COSC548

Streaming Algorithm

Implementations for "An Optimal Algorithm for l1-Heavy Hitters in Insertion Streams and Related Problems"

Project Report

1. Team Member (group of 2)

- Name: Yinzhi Xi
  NetId: yx157
- Name: Jiarong Yu
  NetId: jy576

2. Project Introduction

We finished an implementation-focused project.

The paper we read is:
An Optimal Algorithm for ℓ1-Heavy Hitters in Insertion Streams and Related Problems *from Arnab Bhattacharyya, Palash Dey, and David P. Woodruff*
https://arxiv.org/pdf/1603.00213.pdf

This paper introduces 3 relative algorithms in detail:

1) A simpler, near-optimal algorithm for (ε, φ)-List heavy hitters
2) An optimal algorithm for (ε, φ)-List heavy hitters
3) List Heavy Hitters for ε-Minimum

As the 3rd algorithm(ε-Minimum) is to find top k smallest(its goal and evaluation method is different from the first 2 algorithms) and consider the workload, we choose the first 2 algorithms, which are both a modified version of the Misra-Gries algorithm and Misra-Gries algorithm itself to implement and make comparisons.

The first two algorithms in this paper are both for solving **(ε, φ)-List heavy hitters** problem. In the following parts of this project, we will simplistically call them Algorithm 1 and Algorithm 2. The **Misra-Gries** algorithm will be called as Algorithm 3.

For the above 3 algorithms, the input of them is a stream S of length m over U=[n].
Let f(x) be the frequency of x∈U in S.
The output of them is :
A set $X \subseteq U$ and a function $\hat{f} : X \to N$ such that if $f(x) >= \varphi m$, then $x \in X$ and $f(x) - \varepsilon m < \hat{f}(x) < f(x) + \varepsilon m$, and if $f(y) <= (\varphi-\varepsilon)m$, then y not $\in X$ for every $x,y \in U$.

We choose a suitable dataset, which contains 10 million English words and write a Python program to for word-count to calculate exact term frequency as the baseline. Then we implemented the 2 algorithms in this paper and the Misra-Gries algorithm.

In the process of studying the 2 algorithms in this paper, we have met many difficulties and problem, but we have successfully overcome most of them. We have seriously tested what worked and what did not, and have done a lot of reason analysis and comparative experiments under different parameters.

We carefully do comparisons for their runtime, space, accuracy, etc. and study their performance in contract of the chosen baseline. Then we generate our own conclusion which agree to the conclusions in paper.

3. Dataset

Wiki dump:
We download the Wikipedia dataset. The whole sizeof this data set is around 5 GB.
After preprocessing like deleting non-English words, removing special symbols etc. and split sentences into single English words. Then we use a word-count program to calculate the result as the baseline, which cost several hours.
The file that after preprocessing has the size of 1.9GB, and contains 321,320,640 lines. Each line is a single English term.
We run the whole file with Algorithm 1 for several times and have the result. But as for Algorithm 2, our machine cannot provide enough resource to run such a big dataset, so we choose a smaller one, a subset of the data set, which has the size of 58.8 MB, and contains 10,000,000 words for comparison.

4. Programming Language & Machine

We use **Java** for the implementation of the 3 algorithms.
We use **Python** for word-count to calculate exact term frequency as the baseline.
The machine we use is MacBook Pro, with OS system.

5. Algorithms Implementation & Comparison

5.1 Baseline: exact results, for basic comparison

We use Python to write a program for preprocessing and calculate the lines of English terms.

We also write a word-count Python program to calculate the result as the baseline with our dataset and use this approach as baseline. Since this approach is deterministic and we could use this result to check if our approach fits the error requirements.

List the top 10 results in the big dataset as an example:

the 17034059
of 11270916
in 8976692
and 8583828
a 5862106
ref 5156109
to 5052359
category 3527028
for 3404304
he 3316266

List the top 10 results in the small dataset as an example:

the 595882
of 320030
and 268328
in 263376
a 204458
to 184619
ref 161371
for 110893
was 103802
on 93354

5.2 Algorithm 1: Interpretation, Explanation & Implementation details

- **Interpretation**

Algorithm 1 is a randomized one-pass algorithm for the (ε, φ)-List heavy hitters problem which succeeds with probability at least 1 − δ using O(ε^−1(log ε^−1 + log log δ^−1) + φ^−1 log n + log log m) bits of space. Moreover, Algorithm 1 has an update time of O(1) and reporting time linear in its output size.

We use a modified version of the Misra-Gries algorithm to estimate the frequencies of items in S, each update is with the probability p. The length of the table in the Misra-Gries algorithm is 1/ε. We pick a hash function h randomly. Instead of storing the id of any item x in the Misra-Gries table, we only store the hash h(x) of the id x. We also store the ids (not the hash of the id) of the items with highest 1/φ values in T1 in another table T2. Moreover, we always maintain the table T2 consistent with the table T1 in the sense that the ith highest valued key in T1 is the hash of the ith id in T2. As we use Strings as the id, table T2 stores English terms as Strings.

Output the terms in T2 and corresponding values in T1.

- **Explanation**

This algorithm is a modified & near optimal algorithm based on the Misra-Gries algorithm.

The main difference is, in our algorithm 1, it uses 2 tables to store data. Table T1 to store hash of items and corresponding values, and table T2 to store original items with exact the same order of T1.

Another difference is, algorithm 1 runs with probability p, which is related to ε, δ, m.

As the size of T1 is 1/ε, and the size of T2 is 1/φ. As φ > ε is always true, the size of output should be the same with the size of T2.

- **Implementation details (Parameter and initialization, etc.)**

1) Parameter and initialization

In Algorithm 1, a parameter we have already know is the stream length m, which is 321,320,640 in the big dataset, 10,000,000 in the small one.

At the same time, the parameters that we need to set at first are 3 parameters: φ, ε, δ. Other parameters can all be generated from the choice of the 3 parameters.

Parameters can be generated:
l: the sample size l from the stream S.
p: the probability that x will be updated.

In fact, we took a lot of detours on the choice of parameters. There are several important problems we want to point out.

Firstly, we consider the choice of l, since p = 6l/m, in the small dataset, m = 10,000,000. In order to have a reasonable p, that is, p is better in the range of 0~1. And it is obvious that as p is close to 1, the values of result will be more likely close to the the true values. At the same time, higher p needs smaller ε, which means the results should have higher accuracy. Besides, l is also related to δ, which is also in the range of 0~1, as the algorithm 1 succeeds with probability at least 1 − δ. As for φ, φ is also in 0~1, and φ should satisfy φ>ε.

So after some calculation and test, I find that with m = 10,000,000, a reasonable ε is likely between 0.003 to 0.007. Based on that, we test some parameters and see the performance of algorithm 1.

2) Generation Details of Hash Functions

- Generating integer hashcode

One problem for us is, our dataset stores a bunch of data in Strings, and we need to transform them to integers as randomly as possible. We did some research and find using the inner method *.hashCode()* of Object Class in Java can get an integer hashcode of any kinds of data. After we read the source code, we find the *.hashCode()* method is a good choice, its randomness is good. But the return number include both negative integers and positive integers, so I use hashcode & 0x7FFFFFFF to generate a new hashcode that is always not negative, and keep its randomness.

- Generating Hash Functions

In algorithm 1 and algorithm 2, we both need to generate a (series of) hash function.

We pick a hash function h uniformly at random from a universal family $H = \{h|h : [n] \rightarrow \lceil 4l^2/\delta \rceil\}$ of hash functions of size $|H| = O(n^2)$. Note that picking a hash function h uniformly at random from H can be done using $O(\log n)$ bits of space. Lemma 2 in the paper shows that there are no collisions in S under this hash function h with probability at least $1-\delta/3$.

In order to generate a universal family $H = \{h|h : [n] \rightarrow \lceil 4l^2/\delta \rceil\}$, we use the function:

$$h(x) = ((ax + b) \bmod p) \bmod m$$

In this function, m means the mapped hash table size we choose, as h maps from n to $4*l^2/\delta$, which means $m = 4*l^2/\delta$. p is a prime we choose. In the baseline, the result shows the big dataset contains 88,234 different words while the small dataset contains 55,039 different words. Considering that, we should choose a prime larger than 88,234 in order to avoid hash collisions as possible as we can. And since the parameter a can be chosen from {1, 2, ……, p-1}, b can be chosen from {0, 1, ……, p-1}, we use the *Random* class of Java to randomly generate an {a, b} pair each time we want to generate a new hash function. So we finally choose 99,991 as the prime p. At the same time, after some calculation test, we find it has very little probability that hashRange = $4l^2/\delta$ can be smaller than the prime p. So the final range of hashed item should mainly based on p.

So the total process is: firstly we calculate the hashCode of an item (String), then we generate a random hash function from a universal hash family. Let the input be item x's hashcode, and then calculate the final hashcode h(x), which is a positive integer less or equal to 99,991.

3) Math problems

Claim: this paper use log with e as base.
Prove: We can see their proof of Theorem 1 states that:

$$\Pr[X \leqslant \ell \text{ or } X \geqslant 11\ell] \leqslant \Pr[|X - \mathbb{E}[X]| \geqslant 5\ell] \leqslant \delta/3$$

By applying Chernoff bound we can find that e is the base of log. Since the equation of Chernoff bound is:

$$\mathbb{P}(|X-\mu| \geq \delta\mu) \leq 2e^{-\mu\delta^2/3} \quad \text{for all } 0 < \delta < 1.$$

4) Other specific implementations

When we try to implement the algorithm 1, we tried to do some specific implementations, that is our understanding for somewhere unclear.

- Data Structure

The main data structure used is for T1 and T2. As T1 is a obvious <Integer, Integer> format Map to store <h(x), value>, and should maintain the order. So we choose to use Map.entry() to iterate the Map in order. T2 only stores the original terms, and should keep in a sort of order, so we choose List<String> for T2.

- Our implementation for some specific pseudo-codes

[line 9] Perform Misra-Gries update using h(x) maintaining T1 sorted by values.
As we need to keep a map in order. At first we think of using TreeMap, but TreeMap can only make the map sort by key. In order to make the map sort by value, we write a function to keep the order of T1 after each MG update.

[line 11] if xi is not in T2 then
Firstly we think xi is a typo error because the author use x elsewhere in this paper to represent item.
Then there exists a bug in this line and we cannot just judge whether x is in T2 or not.
As considering whether x is in T2, we need to consider about hash collision. That means, in some situation, x is not in T2, but previously y is in T2 and h(x) = h(y), so T1 only stores one item<h(x)(=h(y)), x’s value+y’s value> while T2 stored y but with this pseudocode it will also store x and result in non-correspondence for T1 and T2 and more bugs in the following processing codes. We write a conditional statement to avoid such situation.

[line 13] For y in T2 such that h(y) is not among the highest 1/φ valued items in T1, replace y with x
We know that after a Misra-Gries update, it’s possible that not just one h(y) will fall out of the highest 1/φ valued items in T1. So we need to make a few modifications. That is, drop all y that h(y) is not among the highest 1/φ valued items in T1, and then add x to T2.

[line 16] Ensure that elements in T2 are ordered according to corresponding values in T1.
Compare elements in T1 and T2 and make sure T2 is ordered as the highest 1/φ valued items in T1.

- Space and Time Usage

In this algorithm we use a Map<Integer, Integer> to store T1, a List<String> to store T2.

T1: (ε^−1)*64*8 bit
T2: (ϕ^−1)*16*8 bit

5.3 Algorithm 2: Interpretation, Explanation & Implementation details

- Interpretation

Algorithm 2 is also a randomized one-pass algorithm for the (ε, φ)-List heavy hitters problem which succeeds with probability at least 2/3 using (ε^−1 log ϕ^−1 + ϕ^−1 log n + log log m)bits of space if n = ω(ε^−1). Moreover, Algorithm 2 has an update time of O(1) and reporting time linear in its output size.

This algorithm also performs Misra-Gries update on T1. For every hashed x with probability ε increment T2 and then calculate t = ⌊log(10−6T2[i, j] 2)⌋ and only if when t >= 0 With probability p, increment T3.

Using number of 200 log(12ϕ^−1) hash functions to calculate approximate frequency and use median as output if the frequency is more than (ϕ − ε/2)s.

- Implementation details (Parameter and initialization, etc.)

1) Parameter and initialization

Paper stated that this algorithm running under knowing the length m of stream S. And it set sampling length = (10^5)*(φ^-2), which means that if we choose φ<=0.01 the sampling length will more than 1 billion. Since our dataset is not so large, some variable need to slightly change to fix smaller dataset. Since l>>m, s will equal to m.

Since dataset is much smaller than a billion if we still choose probability ε, increment T2[i, j], the algorithm tend to output null.
To fit our smaller dataset I try to modified this probability and found that if we choose this probability condition from ε to 10*ε, the performance will much better than original.

The table size is ε and φ concerned and we need at least 100*(ε^-1) memory to record T2 and T3. We can not choose very small ε and φ since memory limited by our laptop. The Min φ could be accepted by our laptop for algorithm 2 is φ=0.01.

Way to reduce reporting time: If we maintain fj and f within insertion operation instead of calculate them in reporting period, it would save time a lot. But this will increase space usage since we need to keep fj and f in the memory. In our code we do not use this method to improve the running time since limitation of memory.

2) Generation Details of Hash Functions

This is basically the same with algorithm 1. But the algorithm 2 should generate the number of 200 log(12ϕ^−1) hash functions while in algorithm 1 we only need to generate 1 hash function.

3) Math problems

This algorithm also use log with e as base. This is actually very similar to Algorithm 1, as algorithm 2 also use Chernoff Bound, so similarly, we know we should use e as the base of log.

4) Other specific implementations

- Implement of median (f1, . . . , ˆf10 log(ϕ^−1))

In our code we designed we way to calculate the median for frequency. First one is use array to store frequency and apply divide and conquer to algorithm to get frequency. The other is use priority queue to store top 10*log(ϕ^-1) frequency and get median of them.
For output dataset X we do not use any data structure to store them but write into disk directly for saving memory concerned.

- Space and Time Usage

In this algorithm we use an Arraylist to store T1, a 2-dimensional array to store T2, a 3-dimensional array to store T3 and priority queue to store frequency for calculation.

T1: 2(ϕ^−1)*2*8 bit
T2: 100(ε^−1)*200 log(12ϕ^−1) * 8 bit

T3: $100\varepsilon^{-1} * 200 \log(12\phi^{-1}) * 4 \log(\varepsilon^{-1}) * 8$ bit
Queue: $10 \log(\phi^{-1}) * 8$ bit

5.4 Algorithm 3 (Misra-Gries)

- **Interpretation & Explanation**

Misra-Gries algorithm is used to solve the frequent elements problem in the data stream model. That is, given a long stream S of input, the Misra-Gries algorithm can be used to compute the value that makes up a majority of the stream. The input is a long stream S of size m. The output is a table which has items from the stream as the keys, and estimates of their frequency as the corresponding values. It takes a parameter k which determines the size of the table, which impacts both the quality of the estimates and the amount of memory used.

This algorithm uses O(k(log(m)+log(n))) space, where n is the maximum value in the stream and m is the length of the stream.

- **Pseudocode**

```
Input:
    A positive integer k
    A finite sequence s taking values in the range 1,2,...,m
output: An associative array A with frequency estimates for each item in s

A := new (empty) associative array
while s is not empty:
    take a value i from s
    if i is in keys(A):
        A[i] := A[i] + 1
    else if |keys(A)| < k - 1:
        A[i] := 1
    else:
        for each K in keys(A):
            A[K] := A[K] - 1
            if A[K] = 0:
                remove K from keys(A)
return A
```

As it is very easy to implement, we will not give more explanation and implementation details in this report.

6. Results & Discussion

- Algorithm 1:
  Since algorithm 1 can run the big dataset within several hours, it is not realistic for us to run it with the big dataset for multiple times. So I run the algorithm 1 for 1 time each with different parameters with the big dataset.(m = 321,320,640)

φ	ε	δ	p	l	T1 size (bit)	T2 size (bit)	Time (ms)	Space (bit)	Error
0.01	0.001	0.001	0.974673	52,197,088	1000*64*8	100*16*8	22,125,214	~T1 space	/
0.01	0.001	0.01	0.716697	38,381,577	1000*64*8	100*16*8	10,630,087	~T1 space	/
0.01	0.0008	0.1	0.716751	38,384,480	1250*64*8	100*16*8	11,906,502	~T1 space	/
0.02	0.0008	0.02	0.998497	53,472,960	1250*64*8	50*16*8	23,036,080	~T1 space	/

We write the codes of evaluation, but we don't calculate the error above because the times is too small to evaluate whether its success rate is >= 1-δ.

Then we test the same algorithm with the small dataset(m = 10,000,000). We test this algorithm with several different parameters each for 10 times.

φ	ε	δ	p	l	T1 size	T2 size	Average Time	Space	Error
0.01	0.005	0.01	0.921158	1,535,263	200*64*8	100*16*8	159,698	~T1 space	0
0.01	0.01	0.01	0.230289	383,816	100*64*8	100*16*8	15,544	~T1 space	0.2
0.02	0.007	0.001	0.639148	1,065,247	142*64*8	50*16*8	45,188	~T1 space	0
0.02	0.01	0.001	0.313183	521,970	100*64*8	50*16*8	24,576	~T1 space	0.1

We find in the result, the time and space are good, but some error rate is higher than 1-δ. However, it is possible because that running only 10 times is too small to evaluate error rate. After carefully comparison, we found every time our generated output has exactly the same top words in the same order with the true word-count result. Some errors occur for values out-of-range. Generally, when δ is small, the accuracy is high.

At the same time, we can find as p goes down, the time needed deceases quickly. In order to make the output values more accurate, we need to make p be very close to 1, and make δ to become small.

Also, there is something I need to point out when calculating whether the output is qualified. As T2 always output 1/φ items. It is probably that there doesn't have 1/φ qualified items. So we need to consider such situation and evaluate the error rate very carefully.

- Algorithm 2:
  Since this algorithm is not sorted the output frequency. For comparison convenience, I sorted the result with the same ordering of baseline besides the algorithm.

  For φ = 0.01, ε = 0.005
  It has probability to output null with small dataset, but after small modified which mentioned in 5.3 the result is also good. When ε<0.005, test results showed as followed with small dataset. We just list a few of top terms.

  the 489724
  of 309412
  and 228742
  in 243776
  a 189992
  to 162312
  ref 131441
  for 109283
  was 93234
  on 72423
  cite 67823
  as 70993
  he 77341
  with 73212
  his 68712
  film 61234
  that 31273
  by 51223
  at 55231
  category 51293

  For φ = 0.01, ε = 0.01
  the 567213
  of 312342
  and 238941

in 259234
a 200034
to 172342
ref 152374
for 123394
was 93864
on 73485
cite 77920
as 72849
he 74234
with 76931
his 73058
film 75321
that 65312
by 60942
at 46723
category 50173

For $\varphi = 0.02$, $\varepsilon = 0.01$
the 623841
of 332041
and 275340
in 258276
a 210342
to 178341
ref 178234
for 148752
was 91234
on 104502
cite 89341
as 76234
he 76305
with 93412
his 67398
film 45931
(we found that “by at category” is not show on the result)

For $\varphi = 0.02$, $\varepsilon = 0.02$
the 643861
of 351034
and 298710
in 266924
a 210423

to 193523
ref 182342
for 150923
was 109342
on 99832
cite 88731
as 82341
he 78012
with 71234
his 60231
film 66234
(we found that “by at category” is not show on the result)

φ	ε	p	l	T1 size	T2 size	T3 size	Time (ms)	Space	Error
0.01	0.005	1	-	3200	5*10^7	4*10^8	7934683	~T3	/
0.01	0.01	1	-	3200	5*10^7	4*10^8	7736090	~T3	/
0.02	0.02	1	-	1600	2*10^7	1.5*10^8	13776093	~T3	/
0.02	0.01	1	-	1600	2*10^7	1.5*10^8	14766233	~T3	/

As mentioned above, with space limitation it is hard to run algorithm with $\varphi<0.01$, so some result may be hard to compare with algorithm 1. But it is true when streaming is extremely large, algorithm 2 use less space compared with algorithms for the same accuracy and constran, which has been proved in the paper.

It seems that under small dataset size, the output result of this algorithm is mostly less than the actual result. But we can not run the algorithm 10k times so may be a coincidence. If so, it may be because the dataset is too small for the paper’s assumption. It also shows that the result fit the algorithm assumption for the output function fˆ :
X → N such that if f(x) >= φm, then x ∈ X and f(x) − εm < fˆ(x) < f(x) + εm, and if f(y) <= (φ−ε)m, then y not ∈X for every x,y ∈ U.median(ˆf1, . . . , ˆf10 log φ^−1)

- Algorithm 3: Misra-Gries Algorithm

Running Misra-Gries Algorithm is very efficient. We can run the big dataset(m = 321,320,640) in less than 1 minute. The running result is as follows:

k	ε	Time	Space	Error

		(ms)	(bit)	
100	0.01	44,296	100*64*8	/
1000	0.001	40,962	1000*64*8	/
2000	0.0005	44,222	2000*64*8	/
5000	0.0002	40,618	5000*64*8	/
10000	0.0001	40,805	10000*64*8	/

From the results, we can see the running time doesn’t change a lot with different k, which means the number of counters. And the space used is linearly related to k. As m increases, k should be increased, and the space used will also become bigger.

7. Problems & Errors (in the paper)

We find 2 Problems in the pseudocode of Algorithm 1, which has been pointed out in implementation details of algorithm 1(5.2) before. We will clarify them again in this part:

- **Typo Error**
  [line 11] *if xi is not in T2 then*

  We think xi is a typo error because the author use x elsewhere in this paper to represent item.

- **Imperfect statements**
  [line 11] *if xi is not in T2 then*

  In the same row. We think there exists a bug in this line and we cannot just judge whether x is in T2 or not.
  As considering whether x is in T2, we need to consider about hash collision. That means, in some situation, x is not in T2, but previously y is in T2 and h(x) = h(y), so T1 only stores one item<h(x)(=h(y)), x’s value+y’s value> while T2 stored y but with this pseudocode it will also store x and result in non-correspondence for T1 and T2 and more bugs in the following processing codes. We write a conditional statement to avoid such situation by simply drop x because we already have y in T2 and corresponding h(y) in T1. Also we can try to avoid hash collision, but if we don’t consider such situation, it will lead to bugs, which happen with a small probability.

8. Conclusion

We implement 2 algorithms in this paper and the Misra-Gries algorithm, and compare their performance. The results mainly accord with the conclusion of the article. We can conclude that the 2 algorithms are suitable for usage in large dataset, in which situation, the space cost will be saved a lot, especially in algorithm 2. But they are much more time-consuming than Misra-Gries.

Under the circumstance of with large enough dataset, algorithm 1 and 2 save more memory than algorithm 3 (Misra-Gries). But with our dataset limitation(m is not big enough), it's difficult for us to use a large enough dataset to prove this conclusion with results, that means it is hard to show that algorithm 1 and 2 is more space-saving than 3 with small dataset.

9. Reference

[1] Arnab Bhattacharyya, Palash Dey, and David P. Woodruff. An optimal algorithm for l1-heavy hitters in insertion streams and related problems. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, 2016.

[2] Misra, J.; Gries, David. "Finding repeated elements". *Science of Computer Programming*. 2 (2): 143–152. doi:10.1016/0167-6423(82)90012-0.

[3] https://en.wikipedia.org/wiki/Misra-Gries_summary