Open access • Proceedings Article • DOI:10.1145/1142351.1142388

Counting triangles in data streams — Source link

Luciana S. Buriol, Gereon Frahling, Stefano Leonardi, Alberto Marchetti-Spaccamela ...+1 more authors

Institutions: Universidade Federal de Santa Maria, University of Paderborn, Sapienza University of Rome

Published on: 26 Jun 2006 - Symposium on Principles of Database Systems

Topics: Multiple edges, Multigraph, Strength of a graph, Line graph and Force-directed graph drawing

Counting Triangles in Data Streams*

Luciana S. Buriol *
Departamento de Eletrônica e Computação
Universidade Federal de Santa Maria, Brazil
buriol@inf.ufsm.br

Gereon Frahling
Heinz Nixdorf Institut
University of Paderborn, Germany
frahling@uni-paderborn.de

Stefano Leonardi †
Dipartimento di Informatica e Sistemistica
Universitá di Roma "La Sapienza", Italy
leon@dis.uniroma1.it

Alberto Marchetti-Spaccamela
Dipartimento di Informatica e Sistemistica
Universitá di Roma "La Sapienza", Italy
alberto@dis.uniroma1.it

Christian Sohler
Heinz Nixdorf Institut
University of Paderborn, Germany
csohler@upb.de

ABSTRACT

We present two *space bounded* random sampling algorithms that compute an approximation of the number of triangles in an undirected graph given as a stream of edges. Our first algorithm does not make any assumptions on the order of edges in the stream. It uses space that is inversely related to the ratio between the number of triangles and the number of triples with at least one edge in the induced subgraph, and constant expected update time per edge. Our second algorithm is designed for incidence streams (all edges incident to the same vertex appear consecutively). It uses space that is inversely related to the ratio between the number of triangles and length 2 paths in the graph and expected update time $O(\log |V| \cdot (1 + s \cdot |V|/|E|))$, where s is the space requirement of the algorithm. These results significantly improve over previous work [20, 8]. Since the space complexity depends only on the structure of the input graph and not on the number of nodes, our algorithms scale very well with increasing graph size and so they provide a basic tool to analyze the structure of large graphs. They have many applications, for example, in the discovery of Web communities, the computation of clustering and transitivity coefficient, and discovery of frequent patterns in large graphs.

We have implemented both algorithms and evaluated their performance on networks from different application domains. The sizes of the considered graphs varied from about $8,000$ nodes and $40,000$ edges to 135 million nodes and more than 1 billion edges. For both algorithms we run experiments with parameter $s = 1,000,\ 10,000,\ 100,000,\ 1,000,000$ to evaluate running time and approximation guarantee. Both algorithms appear to be time efficient for these sample sizes. The approximation quality of the first algorithm was varying significantly and even for $s = 1,000,000$ we had more than 10% deviation for more than half of the instances. The second algorithm performed much better and even for $s = 10,000$ we had an average deviation of less than 6% (taken over all but the largest instance for which we could not compute the number of triangles exactly).

Categories and Subject Descriptors: H.3 [Information Systems]: Information Storage and Retrieval

General Terms: Algorithms, Theory, Performance.

Keywords: Streaming algorithms, graph algorithms, network analysis.

1. INTRODUCTION

Graphs are fundamental structures for modeling complex relationships between data in Web documents, chemical compounds, XML, social networks etc. A basic tool to uncover their structural design principles and to extract relevant information is to mine the most frequent interconnection patterns occurring in the graph.

The computation of network indices based on counting the number of certain small subgraphs is a basic tool in the analysis of the structure of large networks. The clustering coefficient [18] is defined as the normalized sum of the fraction of neighbor pairs of a vertex of the graph that are connected. The related transitivity coefficient of a graph [6], is defined as the ratio between three times the number of triangles and the number of length two paths in the graph. More recently,

*This work was partially supported by the EU within the 6th Framework Programme under contract 001907 "Dynamically Evolving, Large Scale Information Systems" (DELIS)

*Part of this work was done while the author was post-doc at Universitá degli Studi di Roma "La Sapienza"

†Part of this work was done while the author was visiting the School of Computer Science at Carnegie Mellon University

PODS'06, June 26–28, 2006, Chicago, Illinois, USA.

much attention has been devoted to the analysis of complex networks arising in information systems, software systems, overlay networks etc. Mining the most frequent subgraphs is here aimed to identify the building blocks of universal classes of complex networks [14]. As an example, the occurrence of a very large number of certain dense subgraphs has been observed in the Webgraph, the graph formed by Web pages and hyperlinked connections [10], in the attempt of tracing the emergence of hidden cyber-communities. A stochastic model of the growth of the Webgraph [9], the "copying model", has these dense subgraphs as building blocks of the process of network formation.

Finding frequent graph patterns also finds application to graph databases where the classical graph query problem consists of finding all the graphs of the database containing a given query pattern as a subgraph. A direct indexing of the most frequent subgraphs, up to a maximum size, that occur in the graph [19] has for instance been proposed as an alternative to path indexing [16].

Counting the number of certain subgraphs in a large graph is a challenging computational task. The current state of the art provides methods that are either computational infeasible on large data sets or do not provide any guarantee on the accuracy of the estimation. The best known methods for the solution of the simplest non trivial version of this problem, i.e. counting the number of triangles in a subgraph, reduces to matrix multiplication [3]. This is not computational feasible even on graphs of medium size, because of time complexity and the space required to store the whole graph and the related data structures in main memory. Schank and Wagner [15] give an extensive experimental study of the performance of algorithms for counting and listing triangles in graphs.

A natural way to address the problem of computing with massive data sets is to resort to the data stream model [7, 12]. In this model data arrives in a stream, one item at a time, and the algorithms are required to use very little space and per-item processing time. Secondary and slower memory storage devices naturally produce data streams for which multiple passes of computation are usually prohibitive due to the volumes of stored data. In several network contexts, the application receive data without pace from remote sources. Data stream computation allows also to compute on-line relevant quantities without incurring a large cost for organizing and storing data. We think for example of a distributed crawler collecting Web pages and their links, and performing structural analysis of the Webgraph prior to transfer data to a storage device.

Data stream algorithms have been proposed for problems like computation of frequency moments [1], histograms [5], Wavelet transforms [4], and others. This large body of work contrasts with a lack of efficient solutions of natural graph problems in the streaming model of computation [7]. Bar-Yosseff, Kumar and Sivakumar [20] give a first solution for counting triangles in the data stream model. They consider both the "adjacency stream" model where the graph is presented as a sequence of edges in arbitrary order and there is no bound on the degree of a vertex, and the "incidence stream" model where they consider only bounded-degree graphs and all edges incident to a vertex are presented successively. Their algorithms provide an ϵ approximation with probability $1-\delta$ using a number of memory cells in some cases smaller than a naive sampling technique algorithm. The algorithms are obtained through a so called "list" efficient reduction to the problem of computing frequency moments [1]. Subsequently, more algorithms have also been developed for the adjacency stream model [8]. These solutions are still far from being practical for most real world networks.

1.1 Our contribution

In this work we specifically present unbiased estimators for the number of triangles in the graph and the number of cliques of any size. Our data stream algorithms compute a $(1+\epsilon)$-approximation with probability $1-\delta$. A complete overview of the application of our method to other classes of subgraphs will be given in the full version of our work.

When estimating the number of triangles, if the graph is given as a stream of edges in arbitrary order, the data structure uses $O(\frac{1}{\epsilon^2}\cdot\log(\frac{1}{\delta})\cdot(1+\frac{|T_1|+|T_2|}{|T_3|}))$ memory cells, where T_i denotes the set of node-triples having i edges in the induced subgraph. $|T_3|$ is therefore the measure we like to estimate. This is always better than the naive sampling algorithm that requires $O(\frac{1}{\epsilon^2}\log(\frac{1}{\delta})(1+\frac{|T_0|+|T_1|+|T_2|}{|T_3|}))$ memory cells, while it dramatically improves by a cubic factor the solution provided in [20]. Comparing our results in this model with the previous work in [8], we obtain a one-pass algorithm that achieves the same space bound and better update time as the three pass algorithm from [8]. The two other algorithms in [8] either require bounded maximum degree or are incomparable to our result because the space complexity depends on different parameters (e.g., the number of cycles of length 4 and 6 in the graph). The number of memory cells used by our algorithm still depends on the cardinality of T_1, that can be as large as $O(|E|\cdot|V|)$. Our method in the case of graphs in arbitrary order is therefore of practical interest for networks with a large enough number of triangles.

Of greater practical relevance is our method when the graph is given as an incidence stream. The number of memory cells used by our data structure is $O(\frac{1}{\epsilon^2}\log(\frac{1}{\delta})\log(|V|)(1+\frac{|T_2|}{|T_3|}))$. To give a flavor of the quality of our result, observe that $\frac{|T_3|}{|T_2|}$ is exactly equal to 1/3 of the inverse of the *transitivity coefficient* of the graph, a universal measure closely related to the clustering coefficient, whose value for networks of practical interest is hardly bigger than 10^5. Therefore, the number of memory cells of our data structure depends, for all practical purposes, only logarithmically on the size of the network. Our algorithmic results improve by a quadratic factor the result of [20] and always improves over the naive sampling method.

Our method is suitable to be adapted to several other classes of subgraphs. As an example, we provide, in the incidence stream model, an algorithm to estimate the number of cliques of size α that uses $O(\frac{1}{\epsilon^2}\log(\frac{1}{\delta})\log(|V|)(1+\frac{|S_\alpha|}{|K_\alpha|}))$ memory cells, where S_α is the number of stars of cardinality α and K_α is the number of cliques of size α in the graph (see [2] for details).

In the second part of this work, we provide an optimized implementation of our data stream algorithms and test it on networks of various sizes collected in different application domains. Our dataset includes large webgraphs, graphs of the largest online encyclopedia Wikipidia [11], graphs of collaborations between actors and authors. For all networks we consider, a sample of size 10^4 already suffices to provide an excellent approximation of the number of triangles in the in-

cident stream model. The running times of our algorithms, measured in terms of operations performed in main memory, is always lower than the running times measured on the same computing platform for the best known implementation of several heuristics provided in [15]. For the model with stream of edges in arbitrary order, our optimized implementation shows that a sample of size 10^6 is sufficient for a reasonable approximation of the number of triangles of certain networks, whereas it is still too small for some networks for which the algorithm in the incident stream model provides an excellent approximation with a sample of size 10^4.

1.2 Structure of the paper

We present in Section 2 the algorithm for counting the number of triangles in the adjacent stream model, in section 3 the model for counting the number of triangles in the incidence stream model. Section 4 presents the optimized implementations of the algorithms and Section 5 the description of our dataset and the experimental results.

2. STREAMS OF EDGES IN ARBITRARY ORDER

We consider undirected graphs without self-loops. Each edge is an unordered pair of nodes (v, w) such that $(v, w) = (w, v)$. We assume that $V = \{1, \ldots, n\}$ and n is known in advance. We have access to a stream consisting of all edges in the graph. The edges appear in arbitrary order and no edge is repeated in the stream. There is no bound on the degree of the nodes.

2.1 3 Pass Algorithm

We will first present an algorithm which passes three times over the stream. A different algorithm with the same space complexity has been presented in [8]. However, our algorithm has a significantly improved update time and as we later show, we can combine the three passes to a one-pass algorithm.

We introduce a streaming algorithm SampleTriangle, which outputs a $\{0, 1\}$ variable with expected value $3|T_3|/(|T_1|+2\cdot|T_2|+3\cdot|T_3|)$. Using the property $|T_1|+2\cdot|T_2|+3\cdot|T_3| = |E|\cdot(|V|-2)$ (Lemma 2.1) we can estimate $|T_3|$ using $1/\epsilon^2$ parallel runs of algorithm SampleTriangle.

SampleTriangle
1st. Pass:
Count the number of edges $|E|$ in the stream
2nd. Pass:
Sample an edge $e = (a, b)$ uniformly chosen from E
Choose a node v uniformly from $V \setminus \{a, b\}$
3rd. Pass:
if $(a, v) \in E \wedge (b, v) \in E$ **then** $\beta = 1$
else $\beta = 0$
return β

It is easy to see that each of the above passes can be implemented in a single pass over the set of edges (i.e., the input stream) using $O(1)$ memory cells.

Lemma 2.1. *Algorithm* SampleTriangle *outputs a value β with expected value*

$$\mathbf{E}[\beta] = \frac{3|T_3|}{|T_1| + 2\cdot|T_2| + 3\cdot|T_3|}$$

Furthermore $|T_3| = \mathbf{E}[\beta]\cdot|E|\cdot(|V|-2)/3$.

Proof. We look at all triples of nodes in V. Each triple belongs to one of the sets T_0, T_1, T_2, or T_3. The algorithm chooses such a triple by choosing an edge $e = (a, b)$ together with one node $v \in V \setminus \{a, b\}$. Therefore, no triple from T_0 is chosen.

Let us denote by $t = \{v, w, u\}$ a fixed triple from T_1. Wlog. let $(v, w) \in E$ and so $(v, u), (w, u) \notin E$. The algorithm chooses t, iff it samples edge (v, w) and vertex u.

Now assume $t \in T_2$. Then t is chosen by SampleTriangle, iff one of the two edges in the triple is sampled and v equals to the remaining node of the triple. For the same reason, a triple in T_3 is counted whenever one of its three edges and the remaining vertex is chosen. Since there are $|E|\cdot(|V|-2) = |T_1| + 2\cdot|T_2| + 3\cdot|T_3|$ choices for the algorithm to sample an edge and a node, it follows $|T_3| = \mathbf{E}[\beta]\cdot|E|(|V|-2)/3$. □

A streaming algorithm which outputs a good estimate of T_3 easily follows: we start

$$s \geq \frac{3}{\epsilon^2}\cdot\frac{|T_1| + 2\cdot|T_2| + 3\cdot|T_3|}{|T_3|}\cdot\ln(\frac{2}{\delta})$$

parallel instances of SampleTriangle. Each of these instances outputs a value β_i. We use $\frac{1}{s}\sum_{i=1}^{s}\beta_i$ as an estimator for
$\mathbf{E}[\beta] = \frac{3\cdot|T_3|}{|T_1|+2\cdot|T_2|+3\cdot|T_3|}$. We then return

$$\widetilde{T_3} := \left(\frac{1}{s}\sum_{i=1}^{s}\beta_i\right)\cdot|E|(|V|-2)/3$$

as an estimation for the cardinalty of T_3.

Lemma 2.2. *With probability $1-\delta$ the following statement holds:*

$$(1-\epsilon)\cdot|T_3| < \widetilde{T_3} < (1+\epsilon)\cdot|T_3|$$

Proof. We use Chernoff's Bounds:

$$\begin{aligned}
&\mathbf{Pr}\big[\frac{1}{s}\sum_{i=1}^{s}\beta_i \geq (1+\epsilon)\mathbf{E}[\beta]\big] \\
< \;& e^{-\epsilon^2\cdot\mathbf{E}[\beta]\cdot s/3}\mathbf{Pr}\big[\frac{1}{s}\sum_{i=1}^{s}\beta_i \leq (1-\epsilon)\mathbf{E}[\beta]\big] \\
< \;& e^{-\epsilon^2\cdot\mathbf{E}[\beta]\cdot s/2}
\end{aligned}$$

For $s \geq \frac{3}{\epsilon^2}\cdot\frac{|T_1|+2\cdot|T_2|+3\cdot|T_3|}{T_3}\cdot\ln(\frac{2}{\delta})$ the sum of both probabilities is bounded by δ. The lemma follows now from Lemma 2.1 stating that $|T_3| = \mathbf{E}[\beta]\cdot|E|\cdot(|V|-2)/3$. □

Our next step is to consider the update time of our implementation. If we implement the different instances of our algorithm independently of each other, we require $O(\frac{1}{\epsilon^2}\cdot\log(\frac{1}{\delta})\cdot(1+\frac{|T_1|+|T_2|}{|T_3|}))$ time to process each edge during the third pass. We show how to reduce this to expected constant time. Before we invoke the third pass, we collect all edge-vertex pairs chosen by different instances of the algorithm. For each pair with edge $e = (a, b)$ and vertex v we would like to find out whether (a, v) and (b, v) are in E. Therefore, we construct a set M of *missing edges* that for each such edge-vertex pair contains the edges (a, v) and (b, v). Next, we construct a hash table for M using a uniform hash function that requires linear space, as proposed in [13]. Now we can

implement the third pass in the following way. For each edge e, we lookup whether it is in the set M. If $e \in M$ we mark it. These steps can both be done in expected constant time. In a postprocessing step we can then determine the edge-vertex pairs that are triangles.

We summarize our result in the following theorem. We remark that we significantly improve the update time over the previously best result from [8] while achieving the same space complexity. The update time in [8] is roughly proportional to the space complexity compared to expected constant time for our algorithm. Since our experiments show that one still needs a relatively large amount of space this is a significant improvement over the previous results.

THEOREM 1. *There is a 3-Pass streaming algorithm to count the number of triangles in a stream of edges up to a multiplicative error of $1 \pm \epsilon$ with probability at least $1 - \delta$, which needs $O(\frac{1}{\epsilon^2} \cdot \log(\frac{1}{\delta}) \cdot (1 + \frac{|T_1|+|T_2|}{|T_3|}))$ memory cells and constant expected update time.* □

2.2 A 1 Pass Algorithm

In this section we show that the previous 3-pass algorithm can be implemented in one pass using the same amount of space and constant expected amortized update time, if $|E|$ is significantly larger than the number of instances we run.

We first observe that we can find a random edge in one pass by reservoir sampling [17], i.e. choosing the first edge as a sample edge and replacing this edge by the ith edge of the stream with probability $1/i$. It is known that this method can be implemented in $O(\log |V|)$ expected time per sample (not counting the time to read the stream) by randomly choosing the next index of the replacing edge according to an appropriate probability distribution.

When we combine this with the third pass, it may happen that we sample an edge $e = (a, b)$ of the stream together with a node v, but we do not see the edge (a, v) or (b, v) in the subsequent stream (because they appeared before the edge e). In this case, we do not detect a, b, v as a triangle. However, we detect a, b, v, iff (a, b) is the first edge of the triangle that appears in the stream. This changes the expected value of β by a factor of 3.

SAMPLETRIANGLEONEPASS
  $i \leftarrow 1$
  for each edge $e = (u, w)$ in the stream **do**
    Flip a coin. With probability $1/i$ **do**
      $a \leftarrow u$; $b \leftarrow w$;
      $v \leftarrow$ Node uniformly chosen from $V \setminus \{a, b\}$
      $x \leftarrow$ false; $y \leftarrow$ false
    end do
    if $e = (a, v)$ **then** $x \leftarrow$ true
    if $e = (b, v)$ **then** $y \leftarrow$ true
    $i \leftarrow i + 1$
  end for
  if $x =$ true $\wedge\, y =$ true **then** return $\beta \leftarrow 1$ **else** return $\beta \leftarrow 0$

LEMMA 2.3. *Algorithm* SAMPLETRIANGLEONEPASS *outputs a value β having expected value*

$$\mathbf{E}[\beta] = \frac{|T_3|}{|T_1| + 2 \cdot |T_2| + 3 \cdot |T_3|} \ .$$

The Proof is similar to the proof of Lemma 2.1 □

We now start

$$s \geq \frac{3}{\epsilon^2} \cdot \frac{|T_1| + 2 \cdot |T_2| + 3 \cdot |T_3|}{|T_3|} \cdot \ln(\frac{2}{\delta})$$

parallel instances and return the value

$$\widetilde{T_3} := \left(\frac{1}{s} \sum_{i=1}^{s} \beta_i \right) \cdot |E| \cdot (|V| - 2)$$

This is a $(1+\epsilon)$-approximation of $|T_3|$.with probability at least $(1 - \delta)$

By applying the reservoir sampling algorithm from [17] to select the edge, the selection requires $O(\log |V|)$ expected time for each instance of SAMPLETRIANGLEONEPASS. Additionally we use the hash table approach from the previous chapter to efficiently find instances of SAMPLETRIANGLEONEPASS which search for an edge in the stream. Alltogether we get expected $O(1 + s \cdot \frac{\log |E|}{|E|})$ update time per edge in the stream.

THEOREM 2. *There is a 1-Pass streaming algorithm to count the number of triangles in a stream of edges up to a multiplicative error of $1 \pm \epsilon$ with probability at least $1 - \delta$, which needs $O(s)$ memory cells and expected update time $O(1 + s \cdot \frac{\log |E|}{|E|})$, where $s \geq \frac{3}{\epsilon^2} \cdot \frac{|T_1|+2\cdot|T_2|+3\cdot|T_3|}{|T_3|} \cdot \ln(\frac{2}{\delta})$.*

3. INCIDENCE STREAMS

In an incidence stream all edges incident to the same vertex appear subsequently in the stream. That is, first arrive all edges incident to vertex v_1, followed by all edges incident to v_2, and so on. The ordering $v_1, \ldots, v_n$ of the vertices can be arbitrary, i.e. determined by an adversary. We consider undirected graphs and so each edge appears twice (within the incidence list of both incident nodes). There is no bound on the degree of the nodes (in contrast to [20]).

3.1 3 Pass Algorithm

We again will first develop a 3-pass algorithm, and later combine the passes to get a one pass algorithm. Let d_i denote the degree of node v_i. The 3-pass algorithm is presented below.

SAMPLETRIANGLE2
  1st. Pass:
    Count the number of paths of length 2 in the stream
  2nd. Pass:
    Uniformly choose one path using UNIFORMTWOPATH
    Let (a, v, b) be this path
  3rd. Pass:
    Test if edge (a, b) appears within the stream
  if $(a, b) \in E$ **then** $\beta = 1$
  else $\beta = 0$
  return β

We observe that the number of paths of length 2 is exactly $P := |T_2| + 3 \cdot |T_3| = \sum_{i=1}^{|V|} d_i \cdot (d_i - 1)/2$. Thus we can easily count the number of paths of length 2 by determining the degree of each node. This is possible because the edges appear as an incidence stream.

The second pass could be implemented using reservoir sampling. However, we propose a different approach which achieves slightly better amortized running time. The idea is

as follows: if v is incident to the nodes $w_1, w_2, ..., w_d$, we define an order for the possible triangles (v, w_i, w_j), $i < j$. We order the triangles by the last component first, i.e. in the following way: (v, w_1, w_2), (v, w_1, w_3), (v, w_2, w_3), (v, w_1, w_4), etc. We randomly choose a value of k and identify the k-th pair w_i and w_j with respect to this order by computing the two values i and j according to formulas given below (the formulas use the simple fact, that there are $j(j-1)/2$ triangles made of nodes $v, w_1, .., w_j$). The triple is chosen (if the node v is in the middle of enough two paths. Otherwise we search for the k-th two path within the next incidence list).

The algorithm is presented below.

UniformTwoPath
  Select value k uniformly from the set $\{1, \ldots, P\}$
  For each node v in the incidence list **do**
    If $k > 0$ **then**
      Set $j \leftarrow \left\lceil \sqrt{2k + \frac{1}{4}} + \frac{1}{2} \right\rceil$
      Set $i \leftarrow j - \frac{j^2 - j}{2} + k - 1$
      Pass over the complete incidence list of node v
      If incidence list of v contains more than j edges **then**
        $a \leftarrow$ the ith node in the incidence list of v
        $b \leftarrow$ the jth node in the incidence list of v
        $w \leftarrow v$
      end if
      $d \leftarrow$ degree of node v
      $k \leftarrow k - \frac{d^2 - d}{2}$
    end if
  end do
  return edges (w, a) and (w, b)

Lemma 3.1. *Algorithm* SampleTriangle2 *outputs a value β with expected value*

$$\mathbf{E}[\beta] = \frac{3 \cdot |T_3|}{|T_2| + 3 \cdot |T_3|}$$

Proof. We look at all triples of nodes in V. Each triple belongs to one of the sets T_0, T_1, T_2, or T_3. The algorithm chooses such a triple by choosing a node v together with two adjacent edges. Therefore the selected triples bekong to the set $T_2 \cup T_3$. We select a triple from T_2, if we choose the unique node adjacent to both edges and the corresponding edges. A triple from set T_3 can be chosen in three different ways by selecting one of the three nodes of the triple together with both adjacent edges. Since each choice of a path of length two has the same probability, the probability of choosing a triple in T_3 is exactly $3 \cdot |T_3| / (|T_2| + 3 \cdot |T_3|)$ as stated. □

We now start

$$s \geq \frac{3}{\epsilon^2} \cdot \frac{|T_2| + 3 \cdot |T_3|}{|T_3|} \cdot \ln(\frac{2}{\delta})$$

parallel instances of UniformTwoPath and return the value

$$\widetilde{T_3} := \left(\frac{1}{s} \cdot \sum_{i=1}^{s} \beta_i \right) \cdot \left(\sum_{i=1}^{|V|} d_i \cdot (d_i - 1) \right) / 6 \ .$$

This is with probability at least $(1-\delta)$ a $(1 \pm \epsilon)$-approximation of $|T_3|$.

To get small amortized expected update time we proceed as follows. Each time when the incidence list of a new vertex starts, we compute the values i and j for every instance. Then we insert the j-values into a global priority queue keeping a pointer to the corresponding instance. When we now process the incidence list of the current vertex we maintain a global counter for the number of neighbors of the current vertex we have seen. If this number is equal to the smallest value stored in the priority queue we remove it and process the corresponding instance. After the incidence list has been processed, we empty the priority queue. This way, each instance of the algorithm requires $O(1)$ time per vertex. Additionally, we need $O(s \cdot \log |V|)$ time to process the removal of the smallest element in the priority queue. Overall, the amortized cost of the second pass is $O(1 + s \cdot \frac{|V|}{|E|})$, which is constant for moderately large values of $|E|$. To implement the third pass we use hashing in a similar way as in the algorithm for adjacency lists. This leads to expected constant update time for the third pass.

Theorem 3. *There is a 3-Pass streaming algorithm to count the number of triangles in incidence streams upto a multiplicative error of $1 \pm \epsilon$ with probability at least $1 - \delta$, which needs $O(s)$ memory cells and amortized expected update time $O(1 + s \cdot \frac{|V|}{|E|})$, where $s \geq \frac{3}{\epsilon^2} \cdot \frac{|T_2| + 3 \cdot |T_3|}{|T_3|} \cdot \ln(\frac{2}{\delta})$.*

3.2 1 Pass Algorithm

To get a one pass algorithm we will again combine the passes of the algorithm stated before. The first pass only counts the number P of paths of length 2 in the graph. Instead of counting this number in advance, we will start $3 \log |V|$ instances of the streaming algorithm, an instance for each guess $\tilde{P}$ of the number of length-2-paths in the set $\{1, 2, 4, 8, ..., |V|^3\}$. In parallel we will count P. At the end we can find one instance started with a value $\tilde{P}$ satisfying $P \leq \tilde{P} < 2P$. We choose the result of this instance as the result of our algorithm.

If the estimated number of nodes was too high then this instance might not get a sample node at all; since the estimation of the number of nodes is at most twice the real value, this failure happens with probability at most $1/2$ and will be detected. By running $\log(1/\delta)$ parallel instances of the algorithm, with probability $1 - \delta$ at least one of these algorithms will succeed.

To combine the second and third pass we test all edges seen after drawing the sample. Therefore we miss an edge when the incidence lists of both endnodes appear earlier within the stream (i.e. the node v is the last of the triangle nodes). Since this happens with probability $1/3$, the expected value of β decreases by a factor of $1/3$.

Lemma 3.2. *Algorithm* SampleTriangleOnePass2 *outputs a value β having expected value*

$$\mathbf{E}[\beta] = \frac{2 \cdot |T_3|}{|T_2| + 3 \cdot |T_3|}$$

We now start $s \geq \frac{3}{\epsilon^2} \cdot \frac{|T_2| + 3 \cdot |T_3|}{|T_3|} \ln(\frac{2}{\delta})$ parallel instances and return the value

$$\widetilde{T_3} := \left(\frac{1}{s} \cdot \sum_{i=1}^{s} \beta_i \right) \cdot \left(\sum_{i=1}^{|V|} d_i \cdot (d_i - 1) \right) / 4$$

We use techniques to reduce the amortized update time as shown in the previous section. Since we start $O(\log V)$ parallel instances for different guesses of $\tilde{P}$, the amortized update time increases by a factor of $O(\log V)$.

THEOREM 4. *There is a 1-Pass streaming algorithm to count the number of triangles in incidence streams up to a multiplicative error of* $1 \pm \epsilon$ *with probability at least* $1 - \delta$*, which needs* $O(s \cdot \log |V|)$ *memory cells and amortized expected update time* $O(\log(|V|) \cdot (1 + s \cdot (\frac{|V|}{|E|})))$*, where* $s \geq \frac{3}{\epsilon^2} \cdot \frac{|T_2|+3\cdot|T_3|}{|T_3|} \ln(\frac{2}{\delta})$.

4. OPTIMIZED IMPLEMENTATIONS

In Section 2 and 3 we provided a general overview of the algorithms. This section describes some details of several optimization steps used in the implementation. We instead omit a detailed description of the use of memory blocks in order to improve I/O efficiency. Rather than reading from the hard disk edge by edge, a large amount of edges is read from secondary memory and stored in main memory in a block of fixed size. This also allows in the analysis to distinguish between processing time and reading time (the reading time sometimes is larger than the processing time).

We first present the three main methods used for speeding up the running times and then the algorithms that use such methods.

Hash Functions. We use hash functions to quickly find elements in a set. We use one or two elements as keys for our hash function. In case of two values u and v, we use hash functions of the form $h(u, v) = r_1 \cdot u + r_2 \cdot v \mod 2r$, where r_1 and r_2 are two random generated numbers between one and r (sample set size), and $u < v$. In case of only one element, we use hash functions of the form $h(u) = r_1 \cdot u \mod 2r$. The size of the hash table is $2 \cdot r$.

Uniform Sample Selection. We use the following method to select a sample set from a set RR (in our case RR will be the set of edges) of unknown size that is presented as a data stream. We would like to pick a sample of size r and its distribution should be (almost) uniform. The problem is that we do not know $|RR|$ in advance and we have to construct our sample set on the fly when we pass over the stream. We initially set $M = r$ and $m = 1$. Samples are selected with probability $\frac{1}{m}$ until the number of samples taken reaches M. When the number of seen elements reaches M, each sample is evicted from the current sample set with probability 1/2, and M and m are doubled. Samples selected in this fashion are almost uniformly distributed. This technique avoids to run different instances of the algorithm for different guesses of the size of the set RR.

Draw Sample. We next observe that it is not very efficient to throw a biased coin for every element in the data stream, since the inverse of the probability $1/m$ that an element is chosen is proportional to the length of the stream. Instead of throwing a coin we use the probability distribution of the position of the next element of the stream that is taken into our sample set. This distribution depends only on the current position in the stream and the value of m and can be calculated as follows. Assume we want to draw each sample with probability p. Then we choose a value α uniformly at random from $[0, 1]$. Supposing the last sample selected was the element at position k_i in the data stream, then the next sample will be the element with number $k_{i+1} = k_i + \lceil (\frac{\log \frac{\alpha-1}{p-1}}{\log(1-p)} + 1) \rceil$.

```
procedure OptimizedOnePassSampling-ArbOrder(r)
1   s ← 0; A ← 0;
2   x ← 1; m ← 1; M ← r;
3   for each e_s = (a_s, b_s) ∈ E do
4       a ← a_s;
5       b ← b_s;
6       A ← A + 1;
7       if A = x then
8           s ← s + 1;
9           S^s_a ← a;
10          S^s_b ← b;
11          S^s_v ← v uniformly chose from V \ {a, b};
12          S^s_count ← 0;
13          inserIntotHash(a,v,s,pt);
14          insertIntoHash(b,v,s,pt);
15          x = nextSample(1/m, s);
16      end if
17      if A = M then
18          M ← M · 2;
19          m ← m · 2;
20          s ← cleanHalfSampleSet(S);
21      end if
22      checkTri();
21  end for
22  β ← 0;
23  for each s ∈ S
24      if S^s_count = 2 then β ← β + 1;
25  T̃_3 ← (β/|S|) · |N| · |E|;
end procedure
```

Figure 1: Pseudo-code for the one pass algorithm considering an arbitrary list of edges.

4.1 One Pass Algorithm for Streams of Edges in Arbitrary Order

This section presents details of the implementation of the one pass algorithm considering an arbitrary order of the edges of an undirected graph (presented in the subsection 2.2). Figure 1 presents the pseudo-code of the algorithm, including the use of hashing functions, *draw sampling* and *random selection* techniques. Moreover, the algorithm considers the results for r samples, and not only one as presented before.

The main structures used by the algorithm are the sample set, a hash function table and S, a vector of pointers to structures of type sample. Lines 1 and 2 are initializations. The size s of the sample set and the number of edges seen are initialized with zero. The next sample x to be selected is set to the first one, while variables M and m used for the random sampling technique are set to one and r, the number of samples requested by the user.

The loop from lines 3 to 21 analyzes each edge read from the graph. The endpoints of the edge are identified in lines 3 and 4, and the counter of edges is incremented in line 6. If the current edge is the one indicated by the draw sampling technique (line 7), the sample is selected. The sample is composed by two edges (the missing edges) and a counter (lines 8 to 12) that it is used to later check if it is equal to two, indicating that both missing edges were later seen in the stream. Both edges are independently inserted in the

hash function, together with the number of the sample and a point to it. This information is used in the case that later a sample changes position or is removed from the sample set. Next, a new sample that later will be inserted in S is calculated for the draw sample technique (line 15). When the number of samples reaches M (line 17), the value of M and m are doubled and the sample set is cleaned to about half of it (lines 18 to 21). The procedure `cleanHalfSampleSet` removes each sample with probability 1/2. The sample set is rearranged to avoid having gaps, and their respective indices s are updated.

The procedure `checkTri` verifies if the current read edge (a,b) belongs to one or more current samples. The verification is done hashing a and b. For each occurrence of this arc found in the hash list, the correspondent counter is incrementing, indicating that one more of the missing edges were found.

Finally, in lines 23 and 24, the counters of the s samples is verified and in case it is two, it means that both missing edges were seen in the stream and the sample corresponds to a triangle. So the triangle counter β is incremented. The estimated number of triangles $\tilde{T}_3$ is calculated in line 25.

4.2 One Pass Algorithm for Incidence Streams

This section presents details on the optimized implementation of the one pass algorithm considering an incidence list of the edges of an undirected graph (presented in the subsection 3.2). Figure 2 presents the pseudo-code. that uses the described optimizations and considers r samples, instead of an unique sample.

The main structures used are the hash function and the sets S and D. The former is used similarly as in the previous algorithm, while D stores the current list of adjacent edges being read.

The initializations are similar to the ones used by the procedure `OptimizedOnePassSampling-Arbitrary` with the difference that instead of A, the set of edges seen so far, this algorithms uses P, the number of paths of length two seen so far. Moreover, in line 3, the current source node of the adjacent list of edges that is been read is reset to -1.

The loop from lines 4 to 26 analyzes the edges from the graph, one by one. The loop from lines 5 to 15 reads edges (a,b) (line 6 and 7), sets information of the corresponding node to the adjacency list being read (lines 8 to 11), adds the new node to the adjacency list (line 12), checks if this edge closes triangles from the current samples (line 13) and increments P (line 14). For each new edge that is seen, this arc will form a path length 2 with each of the previous edges from the current adjacency list.

When the value of P reaches x, the edge that forms a sample with the current edge read is calculated in line 17, edge (w, v) that forms a triangle with the length 2-path $w - a - v$ is inserted into the hash table (line 18) and a new sample index x is generated (line 19). This loop is executed while P is larger than x. As in the previous algorithm, loop from lines 21 to 25 certify that all samples are uniformly randomly selected. Finally, the number of samples that represent triangles is computed in line 27 and the expected number of triangles is computed in line 28.

The hash functions used in this case store a condensed list of edges, since a unique arc can close many triangles. Thus, each arc is inserted only once in the lists, and each element of the list stores a counter of the number of times this arc

```
procedure OptimizedOnePassSampling-Incidence(r)
1    s ← 0; P ← 0;
2    x ← 1; m ← 1; M ← r;
3    u ← NIL;
4    for each e_s = (a_s, b_s) ∈ E do
5        while P < x do
6            a ← a_s;
7            b ← b_s;
8            if a ≠ u then
9                D ← ∅;
10               u ← a;
11           end if
12           D ← D ∪ {b};
13           checkTri();
14           P ← P + |D| − 1;
15       end while
16       while P ≥ x do
17           w ← [|D| + x − P]-th element from D;
18           insertIntoHash(w,b);
19           x = nextSample(1/m, s);
20       end while
21       while x ≥ M do
22           M ← M · 2;
23           m ← m · 2;
24           s ← cleanHalfSampleSet(S);
25       end while
26   end for
27   β ← calculateT();
28   T̃_3 ← β·P / (3·|S|);
end procedure
```

Figure 2: Pseudo-code for the one pass algorithm considering an incidence list of edges.

was involved in a current sample. When checking if an edge closes triangles (line 13), it is recorded if it is the first or the second occurrence of it since each edge (u, v) is seen twice (in the incidence lists of u and v). Thus, for each arc that closes triangles in the sample set, three counters are stored, c_1, c_2 and c_3, indicating in how many samples the arc is involved when observing his first occurrence, in his second occurrence and after the second occurrence. These values are independently selected for being removed from the sample set (line 24), and the node from the list is exclude, if selected for that, only when $c_1 = c_2 = c_3 = 0$. Furthermore, when calculating the number of triangles (line 27), c_1 contributes twice to β, c_2 once and c_3 discarded. Even when extra tests are done to correctly maintain the condensed hash functions, the algorithm runs faster than not using these hash functions.

5. COMPUTATIONAL EXPERIMENTS

This section presents the computational experiments performed by running the one pass algorithms for estimating the number of triangles in the adjacency and incidence list models.

The codes were written in C/C++, and compiled with the `gcc` compiler version 3.2.2, using the `-O3` optimization option. The experiments were performed on a 2.4 GHz Intel Pentium IV computer with 512 MB of RAM, running

Linux, and compiled with g++ version 3.3.2. Due to space requirements, the experiments for the webgraph were performed in a 2.8 GHz Intel Pentium IV computer with 1 GB of RAM, running Linux. The implementations are available upon request.

CPU times were measured with the system function `getrusage`. The time for reading the graphs is not included in the running times that are reported. It is interesting to mention that, for both triangle counting algorithms (arbitrary and incidence list of edges), the typical running time of our experiments is larger than the reading time only when using 1,000,000 samples. On the other hand, the computation time is typically not much smaller than the reading time. Hence, it worths the effort to reduce the processing time of the algorithm.

We now describe the datasets used in the experiments.

5.1 Datasets

The datasets were divided in three subsets, all of them are comprised of real world instances.

The first subset is composed of only one instance `webgraph`. This instance is a webgraph of 135 million nodes and 1 billion edges obtained from a graph extracted in 2001 by the WebBase project at Stanford [21] by removing the frontier nodes, i.e, the nodes that have indegree equal to one and outdegree equal to zero.

The second set of instances is composed of instances used in the experiments reported in [15]. The instances are:

- `actor2002` and `actor2004`: based on the *Internet Movie Database*. In these instances, two actors (nodes) are connected if they ever stared together in a movie.
- `authors`: based on the *Computer Science Bibliography* at the University of Trier.
- `google-2002`: based on the 2002 Google contest.
- `itdk0304`: is the network of Internet routers (nodes) and their connections (edges) collected by the *Cooperative Association for Internet Data Analysis* (CAIDA).

The third set of instances is originated from the link structure of Wikipedia [11], from an old dump of June 13, 2004 [11]. Wikipedia is nowadays the largest online encyclopedia, available in more than 100 languages. In these graphs, each article is a node and each hyperlink between nodes identifies a directed arc. A graph is extracted from each language. The experiments were performed considering the graphs `wikiEN`, `wikiDE`, `wikiFR`, `wikiES`, `wikiIT` and `wikiPT`, extracted from the English, German, French, Spanish, Italian and Portuguese languages, respectively.

The graphs are mostly sparse, and the dimensions vary from less than ten thousand nodes to more than half million, and from less than 100 thousand edges to more than 50 million edges.

5.2 One Pass Algorithm for Streams of Edges in Arbitrary Order

In this section we report the results for the optimized version of the one pass algorithm for streams of edges presented in arbitrary order. Table 1 presents results for sample sizes r of 10,000, 100,000 and 1,000,000. The table contains five main columns, corresponding to the graph name, the results for each of the sample sizes $r = 10,000$, $r = 100,000$, and $r = 1,000,000$, the value $\frac{T_3}{T_1+2T_2+3T_3}$ computed for each graph instance. For each sample size, three runs were executed with each instance. We report for each run the estimated number of triangles ($\tilde{T_3}$), the quality of the result `Qlt(%)` and the corresponding running time. The quality of the result is measured as the percentage deviation from the optimal, i.e. , $\frac{Opt-\tilde{T_3}}{Opt} \cdot 100$. Thus, a negative number in this context indicates that the estimated number of triangles is sub-estimated, while a positive number indicates a super estimation. If no one of the samples represents a triangle, we set "-" in place of the percentage deviation. We used the algorithm from [15] for computing in main memory the exact number T_3 of triangles (we report on the optimal computation in [2]).

It can be observed in the table that no triangles were found in some of the runs for a sample set of 10,000 samples. For all instances, but one, at least in one of the three runs one or more triangles were identified for sample size of 100,000 and 1,000,000 samples. The instance `google-2002` is actually a very sparse one and it has just a few triangles. The probability of one sample to be a triangle is actually only 0,00000023 (as reported in the fifth column of table 1). was found, even considering a sample set of 1,000,000. Results are not presented for the webgraph since no triangles were found in all the experiments we performed with our first algorithm.

The computation time never exceeds 10 seconds in any of the experiments for $\leq$ 10,000 samples. Time grows for large sample sets, but its increase is not proportional to the increase of the sample set. Note that the final sample size is about 25% smaller than r because, when the sample set reaches r, about half of the sample set is removed. So, the sample size is between $r/2$ and r, in all cases. The average sample sizes for r equal to 10,000, 100,000 and 1,000,000 were 7684, 61676 and 571065, respectively. Similar numbers were found for the incidence list algorithm.

5.3 One Pass Algorithm for Incidence Streams

We present the results for the optimized version of the one pass algorithm for incidence sstreams. Table 2 presents results for a sample size of 10,000, 100,000 and 1,000,000 samples. The fifth column of the table reports the values $\frac{2T_2}{T_2+3T_3}$ for the considered graph instances.

As expected, the number of triangles found in all sample sets is much larger then the numbers presented by the algorithm for streams of edges in arbitrary order. For all runs of all instances, considering the three sample sizes presented, and also a sample size of 1,000 (not presented due to space restrictions), always one or more triangles were found in the sample. The average percentage deviation is very good, even for sample size of 1,000 samples. Considering the absolute values, the average percentage deviation for all instances, but `webgraph`, are 17.72%, 5.10%, 2.17% and 0.85% for the sample sizes of 1,000, 10,000, 100,000 and 1,000,000, respectively.

We consider that an approximation of 5% is a very good estimative, and so, for this algorithm, a sample set of size 10,000 provides already good results.

For the reading times, we observe that reading `wikiPT` takes 0.07 seconds, while for reading `actor2004` takes 58.02 seconds. For a single list of the arcs E_{UnD}, it is spent in reading time half of the time spent for E_{inc}.

Table 1: Results for the one pass algorithm for counting triangles in an undirected graph with edges listed in arbitrary order. Samples of sizes of 10,000, 100,000 and 1,000,000 were considered.

Graph	r=10,000			r=100,000			r=1,000,000			$\frac{T_3}{T_1+2T_2+3T_3}$
	$\tilde{T_3}$	Qlt(%)	Time	$\tilde{T_3}$	Qlt(%)	Time	$\tilde{T_3}$	Qlt(%)	Time	
actor2004	2,724,294,731	131.54	9.10	1,023,579,237	-13.01	12.45	1,152,018,942	-2.09	33.64	6.38977e-05
	5,410,963,294	359.88	6.78	340,984,576	-71.02	12.77	1,089,794,401	-7.38	21.64	
	0	-	9.24	1,375,682,338	16.92	12.81	1,151,929,523	-2.10	21.64	
google-2002	0	-	0.24	0	-	0.65	0	-	0.80	0.02316e-05
	0	-	0.19	0	-	0.64	0	-	0.71	
	0	-	0.24	0	-	0.61	0	-	0.69	
actor2002	1,561,913,432	350.36	6.23	392,150,057	13.07	13.28	299,102,547	-13.76	20.95	6.03383e-05
	0	-	3.83	491,075,426	41.60	7.41	329,973,292	-4.86	14.27	
	0	-	5.60	293,726,779	-15.31	7.39	372,567,786	7.43	14.34	
authors	0	-	0.36	4,932,982	196.19	0.94	923,913	-44.53	1.47	0.65034e-05
	0	-	0.28	0	-	0.93	923,913	-44.53	1.32	
	0	-	0.38	4,890,123	193.62	0.92	1,539,855	-7.54	1.30	
itdk0304	12,378,632	2620.20	0.28	3,089,228	578.86	0.69	384,488	-15.51	1.14	0.38865e-05
	0	-	0.22	0	-	0.72	384,488	-15.51	0.91	
	0	-	0.33	0	-	0.73	384,488	-15.51	0.91	
wikiEN	0	-	1.73	43,383,355	120.73	4.82	18,993,730	-3.36	9.48	1.13561e-05
	0	-	1.27	65,255,967	232.02	2.92	24,457,688	24.44	7.15	
	0	-	1.78	0	-	2.96	19,029,320	-3.18	7.12	
wikiDE	0	-	0.76	7,457,654	-7.69	1.52	6,744,665	-16.52	4.25	3.25277e-05
	0	-	0.55	14,966,638	85.25	1.54	7,663,978	-5.14	3.36	
	0	-	0.83	11,168,455	38.24	1.54	10,217,203	26.47	3.40	
wikiFR	0	-	0.28	3,102,916	-1.90	0.72	2,622,207	-17.10	1.05	11.47360e-05
	5,496,153	73.76	0.21	4,813,794	52.19	0.76	2,923,116	-7.59	0.87	
	0	-	0.29	2,744,281	-13.24	0.71	2,966,103	-6.23	0.88	
wikiES	1,745,767	119.19	0.12	547,117	-31.31	0.40	763,336	-4.16	0.40	10.73930e-05
	887,301	11.40	0.12	545,260	-31.54	0.41	817,860	2.69	0.34	
	4,460,671	460.06	0.15	436,669	-45.17	0.39	790,598	-0.74	0.38	
wikiIT	206,411	-37.12	0.10	210,023	-36.02	0.24	223,244	-31.99	0.23	16.08740e-05
	211,966	-35.43	0.08	314,594	-4.16	0.23	354,564	8.01	0.21	
	645,909	96.76	0.10	289,070	-11.94	0.25	420,224	28.01	0.20	
wikiPT	68,852	-1.70	0.05	69,160	-1.26	0.04	34,580	-50.63	0.06	16.94030e-05
	70,548	0.72	0.04	86,450	23.42	0.06	43,225	-38.29	0.04	
	0	-	0.05	77,805	11.08	0.06	121,030	72.79	0.06	

6. CONCLUDING REMARKS

We have proposed a methodology based on random sampling for counting the number of cliques in data streams. The algorithms provably achieve an arbitrarily good approximation with high probability, use a limited amount of samples and memory cells and constant or at most logarithmic processing time per edge of the graph. We have also detailed some of the features of our optimized implementation of the algorithms for counting the number of triangles in a graph, and reported experimental results on networks from several practical domains. The experimental results show that our algorithms achieve an excellent estimation of the exact values with $10,000$ samples in the incidence list model, and with $1,000,000$ samples in the model in which the graph is presented as list of edges in arbitrary order. These results are independent from the size of the networks and only depend on the frequency of the subgraphs. We plan to adapt this methodology to mining frequent patterns occurring in protein networks, community detection in the Web, construction of indexes in graph databases.

Acknowledgements. We are very thankful to Thomas Schank and Dorothea Wagner for making available their code for counting triangles optimally and putting five of the intances they have used in their paper at our disposal. We also thank A. Broder and S. Muthukrishnan for several helpful discussions.

7. REFERENCES

[1] Noga Alon, Yossi Matias, and Mario Szegedy, *The space complexity of approximating the frequency moments.*, J. Comput. Syst. Sci. **58** (1999), no. 1, 137–147.

[2] L. Buriol, G. Frahling, S. Leonardi, A. Marchetti-Spaccamela, and C. Sohler, *Counting graph minors in data streams*, 2005, DELIS technical report, http://delis.upb.de/paper/DELIS-TR-0245.pdf.

[3] Don Coppersmith and Shmuel Winograd, *Matrix multiplication via arithmetic progressions.*, J. Symb. Comput. **9** (1990), no. 3, 251–280.

[4] Anna C. Gilbert, Yannis Kotidis, S. Muthukrishnan, and Martin Strauss, *Surfing wavelets on streams: One-pass summaries for approximate aggregate queries.*, Proc. of the 27th VLDB, 2001, pp. 79–88.

[5] Sudipto Guha, Nick Koudas, and Kyuseok Shim, *Data-streams and histograms*, ACM Symposium on Theory of Computing, 2001, pp. 471–475.

[6] Frank Harary and Helene J. Kommel, *Matrix measures for transitivity and balance*, Journal of Mathematical Sociology **6** (1979), 199–210.

[7] M. Henzinger, P. Raghavan, and S. Rajagopalan, *Computing on data streams*, 1998, Tech.note 1998-011, Digital Systems Research Center, Palo Alto, CA.

[8] Hossein Jowhari and Mohammad Ghodsi, *New streaming algorithms for counting triangles in graphs.*, Proc. of the 11th COCOON, 2005, pp. 710–716.

Table 2: Results for the one pass algorithm for counting triangles in an undirected graph structured as an incidence list. Samples of sizes of 10,000, 100,000 and 1,000,000 were considered.

Graph	r=10,000			r=100,000			r=1,000,000			$\frac{2T_3}{T_2+3T_3}$
	$\tilde{T_3}$	Qlt(%)	Time	$\tilde{T_3}$	Qlt(%)	Time	$\tilde{T_3}$	Qlt(%)	Time	
webgraph	7,991,057,264	-	153.78	7,541,370,749	-	393.78	7,993,479,298	-	490.56	
	6,461,924,928	-	153.55	7,384,193,673	-	392.20	8,097,287,808	-	490.00	
	9,977,868,646	-	153.69	8,337,706,066	-	393.92	7,591,170,489	-	491.28	
actor2004	1,127,610,593	-4.16	12.29	1,155,564,261	-1.79	33.28	1,181,693,982	0.43	35.84	0.174932
	1,111,095,851	-5.57	12.52	1,192,599,566	1.36	20.28	1,177,782,402	0.10	35.11	
	1,177,449,181	0.07	12.12	1,175,270,762	-0.11	20.30	1,178,307,250	0.14	85.48	
google-2002	43,353	-1.22	0.28	45,489	3.65	1.20	44,765	2.00	4.97	0.004922
	45,293	3.20	0.28	45,435	3.52	1.00	43,704	-0.42	4.85	
	37,346	-14.91	0.27	42,420	-3.34	0.99	44,208	0.73	7.55	
actor2002	344,973,896	-0.53	6.70	345,817,151	-0.29	11.93	347,151,238	0.10	24.36	0.110693
	351,507,109	1.35	6.59	347,683,085	0.25	12.03	345,810,766	-0.29	24.38	
	330,775,554	-4.62	6.62	344,359,433	-0.71	12.00	347,532,178	0.21	55.16	
authors	1,636,611	-1.73	0.43	1,665,394	-0.01	1.21	1,670,148	0.28	4.47	0.227631
	1,586,971	-4.71	0.44	1,648,484	-1.02	1.19	1,665,792	0.02	4.45	
	1,633,188	-1.94	0.44	1,650,487	-0.90	1.20	1,664,291	-0.07	6.86	
itdk0304	458,517	0.76	0.33	449,558	-1.21	1.24	457,604	0.56	4.58	0.040506
	399,317	-12.25	0.34	458,260	0.70	1.11	451,481	-0.79	4.44	
	438,002	-3.75	0.34	453,440	-0.36	1.11	451,358	-0.81	6.40	
wikiEN	21,099,883	7.35	2.19	20,693,869	5.29	5.34	19,938,256	1.44	16.73	0.003876
	17,713,801	-9.87	2.21	20,206,714	2.81	4.78	19,894,603	1.22	16.78	
	20,695,192	5.30	2.19	17,977,501	-8.53	4.78	19,414,246	-1.22	26.72	
wikiDE	7,524,028	-6.87	0.91	8,265,424	2.31	3.24	8,120,882	0.52	10.54	0.027802
	8,327,148	3.07	0.89	8,213,376	1.66	2.44	8,080,158	0.01	10.54	
	8,114,584	0.44	0.94	8,162,754	1.04	2.45	8,024,967	-0.67	16.43	
wikiFR	3,060,821	-3.23	0.34	3,255,383	2.92	1.45	3,125,790	-1.18	7.67	0.038523
	3,476,882	9.92	0.34	3,199,530	1.15	1.29	3,125,613	-1.18	7.61	
	3,447,016	8.98	0.34	3,206,780	1.38	1.28	3,138,100	-0.79	10.63	
wikiES	863,765	8.45	0.18	782,798	-1.72	0.94	793,282	-0.40	5.09	0.042708
	791,437	-0.63	0.18	774,447	-2.76	0.90	800,619	0.52	5.09	
	768,999	-3.45	0.18	827,132	3.85	0.87	803,774	0.92	6.85	
wikiIT	339,404	3.39	0.12	313,241	-4.58	0.75	337,843	2.92	4.16	0.038986
	318,664	-2.92	0.12	308,480	-6.03	0.74	330,290	0.62	4.11	
	305,763	-6.85	0.12	339,498	3.42	0.73	322,894	-1.64	5.53	
wikiPT	70,699	0.94	0.07	70,443	0.57	0.53	70,942	1.28	2.63	0.026090
	62,620	-10.60	0.07	71,136	1.56	0.53	72,329	3.26	2.58	
	80,752	15.29	0.07	69,568	-0.68	0.53	69,203	-1.20	3.32	

[9] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, D. Sivakumar, Andrew Tomkins, and Eli Upfal, *Random graph models for the web graph.*, Proc. of the 41st FOCS, 2000, pp. 57–65.

[10] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew Tomkins, *Trawling the web for emerging cyber-communities.*, Computer Networks **31** (1999), no. 11-16, 1481–1493.

[11] S. Leonardi S. Millozzi L.S. Buriol, D. Donato, *Link and temporal analysis of wikigraphs*, Technical Report (2005).

[12] S. Muthukrishnan, *Computing on data streams*, 2005, athos.rutgers.edu/ muthu/stream-1-1.ps.

[13] Anna Ostlin and Rasmus Pagh, *Uniform hashing in constant time and linear space*, STOC '03: Proceedings of the thirty-fifth annual ACM symposium on Theory of computing (New York, NY, USA), ACM Press, 2003, pp. 622–628.

[14] Shalev Itzkovitz Nadav Kashtan Dmitri Chklovskii Ron Milo, Shai Shen-Orr and Uri Alon, *Network motifs: Simple building blocks of com- plex networks*, Science **298** (2002), no. 509, 824 – 827.

[15] Thomas Schank and Dorothea Wagner, *Finding, counting and listing all triangles in large graphs, an experimental study.*, WEA, 2005, pp. 606–609.

[16] Dennis Shasha, Jason Tsong-Li Wang, and Rosalba Giugno, *Algorithmics and applications of tree and graph searching.*, Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 2002, pp. 39–52.

[17] Jeffrey S. Vitter, *Random sampling with a reservoir*, ACM Trans. Math. Softw. **11** (1985), no. 1, 37–57.

[18] Duncan J. Watts and Steven H. Strogatz, *Collective dynamics of small- world networks*, Nature **393** (1998), 440–442.

[19] Xifeng Yan, Philip S. Yu, and Jiawei Han, *Graph indexing: A frequent structure-based approach.*, SIGMOD Conference, 2004, pp. 335–346.

[20] D. Sivakumar Z. Bar-Yosseff, R. Kumar, *Reductions in streaming algorithms, with an application to counting triangles in graphs*, Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms (2002), 623–632.

[21] J. Zhao, *An implementation of min-wise independent permutation family*, (2005), http://www.icsi.berkeley.edu/ zhao/minwise/.