Programming Assignment 1 Report

Nathan Friend

COM S 535: Algorithms for Large Data Sets:
Theory and Practice

2018/10/02

For each class that you created, list specifications of all public and private methods that you have written.

The complete specifications of all classes in the io.nathanfriend.coms535.pa1 package are available in an interactive Javadoc format included in this assignment's .zip file. To view the Javadoc site, extract the .zip file and double click on /doc/index.html.

For the classes BloomFilterFNV, BloomFilterMurmur explain the process via which you are generating k-hash values, and the rationale behind your process.

- BloomFilterFNV: During initialization, I generate a number of random "salt" strings – one for each hash function. Then, when adding to or querying the filter, I append each salt to the string before each hashing round. This process has two benefits:
    1. I can reuse the FNV hashing strategy multiple times on the same input string to populate my Bloom filter.
    2. The process is deterministic, as I use the same set of salts throughout the lifetime of the Bloom filter instance. This is essential as the identical hashing process must be used for querying and adding to the filter.
- BloomFilterMurmur: The MumurHash implementation accepts two parameters: the input string and a random seed. For each round of hashing, I use the previous iteration's hash value as the seed for the next. This has the same two advantages as the bullet point above.

The random hash function that you used for the class BloomFilterRan, again explain how you generated k hash values.

The hashing process I used for the BloomFilterRan class was similar to the process used in the BloomFilterFNV class. During class initialization, I generate a series of values for *a* and *b*. I store these values as private members of the BloomFilterRan class. When hashing an input string, I used these *a* and *b* values to generate the final hash values of the input.

The experiment designed to compute false positives and your rationale behind the design of the experiment.

To compare the performance of the three Bloom filter implementations, I created a test program (FalsePositives) that instantiates all three Bloom filters and runs a series of tests against all three filters. Specifically, I generate *setSize* random strings (I used random UUIDs) and insert these strings into all three filters. Then, I run *iterations* tests. Each test generates a random string (another UUID) and tests each filter to see if its .appears() method returns true. If it does, I can assume this was a false positive since the chance of a UUID collision is extremely low. I record the number of false positives each Bloom filter produced and output the results of the test run to the console, along with the theoretical ideal ($0.168^{\text{bitsPerElement}}$) for comparison.

The FalsePositives program takes three arguments that allow the test run to be customized (these can also be viewed by running the program with the --help argument):

- --sizeSize: The size of the set to be added to the filter
- --bitsPerElement: The number of bits per element to allocate in the Bloom filter
- --iterations: The number of tests to run

For all the Bloom filter classes report the false probabilities when bitsPerElement are 4, 8 and 10. How do false positives depend on bitsPerElement? Which filter has smaller false positives? If there is a considerable difference between the false positives, can you explain the difference? How far away are the false positives from the theoretical predictions?

Here is the output for --sizeSize=10000, --bitsPerElement=4, and --iterations=100000:

```
False positive rates for setSize=10000, bitsPerElement=4, and iterations=100000:
------------------------------------------------------------------------------
Theoretical:        14.59% (14587 out of 100000 iterations)
BloomFilterFNV:     17.27% (17266 out of 100000 iterations)
BloomFilterMurmur:  14.83% (14833 out of 100000 iterations)
BloomFilterRan:     20.43% (20428 out of 100000 iterations)
```

For --sizeSize=10000, --bitsPerElement=8, and --iterations=100000:

```
False positive rates for setSize=10000, bitsPerElement=8, and iterations=100000:
------------------------------------------------------------------------------
Theoretical:        2.13% (2128  out of 100000 iterations)
BloomFilterFNV:     4.85% (4850  out of 100000 iterations)
BloomFilterMurmur:  2.25% (2251  out of 100000 iterations)
BloomFilterRan:     7.49% (7493  out of 100000 iterations)
```

For --sizeSize=10000, --bitsPerElement=10, and --iterations=100000:

```
False positive rates for setSize=10000, bitsPerElement=10, and iterations=100000:
------------------------------------------------------------------------------
Theoretical:        0.81% (813   out of 100000 iterations)
BloomFilterFNV:     1.38% (1376  out of 100000 iterations)
BloomFilterMurmur:  0.85% (849   out of 100000 iterations)
BloomFilterRan:     5.36% (5357  out of 100000 iterations)
```

As is apparent from this data, the bitsPerElement parameter has a large effect on the false positive rate of the Bloom filters, regardless of their implementation. The false positive rate drops dramatically as the bitsPerElement parameter increases. Unfortunately, this also means more memory is consumed by the filter.

The BloomFilterMurmur is consistently the best implementation – its false positive rate is very close to the theoretical ideal. The only difference between these three Bloom filter implementations is their choice of hash functions. Because of this, it is reasonable to assume that the Murmur hashing strategy results in the fewest number of collisions and provides the most even hash value distribution.

Describe the how EmpericalComparison is comparing the performances of BloomDifferential and NaiveDifferential. Explain the rationale behind your design.

The EmpericalComparison program tests the performance of the two simulation programs: BloomDifferential and NaiveDifferential. To accomplish this, EmpericalComparison creates an instance of each class. It then run *iterations* retrieval requests against both of these instances. During retrieval, three data points are captured by both classes:

1. The number of file accesses. For example, if a retrieval must access both database.txt and DiffFile.txt, this counts as two accesses.
2. The number of lines read from both database.txt and DiffFile.txt
3. The amount of time spent in retrieval.

Below is an example of the output from an EmpericalComparison test run. This example test run used a set of files that were much smaller than the real files provided with this assignment in order to allow the simulation to include more iterations:

```
NaiveDifferential  performed 1900 file accesses and read 5383315 total lines from the
files. Total time spent: 25.222 seconds
BloomDifferential performed 1001 file accesses and read 4488810 total lines from the
files. Total time spent: 20.765 seconds

BloomDifferential performed 47.32% fewer file accesses, read 16.62% fewer lines, and
performed 17.67% faster.
```

As the data demonstrates, by any measure, the BloomDifferential provides very large performance gains over its naïve counterpart.