CS 1110, ASSIGNMENT 4: SOCIAL INFLUENCE

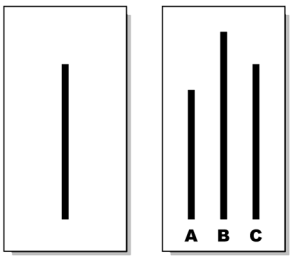
L. LEE AND S. MARSCHNER

Due on CMS on **April 4th at 11:59pm**.
Most recent version available online at:
http://www.cs.cornell.edu/courses/cs1110/2013sp/assignments/assignment4/assignment4.pdf .
These instructions last updated on Monday 25th March, 2013 at 14:40.
Any changes from the handout will be marked in orange and enumerated here on the online version.

1. Preliminaries

Can we computationally model and predict how (and if) people's opinions are affected by those around them? If by "we" we mean people in the emerging and exciting area of computational social science (an area where Cornell can be argued to be #1 in the world), then the answer is "that's an important and active research area, and we're working on it". If by "we" we mean students in CS1110, the answer is ... still "yes"! We've already learned enough in this course to try some simulation experiments, at least as long as we employ some simplifying assumptions.[1]

Diffusion has been an important topic in the social sciences for a long time. One basic question is, how much does the probability of someone adopting a behavior depend on how many of their social contacts have already adopted that behavior? Applications include understanding how opinions form and spread, how actions (recycling, smoking) are promoted, and how collective action occurs. Some of the most striking examples of a social-contagion effect are the infamous Asch social-conformity experiments from the 1950s, where a surprisingly large fraction of people would go along with a unanimous but unambiguously wrong opinion instead of the evidence of their own eyes:



Q: Which of A, B, or C has the same length as the line on the card to the left?

College-student subject, wearing glasses, leaning forward to look more closely at the cards as many of the other (planted-confederate) students gave incorrect answers.

Image sources: http://en.wikipedia.org/wiki/Asch_conformity_experiments; http://www.age-of-the-sage.org/psychology/social/asch_conformity.html

[1] Among them: It's best for us to assume interaction structures that don't have any "re-entry" or "cycles".

1.1. **Learning Objectives.** This assignment is designed to give you (more) practice with the following skills and ideas:

- writing class definitions from specifications
- writing a recursive function with *memoization* to make the computation more efficient
- writing simple for-loops
- working with nested lists
- working with sets
- working with larger coding projects
- using simulation to examine a (social-)scientific question

1.2. **Collaboration and Academic Integrity Policy.** You may do this assignment with one other person. If you are going to work together, then form your group on CMS before submitting. Both parties must perform a CMS action to form the group: The first person proposes, and then the other accepts. Once you've grouped on CMS, only one person submits the files.

If you do this assignment with another person, you must work together. It is against the rules for one person to do some programming on this assignment without the other person sitting nearby and helping.

With the exception of your CMS-registered partner, you may not look at anyone else's code or show your code to anyone else, in any form or manner whatsoever.

1.3. **Getting Help.** If you do not know where to start, find your progress has stalled, or become lost along the way, please see someone immediately; a little in-person help can do wonders. See the staff page for more information on office hours, consulting hours, and making appointments. Piazza is also an excellent resource for getting questions answered quickly (although you are not allowed to post your code there).

1.4. **Files to download.** Create a *new* directory on your hard drive and download the following zip file:
http://www.cs.cornell.edu/courses/cs1110/2013sp/assignments/assignment4/a4.zip)
It contains two files, node.py and trial.py, that you are to complete and submit, plus some unit tests that you are free to modify and use, but will not submit.

2. The CS1110 A4 model

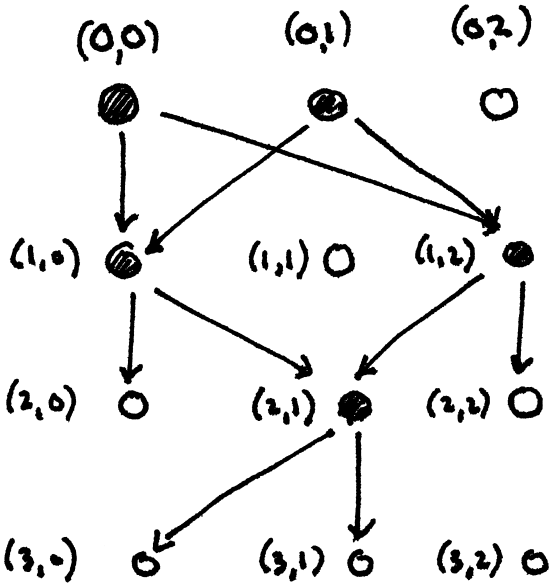
The model you will be implementing is one that we designed to be conceptually simple — to make things more feasible (for you to implement and for us to grade) — but still draws inspiration from classic social-science diffusion models, and which already exhibits interesting properties. One situation represented by this model is where a young person of voting age joins a political-action group if they see two role models do so. Another situation represented by this model is where someone believes a rumor if they hear it from three more senior (authoritative?) people.[2]

2.1. **The ".." integer-range notation.** To describe our setup more formally, it's convenient to set up the following notation. When you see "$a..b$" in this course, assume that a and b are integers and $b \geq a - 1$. Then, "$a..b$" denotes the interval of integers from a and b inclusive. If $b = a$, then this interval is the single integer a. If $b = a - 1$, then this interval is the empty set. The number of elements in $a..b$ is $b + 1 - a$ ("followers minus first").

[2]This model is not a good fit for the Asch experiments. We contemplated also having you implement a subclass that models the Asch social-conformity effect more closely, but elected not to in order to keep the assignment relatively short. The basic idea (yet another Assignment Part that Never Was, for those of you keeping track of what happened with A3) is that both converted and non-converted nodes would contact the next generation, and conversion would occur if and only if $\geq t\%$ of the contacters were converted.

This notation, used in several programming languages, including Perl and Pascal, is very convenient when talking about integers in a range. But don't confuse it with Python's `range` function: "*a*..*b*" in dot-dot notation corresponds to `range(a,b+1)` in Python.

2.2. **Formal description of a trial.** Our simulations will be represented by objects of type *Trial*. Here is a picture of the general idea. Each row represents a generation, earliest towards the top. Some of the nodes (people) in generation 0 are initialized to be *converted*: this could mean that they believe a new idea, or engage in some behavior, whereas the other nodes in that generation don't. We represent converted nodes by filled-in circles. Then, each converted node simultaneously contacts some members of the next generation, as indicated by the arrows. If the number of people contacting a node exceeds some threshold t, then that node is itself converted. In the example below, $t = 2$, and you can see that the idea or behavior in question eventually died out.



Specifically, a *trial* consists of g generations, numbered $0..g-1$, each of which contains n distinct nodes numbered $0..n-1$, for $g \times n$ distinct nodes altogether. Nodes $0..c-1$ of generation 0 are initialized to be converted, and the rest of the nodes are initialized to be not converted. In the above figure, $g = 4, n = 3$, and $c = 2$.

The simulation proceeds generation by generation, with each generation affecting the next generation by individual nodes in generation g contacting some of the nodes in generation $g+1$, except that generation $g-1$ does not do any further contacting. Specifically, when it is a node's turn to contact nodes in the next generation, it randomly selects exactly d distinct next-generation nodes. In the figure above, $d = 2$.

We claim that the behavior of this kind of system is not obvious, and that it is not *a priori* clear whether it is predictable. What configurations lead to a majority of the nodes being converted? Can one predict what fraction of the nodes are eventually converted? What is the chance that the idea or behavior survives until the end of the simulation?

3. Completing Classes Node and Trial

3.1. **node.py.** First, complete the file node.py according to the given specifications, using the incremental, write-a-bit-then-test-before-proceeding paradigm we have been stressing in this course. You will probably write at least one simple for-loop; you can consult the __init__ method in module trial for examples.

3.2. **Experimental finding: the percentage-converted in each generation reaches a fixed point.** Next, implement the methods _propagate_row and frac_of_gen in trial.py. When you have this working, you are ready to engage in some simulation experiments!

In particular, you can run the module testtrial. The function test_against_prediction in that module is interesting. It uses simulation to address the following question: does the percentage of the population that is converted eventually converge to a predictable "fixed point"? The answer is, perhaps surprisingly, "yes", at least for the cases covered by this simulation. (Predicting what this fixed point is requires some knowledge of probability. We can provide the analysis to the curious upon demand.)

3.3. **Function legacy: memoized recursion.** This method involves the most new concepts in this assignment.

We say that the *legacy* of a converted node *nd* is the set of converted nodes in later generations that *nd* can be said to have had a hand in converting. In the figure above, the legacy of $(0, 0)$ is the set $(1, 0), (1, 2)$, and $(2, 1)$. (In other trials, the legacy of a node might not be the set of all converted nodes in later generations.)

You must implement the function that computes a node's legacy in a recursive fashion. However, the most straightforward way to do this turns out to be very costly in terms of memory usage. So, you will employ the technique of *memoization*: you will pass around an object that stores already-computed values of the legacy function, so that you don't have to compute such values (recursively) again.

For example, in the process of computing the legacy of $(1, 0)$, you may determine the legacy of $(2, 1)$. You then store that set in memotable[2][1]. Then, when determining the legacy of $(1, 2)$, you check memotable[2][1] and see that you already computed $(2, 1)$s legacy, and so don't bother re-computing it (recursively).

Another new concept in implementing the legacy function is the set type. Sets in Python are similar to lists, but sets do not contain duplicates. So, it is useful to store sets in our memotable instead of lists (otherwise, the lists would get very big and memotable would get hard for us humans to look at).

- To create an empty set: `set()`
- To add an item x to set myset: `myset.add(x)`
- To make a set myset out of a list mylist (duplicates will be removed): `myset = set(mylist)`
- To assign to variable myset the set of items that are in either set s and set t: `myset = s.union(t)`. This is a way to add multiple items to a set, and have duplicates removed.

See the Python documentation for more information about the set type.

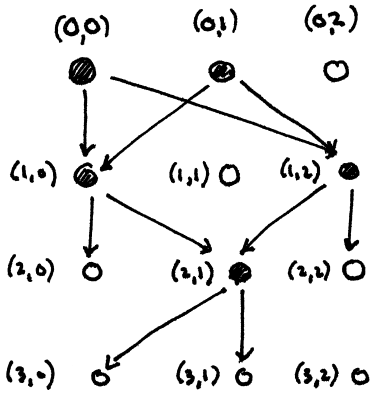
3.4. **Experimental Finding: "phase transitions" in legacy size.** Once your legacy-computation method is working, you can run the application code in module trial.[3] That code checks what the average legacy size is over the initially-converted generation-0 nodes. You'll see that some perhaps seemingly-small changes in parameter values lead to very different average generation-0 legacy sizes: sometimes, almost none of the population is converted, whereas other times, most of the population is converted.

4. Testing and Debugging in this Assignment

You may find debugging challenging in this assignment. Below, we offer some suggestions.

[3]At least, we hope you can. It is possible that the computational resources required for these simulations might exceed what your computer has available; in which case, you'll just have to believe what we say here.

4.1. **Use the print/__str__ functions we gave you.** For this assignment, these functions are probably going to be the most useful in your debugging and testing arsenal. We have provided several methods and code snippets for printing out nodes, trials, memotables, and so on. So, use these to see what your code does. See below right for the output of printing the trial created by function create_sample in module trial. (The "average % converted" is not updated by the code in create_sample, and so is not correct.)




```
>>> import trial
>>> x = trial.create_sample()
>>> print x
g=4 n=3 c=2, d=2 t=2 avg % converted=0.0
...
(0, 0). d=2 t=2 init_converted=True is_converted=True
contacted by nodes:
contacted: (1,0), (1,2)
(0, 1). d=2 t=2 init_converted=True is_converted=True
contacted by nodes:
contacted: (1,0), (1,2)
(0, 2). d=2 t=2 init_converted=False is_converted=False
contacted by nodes:
contacted:
---
(1, 0). d=2 t=2 init_converted=False is_converted=True
contacted by nodes: (0,0), (0,1)
contacted: (2,0), (2,1)
(1, 1). d=2 t=2 init_converted=False is_converted=False
contacted by nodes:
contacted:
(1, 2). d=2 t=2 init_converted=False is_converted=True
contacted by nodes: (0,0), (0,1)
contacted: (2,1), (2,2)
---
(2, 0). d=2 t=2 init_converted=False is_converted=False
contacted by nodes: (1,0)
contacted:
(2, 1). d=2 t=2 init_converted=False is_converted=True
contacted by nodes: (1,0), (1,2)
contacted: (3,0), (3,1)
(2, 2). d=2 t=2 init_converted=False is_converted=False
contacted by nodes: (1,2)
contacted:
---
(3, 0). d=2 t=2 init_converted=False is_converted=False
contacted by nodes: (2,1)
contacted:
(3, 1). d=2 t=2 init_converted=False is_converted=False
contacted by nodes: (2,1)
contacted:
(3, 2). d=2 t=2 init_converted=False is_converted=False
contacted by nodes:
contacted:
---
```

4.2. **Use the test cases we provided.** Look in the provided unit tests. Also, you may wish to make use of the function create_sample in module trial to check your legacy function on.

5. Finishing the Assignment

Submit your node.py and trial.py on CMS, after making sure that they meet the class coding conventions. (Do *not* submit your unit tests.) In particular, you should check that the following are all true:

- There are no tabs in the file, only spaces (this is usually not a problem).
- Functions are separated by two blank lines.
- Lines are short enough that horizontal scrolling is not necessary; about 80 chars is long enough. (You can go to the Smart Editing section of Preferences in Komodo Edit to be able to visually see the 80-character boundary: click the Draw the edge line column box with the value Edge line column box set to 80.)
- You have deleted all lines of the form pass or comments of the form implement me
- At the top of each module that you worked on you have three single-line comments with (1) the module name, (2) your name(s) and netid(s), and (3) the date you finished the assignment.