

# A Near-Optimal Algorithm for Computing the Entropy of a Stream\*

Amit Chakrabarti<sup>†</sup>  
ac@cs.dartmouth.edu

Graham Cormode<sup>‡</sup>  
graham@research.att.com

Andrew McGregor<sup>§</sup>  
andrewm@ucsd.edu

## Abstract

We describe a simple algorithm for approximating the empirical entropy of a stream of  $m$  values up to a multiplicative factor of  $(1 + \varepsilon)$  using a single pass,  $O(\varepsilon^{-2} \log(\delta^{-1}) \log m)$  words of space, and  $O(\log \varepsilon^{-1} + \log \log \delta^{-1} + \log \log m)$  processing time per item in the stream. Our algorithm is based upon a novel extension of a method introduced by Alon, Matias, and Szegedy [1]. This improves over previous work on this problem [10, 20, 24, 7]. We show a space lower bound of  $\Omega(\varepsilon^{-2} / \log^2(\varepsilon^{-1}))$ , demonstrating that our algorithm is near-optimal in terms of its dependency on  $\varepsilon$ .

We show that generalizing to multiplicative-approximation of the  $k$ -th order entropy requires close to linear space for  $k \geq 1$ . In contrast we show that additive-approximation is possible in a single pass using only poly-logarithmic space. Lastly, we show how to compute a multiplicative approximation to the entropy of a random walk on an undirected graph.

**General Terms** Algorithms

**Classification** G.3 Probability and Statistics – Probabilistic Algorithms

**Key Words** Data Streams, Approximation Algorithms, Entropy

## 1 Introduction

The problem of computing the frequency moments of a stream [1] has stimulated significant research within the algorithms community, leading to new algorithmic techniques and lower bounds. For all frequency moments, matching upper and lower bounds for the space complexity are now known [11, 29, 23, 8]. Subsequently, attention has been focused on the strongly related question of computing the *entropy* of a stream. Motivated by networking applications [19, 28, 30], several partial results have been shown on computing the (empirical) entropy of a sequence of  $m$  items in sublinear space [10, 20, 24, 7]. In this paper, we show a simple algorithm for computing an  $(\varepsilon, \delta)$ -approximation to this quantity in a single pass, using  $O(\varepsilon^{-2} \log(\delta^{-1}) \log m)$  words of space. We also show a lower bound of  $\Omega(\varepsilon^{-2} / \log^2(\varepsilon^{-1}))$ , proving that our algorithm is near-optimal in terms of its dependency on  $\varepsilon$ . We then give algorithms and lower bounds for  $k$ -th order entropy, a quantity that arises in text compression, based on our results for empirical (zeroth order)

---

\* An earlier version of this work appeared in the 18th Symposium on Discrete Algorithms [12]

<sup>†</sup> Work supported by NSF Grant EIA-98-02068, NSF CAREER award and by Dartmouth College startup funds.

<sup>‡</sup> Work carried out in part while at Lucent Bell Laboratories.

<sup>§</sup> Work carried out in part while at the University of Pennsylvania and supported by ONR N00014-04-1-0735.

entropy. We also provide algorithms to multiplicatively approximate the entropy of a random walk over an undirected graph. Our techniques are based on a method originating with Alon, Matias, and Szegedy [1]. However, this alone is insufficient to approximate the entropy in bounded space. At the core of their method is a procedure for drawing a uniform sample from the stream. We show how to extend this to drawing a larger sample, according to a specific distribution, of distinct values from the stream. The idea is straightforward to implement, and may have applications to other problems. For the estimation of entropy we will show that keeping a “backup sample” for each estimator is sufficient to guarantee the desired space bounds. In Section 2, we discuss this case and present our algorithm for approximating entropy and in Section 3 we present a time-efficient implementation of this algorithm along with a space lower bound and an adaptation for the sliding-window model. The results pertaining to  $k$ -th order entropy are in Section 4. The extension to entropy of a random walk on a graph is in Section 5.

**Preliminaries:** A randomized algorithm is said to  $(\varepsilon, \delta)$ -approximate a real number  $Q$  if it outputs a value  $\hat{Q}$  such that  $|\hat{Q} - Q| \leq \varepsilon Q$  with probability at least  $(1 - \delta)$  over its internal coin tosses. Our goal is to produce such  $(\varepsilon, \delta)$ -approximations for the entropy of a stream. We first introduce some notation and definitions.

**Definition 1.** For a data stream  $A = \langle a_1, a_2, \dots, a_m \rangle$ , with each token  $a_j \in [n]$ , we define  $m_i := |\{j : a_j = i\}|$  and  $p_i := m_i/m$ , for each  $i \in [n]$ . The *empirical probability distribution* of  $A$  is  $p := (p_1, p_2, \dots, p_n)$ . The *empirical entropy* of  $A$  is defined<sup>1</sup> as  $H(p) := \sum_{i=1}^n -p_i \lg p_i$ . The *entropy norm* of  $A$  is  $F_H := \sum_{i=1}^n m_i \lg m_i$ .  $\square$

Clearly  $F_H$  and  $H$  are closely related, since we can write  $F_H = m \lg m - mH$ . However, they differ significantly in their approximability:  $F_H$  cannot be approximated within constant factors in poly-logarithmic space [10], while we show here an  $(\varepsilon, \delta)$ -approximation of  $H$  in poly-logarithmic space.

**Prior Work:** Predating the recent work on approximating entropy in the data-stream model, Batu et al. [6] considered the problem of approximating entropy in a variety of oracle models including the *combined oracle model* in which an algorithm may request independent samples from the underlying distribution or request the exact value  $p_i$ . Guha, McGregor, and Venkatasubramanian [20] improved upon their results and observed that they led to a two-pass, additive approximation in poly-logarithmic space. They also presented a one-pass, poly-logarithmic space, algorithm that approximated  $H$  up to a constant factor if  $H$  was constant. Chakrabarti, Do Ba, and Muthukrishnan [10] gave a one-pass algorithm for approximating  $H$  up to a  $(1 + \varepsilon)$  factor with sublinear but polynomial in  $m$  space, as well as a two-pass algorithm requiring only poly-logarithmic space. In the networking world, the problem of approximating the entropy of a stream was considered in Lall et al. [24]. They focused on estimating  $F_H$ , under assumptions about the distribution defined by the stream that ensured that computing  $H$  based on their estimate of  $F_H$  would give accurate results. More recently, Bhuvanagiri and Ganguly [7] described an algorithm that can approximate  $H$  in poly-logarithmic space in a single pass. The algorithm is based on the same ideas and techniques as recent algorithms for optimally approximating frequency moments [23, 8], and can tolerate streams in which previously observed items are removed. The exact space bound is

$$O\left(\varepsilon^{-3}(\log^4 m)(\log \delta^{-1}) \frac{\log m + \log n + \log \varepsilon^{-1}}{\log \varepsilon^{-1} + \log \log m}\right),$$

which is suboptimal in its dependency on  $\varepsilon$ , and has high cost in terms of  $\log m$ .

<sup>1</sup>Here and throughout we use  $\lg x$  to denote  $\log_2 x$ .

## 2 Computing the Entropy of a Stream

Consider a data stream  $A$  of length  $m$ , with  $m_i$  and  $n$  defined as in Definition 1. For a real-valued function  $f$  such that  $f(0) = 0$ , we define the following notation:

$$\bar{f}(A; m) := \frac{1}{m} \sum_{i=1}^n f(m_i).$$

We base our approach on the method of Alon, Matias and Szegedy [1] to estimate quantities of the form  $\bar{f}(A; m)$ : note that the empirical entropy of  $A$  is one such quantity with  $f(m_i) = m_i \log(m/m_i)$ .

**Definition 2.** Let  $\mathcal{D}(A)$  be the distribution of the random variable  $R$  defined thus: Pick  $J \in [m]$  uniformly at random and let  $R = |\{j : a_j = a_J, J \leq j \leq m\}|$ .  $\square$

The core idea is to space-efficiently generate a random variable  $R \sim \mathcal{D}(A)$ . For an integer  $c$ , define the random variable

$$\text{Est}_f(R, c) := \frac{1}{c} \sum_{i=1}^c X_i, \tag{1}$$

where the random variables  $\{X_i\}$  are independent and each distributed identically to  $(f(R) - f(R-1))$ . Appealing to Chernoff-Hoeffding bounds one can show that by increasing  $c$ ,  $\text{Est}_f(R, c)$  can be made arbitrarily close to  $\bar{f}(A; m)$ . This is formalized in the lemma below.

**Lemma 1.** Let  $X := f(R) - f(R-1)$ ,  $a, b \geq 0$  such that  $-a \leq X \leq b$ , and

$$c \geq 3(1 + a/\mathbb{E}[X])^2 \varepsilon^{-2} \ln(2\delta^{-1})(a+b)/(a + \mathbb{E}[X]).$$

Then  $\mathbb{E}[X] = \bar{f}(A; m)$  and, if  $\mathbb{E}[X] \geq 0$ , the estimator  $\text{Est}_f(R, c)$  gives an  $(\varepsilon, \delta)$ -approximation to  $\bar{f}(A; m)$  using space  $c$  times the space required to maintain  $R$ .

*Proof.* A straightforward calculation of the expectation shows that  $\mathbb{E}[X] = \bar{f}(A; m)$ . The claim about the space required to maintain the estimator  $\text{Est}_f(R, c)$  is even simpler. So, we focus on the claim about the approximation guarantee of the estimator.

Consider the random variable  $Y := (X + a)/(a + b)$ . First note that  $Y \in [0, 1]$  and that  $\mathbb{E}[Y] = (\bar{f}(A; m) + a)/(a + b)$ . Therefore, Chernoff-Hoeffding bounds imply that, if  $\{Y_i\}$  are independent and each distributed identically to  $Y$ , then

$$\begin{aligned} & \Pr \left[ \left| \frac{1}{c} \sum_{i \in [c]} Y_i - \frac{\bar{f}(A; m) + a}{a + b} \right| > \frac{\varepsilon}{1 + a/\mathbb{E}[X]} \frac{\bar{f}(A; m) + a}{a + b} \right] \\ &= \Pr \left[ \left| \frac{1}{c} \sum_{i \in [c]} Y_i - \mathbb{E}[Y] \right| > \frac{\varepsilon}{1 + a/\mathbb{E}[X]} \mathbb{E}[Y] \right] \\ &\leq 2 \exp \left( -c \left( \frac{\varepsilon}{1 + a/\mathbb{E}[X]} \right)^2 \frac{\bar{f}(A; m) + a}{3(a + b)} \right) \\ &\leq \delta. \end{aligned}$$

Consequently  $\text{Est}'_f(R, c) = c^{-1} \sum_{i \in [c]} Y_i$  is an  $(\varepsilon/(1 + a/\mathbb{E}[X]), \delta)$ -approximation to the quantity  $(\bar{f}(A; m) + a)/(a + b)$ . Note that,  $\text{Est}'_f(R, c) = (\text{Est}_f(R, c) + a)/(a + b)$ . This implies that,

$$\begin{aligned} & \Pr \left[ |\text{Est}'_f(R, c) - \bar{f}(A; m)| > \varepsilon \bar{f}(A; m) \right] \\ &= \Pr \left[ |(a + b) \text{Est}'_f(R, c) - \bar{f}(A; m) - a| > \varepsilon \bar{f}(A; m) \right] \\ &= \Pr \left[ \left| \text{Est}'_f(R, c) - \frac{\bar{f}(A; m) + a}{a + b} \right| > \frac{\varepsilon}{1 + a/\mathbb{E}[X]} \frac{\bar{f}(A; m) + a}{a + b} \right] \\ &\leq \delta . \end{aligned}$$

Therefore,  $\text{Est}_f(R, c)$  gives an  $(\varepsilon, \delta)$ -approximation to  $\bar{f}(A; m)$  as claimed.  $\square$

**Overview of the technique:** We now give some of the intuition behind our algorithm for estimating  $H(p)$ . Let  $A'$  denote the substream of  $A$  obtained by removing from  $A$  all occurrences of the most frequent token (with ties broken arbitrarily) and let  $R' \sim \mathcal{D}(A')$ . A key component of our algorithm (see Algorithm *Maintain-Samples* below) is a technique to simultaneously maintain  $R$  and enough extra information that lets us recover  $R'$  when we need it. Let  $p_{\max} := \max_i p_i$ . Let the function  $\lambda_m$  be given by

$$\lambda_m(x) := x \lg(m/x), \text{ where } \lambda_m(0) := 0, \quad (2)$$

so that  $\bar{\lambda}_m(A; m) = H(p)$ . Define  $X = \lambda_m(R) - \lambda_m(R - 1)$  and  $X' = \lambda_m(R') - \lambda_m(R' - 1)$ . If  $p_{\max}$  is bounded away from 1 then we can show that  $1/\mathbb{E}[X]$  is “small,” so  $\text{Est}_{\lambda_m}(R, c)$  gives us our desired estimator for a “small” value of  $c$ , by Lemma 1. If, on the other hand,  $p_{\max} > \frac{1}{2}$  then we can recover  $R'$  and can show that  $1/\mathbb{E}[X']$  is “small.” Finally, by our analysis we can show that  $\text{Est}_{\lambda_m}(R', c)$  and an estimate of  $p_{\max}$  can be combined to give an  $(\varepsilon, \delta)$ -approximation to  $H(p)$ . This logic is given in Algorithm *Entropy-Estimator* below.

Thus, our algorithm must also maintain an estimate of  $p_{\max}$  in parallel to Algorithm *Maintain-Samples*. There are a number of ways of doing this and here we choose to use the Misra-Gries algorithm [25] with a sufficiently large number of counters. This (deterministic) algorithm takes a parameter  $k$  — the number of counters — and processes the stream, retaining up to  $k$  pairs  $(i, \hat{m}_i)$ , where  $i$  is a token and the counter  $\hat{m}_i$  is an estimate of its frequency  $m_i$ . The algorithm starts out holding no pairs and implicitly setting each  $\hat{m}_i = 0$ . Upon reading a token,  $i$ , if a pair  $(i, \hat{m}_i)$  has already been retained, then  $\hat{m}_i$  is incremented; else, if fewer than  $k$  pairs have been retained, then a new pair  $(i, 1)$  is created and retained; else,  $\hat{m}_j$  is decremented for each retained pair  $(j, \hat{m}_j)$  and then all pairs of the form  $(j, 0)$  are discarded. The following lemma summarizes the key properties of this algorithm; the proof is simple (see, e.g., [9]) and we omit it.

**Lemma 2.** *The estimates  $\hat{m}_i$  computed by the Misra-Gries algorithm using  $k$  counters satisfy  $\hat{m}_i \leq m_i$  and  $m_i - \hat{m}_i \leq (m - m_i)/k$ .  $\square$*

We now describe our algorithm more precisely with some pseudocode. By abuse of notation we use  $\text{Est}_{\lambda_m}(r, c)$  to also denote the algorithmic procedure of running in parallel  $c$  copies of an algorithm that produces  $r$  and combining these results as in (1).

**Maintaining Samples from the Stream:** We show a procedure that allows us to generate  $R$  and  $R'$  with the appropriate distributions. For each token  $a$  in the stream, we draw  $t$ , a random number in the range  $[m^3]$ , as its *label*. We choose to store certain tokens from the stream, along with their label and the count of the number of times the same token has been observed in the stream since it was last picked. We store *two* such

**Algorithm *Maintain-Samples***

```

1. for  $a \in A$ 
2.   do Let  $t$  be a random number in the range  $[m^3]$ 
3.   if  $a = s_0$ 
4.     then if  $t < t_0$  then  $(s_0, t_0, r_0) \leftarrow (a, t, 1)$  else  $r_0 \leftarrow r_0 + 1$ 
5.     else if  $a = s_1$  then  $r_1 \leftarrow r_1 + 1$ 
6.     if  $t < t_0$ 
7.       then  $(s_1, t_1, r_1) \leftarrow (s_0, t_0, r_0); (s_0, t_0, r_0) \leftarrow (a, t, 1)$ 
8.       else if  $t < t_1$  then  $(s_1, t_1, r_1) \leftarrow (a, t, 1)$ 

```

**Algorithm *Entropy-Estimator***

```

1.  $c \leftarrow 16\epsilon^{-2} \ln(2\delta^{-1}) \lg(me)$ 
2. Run the Misra-Gries algorithm on  $A$  with  $k = \lceil 7\epsilon^{-1} \rceil$  counters, in parallel with Maintain-Samples
3. if Misra-Gries retains a token  $i$  with counter  $\hat{m}_i > m/2$ 
4.   then  $(i_{\max}, \hat{p}_{\max}) \leftarrow (i, \hat{m}_i/m)$ 
5.   if  $s_0 = i_{\max}$  then  $r \leftarrow r_1$  else  $r \leftarrow r_0$ 
6.   return  $(1 - \hat{p}_{\max}) \cdot \text{Est}_{\lambda_m}(r, c) + \hat{p}_{\max} \lg(1/\hat{p}_{\max})$ 
7. else return  $\text{Est}_{\lambda_m}(r_0, c)$ 

```

Figure 1: Algorithms for sampling and estimating entropy.

tokens: the token  $s_0$  that has achieved the least  $t$  value seen so far, and the token  $s_1$  such that it has the least  $t$  value of all tokens not equal to  $s_0$  seen so far. Let  $t_0$  and  $t_1$  denote their corresponding labels, and let  $r_0$  and  $r_1$  denote their counts in the above sense. Note that it is easy to maintain these properties as new items arrive in the stream, as Algorithm *Maintain-Samples* illustrates.

**Lemma 3.** *Algorithm *Maintain-Samples* satisfies the following properties. (i) After processing the whole stream  $A$ ,  $s_0$  is picked uniformly at random from  $A$  and  $r_0 \sim \mathcal{D}(A)$ . (ii) For  $a \in [n]$ , let  $A \setminus a$  denote the stream  $A$  with all occurrences of  $a$  removed. Suppose we set  $s$  and  $r$  thus: if  $s_0 \neq a$  then  $s = s_0$  and  $r = r_0$ , else  $s = s_1$  and  $r = r_1$ . Then  $s$  is picked uniformly from  $A \setminus a$  and  $r \sim \mathcal{D}(A \setminus a)$ .*

*Proof.* To prove (i), note that the way we pick each label  $t$  ensures that (w.h.p.) there are no collisions amongst labels and, conditioned on this, the probability that any particular token gets the lowest label value is  $1/m$ .

We show (ii) by reducing to the previous case. Imagine generating the stream  $A \setminus a$  and running the algorithm on it. Clearly, picking the item with the smallest  $t$  value samples uniformly from  $A \setminus a$ . Now let us add back in all the occurrences of  $a$  from  $A$ . One of these may achieve a lower  $t$  value than any item in  $A \setminus a$ , in which case it will be picked as  $s_0$ , but then  $s_1$  will correspond to the sample we wanted from  $A \setminus a$ , so we can return that. Else,  $s_0 \neq a$ , and is a uniform sample from  $A \setminus a$ . Hence, by checking whether  $s_0 = a$  or not, we can choose a uniform sample from  $A \setminus a$ . The claim about the distribution of  $r$  is now straightforward: we only need to observe from the pseudocode that, for  $j \in \{0, 1\}$ ,  $r_j$  correctly counts the number of occurrences of  $s_j$  in  $A$  from the time  $s_j$  was last picked.  $\square$

**Analysis of the Algorithm:** We now analyse our main algorithm, given in full in Figure 1.

**Theorem 4.** *Algorithm Entropy-Estimator uses space  $O(\varepsilon^{-2} \log(\delta^{-1}) \log m (\log m + \log n))$  bits and gives an  $(\varepsilon, \delta)$ -approximation to  $H(p)$ .*

*Proof.* To argue about the correctness of Algorithm *Entropy-Estimator*, we first look closely at the Misra-Gries algorithm used within it. By Lemma 2,  $\hat{p}_i := \hat{m}_i/m$  is a good estimate of  $p_i$ . To be precise,  $|\hat{p}_i - p_i| \leq (1 - p_i)/k$ . Hence, by virtue of the estimation method, if  $p_i > \frac{2}{3}$  and  $k \geq 2$ , then  $i$  must be among the tokens retained and must satisfy  $\hat{p}_i > \frac{1}{2}$ . Therefore, in this case we will pick  $i_{\max}$  — the item with maximum frequency — correctly, and  $p_{\max}$  will satisfy

$$\hat{p}_{\max} \leq p_{\max} \quad \text{and} \quad |\hat{p}_{\max} - p_{\max}| \leq \frac{1 - p_{\max}}{k}. \quad (3)$$

Let  $A, A', R, R', X, X'$  be as before. We break the proof into two cases based on the value of  $\hat{p}_{\max}$ .

**Case 1:** Suppose  $\hat{p}_{\max} \leq \frac{1}{2}$ . The algorithm then reaches Line 7. By Part (i) of Lemma 3, the returned value is  $\text{Est}_{\lambda_m}(R, c)$ . Now (3), together with  $k \geq 2$ , implies  $p_{\max} \leq \frac{2}{3}$ . We lower bound the entropy,  $H(p)$ , in this case: let  $Y \sim p$  be a random variable (i.e.,  $\Pr[Y = i] = p_i$ ), and let  $S$  be any subset of indices such that  $p(S) := \sum_{i \in S} p_i$  satisfies  $\frac{1}{3} \leq p(S) \leq \frac{2}{3}$  (given that  $p_{\max} \leq \frac{2}{3}$ , such an  $S$  is guaranteed to exist and can be found greedily). Now define the random variable  $Z$  to be 1 if  $Y \in S$ , and 0 otherwise. We have

$$\begin{aligned} H(p) &= H(Y) = H(Y, Z) = H(Z) + H(Y | Z) \geq H(Z) \\ &= -p(S) \lg p(S) - (1 - p(S)) \lg(1 - p(S)) > 0.9. \end{aligned}$$

Further, we can show that  $-\lg e \leq X \leq \lg m$ . This is because

$$\lambda'_m(x) = \frac{d}{dx} \left( x \lg \left( \frac{m}{x} \right) \right) = \lg \left( \frac{m}{x} \right) - \lg e,$$

whence  $\lambda'_m(x) - \lambda'_m(x-1) = \lg(1 - 1/x)$ , which shows that  $\lambda_m(x) - \lambda_m(x-1)$  is decreasing in the range  $[1, m]$ . The maximum value is  $\lambda_m(1) - \lambda_m(0) = \lg m$  and the minimum is  $\lambda_m(m) - \lambda_m(m-1) = -\lg e$ . Hence, Lemma 1 implies that  $c$  is large enough to ensure that the return value is a  $(\frac{3}{4}\varepsilon, \delta)$ -approximation to  $H(p)$ .

**Case 2:** Suppose  $\hat{p}_{\max} > \frac{1}{2}$ . The algorithm then reaches Line 6. By Part (ii) of Lemma 3, the return value is  $(1 - \hat{p}_{\max}) \cdot \text{Est}_{\lambda_m}(R', c) + \hat{p}_{\max} \lg(1/\hat{p}_{\max})$ , and (3) implies that  $p_{\max} > \frac{1}{2}$ . Assume, w.l.o.g., that  $i_{\max} = 1$ . Then

$$\mathbb{E}[X'] = \overline{\lambda}_m(A'; m - m_1) = \frac{1}{m - m_1} \sum_{i=2}^n \lambda_m(m_i) \geq \lg \frac{m}{m - m_1} \geq 1,$$

where the penultimate inequality follows by convexity arguments. As before,  $-\lg e \leq X \leq \lg m$ , and hence Lemma 1 implies that  $c$  is large enough to ensure that  $\text{Est}_{\lambda_m}(R', c)$  is a  $(\frac{3}{4}\varepsilon, \delta)$ -approximation to  $\overline{\lambda}_m(A'; m - m_1)$ .

Next, we show that  $\hat{p}_1 \lg(1/\hat{p}_1)$  is a  $(\frac{2}{k}, 0)$ -approximation to  $p_1 \lg(1/p_1)$ , as follows:

$$\frac{|p_1 \lg(1/p_1) - \hat{p}_1 \lg(1/\hat{p}_1)|}{p_1 \lg(1/p_1)} \leq \frac{|\hat{p}_1 - p_1|}{p_1 \lg(1/p_1)} \max_{p \in [\frac{1}{2}, 1]} \left| \frac{d}{dp} (p \lg(1/p)) \right| \leq \frac{(1 - p_1)}{k p_1 \lg(1/p_1)} \cdot \lg e \leq \frac{2}{k},$$

where the final inequality follows from the fact that  $g(p) := (1 - p)/(p \ln(1/p))$  is non-increasing in the interval  $[\frac{1}{2}, 1]$ , so  $g(p) \leq g(\frac{1}{2}) < 2$ . To see this, note that  $1 - p + \ln p \leq 0$  for all positive  $p$  and that  $g'(p) = (1 - p + \ln p)/(p \ln p)^2$ . Now observe that

$$H(p) = (1 - p_1) \overline{\lambda}_m(A'; m - m_1) + p_1 \lg(1/p_1). \quad (4)$$

From (3) it follows that  $(1 - \hat{p}_1)$  is an  $(\frac{1}{k}, 0)$ -approximation to  $(1 - p_1)$ . Note that  $\frac{1}{7}\varepsilon + \frac{3}{4}\varepsilon + \frac{3}{28}\varepsilon^2 \leq \varepsilon$  for  $\varepsilon \leq 1$ . Thus, setting  $k \geq \lceil 7\varepsilon^{-1} \rceil$  ensures that  $(1 - \hat{p}_1) \cdot \text{Est}_{\lambda_m}(R', c)$  is a  $(\varepsilon, \delta)$ -approximation to  $(1 - p_1) \overline{\lambda}_m(A'; m - m_1)$ , and  $\hat{p}_1 \lg(1/\hat{p}_1)$  is a (better than)  $(\varepsilon, 0)$ -approximation to  $p_1 \lg(1/p_1)$ . Thus, we have shown that in this case the algorithm returns a  $(\varepsilon, \delta)$ -approximation to  $H(p)$ , since both terms in (4) are approximated with relative error.

The claim about the space usage is straightforward. The Misra-Gries algorithm requires  $O(k) = O(\varepsilon^{-1})$  counters and item identifiers. Each run of Algorithm *Maintain-Samples* requires  $O(1)$  counters, labels, and item identifiers, and there are  $c = O(\varepsilon^{-2} \log(\delta^{-1}) \log m)$  such runs. Everything stored is either an item from the stream, a counter that is bounded by  $m$ , or a label that is bounded by  $m^3$ , so the space for each of these is  $O(\log m + \log n)$  bits.  $\square$

The algorithm seems to require prior knowledge of  $m$ , although an upper bound clearly suffices (we can compute the true  $m$  as the stream arrives). But we only need to know  $m$  in order to choose the size of the random labels large enough to avoid collisions. Should the assumed bound be proven too low, it suffices to extend the length of labels  $t_0$  and  $t_1$  by drawing further random bits in the event of collisions to break ties. It is clear that this does not affect the correctness of the algorithm, since it is equivalent to drawing the labels on first arrival. By the same argument as above,  $O(\log m)$  bits in total for each label (for the current value of  $m$ ) will suffice to ensure unique labels.

### 3 Extensions, Variations, and Near-Optimality

#### 3.1 Sliding Window Computation

In many cases it is desirable to compute functions not over the whole semi-infinite stream, but rather over a sliding window of the last  $W$  tokens. Our method accommodates such an extension with an  $O(\log^2 W)$  expansion of space (with high probability). Formally, define the sliding window count of  $i$  as  $m_i^w := |\{j : a_j = i \text{ and } j > m - w\}|$ . The empirical probability is given by  $p_i^w := m_i^w / w$  and  $p^w := (p_1^w, \dots, p_n^w)$ , and the empirical entropy is  $H(p^w)$ .

**Lemma 5.** *We can approximate  $H(p^w)$  for any  $w < W$  in space bounded by  $O(\varepsilon^{-2} \log(\delta^{-1}) \log^3 W)$  machine words with high probability.*

*Proof.* We present an algorithm that retains sufficient information so that, after observing the stream of values, given  $w < W$  we can recover the information that Algorithm *Entropy-Estimator* would have stored had only the most recent  $w$  values been presented to it. From this, the correctness follows immediately. Thus, we must be able to compute  $s_0^w, r_0^w, s_1^w, r_1^w, i_{\max}^w$  and  $p_{\max}^w$ , the values of  $s_0, r_0, s_1, r_1, i_{\max}$  and  $p_{\max}$  on the substreams defined by the sliding window specified by  $w$ .

For  $i_{\max}^w$  and  $p_{\max}^w$ , it is not sufficient to apply standard sliding window frequent items queries [2]. To provide the overall accuracy guarantee, we needed to approximate  $p_{\max}$  with error proportion to  $\varepsilon'(1 - p_{\max}^w)$  for a parameter  $\varepsilon'$ . Prior work gives guarantees only in terms of  $\varepsilon' p_{\max}^w$ , so we need to adopt a new approach. We replace our use of the Misra-Gries algorithm with the Count-Min sketch [13]. This is a randomized

algorithm that hashes each input item to  $O(\log \delta^{-1})$  buckets, and maintains a sum of counts within each of a total of  $O(\varepsilon^{-1} \log \delta^{-1})$  buckets. If we were able to create a CM-sketch summarizing just the most recent  $w$  tokens, then we would be able to find an  $(\varepsilon, \delta)$  approximation to  $(1 - p_{\max}^w)$ , and hence also find  $p_{\max}^w$  with error  $\varepsilon(1 - p_{\max}^w)$ . This follows immediately from the properties of the sketch proved in [13]. In order to make this valid for arbitrary sliding windows, we replace each counter within the sketch with an Exponential Histogram or Deterministic Wave data structure [16, 18]. This allows us to  $(\varepsilon, 0)$  approximate the count of each bucket in the sketch within the most recent  $w < W$  timesteps in space  $O(\varepsilon^{-1} \log^2 W)$ . Combining these and rescaling  $\varepsilon$ , one can build an  $(\varepsilon, \delta)$  approximation of  $(1 - p_{\max}^w)$  for any  $w < W$ . The space required for this estimation is  $O(\varepsilon^{-2} \log^2 W \log \delta^{-1} (\log m + \log n))$  bits.

For  $s_0^w, r_0^w, s_1^w$  and  $r_1^w$ , we can take advantage of the fact that these are defined by randomly chosen tags  $t_0^w$  and  $t_1^w$ , and for any  $W$  there are relatively few possible candidates for all the  $w < W$ . Let  $t_j$  be the random label for the  $j$ -th item in the stream. We maintain the following set of tuples,

$$S_0 := \{(j, a_j, t_j, r_j) : j = \operatorname{argmin}_{m-w < i \leq m} t_j, r_j = |\{k : a_k = a_j, k \geq j\}|, w < W\}.$$

This set defines  $j_0^w = \operatorname{argmin}_{m-w < i \leq m} t_j$  for  $w < W$ . We maintain a second set of tuples,

$$S_1 := \{(j, a_j, t_j, r_j) : j = \operatorname{argmin}_{\substack{m-w < i \leq m \\ i \neq j_0^w}} t_j, r_j = |\{k : a_k = a_j, k \geq j\}|, w < W\},$$

and this set defines  $j_1^w = \operatorname{argmin}_{m-w < i \leq m} t_j$  for  $w < W$ . Note that it is straightforward to maintain  $S_0$  and  $S_1$ . Then, for any  $w < W$ , we set

$$(s_0^w, r_0^w) \leftarrow (a_{j_0^w}, r_{j_0^w}) \quad \text{and} \quad (s_1^w, r_1^w) \leftarrow (a_{j_1^w}, r_{j_1^w}).$$

We now bound the sizes of  $S_0$  and  $S_1$ , based on the following fact:

**Fact 6.** *Let  $t = t_1 \dots t_W$  be a sequence of  $W$  integer labels, each drawn uniformly at random such that there are no duplicate values. Define a sequence of values  $T_i$  such that  $T_0 = \operatorname{argmin}_i \{t_i\}$  and  $T_j = \operatorname{argmin}_{i > T_{j-1}} \{t_i\}$ . This sequence terminates when  $T_j = W$ . For  $k$  such that  $T_k = W$ , we have  $k = O(\log W)$  with high probability.*

This fact follows from an analysis due to Babcock, Datar and Motwani [3]: the insight is that if we build a treap over the sequence  $t$  and heapify by the labels, the sequence  $T_0 \dots T_k$  gives precisely the right spine of the treap. This approach yields a strong bound on  $k$ , since with high probability the height of a treap with randomly chosen priorities such as these (i.e., a random binary search tree) is logarithmic.

This fact immediately implies a bound on the size of  $S_0$ , since  $S_0$  exactly corresponds to such a derived sequence  $T_i$ . Further, we can extend this analysis by observing that members of  $S_1$  correspond to nodes in the treap that are also on the right spine, are left children of members of  $S_0$ , or the right descendants of left children. Thus, if the treap has height  $h$ , the size of  $S_1$  is  $O(h^2)$ . For windows of size at most  $W$ , the implicit treap has height  $O(\log W)$  with high probability.

Thus, we need to store a factor of  $O(\log^2 W)$  more information for each instance of the basic estimator. The total space bound is therefore  $O(\varepsilon^{-2} \log(\delta^{-1}) \log^3 W (\log m + \log n))$  bits, since now the estimator is bounded by  $\log W$  rather than  $\log m$ .  $\square$

## 3.2 Efficient Implementation

Observe that a direct implementation of the central algorithm as described in Section 2 has a high cost per token, in terms of processing time: we track a number  $c$  of independent samples, and for each new token in the stream we test whether it is sampled by any copy of the estimator, taking time  $O(c)$ . However, also note that as the stream increases in length, it is increasingly unlikely that a new item will be sampled: over the whole stream, each estimator updates its primary sample  $O(\log m)$  times with high probability. This follows by applying Fact 6 from above over a stream indexed in reverse order: for a stream of length  $m$ , each chosen sample has a smaller label than every prior label. So, for the overwhelming majority of tuples, the decision to sample is negative. In this section, we describe a faster implementation of our main algorithm that capitalizes on these observations.

**Sampling:** We adapt an idea from reservoir sampling [27]: when choosing to sample an item, also determine when the next item will be sampled. Suppose the random label assigned to an item which is sampled is  $t$ . Then for each new item, we are effectively throwing a biased coin so that with probability  $(t - 1)/m^3$  we choose to sample that item. Thus, the number of tosses (tuples) before we choose a new item to sample is given by the geometric distribution. So we can directly draw from this distribution to determine how many items to “skip” before picking the next. The new label  $t'$  is a uniform random variable over the range of the random labels, but conditioned on the fact that  $t' < t$ . So  $t'$  can be chosen uniformly at random from the set  $[t - 1]$ .

For a “backup sample,”  $s_1$ , there are two ways that it can be replaced. Either we sample a new primary item  $s_0$ , and the old primary item replaces the backup sample, or else we observe an item not equal to  $s_0$  whose random label  $t$  satisfies  $t_0 < t < t_1$ , where  $t_0$  and  $t_1$  are the tags of the current primary and backup samples, respectively. The first case can be taken care of whenever the primary sample is replaced, as outlined above. The second case can be handled by similar logic: at each drawing, the probability of the event of the backup sample being replaced is  $(t_1 - t_0 - 1)/m^3$ , and so we can draw from an appropriate geometric distribution to determine the number of input items not equal to the primary sample that must be observed before replacing the backup sample, and also draw a new value of the label.

**Fast Data Structure Maintenance:** We next show that we can use these “waiting times” to more efficiently process the stream. We extend the information kept about each estimator, so instead of  $(s, t, r)$  triples, we now keep  $(s, t, r, u)$ , where  $u$  is used to track when to update the sample. Firstly, we must maintain the counts of the sampled items. Note that the same token,  $a$ , may be sampled by multiple independent estimators at different points in the stream. To ensure that the cost per item in the stream is independent of the number of estimators which are currently sampling the same item, we keep some additional data structures. We maintain the set of items that are currently being sampled by any estimator in an appropriate dictionary structure, along with a counter associated with that item,  $c_a$ . When an estimator chooses to sample an item  $a$  that is not currently sampled, it is added to the set, and the counter  $c_a$  is set to 1. Whenever an estimator chooses to replace its current sample with an item already in the set, instead of setting its  $r$  value to 1, it sets a variable  $r'$  to the current counter for that item,  $c_a$ . Whenever an item  $a$  is seen in the stream, we test whether it is in the set of monitored items, and if so, increment the counter  $c_a$ . When we come to make an estimate of the entropy, we can form the value  $r$  for each estimate by computing the difference between the current value of  $c_a$  and the value  $r'$  of  $c_a$  when the estimator sampled  $a$ . To maintain space bounds, when  $a$  is no longer tracked by any estimator, we remove  $a$  and  $c_a$  from the dictionary structure.

In order to determine when the waiting time for a particular estimator is met and we must use the current item in the stream as the new sample, we make use of heap structures, one for the primary samples, and one

**Algorithm** *Fast-Maintain-Samples*

```

1.  for  $i = 1$  to  $m$ 
2.      do Let  $a$  denote the  $i$ th token in the stream
3.      if  $a \in \text{Dictionary}$  then Increment  $c_a$  by 1
4.      while (Replacement Time of HeapMin of PrimaryHeap =  $i$ )
5.          do (* Replace a primary sample *)
6.              Let  $x = (s_0, r_0, t_0, u_0)$  denote HeapMin of PrimaryHeap;
7.              Let  $t$  be a random uniform number in the range  $[t_0 - 1]$ ;
8.              if  $s_0 \neq a$ 
9.                  then (* Primary Sample becomes Backup Sample *)
10.                     Let  $y$  denote the Backup Sample for  $x$ ;
11.                     Delete  $y$  from its Secondary SubHeap, and clean up if SubHeap now empty;
12.                     Insert  $(s_1 = s_0, t_1 = t_0, r_1 = r_0, u_1 = \text{Geom}((t_0 - t)/m^3))$  into Secondary
13.                     SubHeap for  $a$  as the Backup Sample for  $x$ ;
14.                     if  $a \notin \text{Dictionary}$  then Add  $a$  to Dictionary with  $c_a = 1$ 
15.                     Update  $x$  to  $(s_0 = a, r_0 = c_a, t_0 = t, u_0 = \text{Geom}(t/m^3))$ ;
16.                     Heapify PrimaryHeap;
17.                 if ( $a \in \text{SecondaryHeap}$ )
18.                     then (* Delay resampling for backups of  $a$  *)
19.                         Increase Replacement Time of  $a$  in Secondary Heap by 1;
20.                 while (Replacement Time of HeapMin of SecondaryHeap =  $i$ )
21.                     do (* Replace a backup sample *)
22.                         if  $a \notin \text{Dictionary}$  then Add  $a$  to Dictionary with  $c_a = 1$ 
23.                         Let  $y$  denote the Heap Min of Secondary Heap;
24.                         Let  $z = (s_1, r_1, t_1, u_1)$  denote the Heap Min of  $y$ 's SubHeap;
25.                         Let  $x = (s_0, r_0, t_0, u_0)$  denote the Primary Sample corresponding to  $y$ ;
26.                         Let  $t$  be a random uniform number in the range  $[t_0 \dots t_1 - 1]$ ;
27.                         Update  $z$  to  $(s_1 = a, r_1 = c_a, t_1 = t, u_1 = \text{Geom}((t - t_0)/m^3))$ ;
28.                         Heapify  $y$ 's SubHeap;
29.                         Update Replacement time for  $y$  based on new HeapMin of SubHeap,  $i$ , and  $c_a$ ;
30.                         Heapify Secondary Heap;

```

Figure 2: Efficient Implementation of the Sampling Procedure.

for the backup samples. The primary sample case is the more straightforward: using the above analysis, we compute the position in the input of the next item to sample for each estimator, and store these in a heap ordered by times  $u$ . At each time step, we test whether the current item number is equal to the smallest position in the heap. If so, we update the estimator information (sampled item, random label, count and time to resample) and heapify. This is repeated, if necessary, until the smallest position in the heap exceeds the current item number.

To maintain the backup samples, things become more involved. We keep a heap which contains only one entry for each distinct token  $a$  that is a primary sampled item. The heap key associated with this item is the smallest position at which we would sample a backup item associated with the primary item, assuming that all the intermediate items are not equal to that primary item. Thus, when we see an item  $a$  in the stream,

we first find if  $a$  is in the backup heap, and if so, delay the time to sampling by one time step (and then heapify). We make use of  $c_a$ , the number of copies of  $a$  seen in the input while  $a$  has been monitored by any estimator to derive a modified timestep for the stream  $A \setminus a$ , which has all copies of  $a$  removed. For each  $a$  in the backup heap, we also store a secondary heap consisting of all backup items whose primary item is  $a$ , ordered by their “resampling” time within  $A \setminus a$ .

The secondary heap of backup samples associated with  $a$  gets modified in various ways: (1) when the resampling time of a backup sample,  $u$ , matches the timestamp within  $A \setminus a$ , we remove the minimum value from the heap, re-heapify both the secondary heap for  $a$  and the whole backup heap of heaps (and repeat the resampling procedure if multiple backup items have the same resampling time); (2) when an estimator whose primary sample is  $a$  chooses a new backup sample, its resampling time  $u$  is drawn from the appropriate distribution, the appropriate offset in  $A \setminus a$  is calculated, and a record is inserted into the heap; (3) an item is removed from the heap, because a primary sample is replaced and the previous primary sample becomes the new backup sample. In this case, we simply remove the corresponding entry from the backup heap, and heapify. To ensure space bounds are met, if due to resampling a secondary heap for  $a$  is empty, we delete the empty heap and any related data. We illustrate this in pseudocode in Figure 2, above (we omit for clarity the details of garbage collecting from the data structure that is needed to ensure the space bounds.). Here,  $\text{Geom}(p)$  denotes a sample from a geometric distribution with probability  $p$ .

We claim that this whole procedure still ensures that the algorithm is executed correctly: if the original version of the algorithm were implemented in parallel, and made the same decisions about which items to sample, then the same information (samples and counts) would be computed by both versions. The estimation procedure is as before, however for our estimators, we derive the correct count of each sampled token  $s$  by taking the stored  $r$  value and subtracting  $c_s$ . Next we argue that the time cost of this implementation is much less than for the original version.

**Analysis:** The asymptotic space cost is unchanged, since within the various heaps, each estimator is represented at most a constant number of times, so the amount of space required is still

$$O(\varepsilon^{-2} \log(\delta^{-1}) \log m (\log m + \log n)) \text{ bits.}$$

The cost to reflect each update in the sampled token counts is a constant number of dictionary look-ups to determine whether  $a$  is sampled by any primary or backup samplers, and then a constant amount of work to update counts. This is a total of  $O(1)$  time (expected) if we implement the dictionary structures with hash tables, plus a heap operation to ensure that the heap condition is met. Over the course of the execution of algorithm we can bound the total number of times each estimator updates its primary and backup samples: using a similar argument to the sliding window analysis above, the primary sample is replaced  $O(\log m)$  times with high probability; similarly, the corresponding backup sample is replaced  $O(\log^2 m)$  times. Each heap operation takes time at most logarithmic in the number of items stored in a heap, which is bounded by the number of samples, set to  $c = O(\varepsilon^{-2} \log(\delta^{-1}) \log m)$ . Hence, we have proved the following theorem.

**Theorem 7.** *Algorithm Entropy-Estimator can be implemented such that a length  $m$  stream can be processed in  $O((m + \log^3 m \varepsilon^{-2} \log(\delta^{-1}))(\log \varepsilon^{-1} + \log \log \delta^{-1} + \log \log m))$  time.*

Observe that even for short streams, the term in  $\log m$  is dominated by the term in  $m$ , and so we simplify the bound to  $O(\log \varepsilon^{-1} + \log \log \delta^{-1} + \log \log m)$  per token in the stream. The interested reader is referred to <http://dimax.rutgers.edu/~jthaler/implementation> for code of this implementation due to Justin Thaler. Experiments using this code demonstrate that these asymptotic bounds translate into high throughput (millions of tokens processed per second), small space. They also demonstrate small relative

error over a range of entropy values, in comparison to prior algorithms which incur higher error when  $H(p)$  is small.

### 3.3 Extensions to the Technique

We observe that the method we have introduced here, of allowing a sample to be drawn from a modified stream with an item removed may have other applications. The method naturally extends to allowing us to specify a set of  $k$  items to remove from the stream after the fact, by keeping the  $k + 1$  distinct items achieving the smallest label values. In particular, Lemma 3 can be extended to give the following.

**Lemma 8.** *There exists an algorithm using  $O(k)$  space, that returns  $k$  pairs  $(s_i, r_i)_{i \in [k+1]}$  such that  $s_i$  is picked uniformly at random from  $A \setminus \{s_1, \dots, s_{i-1}\}$  and  $r \sim \mathcal{D}(A \setminus \{s_1, \dots, s_{i-1}\})$ . Consequently, given a set  $S$  of size less than  $k$  and the output of  $\mathcal{A}$  it is possible to sample  $(s, r)$  such that  $s$  is picked uniformly at random from  $A \setminus S$  and  $r \sim \mathcal{D}(A \setminus S)$ .*

This may be of use in applications where we can independently identify “junk” items or other undesirable values which would dominate the stream if not removed. For example, in the case in which we wish to compute the quantiles of a distribution after removing the  $k$  most frequent items from the distribution. Additionally, the procedure may have utility in situations where a small fraction of values in the stream can significantly contribute to the variance of other estimators.

### 3.4 Near-Optimality: A Lower Bound

We now show that the dependence of the above space bound on  $\varepsilon$  is nearly tight. To be precise, we prove the following theorem.

**Theorem 9.** *Any one-pass randomized  $(\varepsilon, 1/4)$ -approximation for  $H(p)$  requires  $\Omega(\varepsilon^{-2}/\log^2(\varepsilon^{-1}))$  bits of space.*

*Proof.* Let GAP-HAMDIST denote the following (one-way) communication problem. Alice receives  $x \in \{0, 1\}^N$  and Bob receives  $y \in \{0, 1\}^N$ . Alice must send a message to Bob after which Bob must answer “near” if the Hamming distance  $\|x - y\|_1 \leq N/2$  and “far” if  $\|x - y\|_1 \geq N/2 + \sqrt{N}$ . They may answer arbitrarily if neither of these two cases hold. The two players may follow a randomized protocol that must work correctly with probability at least  $\frac{3}{4}$ . It is known [22, 29] that GAP-HAMDIST has one-way communication complexity  $\Omega(N)$ .

We now reduce GAP-HAMDIST to the problem of approximating  $H(p)$ . Suppose  $\mathcal{A}$  is a one-pass algorithm that  $(\varepsilon, \delta)$ -approximates  $H(p)$ . Let  $N$  be chosen such that  $\varepsilon^{-1} = 3\sqrt{N}(\lg N + 1/2)$  and assume, w.l.o.g., that  $N$  is an integer. Alice and Bob will run  $\mathcal{A}$  on a stream of tokens from  $[N] \times \{0, 1\}$  as follows. Alice feeds the stream  $\langle (i, x_i) \rangle_{i=1}^N$  into  $\mathcal{A}$  and then sends over the memory contents of  $\mathcal{A}$  to Bob who then continues the run by feeding in the stream  $\langle (i, y_i) \rangle_{i=1}^N$ . Bob then looks at the output  $\text{out}(\mathcal{A})$  and answers “near” if

$$\text{out}(\mathcal{A}) < \lg N + \frac{1}{2} + \frac{1}{2\sqrt{N}}$$

and answers “far” otherwise. We now prove the correctness of this protocol.

Let  $d := \|x - y\|_1$ . Note that the stream constructed by Alice and Bob in the protocol will have  $N - d$  tokens with frequency 2 each and  $2d$  tokens with frequency 1 each. Therefore,

$$H(p) = (N - d) \cdot \frac{2}{2N} \lg \frac{2N}{2} + 2d \cdot \frac{1}{2N} \lg \frac{2N}{1} = \lg N + \frac{d}{N}.$$

Therefore, if  $d \leq N/2$ , then  $H(p) \leq \lg N + \frac{1}{2}$  whence, with probability at least  $\frac{3}{4}$ , we will have

$$\text{out}(\mathcal{A}) \leq (1 + \varepsilon)H(p) \leq \left(1 + \frac{1}{3\sqrt{N}(\lg N + 1/2)}\right) \left(\lg N + \frac{1}{2}\right) < \lg N + \frac{1}{2} + \frac{1}{2\sqrt{N}}$$

and Bob will correctly answer “near.” A similar calculation shows that if  $d \geq N/2 + \sqrt{N}$  then, with probability at least  $\frac{3}{4}$ , Bob will correctly answer “far.” Therefore the protocol is correct and the communication complexity lower bound implies that  $\mathcal{A}$  must use space at least  $\Omega(N) = \Omega(\varepsilon^{-2}/\log^2(\varepsilon^{-1}))$ .  $\square$

## 4 Higher-Order Entropy

The  $k$ -th order entropy is a quantity defined on a sequence that quantifies how easy it is to predict a character of the sequence given the previous  $k$  characters. We start with a formal definition.

**Definition 3.** For a data stream  $A = \langle a_1, a_2, \dots, a_m \rangle$ , with each token  $a_j \in [n]$ , we define

$$m_{i_1 i_2 \dots i_t} := |\{j \leq m - k : a_{j-1+l} = i_l \text{ for } l \in [t]\}|; \quad p_{i_t | i_1, i_2, \dots, i_{t-1}} := m_{i_1 i_2 \dots i_t} / m_{i_1 i_2 \dots i_{t-1}},$$

for  $i_1, i_2, \dots, i_t \in [n]$ . The (empirical)  $k$ -th order entropy of  $A$  is defined as

$$H_k(A) := - \sum_{i_1} p_{i_1} \sum_{i_2} p_{i_2 | i_1} \cdots \sum_{i_{k+1}} p_{i_{k+1} | i_1 \dots i_k} \lg p_{i_{k+1} | i_1 \dots i_k}.$$

$\square$

Unfortunately, unlike empirical entropy,  $H_0$ , there is no small space algorithm for multiplicatively approximating  $H_k$ . This is even the case for  $H_1$  as substantiated in the following theorem.

**Theorem 10.** *Approximating  $H_1(A)$  up to any multiplicative error requires  $\Omega(m/\log m)$  space.*

*Proof.* Let PREFIX denote the following (one-way) communication problem. Alice has a string  $x \in \{0, 1\}^N$  and Bob has a string  $y \in \{0, 1\}^{N'}$  with  $N' \leq N$ . Alice must send a message to Bob, and Bob must answer “yes” if  $y$  is a prefix of  $x$ , and “no” otherwise. The one-way probabilistic communication complexity of PREFIX is  $\Omega(N/\log N)$ , as the following argument shows. Suppose we could solve PREFIX using  $C$  bits of communication. Repeating such a protocol  $O(\log N)$  times in parallel reduces the probability of failure from constant to  $O(1/N)$ . But by posing  $O(N)$  PREFIX queries in response to Alice’s message in this latter protocol, Bob could learn  $x$  with failure probability at most a constant. Therefore, we must have  $C \log N = \Omega(N)$ .

Consider an instance  $(x, y)$  of PREFIX. Let Alice and Bob jointly construct the stream

$$A = \langle a_1, a_2, \dots, a_N, b_1, b_2, \dots, b_{N'} \rangle,$$

where  $a_i = (i, x_i)$  for  $i \in [N]$  and  $b_i = (i, y_i)$  for  $i \in [N']$ . Note that,

$$H_1(A) = - \sum_i p_i \sum_j p_{j|i} \lg p_{j|i} = 0$$

if  $x$  is a prefix of  $y$ . But  $H_1(A) \neq 0$  if  $x$  is not a prefix of  $y$ . This reduction proves that any multiplicative approximation to  $H_1$  requires  $\Omega(N/\log N)$  space, using the same logic as that in the conclusion of the proof of Theorem 9. Since the stream length  $m = N + N' = \Theta(N)$ , this translates to an  $\Omega(m/\log m)$  lower bound.  $\square$

Since the above theorem effectively rules out efficient multiplicative approximation, we now turn our attention to additive approximation. The next theorem (and its proof) shows how the algorithm in Section 2 gives rise to an efficient algorithm that additively approximates the  $k$ -th order entropy.

**Theorem 11.**  $H_k(A)$  can be  $\varepsilon$ -additively approximated with  $O(k^2\varepsilon^{-2} \log(\delta^{-1}) \log^2 n \log^2 m)$  space.

*Proof.* We first rewrite the  $k$ -th order entropy as follows.

$$\begin{aligned} H_k(A) &= - \sum_{i_1, i_2, \dots, i_{k+1}} p_{i_1} p_{i_2|i_1} \cdots p_{i_{k+1}|i_1 i_2 \dots i_k} \lg p_{i_{k+1}|i_1 i_2 \dots i_k} \\ &= \sum_{i_1, i_2, \dots, i_{k+1}} \frac{m_{i_1 \dots i_{k+1}}}{m-k} \lg \frac{m_{i_1 \dots i_k}}{m_{i_1 \dots i_{k+1}}} \\ &= - \sum_{i_1, i_2, \dots, i_k} \frac{m_{i_1 \dots i_k}}{m-k} \lg \frac{m-k}{m_{i_1 \dots i_k}} + \sum_{i_1, i_2, \dots, i_{k+1}} \frac{m_{i_1 \dots i_{k+1}}}{m-k} \lg \frac{m-k}{m_{i_1 \dots i_{k+1}}} \\ &= H(p^{k+1}) - H(p^k) \end{aligned}$$

where  $p^k$  is the distribution over  $n^k$  points with  $p_{i_1 i_2 \dots i_k}^k = m_{i_1 i_2 \dots i_k} / (m-k)$  and  $p^{k+1}$  is the distribution over  $n^{k+1}$  points with  $p_{i_1 i_2 \dots i_{k+1}}^k = m_{i_1 i_2 \dots i_{k+1}} / (m-k)$ . Since  $H(p^k)$  is less than  $k \lg n$ , if we approximate it to a multiplicative factor of at most  $(1 + \varepsilon / (2k \lg n))$  then we have an additive  $\varepsilon/2$  approximation. Appealing to Theorem 4 this can be done in  $O(k^2\varepsilon^{-2} \log(\delta^{-1}) \log^2(n) \log(m))$  space. We can deal with  $H(p^{k+1})$  similarly and hence we get an  $\varepsilon$  additive approximation for  $H_k(A)$ . Directly implementing these algorithms, we need to store strings of  $k$  characters from the input stream as a single  $k$ -th order character; for large  $k$ , we can hash these strings onto the range  $[m^3]$ . Since there are only  $m-k$  substrings of length  $k$ , then there are no collisions in this hashing w.h.p., and the space needed is only  $O(\log m)$  bits for each stored item or counter.  $\square$

## 5 Entropy of a Random Walk

In Theorem 10, we showed that it was impossible to multiplicatively approximate the first order entropy,  $H_1$ , of a stream in sub-linear space. In this section we consider a related quantity  $H_G$ , the *unbiased random walk entropy*. We will discuss the nature of this relationship after a formal definition.

**Definition 4.** For a data stream  $A = \langle a_1, a_2, \dots, a_m \rangle$ , with each token  $a_j \in [n]$ , we define an undirected graph  $G(V, E)$  on  $n$  vertices, where  $V = [n]$  and

$$E = \{\{u, v\} \in [n]^2 : u = a_j, v = a_{j+1} \text{ for some } j \in [m-1]\}.$$

Let  $d_i$  be the degree of node  $i$ . Then the *unbiased random walk entropy* of  $A$  is defined as,

$$H_G := \frac{1}{2|E|} \sum_{i \in [n]} d_i \lg d_i.$$

$\square$

Consider a stream formed by an unbiased random walk on an undirected graph  $G$ , i.e., if  $a_i = j$  then  $a_{i+1}$  is uniformly chosen from the  $d_j$  neighbors of  $j$ . Then  $H_G$  is the limit of  $H_1(A)$  as the length of this random walk tends to infinity:

$$H_G = \frac{1}{2|E|} \sum_{i \in [n]} d_i \lg d_i = \lim_{m \rightarrow \infty} \sum_{i \in [n]} \frac{m_i}{m} \sum_{j \in [n]} \frac{m_{ij}}{m_i} \lg \frac{m_i}{m_{ij}} = \lim_{m \rightarrow \infty} H_1(\langle a_1, a_2, \dots, a_m \rangle),$$

since  $\lim_{m \rightarrow \infty} (m_{ij}/m_i) = 1/d_i$  and  $\lim_{m \rightarrow \infty} (m_i/m) = d_i/(2|E|)$  as the stationary distribution of a random walk on an undirected graph is  $(d_1/(2|E|), d_2/(2|E|), \dots, d_n/(2|E|))$ . See Section 4.3 of Cover and Thomas [15], for example, for more context. We focus on computing  $H_G$  rather than on computing the entropy of a sample walk, since this gives greater flexibility: it can be computed on arbitrary permutations of the edges, for example, and only requires that each edge be observed at least once.

For the rest of this section it will be convenient to reason about a stream  $E'$  that can be easily transduced from  $A$ .  $E'$  will consist of  $m - 1$ , not necessarily distinct, edges on the set of nodes  $V = [n]$ ,  $E' = \langle e_1, e_2, \dots, e_{m-1} \rangle$  where  $e_i = (a_i, a_{i+1})$ . We assume that the random walk is long enough to ensure that every edge is visited at least once, so that  $E$  is the set produced by removing all duplicate edges in  $E'$ .

**Overview of the algorithm:** Our algorithm uses the standard AMS-Estimator as described in Section 2. However, because  $E'$  includes duplicate items which we wish to disregard, our basic estimator is necessarily more complicated. The algorithm combines ideas from multi-graph streaming [14] and entropy-norm estimation [10] and uses min-wise hashing [21] and distinct element estimators [4].

Ideally the basic estimator would sample a node  $w$  uniformly from the multi-set in which each node  $u$  occurs  $d_u$  times. Then let  $r$  be uniformly chosen from  $\{1, \dots, d_w\}$ . If the basic estimator were to return  $g(r) = f(r) - f(r - 1)$  where  $f(x) = x \lg x$  then the estimator would be correct in expectation:

$$\sum_{w \in [n]} \frac{d_w}{2|E|} \sum_{r \in [d_w]} \frac{1}{d_w} (f(r) - f(r - 1)) = \frac{1}{2|E|} \sum_{w \in [n]} d_w \lg d_w .$$

To mimic this sampling procedure we use an  $\varepsilon$ -min-wise hash function  $h$  [21] to map the distinct edges in  $E'$  into  $[m]$ . It allows us to pick an edge  $e = (u, v)$  (almost) uniformly at random from  $E$  by finding the edge  $e$  that minimizes  $h(e)$ . We pick  $w$  uniformly from  $\{u, v\}$ . Note that  $w$  has been chosen with probability proportional to  $(1 \pm \varepsilon)d_w/(2|E|)$ . Let  $i = \max\{j : e_j = e\}$  and consider the  $r$  distinct edges among  $\{e_i, \dots, e_m\}$  that are incident on  $w$ . Let  $e^1, \dots, e^{d_w}$  be the  $d_w$  edges that are incident on  $w$  and let  $i_k = \max\{j : e_j = e^k\}$  for  $k \in [d_w]$ . Then  $r$  is distributed as  $|\{k : i_k \geq i\}|$  and hence takes a value from  $\{1, \dots, d_w\}$  with probability  $(1 \pm \varepsilon)/d_w$ .

Unfortunately we cannot compute  $r$  exactly unless it is small. If  $r \leq \varepsilon^{-2}$  then we maintain an exact count, by keeping the set of distinct edges. Otherwise we compute an  $(\varepsilon, \delta)$ -approximation of  $r$  using a distinct element estimation algorithm, e.g., [4]. Note that if this is greater than  $n$  we replace the estimate by  $n$  to get a better bound. This will be important when bounding the maximum value of the estimator. Either way, let this (approximate) count be  $\tilde{r}$ . We then return  $g(\tilde{r})$ . The next lemma demonstrates that using  $g(\tilde{r})$  rather than  $g(r)$  only incurs a small amount of additional error.

**Lemma 12.** *Assuming  $\varepsilon < 1/4$ ,  $|g(r) - g(\tilde{r})| \leq O(\varepsilon)g(r)$  with probability at least  $1 - \delta$ .*

*Proof.* If  $r \leq \varepsilon^{-2}$ , then  $r = \tilde{r}$ , and the claim follows immediately. Therefore we focus on the case where  $r > \varepsilon^{-2}$ . Let  $\tilde{r} = (1 + \gamma)r$  where  $|\gamma| \leq \varepsilon$ . We write  $g(r)$  as the sum of the two positive terms,

$$g(r) = \lg(r - 1) + r \lg(1 + 1/(r - 1))$$

and will consider the two terms in the above expression separately.

Note that for  $r \geq 2$ ,  $\frac{\tilde{r}-1}{r-1} = 1 \pm 2\varepsilon$ . Hence, for the first term, and providing the distinct element estimation succeeds with its accuracy bounds,

$$|\lg(\tilde{r} - 1) - \lg(r - 1)| = \left| \lg \frac{\tilde{r} - 1}{r - 1} \right| = O(\varepsilon) \leq O(\varepsilon) \lg(r - 1) .$$

where the last inequality follows since  $r > \varepsilon^{-2}$ ,  $\varepsilon < 1/4$ , and hence  $\lg(r-1) > 1$ .

Note that for  $r \geq 2$ ,  $r \lg\left(1 + \frac{1}{r-1}\right) \geq 1$ . For the second term,

$$\begin{aligned} \left| r \lg\left(1 + \frac{1}{r-1}\right) - \tilde{r} \lg\left(1 + \frac{1}{\tilde{r}-1}\right) \right| &\leq |r - \tilde{r}| \lg\left(1 + \frac{1}{\tilde{r}-1}\right) + r \left| \lg\left(\frac{1 + \frac{1}{r-1}}{1 + \frac{1}{\tilde{r}-1}}\right) \right| \\ &\leq O(\varepsilon) + O(\varepsilon) \\ &\leq O(\varepsilon)r \lg\left(1 + \frac{1}{r-1}\right). \end{aligned}$$

Hence  $|g(r) - g(\tilde{r})| \leq O(\varepsilon)g(r)$  as required.  $\square$

**Theorem 13.** *There exists an  $(\varepsilon, \delta)$ -approximation algorithm for  $H_G$  using<sup>2</sup>  $O(\varepsilon^{-4} \log^2 n \log^2 \delta^{-1})$  space.*

*Proof.* Consider the expectation of the basic estimator:

$$\mathbb{E}[X] = \sum_{w \in [n]} \frac{(1 \pm O(\varepsilon))d_w}{2|E|} \sum_{r \in [d_w]} \frac{1 \pm O(\varepsilon)}{d_w} (f(r) - f(r-1)) = \frac{1 \pm O(\varepsilon)}{2|E|} \sum_{w \in [n]} d_w \lg d_w.$$

Note that since the graph  $G$  is revealed by a random walk, this graph must be connected. Hence  $|E| \geq n-1$  and  $d_w \geq 1$  for all  $w \in V$ . But then  $\sum_w d_w = 2|E| \geq 2(n-1)$  and therefore,

$$\frac{1}{2|E|} \sum_{w \in [n]} d_w \lg d_w \geq \lg \frac{2|E|}{n} \geq \lg 2(1 - 1/n).$$

The maximum value taken by the basic estimator is,

$$\max[X] \leq \max_{1 \leq r \leq n} (f(r) - f(r-1)) \leq \left( n \lg \frac{n}{n-1} + \lg(n-1) \right) < (2 + \lg n).$$

Therefore, by appealing to Lemma 1, we know that if we take  $c \geq 6\varepsilon^{-2}(2 + \lg n) \ln(2\delta^{-1})/(\lg 2(1 - 1/n))$  independent copies of this estimator we can get a  $(\varepsilon, \delta)$ -approximation to  $\mathbb{E}[X]$ . Hence with probability  $1 - O(\delta)$ , the value returned is  $(1 \pm O(\varepsilon))H_G$ .

The space bound follows because for each of the  $O(\varepsilon^{-2} \log n \log \delta^{-1})$  basic estimators we require an  $\varepsilon$  min-wise hash function using  $O(\log n \log \varepsilon^{-1})$  space [21] and a distinct element counter using

$$O((\varepsilon^{-2} \log \log n + \log n) \log \delta_1^{-1})$$

space [4] where  $\delta_1^{-1} = O(c\delta^{-1})$ . Hence, rescaling  $\varepsilon$  and  $\delta$  yields the required result.  $\square$

Our bounds are independent of the length of the stream,  $m$ , since there are only  $n^2$  distinct edges, and our algorithms are not affected by multiple copies of the same edge.

Finally, note that our algorithm is actually correct if the multi-set of edges  $E'$  arrives in any order, i.e., it is not necessary that  $(u, v)$  is followed by  $(v, w)$  for some  $w$ . Hence our algorithm also fits into the adversarial ordered graph streaming paradigm, e.g., [5, 17, 14].

<sup>2</sup>Ignoring factors of  $\log \log n$  and  $\log \varepsilon^{-1}$ .

## References

- [1] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147, 1999.
- [2] A. Arasu and G. S. Manku. Approximate counts and quantiles over sliding windows. In *ACM Symposium on Principles of Database Systems*, pages 286–296, 2004.
- [3] B. Babcock, M. Datar, and R. Motwani. Sampling from a moving window over streaming data. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 633–634, 2002.
- [4] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. In *RANDOM*, pages 1–10, 2002.
- [5] Z. Bar-Yossef, R. Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 623–632, 2002.
- [6] T. Batu, S. Dasgupta, R. Kumar, and R. Rubinfeld. The complexity of approximating the entropy. *SIAM J. Comput.*, 35(1):132–150, 2005.
- [7] L. Bhuvanagiri and S. Ganguly. Estimating entropy over data streams. In *European Symposium on Algorithms*, pages 148–159, 2006.
- [8] L. Bhuvanagiri, S. Ganguly, D. Kesh, and C. Saha. Simpler algorithm for estimating frequency moments of data streams. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 708–713, 2006.
- [9] P. Bose, E. Kranakis, P. Morin, and Y. Tang. Bounds for frequency estimation of packet streams. In *International Colloquium on Structural Information Complexity*, pages 33–42, 2003.
- [10] A. Chakrabarti, K. Do Ba, and S. Muthukrishnan. Estimating entropy and entropy norm on data streams. In *Symposium on Theoretical Aspects of Computer Science*, pages 196–205, 2006.
- [11] A. Chakrabarti, S. Khot, and X. Sun. Near-optimal lower bounds on the multi-party communication complexity of set disjointness. In *CCC*, pages 107–117, 2003.
- [12] A. Chakrabarti, G. Cormode, and A. McGregor. A near-optimal algorithm for computing the entropy of a stream. In *ACM-SIAM Symposium on Discrete Algorithms*, 2007.
- [13] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, 2005.
- [14] G. Cormode and S. Muthukrishnan. Space efficient mining of multigraph streams. In *ACM Symposium on Principles of Database Systems*, pages 271–282, 2005.
- [15] T. Cover and J. Thomas. *Elements of Information Theory*. John Wiley and Sons, Inc., 1991.
- [16] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. In *ACM-SIAM Symposium on Discrete Algorithms*, 2002.
- [17] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. On graph problems in a semi-streaming model. *Theoretical Computer Science*, 348(2-3):207–216, 2005.

- [18] P. Gibbons and S. Tirthapura. Distributed streams algorithms for sliding windows. In *ACM Symposium on Parallel Algorithms and Architectures*, 2002.
- [19] Y. Gu, A. McCallum, and D. Towsley. Detecting anomalies in network traffic using maximum entropy estimation. In *Proc. Internet Measurement Conference*, 2005.
- [20] S. Guha, A. McGregor, and S. Venkatasubramanian. Streaming and sublinear approximation of entropy and information distances. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 733–742, 2006.
- [21] P. Indyk. A small approximately min-wise independent family of hash functions. *Journal of Algorithms*, 38(1):84–90, 2001.
- [22] P. Indyk and D. P. Woodruff. Tight lower bounds for the distinct elements problem. In *IEEE Symposium on Foundations of Computer Science*, pages 283–289, 2003.
- [23] P. Indyk and D. P. Woodruff. Optimal approximations of the frequency moments of data streams. In *ACM Symposium on Theory of Computing*, pages 202–208, 2005.
- [24] A. Lall, V. Sekar, M. Ogihara, J. Xu, and H. Zhang. Data streaming algorithms for estimating entropy of network traffic. In *ACM SIGMETRICS*, 2006.
- [25] J. Misra and D. Gries. Finding repeated elements. *Science of Computer Programming*, 2:143–152, 1982.
- [26] N. Nisan. Pseudorandom generators for space-bounded computation. *Combinatorica*, 12:449–461, 1992.
- [27] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985.
- [28] A. Wagner and B. Plattner. Entropy based worm and anomaly detection in fast IP networks. In *14th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WET ICE)*, 2005.
- [29] D. P. Woodruff. Optimal space lower bounds for all frequency moments. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 167–175, 2004.
- [30] K. Xu, Z.-L. Zhang, and S. Bhattacharyya. Profiling internet backbone traffic: behavior models and applications. In *SIGCOMM*, pages 169–180, 2005.