CSC 207 **Assignment 7: Markov Chain** **Fall 2009**

Due: Friday 10 April 2009

Topics: Implementing and using collections.

Collaboration: This homework assignment may be completed individually or in pairs.

Submission: Follow the instructions for submitting programs via P-Web and handing in a printed copy. Be sure to generate test cases and a statement justifying their sufficiency. One only one submission (paper and digital) per group is necessary.

1 Introduction

A Markov chain is a model in which the future depends on the past (or present). Some common examples of things that might be modeled with a Markov chain would be the path followed by a person who is walking. Most of the time, you can predict what direction they will go next based on their last few steps. One of the earliest, and still the most common, use of a Markov chain is for text. The English language (and many others) has remarkable regularity in this regard.

1.1 Definition

If someone said to you, "Go west, young ....," you could probably predict with great certainty that the next word would be "man." Markov chains can capture this predictability of text.

So where might such a model come from? Well, like you and I learned language by experiencing it, we want to create a model that can "learn" from experience in the form of English text. One reasonable way to do this is by counting the frequencies of the word sequences in a stream of text.

Assume the words $w_1, w_2, \ldots, w_{N+1}$ occur sequentially in the example text. We say that word w_{N+1} is a **suffix** of the **prefix** sequence $w_1, w_2, \ldots, w_N$. Using a prefix length of N, we'll create what is called an Nth order Markov model. The more times a certain suffix is associated with its prefix (called its **multiplicity**), the stronger the association, and the more probable we would like it to appear in a "randomly" generated sequence. (More on what that might mean shortly.)

Texts with several repeated phrases are most interesting. For instance, consider the following passage:

> *Blessed are they which do hunger and thirst after righteousness: for they shall be filled.*
> *Blessed are the merciful: for they shall obtain mercy.*
> *Blessed are the pure in heart: for they shall see God.*
> *Blessed are the peacemakers: for they shall be called the children of God.*
>
> Matthew 5:6-9 (King James Version)

Notice there is some repetition of phrases, while others are more uncommon. We may build a table of prefixes and the associated suffixes (including their multiplicity) with $N = 2$. Since every word must have a prefix, how do we start? The prefix of the first "*Blessed*" would be off the left edge of the text. The solution is to let the first prefix be a sequence of N `NULL` words, where `NULL` is something that is not an actual word represented in the text. Similarly, every prefix in the text should have a suffix, so how do we end? The suffix of the last phrase "*of God.*" would be off the right edge of the text. We can do the same thing and say its suffix is `NULL`. The following is an abridged version of the table.

Prefix	Suffixes
`NULL NULL`	*Blessed*
`NULL` *Blessed*	*are*
Blessed are	*the* (3x) *they*
are they	*which*
are the	*merciful: pure peacemakers*
the merciful:	*for*
merciful: for	*they*
for they	*shall* (3x)

Prefix	Suffixes
they shall	*be* (2x) *obtain see*
shall obtain	*mercy.*
shall see	*God.*
shall be	*called filled.*
called the	*children*
the children	*of*
children of	*God.*
of God.	`NULL`

Note that some prefix phrases have several possible "branches." For instance, *"are the"* could be followed by one of three equally likely possibilities. Other prefixes allow no chance for branching; *"for they"* will only be followed by *"shall"*. You should also notice that some branches are more likely than others; "*Blessed are*" is three times more likely to be followed by "*the*" than *"they"*.

1.2 Generating a Model

So how do we generate such a table? The following pseudo-code describes it. N is the order of our model and T represents an input text sequence iterator over words.

Algorithm 1 Generating an order N Markov chain model.

```
BUILD-MARKOV(N, T)
    w_1, ..., w_N ← NULL, ..., NULL
    while HAS-NEXT(T)
        w_{N+1} ← NEXT(T)
        ADD-SUFFIX(w_1, ..., w_N, w_{N+1})
        w_1, ... w_N ← w_2, ... w_{N+1}
    ADD-SUFFIX(w_1, ..., w_N, NULL)
```

The ADD-SUFFIX method would take some representation of the prefix $w_1, \ldots, w_N$ and add the suffix w_{N+1} to the set of suffixes associated with that prefix. The first line of the algorithm establishes the prefix for the beginning of the text and the last line the suffix for the end of the text. The last line of the loop slides along to the next prefix for the next word. With all the right data structures in place, this really is it! Quite a simple algorithm...assuming you have the right data structures.

1.3 Generating Sentences

So what can we do with this Markov model? It turns out that computational linguists have figured out a lot of things we can do, such as language identification, text compression, speech recognition, and many more. We are going to use it as a "generative" model of text. That is, we will use it to create *new* sentences that behave like the text that we've seen. Since this actually isn't a very good model for generating text, it should be fun because they'll be somewhat silly. The algorithm is called a "random walk" model because we are essentially taking random steps from among the branches of the chain.

Algorithm 2 Sampling from an order N Markov chain model.

```
SAMPLE-MARKOV(N)
    w_1, ..., w_N ← NULL, ..., NULL
    w_{N+1} ← RANDOM-SUFFIX(w_1, ..., w_N)
    while w_{N+1} ≠ NULL
        ADD-WORD(w_{N+1})
        w_1, ... w_N ← w_2, ... w_{N+1}
        w_{N+1} ← RANDOM-SUFFIX(w_1, ..., w_N)
```

An important thing to note is that when selecting a random suffix associated with the prefix $w_1, \ldots, w_N$, its likelihood should be proportional to the multiplicity of the suffix for the specified prefix. For instance, say the suffixes (with multiples explicitly shown) are {*be, be, obtain, see*}. Randomly picking a suffix means picking a random number between 1 and 4, or randomly choosing one of the four possible suffix words.

2 Assignment

Your task for this homework is to implement a Markov chain whose table is established from a given stream of text and then sample from it to generate new sentences. To do this, you will also need to use and implement some data structures.

2.1 Implementations

2.1.1 Prefixes

How will we represent a prefix? Recall that a queue is a First-In-First-Out data structure. What if we had queue that only allowed a certain number of elements, say three. When the queue was "full," if we added a new element, it would be at the end, but the oldest element would be eliminated from the queue (automatically deqeueued). This is exactly the behavior we desire from the two algorithm lines that say $w_1, \ldots w_N \leftarrow w_2, \ldots w_{N+1}$. We add w_{N+1} and eliminate the original w_1, with the "index" (place in the queue) of all the other items shifting up by one.

We call this a **fixed length queue**, and have provided an implementation using an array. You should examine the interface `FixedLengthQueue.java` to understand how it is to be used, and you may scan the implementation `ArrayFixedLengthQueue.java` for any details you are curious about. In particular, you should note that it implements `clone`, meets the contracts for `hashCode` and `equals`, and that `null` elements may be added.

For ten points of extra credit on this assignment, you may write (and demonstrably test) a list-based implementation of `FixedLengthQueue`. You may use the Java Collections API to do so. For full-credit, it should be functional as to be completely interchangeable with `ArrayFixedLengthQueue` in your implementation of the Markov chain. That is, the code you submit should actually use your queue implementation.

2.1.2 Suffixes

How will we represent suffixes? Specifically, how will we represent the *set* of suffixes associated with a given prefix. Since we want to randomly sample from the suffixes in proportion to their frequency, we need to store that information. Such a structure is called a **multiset** in both mathematics and data structures. Whereas a set does not allow duplicates, a multiset does. You are provided a `MultiSet` interface. The main thing to note is that the interface guarantees no particular ordering of the elements, only that they are indexable, so you can retrieve the kth element. Thus, if we pick a random (valid) index, the elements that appear more often are more likely to be retrieved.

A simple, but not very efficient implementation has been given in `LinkedMultiSet.java`. This is not efficient because it stores repeated elements multiple times. You are to write a more efficient implementation of this data structure with the following performance: `add` should be $O(1)$ on average, and `get` should be $O(K)$, where K is the number of *unique* items in the multiset.

2.2 Program Specifics

Your program should read the example text from standard input, and take two arguments, one indicating N, the prefix length, and the other giving upper bound on the number of words to generate after building the table:

```
java Markov order length < input.txt > output.txt
```

The only separators for words you should use is whitespace. Ignore any punctuation (that is, keep it around). Why? With enough text, this will make it easy to model sentence structure. The built-in Java class `Scanner` makes it very easy to do this:

```
Scanner scanner = new Scanner(System.in);
while (scanner.hasNext())
   String word = scanner.next());
```

Your program should generate (to standard output) the number of words specified on the command line, unless termination is required by the sampling algorithm (i.e., the `NULL` suffix is encountered).

As always, you should use an object-oriented program design, following the principles of encapsulation and cohesion.

Sample files of the books of Psalms and Proverbs (King James version) are provided for you to use in testing. These have several repeated phrases throughout, and they mix nicely. **Include in your submission one example output from this sample text that you enjoy.**

3 Practica

3.1 Getting Started

Starter files (interfaces, classes, and the sample text) are in the MathLAN directory:

```
~weinman/courses/CSC207/code/markov
```

3.2 Developing

For development, I highly recommend starting with a small text (presumably created by you) for which you can understand what the prefix/suffix table is supposed to be.

Important! In addition, there is a subtlety with how the Collections API works that can be extraordinarily frustrating if you are unaware of it. In particular, some collections use the `hashCode` method to index objects. However, Java *also* stores a reference to the object. If the object later changes, the hash code might change, too. This indicates the modified version of the object would be stored elsewhere. However, the collection still has a reference to the original object, whose index was calculated while it was in a different state. This can be extraordinarily confusing if you are using `toString` methods to test and develop your program. Thus, you are reminded (or told) of the `clone` method (pointed out above), that builds a "copy" (whatever that means for the class in question) of an object. Keep this in mind if you are putting an object into a collection and subsequently (later) modifying it.

You may find that you need to increment the amount of memory available to Java when you run your program. You can do this with the `-Xmx`*`N`*`u` switch to `java`, where *`N`* is a number and *`u`* is one of `k` or `m` indicating the units of kilobytes or megabytes, respectively. For example (and you hopefully won't need this for *your* program) `-Xmx1024m` allows Java to use 1GB (1024MB) of memory.

3.3 Running

You are invited to use other texts as input to your program and generate output. If you find an interesting text and/or sample, please share them. Project Gutenberg (`http://gutenberg.org`) is a fantastic source for copyright free texts. (In order to get meaningful samples from them, it may be necessary to remove the header/footer text.)

How big an N is reasonable? It depends on your text. The size of the table to grows exponentially in N, where the base of the exponent is (roughly) the size of the vocabulary used by the text. This is an upper-bound, and since most sequences do not appear in the text, it is a very loose bound. (You will probably find that as you use larger N, the phrases produced seem less random. Why?)

You might find it interesting to note that Google has such prefix/suffix tables for "teh Intarweb" for N up to 5 (`http://googleresearch.blogspot.com/2006/08/all-our-n-gram-are-belong-to-you.html`), and it's a "mere" 24GB.

Acknowledgment

This project was inspired long ago by an exercise in Chapter 3 of

The Practice of Programming, Brian W. Kernighan and Rob Pike, Addison-Wesley (Reading, MA), 1999

and improved by a similarly inspired project crafted by Curt Clifton and Matt Boutell at Rose-Hulman Institute of Technology.