I/O-Efficient Algorithms on Triangle Listing and Counting

Xiaocheng Hu, Chinese University of Hong Kong
Yufei Tao, Chinese University of Hong Kong
Chin-Wan Chung, Korea Advanced Institute of Science and Technology

This paper studies I/O-efficient algorithms for the *triangle listing problem* and the *triangle counting problem*, whose solutions are basic operators in dealing with many other graph problems. In the former problem, given an undirected graph G, the objective is to find all the cliques involving 3 vertices in G. In the latter problem, the objective is to report just the number of such cliques, without having to enumerate them. Both problems have been well studied in internal memory, but still remain as difficult challenges when G does not fit in memory, thus making it crucial to minimize the number of disk I/Os performed. Although previous research has attempted to tackle these challenges, the state-of-the-art solutions rely on a set of crippling assumptions to guarantee good performance. Motivated by this, we develop a new algorithm that is provably I/O and CPU efficient at the same time, without making any assumption on the input G at all. The algorithm uses ideas drastically different from all the previous approaches, and outperforms the existing competitors by a factor of over an order of magnitude in our extensive experimentation.

Categories and Subject Descriptors: H3.3 [**Information search and retrieval**]: Search process

General Terms: Algorithms, Theory, Performance

Additional Key Words and Phrases: Triangle listing, triangle counting, graphs, I/O-efficient algorithms

1. INTRODUCTION

Given a graph[1] $G = (V, E)$, where V (E) is the set of vertices (edges), the *triangle listing problem* reports all the *triangles* in G, each of which is a clique of 3 vertices u, v, w in G, denoted as Δ_{uvw}. For instance, if G is the graph in Figure 1, the set of triangles is $\{\Delta_{123}, \Delta_{234}, \Delta_{346}, \Delta_{368}, \Delta_{456}, \Delta_{568}\}$. The *triangle counting problem*, on the other hand, aims at returning just the number of triangles, without having to enumerate them: the output for the example of Figure 1 is a single integer 6.

The importance of these problems has long been recognized in the literatures of databases, network analysis, knowledge discovery, and graph theory. Below we list

[1]Unless otherwise stated, a graph in this paper is undirected and simple.

X. Hu and Y. Tao were supported in part by GRF Grants 4164/12 and 4168/13. C.-W. Chung was supported in part by the National Research Foundation of Korea (NRF) grant funded by the Korean Government (MSIP) no. NRF-2014R1A1A2002499 and in part by Defense Acquisition Program Administration and Agency for Defense Development under the contract UD140022PD, Korea.
Authors' addresses: X. Hu and Y. Tao, Department of Computer Science and Engineering, Chinese University of Hong Kong, Sha Tin, New Territories, Hong Kong; email: {xchu, taoyf}@cse.cuhk.edu.hk; C.-W. Chung, Department of Computer Science and Division of Web Science and Technology, KAIST, Daejeon, Korea; email: chungcw@kaist.edu.
This is a preliminary release of an article accepted by ACM Transactions on Database Systems. The definitive version is currently in production at ACM and, when released, will supersede this version.

DOI:http://dx.doi.org/10.1145/0000000.0000000

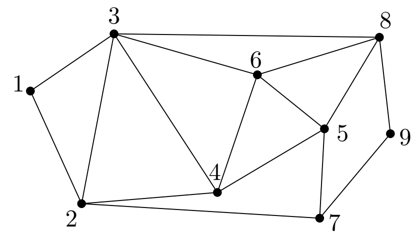


Fig. 1. The input graph in our running example

several representative applications; interested readers may refer to [Chu and Cheng 2012; Kolountzakis et al. 2012; Suri and Vassilvitskii 2011] for additional applications.

Dense Subgraph Mining. Given a graph G, a *dense neighborhood graph* (DN-graph) [Wang et al. 2010] is a subgraph of G, where each pair of connected vertices share at least a number of common neighbors. The most efficient known algorithm [Wang et al. 2010] for DN-graph discovery utilizes a triangle-listing algorithm as a black box. Hence, a faster solution to triangle listing automatically gives rise to an improved algorithm for mining DN-graphs.

Triangular Connectivity. Let u, v be vertices in a graph G. They are *triangularly connected* [Batagelj and Zaversnik 2007] if there is a sequence of triangles $(\Delta_1, \Delta_2, ..., \Delta_s)$ such that u (v) is a vertex in the first (last) triangle Δ_1 (Δ_s), and for every $1 \leq i \leq s-1$, Δ_i shares at least one vertex with Δ_{i+1}. In Figure 1, for instance, vertex 1 is triangularly connected to vertex 5 due to the sequence $(\Delta_{123}, \Delta_{346}, \Delta_{456})$. In *triangular clustering* [Section 5.1, [Schank 2007]], the vertices in G are divided into equivalence classes such that, two vertices are in an equivalence class if and only if they are triangularly connected. The computation of equivalence classes is reduced to finding connected components after all the triangles have been obtained [Schank 2007] (hence, fast triangle listing is again the key).

***k*-truss.** Given a graph G, its k-*truss* $(k \geq 3)$ [Cohen 2009] is the maximum subgraph of G where every edge appears in at least $k-2$ triangles. This is a form of so-called *cohesive subgraphs* that reveal characteristics of social networks [Cohen 2009; Wang and Cheng 2012]. Not surprisingly, the state-of-the-art algorithm [Wang and Cheng 2012] for k-truss computation deploys triangle listing as an initial step.

Network Measurement. A popular approach in studying networks is to interpret their measurements on certain key aspects. A well-known measure is *clustering coefficient* [Watts and Strogatz 1998]. Given a vertex v in a graph, its clustering coefficient equals $t(v)/\binom{d(v)}{2}$, where $t(v)$ is the number of triangles containing v, and $d(v)$ is the degree of v. Another closely related measure is *transitive ratio* [Watts and Strogatz 1998], which is the ratio of the number of triangles and the number of "connected triples" (three vertices form a connected triple if there are at least 2 edges among them). Triangle counting is a crucial step to calculate these measures.

1.1. Motivation

We consider that the input graph G does not fit in memory, and thus needs to be processed by an *external memory graph algorithm*. Recently, such algorithms have received considerable interest (see [Cheng et al. 2011; Chu and Cheng 2012; Hellings et al. 2012] and the references therein), in response to the practical need to analyze massive graphs whose scales exceed the memory capacity of a commodity machine. For example, as of 2011, the social network at Facebook contains more than 721 million active users (a.k.a. nodes), and over 69 billion friendship edges [Bakshy et al. 2012]. If each edge is represented by 2 integers, the entire graph occupies over 550 giga bytes of storage (4 bytes per integer).

I/O-efficient algorithms for triangle listing have been investigated previously. Next, we give an overview of the existing solutions (deferring a detailed coverage to Section 3). Henceforth, let $M = O(|E|)$ be the size of our memory, and B the size of a disk block, both measured in number of words. The values of M and B satisfy $M \geq 2B$, i.e., the memory contains at least two blocks. Finally, denote by K the number of triangles in G.

External Memory Compact Forward (EM-CF) [Menegola 2010]. Incurring $O(|E| + |E|^{1.5}/B)$ I/Os, this algorithm has two main defects. First, it performs at least $|E|$ I/Os, which is prohibitively expensive in most practical environments. Second, its I/O cost is insensitive to M, rendering the algorithm unable to benefit from the availability of extra memory. Note that the I/O complexity of EM-CF does not depend on K because it outputs all triangles in $O(K/B)$ I/Os whereas in general it holds that $K = O(|E|^{1.5})$ (as will be explained in the next section).

External Memory Node Iterator (EM-NI) [Dementiev 2006]. This algorithm runs in $O(|E|^{1.5}/B \cdot \log_{M/B}(|E|/B))$ I/Os. As with EM-CF, its I/O complexity is (almost) insensitive to M, such that the dominating term $|E|^{1.5}/B$ receives no improvement even when memory is abundant.

Graph Partition [Chu and Cheng 2011; 2012]. Neither of the above algorithms is *output sensitive*, namely, their I/O complexity is $\Omega(|E|^{1.5}/B)$, regardless of the output size K. Even though the term $O(|E|^{1.5}/B)$ is compulsory in the worst case where K reaches $\Omega(|E|^{1.5})$, the actual K in a realistic graph is far less than that extreme limit. This makes it interesting to design output sensitive algorithms that are more efficient when K is small.

Recently, Chu and Cheng [Chu and Cheng 2012] made some nice progress in this direction. The crucial idea is to target an I/O complexity of $O(|E|^2/(MB) + K/B)$. The rationale is that $|E|^2/(MB) < |E|^{1.5}/B$ whenever $|E|/M < \sqrt{|E|}$. To see why this is not a stringent inequality, note that even if the memory can hold only 1% of the edges, the inequality is still satisfied as long as $|E| > 10000$. In general, if the memory can accommodate a constant fraction of the input graph (a situation very likely in practice), $O(|E|^2/(MB) + K/B) = O(|E|/B + K/B)$, which is asymptotically optimal because any algorithm must read all edges at least once, and report all the triangles to the disk.

Utilizing several interesting ideas of graph partitioning, Chu and Cheng [Chu and Cheng 2012] presented two algorithms based on graph partitioning that achieve the desired $O(|E|^2/(MB) + K/B)$ bound under a set of assumptions. Unfortunately, as we analyze in Section 3, if any of those assumptions is violated, their algorithms fail to guarantee the target efficiency, and may even suffer from severe performance penalty.

Aggregating Triangles and Witnessing Algorithms. The above listing algorithms can also be used to perform counting with the same I/O performance, by simply avoiding the I/Os for writing triangles to the disk. Specifically, the algorithms of [Chu and Cheng 2012] now finish in $O(|E|^2/(MB))$ I/Os when their assumptions are satisfied, whereas the I/O complexities of EM-CF and EM-NI remain $O(|E| + |E|^{1.5}/B)$ and $O(|E|^{1.5}/B \cdot \log_{M/B}(|E|/B))$, respectively.

All these algorithms, when applied to counting, share a common property: they do the counting by actually *seeing* every triangle in memory. We call such triangle counting algorithms *witnessing algorithms*. The class of witnessing algorithms possesses an appealing feature—they directly support *triangle aggregation* with respect to other aggregate functions as well. In triangle counting, every triangle is implicitly associated with a *weight* 1, such that the answer is merely the sum of all triangles' weights. More generally, the weight of a triangle can be any real number (e.g., when a triangle represents friendships among 3 individuals, its weight can be the average age of those individuals), such that we may alternatively return the sum, average (based on sum and count), maximum (and hence, minimum) of the weights of all triangles.

Generalizing this discussion, a witnessing algorithm can be used to return the total aggregate of all triangles' weights as long as the weight domain D and the aggregation function $\oplus$ satisfy three conditions: (i) $\oplus$ is closed over D, (ii) $\oplus$ is associative and commutative, and (iii) D has an identity element e such that $e \oplus e' = e'$ for any $e' \in D$—such pairs of $(D, \oplus)$ are called *commutative semi-groups*[2]. Besides all the *distributive* aggregate functions popular in database systems, numerous other functions satisfy these conditions as well. As an example, consider D as the set of all possible bit vectors of a fixed length, and $\oplus$ as the AND operator. In this case, the weight of each triangle is a vector in D; and the total aggregate of all triangles is the AND result of their bit vectors. A witnessing algorithm, by simply plugging in this aggregate function, can then immediately be used to compute this total aggregate. Therefore, such an algorithm finds important use in scenarios where the bit-vector (a.k.a. weight) of a triangle is a mergeable sketch, e.g., a bloom filter [Bloom 1970], count-min sketch [Cormode and Muthukrishnan 2005], FM-sketch [Flajolet and Martin 1983], etc.

Henceforth, we will use triangle *counting* as a representative of triangle aggregation. All our discussion extends to commutative semi-group aggregate functions in a straightforward manner.

1.2. Our Contributions

Our first contribution is a new algorithm that settles the triangle listing problem in $O(|E|^2/(MB) + K/B)$ I/Os in *all settings*, namely, with no assumption at all. The algorithm is based on ideas drastically different from those of [Chu and Cheng 2012]. As we will see later, the term $|E|^2/(MB)$ is inevitable, namely, it is impossible to achieve $o(|E|^2/(MB))$ I/Os for all inputs even if $M \ll |E|$.[3] This stands in sharp contrast to triangle listing in memory, where $o(|E|^2)$-time algorithms are well known (e.g., the algorithm of [Chiba and Nishizeki 1985] runs in $O(|E|^{1.5})$ time). In external memory, the I/O complexity $O(|E|^2/(MB) + K/B)$ is thus already worst-case optimal[4] within only a constant factor (even for $M \ll |E|$).

As the next step, we prove that the proposed algorithm is also CPU-efficient: it entails $O(|E| \log |E| + |E|^2/M + \alpha|E|)$ CPU time, where α is the *arboricity* of the input graph—a classic metric for measuring the density of a graph. Section 2 will present an extended introduction to α, while for now, it suffices to note that α never exceeds $O(\sqrt{|E|})$ even in the worst case [Chiba and Nishizeki 1985], and is much smaller for graphs in reality [Lin et al. 2012]. It can be shown that both the terms $|E| \log |E| + |E|^2/M$ and $\alpha|E|$ are inevitable, namely, no algorithm can perform only $o(|E| \log |E| + |E|^2/M)$ or $o(\alpha|E|)$ CPU time for all input graphs (even for $M \ll |E|$). This indicates that our algorithm is *both* I/O and CPU optimal by a constant factor in the worst case.

---

[2]In a semi-group $(D, \oplus)$, it is commonly assumed that each element in D fits in a computer word, and that each calculation by $\oplus$ takes constant CPU time.
[3]The inevitability of $E^2/(MB)$ is trivial for $M = \Omega(|E|)$.
[4]The optimality is claimed with respect to parameters $|E|$, M, and B. Specifically, our algorithm implies the bound $O(|E|^2/(MB) + |E|^{1.5}/B)$ which as will be shown in Section 4.5 is tight even if $M << |E|$.

Besides designing new algorithms, this paper also enhances the understanding of existing algorithms. In this respect we present two new results. First, we prove that the I/O cost of EM-NI can actually be bounded by $O(\alpha \cdot SORT(|E|))$, where α as mentioned earlier is the arboricity, and $SORT(|E|)$ is the I/O cost of sorting $|E|$ elements. The finding reveals why EM-NI is very efficient when the input graph is sparse. In particular, $\alpha = O(1)$ for planar graphs, in which case the I/O complexity of EM-NI becomes $O(SORT(|E|))$. This phenomenon cannot be explained by the previous bound $O(|E|^{1.5}/B \cdot \log_{M/B}(|E|/B))$ on EM-NI.

Second, we revisit an elegant algorithm in [Chu and Cheng 2012] called *randomized graph partitioning* (RGP). We strengthen its analysis with a non-trivial argument to remove a restrictive assumption that was imposed on this algorithm. The removal reduces the remaining assumptions on RGP to some weak conditions that are almost always fulfilled, and thereby considerably improves its applicability.

The last contributions of the paper concern the triangle counting problem. First, our listing algorithm trivially supports triangle counting in $O(|E|^2/(MB))$ I/Os and $O(|E|\log|E| + |E|^2/M + \alpha|E|)$ CPU time. Our algorithm belongs to the witnessing class, and hence, is applicable to any commutative semi-group aggregate functions. Second, we prove, perhaps surprisingly, that no witnessing algorithm is able to solve the triangle counting problem in $O(|E|/B + |E|^{2-\epsilon}/(MB))$ I/Os, no matter how small constant $\epsilon > 0$ is[5], implying that our counting algorithm is essentially I/O-optimal. It can also be shown that the CPU time of our counting algorithm is optimal among all the witnessing algorithms.

1.3. Summary of Experiments

The paper also features an experimental study that is more extensive than those of all the previous work dealing with I/O-efficient triangle listing/counting. We are the first to put all the known algorithms into a direct cross comparison (the papers [Chu and Cheng 2011; 2012] that represent the state of the art unfortunately missed out EM-CF and EM-NI, which we will show are not always the slowest methods). Our experimentation involved both real and synthetic graphs. On the real side, we used *exactly* the same datasets deployed in [Chu and Cheng 2012] to establish the efficiency of DGP and RGP. On the synthetic side, we employed graphs of various distributions, including ones generated from the classic *small world* model [Watts and Strogatz 1998] and the modern popular *recursive matrix* (R-MAT) model [Chakrabarti et al. 2004]. The results demonstrate that MGT outperformed all its competitors by a factor over an order of magnitude in both I/O and CPU efficiency. Furthermore, its performance is consistently good regardless of the graph distribution and size.

1.4. Paper Organization

Section 2 defines a set of frequent notations, and reviews some results from graph theory relevant to our discussion. Section 3 analyzes the previous I/O-efficient algorithms for triangle listing. Section 4 describes the proposed algorithm and proves its theoretical guarantees. Section 5 gives our new results on EM-NI and RGP, and elaborates on their significance. Section 6 presents our lower bound results for triangle counting. Section 7 presents an extensive experimental evaluation to demonstrate the superiority of the proposed technique over the existing solutions. Finally, Section 8 concludes the paper with a summary of our findings.

[5]For example, no witnessing algorithm can guarantee an I/O complexity of $O(|E|/B + |E|^{1.999999999\ldots}/(MB))$.

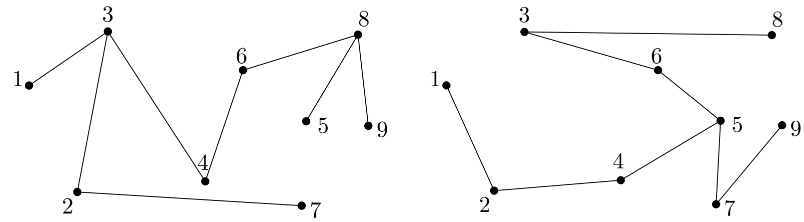


Fig. 2. Two edge-disjoint trees covering all the edges of the graph in Figure 1

1.5. Extensions beyond the Conference Version

A short version of this paper has appeared in [Hu et al. 2013]. The current version extends that preliminary work by proving the aforementioned I/O lower bound on triangle counting. The significance of this lower bound lies in its revelation that triangle counting suffers from an inherent quadratic I/O complexity. As shown in Section 6, this lower bound requires an argument that is drastically different from the one used in the short version to prove a lower bound for triangle *listing*. More specifically, the key to establish the listing lower bound is to argue that the output cost K/B must be large in the worst case (see Section 4.5). This argument does not work for counting where the output cost is trivially small—just a single value.

2. PRELIMINARIES

Basic Notations. As mentioned before, the input to the triangle listing problem is a simple undirected graph $G = (V, E)$, where V and E are the sets of vertices and edges, respectively. We represent an undirected edge between vertices u and v as (u, v) or equivalently as (v, u). For each $v \in V$, $\mathcal{N}(v)$ is the set of *neighbors* of v, where a neighbor is a vertex adjacent to v. Define the *degree* of v as $d(v) = |\mathcal{N}(v)|$.

We consider that G does not have any vertex v with $d(v) = 0$, i.e., an isolated vertex with no incident edge. In fact, if there are such vertices, they can be removed immediately because they obviously cannot appear in any triangle. Finally, we consider that G is given in *adjacency lists*, where the adjacency list of a vertex $v \in V$ stores $\mathcal{N}(v)$ in $O(1 + d(v)/B)$ consecutive blocks.

Arboricity. The *arboricity* is an important notion in graph theory for describing the density of a graph. Formally, the arboricity of a graph G, which is commonly denoted as α, is the minimum number of edge-disjoint forests needed to cover the edges of G. For example, the arboricity of the graph in Figure 1 is $\alpha = 2$, because its edges can be partitioned into 2 forests, as shown in Figure 2 (where each forest is actually a tree), whereas no single forest can cover all the edges apparently.

It is not immediately clear from the above definition why α is a metric for graph density. This is made explicit by a classic result due to Nash-Williams:

PROPOSITION 2.1 ([NASH-WILLIAMS 1964]). *If G has at least 2 vertices, its arboricity α equals:*

$$\alpha = \max_{\forall G'} density(G')$$

where $G' = (V', E')$ is a subgraph of G with $|V'| \geq 2$, and

$$density(G') = \left\lceil \frac{|E'|}{|V'| - 1} \right\rceil .$$

Phrased differently, the arboricity is determined by the densest subgraph of G. In Figure 1, there is more than one densest subgraph: triangle Δ_{123}, for instance, is one, and has a density of $\lceil 3/(3-1) \rceil = 2$.

The proposition leads to the next well-known facts:

COROLLARY 2.2 ([CHIBA AND NISHIZEKI 1985]). *The arboricity α of a graph $G = (V, E)$ satisfies:*

(1) $\alpha \leq \lceil \sqrt{|E|} \rceil$ *in any case;*
(2) $\alpha = O(1)$ *if G is planar.*

Note that α can reach $\Omega(\sqrt{E})$ when G is dense, e.g., when G is a complete graph so that $|E| = \frac{|V|}{2}(|V|-1)$. In practice, a graph (e.g., a social network) is much sparser than a clique, and hence, its arboricity is much lower than $\sqrt{|E|}$ (see [Chiba and Nishizeki 1985; Lin et al. 2012]).

Number of Triangles. As mentioned before, we denote by K the number of triangles in the input graph G. Next, we will get some sense about how large K can possibly be. Consider an edge (u, v) in E. Clearly, any triangle Δ_{uvw} containing the edge must have the property that vertex w is a neighbor of both u and v. As u and v have $d(u)$ and $d(v)$ neighbors respectively, it thus follows that edge (u, v) can appear in at most $\min\{d(u), d(v)\}$ triangles. We therefore have:

$$3K \leq \sum_{(u,v) \in E} \min\{d(u), d(v)\}. \tag{1}$$

Chiba and Nishizeki [Chiba and Nishizeki 1985] observed a delicate connection between the above and arboricity:

PROPOSITION 2.3 ([CHIBA AND NISHIZEKI 1985]). *The right hand side of* (1) *is bounded by $O(\alpha|E|)$.*

It thus follows that $K = O(\alpha|E|)$. This, in turn, suggests $K = O(|E|^{1.5})$ by Corollary 2.2. These bounds are tight in the worst case: when the input G is a complete graph, $K = \binom{|V|}{3} = \Omega(|V|^3) = \Omega(|E|^{1.5})$, while $\alpha = \Omega(\sqrt{E})$ as analyzed before.

Oriented Input. As will be clear later, it is sometimes convenient to work with an *oriented* version $G^\star$ of G. To explain, let us define a total order $\prec$ on V: for any two vertices u, v in G, define $u \prec v$ if

— $d(u) < d(v)$, or
— $d(u) = d(v)$ but u has a smaller id than v.

$G^\star$ is obtained by giving a direction to each edge of G that respects $\prec$. That is, for each edge (u, v) of G, we direct it from u to v in $G^\star$ if $u \prec v$. Figure 3 shows the oriented version of the graph in Figure 1. As will become clear in Section 4, such orientation helps to avoid outputting the same triangle twice, and facilitates the analysis of the running time of our algorithms.

Henceforth, we will write $G^\star = (V, E^\star)$, where $E^\star$ is the set of directed edges decided as above. An edge $(u, v) \in E^\star$ points from u to v (namely, the vertex ordering in the pair is now important). For each vertex v, $\mathcal{N}^+(v)$ represents the set of its out-neighbors, that is, $\mathcal{N}^+(v) = \{u \mid (v, u) \in E^\star\}$. Define $d^+(v) = |\mathcal{N}^+(v)|$ as the *out-degree* of v.

$G^\star$ is stored in adjacency lists, where the adjacency list of a vertex v contains only $\mathcal{N}^+(v)$ in $O(1 + d^+(v)/B)$ consecutive blocks. For instance, in Figure 3, the adjacency

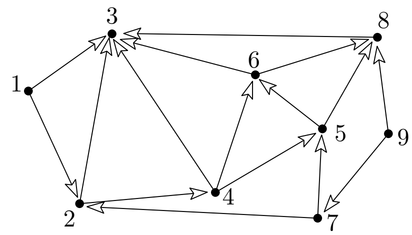


Fig. 3. An oriented version of Figure 1

list of vertex 4 is the set {(vertex) 3, 5, 6}. $G^\star$ can be easily computed from G in $O(SORT(|E|))$ I/Os by sorting.

3. PREVIOUS WORK ON TRIANGLE LISTING AND COUNTING

The triangle listing problem has been extensively studied in internal memory, yielding a large number of algorithms [Chiba and Nishizeki 1985; Eppstein and Spiro 2009; Itai and Rodeh 1978; Latapy 2008; Schank and Wagner 2005]. All of them finish in time $O(|E|^{1.5})$, which is optimal in the worst case where $K = \Omega(|E|^{1.5})$, because $\Omega(K)$ time is needed just to report the triangles. However, these algorithms are not amenable to external memory, as they entail $\Omega(|E|^{1.5})$ I/Os in the worst case due to memory thrashing.

In internal memory, triangle counting can be solved in $O(|V|^{2.376})$ time by resorting to matrix multiplication [Coppersmith and Winograd 1990]. Alon et al. [Alon et al. 1997] gave another algorithm that runs in $O(|E|^{1.41})$ time. These algorithms, whose relative superiority depends on the concrete values of $|V|$ and $|E|$, remain the fastest to date. However, it should be noted that both algorithms are specialized to triangle *counting*, and do not support triangle aggregation in arbitrary commutative semi-groups. For the latter purpose, the fastest algorithm is still to enumerate all the triangles in $O(|E|^{1.5})$ time. In any case, even for counting, the algorithms of [Coppersmith and Winograd 1990] and [Alon et al. 1997] incur $O(|V|^{2.376})$ and $O(|E|^{1.41})$ I/Os in external memory again due to memory thrashing, and hence, are prohibitively expensive in practice. In this work, we are interested in *precise* triangle aggregation, whereas readers interested in only counting and willing to accept errors may refer to [Braverman et al. 2013; Jha et al. 2013; **?**] and the references therein.

In the rest of this section, we extend the description in Section 1 about the existing I/O-efficient algorithms. Focus will be devoted to the solutions of [Chu and Cheng 2012] since they are the state of the art.

3.1. EM-CF

External memory compact forward (EM-CF) [Menegola 2010] accepts an oriented input $G^\star = (V, E^\star)$. For every edge $(u, v) \in E^\star$, it reports a triangle Δ_{uvw} for each $w \in \mathcal{N}^+(u) \cap \mathcal{N}^+(v)$. For example, given edge $(4, 6)$ in Figure 3, it outputs Δ_{463} because vertex 3 is the only common vertex in $\mathcal{N}^+(4) = \{3, 5, 6\}$ and $\mathcal{N}^+(6) = \{3, 8\}$.

Menegola [Menegola 2010] proved that $\mathcal{N}^+(u) \cap \mathcal{N}^+(v)$ can be obtained with $O(1 + \sqrt{|E|}/B)$ I/Os. Thus, the total I/O overhead is $O(|E| + |E|^{1.5}/B)$.

3.2. EM-NI

External memory node iterator (EM-NI) [Dementiev 2006] also takes an oriented input $G^\star = (V, E^\star)$. It executes in two steps:

(1) Obtain the set L of all pairs $(u, \{v, w\})$ such that $(u, v) \in E^\star$ and $(u, w) \in E^\star$.

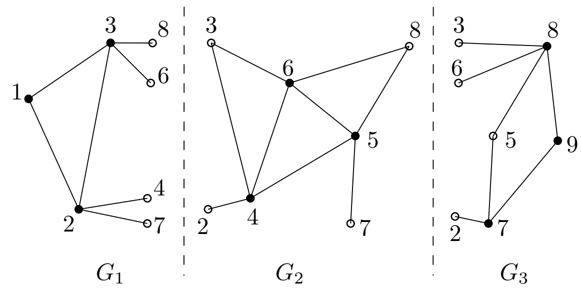


Fig. 4. Extended subgraphs

(2) For each $(u, \{v, w\}) \in L$, check whether $E^\star$ has an edge between v and w, and if so, report Δ_{uvw}.

For example, given the input of Figure 3, the first step returns $L = \big\{(1, \{2, 3\}), (2, \{3, 4\}), (4, \{3, 6\}), (4, \{5, 6\}), (4, \{3, 5\}), (5, \{6, 8\}), (6, \{3, 8\}), (7, \{2, 5\}), (9, \{7, 8\})\big\}$, where each number is a vertex id. The second step then verifies that every pair produces a triangle except $(4, \{3, 5\})$, $(7, \{2, 5\})$ and $(9, \{7, 8\})$.

Dementiev [Dementiev 2006] showed how to perform the two steps in $O(|E|/B + |L|/B \cdot \log_{M/B}(|E|/B))$ I/Os. Menegola [Menegola 2010] further proved $|L| = O(|E|^{1.5})$. It thus follows that EM-NI terminates in $O(|E|^{1.5}/B \cdot \log_{M/B}(|E|/B))$ I/Os.

3.3. Graph Partition

Chu and Cheng [Chu and Cheng 2011; 2012] proposed an algorithmic framework, which we call *graph partition* (GP), for triangle listing. As explained shortly, instantiation of the framework gives rise to different concrete algorithms.

The Framework. Given an input graph $G = (V, E)$ (*not* its oriented version), the framework divides V into disjoint *partitions* $V_1, ..., V_p$. The value of p and the partitioning strategy are precisely what are to be instantiated later. Every triangle Δ_{uvw} in G can now be classified into one of the following:

— **Type-I:** the three vertices u, v, w belong to the same partition.
— **Type-II:** two vertices are in the same partition, while the remaining vertex is in a different partition.
— **Type-III:** the three vertices are in distinct partitions.

For example, assume G to be the graph in Figure 1. Let $p = 3$ and $V_1 = \{1, 2, 3\}$, $V_2 = \{4, 5, 6\}$, and $\{7, 8, 9\}$. Then, Δ_{123}, Δ_{234} and Δ_{368} are of type-I, -II, and -III respectively.

Next, the GP framework reports all the type-I and -II triangles, by resorting to the concept of *extended subgraph*, each of which is a subgraph G_i ($1 \le i \le p$) constructed from a partition V_i. Specifically, G_i is the subgraph induced by the edges adjacent to the vertices of V_i. Figure 4 demonstrates G_1, G_2, G_3 in the aforementioned example on Figure 1. Note that an extended subgraph G_i may contain some vertices absent in V_i. For example, the white vertices $4, 6, 7, 8$ are not in V_1, but appear in G_1 because each of them is a neighbor of a vertex in V_1.

Every triangle of type-I and -II exists in a unique extended subgraph. Making an assumption:

$$\boldsymbol{A_1}\textbf{:}\ \text{Each extended subgraph fits in memory}$$

the GP framework finds those triangles by loading each G_i into memory, and invoking an in-memory triangle listing algorithm.

It remains to report type-III triangles. The framework achieves this goal by converting type-III triangles to the previous types. Observe that a type-III triangle does not use any *intra edge*, i.e., an edge with *both* endpoints in the same partition (e.g., the edges between black vertices in Figure 4). Motivated by this, the GP framework removes all intra edges, and repeat the above on the remaining edges of G, i.e., launching another *iteration*. In a new iteration, the vertices are partitioned differently, so that a type-III triangle of a previous iteration may now become type-I or -II, and hence can be reported.

The partitioning strategy should guarantee $\Omega(M)$ intra edges in every iteration. Therefore, after $O(|E|/M)$ iterations, the left-over edges of G will fit in memory, at which point an in-memory algorithm is deployed to find all the missing triangles.

Deterministic Graph Partitioning (DGP). This algorithm, an instantiation of the above framework, adopts a deterministic strategy to partition V into $V_1, ..., V_p$. Assuming:

$$\boldsymbol{A_2}\textbf{:}\ |V| \le M.$$

DGP first finds an *independent dominating set* D of V. Specifically, D is a maximal set of vertices such that (i) no two vertices in D are adjacent, and (ii) every vertex in V is either in D, or adjacent to a vertex in D. For example, in Figure 1, such a set can be $D = \{1, 4, 7, 8\}$. Then, DGP generates $p = \min\{|D|, \Theta(|E|/M)\}$ partitions based on D (see [Chu and Cheng 2012] for details).

Randomized Graph Partitioning (RGP). As another instantiation, RGP sets $p = \Theta(|E|/M)$ and generates $V_1, ..., V_p$ with a randomized approach: each vertex in V is independently assigned to V_i, where i is chosen uniformly at random from 1 to p.

Discussion on the Assumptions. The efficiency of the GP framework relies on Assumption $\boldsymbol{A_1}$. If $\boldsymbol{A_1}$ does not hold, the in-memory algorithm that the framework uses to find triangles in an extended subgraph G_i will incur memory thrashing, and thus suffer from heavy performance penalty.

Unfortunately, the assumption will *definitely* be violated on DGP in the worst case. To see this, consider that G is a complete graph such that $|V| < M \ll |E| = \binom{|V|}{2}$. It is easy to verify that D can contain only 1 vertex in this case. As a result, $p = 1$, and hence, the extended subgraph G_1 obtained from V_1 is exactly G itself, which does not fit in memory.

On the other hand, the question whether $\boldsymbol{A_1}$ holds on RGP (with sufficiently high probability) was left open in [Chu and Cheng 2012], and still remains unanswered. In this paper, we will close the issue by proving a positive answer based on a non-trivial analysis in Section 5.2.

Chu and Cheng [Chu and Cheng 2012] showed that under the assumption:

$$\boldsymbol{A_3}\textbf{:}\ p = O(M/B), \text{ that is, } M = \Omega(\sqrt{|E| \cdot B})$$

DGP and RGP ensure I/O complexity[6] $O(|E|^2/(MB) + K/B)$, given the simultaneous satisfaction of Assumption $\boldsymbol{A_1}$ and (for DGP) $\boldsymbol{A_2}$.

It is worth mentioning that $\boldsymbol{A_3}$ is necessary to generate p extended subgraphs in $O(|E|/B)$ I/Os (because a block of memory needs to be reserved as the output buffer for each extended subgraph). This assumption can be removed by turning to sorting, but at the expense of increasing the I/O complexities of DGP and RGP to $O(|E|^2/(MB) \cdot \log_{M/B}(|E|/B) + K/B)$.

[6]The complexity is expected for RGP.

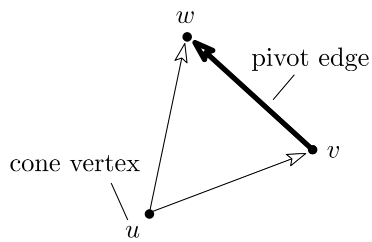


Fig. 5. A triangle in $G^{\star}$ and its pivot edge ($u \prec v \prec w$)

4. A NEW TRIANGLE LISTING ALGORITHM

This section presents a new algorithm called *massive graph triangulation* (MGT), which settles triangle listing with $O(|E|^2/(MB) + K/B)$ I/Os in all circumstances, namely, needing no assumption at all. In the meantime, MGT entails only $O(|E| \log |E| + |E|^2/M + \alpha|E|)$ CPU time, where α is the arboricity of the input graph (see Section 2). Both the I/O and CPU complexities are worst-case optimal as we will prove later.

4.1. Guaranteeing I/O-Efficiency

We will first describe MGT by focusing *only* on I/O efficiency, i.e., pretending that all CPU operations were for free. The algorithm accepts an oriented input $G^{\star} = (V, E^{\star})$ as defined in Section 2, which can be computed from the original input $G = (V, E)$ in $O(SORT(|E|))$ I/Os.

Pivot Edge. Let us make an observation about how a triangle Δ_{uvw} of G appears in the oriented graph $G^{\star}$. Recall that there is a total order $\prec$ on the vertices of V. Assume, without loss of generality, that $u \prec v \prec w$. Thus, the edges of Δ_{uvw} have directions as illustrated in Figure 5. In particular, u, v and w have 2, 1 and 0 outgoing edges in the triangle, respectively. We refer to the outgoing edge of v as the *pivot edge* of Δ_{uvw}, and u as the *cone vertex* of Δ_{uvw}.

Algorithm. MGT runs in *iterations*, each performing two steps:

(1) Load into memory the next cM edges in $E^{\star}$, where $c < 1$ is a constant to be decided later. Let E_{mem} be the set of those edges.
(2) Report all the triangles whose pivot edges are in E_{mem}.

MGT correctly finds all triangles because (i) Step 1 ensures every edge of $E^{\star}$ to appear in E_{mem} in a unique iteration, and (ii) for any triangle Δ, Step 2 guarantees its discovery in the iteration where E_{mem} contains the pivot edge of Δ.

Details of Step 2. Algorithm 1 shows how to implement Step 2 in $O(|E|/B)$ I/Os, plus the minimum cost of outputting the triangles found.

Let V_{mem} be the set of vertices induced by the edges in the E_{mem} returned by Step 1. For example, suppose that E_{mem} consists of the solid edges in Figure 6; then $V_{mem} = \{3, 5, 6, 8\}$. In this case, Step 2 ought to output Δ_{436}, Δ_{456}, and Δ_{568} because their pivot edges $(3, 6)$, $(5, 6)$, and $(6, 8)$ are in E_{mem}.

Step 2 processes each vertex $u \in V$ in turn as follows. First, define:

$$\mathcal{N}_{mem}(u) \; = \; \mathcal{N}^{+}(u) \cap V_{mem} \tag{2}$$

namely, $\mathcal{N}_{mem}(u)$ is the set of out-neighbors of u that appear in V_{mem}. We obtain $\mathcal{N}_{mem}(u)$ by reading the adjacency list $\mathcal{N}^{+}(u)$ from the disk, while in the meantime

ALGORITHM 1: STEP 2 (VERSION 1)

Input: $G^\star = (V, E^\star)$ and a set E_{mem} of edges in memory
Output: All triangles whose pivot edges are in E_{mem}

obtain V_{mem} from E_{mem};
for *each vertex* $u \in V$ **do**
    read $\mathcal{N}^+(u)$ from disk to acquire $\mathcal{N}_{mem}(u)$ in memory ;
    $S \leftarrow$ the set of edges from u to the vertices in $\mathcal{N}_{mem}(u)$;
    find in $S \cup E_{mem}$ all the triangles where u is the cone vertex ;
    release $\mathcal{N}_{mem}(u)$ and S from memory ;
return

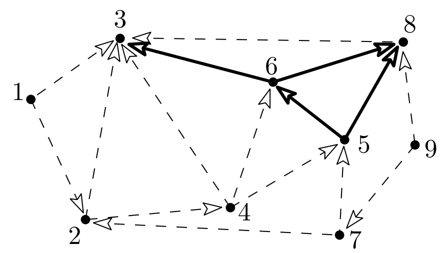


Fig. 6. Illustration of Step 2 (solid edges are in E_{mem})

adding a vertex $v \in \mathcal{N}^+(u)$ to $\mathcal{N}_{mem}(u)$ if $v \in V_{mem}$. Note that

$$|\mathcal{N}_{mem}(u)| \le |V_{mem}| \le 2|E_{mem}| \le 2cM.$$

Hence, by setting c appropriately[7], E_{mem} and $\mathcal{N}_{mem}(u)$ together occupy at most M words of storage, and can co-exist in memory.

The knowledge of $\mathcal{N}_{mem}(u)$ essentially "augments" E_{mem} with up to $|\mathcal{N}_{mem}(u)|$ edges leaving u. For instance, the processing of vertex 4 in Figure 6 gives $\mathcal{N}_{mem}(4) = \{3, 5, 6\}$. Effectively, $\mathcal{N}_{mem}(4)$ permits us to "see" 3 more edges in memory: $(4, 3)$, $(4, 5)$ and $(4, 6)$.

As a crucial fact, *if a triangle Δ_{uvw} (where u is the cone vertex) should be reported by Step 2, it can now be discovered in memory*. To understand, recall that Step 2 needs to report Δ_{uvw} only if the pivot edge $(v, w) \in E_{mem}$. This implies that v and w both belong to V_{mem}, and hence, also to $\mathcal{N}_{mem}(u)$. It follows that edges (u, v) and (u, w) have been "augmented" into memory.

For illustration, consider again vertex 4 in Figure 6. As mentioned before, $\mathcal{N}_{mem}(4)$ reveals edges $(4, 3)$, $(4, 5)$ and $(4, 6)$ in memory. At this moment, Δ_{463} and Δ_{456} (which are the triangles of vertex 4 that Step 2 needs to report) are memory resident.

After processing u, we clear $\mathcal{N}_{mem}(u)$ from memory, and move on to handle the next vertex of V. Note that E_{mem} is kept in memory throughout Step 2. Algorithm 1 summarizes the above in pseudocode.

I/O Complexity. Each iteration performs one scan over the adjacency lists of all vertices in $O(|E|/B)$ I/Os. The number of iterations is $\Theta(|E|/M)$ because each of them loads $\Theta(M)$ distinct edges of $E^\star$ into E_{mem}, except possibly the last iteration. This, as well as the fact that $\Theta(B)$ triangles can be reported in one I/O, proves that the I/O cost of MGT is bounded by $O(|E|^2/(MB) + K/B)$.

[7]For example, c can be $1/4$ in a naive implementation where an edge requires two words to store.

4.2. A CPU-Efficient Algorithm

The MGT algorithm described in the previous section does not achieve the desired bound $O(|E| \log |E| + |E|^2/M + \alpha|E|)$ on CPU time. Towards that purpose, we will modify the algorithm with extra ideas. This subsection will do so under the *small-degree assumption*:

$$d^+(v) \leq cM/2 \quad \text{for all } v \in V \tag{3}$$

where c is the same constant as in our earlier description. Recall that $d^+(v)$ is the out-degree of v in $G^\star$. Section 4.4 will remove this assumption, and obtain the final version of MGT that guarantees the claimed I/O and CPU bounds in all cases.

All-or-Nothing Requirement. Previously, the E_{mem} of an iteration is permitted to include cM arbitrary edges of $E^\star$. Now we require that E_{mem} should contain either *all* the outgoing edges of v or *none* of it, for every $v \in V$. For instance, the set E_{mem} in Figure 6 satisfies this *all-or-nothing requirement*.

We fulfill the requirement by processing one vertex at a time in Step 1 (of MGT). Specifically, let v be the next vertex whose outgoing edges have never appeared in E_{mem}. We add all its outgoing edges to E_{mem} if the size of the resulting E_{mem} does not exceed cM. Otherwise, v is left to the next iteration; and Step 1 terminates here by returning the current E_{mem}. This can also be described in pseudocode as:

ALGORITHM 2: STEP 1

Input: $G^\star = (V, E^\star)$
Output: A set E_{mem} of at least $cM/2$ edges in $E^\star$

$E_{mem} \leftarrow \emptyset$;
for *each vertex* $v \in V$ *whose edges have never entered* E_{mem} **do**
  if $|\mathcal{N}^+(v)| + |E_{mem}| \leq cM$ **then**
    add all the edges of v to E_{mem};
  else
    break ;
return E_{mem}

The above strategy may end up with an E_{mem} with less than cM edges. However, under the small-degree assumption, except the last iteration $|E_{mem}|$ must be at least $cM/2$, because if $|E_{mem}| < cM/2$, then E_{mem} should have been able to take in the (at most $cM/2$) outgoing edges of one more vertex.

Step 2 of MGT proceeds as described earlier. Since $|E_{mem}|$ is still $\Theta(M)$ except possibly in the final iteration, the I/O complexity of the algorithm remains bounded by $O(|E|^2/(MB) + K/B)$.

CPU-Implementation of Step 2. Our description so far has ignored all the in-memory operations, the details of which are to be filled in next. Define

$$V^+_{mem} = \{v \in V_{mem} \mid v \text{ has an outgoing edge in } E_{mem}\}$$

For example, in the example of Figure 6, $V^+_{mem} = \{5, 6\}$. Vertices 3 and 8, which belong to V_{mem}, are not in V^+_{mem} because they have no outgoing edge in E_{mem}.

We create hash structures on $V_{mem}, V^+_{mem}, E_{mem}$ so that:

— Given any vertex v, whether $v \in V_{mem}$ and/or $v \in V^+_{mem}$ can be decided in $O(1)$ time.
— Given any vertices u, v, whether $(u, v) \in E_{mem}$ can be decided in $O(1)$ time.

Clearly, these hash structures occupy only $O(|E_{mem}|)$ space.

For each vertex $u \in V$, Step 2 needs to report all the triangles Δ_{uvw} where $u \prec v \prec w$ and the pivot edge (v, w) exists in E_{mem}. Expanding the procedure in Section 4.1, we first obtain $\mathcal{N}_{mem}(u)$ in $O(|\mathcal{N}^+(u)|)$ time, by using constant time to check whether $v \in V_{mem}$ for each $v \in \mathcal{N}^+(u)$. We then further acquire:

$$\mathcal{N}^+_{mem}(u) \;=\; \mathcal{N}_{mem}(u) \cap V^+_{mem}$$

in $O(|\mathcal{N}_{mem}(u)|) = O(|\mathcal{N}^+(u)|)$ time, by checking whether $v \in V^+_{mem}$ for each $v \in N_{mem}(u)$. In Figure 6, for instance, when u is vertex 4, we have $\mathcal{N}_{mem}(4) = \{3, 5, 6\}$ and $\mathcal{N}^+_{mem}(4) = \{5, 6\}$.

Next, we use $O(|\mathcal{N}^+_{mem}(u)| \cdot |\mathcal{N}_{mem}(u)|)$ time to find the triangles of u that should be reported in the current iteration. For each vertex pair

$$(v, w) \in \mathcal{N}^+_{mem}(u) \times \mathcal{N}_{mem}(u) \text{ with } v \neq w$$

check in constant time whether $(v, w) \in E_{mem}$. If so, report Δ_{uvw}. For instance, to process vertex 4 in Figure 6, the algorithm probes edges $(5, 3)$, $(5, 6)$, $(6, 3)$ and $(6, 5)$ in E_{mem}. The second and third exist in E_{mem}, thus spawning Δ_{456} and Δ_{463}. Algorithm 3 restates the above details in pseudocode:

ALGORITHM 3: STEP 2 (DETAILS EXPANDED)

Input: $G^\star = (V, E^\star)$ and a set E_{mem} of edges in memory
Output: All triangles whose pivot edges are in E_{mem}

obtain V_{mem} and V^+_{mem} from E_{mem} ;
build hash structures on V_{mem}, V^+_{mem}, and E_{mem} ;
for *each vertex* $u \in V$ **do**
    read $\mathcal{N}^+(u)$ from disk to acquire $\mathcal{N}_{mem}(u)$ in memory ;
    obtain $\mathcal{N}^+_{mem}(u)$ from $\mathcal{N}_{mem}(u)$ in memory ;
    for *each* $v \in \mathcal{N}^+_{mem}(u)$ **do**
        for *each* $w \in \mathcal{N}_{mem}(u)$ **do**
            if $v \neq w$ **and** *edge* $(v, w) \in E_{mem}$ **then**
                output Δ_{uvw} ;
    release $\mathcal{N}_{mem}(u)$ and $\mathcal{N}^+_{mem}(u)$ from memory ;
return

The complexity $O(|\mathcal{N}^+_{mem}(u)| \cdot |\mathcal{N}_{mem}(u)|)$ may appear expensive at first glance due to its quadratic nature. Somewhat surprisingly, when one adds up this complexity for all vertices throughout all iterations, the sum turns out to be $O(\alpha|E|)$, as we analyze next.

4.3. Bounding the CPU-Time

This subsection will analyze the CPU time of the modified MGT algorithm under the small-degree assumption. Let us start with a useful fact:

LEMMA 4.1. $\sum_{v \in V} \left(d^+(v)\right)^2 = O(\alpha|E|)$.

PROOF. For any $v \in V$:

$$\left(d^+(v)\right)^2 = d^+(v) \sum_{u \in \mathcal{N}^+(v)} 1 = \sum_{u \in \mathcal{N}^+(v)} d^+(v).$$

Hence:

$$\begin{aligned}
\sum_{v \in V} \left(d^+(v)\right)^2 &= \sum_{v \in V} \sum_{u \in \mathcal{N}^+(v)} d^+(v) \\
&= \sum_{(v,u) \in E^\star} d^+(v) \\
&\leq \sum_{(v,u) \in E^\star} d(v) \\
\text{(by how } (v,u) \text{ is directed)} &= \sum_{(v,u) \in E} \min\{d(v), d(u)\} \\
\text{(by Proposition 2.3)} &= O(\alpha |E|).
\end{aligned}$$

□

By the discussion of the previous subsection, in an iteration of MGT, Step 2 spends

$$\begin{aligned}
& O(|\mathcal{N}^+(u)| + |\mathcal{N}^+_{mem}(u)| \cdot |\mathcal{N}_{mem}(u)|) \\
= \; & O(|\mathcal{N}^+(u)| + |\mathcal{N}^+_{mem}(u)| \cdot |\mathcal{N}^+(u)|)
\end{aligned} \tag{4}$$

time on each vertex $u \in V$. The terms $|N^+(u)|$ of all $u \in V$ add up to exactly $|E^\star| = |E|$. Hence, the total contribution by this term throughout all the $\Theta(|E|/M)$ iterations is $O(|E|^2/M)$. Henceforth, we will focus on the second term $|\mathcal{N}^+_{mem}(u)| \cdot |\mathcal{N}^+(u)|$.

Let $h = \Theta(|E|/M)$ be the number of iterations actually performed by MGT. Denote by $\mathcal{N}^+_{mem}(u,i)$ the content of $\mathcal{N}^+_{mem}(u)$ in the i-th iteration, for $1 \leq i \leq h$. In other words, the total contribution (to the CPU time) of the second term in (4) across all nodes u and all iterations i is at most

$$\sum_{i=1}^{h} \sum_{u \in V} O(|\mathcal{N}^+_{mem}(u,i)| \cdot |\mathcal{N}^+(u)|). \tag{5}$$

Our all-or-nothing requirement (see Section 4.2) ensures:

LEMMA 4.2. $\mathcal{N}^+_{mem}(u,1)$, ..., $\mathcal{N}^+_{mem}(u,h)$ *are mutually disjoint.*

PROOF. The all-or-nothing requirement guarantees that each vertex $v \in V$ belongs to V^+_{mem} in a unique iteration. In other words, the sets V^+_{mem} of all iterations are mutually disjoint. The lemma then follows from the fact $\mathcal{N}^+_{mem}(u,i)$ is a subset of the V^+_{mem} of iteration i, for $1 \leq i \leq h$. □

As $\mathcal{N}^+_{mem}(u,i) \subseteq \mathcal{N}^+(u)$ for each i, the lemma implies:

$$\sum_{i=1}^{h} |\mathcal{N}^+_{mem}(u,i)| \;\leq\; |\mathcal{N}^+(u)| = d^+(u).$$

Hence:

$$\sum_{i=1}^{h} \left(|\mathcal{N}^+_{mem}(u,i)| \cdot |\mathcal{N}^+(u)|\right) \leq \left(d^+(u)\right)^2. \tag{6}$$

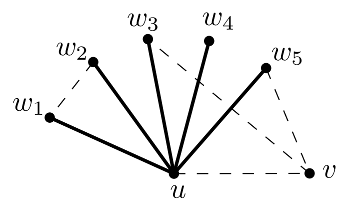


Fig. 7. Illustration for the conversion algorithm (S consists of the solid edges)

Therefore:

$$\begin{aligned}
(5) &= \sum_{u \in V} \sum_{i=1}^{h} O\left(|\mathcal{N}^{+}_{mem}(u,i)| \cdot |\mathcal{N}^{+}(u)|\right) \\
\text{(by (6))} &= \sum_{u \in V} O\big(d^{+}(u)\big)^2 \\
\text{(by Lemma 4.1)} &= O(\alpha |E|).
\end{aligned}$$

Finally, recall that the input to the triangle listing problem is an undirected graph G, from which the oriented version $G^\star$ is computed by sorting, which (when implemented as the standard external sort [Aggarwal and Vitter 1988]) takes $O(|E| \log |E|)$ CPU time. We thus have proved that MGT entails $O(|E| \log |E| + |E|^2/M + \alpha|E|)$ CPU time overall.

4.4. Removing the Small-Degree Assumption

This subsection presents the last component of our MGT algorithm, which deals with the case where the oriented input $G^\star$ does not satisfy the small-degree assumption (see (3)). In other words, there is at least one vertex v such that $d^+(v) > cM/2$.

MGT handles the case by working on the original (undirected) input G, instead of the oriented version $G^\star$. It removes certain edges of large-degree vertices, while ensuring that all the triangles involving those edges have been reported. The edge removal turns G into another graph G', whose oriented version $G'^\star$ satisfies the small-degree assumption. $G'^\star$ is then fed into the algorithm in Section 4.2 to report the remaining triangles.

Converting G to G'. Given $G = (V, E)$, we carry out the conversion as follows:

(1) Identify a vertex $u \in V$ with $d(u) > cM/2$. If u does not exist, the conversion terminates with $G' = G$.
(2) Load a set S of $cM/2$ edges[8] of u into memory.
(3) Report all the triangles that involve at least one edge in S.
(4) Remove the edges in S from E. Repeat from Step 1.

We refer to an execution from Step 1 to 4 as an *iteration*. The number of iterations is bounded by $\Theta(|E|/M)$ because $\Theta(M)$ edges are removed by an iteration. Next, we explain how to implement Step 3 efficiently.

The edges of S form a 2-level tree where u is the root, as shown in Figure 7 where the solid edges constitute S. Let T be the set of leaf vertices of this tree, e.g., $T = \{w_1, w_2, ..., w_5\}$ in Figure 7. Clearly, any triangle Δ involving at least an edge in S must have u as a vertex. Furthermore, Δ must be one of the types below:

[8]The value $cM/2$ is chosen to facilitate understanding. In practice, one can load as many edges of u as the memory can accommodate. The algorithm still works the same way as described subsequently.

— **Type-1:** 2 vertices of Δ are in T.
— **Type-2:** Only 1 vertex of Δ is in T.

In Figure 7, for instance, $\Delta_{uw_1w_2}$ and Δ_{uvw_5} are of type-1 and -2 respectively (note that $u \notin T$).

We create a hash structure on T to permit testing whether $v \in T$ in constant time for any $v \in V$. Both types of triangles can be found easily by a single scan on E. This is in fact obvious for type-1: for every edge $(v, w) \in E$, report Δ_{uvw} if and only if both v and w belong to T.

To find type-2 triangles, we process the adjacency list $\mathcal{N}(v)$ of each vertex $v \neq u$ as follows. First, check whether $u \in \mathcal{N}(v)$, and if not, we are done with v and move on to the next vertex. Otherwise, obtain $T(v) = \mathcal{N}(v) \cap T$ (e.g., $T(v) = \{w_3, w_5\}$ in Figure 7). So far we need to read $\mathcal{N}(v)$ from the disk once, and spend $O(|\mathcal{N}(v)|)$ CPU time. Finally, for every vertex $w \in T(v)$, output a triangle Δ_{uvw}; this requires $O(|T(v)|) = O(|\mathcal{N}(v)|)$ CPU time.

It is thus clear that each iteration of our conversion algorithm performs $O(|E|/B)$ I/Os, and entails $O(|E|)$ CPU time, plus the minimum output cost. Therefore, the overall algorithm has an I/O complexity $O(|E|^2/(MB) + K/B)$ and CPU complexity $O(|E|^2/M + K) = O(|E|^2/M + \alpha|E|)$ (recall from Section 2 that $K = O(\alpha|E|)$).

Putting Everything Together. The graph G' has the property that every vertex $v \in V$ has degree at most $cM/2$. Hence, its oriented version $G'^\star$ satisfies the small-degree assumption due to the obvious fact $d^+(v) \leq d(v)$. We can now apply the algorithm of Section 4.2 to find the remaining triangles. It is easy to verify that every triangle in G is reported exactly once. This completes the whole MGT algorithm, which brings us to this paper's first main result:

THEOREM 4.3. *The MGT algorithm solves the triangle listing problem in* $O(|E|^2/(MB) + K/B)$ ***I/Os and*** $O(|E| \log |E| + |E|^2/M + \alpha|E|)$ ***CPU time.***

As a remark regarding the usage of our algorithm in practice, the component developed in this subsection is mainly of theoretical interest because with the memory capacity of today's machines, it seems rather unlikely that a vertex will have a degree as high as $\Omega(M)$.

4.5. Worst-Case Optimality

We now explain why it is impossible to design an algorithm with I/O complexity $o(|E|^2/(MB))$ even when $M = o(|E|)$. Consider that the input G is a complete graph, and $M \geq |V| = \Theta(\sqrt{|E|})$. The number of triangles equals $\binom{|V|}{3} = \Omega(|V|^3)$. Therefore, any algorithm must incur

$$\begin{aligned}\Omega(|V|^3/B) &= \Omega(|E|^{1.5}/B) \\ &= \Omega(|E|^2/(|V|B)) = \Omega(|E|^2/(MB))\end{aligned}$$

I/Os just to report the triangles. This argument shows that the term $|E|^2/(MB)$ is compulsory in the worst case. Note that Theorem 4.3 implies that the MGT algorithm performs $O(|E|^2/(MB) + |E|^{1.5}/B)$ I/Os even in the worst case, which is therefore already optimal up to a constant factor with respect to parameters $|E|$, M, and B. Note that *this optimality result holds for any* M *satisfying* $M \geq |V|$, that is, as long as all the vertices (but not edges) can be stored in memory.

The above finding also implies a lower bound of $\Omega(|E|^2/M)$ on CPU time because $\Omega(|V|^3) = \Omega(|E|^2/M)$ time is needed just to output triangles. This immediately rules out any algorithm with $o(|E| \log |E|)$ CPU time because $|E| \log |E| = o(|E|^2/M)$ when $M = \Theta(\sqrt{E})$. Given also the necessity of the term $\alpha|E|$ (see Section 2), it follows that any algorithm must incur $\Omega(|E|\log|E| + |E|^2/M + \alpha|E|)$ CPU time in the worst case, matching the upper bound in Theorem 4.3.

5. FINDINGS ON KNOWN ALGORITHMS

This section will strengthen the current understanding about two existing algorithms for triangle listing: EM-NI and RGP, as reviewed in Section 3. For EM-NI, we will reveal for the first time why the algorithm is especially efficient on sparse graphs. For RGP, we will remove a restrictive assumption imposed on its applicability.

5.1. EM-NI

We now prove:

THEOREM 5.1. *The EM-NI algorithm solves the triangle listing problem in $O(\alpha \cdot SORT(|E|))$ I/Os.*

PROOF. As mentioned in Section 3, the previous work has shown that EM-NI performs $O(\frac{|E|}{B} + \frac{|L|}{B}\log_{M/B}\frac{|E|}{B})$ I/Os. Below, we will show that $|L| = O(\alpha|E|)$ which therefore will establish the theorem because $SORT(|E|) = \Theta(|E|/B \cdot \log_{M/B}(|E|/B))$.

Let $G^\star = (V, E^\star)$ be the oriented input to EM-NI. Recall that L is the set of all such pairs $(u, \{v, w\})$ that (u, v) and (u, w) are both in $E^\star$. For each $u \in V$, there are exactly $\binom{d^+(u)}{2}$ such pairs, where $d^+(u)$ is the out-degree of u. Therefore:

$$|L| \;=\; \sum_{u \in V} \binom{d^+(u)}{2} \;\le\; \sum_{u \in V} \left(d^+(u)\right)^2$$

which is $O(\alpha|E|)$ by Lemma 4.1. □

Previously, the I/O-complexity of EM-NI was understood as $O(|E|^{1.5}/B \cdot \log_{M/B}(|E|/B))$ (see Section 3). Hence, Theorem 5.1 is separated from the old result whenever $\alpha = o(\sqrt{|E|})$. Moreover, Theorem 5.1 clearly indicates that the I/O efficiency of EM-NI depends linearly on α, which as discussed in Section 2 measures the graph density. In particular, when G is planar, $\alpha = O(1)$ (see Corollary 2.2), in which case EM-NI finishes in $O(SORT(|E|))$ I/Os. In fact, there are other graph families (e.g., random graphs of the *preferential attachment model*) where the graphs are known to have constant arboricities [Goel and Gustedt 2006].

Theorem 5.1 makes it possible to compare EM-NI and our MGT algorithm in a more sensible manner. Interestingly, *EM-NI never has a better complexity as long as $M \ge |V|$.* To see this, first notice that:

$$\alpha \;\ge\; \frac{|E|}{|V| - 1}. \tag{7}$$

The above inequality results directly from the definition of α as the minimum number of edge-disjoint forests needed to cover all the edges of E: as each forest has at most $|V|-1$ edges, at least $|E|/(|V|-1)$ forests are needed to cover all the $|E|$ edges. Therefore, when $M \ge |V|$,

$$\alpha \;>\; |E|/|V| \;\ge\; |E|/M$$

which makes

$$|E|^2/(MB) < \alpha|E|/B < \alpha \cdot SORT(|E|)$$

namely, the I/O complexity in Theorem 5.1 is never better than that in Theorem 4.3.

5.2. RGP

Recall from Section 3 that the graph partition framework [Chu and Cheng 2012], which is the state of the art, relies on a key assumption $\boldsymbol{A_1}$ to attain its I/O efficiency. Chu and Cheng [Chu and Cheng 2012] instantiated the framework into the DGP and RGP algorithms. As explained in Section 3, unfortunately, Assumption $\boldsymbol{A_1}$ is inherent in DGP and thus impossible to remove. However, it remains open whether the assumption can be eliminated on RGP. The rest of the subsection will answer this question almost in all scenarios.

We will need the following Chernoff bound:

PROPOSITION 5.2 ([ALON AND SPENCER 2000]). *Let $X_1, ..., X_n$ be independent random variables between 0 and 1. Let $X = \sum_{i=1}^{n} X_i$ and $\mu = \sum_{i=1}^{n} \mathbf{E}[X_i]$. Then:*

$$\mathbf{Pr}[X \geq 2\mu] \ \leq \ \exp(-\mu/3).$$

The remainder of this subsection will follow the notations in Section 3.3. In addition, let d_{max} be the largest degree of the vertices in the input graph $G = (V, E)$. We now present the last main result of this paper:

THEOREM 5.3. *Under the condition:*

$$\boldsymbol{A_4}\textbf{:}\ M \geq 24 d_{max} \ln |E|,$$

Assumption $\boldsymbol{A_1}$ holds with probability at least $1 - 1/|E|$.

PROOF. Set $p = c|E|/M$ where c is a constant to be decided later. For each vertex $v \in V$ and each $i \in [1, p]$, define

$$X_i(v) \ = \ \begin{cases} d(v) & \text{if } v \in V_i \\ 0 & \text{otherwise} \end{cases}$$

As v is assigned to V_i with probability $1/p$, $\mathbf{E}[X_i(v)] = d(v)/p$.

Let $X_i = \sum_{v \in V} X_i(v)$, namely, X_i is the sum of the degrees of all vertices in G_i. Let Y_i be the number of edges in the extended subgraph G_i obtained from V_i. We observe:

$$Y_i \ \leq \ X_i. \tag{8}$$

The inequality holds because every edge in G_i is counted at least once by X_i. Clearly:

$$\mathbf{E}[X_i] = \sum_{v \in V} \mathbf{E}[X_i(v)] = \sum_{v \in V} \frac{d(v)}{p} = \frac{2|E|}{p} = \frac{2M}{c}.$$

The random variables $X_i(v)$ of different $v \in V$ are mutually independent. Furthermore, $X_i(v) \leq d_{max}$. Hence, applying Chernoff bound (Proposition 5.2) on the random variables $Z_i(v) = X_i(v)/d_{max}$ of all $v \in V$ gives:

$$\mathbf{Pr}\left[\sum_{v \in V} Z_i(v) \geq \frac{4M}{c \cdot d_{max}}\right] \ \leq \ \exp\left(-\frac{2M}{3c \cdot d_{max}}\right) \Rightarrow$$

$$\mathbf{Pr}\left[X_i \geq \frac{4M}{c}\right] \ \leq \ \exp\left(-\frac{2M}{3c \cdot d_{max}}\right) \tag{9}$$

When $M \geq 3c \cdot d_{max} \ln |E|$, it holds that

$$\exp\left(-\frac{2M}{3c \cdot d_{max}}\right) \ \leq \ \frac{1}{|E|^2} \ < \ \frac{1}{|E|} \cdot \frac{M}{c|E|} \ = \ \frac{1}{p|E|}$$

with which (9) gives:

$$\mathbf{Pr}\left[X_i \geq \frac{4M}{c}\right] \leq \frac{1}{p|E|} \tag{10}$$

G_i can be stored in at most $2Y_i$ words, which by (8) is at most $2X_i$ words. Setting $c = 8$, (10) shows that $2X_i \geq M$ occurs with probability at most $1/(p|E|)$ when $M \geq 24 d_{max} \ln |E|$, namely, the probability for G_i not to fit in memory is at most $1/(p|E|)$.

Therefore, when $\boldsymbol{A_4}$ holds, by union bound the probability that *any* of $G_1, ..., G_p$ does not fit in memory is at most $1/|E|$, thus completing the proof. □

For a massive input graph G with a massive E, Theorem 5.3 shows that $\boldsymbol{A_1}$ holds with extremely high probability as long as the memory is not too small[9]. Note that condition $\boldsymbol{A_4}$ is tight up to only a logarithmic factor, because *when* $M < d_{max}$, $\boldsymbol{A_1}$ *can never be satisfied* such that not only RGP but also the graph partition framework itself will not be able to function. To see this, let v be the vertex in G with degree d_{max}, and suppose that $v \in V_i$, for some $i \in [1, p]$. Then, the extended subgraph G_i created from V_i must contain at least d_{max} edges, and therefore, does not fit in memory.

Theorem 5.3 has reduced the assumptions on RGP's applicability to only $\boldsymbol{A_3}$ and $\boldsymbol{A_4}$, both of which appear reasonable given the memory capacity of today's machines. Perhaps more important is the fact that $\boldsymbol{A_3}$ and $\boldsymbol{A_4}$ can be checked efficiently, by scanning the input graph at most once to glean $|E|$ and d_{max}. In contrast, there does not appear a way to check the original assumption $\boldsymbol{A_1}$, except for letting the algorithm run anyway.

6. AN I/O LOWER BOUND ON TRIANGLE COUNTING

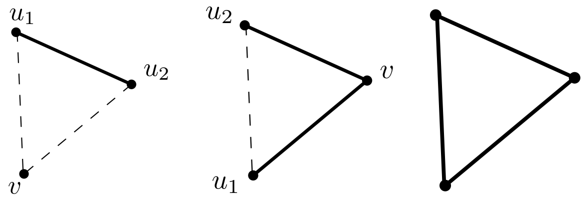
The MGT algorithm we described in Section 4 can be trivially adapted to perform triangle counting in $O(|E|^2/(MB))$ I/Os and $O(|E| \log |E| + |E|^2/M + \alpha|E|)$ CPU time (for this purpose, it suffices to simply ignore the part of the algorithm that outputs the discovered triangles to the disk, but instead, increments a counter whenever such a triangle is seen). As we will see, this is essentially the best efficiency attainable by the class of witnessing algorithms.

We model a witnessing algorithm as follows. At the beginning, the input graph G is stored in $O(|E|/B)$ blocks in the disk; and the memory is empty. At any moment, the algorithm is allowed to keep at most M edges in memory. An I/O operation brings at most B edges into the memory. The algorithm maintains a counter of the number of distinct triangles that have ever existed in memory (i.e., all 3 edges of such a triangle were memory resident simultaneously). It terminates as soon as the counter equals the total number of triangles in G.

It is easy to show that no witnessing algorithm can improve the bound $O(|E| \log |E| + |E|^2/M + \alpha|E|)$ on CPU time—in fact, as far as witnessing algorithms are concerned, triangle counting takes just as much CPU work as triangle listing. Therefore, the same lower bound argument in Section 4.5 still holds on CPU time. What is challenging, however, is prove a tight lower bound on the I/O cost. The I/O argument in Section 4.5 falls short for this purpose. Recall that the crux of that argument is to show that the output cost $\Theta(K/B)$ can be as large as $\Omega(|E|^2/(MB))$. This approach no longer works for triangle counting, where the output cost is trivially small: just a single I/O to output the count.

Based on an entirely different argument, we give in the rest of this section a proof for the following theorem:

[9]It is worth mentioning that the constant 24 in Theorem 5.3 can be reduced by resorting to stronger forms of Chernoff bounds (see the appendix of [Alon and Spencer 2000]) and more careful mathematical derivation.



(a) Type 1 (b) Type 2 (c) Type 3

Fig. 8. Three types of new triangles

THEOREM 6.1. *No witnessing algorithm can guarantee solving the triangle counting problem in $O(|E|/B + |E|^{2-\epsilon}/(MB))$ I/Os, no matter how small constant $\epsilon > 0$ is.*

This theorem thus implies that our MGT algorithm is already optimal within a tiny factor.

6.1. A Weaker Lower Bound

In this subsection, we will prove a lower bound of $\Omega(|E|/B + |E|^{1.5}/(MB))$ on triangle counting. This result is weaker than the final lower bound needed to establish Theorem 6.1, but nonetheless illustrates some ideas behind our methodology.

The crucial question to ask is: how many *new* triangles can an algorithm witness by performing an I/O? Next we show that this number is $O(MB)$. To prove this claim, let us denote by S_{mem} the set of edges already in memory before the I/O, and let S_{io} be the set of edges loaded into the memory by the I/O. Clearly, $|S_{mem}| \leq M$ and $|S_{io}| \leq B$. If a triangle Δ appears in the memory for the first time after the I/O—namely, Δ is a *new triangle*—then Δ must use at least an edge from S_{io}. To understand why, notice that if all 3 edges of a triangle are in S_{mem}, then this triangle was already memory resident before the I/O (and hence, is not new). It thus follows that every new Δ must belong to one of the types below:

— Type 1: Δ uses 2 edges from S_{mem}, and 1 edge from S_{io}.
— Type 2: Δ uses 1 edge from S_{mem}, and 2 edges from S_{io}.
— Type 3: Δ uses 3 edges from S_{io}.

See Figure 8 for an illustration of the three types of triangles, where edges from S_{mem} and S_{io} are dashed and solid, respectively.

Denote by V_{mem} the set of vertices of the edges in S_{mem}, and likewise, by V_{io} the set of vertices of the edges in S_{io}. It is clear that $|V_{mem}| \leq 2M$ and $|V_{io}| \leq 2B$. There can be at most $2MB$ type-1 triangles because every such triangle uniquely corresponds to a pair $(v, (u_1, u_2)) \in V_{mem} \times S_{io}$ (see Figure 8a), whereas $V_{mem} \times S_{io}$ has only $2MB$ such pairs in total. Similarly, $2MB$ is also an upper bound on the number of type-2 triangles because every such triangle uniquely corresponds to a pair $(v, (u_1, u_2)) \in V_{io} \times S_{mem}$ (see Figure 8b). Finally, there can be $O(B^{1.5})$ type-3 triangles because they can be formed only using the B edges in S_{io}, whereas in general, any graph with B edges can have at most $O(B^{1.5})$ triangles (as explained in Section 2).

Now we know that by performing one I/O, an algorithm can witness $O(MB + B^{1.5}) = O(MB)$ (applying $M \geq 2B$) new triangles. This fact holds for *arbitrary* graphs. Therefore, for any graph having $\Omega(|E|^{1.5})$ triangles, a witnessing algorithm must perform $\Omega(|E|^{1.5}/(MB))$ I/Os to see all triangles at least once. Combining with the trivial fact that every algorithm must perform $\Omega(|E|/B)$ I/Os to read all the edges, we obtain an I/O lower bound $\Omega(|E|/B + |E|^{1.5}/(MB))$.

6.2. Number of New Triangles Witnessable in t I/Os

The lower bound obtained in the previous subsection is unfortunately too loose for establishing Theorem 6.1. The looseness arises because our earlier argument assumes that *every* I/O would allow the algorithm to see $\Theta(MB)$ new triangles. It turns out that this is too optimistic. Intuitively, after a "productive I/O" that brings into memory a great number of new triangles, the algorithm would need to perform several "non-productive I/Os" to replace many edges in memory before another productive I/O can occur. Next, we validate this intuition with a formal argument.

Our approach is to analyze how many new triangles can be witnessed in memory by any sequence of $t \geq 1$ consecutive I/Os. Specifically, we will deal with the following problem. First choose any moment of the algorithm. Let S_{mem} be the set of edges currently in memory, and c_1 be the number of distinct triangles that have been seen in memory so far. Then, the algorithm performs t I/Os of its choice, after which we denote by c_2 the number of distinct triangles that have been seen at this moment. The objective is to analyze how large $\delta = c_2 - c_1$ can be, namely, how many new triangles can be witnessed in memory due to the t I/Os.

Next we prove a crucial fact:

LEMMA 6.2 (WITNESSING LEMMA). *If $t \leq M/B$, then $\delta = O(M\sqrt{tB})$.*

Proof of Lemma 6.2. We say that a triangle Δ is *new* if it contributes to δ, namely, Δ is brought into memory for the first time by one of the t I/Os. Let S_{io} be the union of all the edges read by those t I/Os. Hence, $|S_{io}| \leq tB$. It is clear that a new Δ must use at least one edge from S_{io}. Therefore, we can classify Δ into type-1, -2, or -3 by the same way as described in the previous subsection. Once again, denote by V_{mem} the set of vertices of the edges in S_{mem}, and by V_{io} the set of vertices of the edges in S_{io}.

We will bound the numbers C_1, C_2, and C_3 of type-1, -2, and -3 triangles (defined as before), respectively. Let us start with type-1. We denote by $v_1, ..., v_{2M}$ the (at most) $2M$ vertices in V_{mem}. For each $i \in [1, 2M]$, let x_i be the number of edges in S_{mem} that are adjacent to v_i, and are used in forming type-1 triangles. Observe that v_i can participate in at most $\min\{\binom{x_i}{2}, tB\}$ type-1 triangles because (i) every such triangle uses a pair of those x_i edges of v_i, but on the other hand, (ii) every such triangle consumes a distinct edge in S_{io}. Therefore:

$$
\begin{aligned}
C_1 &\leq \sum_{i=1}^{2M} \min\left\{\binom{x_i}{2}, tB\right\} \\
&< \sum_{i=1}^{2M} \min\{x_i^2, tB\} \\
&\leq \sum_{i=1}^{2M} x_i \cdot \sqrt{tB}
\end{aligned}
$$

which is at most $2M \cdot \sqrt{tB}$ by the constraint that $\sum_{i=1}^{2M} x_i \leq 2M$.

Next, we analyze the number C_2 of type-2 triangles with a similar approach. Denote by $v'_1, ..., v'_{2tB}$ the (at most) $2tB$ vertices in V_{io}. For each $i \in [1, 2tB]$, let x'_i be the number of edges in S_{io} that are adjacent to v'_i, and are used in forming type-2 triangles. Observe that v'_i can participate in at most $\min\{\binom{x'_i}{2}, M\}$ type-2 triangles. Therefore:

$$
C_2 \leq \sum_{i=1}^{2tB} x'_i \cdot \sqrt{M}
$$

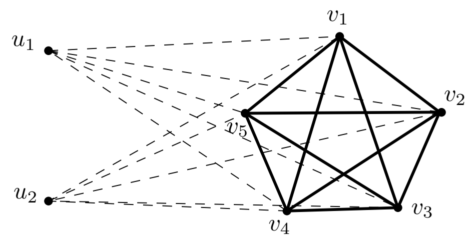


Fig. 9. Tightness of the witnessing lemma

which subject to the constraint $\sum_{i=1}^{2tB} x'_i \le 2tB$ is at most $2tB\sqrt{M}$.

Finally, the number C_3 of type-3 triangles is clearly at most $O((tB)^{1.5})$. Hence, when $t \le M/B$, the number δ of new triangles is at most $2M\sqrt{tB} + 2tB\sqrt{M} + O((tB)^{1.5}) = O(M\sqrt{tB})$. This completes the proof of Lemma 6.2.

Remarks. Setting $t = 1$, one can see from Lemma 6.2 that a single I/O allows an algorithm to witness $O(M\sqrt{B})$ triangles, i.e., fewer than the $O(MB)$ bound obtained by the loose analysis in Section 6.1.

We complete the discussion of the witnessing lemma by showing that it is tight within a constant factor. Consider the situation where the memory already has a complete bipartite graph between a set of vertices $u_1, ..., u_{M/(2\sqrt{tB})}$ and another set of vertices $v_1, ..., v_{\sqrt{tB}}$; and then, the next t I/Os load into memory all the $\binom{\sqrt{tB}}{2} < tB/2$ edges in the clique involving vertices $v_1, ..., v_{\sqrt{tB}}$. Figure 9 illustrates the idea with $M/(2\sqrt{tB}) = 2$ and $\sqrt{tB} = 5$; the dashed edges already exist in memory before the t I/Os, whereas the solid edges are brought into memory by the t I/Os. At this point, the graph in memory has $\frac{M}{2\sqrt{tB}} \cdot \sqrt{tB} + tB/2 = M/2 + tB/2 \le M$ edges in total; namely, the graph can indeed fit in memory. Furthermore, all the triangles in this graph are new (i.e., they are witnessed for the first time during the t I/Os). As each pair of edges adjacent to u_i ($1 \le i \le M/(2\sqrt{tB})$) contributes a triangle, the number of triangles is $\Omega(\frac{M}{2\sqrt{tB}} \cdot (\sqrt{tB})^2) = \Omega(M\sqrt{tB})$.

6.3. Completing the Proof of Theorem 6.1

We now utilize the witnessing lemma to derive a tighter I/O lower bound for triangle counting:

LEMMA 6.3. *Every witnessing algorithm solving the triangle counting problem must perform $\Omega(|E|/B + |E|^{1.5}/(B\sqrt{M}))$ I/Os in the worst case.*

PROOF. We will focus on proving the necessity of the term $|E|^{1.5}/(B\sqrt{M})$. Consider running a witnessing algorithm on a graph with $\Omega(|E|^{1.5})$ triangles. Let H be the number of I/Os it performs. Let us divide the sequence of H I/Os into $\lceil H/t \rceil$ disjoint subsequences, each of which has exactly t I/Os, except possibly the last subsequence which can have less than t I/Os, where t is a value falling in $[1, M/B]$ to be fixed later. By the witnessing lemma, the number of new distinct triangles that can be witnessed in each subsequence is at most $cM\sqrt{tB}$ for some constant $c > 0$. It thus follows that

$$\left\lceil \frac{H}{t} \right\rceil cM\sqrt{tB} \;=\; \Omega(|E|^{1.5}).$$

Therefore, $H = \Omega(\frac{|E|^{1.5}\sqrt{t}}{M\sqrt{B}})$, which is $\Omega(|E|^{1.5}/(B\sqrt{M}))$ by setting $t = M/B$. $\square$

We are now ready to prove Theorem 6.1. Suppose, on the opposite, that there exists a witnessing algorithm solving the triangle counting problem in $O(|E|/B+|E|^{2-\epsilon}/(MB))$ I/Os for some constant $\epsilon > 0$. Note that this bound must hold for any value of $M \in [2B, |E|]$. Now consider $M = |E|^{1-\epsilon}$. In this case, the algorithm guarantees finishing in $O(|E|/B + \frac{|E|^{2-\epsilon}}{|E|^{1-\epsilon}B}) = O(|E|/B)$ I/Os. However, Lemma 6.3 says that any algorithm must perform $\Omega(|E|/B + \frac{|E|^{1.5}}{B\sqrt{|E|^{1-\epsilon}}})) = \Omega(|E|^{1+\epsilon/2}/B)$ I/Os. We thus have arrived at a contradiction.

7. EXPERIMENTS

In this section, we experimentally compare the proposed algorithm against the previous methods for triangle listing in external memory. The next subsection will explain the environments where our experiments were performed. Then, Section 7.2 (7.3) evaluates the efficiency of alternative solutions on real (synthetic) datasets.

7.1. Environmental Setup

All the experiments were performed under Linux (specifically Ubuntu 12.04) on a machine that was running an Intel 3GHz CPU (dual core) and was equipped with 8 giga bytes of memory. The block size B, which was fixed by the operating system, was equal to 4k bytes.

We compared our MGT algorithm against the existing I/O-efficient solutions to triangle listing, namely, EM-CF, EM-NI, DGP and RGP, all of which have been reviewed in Section 3. We implemented all the algorithms in C++, using the gcc compiler with the optimizer option O3. Our implementation is fully memory conscious. Namely, the binary executable of each algorithm accepts (among others) a parameter M that specifies in number of bytes how much memory can be used. It is guaranteed that the algorithm makes full use of the allocated memory, but its memory usage at any instant never exceeds M.

We measured the cost of an algorithm in two aspects: number of I/Os, and overall running time. The former was counted by strictly adhering to the standard external memory model [Aggarwal and Vitter 1988], namely, an I/O reads a block from the disk into memory, or conversely, writes B words in memory to a disk block. The total running time, on the other hand, was measured as the amount of wall-clock time elapsed during the algorithm's execution.

In all cases, the input graph was given in adjacency lists *without* orientation. This is precisely the format assumed by DGP and RGP. If an algorithm (i.e., MGT, EM-CF, and EM-NI) requires an oriented version of the graph (as defined in Section 2), it carried out the orientation on the fly. For these algorithms, each cost we report later has always included the overhead incurred from performing the orientation. Finally, we exclude the output cost (i.e., the time to report the triangles found) because it is identical for all algorithms as they must return exactly the same set of triangles. Viewed differently, the performance we report for each algorithm can also be interpreted as its cost for triangle counting.

7.2. Performance on Real Data

Datasets and Methodology. We deployed five real datasets named *LJ*, *USRD*, *BTC*, *WebUK* and *SubDomain*, respectively. All the datasets except *SubDomain*, which is new dataset introduced in this paper, are exactly the same ones used in [Chu and

Cheng 2012] where state-of-the-art DGP and RGP were developed. The meta information of the datasets is displayed in Table I[10].

Table I. Meta data of real graphs

	LJ	*USRD*	*BTC*	*WebUK*	*SubDomain*
$\lvert V\rvert$	4,846,609	23,947,347	164,660,997	62,338,347	89,247,739
$\lvert E\rvert$	42,851,237	28,854,312	386,411,047	938,715,528	1,940,007,864
$\lvert E\rvert/\lvert V\rvert$	8.84	1.20	2.35	15.06	21.74
$\max_{v\in V} d(v)$	20,333	9	1,637,619	48,822	3,032,590
$\max_{v\in V} d^{+}(v)$	686	4	645	5,692	10,695
disk size	364 M	403 M	4.1 G	7.5 G	15.1 G

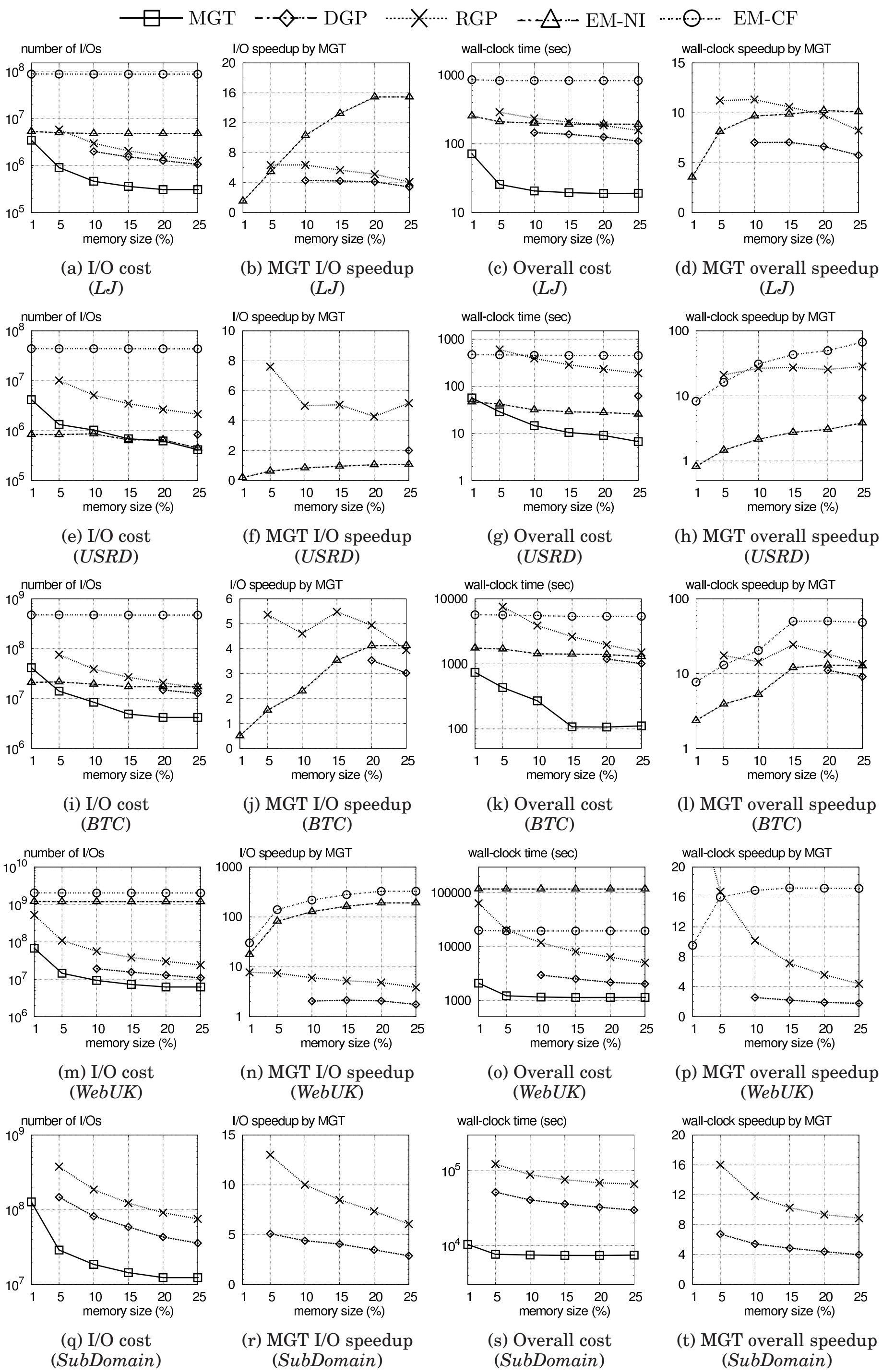
More specifically, *LJ* represents a social network (see *http://www.livejournal.com*) where a vertex corresponds to an individual, and an edge indicates friendship between two persons. *USRD*, on the other hand, is a part of the US road network, where a vertex (edge) is a road junction (segment). *BTC* is an object relational graph where a vertex is a real-world object, and an edge reflects a certain relationship between two objects (e.g., a person *owns* an item). This graph was obtained from the RDF dataset of the Billion Triple Challenge 2009 (*http://vmlion25.deri.ie*). *WebUK* captures the hyperlinks (i.e., edges) among a set of web pages (i.e., vertices) gathered for investigation of web spam (*http://barcelona.research.yahoo.net*). Finally, *SubDomain* was extracted from the Common Crawl 2012 web corpus, where each node represents a subdomain in the web, and each edge represents the existence of hyperlinks between two subdomains (*http://webdatacommons.org/hyperlinkgraph/*).

Clearly, the amount M of memory allocated to an algorithm is the most crucial factor behind its efficiency. If M is so large that the entire input graph fits in memory, all algorithms will behave similarly because they essentially degenerate into in-memory triangle listing. The key to evaluating an external memory algorithm lies in examining how well it performs when only a fraction of the dataset fits in memory. Motivated by this, for each input graph, we varied M from 1% of the graph's disk size (see Table I) to 25%, and in the meantime, compared the performance of different algorithms.

Results. Figure 10 presents all the results of the experiments on the real graphs. In the first row, Figure 10a plots the I/O cost of each algorithm on dataset *LJ* as a function of the memory size. Let the *I/O speedup* of MGT over another algorithm X be defined as the ratio between the numbers of I/Os entailed by X and MGT, respectively (e.g., an I/O speedup 2 means that MGT needs half as many I/Os as X). Figure 10b shows the I/O speedups of MGT over the other algorithms as the memory grows. Figures 10c and 10d demonstrate the corresponding results on the overall running time, noticing that *wall-clock speedup* of MGT is defined by extending I/O-speedup straightforwardly to wall-clock time. The second, third, fourth and last rows of Figure 10 present the outcome of the same experiments on *USRD*, *BTC*, *WebUK* and *SubDomain*, respectively.

In Figures 10a-p, EM-CF and EM-NI are sometimes omitted from a "speedup diagram" if MGT achieves an exceedingly high speedup over them. For example, EM-CF is absent from Figure 10b because it incurred over 100 times more I/Os than MGT (as a result, the inclusion of EM-CF would destroy the diagram's clarity). On the other hand, the disappearance of DGP from a diagram is *always due to its inapplicability*. Recall that this algorithm is subject to several assumptions as discussed in Section 3.3. If any of Assumptions $\boldsymbol{A_1}$, $\boldsymbol{A_2}$ and $\boldsymbol{A_3}$ is violated, DGP fails to execute. In fact, DGP failed in

[10]The graph sizes listed here are different from those provided in [Chu and Cheng 2012], which, however, is due to the errors in [Chu and Cheng 2012], as has been verified by our communication with the authors of [Chu and Cheng 2012].



(a) I/O cost (*LJ*)
(b) MGT I/O speedup (*LJ*)
(c) Overall cost (*LJ*)
(d) MGT overall speedup (*LJ*)
(e) I/O cost (*USRD*)
(f) MGT I/O speedup (*USRD*)
(g) Overall cost (*USRD*)
(h) MGT overall speedup (*USRD*)
(i) I/O cost (*BTC*)
(j) MGT I/O speedup (*BTC*)
(k) Overall cost (*BTC*)
(l) MGT overall speedup (*BTC*)
(m) I/O cost (*WebUK*)
(n) MGT I/O speedup (*WebUK*)
(o) Overall cost (*WebUK*)
(p) MGT overall speedup (*WebUK*)
(q) I/O cost (*SubDomain*)
(r) MGT I/O speedup (*SubDomain*)
(s) Overall cost (*SubDomain*)
(t) MGT overall speedup (*SubDomain*)

Fig. 10. Efficiency comparison on real graphs

at least one setting on every dataset: specifically, $M \leq 5\%$, $\leq 20\%$, $\leq 15\%$, $\leq 5\%$ and $= 1\%$ for ***LJ***, ***USRD***, ***BTC***, ***WebUK*** and ***SubDomain***, respectively. All failures were due to violation of Assumption $\boldsymbol{A_2}$ with only one exception: the failure on ***WebUK*** ($M = 5\%$) was due to $\boldsymbol{A_1}$. RGP failed in fewer occasions than DGP: $M = 1\%$ for ***LJ*** and ***USRD*** due to $\boldsymbol{A_3}$, and $M = 1\%$ for ***BTC*** due to $\boldsymbol{A_1}$. Finally, on the largest dataset ***SubDomain***, some experiments did not terminate within 48 hours: EM-CF (for all M), EM-NI (for all M) and RGP ($M = 1\%$). Their results are hence excluded from Figures 10q-t.

other hand, never failed in any of our experiments. This is perfectly explainable: our analysis in Section 5.2 has reduced its assumptions to $\boldsymbol{A_3}$ and $\boldsymbol{A_4}$, both of which were easily satisfied in our settings.

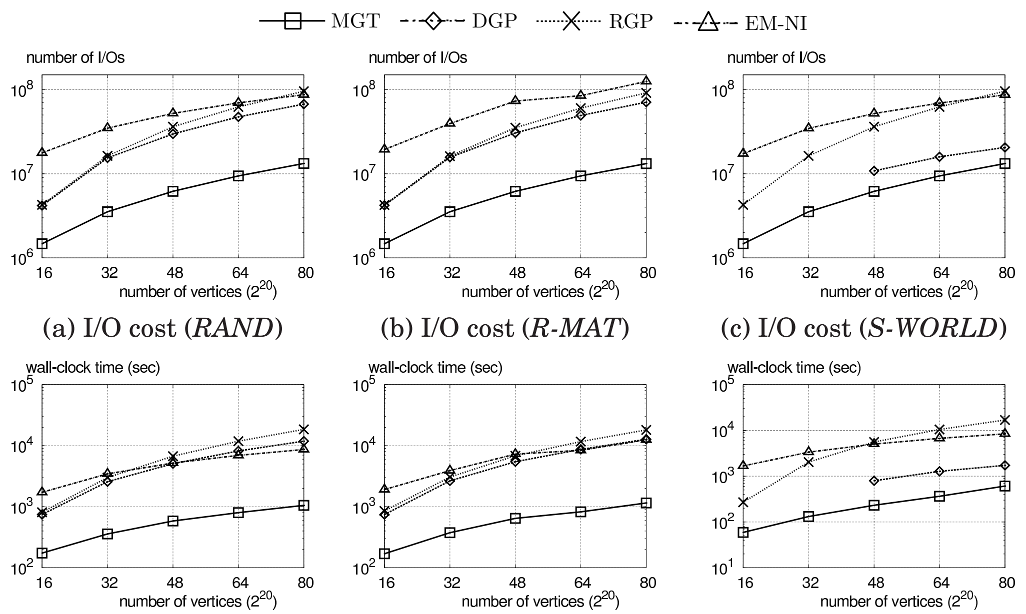
It is evident from Figure 10 that MGT exhibited by far the best performance overall. In a majority of cases, it significantly outperformed all its competitors in both I/O and CPU efficiency. Further observe that, against every other method, MGT was faster in overall execution time by a factor of over an order of magnitude in at least one experiment. These findings confirm the high effectiveness of the proposed techniques in practice, in addition to their rigorous theoretical guarantees which have already been established in earlier sections.

Regarding the other algorithms, EM-CF is clearly the worst-performing solution. This is not surprising because, as mentioned in Section 3.1, its I/O complexity is even greater than $\Omega(|E|)$, which is already prohibitively expensive in reality. EM-NI, on the other hand, is very sensitive to the graph density, as predicted by our analysis in Section 5.1. When the density is low, this algorithm can be fairly efficient, as can be seen from its performance on *USRD* (which is nearly planar). Unfortunately, with the increase of density, the cost of this algorithm grows dramatically, in fact to such an extent that it can be even more expensive than EM-CF (see Figure 10o). DGP is a capable method in the sense that, when it did not fail, it demonstrated acceptable performance (although still several times slower than MGT). Finally, RGP, which enjoys the same I/O complexity as MGT, apparently has a much larger hidden constant in its complexity.

7.3. Performance on Synthetic Data

Datasets and Methodology. Having established the superiority of our MGT algorithm on real data, we now proceed with a set of controlled experiments that aims at comparing the competing algorithms on different types of graphs, and evaluating their scalability with the graph size. Towards this purpose, we generated graphs of three distributions:

— **Random** (*RAND*): Given a pair of values n and m, we generated a random graph with n vertices by creating m edges, each of which connects a vertex pair chosen uniformly at random. This was followed by a clean-up process to eliminate duplicate edges between the same pair of vertices.
— **Recursive Matrix** (*R-MAT*): Proposed by Chakrabarti et al. [Chakrabarti et al. 2004], this model has gained considerable popularity due to its simplicity and ability to emulate a large variety of graphs in reality. It captures the fact that the vertex degree distribution of a real graph often *resembles but is not exactly* a power law. Given a pair of n and m, we created an *R-MAT* graph of n vertices and m edges using the generator published at *http://www.cse.psu.edu/~madduri/software/GTgraph* with its default parameters (the same *R-MAT* parameters were also used in the experiments of [Tai et al. 2011; Zhao et al. 2011]). Finally, duplicate edges were removed by a clean-up process.



(a) I/O cost (*RAND*) (b) I/O cost (*R-MAT*) (c) I/O cost (*S-WORLD*)

(d) Overall cost (*RAND*) (e) Overall cost (*R-MAT*) (f) Overall cost (*S-WORLD*)

Fig. 11. Efficiency comparison on synthetic graphs

— **Small World** (*S-WORLD*): This classic model was first described by Watts and Strogatz [Watts and Strogatz 1998]. Given n and m where m is a multiple of $2n$, an *S-WORLD* graph was obtained as follows. First, imagine putting n vertices on a circle where each vertex v is connected to its $m/(2n)$ nearest neighbors on the left and right, respectively[11]. This creates m edges in total. Then, independently with probability p, each edge of v is replaced by an edge that connects v to another vertex chosen randomly. At the end, a clean-up process was invoked to remove duplicate edges. We set the model parameter p to 1%, as this was the median value in the experiments of the seminal work [Watts and Strogatz 1998].

We set $m = 16n$ in all experiments. For each distribution, we generated 5 graphs by varying n from around 16 to 80 million. The following table gives the meta data of all the synthetic graphs after duplicate removal (these figures apply to all distributions):

Table II. Meta data of synthetic graphs (all distributions)

$\lvert V\rvert$	16×2^{20}	32×2^{20}	48×2^{20}	64×2^{20}	80×2^{20}
$\lvert E\rvert$	2.7×10^8	5.4×10^8	8.1×10^8	10.7×10^8	13.4×10^8
disk size	2.1 G	4.2 G	6.4 G	8.5 G	10.6 G

For each graph, we inspected the efficiency of all algorithms by fixing the amount of allocated memory to 1 giga bytes. The only exception was EM-CF, which was omitted from further inspection due to its huge uncompetitive running time.

Results. Figure 11 demonstrates the comparison results of MGT, DGP, RGP and EM-NI on synthetic graphs, by focusing on the I/O and wall-clock time in the first and

[11]The 1st left neighbor of v is the vertex immediately to the left of v on the ring, the 2nd neighbor is the vertex further to the left, and so on.

second rows, respectively. DGP has no results on *S-WORLD* graphs when $|V| \leq 32 \times 2^{20}$ because it failed due to the violation of $\boldsymbol{A_1}$.

The relative superiority of different algorithms generally follows the patterns observed earlier from real datasets. In every experiment, MGT outperformed all its competitors by a wide margin in both I/O and CPU efficiency. This phenomenon nicely complements the results of the preceding subsection in showing the robustness of MGT's performance, regardless of the graph distribution and the graph size.

8. CONCLUSIONS

Triangle listing and *counting* are important classic problems on graphs that have numerous applications in different domains. Although they have been well studied in internal memory, solving them I/O-efficiently on massive graphs exceeding the memory capacity still remains as a challenging task. Previously, there have been several attempts to tackle the challenge. However, even the state-of-the-art algorithms still entail lengthy execution time, and even so, are haunted by various assumptions that limit the applicability of those algorithms.

In this paper, we have presented a new algorithm named MGT based on fresh ideas drastically different from the previous approaches. The proposed algorithm does not rely on any assumption, and outperformed every other alternative solution by a factor up to at least an order of magnitude in our extensive experimental evaluation. Furthermore, the MGT algorithm is based on a solid theoretical foundation, which proves its excellent efficiency in all settings, regardless of the graph distribution and size. In particular, we have shown that the I/O and CPU complexities of MGT are optimal in the worst case with respect to parameters $|E|$, M, and B.

9. POST ACCEPTANCE REMARK

The I/O optimality claimed in this paper holds for triangle *listing* when $M = \Omega(\sqrt{|E|})$. Recently, Pagh and Silvestri [Pagh and Silvestri 2014] considered a more general problem called triangle *enumeration*, where the goal is to witness each triangle once in internal memory without writing it to the disk (in other words, the cost $\Theta(K/B)$ of *reporting* the triangles need not be counted). By leveraging our MGT algorithm as a building brick, they managed to achieve an I/O complexity (for triangle enumeration) that is better than ours for general values of $|E|$, M, and B. They also independently developed the same witnessing lower bound as in Lemma 6.3. Our paper was submitted to ACM TODS on 27 Sep 2013, whereas the paper [Pagh and Silvestri 2014] first appeared on Arxiv on 3 Dec 2013.

REFERENCES

Alok Aggarwal and Jeffrey Scott Vitter. 1988. The Input/Output Complexity of Sorting and Related Problems. *CACM* 31, 9 (1988), 1116–1127.

Noga Alon and Joel H. Spencer. 2000. *The Probabilistic Methods* (2nd ed.). Wiley, New York.

Noga Alon, Raphael Yuster, and Uri Zwick. 1997. Finding and Counting Given Length Cycles. *Algorithmica* 17, 3 (1997), 209–223.

Eytan Bakshy, Itamar Rosenn, Cameron Marlow, and Lada A. Adamic. 2012. The Role of Social Networks in Information Diffusion. *CoRR* (2012).

V. Batagelj and M. Zaversnik. 2007. Short cycle connectivity. *Discrete Mathematics* 307 (2007), 310–318.

Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *CACM* 13, 7 (1970), 422–426.

Vladimir Braverman, Rafail Ostrovsky, and Dan Vilenchik. 2013. How Hard Is Counting Triangles in the Streaming Model?. In *ICALP*. 244–254.

Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A Recursive Model for Graph Mining. In *SDM*.

James Cheng, Yiping Ke, Ada Wai-Chee Fu, Jeffrey Xu Yu, and Linhong Zhu. 2011. Finding maximal cliques in massive networks. *TODS* 36, 4 (2011), 21.

N. Chiba and T. Nishizeki. 1985. Arboricity and Subgraph Listing Algorithms. *SIAM J. of Comp.* 14, 1 (1985), 210–223.

Shumo Chu and James Cheng. 2011. Triangle listing in massive networks and its applications. In *SIGKDD*. 672–680.

Shumo Chu and James Cheng. 2012. Triangle listing in massive networks. *TKDD* 6, 4 (2012), 17.

Jonathan Cohen. 2009. Graph Twiddling in a MapReduce World. *Computing in Science and Engineering* 11, 4 (2009), 29–41.

Don Coppersmith and Shmuel Winograd. 1990. Matrix Multiplication via Arithmetic Progressions. *Journal of Symbolic Computation* 9, 3 (1990), 251–280.

Graham Cormode and S. Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms* 55, 1 (2005), 58–75.

Roman Dementiev. 2006. *Algorithm engineering for large data sets hardware, software, algorithms*. Ph.D. Dissertation. Saarland University.

David Eppstein and Emma S. Spiro. 2009. The h-Index of a Graph and Its Application to Dynamic Subgraph Statistics. In *WADS*. 278–289.

Philippe Flajolet and G. Nigel Martin. 1983. Probabilistic Counting. In *FOCS*. 76–82.

Gaurav Goel and Jens Gustedt. 2006. Bounded Arboricity to Determine the Local Structure of Sparse Graphs. In *Workshop on Graph-Theoretic Concepts in Computer Science*. 159–167.

Jelle Hellings, George H. L. Fletcher, and Herman J. Haverkort. 2012. Efficient external-memory bisimulation on DAGs. In *SIGMOD*. 553–564.

Xiaocheng Hu, Yufei Tao, and Chin-Wan Chung. 2013. Massive graph triangulation. In *SIGMOD*. 325–336.

Alon Itai and Michael Rodeh. 1978. Finding a Minimum Circuit in a Graph. *SIAM J. of Comp.* 7, 4 (1978), 413–423.

Madhav Jha, C. Seshadhri, and Ali Pinar. 2013. A space efficient streaming algorithm for triangle counting using the birthday paradox. In *SIGKDD*. 589–597.

Mihail N. Kolountzakis, Gary L. Miller, Richard Peng, and Charalampos E. Tsourakakis. 2012. Efficient Triangle Counting in Large Graphs via Degree-Based Vertex Partitioning. *Internet Mathematics* 8, 1-2 (2012), 161–185.

Matthieu Latapy. 2008. Main-memory triangle computations for very large (sparse (power-law)) graphs. *TCC* 407, 1-3 (2008), 458–473.

Min Chih Lin, Francisco J. Soulignac, and Jayme Luiz Szwarcfiter. 2012. Arboricity, h-index, and dynamic algorithms. *TCC* 426 (2012), 75–90.

Bruno Menegola. 2010. *An external memory algorithm for listing triangles*. Technical Report. Universidade Federal do Rio Grande do Sul.

C. St. J. A. Nash-Williams. 1964. Decomposition of finite graphs into forests. *Journal of the London Mathematical Society* 39, 1 (1964), 12.

Rasmus Pagh and Francesco Silvestri. 2014. The input/output complexity of triangle enumeration. In *PODS*. 224–233.

Thomas Schank. 2007. *Algorithmic Aspects of Triangle-Based Network Analysis*. Ph.D. Dissertation. Universitat Karlsruhe, Fakultat fur Informatik.

Thomas Schank and Dorothea Wagner. 2005. Finding, Counting and Listing All Triangles in Large Graphs, an Experimental Study. In *Workshop on Experimental Algorithms (WEA)*. 606–609.

Siddharth Suri and Sergei Vassilvitskii. 2011. Counting triangles and the curse of the last reducer. In *WWW*. 607–614.

Chih-Hua Tai, Philip S. Yu, De-Nian Yang, and Ming-Syan Chen. 2011. Privacy-preserving social network publication against friendship attacks. In *SIGKDD*. 1262–1270.

Jia Wang and James Cheng. 2012. Truss decomposition in massive networks. *PVLDB* 5, 9 (2012), 812–823.

Nan Wang, Jingbo Zhang, Kian-Lee Tan, and Anthony K. H. Tung. 2010. On triangulation-based dense neighborhood graph discovery. *PVLDB* 4, 2 (2010), 58–68.

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

Peixiang Zhao, Charu C. Aggarwal, and Min Wang. 2011. gSketch: On Query Estimation in Graph Streams. *PVLDB* 5, 3 (2011), 193–204.