Efficient semi-streaming algorithms for local triangle counting in massive graphs

Luca Becchetti, Paolo Boldi, Carlos Castillo and Aristides Gionis

2007
Efficient semi-streaming algorithms for local triangle counting in massive graphs

Luca Becchetti
Universita di Roma "La Sapienza"
Rome, Italy
luca.becchetti@dis.uniroma1.it

Paolo Boldi∗
Università degli Studi di Milano
Milano, Italy
boldi@dsi.unimi.it

Carlos Castillo
Yahoo! Research
Barcelona, Spain
chato@yahoo-inc.com

Aristides Gionis
Yahoo! Research
Barcelona, Spain
gionis@yahoo-inc.com

ABSTRACT

In this paper we study the problem of local triangle counting in large graphs. Namely, given a large graph \( G = (V, E) \) we want to estimate the number of triangles incident to every node \( v \in V \) in the graph. Even though the problem of computing the total number of triangles in a graph has been considered before, to our knowledge this is the first paper that addresses the problem of local triangle counting with a focus on the efficiency issues arising in massive graphs. We motivate our problem definition by showing how the distribution of the local number of triangles, and the local clustering coefficient, can be used in many interesting applications. For example, we show that the measures we compute can help to detect the presence of spamming activity in large-scale Web graphs, as well as to provide useful features to assess content quality in social networks.

For computing the local number of triangles we propose two approximation algorithms, which are based on the idea of min-wise independent permutations (Broder et al. 1998). Our algorithms operate in a semi-streaming fashion, using \( O(|V|) \) space in main memory and performing \( O(\log |V|) \) sequential scans over the edges in the graph. Our first algorithm uses \( O(|E|) \) space in external memory during computation, while our second algorithm uses only main memory. We present the theoretical analysis as well as experimental results in large graphs demonstrating the practical efficiency of our approach.

∗Partially supported by the MIUR COFIN Project “Linguaggi formali e automi” and by the EU Project DELIS. Luca Becchetti was partially supported by EU Integrated Project IST-015964 AEOLUS and by Italian National FIRB Project “Tecnologia e Scienza per le reti di prossima generazione”, Project No. RBIN047XH9.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.


1. INTRODUCTION

Graphs are a ubiquitous data representation that is used to model complex relations in a wide variety of applications, including biochemistry, neurobiology, ecology, social sciences, and information systems. Defining new measures of interest on graph data and designing novel algorithms that compute or approximate such measures on large graphs is an important task for analysing graph structures that reveal their underlying properties.

In this paper we introduce and we study the problem of counting local number of triangles in large graphs. In particular, we consider undirected graphs \( G = (V, E) \), in which \( V \) is a set of nodes and \( E \) is a set of edges. For a node \( u \) we define \( S(u) \) to be the set of neighbors of \( u \), that is, \( S(u) = \{v \in V : e_{uv} \in E\} \), and let the degree of \( u \) be \( d_u = |S(u)| \). We are then interested in computing, for every node \( u \), the number of triangles incident to \( u \), defined as:

\[
T(u) = \frac{1}{2} |\{e_{uv} \in E : e_{uw} \in E, e_{uw} \in E\}|.
\]

The problem of counting triangles also translates into computing the local clustering coefficient (also known as transitivity coefficient). For a node \( u \), the local clustering coefficient is defined as \( \frac{2T(u)}{d_u(d_u-1)} \), that is, the ratio between the number of triangles and the largest possible number of triangles in which the node could participate.

Note that the problem of estimating the overall number of triangles in a graph has been studied already (see e.g., [3, 12]); here we deal with the problem of estimating the number of triangles of all the individual nodes in the graph simultaneously. We motivate our problem definition by showing how the local triangle computation can be used in a number of interesting applications.

Our first application involves spam detection: we show that the distribution of the local clustering coefficient can be an effective feature for automatic Web spam detection. In particular, we study the distribution of the local clustering coefficient and the number of triangles in large samples of the Web. Results show that these metrics, in particular the former, exhibit statistical differences between normal and spam pages and are thus suitable features for the automatic detection of spam activity in the Web.
We then apply our techniques to the characterization of content quality in a social network, in our case the Yahoo! Answers community. Following the suggestion of [36] that the type and quality of content provided by the agents is related to the degree of clustering of their local neighborhoods, we perform a statistical analysis of answers provided by users, studying the correlation between the quality of answers and the local clustering of users in the social network.

Additionally, we believe that efficient computation of the local number of triangles and local clustering coefficient can have a larger number of other potential applications, ranging from the analysis of social or biological networks [33] to the uncovering of thematic relationships in the Web [18].

For computing the local number of triangles we propose two approximation algorithms, which rely on well established probabilistic techniques to estimate the size of the intersection of two sets and the related Jaccard coefficient [8, 10, 11]. Our algorithms use an amount of main memory in the order of the number of nodes $O(|V|)$ and make $O(\log |V|)$ sequential scans over the edges in the graph.

Our first algorithm is based on the approach proposed in [9, 10, 11], which uses mini-wise independent hash functions to compute a random permutation of an ordered set. In our case, this is the (labeled) set of nodes in the graph. In practice, to increase efficiency, instead of hash functions we simply use a random number generator to assign binary labels to nodes. Doing this can in principle lead to collisions (i.e., we might have subsets of nodes with the same label). We provide a quantitative analysis of this approach, characterizing the quality of the approximation in terms of the Jaccard coefficient and the role of collisions. A similar analysis had been sketched in [8].

We then proceed to propose a second algorithm that maintains one counter per node in main memory—as opposed to the first algorithm, which requires one counter for each edge. In practice, our second algorithm allows to perform the computation in main memory, thus achieving a considerable speed up. In particular, the processing time is almost halved, while the accuracy is still comparable or sometimes even better than the first algorithm. This is achieved by using a new, simpler, linear function to approximate the Jaccard coefficient of two sets. As a theoretical contribution, we assess the performance of this second algorithm in the framework used to analyze the first one.

We support our findings and analysis by experimental results. In particular, we use our algorithms to estimate the distributions of the number of triangles and of the clustering coefficient in medium and large samples of the Web graph. To the best of our knowledge, this is the first time efficient (semi-streaming) approximation algorithms for counting triangles are described.

The rest of the paper is organized as follows. In the next section we review the related work and in Section 3 we introduce the model of computation and the notation that we will be using throughout the paper. Section 4 describes how to approximate the intersection of two sets using pairwise independent permutations, as described in [10]. Section 5 presents our first algorithm, and Section 6 the main-memory-only algorithm. The last section presents our conclusions and outlines future work.

2. RELATED WORK

Clustering and the distribution of triangles are important to quantitatively assess the community structure of social networks [33] or the thematic structure of large, hyperlinked document collections, such as the Web [18].

There has been work on the exact computation of the number of triangles incident to each node in a graph [1, 4, 27]. The brute-force algorithm for computing the number of triangles simply enumerates all $\binom{|V|}{3}$ triples of nodes, and thus it requires $O(|V|^3)$ time. A more efficient solution for the local triangle counting problem is to reduce the problem to matrix multiplication, yielding an algorithm with running time $O(|V|^{\omega})$, where currently $\omega \leq 2.376$ [16]. If, in addition to counting, one wants to list all triangles incident to each node in the graph, variants of the “node iterator” and “edge iterator” algorithms can be used. A description and an experimental evaluation of those “iterator” algorithms can be found in [34]; however, their running time is $O(|V|^d_{\text{max}})$ and $O|\sum_{v \in V} d_v^2|$, respectively. For the datasets we consider—very large number of nodes and high-degree nodes due to skewed degree distributions—such exact algorithms are not scalable, thus in this paper we resort to approximation algorithms.

In [15] the authors propose a streaming algorithm that estimates the number of triangles with high accuracy, using an amount of memory that decreases as the number of triangles increases. This result has been improved in [12]. We remark that, differently from [15, 12], in this paper we are interested in estimating the local clustering coefficient (and the number of triangles) for all vertices at the same time.

Min-wise independent permutations have been proposed by Broder et al. as a way to estimate the size of the intersection of two sets and the related Jaccard coefficient. Together with the technique of shingles they provide a powerful tool to detect near duplicates in large document collections and the Web in particular [11, 8, 9]. Implementing min-wise independent permutations is infeasible in practice, since they require exponential space [10]. In recent years, families of linear hash functions have been proposed that implement min-wise independent permutations approximately [26, 5]. As explained further in this paper, in order to save computational time we do not use hash functions directly, but rather a pseudo-random generator. This can bring to collisions, but we show that their impact is negligible in practice.

The probabilistic estimation techniques we use have been considered in the past to solve related problems. In [22], the authors use the techniques of shingles and linear hashing to discover subsets of Web pages that share significant subsets of their outlinks, thus extending and making the discovery of cyber-communities in the Web computationally more efficient, in the spirit of [29]. Finally, in [21], the authors apply similar techniques to produce indices of page similarity that extend SimRank [28].

3. PRELIMINARIES

3.1 Semi-streaming graph algorithms

Given the very large size of the data sets used in Web Information Retrieval, efficiency considerations are very important. For concreteness, the total number of nodes $N = |V|$ in the Web that is indexable by search engines is in the
order of $10^{10}$ [23], and the typical number of links per Web page is between 20 and 30.

This fact imposes severe restrictions on the computational complexity of feasible algorithmic solutions. A first approach to modeling these restrictions might be the streaming model of computation [25], which however imposes limitations that are too severe for the problem at hand. Instead, we focus on building algorithmic solutions whose space and time requirements are compatible with the semi-streaming model of computation [19, 17]. This implies a strong implication that the algorithm should perform at most $O(\log N)$ passes over the data stored on secondary storage.

For comparison, suppose we want to measure the number of triangles in a graph in a naïve way. This would imply loading the lists of neighbors of each node in the graph in main memory to be able to count the number of triangles directly. This would need $O(\log(|V|)|E|)$ bits of main memory which is impractical in general. As to this point, note that in many data sets arising in practice, in particular some of the ones we consider in the experiments, we have $|E| = \Omega(|V| \log |V|)$.

### 3.2 Counting triangles

Considered an undirected graph (possibly a symmetrized version of a Web graph) and a vertex $u$, denote by $S(u)$ the set of $u$’s immediate neighbors. Now notice that, for every edge $uv \in E$, the number of triangles to which both $u$ and $v$ belong is $|S(u) \cap S(v)|$. So, the overall number of triangles $u \in V$ is participating in is $\sum_{v \in S(u)} |S(u) \cap S(v)|$. As a result, the basic building block of our approach is an algorithm to estimate the size of the intersection of two sets.

In the next section, we revisit the general technique [10, 11, 8, 9] to estimate the Jaccard coefficient and thus the size of the intersection of two sets $A$ and $B$ defined over the same universe which we assume, without loss of generality, to be $[n] = \{0, \ldots, n-1\}$ and where $n = 2^k$ for some suitable $k$.

### 3.3 Datasets

We ran most of our experiments on three medium-sized crawls gathered by the Laboratory of Web Algorithmics, University of Milan (http://law.dsi.unimi.it/); the graphs were symmetrized and loops were not considered in the computations. We used the WebGraph framework [6] to manipulate the graphs in compressed form. The particular collections we used are listed in Table 1. Note that, at least for some of the collections we consider, $|E|$ is expected to grow as $\Omega(|V| \log |V|)$. Furthermore, consistently with the empirical observations in [30], the average number of edges per node increases over the years. The dataset UK-2006-05 is the crawl that was labeled by a team of volunteers for creating a Web spam collection [13] so we have labels of non-spam/spam for a large set of hosts in that collection. The distribution of the number of triangles in the smaller graph EU-2005 is shown in Figure 1 and follows a power-law.

In addition to the graphs from web crawls, we also used a subgraph from Yahoo! Answers (http://answers.yahoo.com/), a question-answering portal. In the graph, each node represents a user, and a link between two users indicates that one of the users has answered a question asked by the other user. In the system, users can choose among the answers received which one is the best answer, and in the graph, we have identified the users who provide a high proportion of “best answers” to the questions they answer.

![Figure 1: Distribution of the number of triangles in the EU-2005 graph.](image-url)

### 4. ESTIMATING SET INTERSECTION

Without loss of generality, we consider subsets of the universe $[n] = \{0, \ldots, n-1\}$. We measure the overlap of two sets using the Jaccard coefficient: $J(A, B) = \frac{|A \cap B|}{|A \cup B|}$.

A very simple and elegant technique to estimate the Jaccard coefficient has been proposed in several equivalent forms by Broder et al. [8, 9, 10, 11]. Assume we are able to choose
a permutation \( \pi(\cdot) \) mapping \([n]\) onto itself uniformly at random. For every \( X \subseteq [n] \), denote by \( \pi(X) \) the set of the images of elements in \( X \) when \( \pi(\cdot) \) is applied and let \( \min(\pi(X)) \) denote their minimum. Then it can be shown [9] that (i) for every \( a \in A \subseteq [n] \), \( P(a = \arg\min(\pi(A))) = 1/|A| \); (ii) for every \( A, B \subseteq [n] \), \( P(\min(\pi(A)) = \min(\pi(B))) = \mathbb{I}(A, B) \). This property immediately yields a technique to estimate \( J(A, B) \). The algorithm consists in performing \( m \) passes over the data. At each pass, one permutation \( \pi(\cdot) \) among the \( m! \) possible ones is picked uniformly at random and then \( \min(A) \) is computed and compared with \( \min(B) \). Whenever they match, a counter is updated. Let \( C_m \) be the counter’s value after \( m \) passes. Our estimation of \( J(A, B) \) is \( C_m/m \).

Unfortunately, generating permutations uniformly at random requires exponential space [10]. In practice, suitable families of linear hash functions are used (e.g. see [26, 5]).

In this paper, in order to increase the speed of computation, we use a slight modification of this approach, simply assigning random labels to the graph’s vertices. As long as labels are sufficiently random and collisions do not too frequent, we are able to approximate the Jaccard coefficient satisfactorily. In practice, we used the Mersenne Twister, a pseudo-random number generator described in [31] which is a fast generation algorithm for obtaining high-quality pseudo-random numbers.

Figure 2 describes the algorithm’s pseudo-code, which is exactly the standard one given for example in [9], except for the use of random labels. Table 2 summarizes the notation used in the algorithm’s pseudo-code.

**Require:** sets \( A, B \subseteq [n] \), integer \( m \), \( k \) bits
1: for \( i : 1 \ldots m \) do
2: for every \( j \in [n] \), set \( I(j) \) to a value drawn uniformly at random between 0 and \( 2^k - 1 \)
3: Compute \( \mathbf{L}(A) \) and \( \mathbf{L}(B) \)
4: if \( \mathbf{L}(A) = \mathbf{L}(B) \) then
5: \( \text{count} \leftarrow \text{count} + 1 \)
6: end if
7: end for
8: return estimate \( \approx (\text{count}/(\text{count} + m))(|A| + |B|) \)

**Figure 2:** Basic algorithm for the estimation of the intersection of two sets (see also Table 2).

The notation used in the algorithm is summarized in Table 2. Define the following variables: \( W_i = 1 \) and only if, in the \( i \)-th iteration, \( \mathbf{L}(A) = \mathbf{L}(B) \) and \( W = \sum_{i=1}^{m} W_i \). Set \( X = |A \cap B| \). So, our estimator of \( X \) is \( \hat{X} = W/(W + m)(|A| + |B|) \). In fact, the labeling step might assign the same label to multiple vertices. This means that, in each iteration of the algorithm above, the probability that \( \mathbf{L}(A) = \mathbf{L}(B) \) is not exactly equal to \( J(A, B) \), as would be the case if we used min-wise independent permutations [10]. For the sake of completeness, we show that, as long as labels are reasonably random, the trivial labeling scheme we use allows us to estimate \( J(A, B) \) with good accuracy, collisions having a negligible impact. This is stated in the next result, whose proof follows the lines of those given in [11, 8, 9]. We give it here for the sake of completeness, since it considers the role of collisions (an aspect only sketched in [9]), and because its proof provides the framework for the analysis of a faster, approximate algorithm described in Section 6. In order to improve the flow of the presentation, the proof is presented in the Appendix.

**Theorem 1.** For every \( \epsilon > 0 \) and for every number \( m \) of iterations:

\[
P(|X - \hat{X}| > \epsilon X) \leq 2e^{-\frac{2}{4m}J(A,B)} + \frac{m|A \cup B|}{2k - 1}.
\]

In practice, this result states that our estimation of \( |A \cap B| \) differs from the true value by more than a constant factor with a probability that exponentially decays with \( m \) and \( J(A, B) \), while the worst case impact of collisions is summarized in the second term, which is \( o(1) \) as long as \( k = \Omega(\log n + \log m) \), \( m \) typically being in the order of a few tenths.

In the next section, we describe how to apply the same techniques for estimating the number of triangles.

5. **ESTIMATING TRIANGLE COUNT**

In this section we describe an approximating algorithm for counting the number of triangles for each node in the graph. The idea is to compute an approximation \( \hat{T}(u) \) of the number of triangles \( T(u) \) for all vertices in the graph.

5.1 **Algorithm**

The algorithm for computing the number of triangles is written in pseudo-code in Figure 3 and explained in the next paragraphs.

The algorithm performs \( m \) passes. At the beginning of each pass \( p \), a new random vector \( h_p(\cdot) \) is created. Each pass consists of two reads of the graph. In the first read of the graph, at each node we store the minimum label among those of the neighbors of that node. In the second read of the graph, we check, for each edge, if the two minima at the endpoints of the edge are equal; in such a case, one counter \( Z \) for each edge is increased.

After the \( m \) passes, an estimation of the number of triangles of each node is computed as:

\[
\hat{T}(u) = \frac{1}{2} \sum_{v \in S_u} \frac{Z_{uv}}{Z_{uv} + m} (|S(u)| + |S(v)|).
\]

The algorithm is feasible because the counters \( Z_{uv} \), which make most of the memory usage, are accessed sequentially and can be kept on secondary memory. The time complexity of the algorithm is \( O(|m|E) \). The main memory usage is \( O(k|V|) \) bits, basically for storing the node labels and the minima; a natural choice for \( k \) is \( k = \log(|V|) \). The secondary memory usage is \( O(\log(m)|E|) \) bits of temporary space which is less than the space required to store the graph in uncompressed form. The space required in secondary memory is read and written sequentially once for each pass.

Note that the quality of the approximation only depends on local properties of the graph, and does not vary as the graph grows in size. In particular, every term in the sum above has an accuracy that is described by Theorem 1, where \( A = S(u) \) and \( B = S(v) \). So, as stated in the previous section, the approximation improves with the number of passes, and it depends on the Jaccard coefficient so that for nodes with higher Jaccard coefficient the error is smaller.
between the approximation and the real value. We also measured the average relative error:

\[ \text{Error} = \frac{1}{N} \sum_{u} \left( \frac{T(u) - \hat{T}(u)}{T(u)} \right) \]

As a baseline approximation, we assume a constant clustering coefficient \( C \) in the graph, known in advance, and estimate the number of triangles of a node \( u \) as \( \hat{T}(u) = C^{2|S(u)|(|S(u)| - 1)/2} \). For two of the metrics we use for measuring the quality of the approximation below, the value of \( C \) is not relevant: Pearson’s correlation coefficient assumes a linear relationship and Spearman’s rank correlation coefficient is not affected by multiplicative factors.

Next we computed the distribution using our algorithm. For a fixed number of bits \( k \), the accuracy of the approximation increases with the number of passes. In Figure 4 we show the error of these approximations.

Already at 20 passes, involving only 40 sequential reads of the graph, the approximation has \( r \geq 0.90 \) and \( p \geq 0.90 \).

Looking at Spearman’s rank correlation, which is \( r \geq 0.85 \) with 50 iterations for our algorithm, we can see that the baseline algorithm provides a better approximation of the ordering of the nodes by number of triangles in IT-2004, EU-2005 and UK-2006-05. This fact indicates that the overall ordering is dominated by the degree of the nodes involved. However, the correlation coefficient of the baseline approximation is very low (below 0.5, and below 0.1 in UK and WebBase) while the correlation coefficient of the proposed algorithms is above 0.9.

Remark. For the sake of brevity, we only mention here that our algorithms, when applied to estimate the distribution of the number of triangles in the medium or large Web samples we consider, show that this follows a power law like in Figure 1, as observed in [18] for smaller samples.

6. ESTIMATING TRIANGLE COUNT IN MAIN MEMORY

This section describes a modification of previous algorithm that does not make use of external memory for the computation.

Observe that, in the final step of the algorithm presented in Section 5, we computed an estimation of the number of triangles of a node as:

\[ \hat{T}(u) = \frac{1}{2} \sum_{v \in S_u} \frac{Z_{uv}}{Z_{uv} + m(|S(u)| + |S(v)|)} \]

in which \( Z_{uv} \) is the number of minima that were the same between \( u \) and \( v \) during the \( m \) passes, \( 0 \leq Z_{uv} \leq m \).

To avoid the use of external memory, instead of keeping one counter for each edge, we can use one counter for each node, by approximating the number of triangles incident to a vertex \( u \) as:

\[ \hat{T}(u) = \frac{1}{2} \sum_{v \in S_u} \frac{Z_{uv}}{Z_{uv} + m(|S(u)| + |S(v)|)}. \]

The algorithm that uses this approximation is given in Figure 5 and it is explained in the next paragraphs. The proof that it estimates the triangle count with good accuracy is given in the next subsection.

This algorithm is similar in spirit to the one shown in Figure 3, but removing \( Z_{uv} \) from the denominator in the expression of \( \hat{T}(u) \) allows to maintain one counter per node instead of one counter per edge. The algorithm does \( m \) passes, each pass consisting of two reads of the graph. In the first read of the graph, at each node we store the minimum hash value of the neighbors of that node. In the second read of the graph, we check, for each edge, if the two minima at the endpoints of the edge \( (src, dest) \) are equal, and if so a per-node counter \( Z_{src} \) is increased by \(|S(src)| + |S(dest)|\).

After the \( m \) passes, an estimation of the number of triangles of each node is computed as:

\[ \hat{T}(u) = \frac{1}{2} \sum_{v \in S_u} \frac{Z_{uv}}{Z_{uv} + m}. \]

The time complexity of the algorithm is \( O(m|E|) \). The main memory usage is \( O(k|V|) \) bits, basically for storing
6.1 Analysis

We can give a result similar to that of Theorem 1. Namely, for \( u, v \in V \), set \( X = |S(u) \cap S(v)| \) and define \( W \) as in Section 4. In particular, \( W = \sum_{i=1}^{m} W_{i} \) with \( W_{i} = 1 \) if, during the \( i \)-th iteration of the algorithm, the if at line 14 of the algorithm of Figure 5 is true for nodes \( u \) and \( v \). Finally, define

\[
\overline{X} = \frac{W}{1.5m} (|S(u)| + |S(v)|).
\]

We have

**Theorem 2.**

\[
\Pr \left[ \left( \overline{X} > \frac{4}{3} (1 + \epsilon) X \right) \cup \left( \overline{X} < \frac{2}{3} (1 - \epsilon) X \right) \right] \leq 2 e^{-\frac{2m}{3} (|S(u)| + |S(v)|)} + \frac{m |S(u) \cup S(v)|}{2^m - 1}.
\]

**Proof.** Let the variables \( E, \overline{W}(i) \) and \( \overline{W} \) be defined as in the proof of Theorem 1 (see Appendix). We again have:

\[
X = |S(u) \cap S(v)| = \frac{E[\overline{W}]}{E[\overline{W}]} (|S(u)| + |S(v)|).
\]

Figure 4: Accuracy of the approximation of the number of triangles using the two algorithms described in the paper (external memory and main memory). Top: Pearson’s correlation coefficient. Middle: Spearman’s rank correlation coefficient. Bottom: average relative error.

the hash functions, minima, and the per-node counters. Secondary memory is accessed only to read the graph.
Require: graph \( G = (V, E) \), number of iterations \( m \), number of bits \( k \)
1: \( Z \leftarrow 0 \)
2: for \( p : 1 \ldots m \) do
   {This reads the graph \( 2m \) times}
3:   for \( u : 1 \ldots |V| \) do
   {Initialize node labels and min}
4:     \( h_p(u) \leftarrow -k \) random bits
5:     \( \min(u) \leftarrow +\infty \)
6:   end for
7:   for \( src : 1 \ldots |V| \) do
   {Compute minima}
8:     for all links from \( src \) to \( dest \) do
9:       \( \min(src) \leftarrow \min(\min(src), h_p(dest)) \)
10: end for
11: end for
12: for \( src : 1 \ldots |V| \) do
13:   {Compare minima}
14:     for all links from \( src \) to \( dest \) do
15:       if \( \min(src) = \min(dest) \) then
16:         \( Z_{src} \leftarrow Z_{src} + |S(src)| + |S(dest)| \)
17:     end if
18: end for
19: end for
20: for \( u : 1 \ldots |V| \) do
21:   \( \bar{T}(src) \leftarrow \frac{1}{m} Z_u \)
22: end for
23: return \( T(\cdot) \)

Figure 5: Algorithm for estimating the number of triangles of each node in main memory.

Set \( \hat{X} = (X \mid E = 0) \). Since \( 0 \leq E[W] \leq m \), it is easy to see that we have:

\[
\frac{2}{3} X \leq E[\hat{X}] \leq \frac{4}{3} X.
\]

Proceeding like in the proof of Theorem 1, we have:

\[
P\left(\hat{X} > \frac{4}{3}(1 + \epsilon)X\right) \cup \left(\hat{X} < \frac{2}{3}(1 - \epsilon)X\right)
\]

\[
< P\left(\frac{4}{3}(1 + \epsilon)X \cup \frac{2}{3}(1 - \epsilon)X\right) | E = 0
\]

\[
+ P[E > 0]
\]

\[
< P\left(\hat{X} > \frac{4}{3}(1 + \epsilon)X\right) + P\left(\hat{X} < \frac{2}{3}(1 - \epsilon)X\right)
\]

\[
+ \frac{m|S(u) \cup S(v)|}{2^{|u| - 1}}
\]

where the last inequality follows from the definition of \( \hat{X} \). Now:

\[
P\left[\hat{X} > \frac{4}{3}(1 + \epsilon)X\right] + P\left[\hat{X} < \frac{2}{3}(1 - \epsilon)X\right]
\]

\[
\leq P[\hat{X} - E[\hat{X}] > \epsilon E[\hat{X}]].
\]

where the inequality follows from the above given bounds on \( E[\hat{X}] \) in terms of \( X \). Recalling that, by definition, \( \hat{X} = W[(S(u) \cup S(v))/(1.5m)] \) we immediately have:

\[
P[\hat{X} - E[\hat{X}] > \epsilon E[\hat{X}] = P[\hat{W} - E[\hat{W}] > \epsilon E[\hat{W}]].
\]

The rest of the proof now proceeds exactly as in Theorem 1.

6.2 Experimental results

In practice, we observe that the second algorithm saves 40% to 60% of the running time. We ran the experiments for the large graphs in a quad-processor Intel Xeon 3GHz with 16GB of RAM. The wall-clock times required for \( m = 50 \) iterations we observed were:

<table>
<thead>
<tr>
<th>Graph</th>
<th>Nodes</th>
<th>Edges</th>
<th>(ext. mem.)</th>
<th>(main mem.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>WB-2001</td>
<td>118M</td>
<td>1.7G</td>
<td>10 hr 20 min</td>
<td>3 hr 40 min</td>
</tr>
<tr>
<td>IT-2004</td>
<td>41M</td>
<td>2.1G</td>
<td>8 hr 20 min</td>
<td>5 hr 30 min</td>
</tr>
<tr>
<td>UK-2006</td>
<td>77M</td>
<td>5.3G</td>
<td>20 hr 30 min</td>
<td>13 hr 10 min</td>
</tr>
</tbody>
</table>

The experimental results are included in Figure 4. Surprisingly, in many cases the accuracy of the main-memory algorithm is even better than the algorithm that uses secondary memory.

In the implementation, the number of bits necessary to store each counter depends on the number of iterations and on the link density of the graph. For instance, for WB-2001 we used a Java int (32-bits including the sign), but for IT-2004 and UK-2006, a long (64 bits including sign) was necessary to avoid overflow. We started to observe overflow after 60 passes in IT-2004 and after 20 passes in UK-2006. We point out that this is independent from the number of nodes in the graph.

7. APPLICATIONS

An efficient algorithm for local triangle counting is not only interesting as an algorithmic contribution. This section describes two applications of the algorithm for helping in information retrieval tasks in large graphs.

7.1 Detecting Web spam

Spam and non-spam pages exhibit different statistical properties, and this difference can be exploited for Web Spam Detection [20]. In this section we test if the number of triangles is a relevant feature for this task.

We used the WEBSPAM-UK2006 spam collection [13], a public Web Spam dataset annotated at the level of hosts. First we computed the number of triangles for each host in this dataset and plotted the distribution for the non-spam and spam hosts. This is shown in Figure 6. A two-tailed Kolmogorov-Smirnov test indicates that both the number of triangles and the clustering coefficient have distributions that are substantially different in both classes: the larger differences in the cumulative distribution function plot are \( D = 0.32 \) and \( D = 0.34 \) respectively.

We also compared the number of triangles and clustering coefficient with a known set of link-based and content-based features for the hosts in this collection [14]. We sorted all the features by computing the \( \chi^2 \)-squared statistics of each of them with respect to the class label. Using this ranking, the approximated number of triangles was ranked as feature number 60 out of 221, and the approximated clustering coefficient as feature number 14 out of 221; such remarkably high positions make both features well worth being tested as part of a spam detection system.

To complement these results, we estimated the number of triangles at a page level, and considered the average and maximum number of triangles in every host; in all cases we had to use the memory-based approximation algorithm to obtain the estimation, since an exact counting was in this case out of question. The results are shown in Figure 7. Also
in this case, a two-tailed Kolmogorov-Smirnov proved that the spam and non-spam distributions actually differ from each other: for example, the test in the case of average gave $D = 0.09$ with a p-value of $1.54 \cdot 10^{-7}$.

### 7.2 Content quality in social networks

In [36] it is shown that the amount of triangles in the self-centered social network of a user is a good indicator of the role of that user in the community.

Here we perform an exploration trying to verify whether the quality of content provided by a user in a social network is correlated with the local structure of the user in the network. For our dataset, we use a social network extracted from the Yahoo! Answers site. Yahoo! Answers is a community-driven knowledge sharing system that allows users to (i) ask questions on any subject and (ii) answer questions of other users. One notable characteristic of the system is that one answer for each question is selected as the best answer, and one of the user attributes is the fraction of the best answers given by that user.

We consider an undirected graph $G = (V, E)$, where $V$ is a set of users in the system, and an edge $(u, v) \in E$ denotes that the user $u$ has answered to a question posted by user the $v$, or vice versa. For this graph we apply our counting algorithms and we obtain an estimate of the number of triangles at each node, as well as the local clustering coefficient. We focus on a small subset of randomly chosen questions and answers which have been labeled by human judges as “high quality” or “normal”. These questions/answers have originated from a subset of about 9,500 users. Let $H \subseteq V$ be the subset of users who have provided a question or answer of high quality in our sample, corresponding to roughly 30% of the users in this case, and let $N = V \setminus H$ be the rest.

As a proof of concept, we first check if the fraction of best answers for the users differs between the sets $H$ and $N$. The two distributions are shown in Figure 8, in which one sees that users in the high quality set tend to have higher fraction of best answers. The two-tailed Kolmogorov-Smirnov difference of the two distributions is 0.26, and the null hypothesis is rejected with corresponding p-value equal to $1.1 \cdot 10^{-123}$.

Next we explore the correlation of local structure in the user graph with respect to the labeling of users in the classes $H$ and $N$. In particular, we examine if the distribution of the number of triangles and the distribution of the local clustering coefficient differs between the sets $H$ and $N$. The distributions in the case of the numbers of triangles are different. The Kolmogorov-Smirnov test rejects the null
The Kolmogorov-Smirnov difference is 0.27 and the p-value for rejecting the null hypothesis is $1.8 \cdot 10^{-19}$. On the other hand, using only the degree of each user in the graph is not sufficient to distinguish between the two distributions.

8. CONCLUSIONS

We have presented efficient semi-streaming algorithms for counting the local number of triangles in a large graph. To the best of our knowledge, these are the first such algorithms described in the literature. We believe that there are many applications for such algorithms to Web-scale problems, and we have demonstrated two such applications.

For future work, exploring variants of the first algorithm that relax the semi-streaming constraint but still use a small amount of memory is promising. Given that the distribution of the number of triangles is very skewed, the counters $Z_{uv}$ could be compressed. For instance, if the counters follow a power-law, a suitable coding could be used to store them (we may use Elias’ $\gamma$-coding if the distribution has exponent close to 2. $Z_{uv}$ [7] for smaller exponents or the codes recently proposed in [2] for larger ones). Note that each counter will use a variable number of bits depending on the value being stored. This may cause a drop in performance if done in external memory, but could be a good choice if done in main memory.

The Java code we used for computing the estimations, which implements the algorithms we have described, will be made available for the final version of this paper under a GPL license.

Acknowledgments: we thank Massimo Santini and Sebastiano Vigna for valuable comments and feedback about a preliminary version of this work.

9. REFERENCES

APPENDIX

Proof of Theorem 1

Definition 1. Let \([n] = \{1, \ldots, n\}\). A semi-permutation of \([n]\) is a map \(h : [n] \to [n]\).

Notice that we do not require injectivity: there might exist \(i \neq j\) such that \(h(i) = h(j)\).

For every \(A \subseteq [n]\) and \(i = 1, \ldots, m\), define by \(L^i(A)\) the minimum value of labels assigned to elements in \(A\) during the \(i\)-th iteration of the algorithm and by \(M^i(A \cup B)\) the set of elements achieving the minimum. We have the following

Lemma 1. Assume that during the \(i\)-th iteration each element in the set \([n]\) receives a label drawn uniformly at random between 0 and \(2^k - 1\). For all \(A, B \subseteq [n]\) we have

\[
\operatorname{Pr}[L^i(A) = L^i(B) | |M^i(A \cup B)| = 1] = J(A, B).
\]

Proof. Notice that, if \(|M^i(A \cup B)| = 1\), then \((L^i(A) = L^i(B))\) if and only if the minimum is achieved by some element in \(A \cap B\). Assuming an initial order \(\{a_1, \ldots, a_{|A \cup B|}\}\) for elements in \(A \cup B\) and assuming that labelling is described by a map \(m\), every semi-permutation is described by a vector of size \(|A \cup B|\), its \(i\)-th element being \(m(a_i)\).

For every element in \(a_j \in A \cup B\), define by \(\Psi(a_j)\) the set of semi-permutations such that \(i\) \(a_j \in M^i(A \cup B)\) and \(i\) \(M^i(A \cup B) = 1\). Furthermore, for every \(a_j, a_k \in A \cup B\), \(j \neq k\), we have, \(\Psi(a_j) \cap \Psi(a_k) = \emptyset\). For every \(a_j \in A \cup B\), \(j \neq k\), \(\Psi(a_j)\) is the set of values \(P\), by symmetry. Also, note that \(\Psi(a_j) \cap \Psi(a_k) = \emptyset\), for every \(a_j, a_k \in A \cup B\), \(j \neq k\). Hence, the overall number of semi-permutations such that there is one minimum and this minimum is achieved by an element in \(A \cap B\) is \(|A \cap B| \cdot P\).

By the same argument, \(|M^i(A \cup B)| = 1\). Finally, since semi-permutations are chosen uniformly at random:

\[
\operatorname{Pr}[L^i(A) = L^i(B) | |M^i(A \cup B)| = 1] = \frac{|A \cap B| \cdot P}{|A \cup B| \cdot P} = J(A, B).
\]

We will come back to the estimation of \(P\) later in this section.
In the sequel we will use the following, well known fact:

**Fact 1.** For every integer $n > 1$:

\[
(1 - \frac{1}{n})^n < \frac{1}{e} < (1 - \frac{1}{n+1})^n.
\]

We next define the following variables: $W_i = 1$ if and only if, in the $i$-th iteration, $\mathbf{L}'(A) = \mathbf{L}'(A)$ and $W = \sum_{i=1}^m W_i$. Define $E_i = 1$ if $(|\mathcal{M}'(A \cup B)| > 1)$, 0 otherwise and $E = \sum_{i=1} E_i$.

**Lemma 2.**

\[E[W|E=0] = mJ(A, B).\]

**Proof.** We have:

\[
E[W|E=0] = \sum_{i=1}^m P[W_i = 1] = mJ(A, B),
\]

where the second equality follows since $W_i$ is independent of what happens in iterations different from the $i$-th one.

**Lemma 3.**

\[P[E > 0] \leq \frac{mJ(A \cup B)}{2^s - 1}.\]

**Proof.** We set $s = |A \cup B|$ for the rest of this proof. Consider the generic $i$-th iteration. We have:

\[P[E_i = 0] = P[|\mathcal{M}'(A \cup B)| = 1] = \frac{1}{2^s} \left(1 - \frac{x + 1}{2^s}\right)^{s-1}.\]

In deriving the expression above, for every element $a_j \in A \cup B$ and for every possible value $x$, we computed the probability that $a_j$ receives the value $x$ and all other elements receive higher labels. We further have:

\[
\sum_{x=0}^{2^s-1} \left(1 - \frac{x + 1}{2^s}\right)^{s-1} \geq \int_0^{2^s-1} \left(1 - \frac{x + 1}{2^s}\right)^{s-1} dx = \left(1 - \frac{1}{2^s}\right)^{s-1} > e^{-\frac{1}{2^{s-1}}}
\]

Here, the last inequality follows from Fact 1 after simple manipulations, observing that $2^s \geq s$. Hence we have:

\[P[E_i = 1] = 1 - P[|\mathcal{M}'(A \cup B)| = 1] < 1 - e^{-\frac{1}{2^{s-1}}}
\]

where the last inequality follows since $e^{-x} > 1 - x$. This implies the lemma.

**Corollary 1.** If $k = \Omega(|A \cup B| + \log m)$ then $P[E \geq 1] = o(1)$.

Set $X = |A \cap B|$. Our estimator of $X$ is $\overline{X} = W/(W + m(|A| + |B|))$. The next result states that, up to collisions, our estimation of $|A \cap B|$ differs from the true value by more than a constant with a probability that exponentially decays with $m$ and $J(A, B)$.

**Theorem 1.** For every $\epsilon > 0$ and for every number $m$ of iterations:

\[P[|\overline{X} - X| > \epsilon X] \leq 2e^{-\frac{2^s mJ(A, B)}{2^s - 1}}.
\]

**Proof.** First observe that, by definition of the Jaccard coefficient we have $J(A, B) = X/(|A| + |B|) = X/(|A| + |B| + X)$, which implies:

\[X = \frac{J(A, B)}{J(A, B) + 1} (|A| + |B|).
\]

Furthermore we have:

\[
P[|\overline{X} - X| > \epsilon X] = P[|\overline{X} - X| > \epsilon X] \mid (E = 0) \mid P[E = 0] + P[|\overline{X} - X| > \epsilon X] \mid (E > 0) \mid P[E > 0] \leq P[|\overline{X} - X| > \epsilon X] \mid (E = 0) + P[|\overline{X} - X| > \epsilon X] \mid (E = 0) + \frac{mJ(A \cup B)}{2^s - 1},
\]

where the last inequality follows from Lemma 3. Now, for $i = 1, \ldots, m$, set $\widehat{W}(i) = (W_i | E = 0)$ and $\overline{W} = \sum_{i=1} W(i)$. Also observe that, by Lemma 2, $E[\overline{W}] = mJ(A, B)$. Then, for a given realization $\overline{W}$, our “ideal” estimator $\hat{X}$ of $X$ is:

\[\hat{X} = \frac{\overline{W}}{\overline{W} + m(|A| + |B|)}.
\]

Set $J = \sum_{i=1}^m W(i)/m = \overline{W}/m$. By the expression of $X$ derived before and by the definition of $X$ we have that $\hat{X} > (1 + \epsilon)X$ implies:

\[\frac{\hat{X}}{J + 1} > (1 + \epsilon) \frac{J(A, B)}{J(A, B) + 1},
\]

which in turn implies

\[\hat{J} > (1 + \epsilon)J(A, B) > (1 + \epsilon)J(A, B),\]

that is:

\[\overline{W} > (1 + \epsilon)mJ(A, B) = (1 + \epsilon)E[\overline{W}]\]

Analogously, $\hat{X} < (1 - \epsilon)X$ implies:

\[\hat{J} < (1 - \epsilon)J(A, B) \leq (1 - \epsilon)J(A, B),\]

and this in turn implies

\[\overline{W} < (1 - \epsilon)mJ(A, B) = (1 - \epsilon)E[\overline{W}]\]

So we have:

\[P[|\overline{X} - X| > \epsilon X] \leq P[\overline{W} < E[\overline{W}] \mid > \epsilon E[\overline{W}]].
\]

Finally, the $\overline{W}(i)$’s are statistically independent. This clearly follows since $\overline{W}(i)$ only depends $E_i$ and not on $E_j$, for every $j \neq i$. Hence, applying Chernoff bound [32] to $\overline{W}$ we have:

\[P[\overline{W} < E[\overline{W}] \mid > \epsilon E[\overline{W}]] \leq 2e^{-\frac{2^s mJ(A, B)}{2^s - 1}}.
\]