Reductions in Streaming Algorithms, with an Application to Counting Triangles in Graphs

Ziv Bar-Yossef* Ravi Kumar† D. Sivakumar‡

Abstract

We introduce *reductions* in the streaming model as a tool in the design of streaming algorithms. We develop the concept of *list-efficient* streaming algorithms that are essential to the design of efficient streaming algorithms through reductions.

Our results include a suite of list-efficient streaming algorithms for basic statistical primitives. Using the reduction paradigm along with these tools, we design streaming algorithms for approximately counting the number of triangles in a graph presented as a stream.

A specific highlight of our work is the first algorithm for the number of distinct elements in a data stream that achieves arbitrary approximation factors. (Independently, Trevisan [Tre01] has solved this problem via a different approach; our algorithm has the advantage of being list-efficient.)

1 Introduction

In the context of computing with massive data sets, algorithms designed to work in the streaming model [HRR99, AMS99, FKSV99] are gaining popularity, both for their theoretical significance and for their usefulness in practice. In this model, data arrives in a stream, one item at a time, and algorithms have fairly stringent requirements to be considered efficient: they are required to use very little space and per-item processing time (both typically polylogarithmic in the length of the data stream). Also, in many cases, streaming algorithms are required to work correctly even if the input is an array that is presented in an arbitrary order. Nevertheless, in recent years, efficient randomized algorithms have been designed in the streaming model for several fundamental problems, including the approximate computation of frequency moments [AMS99], L_p distances between vectors [AMS99, FKSV99, FS00], histograms [GKS01], wavelet transforms [GKMS01], and others. See [Bab01] for an extensive bibliography on streaming algorithms. Insights from streaming algorithms have also led to derandomization of several approximation algorithms [EIO01, Siv01].

Summary. In this work, we promote *reductions* in the streaming model as a basic tool in the design of efficient streaming algorithms. We begin by underscoring the subtleties involved in designing efficient streaming algorithms via reductions between computational problems. Our analysis leads to the concept of *list-efficient* streaming algorithms that, in conjunction with reductions, are ideally suited for the design of efficient streaming algorithms. Our first technical contributions include the design of list-efficient streaming algorithms for some basic primitives, most notably to the problem of (approximately) computing the number of distinct elements in a data stream. This algorithm improves an algorithm of Alon, Matias, and Szegedy [AMS99]. Using the reduction methodology together with these tools, we design efficient streaming algorithms for approximately counting the number of triangles in a graph presented as a stream. Our triangle algorithms seem to be the first natural graph algorithms in the streaming model.

Stream reductions and list-efficient algorithms. Reductions between computational problems are fundamental tools of complexity theory and algorithm design. Owing to the stringent requirements of efficiency in the streaming model, designing efficient algorithms via reductions turns out to be a rather delicate matter. For example, consider a hypothetical streaming reduction R from problem A to problem B that works as follows: upon reading each data item in an instance (a stream) of problem A, the reduction R outputs a polynomially long sequence of data items to produce an instance (a stream) of problem B. Here, an efficient algorithm for problem B does *not* necessarily translate into an efficient algorithm for problem A (the processing time per data item with respect to the instance of A could now be polynomial, as opposed to polylogarithmic). What combinations of reductions R, together with efficient algorithms for B, then, would give us efficient algorithms for A? We address this issue in Section 3.

*Computer Science Division, University of California at Berkeley, Berkeley, CA 94720, `zivi@cs.berkeley.edu`. Part of this work was done while the author was visiting IBM Almaden Research Center. Supported by NSF Grant CCR-9820897.

†IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120, `ravi@almaden.ibm.com`

‡IBM Almaden Research Center, 650 Harry Road,San Jose, CA 95120, `siva@almaden.ibm.com`

We pinpoint a class of streaming algorithms, which we call *list-efficient*, as appropriate for use in conjunction with streaming reductions. Intuitively, the idea is the following. Since the reduction itself is a streaming algorithm, each sequence of items in the instance of problem B has a succinct representation — the configuration of the reduction algorithm R plus the item of data read from the instance of A. Thus an algorithm for B that can process an entire list of items — *given only its succinct representation* — is a good candidate to compose with the reduction R. In typical applications, we do not expect the succinct representation of a list to be as prosaic as the "internal configuration of reduction algorithm R". Rather, it is more likely to possess more special structure, for example, an interval of integers. This makes the task of constructing list-efficient algorithms for B a natural problem within the realm of streaming algorithm design. Indeed, the notion of list-efficient algorithms includes (and was motivated by) the *range-summable hash functions* of [FKSV99].

List-efficient primitives. We then turn to the design of list-efficient algorithms for certain problems that we consider to be basic primitives and for the case where the lists are intervals (a.k.a. ranges). Our set of primitives consists of the frequency moments F_k, $k \geq 0$. Recall that given a sequence of n items from the set $[m] \doteq \{0, \ldots, m-1\}$, the frequency f_j of item $j \in [m]$ is the number of times j appears in the sequence, and for $k \geq 0$, the k-th frequency moment, F_k, is given by $\sum_{j \in [m]} f_j^k$. Our choice of the frequency moments as the primitives to focus on is motivated by two main factors. The algorithms of [AMS99] for the frequency moments are perhaps the first major results in the design of streaming algorithms; since then they have influenced several other results, notably those of [FKSV99, Ind00, GKMS01]. Consequently, we expect that these problems, as well as ideas that emanate from their solutions, will also impact new research on streaming algorithms. Secondly, we present, in Section 6, two algorithms for counting the number of triangles in a graph; these algorithms are designed following the reduction paradigm we develop, and make essential use of list-efficient algorithms for the frequency moments.

For $k \geq 2$, we show (Section 5) how the algorithms of [AMS99] for approximating F_k to within a factor of $1 \pm \epsilon$ (including a variant of F_2) easily lend themselves to be range-efficient (that is, list-efficient where the lists are intervals). (Note that $k = 1$ is trivial with logarithmic space.)

For the case of F_0, that is, computing the number of distinct elements in a data stream, we make considerable new progress. Computing the number of distinct elements is a fundamental problem in its own right, besides its role as a primitive in streaming algorithm design. Surprisingly, however, this problem has not been fully solved. Alon et al. [AMS99], based on the hashing idea of [Sip83, Sto83, FM85], show how to produce an approximator $\widetilde{F_0}$ in the streaming model, so that $F_0/c \leq \widetilde{F_0} \leq cF_0$, for any constant $c > 2$. We first show (Section 4) how to build on the algorithm of [AMS99] to obtain similar approximations for the more interesting case where $c = 1 + \epsilon$ for any $\epsilon > 0$. Our algorithm uses polylogarithmic (in m and n) space and processing time per data item. Secondly, we show how our algorithm for F_0 can also be made range-efficient. Note that computing F_0 range-efficiently is also a natural computational problem: consider a stream of data, where each entry in the stream is an interval of integers, and the goal is to compute the number of distinct elements in the union of all the intervals. Independently, Trevisan [Tre01] has obtained a streaming algorithm for approximating F_0; however, it is not clear if his algorithm can be made list-efficient (this property is crucial for our application, counting triangles).

Counting triangles in graphs. Finally, in Section 6, we present streaming algorithms to compute the number of triangles in a graph. To the best of our knowledge, these are the first algorithms for any natural graph problem in the streaming model of computation (see also [HRR99]). We consider two models: the "adjacency stream," where the graph is presented as a sequence of edges in arbitrary order, and there is no bound on the degree of any vertex, and the "incidence stream," where we consider bounded-degree graphs and where all edges incident to a vertex are presented successively. In fact, we present unbiased estimators for the number of triangles; the variance of our estimators depend on the number of triangles in the graph. Our algorithms are based on stream reductions and use the range-efficient algorithms for F_0, F_1 and F_2. We also present lower bounds (Section 6.3) on the performance of streaming algorithms for counting triangles.

We note that counting the number of triangles is related to the question of estimating the transitivity of the binary relation represented by the graph; we believe that this will have applications to query plan optimization in databases where "degree of transitivity" of a relation is often a useful measure in deciding how to implement a relational query. We also note that our algorithms for counting triangles easily extend to any constant-sized (directed or undirected) subgraph, albeit with poorer space/time performance. The algorithms for counting small subgraphs in bounded-degree graphs is likely to have applications in the structural analysis of massive graphs like the Web graph.

2 Preliminaries

2.1 Streaming algorithms An *input stream* for a function $f : A^n \to B$ is a sequence of pairs $((\pi(1), x_{\pi(1)}), \ldots, (\pi(n), x_{\pi(n)}))$, where $x \in A^n$ is an input for f and $\pi \in S_n$ is a permutation of this input. We denote such an input stream by $\pi(x)$. A *streaming algorithm* for f is a randomized algorithm that accepts as input an error parameter $\epsilon > 0$ and a confidence parameter $0 \le \delta < 1$, and is given one-pass access to an input stream $\pi(x)$. The algorithm is required to output an ϵ-approximation of $f(x)$ with probability at least $1-\delta$, for any input x and for any permutation π.

The two main measures of complexity for streaming algorithms are the *space* and the *processing time per data item*. The space for given ϵ and δ is the maximum amount of work space the algorithm uses over all possible input streams, all input permutations, and all the random choices of the algorithm. The processing time per data item for given ϵ and δ is the maximum number of steps the algorithm spends on a single pair $(\pi(i), x_{\pi(i)})$ in the stream, where the maximum is taken over all $i \in [n]$, all possible inputs, all permutations of these inputs, and all the random choices of the algorithm. We will be interested in streaming algorithms whose space and processing time per data item are polylogarithmic in n and in $|A|$.

In this paper we consider two kinds of approximation: *relative approximation* and *ratio approximation*. An (ϵ, δ)-relative approximation of a function f, for $0 \le \epsilon \le 1$, gives for any input x a value in the interval $((1-\epsilon)f(x), (1+\epsilon)f(x))$ with probability at least $1-\delta$. A (c, δ)-ratio approximation of f, for $c \ge 1$, gives for any x a value in $((1/c)f(x), cf(x))$.

2.2 Frequency moments The k-th frequency moment, $F_k : A^n \to \mathbf{R}$, takes as input a sequence of n data items $\sigma_1, \ldots, \sigma_n$ from a set $A = \{a_1, \ldots, a_m\}$ of size m and outputs the sum $\sum_{j=1}^{m} f_j^k$, where $f_j \stackrel{\text{def}}{=} |\{i \in [n] \mid \sigma_i = a_j\}|$ is the frequency of a_j in the input sequence. F_0, for example, is the number of distinct data items in the input sequence, and F_1 is simply n.

Alon, Matias, and Szegedy [AMS99] showed several streaming algorithms to approximate the frequency moments that use (in most cases) optimal space and processing time per data item. The algorithms (known as the *AMS algorithms*) assume the domain of the data set items A is simply the set of integers $[m]$. In the following we briefly review these algorithms.

THEOREM 2.1. ([AMS99]) *There is a streaming algorithm that produces a $(c, 2/c)$-ratio approximation of F_0 for any $c > 2$, using $O(\log m)$ space and processing time per data item.*

The algorithm picks a random hash function $h : [m] \to [m]$ from a 2-universal family (e.g., the Carter-Wegman family [CW79]) , and applies h to each data item in the stream. It keeps track of the value r — the maximum number of trailing 0's in the binary representations of $h(\sigma_1), \ldots, h(\sigma_n)$, and outputs in the end $R = 2^r$. Note that the algorithm needs to store only h ($O(\log m)$ bits) and r ($O(\log \log m)$ bits) in memory. At data item σ_i, the algorithm needs to compute $h(\sigma_i)$ and compare it against r; this requires $O(\log m)$ time steps for an appropriately chosen family of hash functions (e.g., the Carter-Wegman one).

THEOREM 2.2. ([AMS99]) *There is a streaming algorithm that produces an (ϵ, δ)-relative approximation of F_2, using $O\left(\frac{1}{\epsilon^2} \log \frac{1}{\delta} (\log m + \log n)\right)$ space and processing time per data item.*

The algorithm runs $O((1/\epsilon^2)\log(1/\delta))$ independent basic estimators X for F_2 in parallel, each one having $E(X) = F_2$ and $\mathrm{Var}(X)/E^2(X) = O(1)$. Using the standard median-of-averages technique, the algorithm obtains an (ϵ, δ)-relative approximation. Each basic estimator X uses a random 4-wise independent hash function $h : [m] \to \{\pm 1\}$, computes the sum $Y = \sum_{i=1}^{n} h(\sigma_i)$, and outputs Y^2. X needs to store only h and Y in its memory (requiring $O(\log m) + O(\log n)$ bits), and to spend $O(\log m + \log n)$ step per data item to compute $h(\sigma_i)$ and update the sum Y.

THEOREM 2.3. ([AMS99]) *There is a streaming algorithm that produces an (ϵ, δ)-relative approximation of F_k, for any $k \ge 2$, using $O\left(\frac{1}{\epsilon^2} \log \frac{1}{\delta} k m^{1-1/k} (\log m + \log n)\right)$ space and processing time per data item.*

The algorithm runs $O((1/\epsilon^2)\log(1/\delta) k m^{1-1/k})$ independent basic estimators X for F_k in parallel, and as before uses median-of-averages to obtain an (ϵ, δ)-approximation. Each estimator X picks a random $i \in [n]$ and counts the number r of $i \le j \le n$ such that $\sigma_j = \sigma_i$ (i.e., the frequency of σ_i in the suffix of the sequence from position i). The estimator outputs $n(r^k - (r-1)^k)$. X needs to store in memory only σ_i ($O(\log m)$ bits) and r ($O(\log n)$ bits). The processing time per data item is $O(\log n + \log m)$.

Finally, note that F_1 can be trivially computed (exactly) using $O(\log n)$ space and $O(\log n)$ processing time per data item, simply by maintaining a counter of the data items.

3 Stream reductions and list efficiency

In this section we introduce the notion of *stream reductions*, as a new technique for streaming algorithm design. Our basic goal is to exploit existing streaming algorithms (such as the AMS algorithms for the frequency moments) to obtain new streaming algorithms for other functions. A stream reduction reduces an input stream for a function f into one or more *virtual input streams* for functions $g_1, \ldots, g_q$, so that approximations of $g_1, \ldots, g_q$ on these virtual streams yield an approximation of f on the original stream.

A stream reduction from a function $f : A^n \to B$ to a collection of functions $g_1 : A_1 \to B_1, \ldots, g_q : A_q \to B_q$ is best defined by describing a streaming algorithm M that simulates q streaming algorithms $M_1, \ldots, M_q$ for $g_1, \ldots, g_q$ and uses their outcomes to output an approximation for f. We will denote the configuration spaces of $M, M_1, \ldots, M_q$ by $C, C_1, \ldots, C_q$ respectively; a *configuration* is a binary string representing the machine's state, work space, head locations, and the values under these heads; we will assume that the configuration space is of size 2^S, where S is the machine's space. The machine M applies three basic procedures:

(1) an *approximation parameter reduction*, $\phi_A : \mathbf{R}^2 \to (\mathbf{R}^2)^q$, which maps the error and confidence parameters for the approximation of f into error and confidence parameters for the approximations of $g_1, \ldots, g_q$.

(2) a *data item reduction*, $\phi_D : A \times C \to A_1^* \times \cdots A_q^*$, which based on the currently read data item from the input stream of f and based on the current configuration of M, produces q lists of data items $L_1, \ldots, L_q$ for the functions $g_1, \ldots, g_q$; each such list may be empty, contain a single data item, or contain several data items.

(3) an *output reduction*, $\phi_O : (B_1 \times C_1) \times \cdots \times (B_q \times C_q) \to B$, which maps the outputs of $M_1, \ldots, M_q$ and their corresponding final configurations into an output for f.

As usual, M is required to output an (ϵ, δ)-approximation of $f(x)$, for any input x and for any permutation π.

Let us consider the space and time requirements of the reduction machine M. The space used by M is clearly the sum of the space used by $M_1, \ldots, M_q$, in addition to the space required to compute the reduction functions ϕ_A, ϕ_D, ϕ_O. Note that M does not need to store the output of ϕ_D after applying it to the currently read data item, because it can sequentially generate the data items in each of the lists $L_1, \ldots, L_q$ and feed them into the simulations $M_1, \ldots, M_q$.

The analysis of M's processing time per data item is more subtle. In a naive implementation, M would generate each of the items in the lists $L_1, \ldots, L_q$ one by one and feed them into the algorithms $M_1, \ldots, M_q$. This would mean that M spends $\sum_{i=1}^q |L_i| P_i$ steps per data item, where P_i is the processing time per data item of machine M_i. The size of the lists L_i may be very large (as large as n), which would imply the processing time of M is prohibitive. Note, however, that each list L_i has a succinct representation: it is fully determined by the currently read data item a and M's configuration c. Sometimes (as in our triangle application), this succinct representation takes an even more explicit form, like a range $[a_s, a_e]$ of values from A_i. Now, if each machine M_i can take the succinct representation of L_i and process all the data items in the list efficiently *as a function of the succinct representation size*, then this would enable M to have an efficient processing time per data item.

The discussion above motivates the following notion of *list efficiency* for streaming algorithms. Let $f : A^n \to B$ be a function we wish to compute by a streaming algorithm. Let $\mathcal{L} \subseteq A^*$ be a class of lists of data items from A; for each $L \in \mathcal{L}$, we denote by $s(L)$ the size of the representation of L. An input $x \in A^n$ is now represented as a stream of lists $L \in \mathcal{L}$. A streaming algorithm M for f is said to be *t-efficient with respect to $\mathcal{L}$*, if it approximates f and spends at most $t(s)$ steps for each list in the input stream, where s is the size of the representation of the list. The algorithm is required to work for any input x, and for any way of representing x as a stream of lists.

One special class of lists we will be focusing on are *ranges* of the form $[a_s, a_e]$, where $a_s \le a_e \in A$. Note that a range can be succinctly represented by its two delimiters. A streaming algorithm is called *range-efficient* if it is list efficient with respect to the class of all ranges. In some cases the data items are vectors rather than scalars; for such vectors of dimension d, we define a *j-th coordinate range*, $(a_1, \ldots, a_{j-1}, [a_{j,s}, a_{j,e}], a_{j+1}, \ldots, a_d)$, to be the list of vectors $\overline{x}$ with $x_i = a_i$ for $i \ne j$ and $x_j \in [a_{j,s}, a_{j,e}]$. We call a streaming algorithm *range-efficient in every coordinate* if it is list efficient with respect to all the coordinate ranges.

List efficiency and range efficiency are crucial in stream reductions, but may be also of independent interest. For example, one could be interested in computing frequency moments of a stream that consists of ranges of integers rather than single integers.

A notion similar to range efficiency played an important role in the work of Feigenbaum et al. [FKSV99], who presented a streaming algorithm for the L_1 distance between vectors. Their algorithm generated for each data item in the input stream a range of integers on which they needed to apply a hash function and sum the resulting values; they defined the notion of *range summable hash functions* as hash functions for which this task can be performed in time polynomial in the size of the representation of the range.

4 Counting distinct elements (F_0) in a stream

The streaming algorithm of [AMS99] for F_0 obtains a c-ratio approximation for F_0, only for $c > 2$. This gives an ϵ-relative approximation only for $\epsilon \geq 1$. It remained open to find an algorithm that obtains an (ϵ, δ)-relative approximation for arbitrarily small ϵ and δ. We present an improvement to the AMS algorithm that yields an ϵ-approximation for any $\epsilon > 0$ (Section 4.1). Trevisan [Tre01] independently discovered an ϵ-approximation algorithm for F_0 for any $\epsilon > 0$ with better parameters. However, his algorithm does not seem to admit a range-efficient implementation (see Section 4.2).

4.1 Approximating F_0 with arbitrary error

THEOREM 4.1. *There exists a streaming algorithm that produces an (ϵ, δ)-relative approximation for F_0 using $O\left(\frac{1}{\epsilon^3} \log \frac{1}{\delta} \log m\right)$ space and processing time per data item.*

Proof. We first describe a two-pass algorithm to approximate F_0, and we then show how to convert it into a one-pass algorithm.

Let $\sigma_1, \ldots, \sigma_n$ be the input for the algorithm, let $T = F_0(\sigma_1, \ldots, \sigma_n)$, and let $a_1, \ldots, a_T$ be the T distinct data items in the input sequence. Our goal is to (ϵ, δ)-approximate T.

Our algorithm runs in parallel $k = O(\log(1/\delta))$ independent estimators $Z_1, \ldots, Z_k$ for T, each one succeeding to obtain an ϵ-relative approximation of T with probability at least $2/3$. It then outputs the median of these estimators; Chernoff bound implies that this median is an ϵ-relative approximation for T with probability at least $1 - \delta$. Let us then describe one such estimator Z.

In the first pass Z uses the AMS algorithm of Theorem 2.1 to obtain a $(c, 2/c)$-ratio approximation of T, where $c = 12$. Thus, with probability at least $5/6$, Z gets a value R such that $T \leq R \leq c^2 T$. In the following we assume that Z succeeds to obtain such a value.

In the second pass Z runs $\ell = \lceil c'/\epsilon^2 \rceil$ independent basic estimators $Y_1, \ldots, Y_\ell$ for T, and outputs their average (c' is another constant to be fixed later). Each basic estimator Y picks a hash function $h : [m] \to [R]$ chosen from a 2-universal family of hash functions. Note that h may be described by $\lceil \log m \rceil + \lceil \log R \rceil$ bits. Set $B = \lceil (4c^2)/\epsilon \rceil$. Denote by $L(h)$ the list of data items a_j that h maps to 0 (i.e., $L(h) = \{a_j \in [T] \mid h(a_j) = 0\}$) and by X its size. Y outputs $R \cdot X'$, where $X' = \min\{X, B\}$.

First note that the computation of Y can be carried out using $O\left((1/\epsilon) \log m\right)$ bits of memory and spending $O\left((1/\epsilon) \log m\right)$ time per data item in the stream. In order to compute X', Y first needs to store the description of the function h ($2 \log m$ bits), and then to maintain the first B elements in the list $L(h)$. This list requires at most $O((1/\epsilon) \log m)$ bits of space. At each data item σ_i encountered in the stream Y needs to compute $h(\sigma_i)$, and if it equals 0 to check whether $\sigma_i \in L(h)$. This requires a total of $O\left((1/\epsilon) \log m\right)$ time steps.

For each $j \in [T]$, let X_j be an indicator random variable which is 1 iff $h(a_j) = 0$. Clearly, $X = \sum_{j=1}^{T} X_j$ and $E(X_j) = \Pr(h(a_j) = 0) = 1/R$, implying that $E(X) = T/R$. Our goal is thus to estimate $T = E(R \cdot X)$; however, we can get only the value of $X' = \min\{X, B\}$. First, show that $E(X')$ is not far from $E(X)$:

CLAIM 4.1. $(1 - \epsilon/2)E(X) \leq E(X') \leq E(X)$

Proof. The right inequality is trivial, since $X' \leq X$ always. To prove the left inequality, we use conditional expectation: $E(X') = E(X' \mid X \leq B) \Pr(X \leq B) + E(X' \mid X > B) \Pr(X > B) = E(X \mid X \leq B) \Pr(X \leq B) + B \Pr(X > B)$.

The pairwise independence of $X_1, \ldots, X_T$ implies that $\mathrm{Var}(X) = \sum_{j=1}^{T} \mathrm{Var}(X_j) = T \cdot (1/R) \cdot (1 - 1/R) \leq T/R \leq 1$. Thus, using Chebyshev's inequality and the fact that $E(X)$ and $\mathrm{Var}(X)$ are at most 1, we have: $\Pr(X > B) = \Pr(X \geq B + 1) \leq \Pr(|X - E(X)| \geq B) \leq \mathrm{Var}(X)/B^2 \leq \epsilon^2/(16c^4)$.

Now, we apply the Cauchy-Schwartz inequality and obtain: $E(X \mid X > B) \Pr(X > B) = \sum_{i=B+1}^{T} i \Pr(X = i) \leq (\sum_{i=B+1}^{T} i^2 \Pr(X = i))^{1/2} \cdot (\sum_{i=B+1}^{T} \Pr(X = i))^{1/2} \leq (E(X^2) \Pr(X > B))^{1/2} \leq (\mathrm{Var}(X) + E^2(X))^{1/2} \cdot \epsilon/(4c^2) \leq \sqrt{2}\epsilon/(4c^2)$, where the last inequality follows from the fact that $E(X) \leq 1$ and $\mathrm{Var}(X) \leq 1$.

Now, using the fact that $E(X) = T/R \geq 1/c^2$, we obtain $E(X \mid X > B) \Pr(X > B) \leq (\epsilon/2)E(X)$. Therefore, $E(X') \geq E(X \mid X \leq B) \Pr(X \leq B) = E(X) - E(X \mid X > B) \Pr(X > B) \geq (1 - \frac{\epsilon}{2})E(X)$. ■

We now show that Z outputs an ϵ-relative approximation of T with probability at least $2/3$. Note that $E(Z) = (1/\ell) \sum_{i=1}^{\ell} E(Y_i) = E(Y) = R \cdot E(X')$. Therefore, by Claim 4.1, $|Z - T| = |Z - R \cdot E(X)| \leq |Z - R \cdot E(X')| + |R \cdot E(X') - R \cdot E(X)| \leq |Z - E(Z)| + (\epsilon/2) \cdot R \cdot E(X) = |Z - E(Z)| + (\epsilon/2)T$.

Since $Y_1, \ldots, Y_\ell$ are independent, $\mathrm{Var}(Z) = (1/\ell^2) \sum_{i=1}^{\ell} \mathrm{Var}(Y_i) = (1/\ell)\mathrm{Var}(Y) = (R^2/\ell)\mathrm{Var}(X')$. Using Chebyshev's inequality, $\Pr(|Z - T| \geq \epsilon T) \leq \Pr(|Z - E(Z)| \geq (\epsilon/2)T) \leq 4\mathrm{Var}(Z)/(\epsilon^2 T^2) = 4R^2 \mathrm{Var}(X')/(\ell\epsilon^2 (R \cdot E(X))^2) \leq 4E(X'^2)/(\ell\epsilon^2 \cdot E^2(X)) \leq 4E(X^2)/(\ell\epsilon^2 \cdot E^2(X))$.

Now, using the fact $E(X) = T/R \geq 1/c^2$, we have: $E(X^2) = \mathrm{Var}(X) + E^2(X) \leq E(X) + E^2(X) \leq$

$E^2(X)(c^2+1)$. Therefore, $\Pr(|Z-T| \geq \epsilon T) \leq 4(c^2+1)/(\ell\epsilon^2)$. Setting $c' = 24(c^2+1)$, we have that the probability that Z outputs an ϵ-relative approximation of T is at least 5/6. Summing up the two error probabilities: the probability that Z does not obtain a value $T \leq R \leq c^2 T$ in the first pass, and the probability it does not output an ϵ-approximation in the second pass, we have that Z gets an ϵ-approximation with probability at least 2/3.

We next show how to make this a one-pass algorithm. Note that in the above description, Y needed to know the crude estimate R for T obtained by the AMS algorithm, since R was set to be the range size of h. The AMS algorithm always outputs a power of 2, and thus $R \in \{1, 2, 4, \ldots, m\}$. Our one-pass algorithm picks a pairwise independent hash function $h : [m] \to [m]$. Note that for every $R = 1, 2, 4, \ldots, m$, the function $h_R : [m] \to [R]$ obtained from h by projecting its output on the last $\log R$ bits is also a pairwise independent hash function.

Y starts its execution assuming $R = 1$, and thus initially uses h_1. Each time the current size of $L(h_R)$ exceeds B, Y replaces h_R by h_{2R}. The crucial point here is that $L(h_{2R}) \subseteq L(h_R)$; thus, when moving from h_R to h_{2R}, Y does not have to rescan the items seen to far to build $L(h_{2R})$. It can simply scan $L(h_R)$, and extract from it the items that belong to $L(h_{2R})$. Let us denote by R_Y the value of R after scanning all the items in the stream.

The estimator Z runs simultaneously the ℓ estimators $Y_1, \ldots, Y_\ell$ as well as a simulation of the AMS algorithm to get a crude estimate R^* for T. We claim that Z can still get the value of $X' = \min(X, B)$ for each of the estimators Y as follows: if $R_Y \leq R^*$, then $L(h_{R^*}) \subseteq L(h_{R_Y})$, and thus Z can extract $L(h_{R^*})$ from $L(h_{R_Y})$ and get $X' = X = |L(h_{R^*})|$; if $R_Y < R^*$, then necessarily $|L(h_{R^*})| > B$, and thus $X' = B$.

It is easy to verify that the storage requirements and the processing time per data item of each estimator Y are still $O((1/\epsilon)\log m)$. Therefore, the total space and processing time per data item are $O((1/\epsilon^3)\log(1/\delta)\log m)$. ■

This algorithm can be adapted to work with a stream of data items of any set A of size m (not necessarily the set of integers $\{1, \ldots, m\}$). All we need is an explicit mapping of A into $\{1, \ldots, m\}$. In particular, the algorithm works even if each data item in the stream is a vector of integers; we map a vector $(\sigma_{i,1}, \ldots, \sigma_{i,d}) \in [m]^d$ into $[m^d]$ by a simple radix transformation: $\sigma_i = \sigma_{i,1}m^0 + \sigma_{i,2}m^1 + \cdots + \sigma_{i,d}m^{d-1}$.

4.2 Approximating F_0 range-efficiently

In this section we show how to implement the streaming algorithm presented in the previous section range-efficiently.

Theorem 4.2. *The algorithm of Theorem 4.1 can be implemented range-efficiently spending* $O\left(\frac{1}{\epsilon^5}\log\frac{1}{\delta}\log^5 m\right)$ *time steps per range.*

Proof. We have to show how to implement each of the two basic building blocks of our algorithm range-efficiently: the AMS algorithm and the estimator Y.

Without loss of generality, we assume m is a power of two; otherwise, we replace it by the smallest power of two above it.

The 2-universal family of hash functions we use for both the AMS simulation and the estimator Y is the Toeplitz family [Gol97]: for $M \leq N$, an $M \times N$ Toeplitz matrix is one whose diagonals are homogeneous, i.e., all the entries in each diagonal contain the same value. Thus, a Toeplitz matrix is completely specified by the values at its first row and its first column. Each function $h : \{0,1\}^N \to \{0,1\}^M$ in the Toeplitz family is specified by a pair (U, v) where U is a random $M \times N$ Toeplitz matrix over $GF(2)$ and $v \in \{0,1\}^M$ is a random vector. Given a vector $x \in \{0,1\}^N$, $h(x) = Ux + v$, where the operations are over $GF(2)$. Note that each h in the family is specified by $N + 2M - 1$ bits.

We now show how, given a Toeplitz hash function $h : \{0,1\}^N \to \{0,1\}^M$ and a range $[a_s, a_e]$, where $a_s \leq a_e \in [2^N]$, to efficiently enumerate the values $x \in [a_s, a_e]$ for which $h(x) = 0$.

Let q be the largest power of two such that the interval $[k2^q, (k+1)2^q)$ for some multiple k is contained in $[a_s, a_e]$. This interval corresponds to the shortest prefix α such that all the values $x \in \{0,1\}^N$ with prefix α are contained in the range $[a_s, a_e]$. We set $c_s = k2^q$ and $c_e = (k+1)2^q$, and show how to enumerate the data items in the range $[c_s, c_e)$ that h maps to 0. The data items in $[a_s, c_s)$ and $[c_e, a_e]$ are enumerated recursively in a similar way.

For each $x \in [c_s, c_e)$, let us denote by $x^{(1)}$ the projection of x on its first $(N-q)$ bits, and by $x^{(2)}$ its projection on the last q bits. Similarly, we denote by $U^{(1)}$ the first $(N-q)$ columns of U and by $U^{(2)}$ its last q columns. Clearly, for all $x \in [c_s, c_e)$, $Ux = U^{(1)}x^{(1)} + U^{(2)}x^{(2)} = U^{(1)}\alpha + U^{(2)}x^{(2)}$. Thus, $h(x) = Ux + v = 0$ for $x \in [c_s, c_e)$ if and only if,

$$U^{(2)}x^{(2)} = -(v + U^{(1)}\alpha) \tag{4.1}$$

This implies that an enumeration of all the solutions of the linear system (4.1) yields an enumeration of all the data items $x \in [c_s, c_e)$ that h maps to 0. We can get the solutions to (4.1) by diagonalizing $U^{(2)}$ using Gaussian Elimination.

The running time of enumerating t data items with this procedure is at most $O(\log(a_e - a_s)N^3 + tN)$, since the procedure has at most $O(\log(a_e - a_s))$ recursive calls, each one requiring a Gaussian Elimination of a matrix of dimension at most $N \times N$, and each enumerated data item requires at most $O(N)$ steps to produce from the diagonalized matrix.

In the simulation of the AMS algorithm, we use a hash function $h : [m] \rightarrow [m]$, and we need to find for each given range $[a_s, a_e]$ whether the value of h on any of the data items in this range has a 0-suffix longer than r (the current longest 0-suffix). Our simulation will sequentially test whether each one of the following set of $(\log m - r)$ equations holds in the interval $[a_s, a_e]$:

$$(Ux + v) \bmod 2^j = 0, \; j = r+1, \ldots, \log m$$

As soon as the simulation finds an equation j which does not hold it stops, and sets $r = j - 1$. We now show how to test each of these equations. For equation j, define U_j to be the last j rows of U, and v_j to be the projection of v on its last j bits. Note that $(Ux+v) \bmod 2^j = 0$ if and only if $U_j x + v_j = 0$. (U_j, v_j) can be viewed as a Toeplitz hash function $h_j : \{0,1\}^{\log m} \rightarrow \{0,1\}^j$. Therefore, using the procedure described above we can enumerate all the solutions to this equation (and in particular, find whether there exists a solution), in time $O(\log^4 m)$. Thus, the total simulation of the AMS algorithm per range requires at most $O(\log^5 m)$ steps.

The computation of our basic estimator Y on a range $[a_s, a_e]$ is carried out as follows. Let R be the current hash range size Y uses. Y sets up the equation system 4.1 corresponding to h and $[a_s, a_e]$. It then starts to enumerate the solutions of only the *first* $\log R$ equations, because these are the solutions corresponding to h_R. If $|L(h_R)|$ exceeds B, Y suspends the solution enumeration, replaces h_R by h_{2R}, extracts from $L(h_R)$ the items that h_{2R} maps to 0, and resumes the enumeration, but now of the solutions of the first $\log R + 1$ equations.

The processing time of Y per range is at most $O(\log^4 m + (1/\epsilon^2) \log^2 m)$, because Y may need to enumerate at most $O((1/\epsilon) \log m)$ solutions ($O(1/\epsilon)$ per value of R), which would take $O(\log^4 m + (1/\epsilon) \log^2 m)$ time, and furthermore, the comparison of each enumerated data item against a list of size at most $O(1/\epsilon)$ requires $O((1/\epsilon) \log m)$ steps in a naive implementation.

To conclude, the total running time of our algorithm per range is $O(1/\epsilon^4 \log(1/\delta) \log^5 m)$, since it runs $O(\log(1/\delta))$ copies of the AMS simulations and $O(1/\epsilon^2 \log(1/\delta))$ copies of the Y estimator.

It can be easily verified that the space requirements of this implementation are not different from what is stated in Theorem 4.1. ∎

A simple generalization shows that we can compute F_0 range-efficiently in every coordinate, even if the data items are vectors rather than scalars:

Proposition 4.1. *The algorithm of Theorem 4.1 can be implemented range-efficiently in every coordinate spending* $O\left(\frac{1}{\epsilon^4} \log \frac{1}{\delta} d \log^5 m\right)$ *time steps per range, where d is the dimension of the input data items.*

Proof. The inputs we consider here are vectors rather than scalars: $\sigma_i \in [m]^d$ for every data item σ_i in the stream. A range is specified by a coordinate $j \in [d]$, an assignment $(a_1, \ldots, a_{j-1}, a_{j+1}, \ldots, a_d)$ to all the coordinates except the j-th one, and a range $[a_{j,s}, a_{j,e}]$ for the j-th coordinate.

In the one dimensional case all the data items in a range shared a common prefix α. Here, all the data items in a range share both a common prefix α and a common suffix β. Thus, the algorithm of Theorem 4.2 requires only a slight modification in order to work for this case too. We split the matrix U into three parts: $U^{(1)}$ — the part that corresponds to α, $U^{(2)}$ – the middle part, and $U^{(3)}$ — the part that corresponds to β. The linear system we will need to solve is the following:

$$U^{(2)} x^{(2)} = -(v + U^{(1)} \alpha + U^{(3)} \beta)$$

The rest of the argument continues as before. ∎

5 Approximating other frequency moments range-efficiently

In this section we first show how the Gilbert et al. [GKMS01] range-efficient implementation of the AMS algorithm for F_2 can be extended to be range-efficient in every coordinate. We then show range-efficient implementations of the AMS algorithm for F_k, $k \geq 2$.

Proposition 5.1. *The AMS algorithm for F_2 can be implemented range-efficiently in every coordinate using* $O\left(\frac{1}{\epsilon^2} \log \frac{1}{\delta} (d \log^2 m + \log n)\right)$ *space and spending* $O\left(\frac{1}{\epsilon^2} \log \frac{1}{\delta} (d \log^{O(1)} m + \log n)\right)$ *steps per range, where d is the dimension of the input data items.*

Proof. Gilbert et al. [GKMS01] show how to implement the AMS algorithm for F_2 (Theorem 2.2) range-efficiently. They exhibit a construction of a 7-wise independent family of hash functions $h : [m] \rightarrow \{\pm 1\}$ based on second order Reed-Muller codes, and prove that the functions in this family are range-summable; that is, given any function h from the family and a range $[a_s, a_e]$, it is possible to compute the sum $\sum_{x=a_s}^{a_e} h(x)$ in polynomial time. This immediately enables running the AMS algorithm for F_2 spending $O(1/\epsilon^2 \log(1/\delta) \log^{O(1)} m)$ steps per range.

Each hash function $h : [m] \rightarrow \{\pm 1\}$ in the Reed-Muller family corresponds to a degree-2 $(\log m)$-variate polynomial $p_h(x_1,\ldots,x_{\log m})$ over $GF(2)$. h is specified by $\binom{\log m}{2}+\log m+1$ bits, corresponding to the coefficients of the polynomial p_h. Each input $x\in[m]$ of h is interpreted as an assignment to the $\log m$ variables, and the value of h is simply $p_h(x)$. It is easy to check that this family is 7-wise independent; [GKMS01] used a fact from Linear Group Theory [Dic58] to give a polynomial time algorithm that computes $\sum_{x=a_s}^{a_e} h(x)$.

We show how an easy extension of the method of [GKMS01] yields an algorithm for F_2 which is range-efficient in every coordinate. In our case each input is a vector $\sigma_i\in[m]^d$. Each hash function $h:[m]^d\rightarrow\{\pm1\}$ corresponds now to a degree-2 ($d\log m$)-variate polynomial p_h. Our main observation is the following: given an assignment $\overline{a}=(a_1,\ldots,a_{j-1},a_{j+1},\ldots,a_d)$ to the $d-1$ coordinates different from j, the polynomial $p_h^{\overline{a}}(x_j)\stackrel{\text{def}}{=}p_h(a_1,\ldots,a_{j-1},x_j,a_{j+1},\ldots,a_d)$ is a degree-2 $\log m$-variate polynomial. Moreover, the coefficients of $p_h^{\overline{a}}$ can be easily computed (in time $O(d\log^2 m)$) from the coefficients of p_h and from the assignment $\overline{a}$. Thus, given p_h and a coordinate range $(j,a_{j,s},a_{j,e},\overline{a})$, we can compute the coefficients of the polynomial $p_h^{\overline{a}}$, and use the [GKMS01] algorithm to compute $\sum_{x=a_{j,s}}^{a_{j,e}} p_h^{\overline{a}}(x)=\sum_{x=a_{j,s}}^{a_{j,e}} p_h(a_1,\ldots,a_{j-1},x,a_{j+1},\ldots,a_d)$. This immediately implies an implementation of the AMS algorithm for F_2 that is range-efficient in every coordinate. ■

Proposition 5.2. *The AMS algorithm for F_k, $k\geq 2$, can be implemented range-efficiently spending $O\left(\frac{1}{\epsilon^2}\log\frac{1}{\delta}km^{1-1/k}(\log m+\log n)\right)$ steps per range.*

Proof. The range-efficient implementation of the algorithm for F_k (Theorem 2.3) is completely straightforward. Each basic estimator X of the algorithm needs to find, given an item σ_i and a range $[a_s,a_e]$, how many times σ_i occurs in the range, and update the counter r accordingly. σ_i either does not occur in the range at all or occurs exactly once, depending on whether $a_s\leq\sigma_i\leq a_e$. Thus, the processing time per range of each estimator X is $O(\log m+\log n)$. ■

An identical argument shows how to implement this algorithm range-efficiently in every coordinate:

Proposition 5.3. *The AMS algorithm for F_k, $k\geq 2$, can be implemented range-efficiently in every coordinate spending $O\left(\frac{1}{\epsilon^2}\log\frac{1}{\delta}km^{d(1-1/k)}(d\log m+\log n)\right)$ steps per range, where d is the dimension of the data items.*

6 Counting triangles in graphs

Our notion of reductions in streaming algorithms leads to the following application. The goal is to design a streaming algorithm that approximates the number of triangles in a graph. As noted in Section 1, counting the number of triangles in large graphs has applications to databases, query optimization, and the World-Wide Web. Counting triangles (and more generally, small cycles) is a well-studied problem in the non-streaming case (cf. [AYZ97]).

Let $G=(V,E)$ be an undirected graph with $|V|=n$ and let the degree of a vertex $v\in V$ be denoted $\deg(v)$. We consider two possible stream representations of graphs:

(1) Each edge $e=\{u,v\}\in E$ is a data item. The data stream consists of edges in E in some arbitrary order. We call this the *adjacency stream* of the graph.

(2) Each node $u\in V$ along with all its neighbors $v_1,\ldots,v_{\deg(v)}$ is a data item. The data stream consists of data items for each of the nodes in V in some arbitrary order. We call this the *incidence stream* of the graph.

The adjacency stream is more appropriate for high degree graphs, while the incidence stream suits bounded-degree graphs better.

Let $\mathcal{V}$ be the set of all (unordered) vertex triples of G (i.e., the set of all subsets of V of size 3). We classify every triple $\{u,v,w\}$ in $\mathcal{V}$ into one of four classes according to how many of the three pairs $\{u,v\}$,$\{u,w\}$, and $\{v,w\}$ are edges in G. This induces a partition of $\mathcal{V}$ into four disjoint subsets, $\mathcal{V}_0,\mathcal{V}_1,\mathcal{V}_2,\mathcal{V}_3$, where $\mathcal{V}_j$ ($j=0,1,2,3$) consists of all triples whose exactly j of their corresponding pairs are edges. We denote by T_0,T_1,T_2,T_3 the sizes of these subsets. Note that T_3 is exactly the number of triangles in the graph G, which is the quantity we wish to approximate.

Our approach for approximating T_3 is to reduce the problem of counting undirected triangles to estimating frequency moments for an appropriate virtual stream. We then apply the basic tools we have developed to approximate these frequency moments. We present (Section 6.1) an algorithm that works for graphs given in the adjacency stream representation; the space and processing time per data item of this algorithm depend cubically on $(T_1+T_2)/T_3$. We present (Section 6.2) an algorithm that works for bounded-degree graphs given in the incidence stream representation; the space and processing time per data item of this algorithm depend quadratically on T_2/T_3. We present (Section 6.3) a space lower bound of $\Omega(n^2)$ for approximating the number triangles in a graph given in the adjacency stream representation; this hints that for general graphs, it is impossible to approximate the number of triangles efficiently with a streaming algorithm.

A naive sampling algorithm to approximate the number of triangles in a graph picks $O((1/\epsilon^2)\log(1/\delta)(1+(T_0+T_1+T_2)/T_3))$ random triples from $\mathcal{V}$ and finds what fraction of them are triangles; this gives an (ϵ, δ)-approximation of T_3. It is not clear whether our adjacency streaming algorithm is strictly superior to naive sampling; in fact, both the sampling algorithm and our algorithm are efficient only on graphs with $\Omega(n^2/\text{poly}\log n)$ edges. On the other hand, our algorithm for the bounded-degree case is strictly superior to naive sampling, since for degree d graphs with $T_2/T_3 = o(\sqrt{n})$, our algorithm runs in $o(n)$ space, while the sampling algorithm requires $\Omega(n^3/(nd^2)) = \Omega(n^2)$ space. This happens whenever the graph has lot of correlations among neighbors of a vertex.

6.1 Adjacency stream algorithm

THEOREM 6.1. *There is a streaming algorithm that for any $\epsilon, \delta > 0$, and for any adjacency stream of a graph with $T_3 > 0$, computes an (ϵ, δ)-relative approximation of T_3 using space $s = O\left(\frac{1}{\epsilon^3} \cdot \log \frac{1}{\delta} \cdot \left(1 + \frac{T_1+T_2}{T_3}\right)^3 \cdot \log n\right)$, and $\text{poly}(s)$ processing time per data item.*

Proof. For each $e = \{u, v\} \in E$, let $\mathcal{V}_e$ denote the set of triples in $\mathcal{V}$ that contain both u and v. Note that $|\mathcal{V}_e| = n - 2$. Each triple in $\mathcal{V}_e$ represents a data item. Let σ denote the data stream consisting of the triples from $\mathcal{V}_e$ for every $e \in E$, where the ordering of e's is arbitrary. It follows $|\sigma| = (n-2)|E|$. Given an adjacency stream of G, σ is easy to construct — upon receiving $\{u, v\}$ in the input stream, output $\{u, v, w\}$ for $w \in V, w \neq u, v$.

Our main observation is the following:

$$F_k = F_k(\sigma) = T_1 \cdot 1^k + T_2 \cdot 2^k + T_3 \cdot 3^k.$$

This is because each triple in T_i contributes i data items to σ. Now, we can set up three linear equations:

$$\begin{pmatrix} F_0 \\ F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \\ 1 & 4 & 9 \end{pmatrix} \cdot \begin{pmatrix} T_1 \\ T_2 \\ T_3 \end{pmatrix}.$$

By inverting the non-singular matrix of this system, we can write T_3 in terms of F_0, F_1, F_2 as $T_3 = F_0 - 1.5F_1 + 0.5F_2$. Thus, we can approximate T_3 by approximating F_0 and F_2.

LEMMA 6.1. *If $\tilde{F}_0$ is an (ϵ', δ)-approximation of F_0 and $\tilde{F}_2$ is an (ϵ', δ)-approximation of F_2 for $\epsilon' = \epsilon T_3/(6(T_1+T_2+T_3))$, then $\tilde{T}_3 = (\tilde{F}_0 - 1.5F_1 + 0.5\tilde{F}_2)/3$ is an $(\epsilon, 2\delta)$-approximation of T_3.*

Thus, the overall algorithm is to use the original stream to construct the virtual stream σ, apply the streaming algorithms for F_0, F_1, and F_2 to get F_1 and ϵ'-approximations $\tilde{F}_0$ and $\tilde{F}_2$, and then output $(\tilde{F}_0 - 1.5F_1 + 0.5\tilde{F}_2)/3$.

The space requirements of this algorithm are governed by the space requirements of the F_0 and F_2 algorithms. Note that here we need those algorithms to be able to deliver accuracies arbitrarily close to 1; thus, we need to use the F_0 algorithm of Theorem 4.1.

In order to achieve the claimed processing time per data item, we use the following representation of triples in the virtual stream. Each unordered triple $\{u, v, w\}$ is represented by the dimension 3 vector (u, v, w) where $u < v < w$. Note that for every edge $e = (u, v)$ in the input stream, the set of triples $\mathcal{V}_e$ in the virtual stream can be represented as three coordinate ranges: $([1, u-1], u, v), (u, [u+1, v-1], v)$, and $(u, v, [v+1, n])$. Thus, we can use the versions of the streaming algorithms for F_0 and F_2 that are range-efficient in every coordinate (see Sections 4 and 5) to compute $F_0(\sigma)$ and $F_2(\sigma)$ with $O((1/\epsilon'^4)\log(1/\delta)\log^{O(1)} n)$ processing time per range. ■

6.2 Incidence stream algorithm

THEOREM 6.2. *There is a streaming algorithm that for any $\epsilon, \delta > 0$ and for any incidence stream of a graph $G = (V, E)$ with $\max_{v \in V} \deg(v) = d$ and $T_3 > 0$, computes an (ϵ, δ)-relative approximation of T_3 using space $s = O\left(\frac{1}{\epsilon^2} \cdot \log \frac{1}{\delta} \cdot \left(1 + \frac{T_2}{T_3}\right)^2 \cdot \log n + d \log n\right)$, and $\text{poly}(s)$ processing time per data item.*

Proof. For each $u \in V$ with neighbors $v_1, \ldots, v_d$, let $\mathcal{V}_u$ denote the set of triples in $\mathcal{V}$ of the form $\{u, v_i, v_j\}$ for $i \neq j$. Note that $|\mathcal{V}_u| = \binom{d}{2}$. Each triple in $\mathcal{V}_u$ represents a data item. Let σ denote the data stream consisting of the triples from $\mathcal{V}_u$ for every $u \in V$, where the ordering of u's is arbitrary. It follows $|\sigma| = n \cdot \binom{d}{2}$. Given an incidence stream of G, σ is easy to construct — upon receiving $u, v_1, \ldots, v_d$ in the stream, output $\{u, v_i, v_j\}$ for $i \neq j$.

Our main observation is the following. By construction, we get for any $k \geq 0$,

$$F_k = F_k(\sigma) = T_2 \cdot 1^k + T_3 \cdot 3^k.$$

This is because, each triple in $\mathcal{V}_2$ contributes one data item to σ and each triple in $\mathcal{V}_3$ contributes three data items to σ. This gives the following linear system:

$$\begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 1 & 3 \\ 1 & 9 \end{pmatrix} \cdot \begin{pmatrix} T_2 \\ T_3 \end{pmatrix}.$$

Therefore, by solving the system, we can write T_3 in terms of F_1, F_2 as $T_3 = (-F_1 + F_2)/6$. But, $F_1 = |\sigma| = n\binom{d}{2}$. Thus, we can approximate T_3 by just approximating F_2.

LEMMA 6.2. *If $\tilde{F}_2$ is an (ϵ', δ)-approximation of F_2 for $\epsilon' = 6\epsilon T_3/(T_2 + 9T_3)$, then $\tilde{T}_3 = (-F_1 + \tilde{F}_2)/6$ is an (ϵ, δ)-approximation of T_3.*

The rest of the proof is similar to the proof of Theorem 6.1, with the following differences: (1) this algorithm needs to run only the F_2 algorithm on the virtual stream; therefore, it requires only $O(1/\epsilon'^2 \cdot \log(1/\delta) \log n)$ space; (2) the algorithm needs additional $O(d \log n)$ space to store each data item $u, v_1, \ldots, v_d$ as it arrives in the input stream; (3) the processing time per data item is multiplied by a factor of $\binom{d}{2}$, since the algorithm processes separately each triple (u, v_i, v_j) (4) we do not need range efficiency here, since we handle each item in the virtual stream individually. ∎

6.3 A lower bound for counting triangles

We show that in general it is impossible to approximate the number of triangles in a graph given as an adjacency stream using $o(n^2)$ space:

THEOREM 6.3. *For all sufficiently large n, there exists a family $\mathcal{G}$ of graphs on $3n$ nodes such that any streaming algorithm A that (ϵ, δ)-approximates T_3 for $0 < \epsilon < 1$ and $0 < \delta < 1/100$ requires $\Omega(n^2)$ space for at least one of the graphs in $\mathcal{G}$.*

The proof is omitted in this extended abstract.

7 Discussion and open problems

Our triangle algorithms easily generalize to directed graphs as well. They can be also generalized to find the number of copies of larger cliques; however, this would require using streaming algorithms for higher frequency moments (for $k \geq 3$), which are not as efficient as the ones for F_0, F_1, F_2.

Some issues left are open are: (1) Finding a range-efficient algorithm for F_0 with a better dependence on $1/\epsilon$; this would directly improve the dependence of our triangle algorithm on the ratio $(T_1+T_2)/T_3$. (2) Getting a more explicit space lower bound for counting triangles in terms of T_1, T_2, T_3. (3) Understanding the situations in which our adjacency stream algorithm is superior to the naive sampling algorithm.

Acknowledgements

We thank T. S. Jayram, Christos Papadimitriou, and Luca Trevisan for helpful discussions. We are especially grateful to Martin Strauss for sharing portions of [GKMS01], and for helpful discussions.

References

[AMS99] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *JCSS*, 58(1):137–147, 1999.

[AYZ97] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997.

[Bab01] S. Babu. Work related to STREAM project, 2001. http://www-db.stanford.edu/stream/related.html.

[CW79] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *JCSS*, 18(2):143–154, 1979.

[Dic58] L. E. Dickson. *Linear Groups with an Exposition of the Galois Field Theory.* Dover, 1958.

[EIO01] L. Engebretsen, P. Indyk, and R. O'Donnell. Derandomized dimensionality reduction with applications. *Proc. 13th SODA*, 2002.

[FKSV99] J. Feigenbaum, S. Kannan, M. Strauss, and M. Viswanathan. An approximate L^1-difference algorithm for massive data streams. *Proc. 40th FOCS*, pp. 501–511, 1999.

[FM85] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *JCSS*, 31(2):182–209, 1985.

[FS00] J. Fong and M. Strauss. An approximate l^p-difference algorithm for massive data streams. *Proc. STACS*, pp. 193–204, 2000.

[GKMS01] A. C. Gilbert, Y. Kotidis, S. Muthuskrishnan, and M. J. Strauss. A few good terms: Efficient streaming computation of wavelet decompositions. Manuscript, 2001.

[Gol97] O. Goldreich. A sample of samplers — a computational perspective on sampling (survey). *ECCC*, TR97-020, 1997.

[GKS01] S. Guha, N. Koudas, and K. Shim. Data streams and histograms. *Proc. 33rd STOC*, pp. 471–475, 2001.

[HRR99] M. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. *DIMACS series in Discr. Math. & Theor. Comp. Sc.*, 50:107–118, 1999.

[Ind00] P. Indyk. Stable distributions, pseudorandom generators, embeddings and data stream computation. *Proc. 41st FOCS*, pp. 189–197, 2000.

[KN97] E. Kushilevitz and N. Nisan. *Communication Complexity.* Cambridge University Press, 1997.

[NW94] N. Nisan and A. Wigderson. Hardness vs. randomness. *JCSS*, 49(2):149–167, 1994.

[Sip83] M. Sipser. A complexity theoretic approach to randomness. *Proc. 15th STOC*, pp. 330–335, 1983.

[Siv01] D. Sivakumar. Algorithmic derandomization via complexity theory. Manuscript, 2001.

[Sto83] L. J. Stockmeyer. The complexity of approximate counting. *Proc. 15th STOC*, pp. 118–126, 1983.

[Tre01] L. Trevisan. A note on counting distinct elements in the streaming model. Manuscript, 2001.

[Yao77] A.C-C. Yao. Probabilistic computations: toward a unified measure of complexity. *Proc. 18th FOCS*, pp. 222–227, 1977.