Approximating Hit Rate Curves using Streaming Algorithms

Zachary Drudi, Nicholas J. A. Harvey, Stephen Ingram, Andrew Warfield, and Jake Wires

Coho Data Inc.
{zach,nick,stephen,andy,jake}@cohodata.com

Abstract

A hit rate curve is a function that maps cache size to the proportion of requests that can be served from the cache. (The caching policy and sequence of requests are assumed to be fixed.) Hit rate curves have been studied for decades in the operating system, database and computer architecture communities. They are useful tools for designing appropriate cache sizes, dynamically allocating memory between competing caches, and for summarizing locality properties of the request sequence. In this paper we focus on the widely-used LRU caching policy.

Computing hit rate curves is very efficient from a runtime standpoint, but existing algorithms are not efficient in their *space usage*. For a stream of m requests for n cacheable objects, all existing algorithms that provably compute the hit rate curve use space linear in n. In the context of modern storage systems, n can easily be in the billions or trillions, so the space usage of these algorithms makes them impractical.

We present the first algorithm for provably approximating hit rate curves for the LRU policy with sublinear space. Our algorithm uses $O\big(p^2 \log(n) \log^2(m)/\epsilon^2\big)$ bits of space and approximates the hit rate curve at p uniformly-spaced points to within additive error ϵ. This is not far from optimal. Any single-pass algorithm with the same guarantees must use $\Omega(p^2 + \epsilon^{-2} + \log n)$ bits of space. Furthermore, our use of additive error is necessary. Any single-pass algorithm achieving multiplicative error requires $\Omega(n)$ bits of space.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Cache analysis, hit rate curves, miss rate curves, streaming algorithms

Digital Object Identifier 10.4230/LIPIcs.APPROX-RANDOM.2015.225

1 Introduction

Caches are a fundamental concept in the design of computer systems. They are a mechanism for addressing the tradeoff between capacity and performance of different memory technologies. Even the early computers of the 1950s involved several memory technologies, and the interplay between these memories led to the foundational research on caching mechanisms in the 1960s.

The importance of caching is even more acute today. The memory hierarchy of a typical modern computer involves three levels of CPU caches, main memory, a mixture of solid-state and spinning disks for secondary storage, not to mention network file storage and web caches. The performance of nearly every modern computer depends crucially on caching mechanisms that aim to store the most appropriate data in the fastest memory.

A hit rate curve is a function that shows the performance benefit of caching as a function of the cache size. The study of hit rate curves began in the 1960s as computer designers aimed to optimize the price-performance ratio of their systems. One of the earliest discussions of hit rate curves appears in Belady's seminal 1966 paper [4] on caching and paging. Naturally,

18th Int'l Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX'2015) / 19th Int'l Workshop on Randomization and Computation (RANDOM'2015).
Editors: Naveen Garg, Klaus Jansen, Anup Rao, and José D. P. Rolim; pp. 225–241

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the question of how to efficiently compute a hit rate curve arose soon thereafter, and several algorithms were developed in the 1970s [19, 5, 25].

Although hit rate curves are not well-known within the theoretical computer science community, they feature prominently in the computer architecture and systems communities. Hit rate curves have been used in memory partitioning [27, 32, 26], garbage collection [30], program analysis [10, 31], workload phase detection [24], and cloud computing [16, 6]. CloudPhysics Inc., a datacenter performance analytics company, has recently developed a cloud-based workload analysis system using hit rate curves [28].

There are two key reasons for the resurgence of interest in hit rate curves. The first is the increasing computational penalty of cache misses. Over the past forty years, CPU speeds have improved roughly a thousandfold, whereas the latency of spinning disks has barely improved tenfold. Consequently, cache misses in storage systems are increasingly costly from a computational standpoint: there has been a substantial increase in the number of computational steps that are wasted while waiting to retrieve the data from disk. This phenomenon motivates the study of more computationally intensive cache analysis to avoid costly cache misses.

The second reason for the interest in hit rate curves is the increased amount of cache sharing, particularly due to the use of virtualization. CPU caches are shared among cores and threads; memory and solid-state storage devices are shared among processes and virtual machines. The marginal utility of increasing a process' cache allocation depends on the resulting performance improvement, which itself is determined by that process' hit rate curve. Consequently, hit rate curves are a useful ingredient for effective cache allocation.

Improved algorithms for computing hit rate curves have been developed over the years, but primarily by applied researchers rather than by theoretical algorithms researchers [5, 22, 1, 10, 32, 13, 21, 28]. There are two main goals of these improvements. The first is improved runtime, e.g. [13]: CPU cache analysis typically involves a fairly small data set but very high speeds, so performing this analysis online requires very fast runtime. The second is improved space usage, e.g. [28]: storage cache analysis typically involves extremely large data sets but at much lower speeds, so space usage is the primary concern. Typically a storage system cannot afford even a single bit of main memory for each block that appears in the secondary storage. Accordingly, most of the algorithms for computing hit rate curves in a storage context involve some sort of sampling or compression, but without rigorous guarantees.

This paper is the first to rigorously study space-efficient algorithms for computing hit rate curves. We focus on the LRU (Least Recently Used) caching policy. LRU and its variants are by far the most widely used caching policy for storage systems: the Linux, MacOS and Windows virtual memory systems all use approximations to LRU. In contrast, CPU caches tend to use simpler policies as speed is the primary concern. One policy that could potentially supplant LRU in a storage context is ARC [20], but its use has been hampered by intellectual property concerns.

1.1 Our Results

Our main result is an algorithm that computes an approximate hit rate curve for the LRU caching policy with a workload of length m involving n cacheable objects. The approximate hit rate curve achieves additive error ϵ at p uniformly-spaced points. The algorithm performs a single pass and uses $O\big(p^2 \log(n) \log^2(m)/\epsilon^2\big)$ bits of space.

We also prove lower bounds on the space. Any single-pass algorithm that achieves additive error ϵ at p uniformly-spaced points must use $\Omega(\min\left\{p^2 + \epsilon^{-2} + \log n, n\right\})$ bits of space.

Moreover, our use of additive error is necessary: any single-pass algorithm that achieves *multiplicative* error requires $\Omega(n)$ bits of space.

1.1.1 Practical Impact

It is often the case that an improved theoretical understanding of a problem leads to improved results in practice as well. That is indeed the case with the present work. Our algorithm has been incorporated into the enterprise storage system built by Coho Data, Inc. In these multi-terabyte storage systems, n and m are on the order of billions, whereas the hit rate curve as displayed to the user only requires p on the order of dozens, so the space used by our algorithm is dramatically less than previous methods. At the time of this writing, the Coho Data system is reporting live workload statistics for over a thousand virtual machines deployed in 30 customer environments, representing a level of workload characterization and insight that has not been possible in the past. A discussion of the practical aspects of this system was published in OSDI 2014 [29].

1.2 Preliminaries

In a caching system, there are n objects which can potentially be stored in the cache. Using the terminology of disk caches, we will refer to each item as a *block*, and we will identify the blocks with the integers $[n] = \{1, \ldots, n\}$. Blocks are requested at discrete points in time, which for simplicity we will always assume to be indexed by the set $[m] = \{1, \ldots, m\}$. There is a sequence of blocks $B = (b_1, \ldots, b_m)$ where $b_t \in [n]$ for each $t \in [m]$. We will refer to B as a sequence of *requests*, and say that the block b_t is requested at time t. A caching policy is a scheme for maintaining a set, called the *cache*, of at most k blocks. At each time step t, the block b_t is added to the cache, and some block may need to be removed to ensure that the cache size is at most k. If block b_t was already in the cache at time t then this request is called a *hit*, otherwise it is called a *miss*.

For a fixed sequence of requests B, a fixed size k, and a fixed caching policy, the *hit rate* is the fraction of requests that are a hit. The typical goal of a caching policy is to maximize the hit rate. If a caching policy is parameterized by the size k, then the *hit rate curve* is the function mapping k to the hit rate under this policy with cache size k.

The LRU caching policy can be defined concisely as: the cache consists of the k most recently requested distinct blocks. Whenever a miss occurs, the newly requested block must be inserted into the cache, and the least recently requested block must be removed from the cache (assuming that the cache was already full). LRU caches satisfy the *inclusion property*: for any fixed sequence of requests, the LRU cache of size k is, at each time step, a subset of the LRU cache of any size $K \geq k$. This follows directly from the definition of the LRU caching policy. Therefore a system with a cache of size K can easily simulate, or determine the hit rate, for any smaller cache. In the extreme case of $K = n$, the hit rate for *every* cache size (i.e., the entire hit rate curve) can be determined. This is the key observation that was exploited in previous algorithms [19].

1.3 The Main Idea

To understand this observation in more detail, consider a request b_t at time t, let $r < t$ be the time of the previous request for block $b_t = b_r$, and let $d = |\{b_r, \ldots, b_{t-1}\}|$ be the number of distinct blocks that were requested since time r. The request at time t would be a hit if the cache size k were at least d, and a miss for any cache size $k < d$ or if r does not

exist. Thus, to compute the hit rate curve, it suffices to determine, for every request b_t, the number of distinct blocks that were requested since the previous request for b_t. All previous hit rate curve algorithms either compute this number using a dictionary data structure or estimate it using statistical arguments [5, 22, 1, 10, 32, 13, 21, 28]. The statistical arguments unfortunately do not give worst-case guarantees.

Our main idea is very natural. We estimate the number of distinct blocks that were requested since the previous request for b_t using an F_0-estimator (distinct element estimator) from the streaming algorithms literature. That is not the end of the story, of course. A single F_0-estimator would only give the number of distinct elements for a single sequence of requests, whereas we require such an estimate for *all suffixes* of B. This leads to the fruitful idea of using a *sliding window* F_0-estimator [9, 7] which, suitably modified, can provide the required estimate for all suffixes. Unfortunately this does not solve the problem either: the algorithm cannot easily determine which suffix to use because it cannot afford to store the previous request time of all blocks – that would also require $\Omega(n)$ bits of space. Ultimately, our algorithm avoids this issue by continuously measuring the contribution from *all* suffixes, and using those contributions to update the histogram in a somewhat intricate way.

1.4 Notation

To state our results precisely, let us fix some notation. Recall that we are only interested in requests that occur at discrete points in time indexed by $t \in \{1, \ldots, m\}$. The set of requested blocks between time t' and strictly before time t is:

$$B(t', t) \;=\; \{\, b_i \,:\, i \in [m] \text{ and } t' \le i < t \,\}.$$

At time t, the most recent request for block b_t occurred at time

$$R(t) \;=\; \max\{\, x \,:\, x < t \text{ and } b_x = b_t \,\}.$$

We define $R(t) = -\infty$ if b_t was not requested before time t. At time t, the number of distinct blocks requested since the most recent request for block b is

$$D(t) \;=\; \begin{cases} |B(R(t), t)| & (\text{if } R(t) > -\infty) \\ \infty & (\text{otherwise}). \end{cases}$$

As observed above, an LRU cache of size k has a hit at time t if and only $D(t) \le k$. The *hit rate curve* is the function $C : [n] \to [0, 1]$ for which $C(k)$ is the hit rate for an LRU cache of size k. Thus

$$C(k) \;=\; |\{\, t \in [m] \,:\, D(t) \le k \,\}|/m.$$

In this paper we are concerned with computing the hit rate curve at p uniformly-spaced points, where p is a parameter. For simplicity, assume that $n = pw$, where w is an integer that denotes the "width" between the points. The *histogram* of D (with width w) is the function $H : [p] \to \mathbb{N}$ where

$$H(i) \;=\; |\{\, t \in [m] \,:\, (i-1)w < D(t) \le iw \,\}|. \tag{1}$$

The fraction of requests that are hits with a cache of size xw is $\sum_{i=1}^{x} H(i)/m$. The hit rate curve at the desired p uniformly-spaced points is

$$C(xw) \;=\; \sum_{i=1}^{x} H(i)/m \qquad \forall x \in [p].$$

1.5 Statement of results

Our main algorithmic result is:

▶ **Theorem 1.** *There is an algorithm, parameterized by p and ϵ, that performs a single pass over the input $B \in [n]^m$ and produces an approximate hit rate curve $\hat{C}$ satisfying*

$$C\big((x-1)w\big) - \epsilon \;\le\; \hat{C}(xw) \;\le\; C(xw) + \epsilon \qquad \forall x \in [p+1], \tag{2}$$

where C is the true hit rate curve for B. The algorithm uses $O\big(p^2 \log(n) \log^2(m)/\epsilon^2\big)$ bits of space.

The accuracy guarantee of (2) is unusual in that it involves approximation in the domain (horizontal error) and in the range (vertical error) of the function. The following theorem shows that both horizontal and vertical error are necessary: as $\epsilon \to n^{-1/2}$ or $w \to n^{1/2}$, we have $s \to \Omega(n)$.

▶ **Theorem 2.** *Suppose there is an algorithm A that uses s bits of space and outputs a function $\hat{C}$ satisfying*

$$C\big((x-1)w\big) - \epsilon \;\le\; \hat{C}(xw) \;\le\; C(xw) + \epsilon \qquad \forall x \in [p+1] \tag{3}$$

where $n = pw$, $p \ge 3$ and $\epsilon \le 1/5$. Then $s \ge \Omega(\min\{p^2 + 1/\epsilon^2 + \log(n), n\})$.

In particular, this result shows that other authors' claims of using "constant space" [28] cannot hold in a worst-case sense. As mentioned earlier, our use of additive vertical error in (2) is necessary. We show in Appendix C that any algorithm with multiplicative vertical error must use linear space.

2 Deterministic Algorithms for Hit Rate Curves

It is obvious from the definitions that the hit rate curve C can be computed exactly in polynomial time. It is not hard to see that it can be computed in $O(m \log n)$ time and $O(n)$ space using a balanced tree [5, 22, 1]. In this paper we are interested in approximations to C; in particular, we are only concerned with its value at p uniformly-spaced points. We begin with Algorithm 1 which computes those values. This algorithm can also be implemented in $O(m \log n)$ time and $O(n)$ space.

Algorithm 1: Algorithm for computing the hit rate curve at $p = n/w$ uniformly-spaced points.

```
Input: A sequence of requests (b_1, ..., b_m) ∈ [n]^m
Initialize the vector H ∈ ℕ^p with zeros
for t = 1, ..., m do
    If D(t) is finite then increment H[⌈D(t)/w⌉] by 1
▷ H[i] satisfies condition (1).
Output the hit rate curve values C(xw) = Σ_{i=1}^{x} H[i]/m for x ∈ [p].
```

This paper considers streaming algorithms that access the request sequence B in a single pass. Such algorithms will not be able to compute the function D from scratch, and must update H using a compact data structure that represents D. We define an abstract data type called a *suffix-structure* to encapsulate that compact representation. A suffix-structure

supports two operations, REGISTER(t, b), which records that block b was requested at time t, and GETSUFFIXF0(t), which estimates the number of distinct blocks requested since time t. A trivial and inefficient implementation of a suffix-structure is shown in Algorithm 2.

Algorithm 2: A trivial suffix-structure.

```
1  c ← 0
2  Function Register(t, b_t):
3      ▷ Assert t = c + 1
4      A[t] ← b_t
5      c ← t
6  Function GetSuffixF0(t):
7      ▷ Assert t ≤ c
8      Return |{A[t], A[t+1], ..., A[c]}|
```

Next we present Algorithm 3, our algorithm that uses a suffix-structure to approximate hit rate curves. All algorithms in this paper for computing hit rate curves are simply instantiations of Algorithm 3 that use different suffix-structures.

Algorithm 3: An algorithm for approximating the hit rate curve at p uniformly-spaced points, given an implementation $\mathcal{S}$ of a suffix-structure.

```
1  Input: A sequence of requests (b_1, ..., b_m) ∈ [n]^m
2  Initialize the vector H ∈ ℕ^p with zeros
3  ▷ For convenience, let τ_i denote (i − 1)w + 1
4  for t = 1, ..., m do
5      ▷ Receive request b_t
6      S.Register(t, b_t)
7      Let c ← ⌈t/w⌉
8      for i = 1, ..., c do
9          Let X_i(t + 1) ← S.GetSuffixF0(τ_i)
10     for i = 1, ..., c − 1 do
11         Increment H[⌈X_i(t)/w⌉] by (X_{i+1}(t+1) − X_{i+1}(t)) − (X_i(t+1) − X_i(t))
12     Increment H[⌈X_c(t)/w⌉] by 1 − (X_c(t+1) − X_c(t))
13 Output the hit rate curve approximation given by C(xw) = Σ_{i=1}^{x} H[i]/m for x ∈ [p].
```

Consider executing Algorithm 3 using a trivial suffix-structure. We now show that its output differs from that of Algorithm 1 only by the presence of horizontal error.

▶ **Lemma 3.** *Let C be the hit rate curve computed by Algorithm 1. Let $\hat{C}$ be the hit rate curve computed by Algorithm 3 using a trivial suffix-structure. Then*

$$C\big((x-1)w\big) \;\le\; \hat{C}(xw) \;\le\; C(xw) \qquad \forall x \in [p].$$

Proof Sketch. First of all, note that line 9 in Algorithm 3 satisfies

$$X_i(t) \;=\; |B(\tau_i, t)| \quad \forall i, t. \tag{4}$$

So $X_i(t+1) - X_i(t)$ is 1 if $b_t \notin B(\tau_i, t)$ and otherwise zero. From this one can infer that the increment of line 11 is 1 precisely when the previous request for b_t occured at a time in

$[\tau_i, \tau_{i+1})$; otherwise the increment is zero. It follows that the update to H in Algorithm 3 is very similar to the update in Algorithm 1. ◀

A formal proof appears in Appendix A.

3 Suffix-structures using black-box F_0-estimators

In this section we design an improved suffix-structure using ideas from streaming algorithms. The main idea is to use F_0-estimators, which are probabilistic data structures supporting two operations. The INSERT operation takes a value $x \in [n]$ and the QUERY operation reports a value v satisfying

$$|S| \;\leq\; v \;\leq\; (1+\alpha)|S|, \tag{5}$$

where S is the set of elements that were inserted so far.

The improved suffix-structure, shown in Algorithm 4, periodically creates F_0-estimators, and inserts each new block into all existing F_0-estimators. Note that it only creates a new F_0-estimator at the times $\tau_i = (i-1)w + 1$ for $i \geq 1$, because Algorithm 3 only ever calls GETSUFFIXF0(τ_i) for some i.

Algorithm 4: An approximate suffix-structure, implemented using F_0-estimators.

```
1 c ← 1
2 Function Register(t, b_t):
3     c ← ⌈t/w⌉
4     if t ≡ 1 (mod w) then
5         Create the new F_0-estimator K[c]
6     for i = 1, ..., c do
7         K[i].INSERT(b_t)
8 Function GetSuffixF0(t):
9     Return K[⌈t/w⌉].QUERY()
```

We consider only F_0-estimators which satisfy the following simple properties.

- **Consistency:** Two consecutive calls to QUERY (without any intervening insertions) return the same value.
- **Idempotency:** Reinserting an item that was previous inserted does not change the value of QUERY.
- **Monotonicity:** The values returned by QUERY do not decrease as more elements are inserted.

There exist F_0-estimators, e.g. [17], that satisfy these properties, for which (5) holds with high probability for poly(m) queries, and which use $s := \text{poly}(1/\alpha, \log(nm))$ bits of space. We do not discuss the exact space usage here as our algorithm of Section 4 will achieve even better space usage.

We now analyze Algorithm 3 when executed with the approximate suffix-structure of Algorithm 4. Our aim is to show that its output is a good approximation to the true hit rate curve. We will use F_0-estimators with accuracy parameter $\alpha = \epsilon w/2n = \epsilon/2p$.

3.1 Accuracy

The following theorem compares the outputs of Algorithm 3 when executed with a trivial suffix-structure or an approximate suffix-structure.

▶ **Theorem 4.** *Let C be the hit rate curve produced by Algorithm 3 using a trivial suffix-structure. Let $\hat{C}$ be the hit rate curve produced using an approximate suffix-structure. Then*

$$C\big((x-1)w\big) - \epsilon \;\le\; \hat{C}(xw) \;\le\; C(xw) + \epsilon \qquad \forall x \in [p+1]. \tag{6}$$

▶ **Corollary 5.** *Let C^* denote the true hit rate curve. Let $\hat{C}$ refer to the hit rate curve produced from Algorithm 3 using an approximate suffix-structure, with $2p$ points instead of p. (That is, with $w' = w/2$ instead of w). Then C^* and $\hat{C}$ satisfy*

$$C^*\big((x-1)w\big) - \epsilon \;\le\; \hat{C}(xw) \;\le\; C^*(xw) + \epsilon \qquad \forall x \in [p+1].$$

This is the condition guaranteed by Theorem 1.

Proof. Combining (6), Lemma 3 and the definition of α yields

$$C^*\big((x-2)w'\big) - \epsilon \;\le\; \hat{C}(xw') \;\le\; C^*(xw') + \epsilon \qquad \forall x \in [2p+1].$$

Substituting $w/2$ for w' completes the proof. ◀

3.2 Space usage

After calling $\mathcal{S}.\textsc{Register}(t, b_t)$ m times, the approximate suffix-structure will have created $\lceil m/w \rceil$ F_0-estimators, each of which uses s bits of space, so the total space usage is $O(ms/w)$ bits. This does not quite meet our goal of $\text{poly}(p, 1/\epsilon, \log(nm))$ bits. The algorithm can be improved to use only $O(ps/\epsilon)$ bits (while still using the F_0-estimators as a black box) but we do not discuss that improvement here, as the algorithm of Section 4 achieves even better space usage. Details of this improvement may be found in [11].

Proof of Theorem 4. Let H and X_i refer to the quantities using the trivial suffix-structure, and let $\hat{H}$ and $\hat{X}_i$ refer to the corresponding quantities using the approximate suffix-structure. We require the following proposition, which is proven in Appendix B.

▶ **Proposition 6.** *For any times $a \le b$ and any index i, we have $X_i(a) - X_{i+1}(a) \ge X_i(b) - X_{i+1}(b)$.*

The histogram H and the hit rate curve C are obtained by summing contributions from each consecutive pair of cardinality values X_i and X_{i+1}. The same is true of $\hat{H}$ and $\hat{C}$, using instead the pair $\hat{X}_i$ and $\hat{X}_{i+1}$. So, to prove (6), we will show that the contribution from the pair $\hat{X}_i$ and $\hat{X}_{i+1}$ to $\hat{C}$ approximately equals the contribution from the pair X_i and X_{i+1} to C.

3.3 Contribution to C

Fix any $x \in [p]$ and recall that $C(xw) = \sum_{j=1}^{x} H[j]/m$. By considering lines 11 and 12 of Algorithm 3 we see that the pair X_i and X_{i+1} can only contribute to $C(xw)$ while $\lceil X_i(t)/w \rceil \le x$. So, let T_i be the first time t at which $\lceil X_i(t)/w \rceil > x$, i.e.,

$$T_i \;=\; \min\{\, t \,:\, X_i(t) > xw \,\}.$$

At all times $t \geq T_i$, the pair X_i and X_{i+1} do not contribute to $C(x)$. The contribution to $m \cdot C(xw)$ from the pair X_i and X_{i+1} is

$$\begin{cases} X_{i+1}(t+1) - X_{i+1}(t) - X_i(t+1) + X_i(t) & (\text{for } t \in \{\tau_{i+1}, \ldots, T_i - 1\}) \\ 1 - X_i(t+1) + X_i(t) & (\text{for } t \in \{\tau_i, \ldots, \tau_{i+1} - 1\}). \end{cases}$$

Summing up, the total contribution is

$$\begin{aligned} &\sum_{\tau_i \leq t < \tau_{i+1}-1} \big(1 - X_i(t+1) + X_i(t)\big) + \\ &\sum_{\tau_{i+1} \leq t < T_i} \big(X_{i+1}(t+1) - X_{i+1}(t) - X_i(t+1) + X_i(t)\big) \\ &\quad = \; w - X_i(\tau_{i+1}) + X_i(\tau_i) + X_i(\tau_{i+1}) - X_{i+1}(\tau_{i+1}) + X_{i+1}(T_i) - X_i(T_i) \\ &\quad = \; w + X_{i+1}(T_i) - X_i(T_i). \end{aligned} \tag{7}$$

3.4 Contribution to $\hat{C}$

Similarly, let $\hat{T}_i = \min\{\, t : \hat{X}_i(t) > xw \,\}$. Then at all times $t \geq \hat{T}_i$, the pair $\hat{X}_i$ and $\hat{X}_{i+1}$ do not contribute to $\hat{C}(x)$. (This assertion uses the Monotonicity property.) Summing up as before, the total contribution of the pair $\hat{X}_i$ and $\hat{X}_{i+1}$ to $m \cdot \hat{C}(xw)$ is

$$w + \hat{X}_{i+1}(\hat{T}_i) - \hat{X}_i(\hat{T}_i). \tag{8}$$

3.4.1 Upper bound on contribution to $\hat{C}(xw)$

The difference between the contribution of $\hat{X}_i$ and $\hat{X}_{i+1}$ to $m \cdot \hat{C}(xw)$ and the contribution of X_i and X_{i+1} to $m \cdot C(xw)$ is the difference between (8) and (7), namely

$$\hat{X}_{i+1}(\hat{T}_i) - \hat{X}_i(\hat{T}_i) - X_{i+1}(T_i) + X_i(T_i). \tag{9}$$

We now upper bound this quantity. First note that $\hat{T}_i \leq T_i$, by (5). Then Proposition 6 shows that (9) is at most

$$\begin{aligned} &\hat{X}_{i+1}(\hat{T}_i) - \hat{X}_i(\hat{T}_i) - X_{i+1}(\hat{T}_i) + X_i(\hat{T}_i) \\ &\quad \leq \; \alpha X_{i+1}(\hat{T}_i) \quad (\text{by (5)}) \\ &\quad \leq \; \alpha X_i(\hat{T}_i) \quad (\text{by definition of } X_i \text{ and } X_{i+1}) \\ &\quad \leq \; \alpha(xw + 1) \quad (\text{since } \hat{T}_i \leq T_i \text{ and by definition of } T_i). \end{aligned} \tag{10}$$

3.4.2 Lower bound on contribution to $\hat{C}(xw)$

For the lower bound, we must consider the contribution of X_i and X_{i+1} to $C((x-1)w)$. Define $T_i' = \min\{\, t : X_i(t) > (x-1)w \,\}$. Arguing as before, we get that the total contribution of this pair to $m \cdot C((x-1)w)$ is

$$w + X_{i+1}(T_i') - X_i(T_i'). \tag{11}$$

The difference between (8) and (11) is

$$\hat{X}_{i+1}(\hat{T}_i) - \hat{X}_i(\hat{T}_i) - X_{i+1}(T_i') + X_i(T_i'). \tag{12}$$

We now claim that $T'_i < \hat{T}_i$. By definition of T'_i, we have $X_i(T'_i) = (x-1)w+1$. By definition of α, we have $\alpha n = \epsilon w < w$. So, by (5),

$$\begin{aligned}\hat{X}_i(T'_i) &\leq (1+\alpha)X_i(T'_i) \leq (1+\alpha)\big((x-1)w+1\big)\\ &\leq (x-1)w+1+\alpha n < xw+1 \leq \hat{X}_i(\hat{T}_i).\end{aligned}$$

By the Monotonicity property, the claim is proven. So we may use Proposition 6 to show that (12) is at least

$$\begin{aligned}&\hat{X}_{i+1}(\hat{T}_i) - \hat{X}_i(\hat{T}_i) - X_{i+1}(\hat{T}_i) + X_i(\hat{T}_i)\\ &\geq -\alpha X_i(\hat{T}_i) \quad \text{(by (5))}\\ &\geq -\alpha(xw+1) \quad \text{(since } \hat{T}_i \leq T_i \text{ and by definition of } T_i\text{).}\end{aligned} \tag{13}$$

3.5 Proof of (6)

We now combine our previous observations to establish (6). Recall that $c = \lceil m/w \rceil$ is the total number of F_0-estimators. Summing (10) over all i, we obtain that

$$m\hat{C}(xw) \leq mC(xw) + \alpha c(xw+1) \leq mC(xw) + 2\alpha mx.$$

This proves the second inequality of (6). The first inequality of (6) follows analogously from (13). ◀

4 Suffix-structures using a timestamped F_0-estimator

As mentioned in Section 1, the idea of *sliding window estimators* [9] is very relevant for estimating properties of the suffixes of a stream. One of the main ideas is to modify known streaming estimators by incorporating *timestamps*. By restricting the estimator's data structure to entries with sufficiently large timestamps, one can construct an estimator for the desired suffix of the stream. Let us now discuss that idea for the special case of F_0-estimators [9, §7.5].

Many F_0-estimators rely on a $\{0,1\}$-matrix M that is continously updated while processing the stream [2, 14, 3, 17]. Each item b in the stream is hashed to a binary string σ, and then M is updated based on $\mathrm{lsb}(\sigma)$, the number of trailing zeros in σ. We will call such a matrix M a *bitmatrix*.

The simplest F_0-estimator [2] uses a single hash function h, and the matrix M has a single column. At any point during stream processing, M_j is set to 1 if a block b was observed with $\mathrm{lsb}(h(b)) \geq j$. After the stream is processed, the algorithm outputs 2^{j^*}, where j^* is the greatest index of a non-zero row. This gives only a $O(1)$ approximation. Other algorithms refine this estimate by using additional columns and another hash function g, which determines which column to update. The estimate could be, for example, a function of the average of the lowest non-zero value in each column [12, 14], or the number of non-zero elements below a certain row (Algorithm 3 in [3]). All of these algorithms can boost their success probability by taking medians of independent, parallel instantiations.

Suppose that Algorithm 4 uses an F_0-estimator of this type, and that all instantiations of that estimator use the same hash functions h and g. Let M^j denote the bitmatrix corresponding to the j^{th} F_0-estimator. For any $j \leq j'$, M^j has undergone all of the updates that $M^{j'}$ has, so it follows that $M^j \geq M^{j'}$ in an entrywise sense. This observation leads to following idea: instead of storing the bitmatrices for each estimator separately, we can store a single unified matrix from which all bitmatrices can be computed.

Algorithm 5: An implementation of a suffix-structure based on a timestamped F_0-estimator. We consider a F_0-estimator that is based on a bitmatrix as described above. The algorithm $\mathcal{A}$ specifies how the bitmatrix is updated on an INSERT operation, and how to produce the estimate for the QUERY operation. The set $\mathcal{Q}$ has many independent copies of the hash functions and the resulting table. We need only $|\mathcal{Q}| = O(\log(1/\delta))$ with $\delta = m^{-3}$.

Data: A collection $\mathcal{Q}$ of pairs $(Q, \mathcal{H})$, where Q is a matrix, $\mathcal{H}$ is a set of hash functions
An algorithm $\mathcal{A}$ for estimating F_0 from a bitmatrix

$c \leftarrow 1$
Function `Register`(t, b_t)**:**
  $c \leftarrow \lceil t/w \rceil$
  for $(Q, \mathcal{H}) \in \mathcal{Q}$ **do**
    Update Q using b_t according to $\mathcal{A}$
Function `GetSuffixF0`(t')**:**
  for $Q \in \mathcal{Q}$ **do**
    Let $r = \lceil t'/w \rceil$
    Define the bitmatrix M^r by $M^r_{i,j} = \begin{cases} 1 & \text{if } Q_{i,j} - r \geq 0 \\ 0 & \text{otherwise} \end{cases}$
    Feed M^r into algorithm $\mathcal{A}$ to obtain estimate R_Q
  Return the median of estimates R_Q

4.1 Analysis

In order to analyze Algorithm 5, we must specify $\mathcal{A}$, a concrete F_0-estimator. We will use Algorithm 2 from the paper of Bar-Yossef et al. [3].[1] In this algorithm, each matrix Q has $\log(n)$ rows, and $k = O(1/\alpha^2)$ columns. Each collection $\mathcal{H}$ consists of k t-wise independent hash functions, where t is $O\big(\log(1/\alpha)\big)$. To update Q, for $j \in [k]$, we set $Q_{i,j} = c$ if $\mathrm{lsb}(h_j(b_t)) \geq i$. Given the bitmatrix M, the F_0-estimator can produce its estimate.

▶ **Proposition 7.** *The space used by Algorithm 5 is* $O\big(p^2 \log(n) \log(m) \log(1/\delta)/\epsilon^2\big)$ *bits.*

Proof. Each Q has $O(1/\alpha^2)$ columns, $\log n$ rows, and each cell requires $\log m$ bits space. Thus Q requires $O\big(\log(n)\log(m)/\alpha^2\big)$ bits of space. Each collection $\mathcal{H}$ requires only $O\big(\log^2(1/\alpha)\log n\big)$ bits of space, which is negligible. We have $\log(1/\delta)$ such pairs $(Q, \mathcal{H})$. Thus the total space requirement is $O\big(\alpha^{-2}\log(n)\log(m)\log(1/\delta)\big)$. Substituting $\alpha = \epsilon/p$ completes the proof. ◀

▶ **Theorem 8.** *Algorithm 3 using Algorithm 5 as its suffix-structure satisfies* (2)*.*

Proof. Let $X_i(t)$ be the result of GETSUFFIXF0(τ_i) at time t, which is an estimate of $|B(\tau_i, t)|$. It suffices to show that $X_i(t) \in [1, 1+\alpha] \cdot |B(\tau_i, t)|$ with high probability, in which case the argument of Theorem 1 applies.

[1] It is natural to wonder why we do not use the optimal algorithm of Kane et al. [17]. The reason is that the Kane et al. algorithm is an enhancement of the Bar-Yossef et al. algorithm that manages to save some additional space. In contrast, our algorithm will consume extra space by adding timestamps, which ruins the space-saving enhancements of Kane et al.

For any i, t, by taking the median of $\log(1/\delta)$ estimates of $\mathcal{A}$, we have $\Pr[|X_i(t) - |B(\tau_i, t)|| > \alpha|B(\tau_i, t)|] \leq \delta$. At time t, the algorithm computes estimates for t/w F_0-estimators, and thus $O(m^2)$ estimates are taken in total. By a union bound, the probability that any estimate is poor is $\leq \delta m^2$. ◀

Combining Proposition 7 and Theorem 8 and taking $\delta = 1/m^3$ proves Theorem 1.

5 Discussion

In this paper we have presented the first single-pass, sublinear space algorithm with provable guarantees for estimating hit rate curves for the LRU caching policy. The space usage is $O(p^2 \log(n) \log^2(m)/\epsilon^2)$ bits. This space usage is not far from optimal, due to our $\Omega(p^2 + \epsilon^{-2} + \log(n))$ lower bound. A practical implementation of this algorithm has been deployed in an enterprise storage system [29].

As this is the first theoretical paper on hit rate curve computation, it suggests several directions for further algorithmic research.

- It would be nice to improve either our upper bound or lower bound on the space usage. Our suspicion is that the lower bounds can be improved.
- In practice the runtime of our algorithm is very good [29], but we have not studied the runtime from a theoretical perspective. In particular, optimizing the runtime of Algorithm 5 seems quite interesting.
- Instead of approximating the hit rate curve at p *uniformly*-space points, it is natural to wonder whether the p points can be adaptively chosen during the algorithm.
- It would be interesting to extend our techniques beyond just the LRU caching policy. The algorithm of Mattson et al. [19] works for all policies that satisfy the inclusion property – is there a single-pass streaming algorithm for all such policies?
- A key operation in our algorithm is to take the difference of F_0-estimators. (In fact, we estimate $|A \setminus B|$ where $B \subseteq A$.) There are F_0-estimators that have been explicitly designed for this purpose, e.g., [15]. It would be interesting to study whether such specialized F_0-estimators can improve our algorithm, either theoretically or practically.

Acknowledgements. We thank David Woodruff for a helpful discussion about distinct element estimators. We thank Andrew McGregor and the anonymous reviewers for helpful comments.

References

1 George S. Almási, Călin Caşcaval, and David A. Padua. Calculating stack distances efficiently. In *Proceedings of the 2002 workshop on memory system performance (MSP'02)*, pages 37–43, 2002.

2 Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 20–29. ACM, 1996.

3 Ziv Bar-Yossef, TS Jayram, Ravi Kumar, D Sivakumar, and Luca Trevisan. Counting distinct elements in a data stream. In *Randomization and Approximation Techniques in Computer Science*, pages 1–10. Springer, 2002.

4 L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

5 Brian T Bennett and Vincent J. Kruskal. LRU stack processing. *IBM Journal of Research and Development*, 19(4):353–357, 1975.

6 Hjortur Bjornsson, Gregory Chockler, Trausti Saemundsson, and Ymir Vigfusson. Dynamic performance profiling of cloud caches. In *Proceedings of the 4th annual Symposium on Cloud Computing (SoCC)*. ACM, 2013.

7 Vladimir Braverman and Rafail Ostrovsky. Smooth histograms for sliding windows. In *Foundations of Computer Science, 2007. Proceedings. 48th Annual IEEE Symposium on*, pages 283–293. IEEE, 2007.

8 Amit Chakrabarti and Oded Regev. An optimal lower bound on the communication complexity of gap-hamming-distance. *SIAM Journal on Computing*, 41(5):1299–1317, 2012.

9 Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. *SIAM J. Comput.*, 31(6):1794–1813, 2002.

10 Chen Ding and Yutao Zhong. Predicting whole-program locality through reuse distance analysis. In *PLDI*, pages 245–257. ACM, 2003.

11 Zachary Drudi. A streaming algorithms approach to approximating hit rate curves. Master's thesis, University of British Columbia, 2014.

12 Marianne Durand and Philippe Flajolet. Loglog counting of large cardinalities. In *Algorithms-ESA 2003*, pages 605–617. Springer, 2003.

13 David Eklov and Erik Hagersten. StatStack: Efficient modeling of LRU caches. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 55–65. IEEE, 2010.

14 Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. *DMTCS Proceedings*, 0(1), 2008.

15 Sumit Ganguly, Minos Garofalakis, and Rajeev Rastogi. Tracking set-expression cardinalities over continuous update streams. *The VLDB Journal*, 13(4):354–369, 2004.

16 Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Geiger: Monitoring the buffer cache in a virtual machine environment. In *ASPLOS*, pages 14–24. ACM, 2006.

17 Daniel M Kane, Jelani Nelson, and David P Woodruff. An optimal algorithm for the distinct elements problem. In *Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 41–52. ACM, 2010.

18 E Kushilevitz and N Nisan. Communication complexity, 1997.

19 Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

20 Nimrod Megiddo and Dharmendra S Modha. ARC: A self-tuning, low overhead replacement cache. In *FAST*, volume 3, pages 115–130, 2003.

21 Qingpeng Niu, James Dinan, Qingda Lu, and P Sadayappan. Parda: A fast parallel reuse distance analysis algorithm. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 1284–1294. IEEE, 2012.

22 Frank Olken. Efficient methods for calculating the success function of fixed space replacement policies. Master's thesis, University of California, Berkeley, 1981.

23 Alexander A. Razborov. On the distributional complexity of disjointness. *Theoretical Computer Science*, 106(2):385–390, 1992.

24 Xipeng Shen, Yutao Zhong, and Chen Ding. Locality phase prediction. In *ASPLOS*, pages 165–176. ACM, 2004.

25 Alan Jay Smith. Two methods for the efficient analysis of memory address trace data. *Software Engineering, IEEE Transactions on*, 3(1):94–101, 1977.

26 Gokul Soundararajan, Daniel Lupei, Saeed Ghanbari, Adrian Daniel Popescu, Jin Chen, and Cristiana Amza. Dynamic resource allocation for database servers running on virtual storage. In *FAST*. USENIX, 2009.

27 Harold S Stone, John Turek, and Joel L. Wolf. Optimal partitioning of cache memory. *Computers, IEEE Transactions on*, 41(9):1054–1068, 1992.

28 Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient MRC construction with SHARDS. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*, pages 95–110. USENIX, 2015.

29 Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas J. A. Harvey, and Andrew Warfield. Characterizing storage workloads with counter stacks. In *OSDI*, 2014.

30 Ting Yang, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. CRAMM: Virtual memory support for garbage-collected applications. In *OSDI*, pages 103–116. ACM, 2006.

31 Yutao Zhong, Maksim Orlovich, Xipeng Shen, and Chen Ding. Array regrouping and structure splitting using whole-program reference affinity. In *PLDI*, pages 255–266. ACM, 2004.

32 Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. Dynamic tracking of page miss ratio curve for memory management. In *ASPLOS*, pages 177–188. ACM, 2004.

A Proofs from Section 2

Proof.]Proof of Lemma 3] Let H and $\hat{H}$ respectively denote the histograms computed by Algorithms 1 and 3. Note that $b_t \notin B(t',t)$ for $t' > R(t)$ but $b_t \in B(t',t)$ for $t' \le R(t)$. Because of (4), we have

$$X_i(t+1) - X_i(t) \;=\; \begin{cases} 1 & (\text{if } R(t) < \tau_i \le t) \\ 0 & (\text{if } 1 \le \tau_i \le R(t)) \end{cases}.$$

It follows that the increment in line 11 equals 1 if $\tau_i \le R(t) < \tau_{i+1}$ and otherwise it equals zero. Similarly, the increment in line 12 equals 1 if $\tau_c \le R(t)$. At most one of these conditions can hold, so for each value of t, Algorithm 3 increments at most one entry of $\hat{H}$. Specifically, if $R(t)$ is finite then the algorithm increments $\hat{H}[\lceil X_{i^*}(t)/w \rceil]$ where $i^* = \lceil R(t)/w \rceil$.

When $R(t)$ is finite, we have $\tau_{i^*} < R(t) \le \tau_{i^*+1}$. Since $X_{i^*}(t) = |B(\tau_{i^*}, t)|$ and $D(t) = |B(R(t), t)|$, we derive $X_{i^*+1}(t) \le D(t) \le X_{i^*}(t)$. However,

$$\begin{aligned} X_{i^*}(t) - X_{i^*+1}(t) \;&=\; |B(\tau_{i^*}, t)| - |B(\tau_{i^*+1}, t)| \;=\; |B(\tau_{i^*}, t) \setminus B(\tau_{i^*+1}, t)| \\ &\le\; |B(\tau_{i^*}, \tau_{i^*+1})| \;\le\; w. \end{aligned}$$

So

$$\left\lceil \frac{X_{i^*}(t)}{w} \right\rceil - 1 \;\le\; \left\lceil \frac{D(t)}{w} \right\rceil \;\le\; \left\lceil \frac{X_{i^*}(t)}{w} \right\rceil.$$

Algorithm 1 increments $H[\lceil D(t)/w \rceil]$, whereas Algorithm 3 increments $\hat{H}[\lceil X_{i^*}(t)/w \rceil]$. So

$$\underbrace{\sum_{i=1}^{x} \hat{H}[i]}_{\hat{C}(xw)} \;\le\; \underbrace{\sum_{i=1}^{x} H[i]}_{C(xw)} \;\le\; \underbrace{\sum_{i=1}^{x+1} \hat{H}[i]}_{\hat{C}\left((x+1)w\right)}.$$

Rearranging this yields the desired inequality. ◀

B Proofs from Section 3

Proposition 6. *For any times $a \le b$ and any index i, we have $X_i(a) - X_{i+1}(a) \ge X_i(b) - X_{i+1}(b)$.*

Proof. Recall that $X_i(t) = |B(\tau_i, t)|$. As $\tau_i < \tau_{i+1}$, we get $X_i(t) - X_{i+1}(t) = |B(\tau_i, \tau_{i+1}) \setminus B(\tau_{i+1}, t)|$. Thus

$$\begin{aligned} & X_i(a) - X_{i+1}(a) - (X_i(b) - X_{i+1}(b)) \\ & \quad = |B(\tau_i, \tau_{i+1}) \setminus B(\tau_{i+1}, a)| - |B(\tau_i, \tau_{i+1}) \setminus B(\tau_{i+1}, b)| \\ & \quad \geq 0, \end{aligned}$$

as $B(\tau_{i+1}, a) \subset B(\tau_{i+1}, b)$. ◀

C Lower Bounds

In this section we prove lower bounds on the space needed by one-pass algorithms to compute approximate hit rate curves. As is typical with streaming algorithms, our lower bounds are based on communication complexity [18].

C.1 Lower Bounds for Hit Rate Curve Estimation

Our lower bounds are based on reductions from the Gap Hamming Distance (GHD) problem. In $\mathrm{GHD}_{q,t,g}$, Alice and Bob are respectively given vectors $x, y \in \{0,1\}^q$. They are required to determine whether the Hamming distance between x and y, denoted $d(x,y)$, is less than $t-g$ or greater than $t+g$, outputting 0 or 1, respectively. The following optimal lower bound for GHD is known.

▶ **Theorem 9** (Chakrabarti-Regev [8]). *Any protocol that solves $GHD_{q,q/2,g}$ with probability $\geq 2/3$ communicates $\Omega(\min\{q, q^2/g^2\})$ bits.*

Proof of Theorem 2. Let $c = 10$ and $q = n/(c+2)$. Consider an instance of $\mathrm{GHD}_{q,q/2,g}$, where g will be specified later. Alice has $x \in \{0,1\}^q$ and Bob has $y \in \{0,1\}^q$. Let us say that $\mathrm{GHD}(x,y) = 0$ if $d(x,y) < q/2 - g$, and $\mathrm{GHD}(x,y) = 1$ if $d(x,y) > q/2 + g$.

Alice and Bob produce an input stream to the algorithm A as follows. Each stream element is a member of $[n]$. The elements in $[cq] \subset [n]$ are called "type-1", and those in $[n] \setminus [cq]$ are called "type-2". Alice first provides to A the type-1 elements, then certain type-2 elements. Specifically, she provides the sequence $(1, 2, \ldots, cq)$, then provides $cq + j$ if $x_j = 1$ and $(c+1)q + j$ if $x_j = 0$, for $j \in [q]$. She then communicates A's state to Bob, which requires s bits of communication. Bob first provides to A certain type-2 elements, then the type-1 elements. Specifically, he provides $cq + j$ if $y_j = 0$ and $(c+1)q + j$ if $y_j = 1$, then provides the sequence $(1, 2, \ldots, cq)$. The total length of the stream provided to A is exactly $m = 2(c+1)q$.

Observation 1: The number of type-2 elements that occur in the stream is exactly $2q$. The number that occur twice is exactly $d(x,y)$. Hence, the number of *distinct* type-2 elements that occur in the stream is exactly $2q - d(x,y)$.

Observation 2: Every type-1 element appears exactly twice in the stream. The number of distinct elements that occur between those two occurrences is exactly $cq + 2q - d(x,y)$. Depending on whether $\mathrm{GHD}(x,y)$ is 0 or 1, we have

$$\begin{array}{lrcl} \text{If } \mathrm{GHD}(x,y) = 0: & cq + 2q - d(x,y) & > & (c + 3/2)q + g \;=:\; H \\ \text{If } \mathrm{GHD}(x,y) = 1: & cq + 2q - d(x,y) & < & (c + 3/2)q - g \;=:\; L \end{array}$$

The only other requests that possibly contribute to C are Bob's type-2 elements, of which there are only q. (Alice's inputs contribute nothing to C.)

Case 1: $w \le \epsilon n$. Equivalently, $p \ge 1/\epsilon$. In this case we take $g = w$. Let $\beta = (c+3/2)q/w$. Then

$$L = (\beta - 1)w \qquad \text{and} \qquad \beta w < H. \tag{14}$$

Suppose that $\hat{C}$ satisfies (3). If $\mathrm{GHD}(x, y) = 1$ then

$$\begin{aligned}
\hat{C}(\beta w) &\ge C\big((\beta-1)w\big) - \epsilon \qquad \text{(by (3))} \\
&= C(L) - \epsilon \qquad \text{(by (14))} \\
&\ge \frac{cq}{m} - \epsilon \qquad \text{(by Observation 2)} \\
&\ge \frac{c}{2(c+1)} - \frac{1}{5} > \frac{1}{4} \qquad (c \ge 10 \text{ and } \epsilon \le 1/5).
\end{aligned}$$

On the other hand, if $\mathrm{GHD}(x, y) = 0$ then

$$\begin{aligned}
\hat{C}(\beta w) &\le C(\beta w) + \epsilon \qquad \text{(by (3))} \\
&\le C(H) + \epsilon \qquad \text{(by (14))} \\
&\le \frac{q}{m} + \epsilon \qquad \text{(by Observation 2)} \\
&\le \frac{1}{2(c+1)} + \frac{1}{5} < \frac{1}{4} \qquad (c \ge 10 \text{ and } \epsilon \le 1/5).
\end{aligned}$$

Therefore Alice and Bob can distinguish whether $\mathrm{GHD}(x, y)$ is 0 or 1. The number of bits of space necessary is $\Omega(\min\{q, q^2/g^2\}) = \Omega(\min\{n, p^2\})$.

Case 2: $w > \epsilon n$. Equivalently, $1/\epsilon > p$. In this case we take $g = 22\epsilon q$. Set $\beta = 1 + \lceil p/(c+2) \rceil$. Then

$$\begin{aligned}
(\beta - 1)w &\ge \frac{p}{c+2} w = q \\
\beta w &< (2 + \tfrac{p}{c+2})w = (\tfrac{2}{p} + \tfrac{1}{c+2})n \le (\tfrac{2}{3} + \tfrac{1}{c+2})n < \tfrac{c}{c+2} n = cq.
\end{aligned}$$

By observation 2, the type-1 elements do not contribute to $C(cq)$. So consider any type-2 element. If it appears twice, then the number of distinct elements between those two appearances is at most q. By observation 1, the number of type-2 elements that appear twice is exactly $d(x, y)$. It follows that $C(cq) = C(q) = d(x, y)/m$. So, if $\hat{C}$ satisfies (3), we have

$$\frac{d(x,y)}{m} - \epsilon \le C(q) - \epsilon \le \hat{C}(\beta w) \le C(cq) + \epsilon = \frac{d(x,y)}{m} + \epsilon.$$

Thus

$$|m \cdot \hat{C}(\beta w) - d(x, y)| \le m\epsilon = 2(c+1)q\epsilon < g.$$

It follows that Alice and Bob can distinguish whether $\mathrm{GHD}(x, y)$ is 0 or 1. The number of bits of space necessary is $\Omega(\min\{q, q^2/g^2\}) = \Omega(\min\{n, 1/\epsilon^2\})$.

The lower bound of $\Omega(\log n)$ is left as an easy exercise. ◀

C.2 Impossibility of Multiplicative Error

Lastly, we show that any algorithm with multiplicative vertical error must use linear space, even if it also has additive horizontal error.

▶ **Theorem 10.** *Let $n = pw$ where $w \geq 1$ is arbitrary. Let $\epsilon \in [0, 1)$ be arbitrary. Suppose there is a single-pass algorithm A that uses s bits of space and outputs a function $\hat{C}$ satisfying*

$$(1-\epsilon)\cdot C\big((i-1)w\big) \;\leq\; \hat{C}(iw) \;\leq\; (1+\epsilon)\cdot C(iw) \qquad \forall i \in [p+1]. \tag{15}$$

Then $s = \Omega(n)$.

Proof. The disjointness problem DISJ : $\{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ is defined to be $\mathrm{DISJ}(x, y) = \prod_i (1 - x_i y_i)$. Razborov's lower bound [23, 18] implies that, even under the promise that $\sum_i x_i = \sum_i y_i = (n+1)/4$ and $\sum_i x_i y_i \in \{0, 1\}$, any randomized communication protocol that can decide DISJ must use $\Omega(n)$ bits of communication.

C.2.1 Reduction

Alice and Bob produce an input stream to the algorithm A as follows. Each stream element is a member of $[n]$. Alice provides to A the set $\{\, i \in [n] : x_i = 1 \,\}$ in any order. She then communicates A's state to Bob, which requires s bits. Bob provides to A the set $\{\, i \in [n] : y_i = 1 \,\}$ in any order. The total length of the stream provided to A is exactly $m = (n+1)/2$.

If $\mathrm{DISJ}(x, y) = 1$ then every stream element is distinct, so $C(n) = 0$. On the other hand, if $\mathrm{DISJ}(x, y) = 0$ then the promise ensures that $C(n) = 1/m$. Let us apply (15) with $i = p + 1$, and recall from the definition of C that $C(pw) = C\big((p+1)w\big)$. It follows that Alice and Bob can decide $\mathrm{DISJ}(x, y)$, so Razborov's result implies that $s = \Omega(n)$. ◀