Lab 4 - *BabbleLab*

Due: *Mon, Sept 29th, 2014 @ 11:59 pm*

Parallel and Sequential Data Structures and Algorithms 15-210 (Fall '14)

1 Introduction

Often when we say "I love you" what we really mean is: "You're pretty close to the defense of advertising agencies", but it's no big deal. –Mark V Shaney

This week, you will be implementing a simplified version of the **Dissociated Press** algorithm. Taking content from an input corpus (a body of text), the algorithm generates a string of facetious garbage, structured as if it were "babbled" by a monkey on a typewriter. It's similar to a predictive text algorithm, except it randomly chooses each successive word.

In order to Babble, you'll have to implement a Markov chain data structure which is capable of efficiently performing a random walk.

This lab is a bit of a change of pace from previous labs – hopefully you will find it to be less algorithmically challenging, and a bit more fun. You will also become more familiar with our table library.

2 Files

After downloading the assignment tarball from Autolab, extract the files by running:

```
tar -xvf babblelab-handout.tgz
```

from a terminal window. Some of the files worth looking at are listed below. You should only modify the files denoted by *, as these will be the only ones handed in by the submission script.

1. `Makefile`
2. `data/`
3. * `MkRTableMarkovChain.sml`
4. * `MkBabble.sml`
5. `Tests.sml`

Additionally, you should create a file called:

```
written.pdf
```

which contains the answers to the written parts of the assignment.

3 Submission

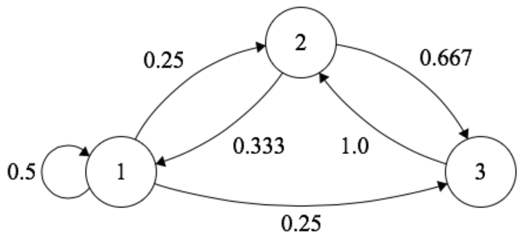
To submit your assignment to Autolab, open a terminal, `cd` to the `babblelab` folder, and run:

```
make
```

Alternatively, run `make package`, open the Autolab webpage and submit the `handin.tgz` file via the "*Handin your work*" link.

4 Markov Chains

A *Markov chain* consists of a collection of states and transitions between those states. Each transition is associated with a probability. The sum of probabilities of outgoing transitions from any state must add up to 1. Here's an example:



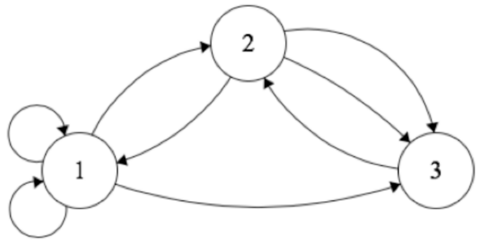
In this lab, we'll be using Markov chains to perform a *random walk*: starting from some state, we transition to a new state by picking an outgoing transition randomly with respect to their probabilities. We arrive at the new state, and repeat. (This can either continue forever, or terminate after some number of states have been visited.)

One reasonable implementation of a Markov chain would be a nested table. Each state would map to a table which maps possible new states to their transition probabilities. The above example would look like this:

$$\begin{aligned}\{1 &\mapsto \{1 \mapsto 0.5,\ 2 \mapsto 0.25,\ 3 \mapsto 0.25\},\\ 2 &\mapsto \{1 \mapsto 0.333,\ 3 \mapsto 0.667\},\\ 3 &\mapsto \{2 \mapsto 1.0\}\}\end{aligned}$$

However, it's actually quite slow to perform a random walk using this representation. If n is the number of states, then we need $O(n)$ work to select a random transition at each state.

To optimize random walks, we'll use a representation which stores probabilities implicitly rather than explicitly. Specifically, we won't label transitions with probabilities, and instead allow duplicate transitions. We'll use a table which maps states to a sequence of possible next states – the probability of transitioning to a particular state is defined as the number of times it appears in the transition sequence divided by the total length of the sequence. Let's convert the original example into this representation:



$$\begin{aligned}\{1 &\mapsto \langle 1,1,2,3\rangle,\\ 2 &\mapsto \langle 1,3,3\rangle,\\ 3 &\mapsto \langle 2\rangle\}\end{aligned}$$

This way, to select a random transition at some state, we can simply select a random index in its transition sequence. This is constant time. Note that the ordering within the transition sequences doesn't matter. A convenient consequence of this representation is that it's really easy to *merge* two separate Markov chains.

4.1 Implementation

Throughout, we will say that two chains are equivalent if they represent the same Markov chain. The ordering of elements within each state's transition sequence doesn't matter. We've defined `type chain = (Key.t Seq.seq) RTable.table`. You are not allowed to modify this definition.

For details on randomness and `RANDOM210`, see Section 4.2. For information on `RTABLE`, see Section 4.3.

Task 4.1 (5%). In `MkRTableMarkovChain.sml` write **one line of code** to implement the function

```
val build : (Key.t * Key.t) Seq.seq -> chain
```

Given a sequence of transition pairs S, `(build` S`)` should evaluate to the appropriate Markov chain. Each pair (k_1, k_2) in S indicates a single transition from k_1 to k_2 (i.e. one occurrence of k_2 in k_1's transition sequence). The example on the previous page could be constructed with

$$\texttt{build}\ \langle(1,1),(1,1),(1,2),(1,3),(2,1),(2,3),(2,3),(3,2)\rangle$$

Note that the order of elements within S doesn't matter – any permutation of the above sequence would build equivalent chains. **For full credit**, your implementation must have $O(|S|\log|S|)$ work and $O(\log^2|S|)$ span.

Task 4.2 (5%). In `MkRTableMarkovChain.sml`, write **one line of code** to implement the function

```
val merge : chain * chain -> chain
```

If given chains $C_1 =$ `build` S_1 and $C_2 =$ `build` S_2, then `(merge (`C_1`,`C_2`))` should be equivalent to `(build (Seq.append (`S_1`,`S_2`)))`.

Task 4.3 (20%). In `MkRTableMarkovChain.sml`, implement the function

```
val shrink : chain -> real RTable.table RTable.table
```

Given a chain C, `(shrink C)` should evaluate to the first representation described on the previous page. Specifically, for each pair of states k_1, k_2 in the chain,

`(RTable.find (valOf (RTable.find (shrink` C`)` k_1`))` k_2`)`

should evaluate to `SOME` p if there is a transition from k_1 to k_2 with probability p, and `NONE` otherwise.[1]

For full credit, your implementation must have $O(nm\log m)$ work and $O(\log n + \log^2 m)$ span, where n is the number of states and m is the length of the longest transition sequence within C.

[1]Technically, this code will raise the exception `Option` if k_1 is not present in C, but we'll assume that it is.

Task 4.4 (25%). In `MkRTableMarkovChain.sml`, implement the function

```
val iter : ('a * Key.t -> 'a) -> 'a -> int -> chain -> Rand.rand -> 'a
```

Where (`iter` f b n C r) evaluates to the result of (`Seq.iter` f b S) where S is a sequence of states obtained from C via a random walk. S should have length *at most* n (see below). If $n = 0$ or C is empty, then `iter` should return b. **For full credit**, your implementation must have $O(W_{\texttt{Seq.iter}} + n \log |C|)$ work and span, where $W_{\texttt{Seq.iter}}$ is the work of (`Seq.iter` f b S).

Some notes:

- You will need to use the function `rselect` from `RTABLE` to select a state as the beginning of the random walk. See Section 4.3.
- The input r is a random seed. See Section 4.2.
- "S should have length at most n" – In our representation of a Markov chain, it is possible for a state to have no outgoing transitions. (From a mathematical viewpoint, this is super broken. But for our purposes, it's actually a desired behavior.[2] This will become clear when we use Markov chains for Babbling.) If $|S| < n$ and the final state reached by your random walk has possible outgoing transitions, then **you will not receive full points.**

[2]"It's not a bug, it's a feature!"

4.2 RANDOM210

In this course, we use deterministic randomness. Functions which generate (pseudo-)random numbers are given a seed, and produce a new seed as an additional result. If you request a random number multiple times using the same seed, you will get the same number back each time.

A general rule of thumb for "correct" use of seeds is to only use each seed once. In this lab, incorrect use of seeds likely won't affect correctness, but will affect the quality of the output. If you are unsure of how best to use seeds, ask a TA on Piazza or at office hours.

Here are a few functions from the `RANDOM210` signature which you will find useful. We will be assuming cost bounds for the `MkMersenneTwister` implementation.

- `val fromInt : int -> rand`

  Given some arbitrary integer x, (`fromInt` x) creates a seed in constant time.

- `val randomInt : rand -> ((int * int) option) -> (int * rand)`

  Given a seed r, (`randomInt` r `NONE`) returns (x,r') in constant time, where x is a random integer, and r' is a fresh seed. Alternatively, (`randomInt` r (`SOME` (i,j))) bounds x within the range $[i, j)$. Don't call it with $i = j$... bad things will happen.

- `val randomIntSeq : rand -> ((int * int) option) -> int -> (int Seq.seq * rand)`

  (`randomIntSeq` r `rangeOpt` n) is analogous to (`randomInt` r `rangeOpt`) except that it returns a sequence of n random integers. It has $O(n)$ work and span.

- `val seedSeq : rand -> int -> rand Seq.seq`

  Given a seed r, (`seedSeq` r n) returns a sequence of n fresh seeds in $O(n)$ work and span.

4.3 RTABLE

To aid in the implementation of your Markov chains, we've extended the `TABLE` interface with an additional function:

```
signature RTABLE =
sig
  include TABLE
  structure Rand : RANDOM210
  ...
  val rselect : 'a table -> Rand.rand -> Key.t option
end
```

Given a table T and a seed r, (`rselect T r`) returns either (`SOME` k), if T is nonempty and k is a random key from T, or `NONE`, otherwise. It has $O(\log|T|)$ work and span. All other functions in the `RTABLE` interface are identical to the corresponding functions in `TABLE`.

5 Babble

I spent an interesting evening recently with a grain of salt. –Mark V Shaney

5.1 The Algorithm

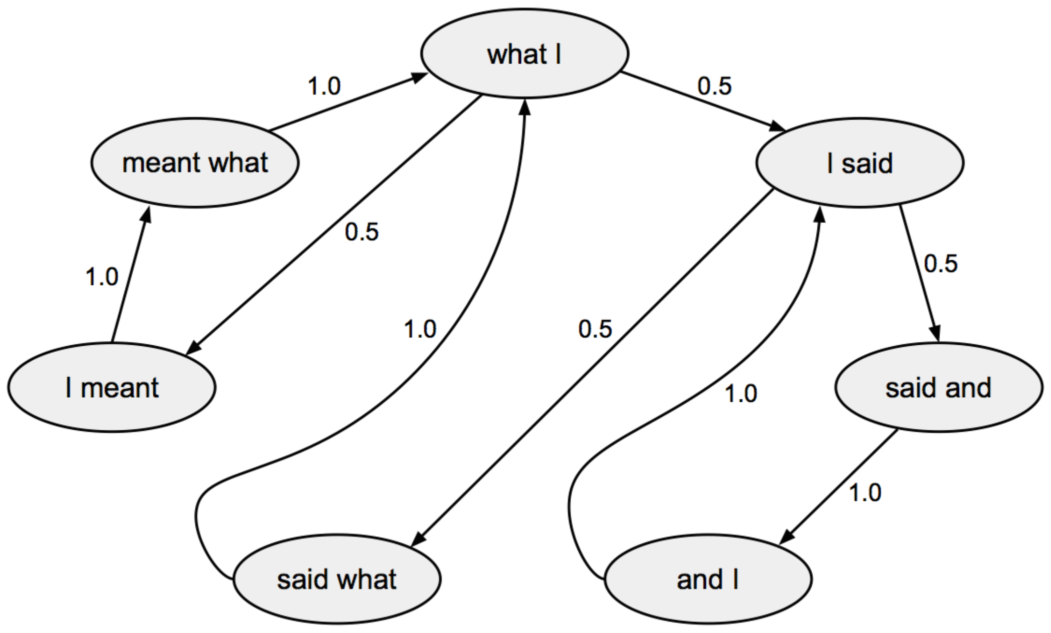
INPUT: A token sequence S and an integer $k \geq 2$

OUTPUT: A token sequence S' which satisfies the k**-gram property**.

We define the k**-gram property** as follows: each contiguous subsequence of length k from S' must also be a valid contiguous subsequence of S. This guarantees that the output is statistically similar to the input while still allowing them to differ from one another.

We can model the input sequence as a Markov chain where states are k-grams. We create a transition from a k-gram G to another k-gram H if G and H overlap maximally in the input – i.e., if $G =$ `subseq S (`i`,`k`)` and $H =$ `subseq S (`i`+1,`k`)`. For example, here is a tokenized input corpus and its corresponding Markov chain for $k = 2$:

Corpus:	"I meant what I said and I said what I meant."
Tokens:	⟨I, meant, what, I, said, and, I, said, what, I, meant⟩
k-grams:	⟨⟨I, meant⟩, ⟨meant, what⟩, ⟨what, I⟩, ⟨I, said⟩, ⟨said, and⟩, ⟨and, I⟩, ⟨said, what⟩⟩



To Babble, we then just randomly walk through this Markov chain and output the non-overlapping token at each step. An example starting with "I said" could be

"I said what I said and I said what I meant what I said."

5.2 Implementation

Task 5.1 (10%). In `MkBabble.sml`, implement the function

```
val chainFromCorpus : Tok.token Seq.seq -> int -> Mkvc.chain
```

Where (`chainFromCorpus` S k) returns the Markov chain which models k-gram transitions within S, as described above. You may assume $|S| > k$. **For full credit**, your implementation must have $O(|S| \log |S|)$ work and $O(\log^2 |S|)$ span.

Task 5.2 (10%). In `MkBabble.sml`, implement the function

```
val babble : Mkvc.chain -> int -> Rand.rand -> Tok.token Seq.seq
```

Where (`babble` (`chainFromCorpus` S k) n r) produces a sequence S' of length at most n which satisfies the k-gram property against S. **For full credit**, (`babble` C n r) must have $O(n \log |C|)$ work and span.

$|S'| = 0$ is acceptable if $n = 0$ or C is empty. $|S'| < n$ is acceptable only if the random walk through C terminated early. As mentioned earlier, this is actually a desired behavior, because the corpus might end in a k-gram which did not appear anywhere else in the corpus.

Because the chosen k-grams overlap, it gets a little messy when assembling the final token sequence. To avoid this, feel free to randomly walk slightly longer than necessary and then drop a few tokens off either the front or the back to make your life easier. The specifics aren't super important as long as the output satisfies the k-gram property.

6 Testing

6.1 Markov Chains

To test your implementation of Markov chains, add tests to `Tests.sml`. Then run some or all of the following at the SML/NJ REPL.

```
- CM.make "sources.cm";
...
- MkvcTester.testBuild ();
- MkvcTester.testMerge ();
- MkvcTester.testShrink ();
- MkvcTester.testIter ();
```

6.2 Babbling

We've defined a function

```
val mix : string list -> int -> int -> int -> string
```

where (`mix corpuses` k n x) produces n Babbled tokens from the specified input corpuses with grams of length k and random seed (`Rand.fromInt` x). (If you're curious, `mix` works by merging Markov chains from separate corpuses.)

`mix` exists multiple times – once for each tokenization strategy. We've provided three for you: `CB.mix` (character tokens), `WB1.mix` (word tokens using spaces and non-printable characters as delimiters), and `WB2.mix` (word tokens using non-alphanumeric characters as delimiters). Try them all! `CB` with $k = 5$ works very well, although both `WB1` and `WB2` perform decently with $k = 2, 3$. You can create your own tokenization strategies, if you'd like – take a look at `MkWordToken1.sml` for an example, and ask on Piazza if you get stuck.

Finally, `mix` is **staged**, meaning that if you partially apply its first two arguments, you can avoid doing the expensive computation of constructing the Markov chains over and over again.

Here are a few examples.

```
- CM.make "sources.cm";
...
- val wordKenn = WB1.mix ["data/kennedy.txt"] 3;
- val charMobyBama = CB.mix ["data/mobydick.txt", "data/obama.txt"] 5;
- print (wordKenn 200 42);
- print (wordKenn 300 15210);
- print (charMobyBama 500 42);
```

7 Written

Write all answers to the questions below in `written.pdf`.

7.1 Fun with Sets

You are so touched by the stories generated by your babbling monkeys that you have decided to publish them in a book. However, your simian friends smartly babbled in parallel, leaving you with the task of merging their sets of babbled documents.

Task 7.1 (5%). Based on the cost specification for Sets, what is the asymptotic work and span for taking the union of two sets one of size n and the other of size $\sqrt{n}$? You can assume the comparison for sets takes constant work. Please give tight big-O bounds and simplify as much as possible.

Task 7.2 (5%). In functional programming, all values are immutable. Therefore, taking $X = A \cup B$ will result in a new set X, and we will still have the original sets A and B.

It seems that since X is at least as large as $n = \max\{|A|, |B|\}$, it would take at least $\Omega(n)$ work to generate a new X, especially since we can't modify A or B. Does this break your analysis above? Explain in a couple sentences how this could be possible (i.e. why our lower bound is not the case).

7.2 Random Questions

Since monkeys are such an effective source of randomness, why not put them to use solving difficult problems? To get some practice ordering monkeys around, you first practice your randomization skills on the problems below.

Task 7.3 (5%). The randomized quicksort algorithm from lecture is run on a sequence S where $|S| = n$. For the sake of simplicity, you may assume that n is divisible by 4.

Consider the *sorted* sequence of elements. What is the probability (as a function of n) that the element at index $\frac{n}{4}$ is compared to both the element at index $\frac{n}{2}$ and the element at index $\frac{3n}{4}$?

Task 7.4 (10%). Consider the following algorithm to find the 3 largest elements in a sequence:

```
fun threeLargest lt b S =
  let
    fun update ((max, max2, max3), new) =
      if lt (new, max3) then (max, max2, max3)
      else if lt (new, max2) then (max, max2, new)
      else if lt (new, max) then (max, new, max2)
      else (new, max, max2)
  in
    iter update (b, b, b) S
  end
```

You may assume that:

- `lt` (x, y) returns `true` if $x < y$, and `false` otherwise.
- `lt` (b, x) is `true` for any $x \neq b$.
- The input sequence is randomly permuted and every element is distinct.
- $|S| > 3$

Determine the expected number of calls to `lt` as a function of $n = |S|$.