SANDIA REPORT

SAND2017-1475
Unlimited Release
Printed February 2017

Final Report: Sampling-Based Algorithms for Estimating Structure in Big Data

Kevin Matulef

Sandia National Laboratories

SAND2017-1475
Unlimited Release
Printed February 2017

Final Report: Sampling-Based Algorithms for Estimating Structure in Big Data

Kevin Matulef
Sandia National Laboratories
P.O. Box 5800
Livermore, CA 94550
kmmatul@sandia.gov

Abstract

The purpose of this project was to develop sampling-based algorithms to discover hidden structure in massive data sets. Inferring structure in large data sets is an increasingly common task in many critical national security applications. These data sets come from myriad sources, such as network traffic, sensor data, and data generated by large-scale simulations. They are often so large that traditional data mining techniques are time consuming or even infeasible. To address this problem, we focus on a class of algorithms that do not compute an exact answer, but instead use sampling to compute an approximate answer using fewer resources.

The particular class of algorithms that we focus on are *streaming algorithms*, so called because they are designed to handle high-throughput streams of data. Streaming algorithms have only a small amount of working storage – much less than the size of the full data stream – so they must necessarily use sampling to approximate the correct answer. We present two results:

- A streaming algorithm called `HyperHeadTail`, that estimates the degree distribution of a graph (i.e., the distribution of the number of connections for each node in a network). The degree distribution is a fundamental graph property, but prior work on estimating the degree distribution in a streaming setting was impractical for many real-world application. We improve upon prior work by developing an algorithm that can handle streams with repeated edges, and graph structures that evolve over time.

- An algorithm for the task of maintaining a weighted subsample of items in a stream, when the items must be sampled according to their weight, and the weights are dynamically changing. To our knowledge, this is the first such algorithm designed for dynamically evolving weights. We expect it may be useful as a building block for other streaming algorithms on dynamic data sets.

Acknowledgments

The HyperHeadTail algorithm in Chapter 1 was developed jointly with Andrew Stolman, who was instrumental in designing both the algorithm and the implementation. A large part of Chapter 1 was taken verbatim from a paper that he co-wrote.

The PI would like to thank Ali Pinar and Tammy Kolda for mentorship and helpful suggestions. Many others also contributed their time, expertise, and encouragement, including Jon Berry, Cindy Philips, Grey Ballard, Seshadhri Comandur, Madhav Jha, and Erdem Sariyuce.

Contents

Preface

Note that the recipient of this LDRD, the author of this report, left Sandia to pursue other opportunities before the completion of the project. Thus, some of the research that follow represents unfinished work.

Summary of Accomplishments

This document constitutes the final report for LDRD #186366: Sampling-Based Algorithms for Estimating Structure in Big Data. The project, though cut short due to the PI's departure from Sandia, led to two primary research results:

- The first is a streaming algorithm, `HyperHeadTail`, used for estimating the degree distribution of a dynamic, evolving graph. This work was done jointly with Andrew Stolman, a Ph.D. student at the University of California, Santa Cruz, who the PI supervised over the summer of 2016 and continued to collaborate with afterwards.
  - The work was written up in a paper entitled "HyperHeadTail: a Streaming Algorithm for Estimating the Degree Distribution of Dynamic Multigraphs." It will be submitted soon to a peer-reviewed conference.
  - An implementation of the algorithm was written and released under an open-source license for the academic community. It can be found at **https://github.com/astolman/HyperHeadTail**
  - The PI gave an invited talk on the result at the Workshop on Local Algorithms (WOLA), sponsored by MIT and Microsoft Research New England, in October 2016.

  A complete draft of the paper resulting from this work can be found in Chapter 1.

- The second is an algorithm, `Dynamic Weighted Reservoir`, for maintaining a weighted subsample of items in a stream, when the items must be sampled according to their weight, and the weights are dynamically changing. This solves a problem that, to the best of our knowledge, has not been solved before, and we expect it to be useful as a building block for other streaming algorithms.

  The basic version of this algorithm is written up in Chapter 2. This research may lead to publication, but is currently a work in progress.

Chapter 1

HyperHeadTail: a Streaming Algorithm for Estimating the Degree Distribution of Dynamic Multigraphs

1.1 Introduction

Many massive datasets are naturally interpreted as a stream of edges in a graph. Packets sent between IP addresses, messages sent between users of a social network, vehicles dispatched between locations — all can be interpreted as edges between nodes in a network. Often the number of nodes and edges is so large that storing and computing on the whole stream is time and space-intensive and, in some cases, infeasible. To make sense of such networks, we need algorithmic tools that can analyze them in real-time, using only a fraction of the space required to store the whole stream. The goal is to to get an approximate, holistic view of the network rather than requiring an exact answer.

One of the most fundamental properties of a network is its degree histogram. That is, for every positive integer k, how many nodes in the network have degree k? When this is normalized by the number of vertices in the graph, it is referred to as the degree distribution. The degree distributions of different graphs have been widely studied [5, 12]. Famously, social networks often empirically exhibit a power law (or more generally "heavy-tailed") degree distribution [2]. Yet, despite this importance, little research has been done on estimating the degree distribution of real-world graphs in a streaming context.

1.1.1 Problem Statement

We assume our input is a stream consisting of entries of the form $\langle u, v, t \rangle$ where u, v are vertices from some vertex set V (the elements of V might be unknown before they arrive in the stream), and t is an integral timestamp. The entries arrive in increasing timestamp order. Call such a stream a *dynamic edge stream*. For a given time period t_i to t_j (where $t_i < t_j$) we define $G_{(t_i,t_j)}$ to be the graph formed by the unique edges that arrived between time t_i and t_j, and the vertices incident upon those edges.

For vertex $v \in V$, let $d(v)$ denotes its degree. Let $n(d)$ be the number of vertices of degree d, and $N(d)$ be the number of vertices of degree at least d. The *degree histogram* is the sequence $\{n(d)\}$ for $d = 1, 2, 3, \ldots$ and the *complementary cumulative degree histogram* (CCDH) is the sequence $\{N(d)\}$. Technically, the *degree distribution* is the degree histogram, normalized by the number of nodes in V. However, consistent with the literature, we focus on approximating the CCDH instead of the degree distribution[5]. This is because the degree distribution itself is quite noisy in real data, and sensitive to minor changes. Also, informally, we refer to elements of $\{N(d)\}$ for small values of d as the *head* of the CCDH, and for large values of d as the *tail* of the CCDH. Our goal is to develop an algorithm that, with minimal space usage, can stop at any time t_j in the stream and estimate the CCDH of $G_{(t_j - w, t_j)}$ for any positive integer w.

1.1.2 Previous Work

The only previous work on estimating the degree distribution in a streaming context comes from Simpson, Seshadhri, and McGregor [23]. They develop an algorithm called `HeadTail`, and demonstrate its efficacy on several real-world graph datasets. Crucially, they also define a new distance-like metric called the "Relative Hausdorff distance" for evaluating the quality of `HeadTail`'s approximations.

The `HeadTail` algorithm suffers from two flaws that prevent it from being deployed in real-world systems. First, it is designed for graphs, not multigraphs. In other words, it assumes it will only see each unique edge once in the stream. This means that it cannot distinguish between the case when a node A has many unique neighbors, or just one neighbor that it connects to many times. For communication networks, such as network traffic data, multiple edges between the same pair of nodes A and B is very common, but it important that we are able to distinguish this from the case when A connects to many different nodes and plays a more critical role in the network.

Second, the `HeadTail` algorithm is designed to handle a static graph, not a dynamic one. In the model of [23], the algorithm sees the entire edge stream once, and then must output an approximation. But for many applications, we would like to maintain an approximation over a variable length time-window, so that we can compare the approximation from, say, the last day's worth of data to the last week's or month's, etc. We would also like an approximation that remains up-to-date over the most recent time window, without recomputing from scratch each time. Such an approximation can help us detect anomalies or large changes in the degree distribution in real-time.

1.1.3 Contributions

The Algorithm `HyperHeadTail`: In this work, we develop an extension of `HeadTail` called `HyperHeadTail` that is much better suited to real world data sets since it allows for graph streams with duplicate edges, and also allows for querying over variable-length, sliding time windows. The algorithm is described in detail in the next section.

The Incorporation of `HyperLogLog`: In order to achieve accurate estimates on streams with repeated edges, we incorporate the state-of-the-art algorithm `HyperLogLog` [13] in a novel way. The original algorithm was designed to tackle the problem of counting the number of unique elements in a single stream, but we use it to count the number of unique edges incident upon each vertex that we track. We also adjust the number of tracked node, and hence the number of `HyperLogLog` instances deployed, using the idea of *legitimacy lists*.

Legitimacy Lists: Naively employing the `HyperLogLog` algorithm at tracked nodes leads to a problem when we try to build an algorithm that works for variable-length time windows. Either the algorithm achieves a high degree of accuracy at small window sizes and has a high space usage, or it uses significantly less space at the cost of accuracy in smaller windows. In order to circumvent this, `HyperHeadTail` incorporates a new data structure which we call *Legitimacy Lists* in order to use only a small amount of additional space while maintaining accuracy at all time scales.

Fixing Relative Hausdorff Distance: The original paper of Simpson, et. al. presented a new distance metric for degree histograms they called *Relative Hausdorff Distance.* We believe it is a useful distance metric, but it suffers from some shortcomings when applied to discrete objects like degree histograms that we address in this paper.

Empirical Results: We implement `HyperHeadTail` and demonstrate its efficacy on both real-world and synthetic datasets. In our empirical tests, `HyperHeadTail` nearly matches the performance of `HeadTail` on edge streams without duplicate edges, showing that the additional functionality can be achieved at little cost. On edge streams with duplicate edges, `HyperHeadTail` is vastly superior to naive methods, and can effectively track changes to the degree distribution over time. We discuss our experimental results in Section 1.3.

1.2 Algorithm Details

1.2.1 Review of the HeadTail Algorithm

Since our algorithm, `HyperHeadTail`, builds heavily on the `HeadTail` algorithm of [23], we review it here.

The motivating idea behind `HeadTail` is that two different sampling methods can be deployed simultaneously to capture accurate estimates of both the head and the tail of the degree distribution. A uniform random sample of vertices will accurately estimate the number of nodes of low, frequently occurring degrees, while a sample of vertices biased by degree will accurately find high degree vertices at the tail.

`HeadTail` works on a model where the edge stream consists of timestamp-less $(src, dest)$ pairs. The algorithm maintains two sets of pairs called S_h and S_t, also referred to as the *head set* and *tail set*, respectively. The head and tail set contain pairs of the form (v, c) where v is a vertex and c is a count of the encountered edges incident upon that vertex. These are implemented as hash tables with vertex labels as keys and a count for each vertex as the value.

The head set contains a uniform random sample of the nodes in the graph chosen with probability p_h and maintains an exact count of their degrees. The tail set maintains a sample of the vertices which is biased by degree. This is achieved by flipping a coin with probability p_t each time an edge is observed. So if a vertex has more edges, it has more chances to enter S_t. We estimate the number of edges that passed in the stream before via the loss factor, $\ell(r)$, where r is the number of edges we saw in the stream for the given vertex. The loss factor is:

$$\ell(p_t, r) = \frac{1 - p_t - (1-p_t)^{r+1} - rp_t(1-p_t)^r}{p_t(1-(1-p)^r)}$$

Theorem 1. *A random variable drawn from a geometric distribution with parameter p_t conditioned on the event that it is less than d has expectation $\ell(d)$. We refer the reader to [23] for the proof.*

The point at which `HeadTail` switches between the head estimate and tail estimate is the threshold degree, d_{thr} in `Estimate` is calculated based on a Chernoff bound of the head set. It is the point in the CCDH at which there are few enough nodes that we no longer have confidence in a uniform random sample of nodes to accurately capture it. It is important to note that the authors of [23] were unable to prove general guarantee on the accuracy of the tail estimate, but in practice it works very well.

1.2.2 Sliding Windows and Legitimacy Lists

In the streaming literature, a model that allows for querying an algorithm over various time windows is referred to as the *sliding window model* [8]. We seek to adapt `HeadTail` to such a model.

The simple counters of HT are insufficient in our sliding window model. For example, it is impossible to discern from a straight count of all the edges incident upon a vertex how many of those edges arrived in the past ten, twenty or one thousand time units. Furthermore, we do not wish to exclude streams which contain repeated edges, whether those parallel edges have the same timestamp or do not. It is clear that we require a more sophisticated mechanism.

Fortunately, we can leverage past work on estimating the number of distinct elements in a streams. We use the state-of-the-art cardinality estimator `HyperLogLog` developed by Flajolet et al [13] which has been modified to handle sliding window models [3]. We refer to these counters as `Sliding HyperLogLog` objects (SHLL). For the remainder of this paper, we will treat SHLL as a black box which gives accurate estimations over any time window.

One of the key departures from `HeadTail` is that `HyperHeadTail` allows p_h and p_t to vary as functions of time. For short time window queries, the graph we are interested is rather small and so rather large values of p_h and p_t are needed to ensure accurate estimates. However, we do not wish p_h to remain large for large windows since this would entail tracking a large fraction of the nodes in the graph. If we allow p_h and p_t to be decreasing functions of time, we can incorporate this idea into our algorithmic design.

Accomplishing a p_h which varies with time for the head set is quite simple. Suppose that we encounter an edge (u,v) in the stream and the hash value of u is h such that $h \leq p_h(0)$, but there exists some w such that $h \geq p_h(w)$. If we make a note of this hash value and of the time at which we observed this edge, say it is t, we can delete this vertex from S_h at time $t+w$. Of course, checking every node's legitimate status in S_h at every time unit is quite time consuming and so we only do so periodically in a manner we will describe in more detail further on.

For the tail set, it is not enough to track a single hash value, since the hash values of each incident edge are relevant to the operation of the algorithm. For each vertex, v, in S_t, and each window size w, we wish to answer the question *Would v have been entered into S_t if we ran the graph stream of the last w time units through `HeadTail`?* In order to answer this question, we maintain a *Legitimacy List* for each node in S_t. These are lists of pairs of the form (h,t), where h is the hash value of an edge we observed in the stream and t is the timestamp at which we observed it. Naively, we could keep track of the hash value and timestamp of each edge we see, and when it came time to estimate the degree of that vertex, we could look through the edges that we observed in the last w time units and check to see if there is among them a hash value which is less than $p_t(w)$. However, it is not necessary to record each edge we observe. Suppose we encounter an edge with hash value h_1 at time t_1, and later on, at time t_2, we encounter an edge with hash value $h_2 < h_1$. The first edge is no longer necessary to answer the question, would this node have been tracked by `HeadTail`? Since if h_1 is a small enough hash value to answer this question in the affirmative, so must h_2. So, when a new hash is added to a legitimacy list, we check the list and delete from it all entries which have a greater hash value. In this way, we maintain the invariant that the legitimacy lists are sorted in a strictly decreasing order of hash values and increasing order of timestamps.

The legitimacy lists reduce the problem of deciding whether a given node in S_t should be counted for a given evaluation to the problem of tracking the minimum of a stream over a sliding window. This is analyzed in detail in [8]. We present the result below and refer the reader to [8] for the proof.

Theorem 2. *The space required for each legitimacy list is $O(\log d(v))$ where $d(v)$ is the degree of the vertex v.*

1.2.3 A Note on Notation

Algorithm 1 contains the pseudo code for our algorithm. Just as with HT, we maintain a head set and tail set, still referred to as S_h and S_t respectively. If v is a vertex in either set, $S_t(v).SHLL$ and $S_h(v).SHLL$ refer to the SHLL counter associated with that vertex in those respective sets. $S_t(v)$.`LegitimacyList` is the Legitimacy List (which we will explain in the next section) associated with that vertex, and $S_h(v)$.`LastUpdated` is the timestamp for the last time v was updated in S_h.

1.2.4 HyperHeadTail

In this section, we present a detailed overview of each of the methods of the `HyperHeadTail` algorithm. The pseudocode for these can be found in Algorithm 1.

Overview

`HyperHeadTail` is designed to give a running estimate of the CCDH of an edge stream. It is initialized with the parameters p_h and p_t which must be monotonically decreasing functions of time. These roughly correspond to the probabilities that a vertex will be tracked by the algorithm.

The algorithm operates by examining the edges in a stream one at a time in order of increasing timestamp. Each time an edge is encountered. The `Update` method of Algorithm 1 is called. This method alters the underlying data structures to incorporate the new information.

The persistent data structures maintained by `HyperHeadTail` are the two sets S_h and S_t. These are hash tables which map keys in V to the information required to compute the degree of each vertex over sliding windows. What information is required varies between the two sets. In both cases, we require a `Sliding HyperLogLog` counter, and while the head set requires only the timestamp of the time at which it was last updated, S_t requires that each node also have a Legitimacy List, and time of arrival associated with it.

The data structures are initialized so that `Sliding HyperLogLog` counters are empty. S_h begins with all the elements of V which have hash values less than $p_h(0)$, while S_t begins operation completely empty.

At any time, a user of `HyperHeadTail` may call the method `Estimate`(w) which returns an estimate of the CCDH over the last w time units from the time at which `Estimate` was called. This computes from scratch the entire CCDH vector.

`Update`

This is the method which updates the data structures every time an edge $\langle u, v, t\rangle$ is encountered. Since there are two data structures, S_h and S_t, `HyperHeadTail` must update each of them.

For the head set, `HyperHeadTail` considers each endpoint. For each endpoint, v, we check the hash value of each endpoint. If that hash value is less than $p_h(0)$, we say that vertex is in S_h. In this case, we increment $S_h(v).SHLL$, and change $S_h(v)$.`LastUpdated` to the current timestamp. If the vertex is not in S_h, we simply do nothing and move to the next step.

For the tail set, we also consider each endpoint in turn. For a given endpoint, v, we first check to see if v is in S_t. If it is, we increment v's `Sliding HyperLogLog` counter, and add (h,t) to v's Legitimacy List, where h is the hash value of the edge (v,u), and t is the current timestamp.

Then, in order to maintain the invariant that v's legitimacy list is sorted in decreasing order of hash values, we search through it and delete from it all hash values which are greater than h.

If an endpoint is not in S_t, it is given an opportunity to join S_t. We compute the hash value of (u,v), and if it less than $p_t(0)$, we enter it into S_t and note the time at which it was added.

`Clean_`S_h and `Clean_`S_t

Each clean method is called once every $\delta|S_h|$ and $\delta|S_t|$ calls to `Update`. Since the cleaning methods each require time that is roughly linear in the size of its respective set, there is a tradeoff between the time we spend cleaning the sets, and the amount of unnecessary information we store in memory. We found that keeping the calls down to a constant fraction of the size of the sets, we were able to achieve a good balance. Further customization and more sophisticated schemes are an avenue of possible future research.

Each of the clean methods search for nodes which may not be relevant to any call to `Estimate`. For S_h, the clean method simply checks the time at which each node was last updated and the hash value of that vertex. If the vertex was last updated so long ago that it's hash value is less than $p_h(t-t')$ for the current timestamp t and it's `LastUpdated` value t', all of it's information is cleared. For S_t, the process is quite similar, however, we need to check it's legitimacy list to see if there is a single entry there which justifies its presence, i.e. some element (h,t_1) such that $h \leq p_t(t-t_1)$. If no such entry exists, the vertex is purged from S_t.

`Estimate`

The `Estimate` method computes the CCDH over a given window size, w. First, we collect the counts from each `Sliding HyperLogLog` counter in S_h for each node in S_h with a hash value which is less than $p_h(w)$. We will refer to these counts as $c_h(v)$ for each vertex v. Observe that these counts are guaranteed to take into account every edge seen by `HyperHeadTail` over the last w time units since every edge incident upon a vertex in S_h causes `Update` to increment that edge's `Sliding HyperLogLog` counter unconditionally.

This is not the case for vertices in S_t, and the situation is further complicated by the sliding window. In `HeadTail`, the authors assumed a loss factor of $\ell(p_t,r)$ for every node in S_t since the stream was simply run from beginning to end and then `Estimate` was queried. Here, we may be querying at a time and for a window where some of the nodes in S_t entered into S_t within the last w time units, and others have been in S_t for longer. If the vertex has been in S_t for longer than w time units, then we have seen every edge incident upon it, and we call such a vertex an *old timer*, and we can set $c_t(v) = S_t(v).SHLL.$`Estimate`(w). Otherwise, call the vertex a *newcomer*, and we must add in the loss factor $\ell(p_t(0),r)$.

With the counts for each vertex decided, evaluation proceeds much as in `HeadTail`. We only expect to have captured a $p_h(w)$ fraction of the nodes for each degree in $c_h(v)$, and so we multiply the counts for the number of nodes by $1/p_h(w)$. For each degree r in $c_t(v)$, we expect to have captured a $(1-p_t(w))^r$ fraction of the newcomers, and so divide the number of newcomers with each degree by that. The counts are added together, d_{thr} is calculated just as in `HeadTail`, and the resulting vector is returned.

The `HyperHeadTail` Algorithm

```
function UPDATE($\langle u,v,t \rangle$)
    if $u \notin S_t$ AND $h(u,v) \leq p_t(0)$ then
        Add $u$ to $S_t$
        $S_t(u)$.TimeAdded $\leftarrow t$
    if $u \in S_t$ then
        Update $S_t(u)$'s SlidingHyperLogLog counter.
        Add $(h(u,v),t)$ to front of $S_t(u)$.LegitimacyList
        for $(h(u,v),t) \in S_t(u)$.LegitimacyList do
            Let $(h(u,v'),t')$ be the previous entry if it exists
            if $h(u,v) \geq h(u,v')$ then
                remove $(h(u,v),t)$ from LegitimacyList
    if $u \in S_h$ then
        Increment $S_h(u)$'s SlidingHyperLogLog counter
        Set $S_h(u).LastUpdate$ to $t$
    $c_h \leftarrow c_h + 1$
    $c_t \leftarrow c_t + 1$
    if $c_h > \delta|S_h|$ then
        Clean_$S_h$(t)
        $c_h \leftarrow 0$
    if $c_t > \delta|S_t|$ then
        Clean_$S_t$(t)
        $c_t \leftarrow 0$
function CLEAN_$S_h(t)$
    for $v \in S_h$ do
        if $h(v) \geq p_h(t - S_h(u)$.LastUpdated$)$ then
            Evict $v$ from $S_h$
function CLEAN_$S_t(t)$
    for $v \in S_t$ do
        Legit $\leftarrow$ False
        for $(h,t')$ in $S_t(v)$.LegitimacyList do
            if $h < p_t(t-t')$ then
                Legit $\leftarrow$ True
            if Legit is False then
                Evict $v$ from $S_t$
function ESTIMATE$(t,w)$
    for $v \in S_h$ do
        if hash$(v) \leq p_h(w)$ and $S_h(v)$.LastUpdated $\geq t - w$ then
```

```
            $c_h(v) \leftarrow$ the value returned by evaluating $S_h(v).SHLL$ for window $w$
    for $v \in S_t$ do
        if $S_t(v)$.LegitimacyList contains an entry $(h,t')$ such that $h \leq p_t(w)$ then
            if $t - S_t(v)$.TimeAdded $> w$ then
                $c_t(v) \leftarrow S_t(v)$.SHLL.Estimate$(t,w)$
                $v$ is an old timer
            else
                $r \leftarrow S_t(v)$.SHLL.Estimate$(t,w)$
                $c_t(v) \leftarrow r + \ell(p_t(0),r)$
                $v$ is a newcomer
    Set $c_h(v)$ to the count for each SHLL counter of $v \in S_h$
    $g_t(r) \leftarrow |\{v : c_t(v) = r \wedge v \text{ is an old timer}\}| + 1/(1-(1-p_t(w))^r)|\{v : c_t(v) = r \wedge v \text{ is a newcomer}\}|$
    $g_h(r) \leftarrow 1/p_h(w)|\{v : g_h(v) = r\}|$
    $d_{thr} \leftarrow \max_d \sum_{r \geq d} g_h(r) \geq clog(n)/(\varepsilon^2 p_h(w))$
    for each degree, $d$ do
        if $d \leq d_{thr}$ then
            $\hat{N}(d) \leftarrow \sum_{r \geq d} g_h(r)$
        else
            $\hat{N}(d) \leftarrow \sum_{r \geq d} g_t(r)$
    return $\hat{N}(r)$
```

1.2.5 Time and Space Complexity

The space maintained by HHT is precisely that used by S_h and S_t. The number of nodes in these sets depends heavily on the functions p_h and p_t as well as the nature of the stream. If the stream is such that the graph is rapidly changing in terms of which nodes appear in the stream, there may be very little overlap between the nodes being kept for small windows and those kept for larger windows. If, on the other hand, the stream is more static in nature, i.e. the nodes which are active on a small scale are the same as the nodes active on a large scale, then there may be a high level of overlap. We state our bounds in terms of the space used per node.

Theorem 3 (Space & Time Complexity of HHT). *For each node v tracked by* `HyperHeadTail`*, the space used is $O(\log\log d(v) \log d(v))$, where $d(v)$ is the degree of v. For each call to* `Update`*, the expected running time is $O(\log d(v))$.*

Proof. The space required for each tracked node is the space required by each SHLL counter, plus the space for the legitimacy lists for nodes in S_t. Following the analysis in [3], SHLL requires size that is $O(\log\log n * \log n)$ where n is the true cardinality of the multiset. Here, the multiset we are estimating is the neighborhood of a vertex. So we can say that each SHLL counter uses space that is $O(\log\log d(v) \log d(v))$ where $d(v)$ is the degree of vertex v in the graph formed by all the edges since the start of the stream. The legitimacy list is an extension of the minimum sliding window problem posed in [8], and following their analysis, it requires $O(\log d(v))$ space. Thus, the overall space used by each node is $O(\log\log d(v)\log d(v))$.

The time complexity of the `Update` operation is simply the time needed to perform the `Update` routine of the SHLL objects, which is $O(\log d(v))$ [8]. The time complexity of updating the legitimacy lists is also $O(\log d(v))$ since we must traverse each of the $O(\log d(v))$ entries in it. The cleaning routines, as previously noted, can be amortized over every call to `Update` and so do not affect the asymptotic time complexity. $\square$

For a given window size w, the expected number of tracked nodes is $O(p_t(w)n_w + p_h(w)m_w)$ where n_w is the number of unique nodes in that time window, and m_w is the number of unique edges. In Section 1.3, we discuss the settings of p_h and p_t that we use in practice for different window sizes, and show empirically that the number of nodes tracked is quite small.

1.3 Experimental Results

1.3.1 Distance Measures for CCDH Estimates

One issue raised by [23] is that typical distance measures for comparing CCDH estimates do not capture our intuitive notion of distance. For instance, consider two graphs: the first a matching consisting of $n/2$ disconnected edges, and the second a star graph on n vertices (i.e. one central node of degree $n-1$ with an edge to each of the other vertices in the graph). In many respects, the degree distributions of the two graphs are quite similar - indeed, the only difference is the single node in the star graph that has degree $n-1$ instead of degree 1. However, our intuition is that the two graphs are structurally quite different. Since the tail of the degree distribution contains the nodes of highest degree in the graph, it has a large bearing on the structure of the graph. We would like to use a distance metric between degree distributions that is sensitive to such differences in the tails.

One of the main contributions of [23] was the introduction of a new distance measure, the Relative Hausdorff (RH) distance, defined as follows:

Definition 1 (Relative Hausdorff distance)**.** *Let F and G be non-trivial CCDHs. For $\varepsilon, \delta > 0$, we say that G* is (ε, δ)-close *to F if:*

$$\forall d, \exists d' \in [(1-\varepsilon)d, (1+\varepsilon)d] \; s.t. |F(d) - G(d')| \leq \delta F(d)$$

The Relative Hausdorff (RH) distance *between F and G, denoted $RH(F, G)$, is defined as the minimum ε such that G is $(\varepsilon, \varepsilon)$-close to F and F is $(\varepsilon, \varepsilon)$-close to G.*

Note that the RH distance can be greater than 1. Intuitively, the RH distance captures the properties we would like in an approximation. In particular, if F is the true CCDH of some graph, and G is an approximation within a small RH distance, then for small d (where $F(d)$ is large), G

must produce a decent approximation to the number of nodes of degree d, while for large d (where $F(d)$ is small), G must find the number of nodes of degree at least d almost exactly, but it can be slightly off in the degree estimate (d' instead of d).

In [23], the authors argue that the RH distance is more appropriate than other oft-used distance measures. For instance, the Kolmogorov-Simonoff D-Statistic, defined as $\max_d |F(d) - G(d)$—, is often used for comparing degree distributions. However it is insufficient for capturing differences in the tails of distributions. In the earlier example of the matching versus the star graph, the KS distance tends to zero as the number of vertices is increased, but the RH distance remains quite large. Conversely, a k-clique versus a $(k-1)$-clique would have large KS distance, but the RH distance is very small (another popular distance metric, Earth Mover distance, would similarly assign a large distance between the k-clique and $(k-1)$-clique).

Fixing a flaw in RH distance

The RH distance as defined in [23] suffers from a flaw, in that it can assign large distance to graphs that are intuitively close. Consider two graphs, G and G', both on $n+2$ vertices. Let G be the bipartite graph in which two vertices have an edge to each of the other n vertices in G, and let G' be the same graph with one edge removed. These graphs are intuitively very similar. However, the $RH(ccdh(G'), ccdh(G))$ is 1, which is quite large. This is because G has two nodes of degree n, while G' has a one node of degree n and one node of degree $n-1$. Thus, $|G'(n) - G(d)|$ is at least 1 for every d.

This flaw is removed if we "smooth out" the CCDHs by requiring them to be continuous. We can convert the CCDH from a step function to one that connects the points at every integer value of d with line segments.

Definition 2 (Smoothed CCDH)**.** *Let $F : \mathbb{Z}^+ \to \mathbb{Z}^+$ be the CCDH of a graph G. Then the* Smoothed CCDH *of G is a the continuous function $F' : \mathbb{R}^+ \to \mathbb{R}^+$ formed by connecting $F(d)$ to $F(d+1)$ with line segments for all d.*

The RH-distance between smoothed CCDHs more accurately reflects our intuition. For instance, in the example above, the RH distance between the smoothed CCDHs corresponding to G and G' above asymptotically tends to zero as n tends to infinity.

In the rest of this work, when we refer to the RH distance between two CCDHs, we implicitly mean the RH distance between the smoothed versions.

1.3.2 Datasets

We performed our experiments on a mixture of synthetic and publicly available real world datasets.

The synthetic tests were carried out on streams generated using the "fast" Chung Lu method outlined in section 2.3 of [19]. The process is twofold. First we assign a random desired degree to each node in the graph by drawing from a zeta distribution with parameter 3 (i.e. for all $d \in \mathbb{N}$, $\Pr[degree(v) = d] \propto d^{-3}$). Then edges were generated one at a time with the endpoints chosen at random proportional to the degrees of the nodes. The Zeta3 dataset refers to a collection of 30 graph streams generated in this manner. The graphs in these streams have 10^6 nodes and approximately 1.37 million unique edges, and the streams on average contain only 40 repeated edges.

The real world datasets we used are the AS-733 dataset, a snapshot of the autonomous system network for 733 days [21] and the AUTH dataset [15], a network of system authentication on the Los Alamos National Labs network. The AS-733 dataset has approximately 7000 nodes and 45645 unique edges. Overall, the entire stream contains 11965533 edges, and so every edge is repeated an average of 262 times. The AUTH dataset has 33644 nodes and 312283 unique edges. The total stream contains over 7×10^8 entries, with each edge repeated an average of over 2000 times. Both of these real world datasets contain many repeated edges, whereas our synthetic datasets contain relatively few. Our datasets are summarized in Table 1.1.

Dataset	**—V—**	**Unique Edges**	**Total edges**
Zeta3	1M	1.37M	1.37M
AS-733	7K	45K	12M
AUTH	34K	312K	700M

Table 1.1. Summary of datasets used.

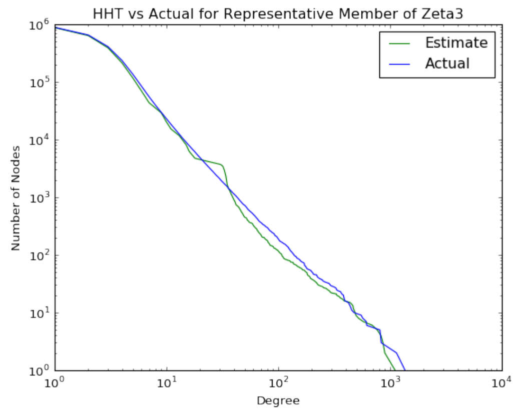
Figure 1.1 shows the quality of our estimates on a typical run of HHT for each dataset.

1.3.3 Comparison with HeadTail

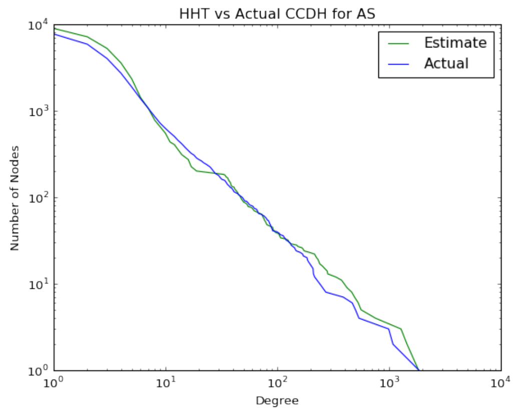
Our goal in developing HHT was to add extra functionality to HT with minimal impact on performance. HHT is designed to handle duplicate edges, but we expect some loss of accuracy compared to HT due to the use of the `Sliding HyperLogLog` (SHLL) for approximating the number of distinct neighbors of a node.

To measure the accuracy loss, we modify each dataset so that duplicate edges are removed. We then run HHT and HT with constant p_h and p_t set to .01. We modify HHT to query over a window that encompasses the whole stream, and to avoid calling the `Clean` methods so that there is as little algorithmic difference between it and HT as possible. The results are displayed in Table 1.2.

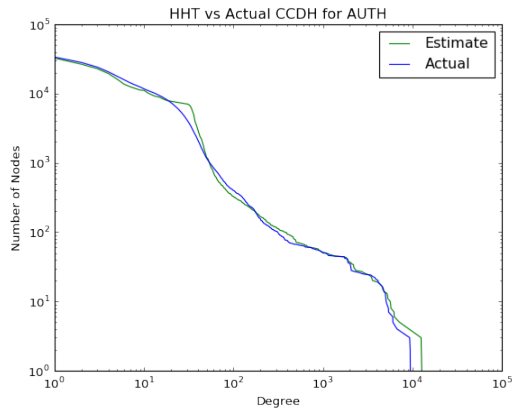
HeadTail is more accurate than HyperHeadTail on the real-world streams since it does not have the added noise of the SHLL counters, but the RH error is still quite small. The difference in accuracy becomes smaller for the synthetic streams since they have many more unique edges and so the added noise from the SHLL counters has a smaller impact. We note that the RH error of HHT can be lessened by allocating more storage space to SHLL to get more accurate distinct element counters.



(a) Estimated CCDH on whole stream of one of the graphs in Zeta3. The RH distance is 0.28



(b) Estimated CCDH on the AS-733 dataset. The RH distance is 0.30.



(c) Performance of `HyperHeadTail` on various datasets with p_h and p_t set to 0.01.

Figure 1.1.

Dataset	HT distance	HHT distance
Zeta3	.36	.33
AS-733	.21	.30
AUTH	.07	.26

Table 1.2. Comparison of HHT and HT approximation quality. In both cases, p_h and p_t each set to the constant 0.01. The numbers for the Zeta3 results are an average over all 30 streams in that dataset.

We also compared HHT with HT over variable-length time windows. To do this, we modified the streams in the Zeta3 dataset. We truncated each stream to include only those edges in each time window, removed all duplicate edges, and ran HT on those modified streams with p_h and p_t set to HHT's $p_h(w)$ and $p_t(w)$ respectively for each window size w. We summarize the results in Table 1.3.

The actual space usage of `HyperHeadTail` varies based on the implementation of SHLL and the setting of SHLL's internal parameters. Larger space usage will yield more accurate counts, and hence closer RH-distance. For the implementation we used in these tests, the maximum space used by any node in S_h or S_t was approximately 1.6 kb, and the average was on the order of several hundred bits per node. While this is a factor of 2-50x more than `HeadTail`, we stress that `HyperHeadTail` is capable of handling repeated edges without increasing the storage requirements, and in real-world streams, edges might be repeated hundreds or thousands of times.

1.3.4 Performance Tuning

The performance of `HyperHeadTail` involves a tradeoff between accuracy and space usage based on the settings of p_h and p_t. As with `HeadTail`, larger p_h and p_t values lead to higher accuracy, but also higher space usage. The authors of `HeadTail` were unable to prove formal guarantees on this relationship, and `HyperHeadTail` suffers from this same lack of formal analysis. However, Table 1.4 illustrates the tradeoff empirically, by plotting the RH accuracy achieved with different p_h and p_t values applied to the Zeta3 dataset.

Additionally, `HyperHeadTail` is meant to be applied to variable-length time windows, but different p_h and p_t values are more appropriate for different size windows. Smaller windows require larger p_h and p_t to achieve the same RH accuracy. To illustrate this point, Figure 1.2 contains a plot from a run of HHT with constant p_h and p_t queried over different windows. The same setting of the parameters does not provide the same accuracy for streams of different sizes.

Tuning p_h and p_t is thus a manual process. We performed many experiments on the synthetic data in order to give recommended p_h and p_t for various stream sizes. This data will be made available online at [25]. We found that with proper adjustment of the parameters, we were able to

Window Size	RH-dist of HT	RH-dist of HHT
10000	.52	.70
20000	.48	.35
30000	.38	.34
40000	.22	.35
50000	.23	.32
60000	.22	.33
70000	.22	.31
80000	.20	.30
90000	.21	.33
100000	.22	.31
200000	.22	.35
300000	.16	.28
400000	.13	.26
500000	.16	.26
600000	.17	.24
700000	.17	.24
800000	.17	.24
900000	.18	.24

Table 1.3. RH distance of HHT estimates compared to HT estimates over various time windows. Each had p_h and p_t set to a constant value of .01. The distances are an average over all 30 graphs in the Zeta3 dataset. The extra noise caused by the SHLL counters diminishes over the larger windows, and should by negligible in graph streams of sufficient size.

maintain an RH accuracy of 0.27 on all windows compromising 200,000 or more unique edges by tracking just 16% of nodes in the graph. If we require a minimum accurate window of 900,000 or more unique edges, we were able to maintain the same accuracy by tracking just 8% of nodes. Figure 1.3 contains a more detailed overview of HHT's performance on all of the streams in Zeta3 with properly configured p_h and p_t.

1.3.5 Anomaly Detection

One of the main motivations for developing a streaming algorithm for dynamically changing streams is to provide a tool for detecting time-based anomalies in the network. There has been a rich body of research on how degree distributions change over time and using the degree distribution along with other metrics to detect anomalies [20, 16].

As a demonstration of the efficacy of HHT for this task, we ran HHT on the AS-733 dataset

RH dist		p_h					
		.001	.02	.04	.06	.08	.10
p_t	.001	2.250	1.086	1.086	1.073	1.0	0.898
	.02	0.875	0.421	0.272	0.272	0.272	0.272
	.04	0.681	0.361	0.215	0.226	0.239	0.250
	.06	0.571	0.228	0.241	0.250	0.258	0.250
	.08	0.500	0.194	0.258	0.258	0.275	0.258
	.10	0.429	0.194	0.272	0.281	0.281	0.260

Table 1.4. Averages of HHT run on all 30 streams in Zeta3. The window size was set to 10^6 for each of these experiments. Each cell contains the average performance of HHT with p_h and p_t set to constant values corresponding to the row and column.

and estimated the degree distribution for each day at the end of that day and then compared that estimation to the day before. If the distance between the two is high, then we expect large changes in the graph. As Figure 1.4 shows, HHT tracks the true changes quite well. Large structural changes in the network are readily apparent in consecutive estimates with large RH distance.

1.4 Discussion and Future Work

`HyperHeadTail` is the first streaming algorithm for estimating the degree distribution that is capable of handling duplicate edges and variable-length sliding time windows. There has been previous work, such as [26], on estimating the degree distribution of a graph via sampling, and this work can be adapted to the streaming setting. However, the algorithm of [26] involves solving complex optimization problems. In our tests, it was simply infeasible to run [26] on the graph streams in our datasets.

The naive algorithm for tracking the degree distribution in a streaming setting is to keep a `Sliding HyperLogLog` counter for every node. However, `HyperHeadTail` shows that this is unnecessary, since an accurate estimate can be obtained using only around 10% of this space.

The main avenue for future research is to prove formal approximation guarantees on the estimates provided by both `HeadTail` and `HyperHeadTail`. Although the authors of [23] prove bounds on the accuracy of the head estimate of `HeadTail`, the accuracy of the tail estimate is still not quantified. The greatest weakness of both `HeadTail` and `HyperHeadTail` is that they require manual tuning of the p_h and p_t parameters. A more precise mathematical understanding of the space versus accuracy tradeoff might lead to a method for setting these parameters automatically.

Despite the current lack of mathematical guarantees, we find `HyperHeadTail` to be a robust and effective tool in practice, and we expect it to be helpful to network scientists studying the

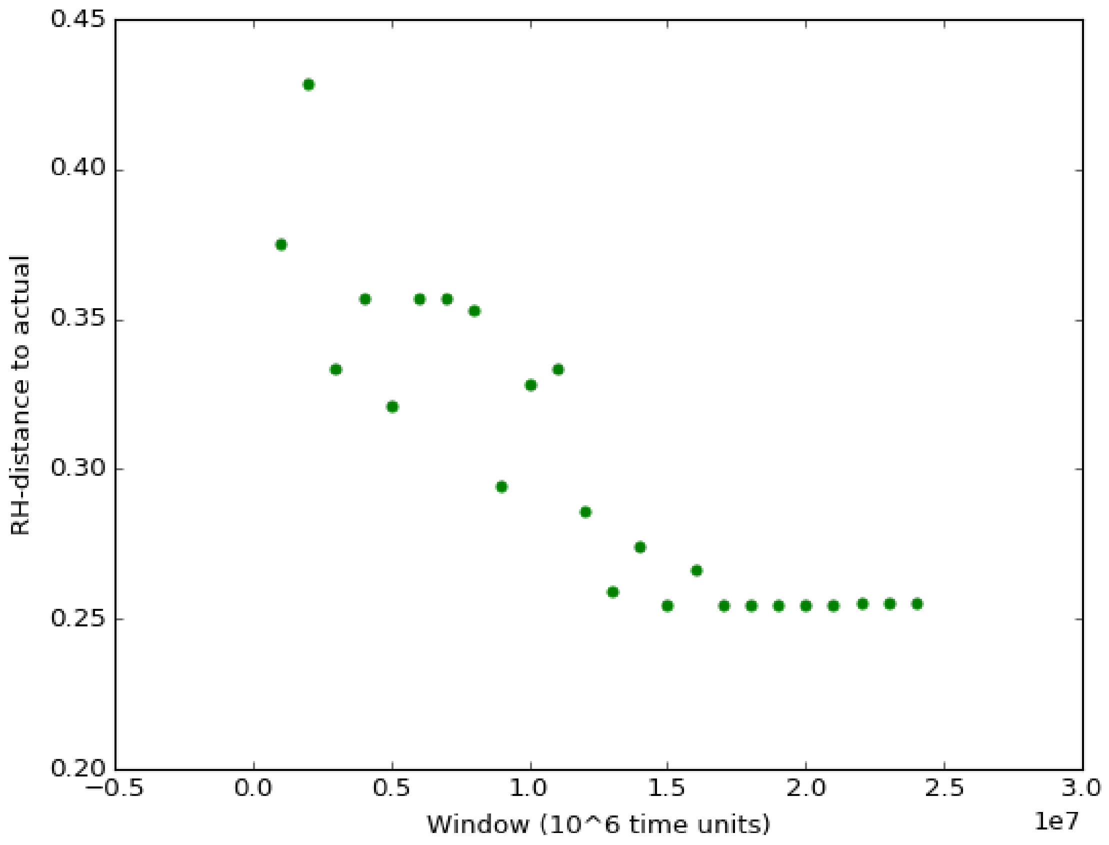


Figure 1.2. HHT performance on AUTH over increasing window size queries., with p_h and p_t set to a constant value of .01. The estimates are less accurate on smaller windows but converge to a consistent, relatively accurate estimate.

evolution of dynamic graphs.

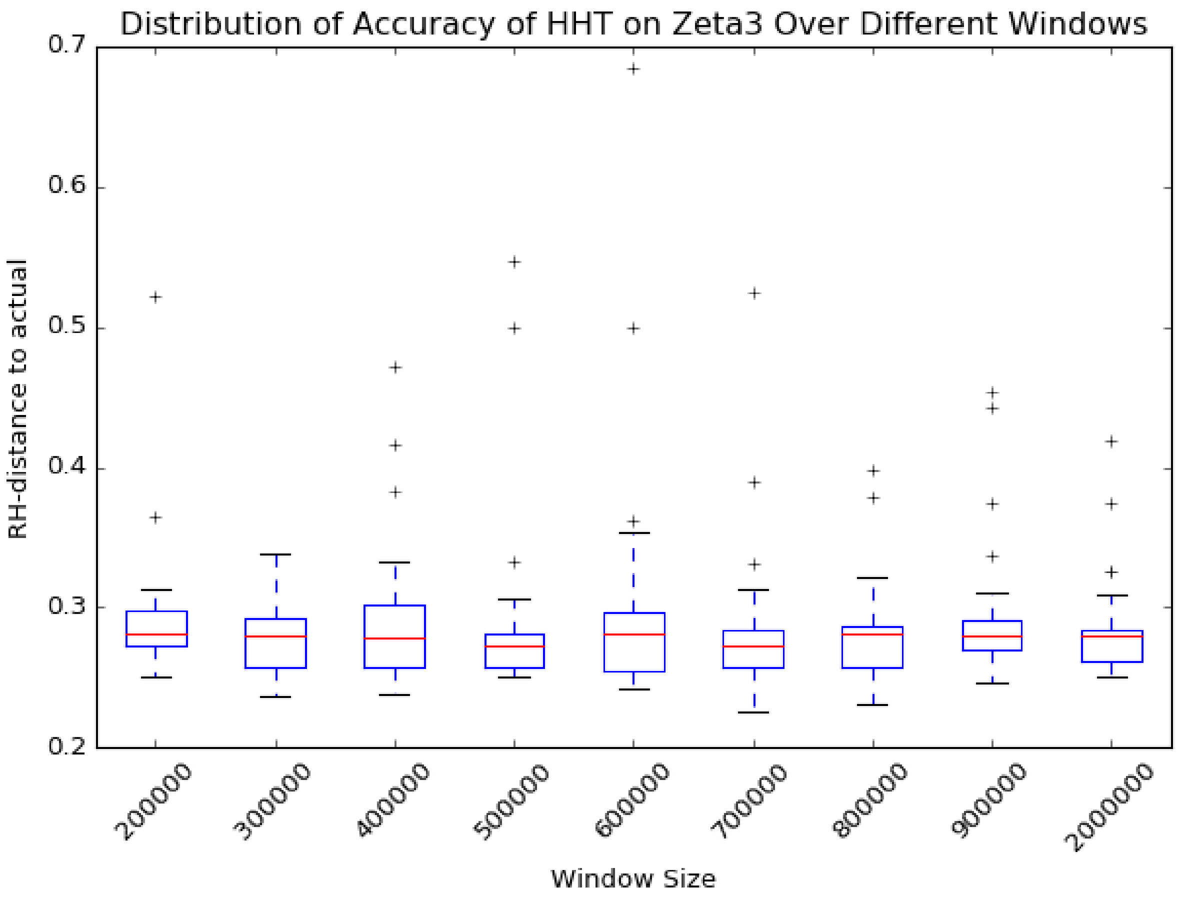
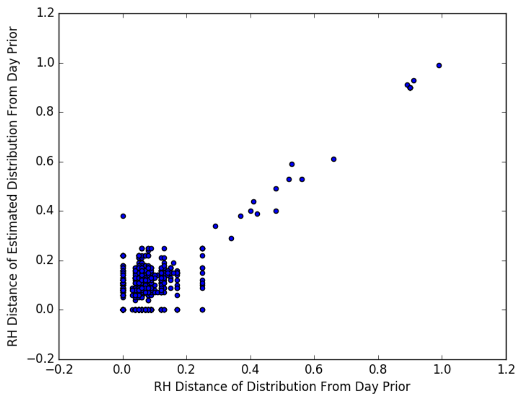
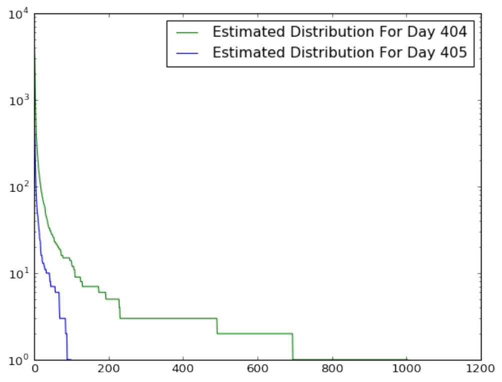


Figure 1.3. HHT performance on 30 different streams. We tuned p_h and p_t to offer constant levels of accuracy over the various window sizes.



(a) Each point represents the difference between the CCDH on two consecutive days from the AS-733 dataset. The horizontal axis represents the RH distance as estimated by HHT, and the vertical axis represents the true RH distance. Points in the upper right corner represent properly detected anomalies. False positive would appear in the top left corner, and false negatives in the bottom right.



(b) Anomaly detection on the AS-733 dataset.

Figure 1.4. CCDH estimates from two consecutive days in the AS-733 dataset. This illustrates one of the points in the top-right of figure (a), where the two consecutive CCDHs have high RH distance > 0.9

Chapter 2

Weighted Reservoir Sampling in the Presence of Dynamic Weight Updates

Maintaining a sample of m items from a data stream of size n, where $n >> m$, is a classic problem in the theory of small-space algorithms. We consider a variant of this problem, where the items in the stream are weighted, and the stream consists of updates to those weights. The goal, at any given point in the stream, is for each item to appear in the sample with probability proportional to its current weight. We describe an algorithm for maintaining such a sample, when the items are sampled with replacement. We also discuss a variant of the problem when the items are sampled without replacement.

2.1 Preliminaries

The size of the sample is m. The length of the stream is n. The stream consists of elements of the form (k, w_{old}, w_{new}) where k is a key for the item, and $w_{old}, w_{new} \in \mathbb{R}^+$ indicate the old and new weight of item k. In general, the weights may increase or decrease. That is, w_{new} may be larger or smaller than w_{old}.

When the weights increase, we allow updates of the form (k, Δ) where $\Delta \in \mathbb{R}^+$ is a positive real number indicating the difference $w_{new} - w_{old}$. However, when the weights decrease, we require both the old and the new weight in the update. To see why this is necessary, supposed we only included the difference. Consider the sequence (a,2), (b,1), (a,-1), (b,-1), (c,2) with a sample of size $m = 1$. At the end of this sequence, a should be in the sample with probability $1/3$, and c with probability $2/3$. b should be in the sample with probability zero. However, the only way to ensure that b is not in the sample is to delete b when its weight decreases. But note that this implies deleting a from the sample when a's weight decreases too, since a's weight decreases by the same amount at the same time.

There are two variants of the problem: sampling with replacement and sampling without replacement. In the former variant, the same item is allowed to occur multiple times in the sample. The only requirement is that each member of the sample is an independent draw from the set of items, with each item sampled with probability proportional to its weight. In the latter variant, the sample must consist of m *unique* items. This latter variant is harder, since the sampling-with-replacement problem can be solved by using m different copies of a samping-without-replacement algorithm with sample-size 1. We discuss the details below.

2.2 Sampling with Replacement

Sampling m items with replacement is reducible to the case of sampling a single item without replacement. We simply repeat the algorithm m times. Sampling a single item (m=1) with the correct probability is relatively easy, but still not trivial. The difficulty stems from weight decreases. If all the updates were increases, we could simply keep a running total $t = \sum_i w_i$ and then upon receiving the next item (k, Δ) where $\Delta = w_{new} - w_{old} > 0$ we swap the current item k into the sample with probability Δ/t. However, what if the update is a decrease of the form (k, w_{old}, w_{new}) where $w_{old} > w_{new}$ and k is currently in the sample? We could delete k from the sample but that would likely leave a "hole" in the sample. Moreover, conditioned on the hole being filled, the new entries will be sampled with relatively greater probability than the old items (since the denominator t has now decreased).

Let's use the following sequence as an example: $(a,2),(b,1),(a,2 \to 1),(c,1)$. At the end of this sequence, a, b, and c all have weight 1, so they should all appear in the sample with equal probability. But naively, it is not clear how to make this happen. Specifically, it is not clear how the algorithm should behave when the decrease occurs. Suppose the algorithm simply deletes a from the sample (if it appears), then updates the total weight t (which decreases by one in this case), and swaps a back into the sample with probability proportional to its new weight. If a does not appear, then the total t is simply updated, and future inserts occur with probability computed using this new total. With this strategy, the probability that a appears in the sample at the end of the sequence is $2/3 * 1/2 = 1/3$, the probability that c appears is also $1/3$, but the probability that b occurs is $2/3 * 1/3 = 2/9$. Thus, b is sampled disproportionally less (the remaining $1/9$ probability is the chance the sample remains empty at the end of this sequence, since nothing is swapped in to replace the deleted a).

The trick here is to adopt a weighted version of the *Random Pairing* method of [14]. This method ensures the correct probabilities by pairing *weight increases* later in the stream with *weight decreases* earlier in the stream. In essence, the keys with weight increases are *fulfilling the role* of the earlier ones whose weight decreased.

Here is the Dynamic Weighted Reservoir algorithm for sampling a single item ($m = 1$):

```
Dynamic Weighted Reservoir Sample
1: $S$: the sampled item
2: $T$: the total weight of all items, initialized to 0
3: $d$: weight of uncompensated deletions, always non-negative, initialized to 0
4: rand: returns a uniform random number in the interval $[0, 1)$
5:
6: procedure INCREMENT($k, w$)                    ▷ Add $w > 0$ to item $k$'s current weight.
```

```
7:      $T \leftarrow T + w$
8:      **if** $d = 0$ **then**
9:          **if** rand() $< \frac{w}{T}$ **then**
10:             $S \leftarrow k$
11:     **else if** $d \geq w$ **then**
12:         **if** $S = \emptyset$ **then**
13:             **if** rand() $< \frac{w}{d}$ **then**
14:                 $S \leftarrow k$
15:         $d \leftarrow d - w$
16:     **else if** $w > d$ **then**
17:         INCREMENT(k,d)
18:         INCREMENT(k,w-d)
19:
20: **procedure** DELETE($k, w$)                    ▷ Delete item $k$ whose current weight is $w$
21:     $T \leftarrow T - w$
22:     $d \leftarrow d + w$
23:     **if** $k \in S$ **then**
24:         $S \leftarrow \emptyset$
25:
26: **procedure** REWEIGHT($k, w_{old}, w_{new}$):    ▷ When $w_{new} > w_{old}$ just use INCREMENT
27:     DELETE($k, w_{old}$)
28:     INCREMENT($k, w_{new}$)
```

We prove the correctness of the algorithm via the following Theorem:

Theorem 4. *After the i'th stream operation, let T_i be the total weight of all the elements, d_i be the value of d, and $w(k)_i$ be the weight of item k. Then for all $i \geq 1$, after the i'th operation the probability of the sample being empty is $\frac{d_i}{T_i+d_i}$, and the probability of the sample containing item k is $\frac{w(k)_i}{T_i+d_i}$.*

Proof. We will prove the theorem by induction. For the base case, we assume that the first operation is not a deletion, so the value of d is 0. It is easy to see that the call INCREMENT(k, w) just inserts k with probability $\frac{w(k)_1}{T_1} = 1$. All other possibilities occur with zero probability, and all other items have zero weight, so the lemma holds.

Now assume the lemma holds for operation $i-1$. We will show it holds for operation i. We divide this into two cases:

Case i: DELETE operations. Suppose operation i is a delete operation of the form DELETE($k, w(k)_{i-1}$). By the inductive hypothesis, the probability that the sample is already empty is $\frac{d_{i-1}}{T_{i-1}+d_{i-1}}$ and the probability that the sample equals k is $\frac{w(k)_{i-1}}{T_{i-1}+d_{i-1}}$. Thus,

$$\begin{aligned}
\Pr[S=\emptyset \text{ after operation i}] &= \frac{d_{i-1}+w(k)_{i-1}}{T_{i-1}+d_{i-1}} && (2.1)\\
&= \frac{d_i}{T_{i-1}+d_{i-1}} && (2.2)\\
&= \frac{d_i}{T_{i-1}-w(k)_{i-1}+d_{i-1}+w(k)_{i-1}} && (2.3)\\
&= \frac{d_i}{T_i+d_i} && (2.4)
\end{aligned}$$

Moreover, for another item l not equal to k, the probability that the sample equals l after deleting item k is unchanged. Formally, it is $\frac{w(l)_{i-1}}{T_{i-1}+d_{i-1}} = \frac{w(l)_i}{T_i+d_i}$.

Case ii: INCREMENT operations. Suppose operation i is an increment operation of the form INCREMENT(k,a). It suffices to handle the two cases: $d_{i-1}=0$ and $d_{i-1}\geq a$. It is easy to verify that the third case, $d_{i-1}<a$, is handled by the recursion.

First we examine the probability of the sample being empty. If $d_{i-1}=0$ then it is easy to see that $d_i=0$ as well, and there is no probability of the sample being empty.

If $d_{i-1}>a$ then the probability of being empty is equal to the probability that the sample is already empty, times the probability that it does not get filled.

$$\begin{aligned}
\Pr[S=\emptyset \text{ after operation i}] &= \frac{d_{i-1}}{T_{i-1}+d_{i-1}}\cdot(1-\frac{a}{d_{i-1}}) && (2.5)\\
&= \frac{d_{i-1}-a}{T_{i-1}+d_{i-1}} && (2.6)\\
&= \frac{d_i}{T_{i-1}+d_{i-1}} && (2.7)\\
&= \frac{d_i}{T_{i-1}+a+d_{i-1}-a} && (2.8)\\
&= \frac{d_i}{T_i+d_i} && (2.9)
\end{aligned}$$

Now we examine the probability that the sample S contains a particular item. For the item k being incremented, we examine the probability that S contains k.

If $d_{i-1}=0$, then probability that the sample gets item k is the probability that the sample already contained item k, plus the probability that the sample didn't contain item k, but gets assigned it on line 10, i.e.

$$
\begin{aligned}
\Pr[k \in S \text{ after operation i}] &= \frac{w(k)_{i-1}}{T_{i-1}} + (1 - \frac{w(k)_{i-1}}{T_{i-1}}) \cdot \frac{a}{T_i} && (2.10)\\
&= \frac{w(k)_i - a}{T_i - a} + (\frac{T_i - w(k)_i}{T_i - a}) \cdot \frac{a}{T_i} && (2.11)\\
&= \frac{w(k)_i}{T_i} && (2.12)
\end{aligned}
$$

If $d_{i-1} > a$, then probability that the sample contains k is the probability that the sample already contained item k, plus the probability that the sample is empty, and then gets filled with item k on line 14. That is

$$
\begin{aligned}
\Pr[k \in S \text{ after operation i}] &= \frac{w(k)_{i-1}}{T_{i-1} + d_{i-1}} + \frac{d_{i-1}}{T_{i-1} + d_{i-1}} \cdot \frac{a}{d_{i-1}} && (2.13)\\
&= \frac{w(k)_{i-1} + a}{T_{i-1} + d_{i-1}} && (2.14)\\
&= \frac{w(k)_i}{T_{i-1} + a + d_{i-1} - a} && (2.15)\\
&= \frac{w(k)_i}{T_i + d_i} && (2.16)
\end{aligned}
$$

Since the probability of being empty and the probability of being k are correct, the probability of being any other item j must be proportionally correct from earlier steps.

□

2.3 Sampling without Replacement

Weighted sampling without replacement is a more subtle problem than sampling with replacement. Part of the subtlety comes from simply defining the problem. There are least two different variants of the sampling-without-replacement problem: sampling-with-defined-weights, and sampling-with-defined-probabilities (see the discussion in Efraimidis [10]). The first definition they credit to Chao [4]:

Definition 3 (Chao)**.** *(Weighted Random Sampling without Replacement, Proportional).*
Input: A population of n weighted items and a size m for the random sample.
Output: A weighted random sample of size m. The probability of each item to be included in the random sample is proportional to its relative weight.

Note that the definition of Chao does not always make sense. For instance, when $m = n$, every item will be chosen with probability 1, regardless of the relative weights. To remove this complication, we say that a weight is *infeasible* if it is greater than $1/m$, meaning that it should be chosen with probability greater than 1. We then stipulate that an algorithm which solves the weighted random sampling problem (under Chao's definition) must choose all items with infeasible weights with probability 1, until the set of resulting weights becomes feasible.

The definition of Chao is appealing because when the weights are feasible, it is easy to understand and compute the final probability of each item ending up in the sample. The appeal diminishes, however, when we note that the feasibility constraints imply that *the order of the stream matters*. For instance, consider a sample of size 2 from the stream $1,2,3,4$. It is easy to see that the feasibility constraint implies that item 3 with either item 1 or 2 with probability 1. Hence there is no chance that the final sample will contain both items 1 and 2. However, if the order is reversed, and the items are inserted in decreasing weight order $4,3,2,1$ then all probabilities will be feasible at all times, and hence the probability that both 1 and 2 end up in the sample is non-negative.

The reliance on stream order makes this definition of sampling without replacement unsuitable for our purposes. With dynamically changing weights, there is no guarantee that the weights will remain feasible at all times, even if the final set of weights is feasible.

An alternative definition of weighted sampling with replacement comes from the work of [11].

Definition 4. *(Weighted Random Sampling without Replacement, Sequential)*
Input: A population of n weighted items and a size m for the random sample,
Output: A weighted random sample of size m. In each round, the probability of every unselected item to be selected in that round is proportional to the relative item weight with respect to the weights of all unselected items.

In other words, this definition of weighted sampling without replacement works according to a sequential procedure: Select the first item with probability proportional to its weight, then select the next item with probability proportional to its weight among the remaining items, and continue like this until m items are selected.

This sequential procedure defines a probability distribution over *permutations* of items. What Efraimidis and Spirakis show in [11] is that if for each item you draw a random uniform number U_i from $[0,1]$, then associate the key $k_i = U_i^{w_i}$ with the item, then the distribution over permutations of the items implied by the ordering of the k_i's is the same as with the sequential procedure. Thus, to maintain a reservoir of size m, we can simply maintain an ordered list of the m largest k_i's, and items will appear in the sample with probability equal to the sequential procedure.

The naive approach to deal with deletions using [11] is to simply keep a larger reservoir of say $m' > m$ items, and insert an item if its random key k_i is larger than the smallest item in the reservoir. If an item is deleted, we simply delete it from the reservoir, and hope that we still have more than m items left. For new items, we continue computing keys before and insert if the new key is larger than any currently stored one. The problem with this approach is that if the reservoir size ever dips below m, then it will never get back up.

For instance, if we want a sample of a single item (as above) we could keep track of the top 100 keys, and at any given time our sample is simply the largest key. As long as, at any given time, we haven't deleted all of the top 100 items, then our sample will be correct. However, in the rare event that there is a sequence of deletions which *does* delete the top 100 items, then our sample will be empty no matter how many items are inserted in the future. We could try to remedy this by always taking the next item in the event that the sample is empty, but this is incorrect, since it ignores that item's weight.

2.4 Open Questions

There are multiple open questions from this work. For instance:

- Is it possible to develop a weighted random sampling algorithm with dynamically changing weights, *without replacement* under the sequential model? We have a presented an algorithm to do it *with replacement*, and [11] shows that you can sample without replacement when the weights are not evolving. Can we combine the two?
- What other streaming algorithms can benefit from the sampling routine presented here? Reservoir sampling is a basic building block of many streaming algorithms, so it is natural to wonder whether other streaming algorithms can also be made to work on dynamically changing data sets.

We anticipate that these questions will inspire future research on streaming algorithms for a dynamic, evolving world.

References

[1] Grey Ballard, Ali Pinar, Tamara G. Kolda, and C. Seshadri. Diamond sampling for approximate maximum all-pairs dot-product (MAD) search. In *ICDM 2015: Proceedings of the 2015 IEEE International Conference on Data Mining*, November 2015.

[2] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999.

[3] Yousra Chabchoub and Georges Hébrail. Sliding hyperloglog: Estimating cardinality in a data stream over a sliding window. In *2010 IEEE International Conference on Data Mining Workshops*, pages 1297–1303. IEEE, 2010.

[4] M. T. CHAO. A general purpose unequal probability sampling plan. *Biometrika*, 69(3):653, 1982.

[5] Aaron Clauset, Cosma Rohilla Shalizi, and Mark EJ Newman. Power-law distributions in empirical data. *SIAM review*, 51(4):661–703, 2009.

[6] Edith Cohen and David D. Lewis. Approximating matrix multiplication for pattern recognition tasks. In *SODA '97: Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 682–691, 1997.

[7] Edith Cohen and David D. Lewis. Approximating matrix multiplication for pattern recognition tasks. *J. Algorithms*, 30(2):211–252, 1999.

[8] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. *SIAM Journal on Computing*, 31(6):1794–1813, 2002.

[9] Tom Dial. https://github.com/dialtr/libcount, 2016.

[10] Pavlos S. Efraimidis. *Weighted Random Sampling over Data Streams*, pages 183–195. Springer International Publishing, Cham, 2015.

[11] Pavlos S Efraimidis and Paul G Spirakis. Weighted random sampling with a reservoir. *Information Processing Letters*, 97(5):181–185, 2006.

[12] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In *ACM SIGCOMM computer communication review*, volume 29, pages 251–262. ACM, 1999.

[13] Philippe Flajolet, Philippe Flajolet, Éric Fusy, Olivier Gandouet, and et" al. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. *IN AOFA '07: PROCEEDINGS OF THE 2007 INTERNATIONAL CONFERENCE ON ANALYSIS OF ALGORITHMS*, 2007.

[14] Rainer Gemulla, Wolfgang Lehner, and Peter J. Haas. A dip in the reservoir: Maintaining sample synopses of evolving datasets. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 595–606. VLDB Endowment, 2006.

[15] Aric Hagberg, Alex Kent, Nathan Lemons, and Joshua Neil. Credential hopping in authentication graphs. In *2014 International Conference on Signal-Image Technology Internet-Based Systems (SITIS)*. IEEE Computer Society, Nov. 2014.

[16] Dong Han, Fugee Tsung, and Xianghui Ning. Detection and diagnosis of distribution changes of degree ratio in complex networks. *Communications in Statistics-Theory and Methods*, 44(9):1911–1938, 2015.

[17] Stefan Heule, Marc Nunkesser, and Alexander Hall. Hyperloglog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 683–692. ACM, 2013.

[18] H. Jegou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, 2011.

[19] Tamara G. Kolda, Ali Pinar, Todd Plantenga, and C. Seshadhri. A scalable generative graph model with community structure. *SIAM Journal on Scientific Computing*, 36(5):C424–C452, 2014.

[20] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 1(1):2, 2007.

[21] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[22] C. Seshadhri, Ali Pinar, and Tamara G. Kolda. Wedge sampling for computing clustering coefficients and triangle counts on large graphs. *Statistical Analysis and Data Mining*, 7(4):294–307, August 2014.

[23] Olivia Simpson, C. Seshadhri, and Andrew McGregor. Catching the head, tail, and everything in between: A streaming algorithm for the degree distribution. In *Proceedings of the 2015 IEEE International Conference on Data Mining (ICDM)*, ICDM '15, pages 979–984, Washington, DC, USA, 2015. IEEE Computer Society.

[24] Olivia Simpson, C. Seshadhri, and Andrew McGregor. Catching the head, the tail, and everything in between: a streaming algorithm for the degree distribution. *CoRR*, abs/1506.02574, 2015.

[25] Andrew Stolman and Kevin Matulef. https://github.com/matulef/hyperheadtail, 2016.

[26] Yaonan Zhang, Eric D Kolaczyk, Bruce D Spencer, et al. Estimating network degree distributions under sampling: An inverse problem, with applications to monitoring social media networks. *The Annals of Applied Statistics*, 9(1):166–199, 2015.

DISTRIBUTION:

1	MS 0899	Technical Library, 8944 (electronic copy)
1	MS 0359	D. Chavez, LDRD Office, 1911

v1.40

Sandia National Laboratories