goes forward towards the bottom of the image.

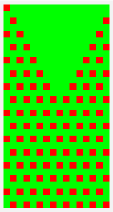
`java Chirp c 4 20 cycle-even-004.png` *- did not synchronize*



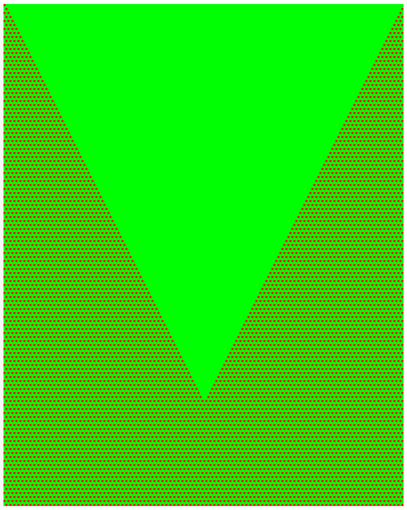
`java Chirp c 12 20 cycle-even-012.png` *- did not synchronize*



`java Chirp c 16 30 cycle-even-016.png` *- did not synchronize*



`java Chirp c 200 250 cycle-even-200.png` *- did not synchronize*



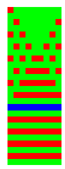
We can see from the images generated that any even amount of crickets that we try will not synchronize. This is due to the fact that an even amount of crickets means at no point will two adjacent crickets chirp at the same time. Consider the Cycle A -> B -> C -> D -> A, with an even amount of vertices. If A chirps at t=0, B and D will chirp at t=2. Then A and C will chirp at t=4, and B and D will chirp at t=6. Since the crickets that are chirping at the exact same time are not adjacent, the synchronization process cannot start. This holds true for any amount of even crickets in the cycle graph.

5. (Odd number cycle graph) Do the networks synchronize? If so, how long do the networks take to synchronize? Why are the networks behaving in this fashion?

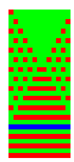
---

note: The blue horizontal line indicates the time at which the crickets synchronized.

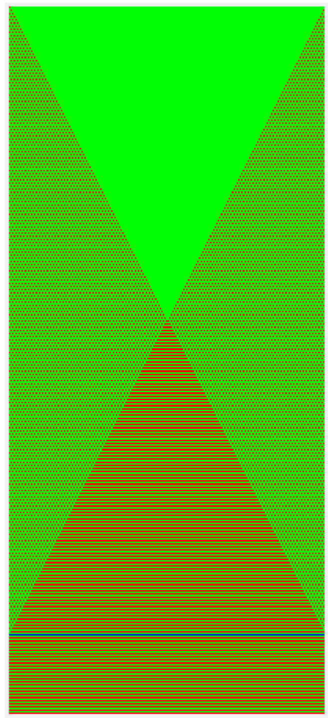
```
java Chirp c 9 25 cycle-odd-009.png
Cycle V = 9: synchronized at t=16.
```



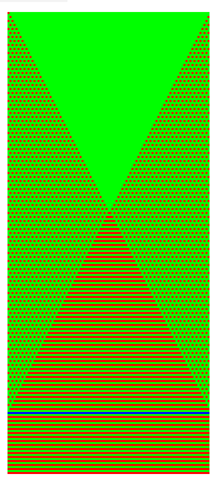
```
java Chirp c 13 30 cycle-odd-013.png
Cycle V = 13: synchronized at t=24.
```



```
java Chirp c 201 450 cycle-odd-201.png
Cycle V = 201: synchronized at t=400.
```



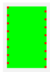
```
java Chirp c 101 230 cycle-odd-101.png
Cycle V = 101: synchronized at t=200.
```



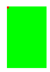
The crickets in every cycle graph with an odd number of vertices synchronizes according to the following equation $\textit{time to synchronize} = (\textit{number of crickets} - 1) * 2$. This is because it takes $(\textit{number of crickets} - 1)\ \textit{ticks}$ for the first chirp to spread to all the crickets in the network. (This is at the bottom of the "green triangle" in the above pictures.) At that point, 2 crickets are guaranteed to chirp at the same time. For instance, A -> B -> C -> A. When A chirps at t=0, B and C will chirp at t=2. Then A, B, and C will chirp at t=4. The initial chirp must travel all the way around the network, meet in the middle with 2 adjacent crickets chirping at the same time, and then travel all the way back to the beginning cricket.

6. (Random graph) Do the networks synchronize? If so, how long do the networks take to synchronize? Why are the networks behaving in this fashion?

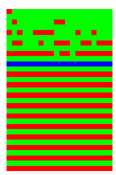
`java Chirp r 20 30 random-20-.08.png 15432 .08` *- did not synchronize*



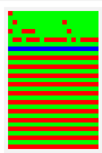
`java Chirp r 20 30 random-20-.1.png 12345 .1` *- did not synchronize*



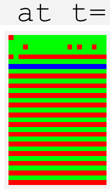
```
java Chirp r 20 30 random-20-.15.png 12359 .15
Random V = 20, p = 0.15: synchronized at t=10.
```



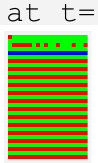
```
java Chirp r 20 30 random-20-.2.png 23451 .2
Random V = 20, p = 0.2: synchronized at t=8.
```



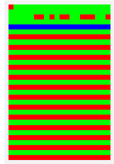
```
java Chirp r 20 30 random-20-.3.png 34512 .3
Random V = 20, p = 0.3: synchronized at t=6.
```



```
java Chirp r 20 30 random-20-.4.png 45123 .4
Random V = 20, p = 0.4: synchronized at t=4.
```

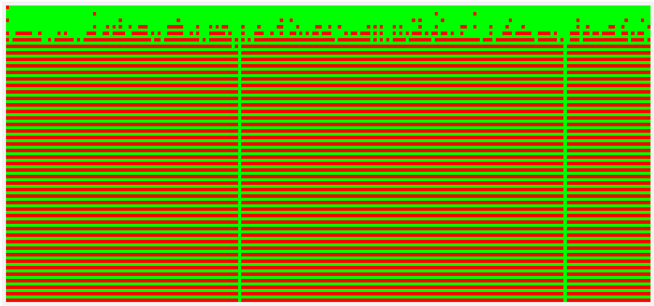


```
java Chirp r 20 30 random-20-.5.png 51234 .5
Random V = 20, p = 0.5: synchronized at t=4.
```

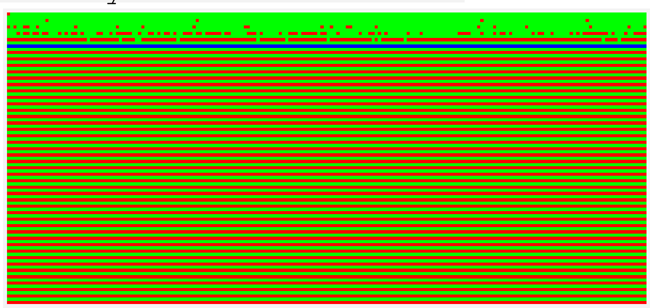


```
java Chirp r 200 90 random-200-.02.png -321540 .02
```

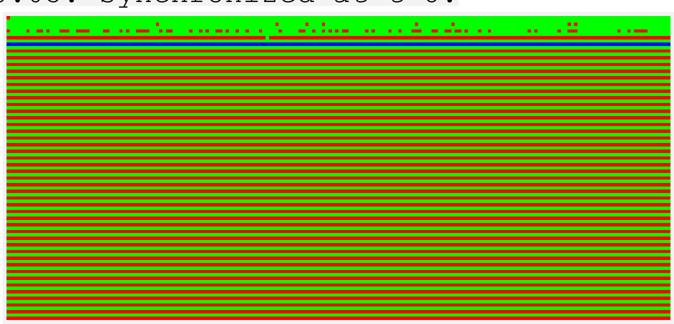
- did not synchronize



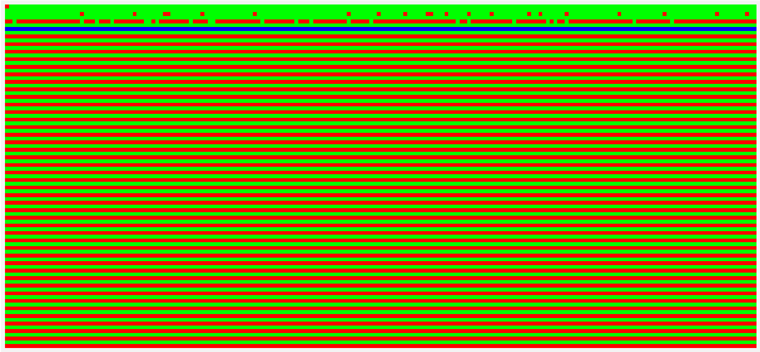
```
java Chirp r 200 90 random-200-.03.png -432150 .03
Random V = 200, p = 0.03: synchronized at t=10.
```



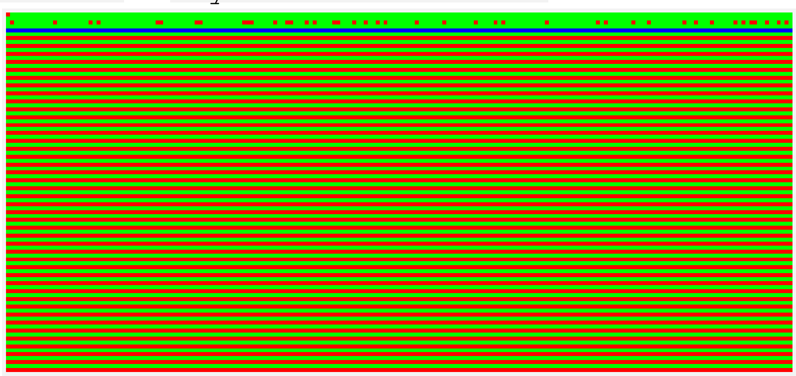
```
java Chirp r 200 90 random-200-.05.png -51234 .05
Random V = 200, p = 0.05: synchronized at t=8.
```



```
java Chirp r 200 90 random-200-.1.png 12345 .1
Random V = 200, p = 0.1: synchronized at t=6.
```



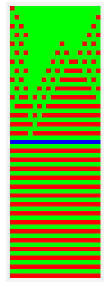
```
java Chirp r 200 90 random-200-.2.png 23451 .2
Random V = 200, p = 0.2:    synchronized at t=4.
```



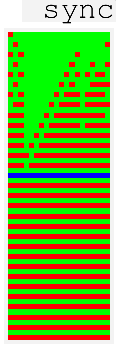
Depending on what parameters are used for random graphs, the crickets will usually synchronize, but not always. Some graphs can have vertices with no edges connected to them when *p* is low enough which means those crickets will never hear other crickets chirp so won't ever chirp themselves. When we have a small amount of crickets, like 20, a low edge probability will cause the crickets to take longer to synchronize. As we saw in the previous project, this combination of vertices and edge probability tends to yield graphs with longer average distances. The cycle graphs mentioned in the first two questions have a pretty high average distance compared to the rest of the graphs generated which is part of the reason those graphs took so long to synchronize, so introducing this aspect into random graphs will produce similar results. We see that the higher and higher edge probability we use with any amount of vertices will take less and less time. This is because we are increasing the connectivity of the graph and adding more and more edges. The more edges the graph has, the higher the chance is that 2 adjacent crickets will chirp at the same time. Once 2 adjacent crickets chirp, they will forever chirp since they are chirping and hearing the other person chirp at the same time. Meanwhile, while these infinitely chirping crickets chirp, their chirps "echo" outwards in the network

7. (Small-world graph) Do the networks synchronize? If so, how long do the networks take to synchronize? Why are the networks behaving in this fashion?

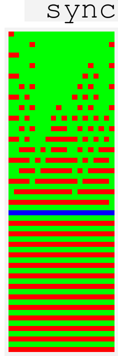
```
java Chirp s 20 60 small-20-k1-.1.png 1 23456 .1
Small-world V = 20, k = 1, p = 0.1:   synchronized at t=30.
```



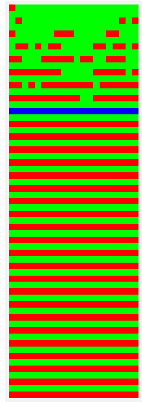
```
java Chirp s 20 60 small-20-k1-.2.png 1 34561 .2
Small-world V = 20, k = 1, p = 0.2:   synchronized at t=28.
```



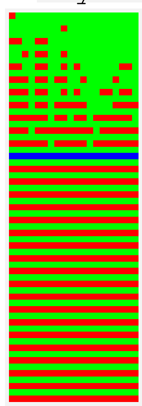
```
java Chirp s 20 60 small-20-k1-.5.png 1 62345 .5
Small-world V = 20, k = 1, p = 0.5:   synchronized at t=34.
```



```
java Chirp s 20 60 small-20-k1-.5n2.png 1 -92245 .5
Small-world V = 20, k = 1, p = 0.5:   synchronized at t=16.
```

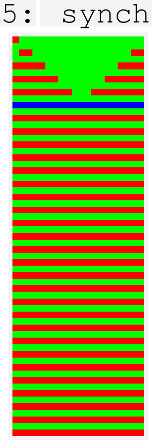


```
java Chirp s 20 60 small-20-k1-.8.png 1 43260 .8
Small-world V = 20, k = 1, p = 0.8:   synchronized at t=22.
```



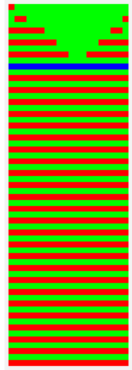
The above graphs begin producing patterns similar to the cycle graph. This is because they are near-k-regular graphs where k=1. The cycle graphs are k-regular where k=1. Looking at the resulting synchronization times, these kinds of graphs don't show a consistent pattern. We see that 2 different trials where k=1 and p=0.5 produce synchronization times of 16 and 34. The element of randomness introduced into the graph is responsible for this and the fact that the graphs are so sparse.

```
java Chirp s 20 60 small-20-k2-.025.png 2 234562 .025
Small-world V = 20, k = 2, p = 0.025: synchronized at t=10.
```



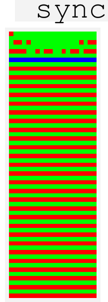
```
java Chirp s 20 60 small-20-k2-.05.png 2 234561 .05
```

```
Small-world V = 20, k = 2, p = 0.05:  synchronized at t=10.
```

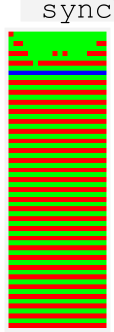


```
java Chirp s 20 60 small-20-k2-.1.png 2 234560 .1
Small-world V = 20, k = 2, p = 0.1:   synchronized at t=8.
```

```
java Chirp s 20 60 small-20-k2-.2.png 2 345610 .2
Small-world V = 20, k = 2, p = 0.2:   synchronized at t=6.
```

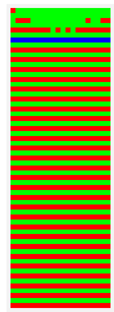


```
java Chirp s 20 60 small-20-k2-.3.png 2 456230 .3
Small-world V = 20, k = 2, p = 0.3:   synchronized at t=8.
```



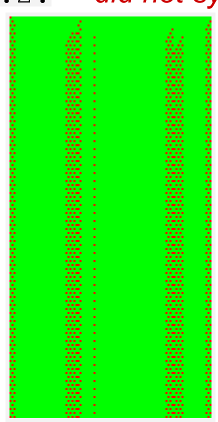
```
java Chirp s 20 60 small-20-k3-.025.png 3 234562 .025
Small-world V = 20, k = 3, p = 0.025: synchronized at t=8.
```

```
java Chirp s 20 60 small-20-k3-.3.png 3 456230 .3
Small-world V = 20, k = 3, p = 0.3:   synchronized at t=6.
```



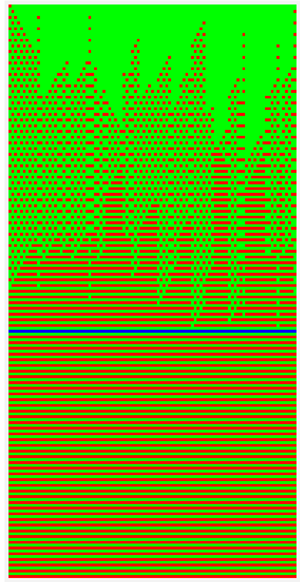
We see that increasing k lowers the synchronization time. This is because we are drastically increasing the number of edges in the graph. Increasing rewire probability, *p*, also decreases the sync time because of the added entropy into the graph. If rewire probability is low, we are left with a graph that is similar to a cycle graph, and as we saw, those took very long to synchronize. But when we mix a few edges around, it increases entropy and instead of the initial chirp having to wait for the effect to propagate around the entire cycle, a rewired edge can short circuit this chirp to the other side of the cycle and decrease sync time.

```
java Chirp s 101 200 small-101-k1-.2.png 1 34561 .2
Small-world V = 101, k = 1, p = 0.2:
```
- did not synchronize



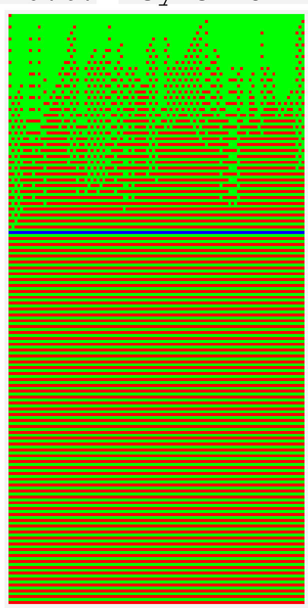
The network here did not synchronize due to the limited connectivity of the graph.

```
java Chirp s 101 200 small-101-k1-.3.png 1 45623 .3
Small-world V = 101, k = 1, p = 0.3:  synchronized at t=114.
```



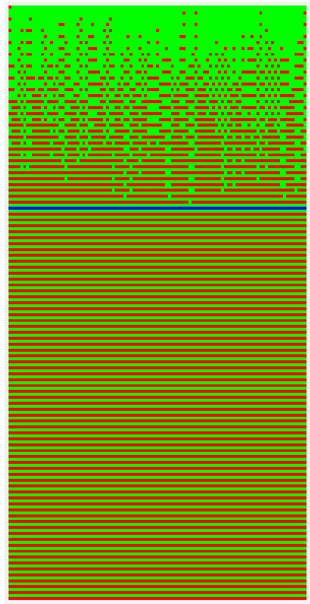
java Chirp s 101 200 small-101-k1-.6.png 1 65423 .6

Small-world V = 101, k = 1, p = 0.6: synchronized at t=74.



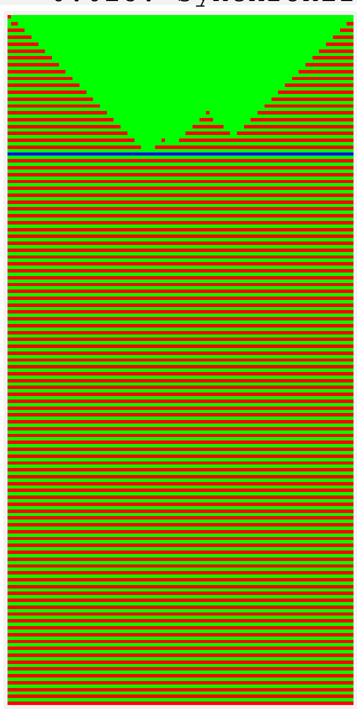
java Chirp s 101 200 small-101-k1-.9.png 1 32645 .9

Small-world V = 101, k = 1, p = 0.9: synchronized at t=68.

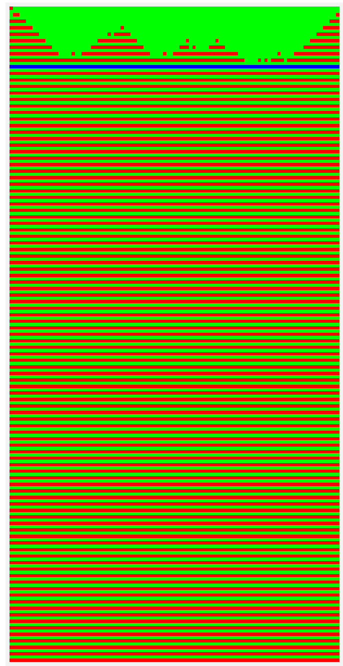


Above, we see the same effect we talked about with k=1 amplified with more vertices. The more edges we rewire, the more entropy we introduce into the network which decreases sync time because we increase the chances that 2 or more adjacent crickets will begin chirping at the same time.

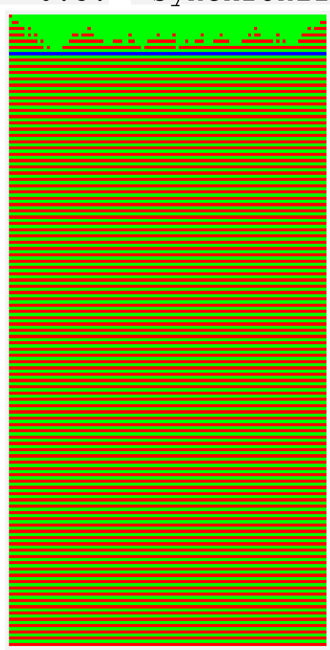
```
java Chirp s 101 200 small-101-k2-.025.png 2 234562 .025
Small-world V = 101, k = 2, p = 0.025: synchronized at t=40.
```



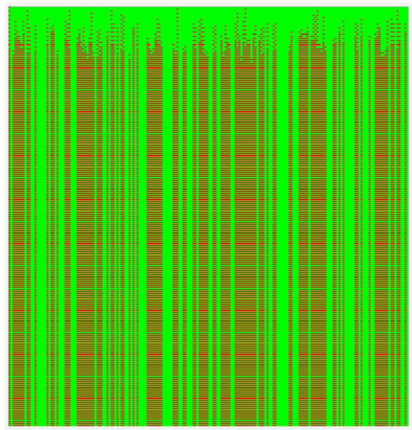
```
java Chirp s 101 200 small-101-k2-.05.png 2 234561 .05
Small-world V = 101, k = 2, p = 0.05:  synchronized at t=18.
```



```
java Chirp s 101 200 small-101-k2-.3.png 2 456230 .3
Small-world V = 101, k = 2, p = 0.3:   synchronized at t=12.
```



```
java Chirp s 200 400 small-200-k1-.9.png 1 32645 .9
Small-world V = 200, k = 1, p = 0.9:
```
- did not synchronize



```
java Chirp s 200 400 small-200-k1-1.png 1 26543 1
Small-world V = 200, k = 1, p = 1.0:  synchronized at t=112.
```

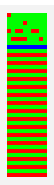
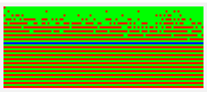
8. (Scale-free graph) Do the networks synchronize? If so, how long do the networks take to synchronize? Why are the networks behaving in this fashion?

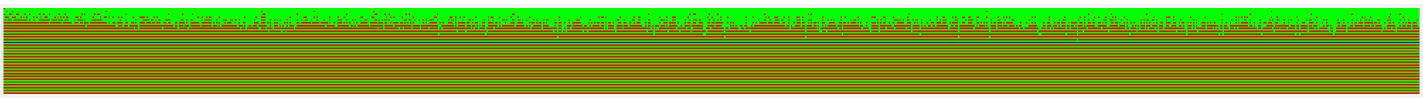
---

note: Some of these images are too large to include in this report.

```
java Chirp f 10 40 scale-free-10-1.png 1 19429
java Chirp f 100 40 scale-free-100-1.png 1 23456
java Chirp f 1000 60 scale-free-1000-1.png 1 238910
java Chirp f 10000 60 scale-free-10000-1.png 1 1938
```

```
Scale-free V = 10, dE = 1:        synchronized at t=8.
Scale-free V = 100, dE = 1:       synchronized at t=18.
Scale-free V = 1000, dE = 1:      synchronized at t=24.
Scale-free V = 10000, dE = 1:     synchronized at t=32.
```

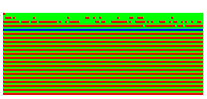
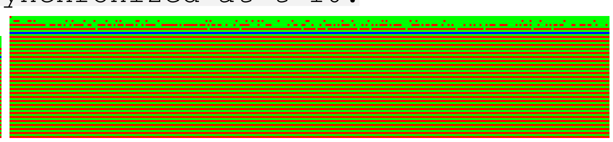
 

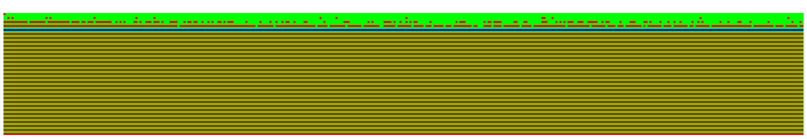


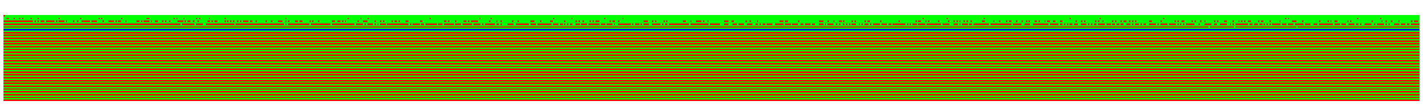
Observe an interesting phenomenon above. The

```
java Chirp f 100 40 scale-free-100-2.png 2 34567
java Chirp f 200 50 scale-free-200-2.png 2 02341
java Chirp f 300 60 scale-free-300-2.png 2 43829
java Chirp f 400 60 scale-free-400-2.png 2 33939
java Chirp f 1000 60 scale-free-1000-2.png 2 100283
```

```
Scale-free V = 100, dE = 2:       synchronized at t=8.
Scale-free V = 200, dE = 2:       synchronized at t=10.
Scale-free V = 300, dE = 2:       synchronized at t=8.
Scale-free V = 400, dE = 2:       synchronized at t=8.
Scale-free V = 1000, dE = 2:      synchronized at t=10.
```

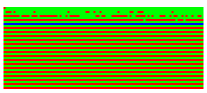
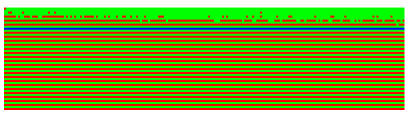
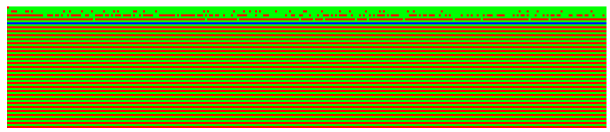
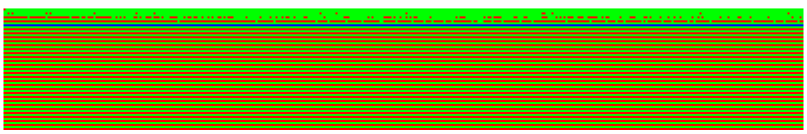
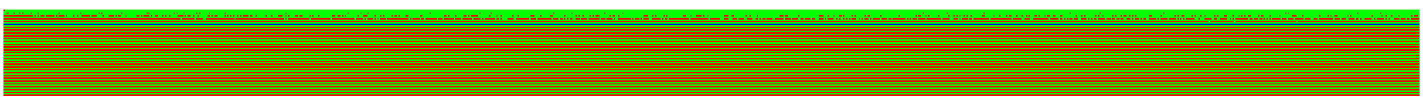




```
java Chirp f 100 40 scale-free-100-3.png 3 45678
java Chirp f 200 50 scale-free-200-3.png 3 34120
java Chirp f 300 60 scale-free-300-3.png 3 38294
```

```
java Chirp f 400 60 scale-free-400-3.png 3 39393
java Chirp f 1000 60 scale-free-1000-3.png 3 108392
java Chirp f 10000 12 scale-free-10000-3.png 3 19928
java Chirp f 50000 20 scale-free-50000-3.png 3 814832
```

```
Scale-free V = 100, dE = 3:       synchronized at t=8.
Scale-free V = 200, dE = 3:       synchronized at t=8.
Scale-free V = 300, dE = 3:       synchronized at t=8.
Scale-free V = 400, dE = 3:       synchronized at t=8.
Scale-free V = 1000, dE = 3:      synchronized at t=8.
Scale-free V = 10000, dE = 3:     synchronized at t=10.
Scale-free V = 50000, dE = 3:     synchronized at t=10.
```

9. Compare and contrast the results from Questions 6–8. Discuss what is causing the differences in behavior between random, small-world, and scale-free graphs.

---

10. Write a paragraph describing what you learned from this project.

Appendix A (Output):

```
######################################
# CRICKET CHIRPING AUTOMATION FILE   #
######################################
# author: Jimi Ford
# version: 3-12-15
######################################
```

```
# EVEN NUMBER VERTICES CYCLE GRAPHS #
#####################################

Cycle V = 2:  did not synchronize.
Cycle V = 4:  did not synchronize.
Cycle V = 6:  did not synchronize.
Cycle V = 8:  did not synchronize.
Cycle V = 10: did not synchronize.
Cycle V = 12: did not synchronize.
Cycle V = 14: did not synchronize.
Cycle V = 16: did not synchronize.
Cycle V = 200:       did not synchronize.

####################################
# ODD NUMBER VERTICES CYCLE GRAPHS #
####################################

Cycle V = 3:  synchronized at t=4.
Cycle V = 5:  synchronized at t=8.
Cycle V = 7:  synchronized at t=12.
Cycle V = 9:  synchronized at t=16.
Cycle V = 11: synchronized at t=20.
Cycle V = 13: synchronized at t=24.
Cycle V = 15: synchronized at t=28.
Cycle V = 101:       synchronized at t=200.
Cycle V = 201:       synchronized at t=400.

#################
# RANDOM GRAPHS #
#################

############## 20 CRICKETS #############

Random V = 20, p = 0.07:    did not synchronize.
Random V = 20, p = 0.08:    did not synchronize.
Random V = 20, p = 0.09:    did not synchronize.
Random V = 20, p = 0.1:     did not synchronize.
Random V = 20, p = 0.15:    synchronized at t=10.
Random V = 20, p = 0.2:     synchronized at t=8.
Random V = 20, p = 0.3:     synchronized at t=6.
Random V = 20, p = 0.4:     synchronized at t=4.
Random V = 20, p = 0.5:     synchronized at t=4.
Random V = 20, p = 0.6:     synchronized at t=4.
Random V = 20, p = 0.7:     synchronized at t=4.
Random V = 20, p = 0.8:     synchronized at t=4.
Random V = 20, p = 0.9:     synchronized at t=4.
Random V = 20, p = 1.0:     synchronized at t=4.

############# 200 CRICKETS #############
```

```
Random V = 200, p = 0.02:   did not synchronize.
Random V = 200, p = 0.03:   synchronized at t=10.
Random V = 200, p = 0.04:   synchronized at t=10.
Random V = 200, p = 0.05:   synchronized at t=8.
Random V = 200, p = 0.06:   synchronized at t=6.
Random V = 200, p = 0.07:   synchronized at t=6.
Random V = 200, p = 0.08:   synchronized at t=6.
Random V = 200, p = 0.09:   synchronized at t=6.
Random V = 200, p = 0.1:    synchronized at t=6.
Random V = 200, p = 0.2:    synchronized at t=4.
Random V = 200, p = 0.3:    synchronized at t=4.
Random V = 200, p = 0.4:    synchronized at t=4.
Random V = 200, p = 0.5:    synchronized at t=4.

######################
# SMALL-WORLD GRAPHS #
######################

######## 20 CRICKETS | k=1 #############

Small-world V = 20, k = 1, p = 0.05:    synchronized at t=32.
Small-world V = 20, k = 1, p = 0.1:     synchronized at t=30.
Small-world V = 20, k = 1, p = 0.2:     synchronized at t=28.
Small-world V = 20, k = 1, p = 0.3:     synchronized at t=32.
Small-world V = 20, k = 1, p = 0.4:     synchronized at t=30.
Small-world V = 20, k = 1, p = 0.5:     synchronized at t=34.
Small-world V = 20, k = 1, p = 0.5:     synchronized at t=16.
Small-world V = 20, k = 1, p = 0.6:     synchronized at t=20.
Small-world V = 20, k = 1, p = 0.7:     synchronized at t=30.
Small-world V = 20, k = 1, p = 0.8:     synchronized at t=22.
Small-world V = 20, k = 1, p = 0.9:     synchronized at t=24.
Small-world V = 20, k = 1, p = 1.0:     synchronized at t=20.

######## 20 CRICKETS | k=2 #############

Small-world V = 20, k = 2, p = 0.025:   synchronized at t=10.
Small-world V = 20, k = 2, p = 0.05:    synchronized at t=10.
Small-world V = 20, k = 2, p = 0.1:     synchronized at t=8.
Small-world V = 20, k = 2, p = 0.2:     synchronized at t=6.
Small-world V = 20, k = 2, p = 0.3:     synchronized at t=8.
Small-world V = 20, k = 2, p = 0.4:     synchronized at t=8.
Small-world V = 20, k = 2, p = 0.5:     synchronized at t=6.
Small-world V = 20, k = 2, p = 0.6:     synchronized at t=10.
Small-world V = 20, k = 2, p = 0.7:     synchronized at t=8.
Small-world V = 20, k = 2, p = 0.8:     synchronized at t=6.
Small-world V = 20, k = 2, p = 0.9:     synchronized at t=6.
Small-world V = 20, k = 2, p = 1.0:     synchronized at t=8.

######## 20 CRICKETS | k=3 #############
```

```
Small-world V = 20, k = 3, p = 0.025:    synchronized at t=8.
Small-world V = 20, k = 3, p = 0.05:     synchronized at t=8.
Small-world V = 20, k = 3, p = 0.1:      synchronized at t=8.
Small-world V = 20, k = 3, p = 0.2:      synchronized at t=6.
Small-world V = 20, k = 3, p = 0.3:      synchronized at t=6.
Small-world V = 20, k = 3, p = 0.4:      synchronized at t=6.
Small-world V = 20, k = 3, p = 0.5:      synchronized at t=6.
Small-world V = 20, k = 3, p = 0.6:      synchronized at t=6.
Small-world V = 20, k = 3, p = 0.7:      synchronized at t=6.
Small-world V = 20, k = 3, p = 0.8:      synchronized at t=6.
Small-world V = 20, k = 3, p = 0.9:      synchronized at t=6.
Small-world V = 20, k = 3, p = 1.0:      synchronized at t=6.

######## 101 CRICKETS | k=1 #############

Small-world V = 101, k = 1, p = 0.1:     did not synchronize.
Small-world V = 101, k = 1, p = 0.2:     did not synchronize.
Small-world V = 101, k = 1, p = 0.3:     synchronized at t=114.
Small-world V = 101, k = 1, p = 0.4:     did not synchronize.
Small-world V = 101, k = 1, p = 0.5:     did not synchronize.
Small-world V = 101, k = 1, p = 0.6:     synchronized at t=74.
Small-world V = 101, k = 1, p = 0.7:     did not synchronize.
Small-world V = 101, k = 1, p = 0.8:     did not synchronize.
Small-world V = 101, k = 1, p = 0.9:     synchronized at t=68.
Small-world V = 101, k = 1, p = 1.0:     did not synchronize.

######## 101 CRICKETS | k=2 #############

Small-world V = 101, k = 2, p = 0.025:   synchronized at t=40.
Small-world V = 101, k = 2, p = 0.05:    synchronized at t=18.
Small-world V = 101, k = 2, p = 0.1:     synchronized at t=16.
Small-world V = 101, k = 2, p = 0.2:     synchronized at t=14.
Small-world V = 101, k = 2, p = 0.3:     synchronized at t=12.
Small-world V = 101, k = 2, p = 0.4:     synchronized at t=12.
Small-world V = 101, k = 2, p = 0.5:     synchronized at t=12.
Small-world V = 101, k = 2, p = 0.6:     synchronized at t=12.
Small-world V = 101, k = 2, p = 0.7:     synchronized at t=12.
Small-world V = 101, k = 2, p = 0.8:     synchronized at t=12.
Small-world V = 101, k = 2, p = 0.9:     synchronized at t=12.
Small-world V = 101, k = 2, p = 1.0:     synchronized at t=12.

######## 101 CRICKETS | k=3 #############

Small-world V = 101, k = 3, p = 0.025:   synchronized at t=24.
Small-world V = 101, k = 3, p = 0.05:    synchronized at t=12.
Small-world V = 101, k = 3, p = 0.1:     synchronized at t=14.
Small-world V = 101, k = 3, p = 0.2:     synchronized at t=10.
Small-world V = 101, k = 3, p = 0.3:     synchronized at t=8.
Small-world V = 101, k = 3, p = 0.4:     synchronized at t=8.
Small-world V = 101, k = 3, p = 0.5:     synchronized at t=8.
```

```
Small-world V = 101, k = 3, p = 0.6:     synchronized at t=8.
Small-world V = 101, k = 3, p = 0.7:     synchronized at t=8.
Small-world V = 101, k = 3, p = 0.8:     synchronized at t=8.
Small-world V = 101, k = 3, p = 0.9:     synchronized at t=10.
Small-world V = 101, k = 3, p = 1.0:     synchronized at t=8.


######## 200 CRICKETS | k=1 #############

Small-world V = 200, k = 1, p = 0.1:     did not synchronize.
Small-world V = 200, k = 1, p = 0.2:     did not synchronize.
Small-world V = 200, k = 1, p = 0.3:     did not synchronize.
Small-world V = 200, k = 1, p = 0.4:     did not synchronize.
Small-world V = 200, k = 1, p = 0.5:     did not synchronize.
Small-world V = 200, k = 1, p = 0.6:     did not synchronize.
Small-world V = 200, k = 1, p = 0.7:     did not synchronize.
Small-world V = 200, k = 1, p = 0.8:     did not synchronize.
Small-world V = 200, k = 1, p = 0.9:     did not synchronize.
Small-world V = 200, k = 1, p = 1.0:     synchronized at t=112.

######## 200 CRICKETS | k=2 #############

Small-world V = 200, k = 2, p = 0.025:   synchronized at t=58.
Small-world V = 200, k = 2, p = 0.05:    synchronized at t=22.
Small-world V = 200, k = 2, p = 0.1:     synchronized at t=22.
Small-world V = 200, k = 2, p = 0.2:     synchronized at t=16.
Small-world V = 200, k = 2, p = 0.3:     synchronized at t=12.
Small-world V = 200, k = 2, p = 0.4:     synchronized at t=14.
Small-world V = 200, k = 2, p = 0.5:     synchronized at t=12.
Small-world V = 200, k = 2, p = 0.6:     synchronized at t=12.
Small-world V = 200, k = 2, p = 0.7:     synchronized at t=14.
Small-world V = 200, k = 2, p = 0.8:     synchronized at t=12.
Small-world V = 200, k = 2, p = 0.9:     synchronized at t=14.
Small-world V = 200, k = 2, p = 1.0:     synchronized at t=12.

######## 200 CRICKETS | k=3 #############

Small-world V = 200, k = 3, p = 0.025:   synchronized at t=24.
Small-world V = 200, k = 3, p = 0.05:    synchronized at t=16.
Small-world V = 200, k = 3, p = 0.1:     synchronized at t=12.
Small-world V = 200, k = 3, p = 0.2:     synchronized at t=12.
Small-world V = 200, k = 3, p = 0.3:     synchronized at t=10.
Small-world V = 200, k = 3, p = 0.4:     synchronized at t=10.
Small-world V = 200, k = 3, p = 0.5:     synchronized at t=10.
Small-world V = 200, k = 3, p = 0.6:     synchronized at t=8.
Small-world V = 200, k = 3, p = 0.7:     synchronized at t=8.
Small-world V = 200, k = 3, p = 0.8:     synchronized at t=10.
Small-world V = 200, k = 3, p = 0.9:     synchronized at t=10.
Small-world V = 200, k = 3, p = 1.0:     synchronized at t=10.
```

```
#####################
# SCALE-FREE GRAPHS #
#####################


# f 10 15 scale-free-010-2.png 2 12345
######## dE=1 ########################
Scale-free V = 10, dE = 1:  synchronized at t=8.
Scale-free V = 100, dE = 1:       synchronized at t=18.
Scale-free V = 1000, dE = 1:      synchronized at t=24.
Scale-free V = 10000, dE = 1:     synchronized at t=32.
# f 50000 60 scale-free-50000-1.png 1 138
# synchronized at t=36
######## dE=2 ########################
Scale-free V = 100, dE = 2:       synchronized at t=8.
Scale-free V = 200, dE = 2:       synchronized at t=10.
Scale-free V = 300, dE = 2:       synchronized at t=8.
Scale-free V = 400, dE = 2:       synchronized at t=8.
Scale-free V = 1000, dE = 2:      synchronized at t=10.
######## dE=3 ########################
Scale-free V = 100, dE = 3:       synchronized at t=8.
Scale-free V = 200, dE = 3:       synchronized at t=8.
Scale-free V = 300, dE = 3:       synchronized at t=8.
Scale-free V = 400, dE = 3:       synchronized at t=8.
Scale-free V = 1000, dE = 3:      synchronized at t=8.
Scale-free V = 10000, dE = 3:     synchronized at t=10.
Scale-free V = 50000, dE = 3:     synchronized at t=10.
```

Appendix B (Source code):

```java
import java.io.IOException;

/**
 *
 * @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");
        }
```

```java
    }

    /**
     * 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();
    }
}
```

```java
import java.io.IOException;

/**
 *
 * @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,
                             K_SEED_INDEX = 5,
                             REWIRE_PROBABILITY_INDEX = 6;

    public static void main(String[] args) {
        if(args.length != 4 && args.length != 5 &&
                args.length != 6 && args.length != 7) usage();
        int crickets = 0, ticks = 0, k = 0, dE = 0;
        long seed = 0;
        double prob = 0;
        char mode;
        String outputImage = args[OUTPUT_IMAGE_INDEX];

        try {
            crickets = Integer.parseInt(args[NUM_VERTICES_INDEX]);
        } catch (NumberFormatException e) {
            error("<num vertices> must be a number");
        }
        try {
            ticks = Integer.parseInt(args[NUM_TICKS_INDEX]) + 1;
        } catch (NumberFormatException e) {
            error("<num ticks> must be numeric");
        }
        mode = args[GRAPH_TYPE_INDEX].toLowerCase().charAt(0);
```

```java
        if(!(mode == 'c' || mode == 'r' || mode == 'k' ||
                mode == 's' || mode == 'f')) {
            error("<graph type> must be either 'c' for cycle, "
                    + "'r' for random, "
                    + "'k' for k-regular, "
                    + "'s' for small-world, "
                    + "'f' for scale-free");
        }
        UndirectedGraph g = null;
        CricketObserver o = new CricketObserver(crickets, ticks);
        switch(mode) {
        case 'r': // RANDOM GRAPH
            try {
                seed = Long.parseLong(args[SEED_INDEX]);
                prob = Double.parseDouble(args[EDGE_PROBABILITY_INDEX]);
                g = UndirectedGraph.randomGraph(new Random(seed), crickets, prob, o);
            } catch(NumberFormatException e) {
                error("<seed> and <edge probability> must be numeric");
            } catch(IndexOutOfBoundsException e) {
                error("<seed> and <edge probability> must be included with random graph mode");
            }
            break;
        case 'c': // CYCLE GRAPH
            g = UndirectedGraph.cycleGraph(crickets, o);
            break;
        case 'k': // K-REGULAR GRAPH
            try {
                k = Integer.parseInt(args[K_INDEX]);
                g = UndirectedGraph.kregularGraph(crickets, k, o);
            } catch (NumberFormatException e) {
                error("<k> must be an integer");
            } catch (IllegalArgumentException e) {
                error("<k> must be < the number of crickets");
            }
            break;
        case 's': // SMALL WORLD GRAPH
            try {
```

```
                    k = Integer.parseInt(args[K_INDEX]);
                    prob = Double.parseDouble(args[REWIRE_PROBABILITY_INDEX]);
                    seed = Long.parseLong(args[K_SEED_INDEX]);
                    g = UndirectedGraph.smallWorldGraph(new Random(seed), crickets, k, prob, o);
                } catch (NumberFormatException e) {
                    error("<k> must be an integer < V, <rewire probability> must be a number "
                            + "between 0 and 1, and <seed> must be numeric");
                } catch (IllegalArgumentException e) {
                    error("<k> must be < the number of crickets");
                }
                break;
            case 'f':
                try {
                    dE = Integer.parseInt(args[DE_INDEX]);
                    seed = Long.parseLong(args[DE_SEED_INDEX]);
                    g = UndirectedGraph.scaleFreeGraph(new Random(seed), crickets, dE, o);
                } catch (NumberFormatException e) {
                    error("<dE> and <seed> must be numeric");
                } catch (IndexOutOfBoundsException e) {
                    error("<dE> and <seed> must be supplied");
                }
        }

        g.vertices.get(0).forceChirp();
        Ticker.tick(g, ticks);



        try {
            ImageHandler.handle(o, outputImage);
        } catch (IOException e) {
            error("Problem writing image");
        }
        int sync = o.sync();
        String description;
        switch(mode) {
```

```java
        case 'c': // CYCLE GRAPH
            description = "Cycle V = " + crickets +":";
            handleOutput(description,sync);
            break;
        case 'r': // RANDOM GRAPH
            description = "Random V = " + crickets +", p = " + prob + ":";
            handleOutput(description,sync);
            break;
        case 'k': // K-REGULAR GRAPH
            description = "K-regular V = " + crickets +", k = " + k + ":";
            handleOutput(description,sync);
            break;
        case 's': // SMALL-WORLD GRAPH
            description = "Small-world V = " + crickets + ", k = " + k +
                ", p = " + prob + ":";
            handleOutput(description,sync);
            break;
        case 'f': // SCALE-FREE GRAPH
            description = "Scale-free V = " + crickets +", dE = " + dE + ":";
            handleOutput(description,sync);
            break;
        }

    }

    private static void handleOutput(String description, int sync) {
        System.out.print(description);
        if(sync >= 0) {
            System.out.println("\t"+" synchronized at t="+sync+".");
        } else {
            System.out.println("\t "+(char)27+"[31m"+  "did not synchronize." +
                    (char)27 + "[0m");
        }
    }

    private static void error(String msg) {
```

```
        System.err.println(msg);
        usage();
    }

    private static void usage() {
        System.err.println("usage: java Chirp <graph type> <num vertices> <num ticks> "
                + "<output image> {(<seed> <edge probability>), or "
                + "(<k>), or "
                + "(<k> <seed> <rewire probability>), or "
                + "(<dE> <seed>)}");
        System.exit(1);
    }
}
```

```java

public class Cricket extends Vertex {

//  private boolean[] chirp = new boolean[3];
    private boolean[] chirp = new boolean[2];
    private boolean willChirp;
    private int currentTick = 0;
    private final CricketObserver observer;

    public Cricket(int n, CricketObserver o) {
        super(n);
        this.observer = o;
    }

    public void forceChirp() {
        willChirp = chirp[0] = true;
    }

    public void emitChirp() {
        if(willChirp) {
            willChirp = false;
            int n = super.degree();
            for(int i = 0; i < n; i++) {
                edges.get(i).other(this).hearChirp();
            }
            observer.reportChirp(currentTick, super.n);
        }
    }

    private void hearChirp() {
        chirp[1] = true;
    }

    public void timeTick(int tick) {
        currentTick = tick;
        willChirp = chirp[0];
        chirp[0] = chirp[1];
//      chirp[1] = chirp[2];
//      chirp[2] = false;
        chirp[1] = false;
    }
```

```java

    public boolean directFlight(Cricket other) {
        boolean retval = false;
        if(equals(other)) return true;
        int e = super.degree();
        Cricket o;
        for(int i = 0; i < e && !retval; i++) {
            o = super.edges.get(i).other(this);
            retval = o.equals(other);
        }
        return retval;
    }

    public boolean equals(Object o) {
        if( !(o instanceof Cricket)) {
            return false;
        }
        if(o == this) {
            return true;
        }
        Cricket casted = (Cricket) o;

        return casted.n == this.n;
    }
}

```

```java

public class CricketObserver {

    public final int crickets, ticks;
    private boolean[][] chirps;

    public CricketObserver(int crickets, int ticks) {
        this.crickets = crickets;
        this.ticks = ticks;
        chirps = new boolean[ticks][crickets];
    }

    public void reportChirp(int tick, int n) {
        chirps[tick][n] = true;
    }

    public boolean chirped(int tick, int cricket) {
        return chirps[tick][cricket];
    }

    public int sync() {
        int row = 0;
        while(row < ticks) {
            if(sync(row)) return row;
            row++;
        }
        return -1;
    }

    private boolean sync(int tick) {
        boolean retval = true;
        for(int i = 0; i < crickets && retval; i++) {
            retval = chirps[tick][i];
        }
        return retval;
    }

//  private boolean equal(boolean[] a, boolean[] b) {
//      boolean retval = true;
//      if(a.length == b.length) {
//          for(int i = 0; i < a.length && retval; i++) {
```

```
//              retval = a[i] == b[i];
//          }
//      }
//      return retval;
//  }
}
```

```java
import java.io.BufferedOutputStream;


public class ImageHandler {

    public static final byte SILENT = 0,
                             CHIRPED = 1,
                             SYNC = 2;

    public static void handle(CricketObserver o, String out) throws FileNotFoundException {
        AList<Color> palette = new AList<Color>(); // green
        Color green = new Color().rgb(0, 255, 0);
        Color red = new Color().rgb(255, 0, 0); // red
        Color blue = new Color().rgb(0,0,255); // blue
        palette.addLast (green);
        palette.addLast (red);
        palette.addLast (blue);


        OutputStream imageout =
                new BufferedOutputStream (new FileOutputStream (new File(out)));
        IndexPngWriter imageWriter = new IndexPngWriter
                (o.ticks, o.crickets, imageout, palette);
        ByteImageQueue imageQueue = imageWriter.getImageQueue();
        byte[] bytes;
        boolean chirped;
        int sync = o.sync();
        for(int i = 0; i < o.ticks; i++) {
            bytes = new byte[o.crickets];
            for(int j = 0, cricket = 0; j < bytes.length; j++, cricket++) {
                if(i != sync) {
                    chirped = o.chirped(i, cricket);
                    bytes[j] = chirped ? CHIRPED : SILENT;
                } else {
                    bytes[j] = SYNC;
                }
            }
            try {
```

```java
                imageQueue.put(i, bytes);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        try {
            imageWriter.write();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

```java

public class Ticker {

    public static void tick(UndirectedGraph g, int ticks) {
        for(int i = 0; i < ticks; i++) {
            g.tick(i);
        }
    }
}

```

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

/**
 * 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) {
```

```java
            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);
    }

    /**
     * Get the <I>other</I> vertex given a certain vertex connected to
     * this edge
     *
     * @param current the current vertex
     * @return the other vertex connected to this edge
     */
    public Cricket other(Cricket current) {
        if(current == null) return null;
        return current.n == a.n ? b : a;
    }
}
```

```java
// File:    UndirectedGraph.java

import java.util.ArrayList;

/**
 * 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;

    // Prevent construction
    private UndirectedGraph() {

    }

    /**
     * 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));
        }
    }

```

```java
    /**
     * 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) {
```

```java
                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);
                }
            }
        }
    }

    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();
        }
    }
```

```java

    /**
     * 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;
    }

    public static UndirectedGraph cycleGraph(int v, CricketObserver o) {
        return kregularGraph(v, 1, o);
    }

    public static UndirectedGraph kregularGraph(int v, int k, CricketObserver o) {
        return smallWorldGraph(null, v, k, 0, o);
```

```java
    }

    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;
    }

    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
        int[] prob;
        Cricket next, temp;
        ArrayList<Cricket> existing = new ArrayList<Cricket>();
        existing.add(a); existing.add(b); existing.add(c);
        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 {
                // potential bug - when do i add in the current vertex to the
                // probability distribution?
                int sumD = sumDeg(g);
                prob = new int[sumD];
                setProbabilityDistribution(g, prob);
                for(int e = 0; e < dE; e++) {
                    do {
                        int chosen = (int) Math.floor(prng.nextDouble() * prob.length);
                        temp = g.vertices.get(prob[chosen]);
                    } while(next.directFlight(temp));
                    edge = new UndirectedEdge(edgeCount++, next, temp);
                    g.edges.add(edge);
```

```
                }
            }
        }

        return g;
    }

    private static void setProbabilityDistribution(UndirectedGraph g, int[] prob) {
        Vertex v;
        int degree = 0;
        int counter = 0;
        for(int i = 0; i < g.v; i++) {
            v = g.vertices.get(i);
            degree = v.degree();
            for(int j = counter; j < degree + counter; j++) {
                prob[j] = v.n;
            }
            counter += degree;
        }
    }

    private static int sumDeg(UndirectedGraph g) {
        int retval = 0;
        Vertex v;
        for(int i = 0; i < g.v; i++) {
            v = g.vertices.get(i);
            retval += v.degree();
        }
        return retval;
    }
}
```

```java
// File:    Vertex.java

import java.util.ArrayList;

/**
 * Class Vertex represents a single vertex in a graph. Vertices can be connected
 * to other vertices through undirected edges.
 *
 * @author Jimi Ford
 * @version 2-15-2015
 */
public class Vertex {

    // private data members
    protected ArrayList<UndirectedEdge> edges = new ArrayList<UndirectedEdge>();

    /**
     * The unique identifier for this vertex
     */
    public final int n;

    /**
     * Construct a vertex with a unique identifier <I>n</I>
     *
     * @param n the unique identifier to distinguish this vertex from
     *          all other vertices in the graph
     */
    public Vertex(int n) {
        this.n = n;
    }

    /**
     * Get the number of edges connected to this vertex
     *
     * @return the number of edges connected to this vertex
     */
    public int degree() {
        return edges.size();
    }
```

```java

    /**
     * Get the reference to the collection of edges connected to
     * this vertex.
     *
     * @return the reference to the collection of edges
     */
    public ArrayList<UndirectedEdge> getEdges() {
        return this.edges;
    }

    /**
     * Add an edge to this vertex
     *
     * @param e the edge to add
     */
    public void addEdge(UndirectedEdge e) {
        this.edges.add(e);
    }

    /**
     * Compare another object to this one
     *
     * @param o the other object to compare to this one
     * @return true if the other object is equivalent to this one
     */
    public boolean equals(Object o) {
        if( !(o instanceof Vertex)) {
            return false;
        }
        if(o == this) {
            return true;
        }
        Vertex casted = (Vertex) o;

        return casted.n == this.n;
    }
}

```