CS201 Markov Project Analysis

Keping Wang (netid: kw238)

October 6, 2016

1 How I did the benchmark

It is hard to write an accurate benchmark on the JVM because JVM is an adaptive virtual machine which automatically does optimizations for the source code.[1]

The `BenchMark.java` file provided by the instructor doesn't avoid containing flaws in benchmarking, for example:

1. It lacks warm up rounds. I do not fully understand how JVM and CPU work, but the running time of a piece of good does show great volatility in the first several iterations.

2. It uses unnecessary multithreading. If I want an accurate benchmark for a piece of code, I would really be willing to sacrifice some of my time waiting in front of the computer in exchange for a guaranteed accurate measurement. Multithreading just affects the running time measurement too much. When the number of threads exceeds the number of CPU cores, the threads are slowed down a lot. And when some threads are finished first, the remaining threads will be allocated with more computing power. A fixed thread pool is perhaps better, but still not something that I would use.

3. The running time of `.setTraining()` and `.getRandomText` are measured together (with the confusion of threads), but they could be measured separately.

4. The JVM might have done some optimizations which haven't been taken into account by `BenchMark.java`. But I'm not sure about this.

So the recommended way to write benchmarks for a person without comprehensive knowledge of the JVM is to use a well written framework. Here I chose JMH (Java Microbenchmark Harness)[2], which basically works by generating synthetic benchmark code by reading annotations in the source code.

To use JMH, I had to create a separate Maven project for benchmarking. I wrote `MarkovBench.java` and `MapBench.java` respectively to benchmark the time of the Markov model and map insertion. Besides, I made the original `markov-start-fall16` a Maven project so that I could easily add it as a dependency in my benchmarking Maven project.

[1] `http://www.oracle.com/technetwork/articles/java/architect-benchmarking-2266277.html`

[2] `http://openjdk.java.net/projects/code-tools/jmh/`

One thing to note about measuring `.getRandomText(int T)` is that this method is not always generating a text of size T. To test the running time of generating T characters, if the call stops before generating the required length, I iteratively call `.getRandomText()` until all T length is generated.

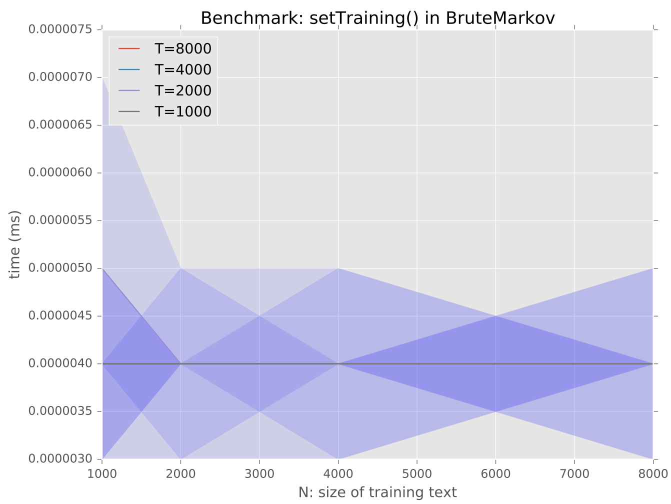
JMH automatically generated benchmark output. I directed the output into CSV files and then visualized the results using Python `pandas` and `matplotlib`. The Python code is in `analysis/BenchAnalysis.ipynb`.

What follows is the running time analysis of the Markov model in section 2 and map insertion in section 3. A preview of the conclusion: all the hypotheses made in the google doc are true.

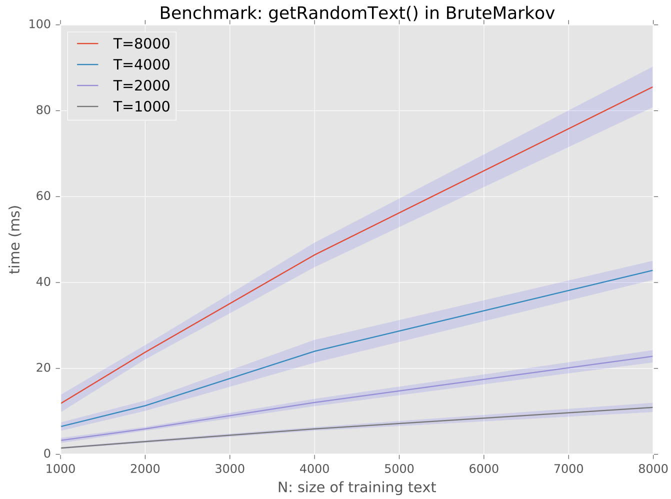
2 Markov Time Analysis

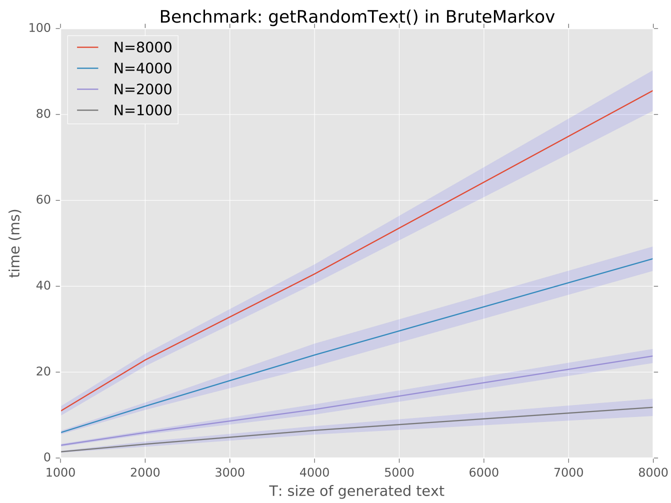
All the running time is illusated with graphs. For all the following graphs, the y axis is the running time in milliseconds (10^{-3}s), and the blue shades in all the following graphs are 99.9% confidence intervals. The benchmarks with respect to N and T use 6-grams $k = 6$. The source training text is `hawthorne.txt`.

The **.setTraining()** method in **BruteMarkov** is $O(1)$ since it does nothing other than passing a reference.

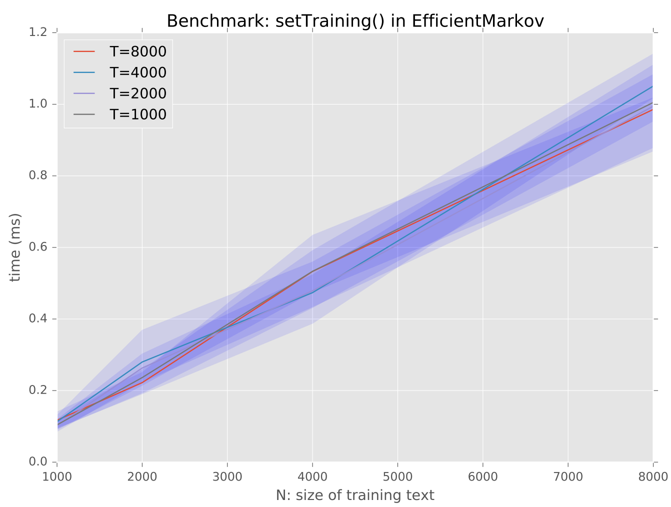


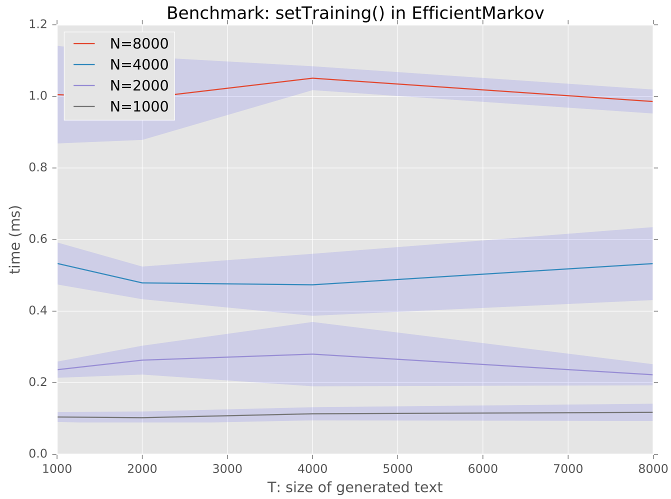
The **.getRandomText()** method in **BruteMarkov** is $O(NT)$, which means the running time is proportional to N and proportional to T.



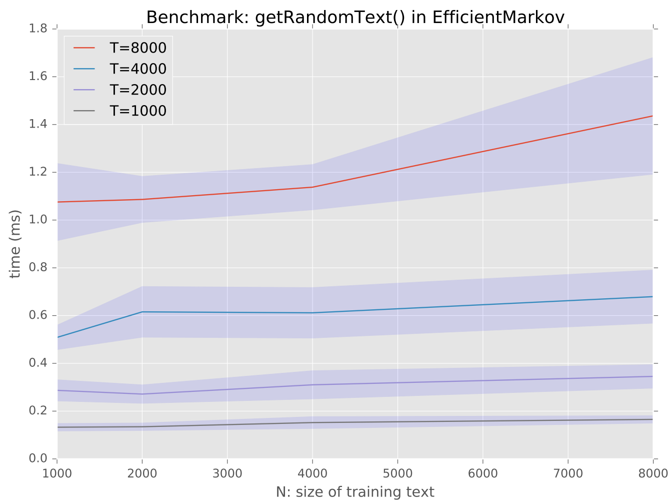


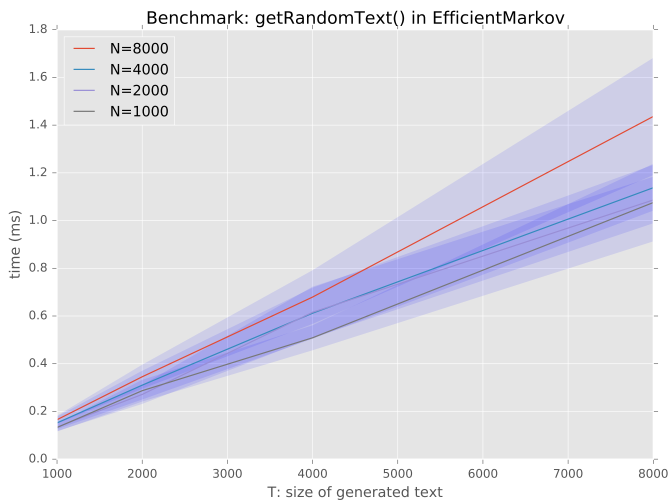
The **.setTraining()** method in **EfficientMarkov** is $O(N)$.



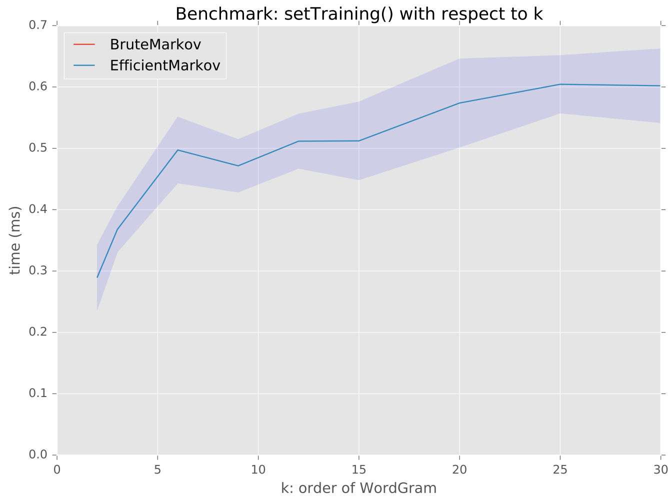


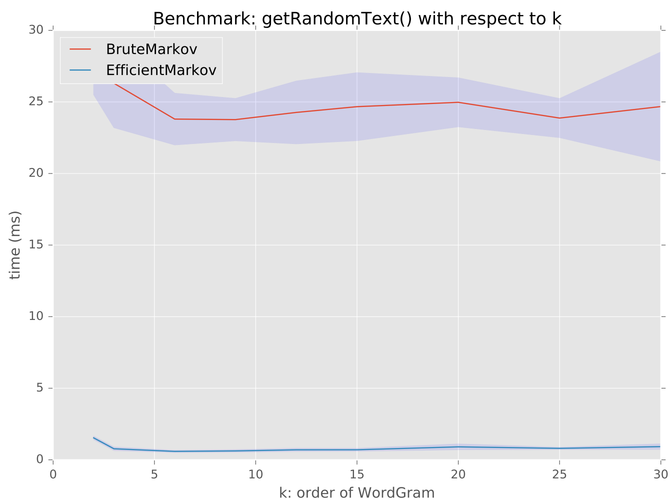
The **.getRandomText()** method in **EfficientMarkov** is $O(T)$.





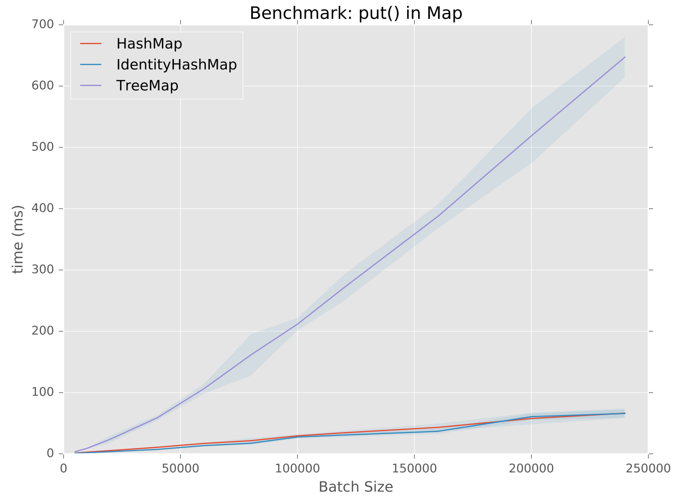
As for k, the running time of some methods are related to k, and the specific relationship depends on the source text used for training. However, all the methods have running time independent of k asymptotically. These graphs has $N = 4000$ and $T = 4000$. (The running time of `.setTraining()` in `BruteMarkov` is so close to 0 that we cannot see the line when it is ploted in the same graph as `.setTrainig()` in `EfficientMarkov`)

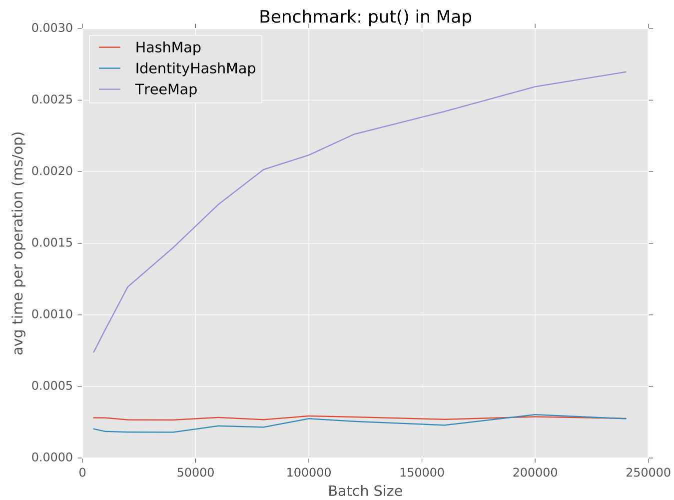




3 Map Insertion Time Analysis

Inserting (calling **.put()** method with) U unique keys is $O(U)$ for **HashMap** $O(U \log U)$ for **TreeMap**. Furthermore, I measured the performance for **IdentityHashMap**. (In Java, `HashMap` is implemented using separate chaining, storing collisions using a `LinkedList`, while `IdentityHashMap` is implemented using linear probing, moving on to the next bucket on collision. Since Java 8, `HashMap` automatically transforms a bucket that is too huge into a BST.)


Benchmark: put() in Map
HashMap
IdentityHashMap
TreeMap
time (ms)
Batch Size
0
100
200
300
400
500
600
700
50000
100000
150000
200000
250000


Benchmark: put() in Map
HashMap
IdentityHashMap
TreeMap
avg time per operation (ms/op)
Batch Size
0.0000
0.0005
0.0010
0.0015
0.0020
0.0025
0.0030
0
50000
100000
150000
200000
250000