# COMS 535 Project 1 Report

Siying Lyu

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

Answer:

I implemented a interface BloomFilter.java for he convenience of using BloomFiilterFNV, BloomFiilterMurmur and BloomFiilterRan in other classes. The CMSTest.java class is for the purpose of completing the report task.

|  |  |
| --- | --- |
| BloomFiilterFNV | BloomFilterFNV(int, int);  functions(String): int[];  add(String): void;  appears(String): boolean;  dataSize(): int;  numHashes(): int;  fnv(String): long |
| BloomFilterMurmur | BloomFilterMurmur(int, int);  functions(String): int[];  add(String): void;  appears(String): boolean;  dataSize(): int;  numHashes(): int;  murmur(String): long |
| BloomFilterRan | BloomFilterRan(int, int);  functions(String): int[];  add(String): void;  appears(String): boolean;  dataSize(): int;  numHashes(): int;  ran(String): long;  prime(int): int;  isPrime(int): boolean |
| FalsePositives | FalsePositive(int, int);  generater(): String;  testfp(String, int): void |
| BloomDifferential | BloomDifferential(File, File);  createFilter(String, int): void;  retrieveRecord(String): String;  setup(String): String[];  checkLine(String): boolean |
| NaiveDifferential | NaiveDifferential(File);  retrieveRecord(String): String;  setup(String): String[];  checkLine(String): boolean |
| EmpericalComparison | EmpericalComparison(File, File, File);  compareTime(String, int, int) |
| CMS | CMS(float, float, ArrayList<String>);  approximateFrequency(String): int;  approximateHeavyHitter(float, float): ArrayList<String>;  fnv(String): long  functions(String): int[]; |

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

Answer:

In both classes, we use:

hash = fnv(str);

hash = (hash % filterSize + i \* hash / filterSize) % filterSize

to get k-hash values. *i* is from 0 to k-1. Since the *i*th hash function is not only the function of str, but also the function of *i.* Therefore, if h1(a) == h1(b), we guarantee that h2(a) != h2(b).

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

Answer:

We randomly generate two prime arrays A and B, with same length. Find the closest prime p which is greater then the filter size. For each str, we do:

x = str.hashCode();

hash = (A[i] \* x + B[i]) % p;

hash = hash % filterSize;

to get k-hash values. *i* is from 0 to k-1. Since A[i] and B[i] are randomly picked, we guarantee that if h1(a) == h1(b), h2(a) != h2(b).

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

Answer:

We need two sets: the set is used to construct the bloom filter. The testSet is used to test whether the string in testSet is in set or not. All the strings in the two sets are generated randomly and we guarantee the testSet only contains those strings that are not in the set. In the testfp method, we add all the strings in the set are added in to the bloom filter and we test if the string from the testSet returns true on bloomfiler.appears() method. If it returns true, it indicates that the bloom filter gives the wrong answer and we count the number of all wrong answers. The false positive is evaluated by the number of all wrong answers divided by the testSet size.

5. 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?

Answer:

|  |  |  |  |
| --- | --- | --- | --- |
|  | 4 bitsPerElement | 8 bitsPerElement | 10 bitsPerElement |
| Theoretical | 0.1449 | 0.0210 | 0.0080 |
| FNV | 0.2199 (0.52) | 0.0714 (2.40) | 0.0505 (5.31) |
| Murmur | 0.2110 (0.46) | 0.0677 (2.22) | 0.0514 (5.42) |
| Random | 0.2194 (0.51) | 0.0655 (2.12) | 0.0499 (5.24) |
| Variance | 2.4617E-05 | 9.0218E-06 | 5.7880E-07 |

Random bloom filter outperforms in both 8 and 10 bitsPerElement cases. Therefore, the random bloom filter is the best. As the bitsPerElement increases, the false probabilities decreases. The difference among the false positives is not dramatic (shown with the variances). This indicates that the hash functions fits the property of if h1(a) == h1(b), h2(a) != h2(b).

The differences between experimental values and theoretical value are represented in the parentheses behind each number. Since the theoretical value drops exponentially, the difference between experimental values and theoretical value grows with the increasing of bitsPerElement, although they decrease in absolute values.

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

Answer:

Space aspect is clear. BloomDifferential create an additional space for Differential.txt. Therefore, only the time is needed to be compare. For BloomDifferential, we initialize the bloom filter (FNV) with the DiffFile.txt (input as diff) and store database.txt (input as data) in a new hashMap. For NaiveDifferential, only the database.txt is stored in a new hashMap. Then we compare the time difference between BloomDifferential and NaiveDifferential on the retrieveRecord(String) method with the test string from grams.txt (input as keys).

7. Create CMS data structure by adding words from the file given file shakespear.txt. Add only words whose length is at least 3 and do not add the words “the” or “The” to the data structure. Take ε = 1/100 and δ = 2-20.

Answer: See class CMSTest.java

8. Using the above built CMS data structure, compute a set L that is h<0.04, 0.03> approximate heavy hitter. Report the following :

– Number of 0.04 heavy hitters that are in L.

Answer: 0

– Number if 0.025 heavy hitters that are in L.

Answer: 1

– Number if items in L that are not 0.04 heavy hitters.

Answer: 1

– Total number of strings that are added to the data structure, and the total number of distinct strings that are added to the data structure.

Answer:

Total number of strings: 683014

Total number of distinct strings: 23485

– An estimate of total memory used to store the CMS data structure.

Answer: The size of CMS is (2/ε)\*log(1/δ) = 200 \* 20. One int takes 32 bits. In total should be 200 \* 20 \* 32 = 128Kb.