CS 1110, ASSIGNMENT 3: SOCIAL INFLUENCE

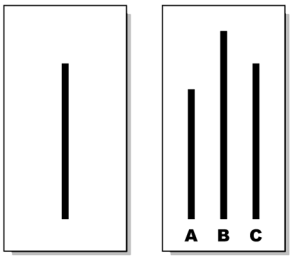
L. LEE AND S. MARSCHNER

Due on CMS on **March 28th at 11:59pm**.
Most recent version available online at:
http://www.cs.cornell.edu/courses/cs1110/2014sp/assignments/assignment3/assignment3.pdf .
These instructions last updated on Thursday 20th March, 2014 at 22:31.
Any updates will be marked in orange and enumerated and dated here.
Thu Mar 20, 10:30pm: linenumbers added for easier reference. Minor typo fix around line 342.

1. Preliminaries

> *...I'm amazed at how you were able to use python to model a psychological study, and I now see the full extent of what you said in the beginning of the semester about how the only limits of what we could do with python is our own imagination. I just hope to be able to reach that level of creative thinking very soon.*
> —student from a previous semester

One of the most striking examples of an apparent social-influence effect is the infamous Asch set of social-conformity experiments in the 1950s, where a surprisingly large fraction of people would go along with a strong-majority but unambiguously wrong opinion instead of the evidence of their own eyes. Specifically, a group of "test subjects" were shown the cards and asked the question given below left.



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

Clearly, the answer is C. However, unbeknownst to the *real* test subject, all the other people in the room had agreed ahead of time that most of them would give the same incorrect answer. Under these circumstances, in a disquietingly large percentage of cases, the real subject went along with the obviously wrong majority.[1]

[1] We could try this with an iClicker exercise sometime.

Now, let's ask the question, can we *mathematically* or *computationally* model and predict how people's opinions are affected by those around them? Applications include understanding how opinions form and spread, how actions (both "good" ones like recycling and "bad" ones like smoking) are promoted, and how collective action occurs.

If by "we" in the paragraph above we mean people in the emerging and exciting area of computational social science (an area in which Cornell can be argued to be #1 in the world), then the answer is "that's an important and active research question, 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 run some simulation experiments that get at this question, at least as long as we employ some simplifying assumptions.[2]

Contents

1.1. Learning objectives.

This assignment involves:

- writing a recursive function for computation on a specified data structure
- using *caching* (storage of pre-computed results) to make recursion more efficient
- writing class definitions from specifications
- writing for-loops
- working with nested lists

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

- working with *sets*, which allow us to ignore ordering between items
- working with larger coding projects
- using visualization to debug (we've provided the visualizer)
- using simulation to examine scientific questions

1.2. **Collaboration and academic integrity policy.** You may do this assignment with one other person. Regardless of whether you have grouped with that person for previous assignments or not, if you are going to work together with someone on this assignment, then form your group on CMS for this assignment 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.

The academic-integrity policy description on the course website has been revised, with particular emphasis on how to comment your code to give credit appropriately. Be sure to consult this new description before submitting.

1.3. **Files to download.** Create a *new* directory on your hard drive and download into it the following zip file:

http://www.cs.cornell.edu/courses/cs1110/2014sp/assignments/assignment3/a3.zip

It contains two files, node.py and trial.py, that you are to complete and submit. It also contains two unit tests and a visualization module; you are free to modify and use these three files, but don't submit them.

1.4. **Managing your time.** Don't panic over the length of this assignment writeup! There actually isn't very much code to write. We do suggest you plan so as to have at least two days to work on the very last section of this assignment (§6) — not because it's long, but because it's subtle. This should be easily accomplished if you make progress every day or so.

1.5. **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: you can learn from other people's questions and posts and get your own questions answered quickly (although you should not post your actual code there).

2. The CS1110 A3 Model

The model you will be implementing draws inspiration from classic social-science diffusion models. It is simple, but even so already exhibits interesting properties. It models the idea that each older generation can directly influence the next generation; and the question we're interested in learning from this model is, how long does a generation's *indirect* influence last through successive generations?

One scenario 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 scenario represented by this model is where someone believes a rumor if they hear it from three more senior (authoritative?) people. A third scenario is the running example we'll take throughout this assignment writeup:

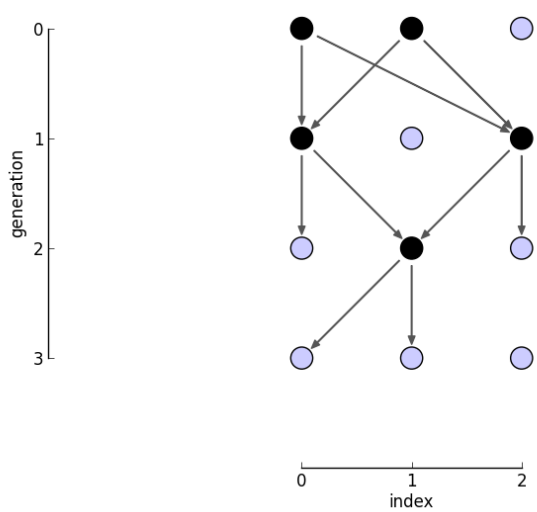
We imagine that all the students in CS1110 this semester collectively represent "Generation 0" of Cornell students, and some of you catch the CS1110 bug and are thus *converted* into CS1110 evangelists. The converted students in Generation 0 contact various students in the next generation, Generation 1, to convince those young'uns to become CS1110 converts. But students in Generation 1 are not too easily swayed; each only becomes converted if at least t converted members of Generation 0 contact them. Next, Generation 2 comes along. The converts in Generation 1 contact various students in Generation 2, and again, the students in Generation 2 are wary, so each one only becomes converted if at least t converted members of Generation 1 contact them. And so on.

The question then is, in the far future, does everyone become a CS1110 convert? Or does the idea of CS1110 being excellent eventually die out? Or does the fraction of CS1110 converts in each generation converge to some fixed percentage? Or does that fraction go in waves, rising and falling but never converging? And how does the answer depend on the initial conditions — how many initial converts, how many people per generation, what t is, etc.?

2.1. **The ".." integer-range notation.** To describe our setup more formally, it's convenient to set up the following mathematical 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 $b + 1 - a$ (a handy mnemonic, told to us by Prof. David Gries: "follower minus first").

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 double-dot notation corresponds to `range(a,b+1)` in Python.

2.2. **Formal description of a trial.** We're going to empirically investigate the future fate of CS1110 by simulating the contacting process to see what happens. Our simulations will be represented by objects of type *Trial*, and the people in the simulations will be represented by objects of type *Node*. Below is a picture of the general idea. A trial contains a sequence of successive *generations*, each containing the same number of people; generations are depicted as rows of circles in the picture below, generation 0 at the top. To tell the nodes in a generation apart, we number each node by an *index*. Some of the nodes (people) in generation 0 are *initialized* to be *converted*. We draw converted and non-converted nodes by black and non-black circles, respectively, so in the picture, the generation-0 nodes at index 0 and 1 are initially converted, but the node at index 2 isn't. Then, each converted node simultaneously contacts some members of the next generation (i.e, some of the nodes in the row "below"), 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 although a majority of generation 0 started out being converted, by generation 2, only one node is converted, and at generation 3 the CS1110 idea has died out, since none of the nodes in that generation heard about CS1110 from enough people to be converted themselves.



More formally — and recalling the "double-dot" notation explained in §2.1 — 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, where each converted node in generation g' contacts exactly d randomly-selected nodes in generation $g' + 1$ — except that generation $g - 1$ does not do any further contacting, since it's the last one. In the figure above, $d = 2$.

For convenience, our printouts use (x, y) to refer to the node in generation x with index y. For example, the "youngest" converted node in the picture above would be called (2,1).

2.3. **Class invariants.** In the class definitions for Node and Trial, you'll see docstrings describing the *class invariants*: statements giving the meaning of and constraints on the attributes of the objects in each of these classes. *Read these very carefully, and make sure you understand them before proceeding.* For any methods or functions you write, it must be the case that if the class invariants held before the method or function is called, then they must hold after execution finishes — "invariant" means "non-changing".

3. Tools and Tests Provided

We've provided you with a number of utilities to help you incrementally test the correctness of your code and see what it is doing. But you are responsible for deciding whether you need to write more test cases or perform additional checks.

3.1. **A trial-visualizer.** As soon as you have the node __init__ function at least creating the expected attributes (whether or not their values are correct), you can create trials, and can use our visualizer to check nodes. A short way to try this out is to run nodetest.py; you'll see a three-layer trial whose *expected* attribute values can be determined by looking at the constructor expressions in the definition of function create_three_layer_trial in nodetest.py. Clicking on any node in the visualization produces a printout of the *current* attribute values of that node, so you can check what the expected values are against what the visualizer tells you. (The colors might be wrong, depending on how much of the assignment you've completed so far.) A printout gives you directions for how to pan, zoom, and proceed after you're done with that visualization. Note that it can take a little time for the visualizer to give control back to you after its window is closed.

Another way to try this out the visualizer (again, once Node's __init__ is at least partially working, and once you've imported `trial`) is with the following lines:

```
tr = trial.create_sample()
trial.showit(tr)
```

This is what we used to produce the figure above. The code in the definition of trial.create_sample will show what the expected values for each node's attributes are. Alternatively, you can use trial.create_dense_sample instead of trial.create_sample.

3.2. **Printing.** If you prefer to see a printout of all the node attribute values in a trial all at once, rather than clicking on individual nodes, you can invoke the __str__ functions we wrote for you using `print`. For example, here's an interaction given our solution code. (You can use the printout below to help verify whether your code is correct!)

```
>>> import trial
>>> t = trial.create_sample()
>>> t.nodelist[0][0].get_legacy() # output suppressed
>>> t.nodelist[0][1].get_legacy() # output suppressed
>>> print t
g=4 n=3 c=2, d=2 t=2 avg % converted=0.333333333333
...
```

```
g=4 n=3 c=2, d=2 t=2 avg % converted=0.333333333333
...
(0, 0). d=2 t=2 init_converted=True is_converted()=True
contacted by:
contacted: (1,0), (1,2)
_legacy: (1,0), (2,1), (1,2)
(0, 1). d=2 t=2 init_converted=True is_converted()=True
contacted by:
contacted: (1,0), (1,2)
_legacy: (1,0), (2,1), (1,2)
(0, 2). d=2 t=2 init_converted=False is_converted()=False
contacted by:
contacted:
_legacy:
---
(1, 0). d=2 t=2 init_converted=False is_converted()=True
contacted by: (0,0), (0,1)
contacted: (2,0), (2,1)
_legacy: (2,1)
(1, 1). d=2 t=2 init_converted=False is_converted()=False
contacted by:
contacted:
_legacy:
(1, 2). d=2 t=2 init_converted=False is_converted()=True
contacted by: (0,0), (0,1)
contacted: (2,1), (2,2)
_legacy: (2,1)
---
(2, 0). d=2 t=2 init_converted=False is_converted()=False
contacted by: (1,0)
contacted:
_legacy:
(2, 1). d=2 t=2 init_converted=False is_converted()=True
contacted by: (1,0), (1,2)
contacted: (3,0), (3,1)
_legacy:
(2, 2). d=2 t=2 init_converted=False is_converted()=False
contacted by: (1,2)
contacted:
_legacy:
---
(3, 0). d=2 t=2 init_converted=False is_converted()=False
contacted by: (2,1)
contacted:
_legacy:
(3, 1). d=2 t=2 init_converted=False is_converted()=False
contacted by: (2,1)
contacted:
_legacy:
(3, 2). d=2 t=2 init_converted=False is_converted()=False
contacted by:
contacted:
_legacy:
---
```

3.3. **Tests.** Run nodetest.py and trialtest.py to execute a number of useful tests of your code.

4. Implementing the Basic Node and Trial Functionality

4.1. **Additions to node.py.** Before being able to run trials, we need to be able to correctly create the nodes that make up those trials, which means writing the node initializer. We've written the more difficult parts of the __init__ method's body for you in node.py; take a look at that code and corresponding comments and make sure you understand what's going on. Then, complete the body of that function according to its specification. You can consult the __init__ method for class Trial in module trial for examples. Test your code by running nodetest.py; you should get a visualization you can click on to get node attribute values, although all the nodes will have the same colors at this point; and when you dismiss the visualization window you should somewhere see a message "test cases for __init__ passed".

We also need to be able to determine whether a node has been converted or not. Complete the body of method is_converted according to its specification and the implementation comments we've given you. Test by running nodetest.py again; this time, the nodes in the first visualization should have the right colors — all five of the generation-0 nodes plus node (1,1) should be black, showing that they are converted.

Finally, we need each node to be able to randomly contact the right number of distinct nodes. The way we indicate that a node n1 has contacted another node n2 is that n1's list `contacted` should contain n2, and n2's contacted_by list should contain n1. Complete method `contact` according to its specification and the comments given to you in the code. Test by running nodetest.py again; if all is well, after you dismiss the first visualization, you should see a visualization of the result of calling method `contact` for nodes (1,0) and (1,1) and a printout of what will be displayed if your code passes our test cases. Then, dismiss the visualization window and see whether somewhere you get the message

```
test cases for contact passed
test cases for is_converted passed
```

Quit and try re-running nodetest.py again up to this point a few times; since your `contact` function is supposed to be contacting potentially different sets of nodes each time it is run, you should probably see differences in which nodes in generation 2 get contacted.

4.2. **Additions to trial.py.** We also need trial objects to be able to simulate the process of one generation influencing the next, which means that the converted nodes in a given generation should contact various randomly-selected nodes in the next generation. Implement method _propagate_row of the class Trial according to its specification, making use of the `contact` function you just wrote. Understanding the class invariants will come very much in handy here, in terms of figuring out how to refer to the nodes in a given generation.

To test, run trialtest.py; if all is well, somewhere you should see the line "tests for propagation of row passed".

5. Experiment: The Percentage Converted Eventually Converges to a Fixed Point We Can Compute With Frightening Accuracy, *Or*, Random Human Behavior Can Be Surprisingly Easy To Predict

OK, now for some scientifically interesting questions! First, can we predict what percentage of each future generation is converted?

Let's take a sample case. Suppose we have 100 generations, each of size 1000, and we start out with only a very *few* people initially converted — just $c = 4$, say. And furthermore, suppose each converted person only contacts 2 people in the next generation ($d = 2$). Then it seems certain that the idea of CS1110 is going to pass into oblivion, as there are only a few people taken with it, and they aren't exactly shouting it from the rooftops.

But, suppose that the CS1110 idea is so catchy that a person only has to hear about it from one person in order to be converted ($t = 1$). Then, the "CS1110 bug" will survive to the last generation. But will the converts be a teeny tiny possibly despised minority, or could they actually end up ruling the future world?

It seems like this should be impossible to predict, since there's randomness built into the process. For instance, if in every generation it just happened that all the converts contacted the exact same two people, then in the end the fraction of the 1000 people in the last generation that are converts would be $2/1000 = .2\%$. But that might not happen, right?

OK, here's the astonishing thing:[3] The fraction of people converted in a generation under the conditions described above tends to *converge* to one of *only two* numbers: either to 0 or to ... 79.7%, a sizeable majority. Yeah, that precise! Those two numbers come from the fact that they are the two solutions to $1 - x = e^{-xd}$ when $d = 2$. If we increase the number of people a convert contacts by just a little bit, say $d = 3$, then the majority tends to increase to a whopping 94%!!

Here's your chance to verify this fact empirically. To do so, we need to compute the fraction of nodes in a given generation that are converted; complete method frac_of_gen to do so.

Once you've done that, take a look at function test_against_prediction in trialtest.py and the application code that calls it. What the function does is run your _propagate_row method on each generation and, for generations 25 and on, prints out the fraction of people converted in that generation. Now, run trialtest.py, and you should see that the sequence of fraction-converted numbers quickly converges to the two predictions given above. You can try running it several times if you like, to see if this is a repeatable experiment.

Moral: The set of people who have an idea might be small (low c) and not able to reach a very large audience (low d); but if that idea is compelling enough (low t), **those people can change the world**.

6. Experiment: How Much Difference Can Any One Individual Make?

We say that the *legacy* of a convert is the set of people in later generations that the convert 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 $(2, 1)$. It's interesting to ask how big a legacy can get; for instance, can there be a single "mitochondrial Eve" for CS1110 that almost all future CS1110 converts are "descended" from?

To look into this, we need to compute the legacy of a node.

6.1. Idea: recursion.

This is a naturally recursive thing to do: the legacy of a node x is the set of people in its `contacted` list who are converted, combined with the legacies of all those converted contactees. Again, in the example above, $(0, 0)$'s legacy is $(1, 0)$ plus $(1, 2)$ plus the legacy of $(1, 0)$ plus the legacy of $(1, 2)$, although in that example the two latter legacies turn out to be the same. So, conceptually, the path is clear.

However, the straightforward implementation of this idea is very costly in terms of memory usage, even for just moderately-sized trials, because the same recursive computation is done many times. For instance, computing the legacy of $(0, 0)$ requires computing the legacy of $(1, 0)$, and computing the legacy of $(0, 1)$ also requires computing the legacy of $(1, 0)$. Similarly, since both $(0, 0)$ and $(0, 1)$ contact $(1, 2)$, the legacy of $(1, 2)$ would also seem to have to be computed twice. We have better things to do than compute the same thing over and over!

6.2. Idea: caching.

Here's an idea computer scientists use in such situations: store the answer to a repeated computation somewhere easily accessible; then, if you need that computed value again, just look at your stored version rather than recomputing from scratch.

[3] We learned of this fact and its proof from Prof. Jon Kleinberg.

Specifically, it's now time to make use of the hidden node attribute _legacy to cache the legacy of a given node the first time we compute it, so we never have to calculate it again. To indicate the case that we haven't computed it before, we initialize _legacy to None for each node.

With this idea in hand, here's a battle plan for implementing method get_legacy.

- If this node's _legacy is not None, just return the stored value of _legacy. Done!
- Otherwise, if this node has a non-empty list of contacts, then we need to collect the set of converted contacts, if any, plus the legacies of those converted contacts. Computing the legacies of the converted contacts involves a call of the get_legacy method of each such contact. Store the computed legacy value in _legacy so we never have to do this again, and then return it.
- If this node has an empty ~~`contact`~~`contacted` list, then the recursive case doesn't hold. But if that's the situation, then this node's legacy is empty. So make _legacy empty rather than None, and return the empty legacy.

6.3. **Idea: the set type.** There's another bit of waste in the recursive battle plan sketched above that would be good to trim. Looking again at the figure above, note that if we combined a list of the legacy of $(1, 0)$ with a list of the legacy of $(1, 2)$, then the node $(2, 1)$ would show up twice, since it's in both legacies. It would be nice to not have to deal with duplicates, so that we aren't storing large lists with lots of redundancies.

The Python *set* type turns out to be handy for this, because making a new set out of two sets removes duplicates automatically. Specifically,

- 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` or set `t`: `myset = s.union(t)`. This is a way to add multiple items to a set, and have duplicates removed.
- To create an empty set: `set()`
- To add an item `x` to set myset: `myset.add(x)`

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

To test, run nodetest.py. Dismiss the first two visualizations; you should see "beginning test of get_legacy" printed out. Write down what the instructions say you should compute by hand, and then dismiss the visualization. You'll get a message "Now the legacies have been computed"; do what the instructions say to do to check your answers for each node. Dismiss the latest visualization, and if your legacy computations pass our assertion tests, you'll get a visualization of a new trial to hand-check legacies for; repeat the process.

It will feel so good at the end when you get to the much-hoped-for message, "All test cases for node passed".

6.4. **Observation: "phase transitions" in legacy size.** Once your legacy-computation method is working, you can run trial.py.[4] Its application code checks what the average legacy size is over the initially-converted generation-0 nodes, for a variety of initial conditions. You'll see that some perhaps seemingly-small changes in those initial settings lead to very different average generation-0 legacy sizes. For instance, in one of the experiments, changing d from 3 to 6 causes the average generation-0 legacy size to jump from something like .05% to about 80% of all future students! You can try these experiments more than once, too, to see what kind of variance there is.

Moral: it's can be very much worth the effort to cover that one last mile — to reach out to just one or two or three more students and see if you can convey to them the joy and importance of programming.

[4] If the computations seem to take forever, you probably aren't implementing the caching idea correctly. But they do still take a little bit of time even with caching.

7. Finishing the Assignment

Double-check that the headers of your files include your name(s) and netid(s) and properly acknowledge all your sources (classmates, webpages, etc.), as mentioned in §1.2.

Before submitting, read through your code once more. You might come up with more elegant ways to do things, notice variables you don't need, see ways to make your code cleaner, and so on. Also, check that your code meets the class coding conventions, such as:

- 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/Edit in Komodo Edit to make it display the 80-character boundary: click the "Draw the edge line column" box and have the value of the "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 file that you worked on, you have three single-line comments with (1) the file name, (2) your name(s) and netid(s), and (3) the date you finished the assignment.

Finally, submit your node.py and trial.py on CMS. CMS lets you resubmit up until the submission deadline; the last submission that makes it in before the deadline is the one that will be graded. It is thus wise to make sure to get at least one submission in substantially before the deadline, just in case you lose track of time.