

CS51 Project 1c: An Infinite Number of Monkeys

Due: Tuesday, 3 March 2009 at 11:59 PM

Total Points: 45 (including 5 style points)

This is the third part of Project 1. By now, you've worked a bit with dictionaries and queues in the process of learning about interfaces, and then built a generative grammar (Arrr!). In this piece of the project, we will implement a Markov chain babbling.

1 Downloads

This project will require you to download multiple files. You will still only need to submit your .scm file. Download the following files from the website and place them in the same directory in which you will be saving your work.

- proj1c-utils.ss: I/O functions.
- graph.ss: Undirected graphs with weighted edges.
- rj-text: Act I of Romeo and Juliet. You may use any subset of this file for your babble. (Or any other text you feel like using, but this one has been pre-formatted for you.)

2 What you must do

From now on, we'll be doing our best to provide an up-front description of what you must actually turn in. The first part of these assignments tends to be write-up, which can be pretty long, and sometimes it's hard to tell what the actual exercises are.

2.1 Write-up

- Markov Chain Babbling
- Random walk on a graph

- Our chosen implementation
- Logistics and utilities

2.2 Exercises

- Helper functions and tests for `tokens-to-graph`.
- Helper functions and tests for `babble-ngrams`.
- Actual babbling, minutes-spent and feedback.
- Challenge problem, worth no points.

Feel free to skip any parts of the write-up that you already understand, but do be sure you don't miss anything you'll need for the assignment. The actual exercises begin on page 5.

3 Markov Chain Babbler

For this portion of the problem set, you will implement a “babbler” in Scheme using a probabilistic algorithm. A *Markov chain* is, loosely defined, a “chain” built by choosing each “link” with a probability based on no more memory than a few preceding links in the chain. A Markov chain is thus essentially a random walk on a graph. You start with a graph with some nodes and edges (links) between them; at each node, you choose semi-randomly where to move based on some information you have about the probabilities associated with each of the edges that you might choose to follow.

This algorithm can be used to model natural language, where each link is a clearly defined piece of the language. Such “links” could be phonemes, letters, words, or even phrases. For this assignment, we will use words, punctuation marks, and end-of-paragraph (`eop`) for our links. All of our words, including `eop`, will be represented in Scheme as **symbols**, which is a data type that we haven't discussed yet this semester. You don't need to know much about symbols; basically, `eop` may appear in your Scheme interpreter with a single quote in front of it. Note also that Scheme might represent lists with a single quote in front; this is equivalent to the `(list ...)` constructor. So, `'(1 2 3)` is the same thing as `(list 1 2 3)`. Feel free to use whichever syntax you prefer.

The *order* of a Markov chain refers to the number of previous links that are used to generate new links. Thus, a zeroth-order Markov chain would simply be random selections of words from the text; first-order would look back one word, etc.

The goal is to generate new text which is similar in the following way to an existing body of text: For zeroth-order, each word in the generated text occurs in the input, and no more restrictions are imposed. For first-order, each sequence of two words in the generated text

appears in the input. For second order, each sequence of three words in the generated text appears in the input. It is easy to see how increasing the order produces text which is “more like” the input; if we were to babble using an n th-order Markov chain given $n + 1$ words of input, clearly our resulting text would be identical to the input text.

3.1 Generating the probabilities

The probabilities for the Markov chain are extracted from a large sample of English text. To create a first-order chain, we read all the words into a list, then generate a list of *bigrams* from this list. (A bigram is two consecutive words.) Using the list of bigrams, we can determine the next word from the previous output word by extracting from the list all bigrams that begin with the previous word and randomly choosing one of these bigrams. For example, the phrase “a man, a plan, a canal: panama!” would have the following list of bigrams:

```
((eop a) (a man) (man ,) (, a) (a plan) (plan ,) (, a) (a canal)
 (canal :) (: panama) (panama !) (! eop))
```

3.2 Generating the babble

To begin the babbling process, we find an `eop` at the beginning of a bigram. Since `eop` (end of paragraph) indicates the end of a paragraph, the rest of a bigram beginning with `eop` is a legal beginning for a paragraph. In the above example, we find `(eop a)` is the only one, so we begin our phrase with `a`.

To find the next word, we take all bigrams beginning with `a` and choose one at random. From the filtered alternatives `((a man) (a plan) (a canal))`, we might randomly choose `(a plan)`, in which case the next word is `plan`. Thus the *next* word is chosen from the list of bigrams beginning with `plan`—`(plan ,)` is the only one. Our phrase is now `a plan,.` The next word comes from the list `((, a) (, a))`—regardless of the choice, it’s still `a`, yielding `a plan, a`.

This is where the chain makes things interesting. We can proceed to any bigram beginning with `a` yet again. Following the chain to the end, choosing randomly each time, might yield these results:

- a man, a plan, a plan, a man, a man, a canal: panama!
- a canal: panama!
- a plan, a plan, a plan, a plan, a canal: panama!

How can we improve the results? One way to make the result sound more sensible is to use a higher order Markov chain, where each new link in the chain is based on two or more previous words, not just the previous word alone. For a second-order chain using our example above, the set of trigrams would be:

```
((eop a man) (a man ,) (man , a) (, a plan) (a plan , ) (plan , a)
(, a canal) (a canal :) (canal : panama) (: panama !) (panama ! eop))
```

To generate a chain from the trigrams, we begin again by finding a trigram beginning with `eop` and using the rest of the list as the seed. Since `(eop a man)` is the only one, our result begins `a man`. We then randomly choose from all trigrams beginning with the last two words of the result; this proceeds as before, always matching the last two words of the babble thus far with the first two words of a selected trigram. Here are a few random second-order chains:

- a man, a plan, a plan, a plan, a plan, a plan, a canal: panama!
- a man, a canal: panama!
- a man, a plan, a plan, a canal: panama!

We notice now that, unlike in the bigrams, “`, a man`” can never appear in the result. Furthermore, every chain must begin with “`a man`”, cycle through zero or more “`a plan,`”s, and end in “`a canal: panama!`”. The second-order chain prevents the reordering of words that occurred in the first-order result. Clearly it would be more interesting if we started with a larger corpus.

4 Language

For the purposes of this assignment, we will be working in the full Scheme language. We’ll be using a graph module that we’ve provided for you. Be sure that you’re working in the Module language. The reason is that we’re giving you a couple of functions that deal with Scheme I/O to help you write your babbler, and these functions need the full Scheme functionality in order to work.

5 Monkeys

The input text we will be using to babble with is from Shakespeare. We suggest using Act I of *Romeo and Juliet*. Should you not like this classic story of star-crossed lovers, Shakespeare’s full works are available for download at <http://shakespeare.mit.edu> (although be aware that you might run into some weird formatting issues unless you’re careful about how you save things). For testing purposes, you might find it more useful to use the Panama sentence. For this assignment, you will need to save your code locally on your computer as well as remotely on the server; make sure that these text files are saved in the same directory as your Scheme file.

6 Utilities

We've provided two functions, one of which you will need and one of which you might want. We've also provided a module for undirected graphs with edge weights.

- (`filename->tokens filename`) takes a filename (as a string) and returns the words in the file as a list of symbols. You'll need this.
- (`display-tokens lst`) prettyprints the tokens in `lst` to make it easier to read. You shouldn't use this in your solution, but it can make your output look nice.
- The graphs module is in your `.scm` file. Take a look if you're curious; all you really need to know about is what's exported for you. The edge struct that's exported has a source node, a destination node and a numerical edge weight.

7 Group Frequencies

The first step we need to do is to figure out which groups of words occur in the input text. Once we know which groups are present, we can chain them together according to some rule to form our output. The first natural question is: what value of n are we using?

We will leave this somewhat up to you. We suggest using $n = 3$ for this assignment. However, if you really want to see the monkeys babble, you may choose $n = 2$; if you want to feel like Shakespeare, you may choose $n = 4$. Please choose n in the range $[2, 4]$.

The second question is, how are we representing all of these word groups? The main concern is frequency. We want to give more weight to word groups that appear more often. If "through yonder window" appears five times in the text, and "through yonder door" appears only once, we'd like our babbler to be five times more likely to choose "window" as the next token than "door" if the last two words were "through yonder".

We'll be representing our frequency data in a directed graph with **weighted edges**. The nodes will be lists of n tokens. The edges will be directed from one node to another, with the weight on an edge being the number of times in the input text that we move from the list of n tokens at the source of the edge to the list of n tokens at the destination of the edge.

8 Generating the Graph

(17 points total)

We'll need to start out by writing functions that take the corpus of text as input and make this graph. Our ultimate goal is a function, `ngrams-to-graph`, that takes as input the list of tokens from the file and gives back a graph, where the nodes are lists of three tokens, and the edges lead to the next trigram, with edge weights equal to how many times that transition

occurs in the input text. (All of this assumes $n = 3$; substitute appropriate numbers of tokens for other values of n).

Let's break this down, shall we?

Exercise 1.

17 points

We first want to write a function that takes a list of tokens and returns a list of the ngrams that this list of tokens produces. There's one subtlety: we want to wrap the first $n-1$ tokens in the list around to the end of the list. So, if $n = 3$ and our input is the list (A man , a plan . eop), we want to first transform it into the list (A man , a plan . eop A man), and then create all three-tuples, giving: ((A man ,) (man , a) (, a plan) (a plan .) (plan . eop) (. eop A) (eop A man)). The reason is that our babbler otherwise could get stuck if it were passed the last ngram in the text as input, and we didn't want to stop at 'eop'; it's possible that nowhere else in the text does this last ngram appear.

(a) [5 points] For each function that you are asked to write below, provide tests below the function definitions that convince you that your function is working correctly in all cases of proper input. (You may assume properly-formed input for the purposes of this assignment.)

8.1 Grouping

(b) [3 points] Write a function called `gen-ngrams` that takes a list of tokens and an integer value n and returns the list of ngrams that can be constructed from the list, including the last $n-1$ ngrams that represent wrapping the end of the text back around to the beginning.

You will likely find it useful to write a helper function, (`sublist n lst`), that returns a list of the first n tokens in `lst` (or the first k tokens if there are only k tokens in the list and $k < n$). You may find it helpful to write another helper function. Make sure you only end up with full ngrams!

8.2 Graphing

Now that we can produce the ngrams, we can write the `ngrams-to-graph` function that takes as input a list of ngrams, each represented as a list of symbols. It returns a graph such that each node represents an ngram, with each of its directed, outgoing edges leading to all ngrams whose first $n-1$ tokens correspond to its last $n-1$ tokens.

We'll need to break this function down into some helper functions. The hard part of building this graph will be dealing with the edge weights. Keep in mind that if we run across a series of n words that we've seen before, the edge going from the first $n-1$ tokens to the second $n-1$ tokens should have its weight incremented by one; but, if we've never seen these n words before, we want to create the edge and give it an initial weight of one. Luckily for you, we've provided just such a function: `graph-increment-edge` in the `graphs` module will

take care of this for you.

(c) [2 points] Okay, that's all well and good. But if we have a current ngram, how do we know which other ngrams we should connect it to? (In other words, which edges do we need to increment?) Write a function `is-prefix-of?` that takes two lists and returns true if and only if the first is a prefix of the second. (Hint: the empty list is a prefix of every other list.)

(d) [6 points] Now for the real deal. Write a function `ngrams-to-graph` that takes a list of ngrams and returns the graph built from these ngrams, as described above. Make sure you understand what this graph should look like. You will likely need to write a helper function that takes more inputs than just one copy of the list of ngrams. `foldr` and `filter` will likely both be useful in conjunction with the functions you've written above and the graph functions provided. Think carefully about what base case you want if you decide to use `foldr` in your recursive call.

(e) [1 point] Write a wrapper function `tokens-to-graph` that takes a list of tokens and returns the graph of its ngrams. Given the functions you've already written, this should be an extremely simple one-liner.

9 Babble, monkeys!

(23 points total)

Now that we have our graph, we can use it to generate babble!

You'll recall from lecture the concept of a random walk on a graph. This concept is what we will be using to babble our sentences. We want to write a function `babble-ngrams` that takes as input a graph as formed above and a starting list of n tokens, and babbles until it reaches an ngram that ends in the 'eop' symbol.

Exercise 2.

23 points

9.1 Randomness

The first thing we will need to do is figure out a way to pick a random edge to follow, based on the weights of all of the edges we could choose. If we have edges with weights 1, 3, and 2 going out of a certain node, we want to pick the first edge with probability 1/6, the second with probability 3/6, and the third with probability 2/6.

(a) [5 points] For each function that you are asked to write below, provide tests below the function definitions that convince you that your function is working correctly in all cases of proper input. (You may assume properly-formed input for the purposes of this assignment.)

(b) [2 points] Use `map` and `fold` (this is a requirement, not a suggestion) to write a function `total-edge-weights`, which takes as input a node and a graph and returns the total weight of all outbound edges from that node in that graph. Take a look at the graph contract to

find the graph function(s) you'll need. This function should be one line.

Now that we can total up the edge weights leaving a certain node, we can use this information to enable us to pick a random edge with the correct weighting.

(c) [3 points] Write a function `pick-random-edge` that takes a node and a graph and chooses a random outbound edge from that node, weighted according to the distribution of the weights. You will likely find the `total-edge-weights` function you wrote above, as well as the Scheme function (`random n`), useful here.

Hint: think about generating a random number and proceeding down the list until the total weight of all of the edges you've passed so far exceeds the number generated. Defining the right helper function - think about what parameters it should take - will be crucial here.

9.2 Babble

(d) [4 points] Awesome! We're finally ready to start babbling. Write a function `chain-ngrams` that takes a graph and a starting n-gram. It should return a list of n-grams, starting with the given one, and ending with an n-gram that ends in 'eop'. You may construct the list either in the correct order or reverse order, but do **not use append**. The most efficient method builds the list in reverse order for reasons that we'll see in the coming weeks, but as long as you do not use `append`, either way is fine. **Whichever way you choose to build the list, keep the order in mind as you write the next function.** You may assume that starter is always a node in graph. Use a helper function if necessary. Examples (our examples are in reverse order):

```
(define panama-sentence '(eop A man , a plan , a canal : panama ! eop))
(define panama-graph (ngrams-to-graph (gen-ngrams n panama-sentence)))

(chain-ngrams panama-graph '(panama ! eop)) --> ((panama ! eop))
(chain-ngrams panama-graph '(A man ,)) -->
((panama ! eop) (: panama !) (canal : panama) (a canal :) (, a canal)
(plan , a) (a plan ,) (, a plan) (plan , a) (a plan ,) (, a plan)
(man , a) (A man ,))
(chain-ngrams-panama=graph '(A man ,)) -->
((panama ! eop) (: panama !) (canal : panama) (a canal :) (, a canal)
(man , a) (A man ,))
```

(e) [3 points] Finally, write a function `combine-ngrams` that takes a list of connected n-grams (either reversed or in-order, depending on how you wrote the last function) and turns it into an in-order list of tokens. **You may not use reverse or append in this function or any helper functions you write for it.**

Our actual babbler can now be simply composed of calls to certain of the above functions.

(f) [3 points] Write a function `babble-ngrams`, taking a graph. It should pick a random starting ngram **that begins with 'eop'** and return a list of tokens, starting with the chosen ngram and ending in 'eop', and representing a valid random walk on the graph of n-tuples generated in Exercise 1. Recall how you chose a random element from a list on Pirates!.

9.3 Fun!

In your definitions window, type the following line:

```
> (define my-graph (ngrams-to-graph (gen-ngrams n '“rj-text”')))
```

This will open the text file “rj-text” (you might have to move it around to get it in the right directory); generate ngrams from it; and then build the graph. Now go grab a drink and relax. Come back in ten minutes (seriously). You only want to have to evaluate this step once, so test your babbler thoroughly on the panama graph before attempting this!

If your graph-generating step is taking too long ($\sim 15min$), we recommend cutting the file down. Taking the first 1/4 of the file should have dramatic results in terms of time improvement. (Why? See the Challenge Problem at the end of the project.)

For reference, our solution takes about 75 seconds to generate the graph from an input text of just over 2000 words. If you'd like, include a comment telling us how much you beat us by!

(g) [2 points] Reproduce your three favorite babbles below. You may replace Romeo and Juliet with the text of any Shakespeare play (or indeed, whatever text you want). Include in a comment the text that you babbled from.

(h) [1 point] You know what to do. Please also leave us comments if you have any!

10 Challenge problem

Exercise 3.

0 points

This problem is worth no points; it is intended merely as a thought exercise for you.

(a) [0 points] What is the Big-O complexity of our overall graph-generating algorithm? You will need to know what the graph module is doing internally in order to be able to answer this question; feel free to go muck around inside of it, as you can always download a fresh copy. What's stupid about the code we've written? (Hint: a lot.) How could you improve some of these functions?

(b) [0 points] Use the profiler tool in DrScheme to figure out what function is causing the biggest slow-down (the answer might come as a surprise). You can turn on the profiler by going to Language – Choose Language... and clicking on the Show Details box. Then select the appropriate options from the right-hand menu and play around!

What are some ways we could improve the performance of this slow function that could make our graph generation a ton faster, and in turn allow us to babble on larger inputs?

(c) [0 points] Fix this slow function and impress us with your blazing fast graph generator.