Programming Assignment 1

Collin Vincent
Ryan McCullough

Specifications

BloomDifferential.java:

BloomDifferential(BloomFilter filter, String diffPath, String databasePath)

The constructor sets the paths to instance variables for use in other methods and sets the filter instance variable to equal createFilter passing on the bloom filter argument.

createFilter(BloomFilter filter)

uses a buffered reader to loop through every line of DiffFile.txt and extract the keys, entering them into the bloom filter it took in as it goes.

retrieveRecord(String key)

if appears(key) returns true it loops through DiffFile.txt returning the record if found, otherwise it continues on to loop through database.txt, returning the record if it is found there.

NaiveDifferential.java:

NaiveDifferential(String diffPath, String databasePath)

The constructor simply sets the database path to an instance variables for use later. Then it reads the DiffFile into a hash map in main memory.

retrieveRecord(String key)

Checks the hash map in main memory for the key and returns the corresponding record if it exists. Then it opens a new buffered reader and reads database.txt line by line, returning the line if it contains the key.

EmpericalComparison.java:

EmpericalComparison(String keysFilePath, String diffFilePath, String databaseFilePath, int setSize, int bitsPerElement)

The constructor sets instance variables for file paths and size attributes of the bloom filters then sets the keys ArrayList instance variable equal to getKeys(keysFilePath).

getKeys(keysFilePath)

loops through the keys file line by line with a buffered reader collecting the keys and adding them to the ArrayList, then returns the ArrayList.

getBloomFNV()

returns a BloomDifferential that contains a FNV bloom filter using the BloomDifferential and BloomFilterFNV constructors with saved instance variables for arguments.

getBloomMurmur()

same as above but with a murmur bloom filter.

getBloomRan()

same as above but with our BloomFilterRan class.

BloomDiffTime(BloomDifferential bloomDiff, String key)

tracks the time time taken to use retrieveRecord(key) with the given BloomDifferential using System.nanoTime().

NaiveDiffTime(NaiveDifferential ndiff, String key)

same as above but with a NaiveDifferential.

averageBloomTime(BloomDifferential bloomDiff)

initialize a long to keep track of the times, loop through each of the keys in the keys ArrayList and call BloomDiffTime for each one, adding the result to our running total, finally return the running total divided by the size of the keys list.

averageNaiveTime(NaiveDifferential naiveDiff)

same as above but with NaiveDifferential

Main method:

puts the desired values into a EmpericalComparison constructor then it will print out the average time for each bloom filter and the NaiveDiffferential by calling averageBloomTime/averageNaiveTime, as well as the memory usage of each using Runtime.getRuntime().totalMemory().

BloomFilter:

This is just an interface so I'm not going to talk about its methods but I don't want you to think I forgot it. It's just used to program the other classes to so the filters are interchangeable

BloomFilterAbstract:

This abstract class contains all the code that will be used by all three implementations.

public BloomFilterAbstract(int setSize, int bitsPerElement)

This constructor creates the BitSet used as the table. It also stores the setSize and bitsPerElement values into fields, and creates the hash arraylist

private void setHashFunction(Constructor<? Extends BFHash> c)

This function is called by subclass constructors. It passes a constructor function that is used to fill the hash array list.

public void add(String s)

This function will add a String s to the bloom filter. It uses a thread pool to execute the hashing functions.

public boolean appears(String s)

This function takes a String s and returns the blooms filters estimation if the string has been added to the set.

private void startHash(String s, LinkedBlockingQueue<int[]> queue)

This function is used by the add function to start the hashing functions on the thread pool. The queue is used by the BFHash instances to communicate their result back to the main thread.

public int filterSize()

This function returns the size of the BitSet used as the table for this bloom filter.

public int dataSize()

This function returns the number of strings that have been successfully added to the bloom filter.

public int numHashes()

This function returns the number of hash functions used by the bloom filter.

public static void shutdown()

This function closes down the thread pool used by all bloom filters. Since the thread pool is static calling shutdown will close the thread pool for all bloom filters. This function should be called at the end of a program to allow it to exit normally.

BloomFilterFNV

This class implements a BloomFilter that uses the FVN hash function

public BloomFilterFNV(int setSize, int bitsPerElement)

This constructor calls super then calls setHashFunction to set the super class to use the FNVHash class to hash.

BloomFIlterMurmur

This class implements a BloomFilter that uses the FVN hash function

public BloomFilterMurmur(int setSize, int bitsPerElement)

This constructor calls super then calls setHashFunction to set the super class to use the MurHash class to hash.

BloomFilterRan

This class implements a BloomFilter that uses the FVN hash function

public BloomFilterRan(int setSize, int bitsPerElement)

This constructor calls super then calls setHashFunction to set the super class to use the RanHash class to hash.

BFHash

This is an abstract class that defines some functionality for all the hashing classes.

public void run()

This function is used by the thread pool to execute the hashing. It places the resulting int value from the hash into a 2D array with the second element being a in that can be set by the user.

public void setQueue(LinkedBlockingQueue<int[]> q)

This function takes a queue and stores it as a field to be used after this class is ran.

public void setIndex(int i)

This function will take an int and store it in a field that will be placed into the queue along with the hash result.

public void setString(String s)

This function takes a String and places it into a field. This is the string that will be hashed when this class is executed via the run function.

public void generateFunction(int t, int N, double k)

This function is not implemented here but declared to allow extending classes to use these 3 variables to generate a random hash function.

public int getHash()

This function is not implemented here but is declared to allow extending classes to use their hashing function to generate a result.

FNVHash

This class implements the FNV hashing algorithm and extends BFHash

public void generateFunction(int t, int N, double k)

This function sets t*N to a field to be used later to bound the result of the hash function. It then generates a random integer to be used as the offset.

public int getHash()

This function calculates the hash of the string set by the parent class using the FNV hashing algorithm. At the end it takes the result and gets the absolute value incase the cast to integer causes the result to be negative and then modulates it by p to bound the result.

MurHash

This class implements the Murmur hashing algorithm and extends BFHash

public void generateFunction(int t, int N, double k)

This function generates a random integer to be used as the seed. It also sets t*N as a field to be used to bound to result of the hashing function.

public int getHash()

This function calls the hash64 function with the byte array of the string set by the parent class. It then takes the absolute value of the result and modulates by the field set in the constructor to bound the result.

RanHash

This class implements the Ran hashing function defined in the assignment document and extends BFHash

public void generateFunction(int t, int N, double k)

This function generates a random hash function by getting the next largest prime starting at t*N. It then stores t*N in a field to be used to bound the result of the hash later.

private long nextPrim(long p)

This function finds the next largest prime starting at p.

private boolean isPrime(long p)

This function returns true if p is prime and false if p is not prime.

public String toString()

This function returns a string representation of the current state of the class instance.

public int getHash()

This function calculates a hash value from a string set by the parent class as defined in the assignment document. Then modulates by the field set in generateFunction to bound the result.

CMS

public CMS(float epsilon, float delta, ArrayList<String> s)

This function stores the ArrayList s as a field. It then constructs a 2D int array to store the occurrences of strings in s, and a BFHash array to store the random hash functions. It then creates the random hashes and sets their indexes so that the same hash result always maps to the same row even if they finish execution out of order. After that it fills the counter array using the algorithm discussed in class.

public int approximateFrequency(String x)

This function finds and returns the min value of the all the values in the counter array corrisponding the the String x based on the hashed values from the random hash functions.

public ArrayList<String> approximateHeavyHitters(float q, float r)

This function loops through all strings in s and finds if their approximateFrequency is at least qN. If it is then it adds the string to an arraylist and returns the resulting arraylist.

public string toString()

This function returns a string representation of the counter array. This is not practical for large counter arrays and was mainly used on small ones to visuals the behavior of my implementation.

public long averageFrequency()

This function returns the sum of all approximate Frequencies divided by the size of s.

public void shutdown()

This function shuts down the FixedThreadPool used by this class to calculated the hashes asynchronously.

private void startHash(String s)

This function begins the execution of the hash functions in the FixedThreadPool.

FalsePositives

This class test the false positive rate of bloom filters

public static void main(String[] args)

This function reads a file path from the args array and then an optional integer which is the number of words from the file to place in the bloom filters at the start. It then class read file and then constructs 9 bloom filters 3 of each type each with different bitsPerElement values. It also calls FPTest with each bloom filter.

public static void FPTest(BloomFitler r, int n)

This function adds the strings from one arraylist to the BloomFilter r. It then calls r.appears on every string in a second arraylist. It assumes that the two arraylist are disjoint so every true returned is a false positive.

public static void readFile(String pathname, int NUM_STRINGS)

This function reads the file located at the pathname and adds the first NUM_STRINGS words in the file a arraylist. It then adds the rest of the words in the file to a second arraylist as long as they have not already been added to the first arraylist.

Pa1Test

This is the only test class that I am documenting because it's the only one used to answer questions in this report.

public static void main(String[] args)

This function takes a file path from the first argument and epsilon and delta values from the second and third argument. It then reads the file and fills the words in the file into and arraylist as specified in the assignment document. After that it constructs a cms instance with epsilon delta and the word arraylist. After that it enters and infinite loop that reads command from stdin and calls the corresponding functions on the cms instance. This allows me to do a large number of test with the cms without having to reread the file and recalculate the cms everytime.

public static void heavyHitter(CMS cms, float q, float r)

This function just runs the approximateHeavyHitter function with the q and r values but it then takes the result and prints a answer.

public static void fillArrayList(File f)

This function reads the file and uses regex to pull out all words that are at least 3 characters long and only considers alphanumeric characters and apostrophes to be characters. It stores all the words that match the pattern into an arraylist.

Generating K Hash values

FNV

For FNV hash I made a class that stores a random offset Integer. I then create K instances of these classes and store them in an array. The reason I create and store K class instances instead of just K integers is because I use these classes as runnables and execute the hashing in a FixedThreadPool since they do not need to happen synchronously.

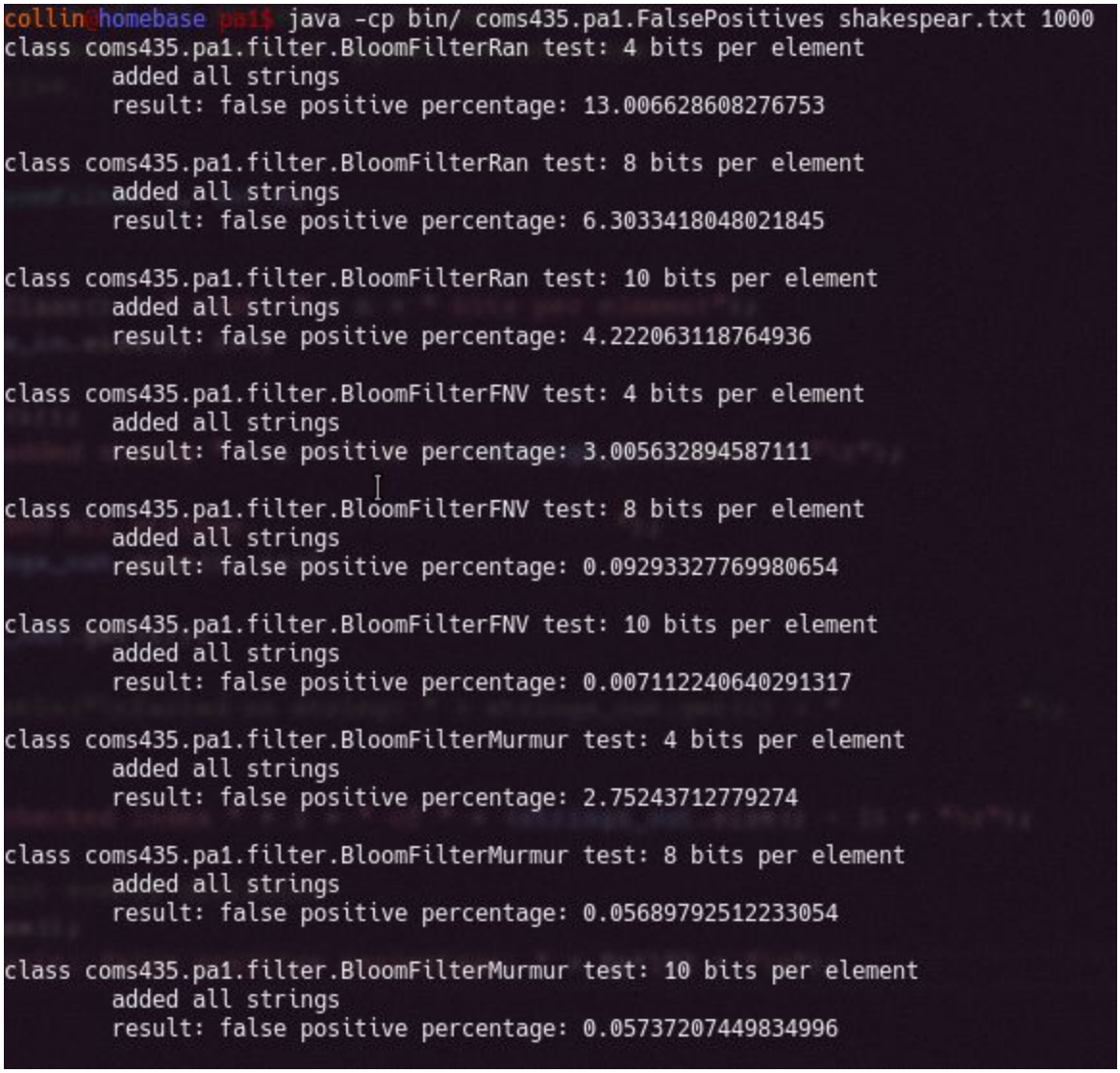
Murmur

For Murmur hash I do that same but i generate a random integer seed that is used.

Ran

For Ran hash I find the next largest prime p based on set size then generate two random ints that are bounded by 0 and p for each class.

False Positive vs Bits Per Element



To test the false positive rate i read the first n words from a text file into an array. The text file path and n are read as arguments to the program. I then read the rest of the file

and add all words that did not appear in the first n into a different array. I add the first array to the bloom filters then test if the words in the second array return true from the appear function. If they do then I know its a false positive and I count it as such. Then at the end i divide the number of false positives by the number of elements in the second array and multiply that by 100 to get a percentage. It's clear that the accuracy increase with the number of bits per element. But it also seems that there are serious diminishing returns. I have tested this code with a number of bits per element, files, and N values and it seems that the filters constantly perform worst to best in this order; Ran, FNV, Murmur.

Empirical Comparison

EmpericalComparison.java is set up to be a customizable comparison tool. The Emperical comparison constructor takes in size variables for the bloom filters and three file paths as arguments: the DiffFile.txt, database.txt, and a path to another file we made called the key file. The key file consists of a set of keys, one per line to be retrieved in the comparison. Users can input a key file with a very large number of keys if they want a very large sample size, or a small key file if they want the comparison to finish in a timely manner with this functionality.

The different bloom filters may have different performances, so we made methods to get BloomDifferential objects using each type of filter, and test each bloom Filter separately in the main method, as well as the NaiveDifferential.

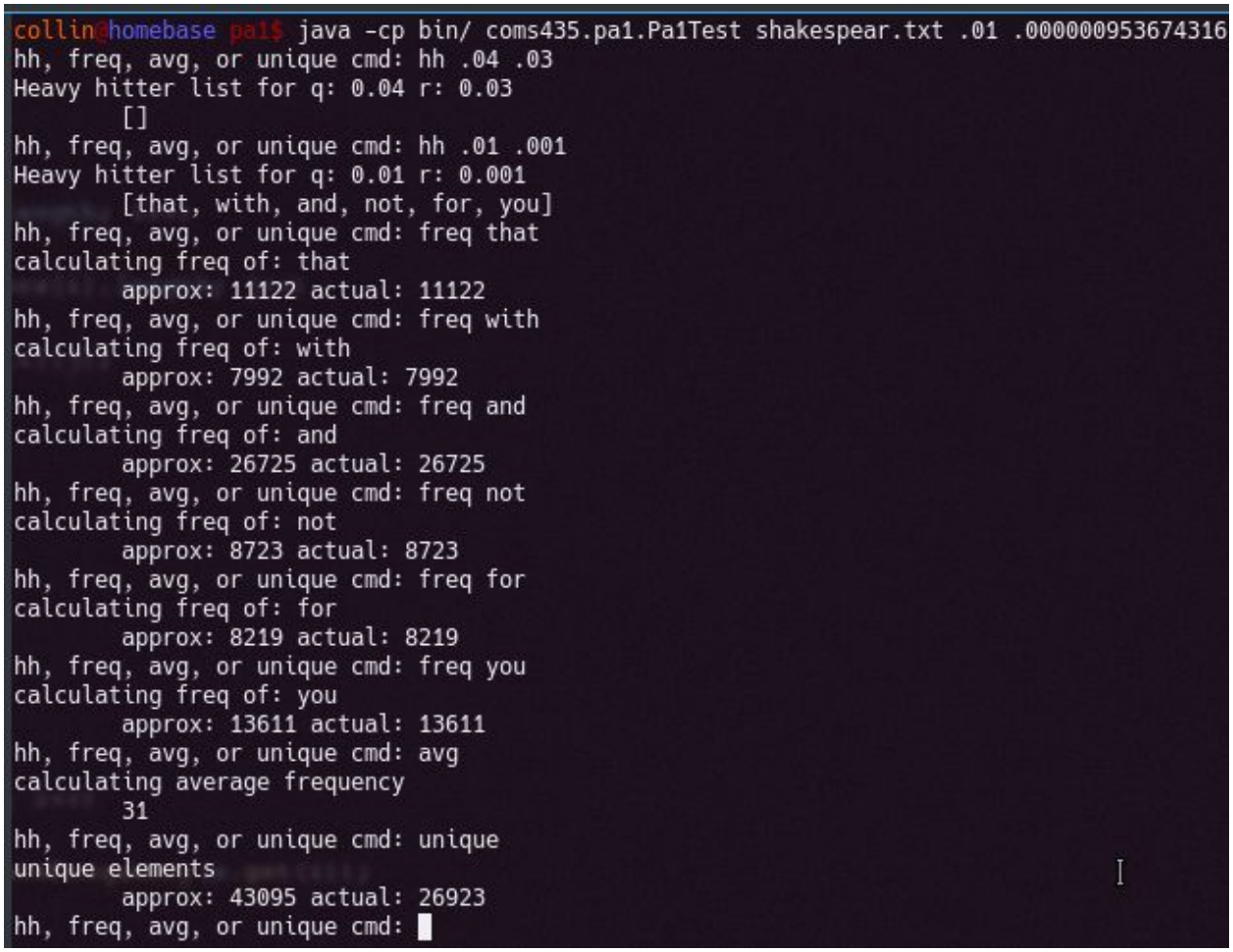
One type of performance that we were looking to measure was the time it took for retrieveRecord to return. We used the system.nanoTime() method to get the time directly before and after the retrieveRecord method runs, and find the difference. To get a more accurate comparison, we chose to average together the retrieveRecord time for each key given in the the keyFile, for each Differential object. We found that Naive Differential was significantly faster than all of the bloom differentials, in one comparison of the time taken to retrieve 50 records chosen at random from grams.txt, Naive Differential was 3300 times faster than the fastest Bloom Filter using Bloom Differential.

If the user wants to collect individual times for each of the keys they input, they can use the BloomDiffTime or NaiveDiffTime methods which take in a key and return a single time.

The other type of performance comparison we tracked was memory usage. We utilized the Runtime.getRuntime().totalMemory() methods to compare the memory usage after all of the keys had been retrieved by the Differential class. When we tested with retrieving 50 records we found that the Naive Differential took 13 time as much memory as the Bloom Differentials. Using a larger Diff file we expect the Naive Differential to use even more memory in comparison to Bloom Differential.

To get a fair comparison we used two key files: easyKeys which contains 5 keys we know are in the diff file and databaseKeys which contains the first 50 keys in grams.txt. For easyKeys we expected the NaiveDifferential to be faster than the BloomDifferential and wanted to test that assumption. databaseKeys is meant to be a more fair comparison. We chose to use the first 50 keys so that the comparison would finish in a timely manner. The combination of the two give a mix of keys found in the DiffFile and keys found in the database proportional to totals in each of the files.

Shakespeare CMS



.04 Heavy Hitters

When calculating HH<.04,.03> i get an empty set. I found the most frequent items with HH<.01,.01> as shown in the picture above. The most frequent among them is the word “and”. It has 11122 occurrences meaning it has a .0388 frequency in the list. Since .0388 is not greater than .04 it is not included in set L. now this could be a result of my not perfectly understanding the heavy hitter concept, but as far as I can tell the heavy hitter set must have all elements that

have a frequency at least q. I am pretty confident in my frequency calculations because I actually count the real frequency and store it in a hashmap while reading the file. That's what the actual value represents in my programs output. This could be the result of malformed regex I suppose.

.025 Heavy Hitters

Let’s assume i did have a list with elements and talk about this question theoretically. All of the elements of the set would have to at least be .03 heavy hitters for them to exist in the set so every element must be a .025 Heavy hitter.

Not .04 Heavy Hitters

I cannot calculate the the actual set of elements that are not actually .04 heavy hitters because I don’t have a string set. Theoretically there could be some elements in the approximate heavy hitter list that are not actually .04 heavy hitters because there is a δ amount of inaccuracy. You would have to go through the set and find each actually occurence number, then divide that by the set size. If the result is less than .04 than that element in the approximate list is not actually a .04 heavy hitter.

Total Strings and Distinct Strings

After reading the file there are 688137 words, and 26725 unique words with lower case conversion. I calculated the actual unique word value by taking the length of the key list in the hashset that I used, and the approximate value by taking the size of the set stored in the cms. I trim this arraylist down from the original by only placing a string in the set if the approximate frequency before it was placed in the set subtracted by the probable overestimate is less than 1. The issue is that it's not very accurate. The total words is just the size of the array list given to cms.

Total Memory

From watching top I can see that the java program running the CMS for this was using 8.8% of my systems ram which is 7925 Megabytes meaning it used a total of 697.4 Megabytes. This includes the memory used to store the strings in the array list and a hashset that i used for comparing answers about frequency and unique strings.

Bibliography

"Bloom Filter." *Wikipedia*, Wikimedia Foundation, 20 Sept. 2018, en.wikipedia.org/wiki/Bloom_filter.
"Fowler–Noll–Vo Hash Function." *Wikipedia*, Wikimedia Foundation, 22 Apr. 2018, en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo_hash_function.

"In Java, What Is the Best Way to Determine the Size of an Object?" *Stack Overflow*, stackoverflow.com/questions/52353/in-java-what-is-the-best-way-to-determine-the-size-of-an-object.

"What Is the Time Complexity of the Algorithm to Check If a Number Is Prime?" *Software Engineering Stack Exchange*, softwareengineering.stackexchange.com/questions/197374/what-is-the-time-complexity-of-the-algorithm-to-check-if-a-number-is-prime.

yoavfreund. "Bloom Filters." *YouTube*, YouTube, 16 May 2012, www.youtube.com/watch?v=bEmBh1HtYrw.