Reproducing Results from: Constant Time Updates in Hierarchical Heavy-Hitters

Tommy Fan and Austin Poore

ABSTRACT

We present and explain our efforts to reproduce key results from "Constant Time Updates in Hierarchical Heavy-Hitters" [2] for CS 244, a graduate networking course at Stanford University. In particular, we run several experiments using code from the original authors to attempt to reproduce their performance and error metrics on our own servers. In addition, we present an evaluation of our own implementation of the algorithm in the one-dimensional setting.

ACM Reference format:
Tommy Fan and Austin Poore. 2018. Reproducing Results from: Constant Time Updates in Hierarchical Heavy-Hitters. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 6 pages.
DOI:

1 INTRODUCTION

In the paper *Constant Time Updates in Hierarchical Heavy-Hitters* [2], R. Ben Basat et al. present and analyze a fast randomized algorithm to compute hierarchical heavy-hitters in a data stream.

Heavy-hitters are items in a data stream that appear at an unusually high frequency, and the problem of identifying them is well-studied. There are numerous existing algorithms and data structures that compute or approximate the set of heavy hitters in a given stream, such as the count-min sketch algorithm[3].

Ben Basat's paper, which provides the results we've attempted to reproduce, deals with a related, but slightly different problem: identifying *hierarchical* heavy-hitters (HHH). This problem pertains to data streams where the items can be viewed as belonging to some form of hierarchy. For example, a hierarchy with which we are commonly concerned in computer networking is determined by IP address prefixes. In this case, we say that IP addresses with shared prefixes belong to the same hierarchy. A 1-byte hierarchy for IPv4 is structured as follows, from most general to most specific: $(*.*.*.*)$, $(s1.*.*.*)$, $(s1.s2.*.*)$, $(s1.s2.s3.*)$, $(s1.s2.s3.s4)$. From this perspective, hierarchical heavy hitters represent groups of flows sharing IP prefixes which are responsible for an unusually high proportion of the total traffic. Hierarchies can also extend to multiple dimensions. For example, we can define hierarchies over source and destination addresses in the two-dimensional case (e.g. $[(s1.s2.s3.*), (d1.d2.*.*)]$ represents a possible entry in a two-dimensional hierarchy).

1.1 Motivation

In a networking context, identifying HHH is useful for applications like anomaly or DDoS detection. While regular heavy-hitter detection is able to identify single IP addresses that account for a large proportion of Internet traffic, the algorithm isn't sophisticated enough for DDoS detection because in a DDoS attack, each individual machine only accounts for a small amount of traffic. Aggregating over shared IP prefixes, however, can help identify a common source (which might represent a network of attackers).

These applications operate on the network and it's therefore important that they run close to the line-rate. Previous work on deterministic algorithms to solve the HHH problem are not efficient enough: the authors note in [2] that "per packet complexity of [the update step of] existing HHH algorithms is proportional to the size of the hierarchy". Since the run-time scales with the size of the hierarchy (or its square, for two-dimensional), it can quickly grow unmanageably large (and will only get worse when IPv6 is widely adopted). This makes it impossible for the algorithm to keep up with the network, and it will quickly fall further and further behind. [2]'s key contribution is a fast randomized hierarchical heavy-hitters (RHHH) algorithm that performs worst-case constant-time updates at the expense of some accuracy. However, as their results show, the error rate decreases significantly as the size of the packet stream increases.

Existing deterministic HHH-detection algorithms like [4] tend to work by keeping an instance of some heavy hitters detection algorithm for each level of the hierarchy and updating each one with every new packet. Clearly this scales with the size of the hierarchy, since that's what determines the number of individual heavy hitters counters being utilized, and each one is updated at each step. RHHH works only slightly differently: when a packet arrives, exactly one of the heavy hitters instances is randomly selected and updated, for a worst-case constant-time update. As a result of introducing randomness into the selection of the individual heavy-hitters instances to be updated, the authors propose a probabilistic relaxation of the canonical hierarchical heavy hitters problem. In theory, solving this problem with their proposed algorithm leads to much more efficient packet processing without too much error on streams of sufficient length (the authors prove several convergence bounds in [2]).

In this project, we attempt to reproduce some of the paper's key results. In particular, we reproduce figures 2, 3, 4 and 5 from their paper using code from the authors' repository on Github. Figures 2, 3, and 4 are plots of various error metrics, while figure 5 compares RHHH's empirical speed to some other existing deterministic algorithms, where the authors saw a speedup of up to 62x. Additionally, we wrote and evaluated our own implementation of the one-dimensional algorithm based on the algorithm's pseudo-code provided in the paper, and we present these results in section 3.

1.2 Terminology and Definitions

We'll generally attempt to stick to the naming conventions and notation used in [2]. For readers' convenience, we've reproduced here a table of some of the commonly used variable names to which we will refer throughout the paper.

Symbol	Meaning
P	The set of hierarchical heavy hitters reported by RHHH.
N	The current number of packets seen.
f_p	The true frequency of prefix p.
$\hat{f_p}$	The estimated frequency of prefix p.
θ	The threshold parameter used to identify heavy hitters.
ϵ	The algorithm's error guarantee.
$C_{q\|P}$	Conditioned frequency of q with respect to P.
$G(q\|P)$	Subset of P with the closest prefixes to q.

2 REPRODUCTION (AUTHORS' CODE)

Our initial reproduction attempts were fairly straightforward, as the authors provided us with a link to their GitHub repository (https://github.com/ranbenbasat/RHHH) containing implementations of the RHHH algorithm in one and two dimensions, in addition to some scripts to transform the packet traces (pcap files) into the proper input format. The paper was also generally quite specific about the various parameter settings under which experiments were run, which made it easy for us to replicate and run the algorithms under the same conditions. However, the authors' repository did not contain any code to parse the outputs and graph them. Our task was to obtain the data, set up a machine that could run the code, automate several runs of the authors' code, and collect and display the results. We were able to obtain and use the same packet traces as the original paper ([1]).

2.1 Experimental Setup and Methodology

We ran all of our experiments on a Google Cloud server with 2 CPUs and 12 GB of RAM, running Ubuntu 16.04. The original experiments tested packet traces up to one billion (2^{30}) packets long, but we limited our experiments to traces up to 32 million (2^{25}) packets due to time constraints (and because the results in the original paper were clearly visible already in packet traces of that size). Like the original paper, we used $\epsilon = .001$ and $\theta = .01$ for our experiments. We used data collected from three CAIDA packet traces (Chicago2015, Chicago2016 and San Jose2014). For each packet trace, we ran each experiment five times (as the original authors did), and then computed 95% confidence intervals using a standard student-t test (which appear in our figures as error bars).

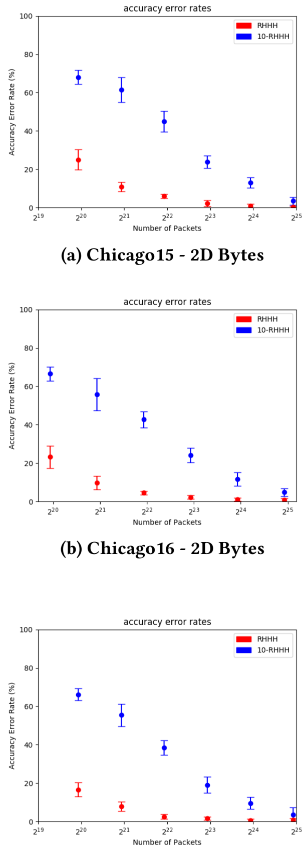
The steps in our experiment are as follows:

- Download packet traces from CAIDA and parse them into IP source, destination pairs
- Run the selected algorithm for varying stream lengths ranging from 1 million to 32 million pairs
- Repeat for a total of 5 trials for each algorithm
- Run a checker program that compares our reported HHH with the actual ones (from a deterministic algorithm) and reports various error metrics
- Run a python script to compute 95% confidence intervals and plot the error rates for each trace

2.2 Reproducing Figure 2

Figure 2 of the paper plots the accuracy error rate against the stream length (number of packets). It compares the regular RHHH algorithm with 10-RHHH, a modified algorithm that samples and updates only 10% of the stream's elements (ignoring the other 90%). As expected, 10-RHHH has higher errors, but the plot shows that it quickly decreases as the stream size grows.

Just like the paper, we say that for a prefix p, there's an accuracy error if $|f_p - \hat{f_p}| > 0.001N$. In other words, if the estimated frequency of a prefix deviates from its actual frequency by more than $0.001N$ (where N is the total number of packets seen so far), we say that it is inaccurate. The error rate is calculated by the number of accuracy errors divided by the number of HHH that the algorithm found. Our reproduction of Figure 2 is shown below.



(a) Chicago15 - 2D Bytes

(b) Chicago16 - 2D Bytes

(c) SanJose14 - 2D Bytes

Figure 2: Accuracy error rate (reproduction)

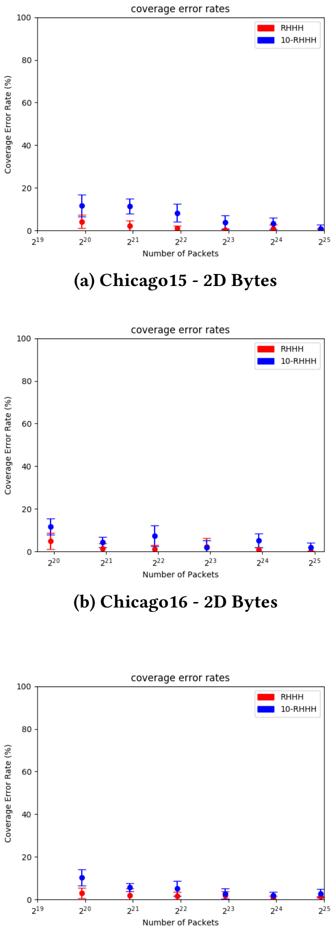
The accuracy error graphs take on the same shape as the original ones. Although the exact numbers may vary a bit, our results match what we expect from the paper. The accuracy error rate for 10-RHHH is drastically higher than that of RHHH, but for both algorithms, the accuracy error rate sharply decreases as we increase the stream length. At 32 million packets, both algorithms' error rates are close to 1%. Seeing as internet-scale traces (all of the flows passing through a router, for instance) are likely to eclipse this amount of traffic, either one of these algorithms seems fairly reasonable. 10-RHHH is faster by nature of ignoring the vast majority of the packets it sees, so from an accuracy perspective it seems like a reasonable choice (especially as streams get longer).

One notable difference is the exact value of the accuracy error rate for 10-RHHH (Chicago2015) at 2 million packets. In our graph, this value is at around 63% while in the original paper, it's at around 55%. However, given the variability of this randomized algorithm, this is expected.

2.3 Reproducing Figure 3

Similar to Figure 2, Figure 3 of the paper plots another error metric, the coverage error rate, against the stream length. Intuitively, the coverage error reports false negatives - or real HHH that are missing from the algorithm's output. It is formally defined by the paper as $q \notin P$ such that $C_{q|P} \geq N\theta$.

For our reproduction efforts, we ran RHHH and 10-RHHH with increasing stream lengths from 1 million to 32 million (similar to the setup in the previous section) on the three traces. The results are shown below.



(a) Chicago15 - 2D Bytes

(b) Chicago16 - 2D Bytes

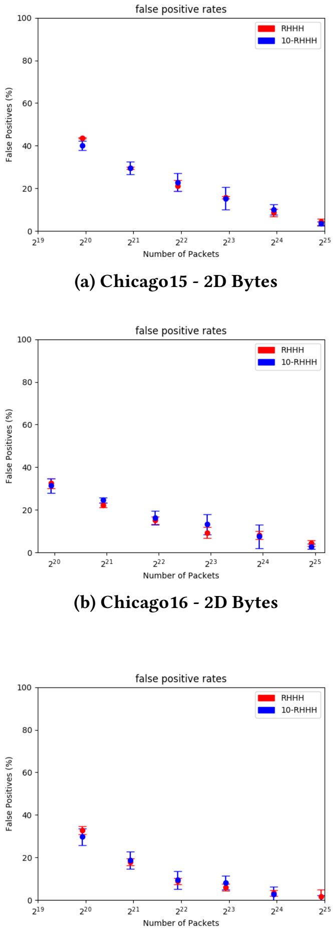
(c) SanJose14 - 2D Bytes

Figure 3: Coverage error rate (reproduction)

Just like figure 3 in the original paper, our graphs demonstrate a slightly downward trend in the coverage error rate as the stream length increases. It's interesting to note that in both our reproduced figures and the original figures, the coverage error rate is very low for both RHHH and 10-RHHH. Even at 1 million packets, the coverage error rate is less than 10%, which suggests that these algorithms rarely report false negatives. When compared to the false-positive rate shown in the next section which are much higher, we can deduce that these randomized algorithms err on the side of reporting a heavy-hitter (perhaps because it over-estimates counts).

2.4 Reproducing Figure 4

Figure 4 of the paper plots a third error metric, the false positive rate, against the stream length. These are the items mistakenly classified as HHH by the algorithm, but which do not actually meet the required threshold.



(a) Chicago15 - 2D Bytes

(b) Chicago16 - 2D Bytes

(c) SanJose14 - 2D Bytes

Figure 4: False-positive rate (reproduction)

Unlike figure 2 where 10-RHHH exhibits much higher accuracy error rates than RHHH, this figure shows that false-positive rates are actually quite similar between the two algorithms. Even though

10-RHHH ignores 90% of the packets, it doesn't seem to have too much of an impact on the number of false-positives reported.

Again, our figures show the same downward-trend as the original graphs. At 1 million packets, the false-positive rates hover around 40% for RHHH and 10-RHHH, but decreases to around 2% at 32 million packets.

2.5 Reproducing Figure 5

Figure 5 is somewhat different from the previous figures we've reproduced so far, none of which ought to be machine-dependent. This is an empirical analysis of runtime performance, though, so the machine on which the code is being run could end up being important. In particular, this figure in the original paper plots the speed of the update steps as a function of ϵ (the probabilistic accuracy guarantee) for RHHH, 10-RHHH, and some other existing baseline algorithms like Partial Ancestry and Full Ancestry. To vary ϵ, we changed the number of counters used by the Space-Saving algorithm, which estimates counts at each hierarchy.

Since the primary goal of these HHH algorithms is that they're able to keep up with the network, it makes sense that the performance of the update step is the one that we need to measure. Once the data has been collected, it can be analyzed offline and output HHHs. The update step, on the other hand, needs to be fast because it occurs live on the network.

2.5.1 Experimental setup. Like the original paper, we use [5]'s implementation of the baseline algorithms for comparison. The chart below presents the results from running the two-dimensional byte-granularity algorithms on the Chicago 2016 trace. We fixed the number of packets to be 3 million, although the original authors used 250 million in their experiment. We ran the four algorithms multiple times by varying the ϵ parameter in range of 2^{-14} to 2^{-1}.

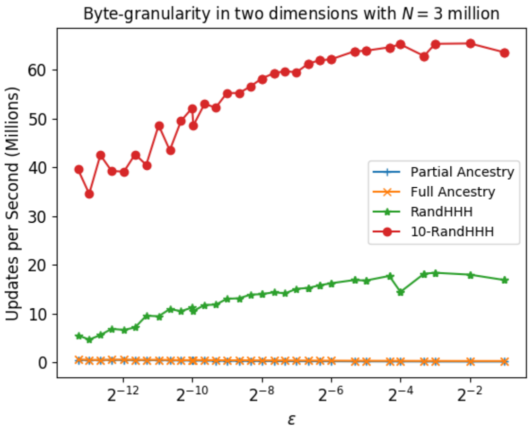


Figure 5: Update speed comparison

Overall, our reproduced figure 5 is quite similar to theirs. Even though we used way fewer packets in our stream, it didn't affect the number of updates per second. 10-RHHH was able to achieve 40 million to 60 million updates a second, just like in the original experiment. By contrast, partial and full-ancestry were extremely slow in comparison, as expected. One notable observation is that 10-RHHH was able to only achieve 4x the number of updates as RHHH even though it samples only 10% of the packets. Perhaps the required overhead accounts for this discrepancy.

3 REPRODUCTION (OUR IMPLEMENTATION)

While it was really nice to have the authors' original implementation of RHHH available to us, we were also interested in finding out whether we could reproduce their results ourselves with our own implementation. We didn't look at the authors' original implementation before or during the process of writing our own, except to figure out how to use the space-saving library they had incorporated as a subroutine (for the individual levels of the hierarchy). Instead, we limited ourselves to the paper itself and the pseudocode it contains, and we wrote our own version of RHHH for one-dimensional hierarchical IP address data in C.

We kept much of the same data infrastructure since we were only interested in reproducing the core RHHH algorithm. This meant that we had to constrain our input and output formats to the same ones that the authors had used, which allowed us to use the same pcap file parser. Our implementation would output a list of heavy-hitters, including their IP address, mask, hierarchy and count estimates. The remaining work, for example counting false-positives and negatives, was done by a check.cpp file that the authors included.

3.1 Results

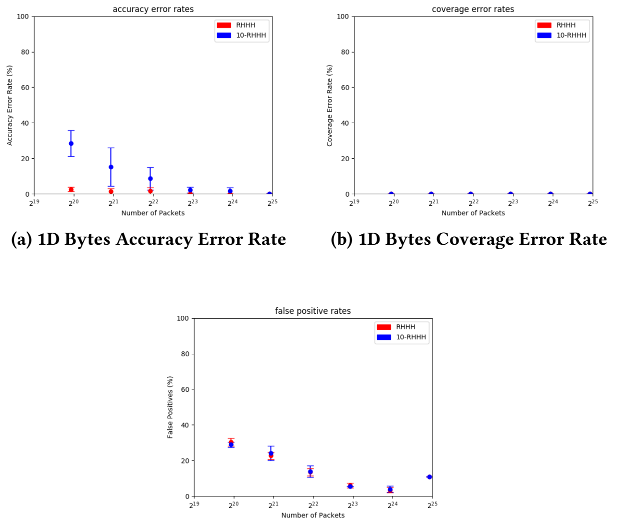
The short answer is that this portion of our reproduction was successful. Our implementation achieved comparable results to the code provided by the original authors. However, this is not to say that there weren't a few difficulties along the way. We ended up corresponding with R. Ben Basat several times along the way to resolve some questions about definitions in the paper, the adequacy of our machine's specifications, and other miscellaneous details, and his guidance was extremely helpful! In particular, we were confused about a seemingly contradictory definition and example related to the use of the generalization relation, but Ben Basat helped us figure out the correct meaning as it was used in the algorithm's pseudocode. Another point of confusion was the $2Z_{1-\delta}$ constant that's added to the count estimate in the RHHH algorithm. For simplicity, Ben Basat told us to use a constant instead of computing this inverse cumulative normal distribution function.

We generated our data by running our one-dimensional implementation on the Chicago 2015 trace, and found that the results were quite good. Our algorithm didn't make any coverage errors, even on small data sets, and error and false positive rates generally converged the way we would've expected from the theory (and the authors' original results). We can't make a direct comparison here, since the authors only show two-dimensional plots in the original paper's versions of figure 2 and figure 3, but we are satisfied that we were able to reproduce the same general trends.

4 DISCUSSION

4.1 Reproducibility

Overall, we consider this to be a successful reproduction attempt. Not only were we able to obtain similar numbers to the original



(a) 1D Bytes Accuracy Error Rate

(b) 1D Bytes Coverage Error Rate

(c) 1D Bytes False Positive Rate

Figure 6: Error rates (our implementation) - Chicago2015

paper when running the code on our machine, but more importantly, we were able to write our own implementation of the algorithm based on the information available in the paper itself and achieve good results. All of the figures we generated ended up looking very similar to their counterparts in [2]. The algorithm is randomized, so of course the numbers are not exactly the same, but the trends are obviously similar.

4.2 Limitations

There were a few limitations to our reproduction efforts. The first and most obvious one is that many of our experiments were run with the authors’ own code, on the same (or very similar, albeit smaller) data sets. Thus, it doesn’t seem particularly surprising that our results ended up being very similar despite the difference in machine and randomness in the algorithm. We only implemented the 1-dimensional algorithm ourselves, so for all of the other experiments we rely quite heavily on the code from the authors. Even for the trials of our own implementation, we use the authors’ script to collect the results, which is another dependency.

4.3 Future Work

Given more time to work on the project in the future, we have a few ideas for some potentially interesting additional reproduction efforts. The most obvious next step would be to write our own implementation of the two-dimensional algorithm and measure its performance. However, given the success of our reproduction of the one-dimensional algorithm (and the fact that the two algorithms are not too different – we mostly just need to enumerate more possible entries in the hierarchy matrix) and the results obtained from running the authors’ code for the two-dimensional algorithm, we are fairly confident that the results would also be reproducible.

Our reproduction efforts also used the same packet trace data sets as the original paper. While this is valuable and likely helped ensure that our results were closer to those presented in the paper, it would also be interesting to test the algorithms on new packet traces and confirm that the same performance can be obtained. We’d expect the algorithm to generalize to different traces, but if for some reason it doesn’t, it might be unsuitable for use in real systems. In any case, running on different data would be a good sanity check, and perhaps a more robust confirmation of the reproducibility of the paper’s results. A potential idea that we had was to run tcpdump on a Stanford router and collect anonymous packet traces on the school network.

We were also curious about the trade-off between accuracy and the complexity of the update step. Recall that in RHHH’s update step, we randomly selected one hierarchy and increment its count. What if we randomly sampled two or more hierarchies? Surely that would make the update step slower, but what will the effects on the accuracy error, coverage error and false-positive rate be? Would they be significantly higher? We’d explore this trade-off if we had more time.

5 CONCLUSION

In this paper, we set out to present the findings from our efforts to reproduce several of the key results from [2]. Our work falls into two broad sections: in the first, we use the original authors’ code to run their experiments on our own machine and generate fresh data, which we then use to reproduce four of the figures from the paper detailing error metrics and performance. Then, in the second, we present the empirical results from our own implementation of the one-dimensional algorithm, which were comparable to the results from the authors’ implementation. As a result, we feel very confident in endorsing the paper as being reproducible.

As part of our reproduction efforts, we’ve also written some scripts to automate the processes of running all of the experiments and generating the resulting plots which, when packaged with the authors’ original code, will make it even easier for future reproduction efforts to confirm the paper’s findings. All of the code can be found here: https://github.com/lechengfan/RHHH.

6 ACKNOWLEDGEMENTS

We would like to thank the authors of this paper for all their help in making their paper reproducible. In particular, Ran Ben Basat promptly answered all the questions that we had and clarified points of confusion for us. His emails to us were very detailed and helped us reach a greater level of understanding of the paper.

We also want to thank the professors of CS 244, Keith Winstein and Nick McKeown, as well as the TA’s, Saachi Jain and Emre Orbay, for reading our project proposal and guiding us along the way.

REFERENCES

[1] The CAIDA UCSD Anonymized Internet Traces - May 2018. (????). http://www.caida.org/data/passive/passive_dataset.xml

[2] Ran Ben Basat, Gil Einziger, Roy Friedman, Marcelo C. Luizelli, and Erez Waisbard. 2017. Constant Time Updates in Hierarchical Heavy Hitters. *SIGCOMM* (2017), 127–140.

[3] Graham Cormode and S. Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58 – 75. DOI: http://dx.doi.org/https://doi.org/10.1016/j.jalgor.2003.12.001

[4] M. Mitzenmacher, T. Steinke, and J. Thaler. 2012. Hierarchical Heavy Hitters with the Space Saving Algorithm. *In Proceedings of the Meeting on Algorithm Engineering & Expermiments, ALENEX* (2012), 160–174.

[5] Michael Mitzenmacher, Thomas Steinke, and Justin Thaler. 2012. Hierarchical Heavy Hitters with the Space Saving Algorithm. In *Proceedings of the*

Meeting on Algorithm Engineering and Experiments (ALENEX '12). SIAM, 3600 Market Street, 6th Floor, Philadelphia, PA 19104-2688 USA, 160–174. http://siam.omnibooksonline.com/2012ALENEX/data/papers/027.pdf