Massive Graph Triangulation*

Xiaocheng Hu[1] Yufei Tao[1,2] Chin-Wan Chung[2,3]
[1]Department of Computer Science and Engineering, CUHK, Hong Kong
[2]Division of Web Science and Technology, KAIST, Korea
[3]Department of Computer Science, KAIST, Korea

ABSTRACT

This paper studies I/O-efficient algorithms for settling the classic *triangle listing* problem, whose solution is a basic operator in dealing with many other graph problems. Specifically, given an undirected graph G, the objective of triangle listing is to find all the cliques involving 3 vertices in G. The problem has been well studied in internal memory, but remains an urgent difficult challenge when G does not fit in memory, rendering any algorithm to entail frequent I/O accesses. Although previous research has attempted to tackle the challenge, 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 outperformed the existing competitors by a factor over an order of magnitude in our extensive experimentation.

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

Keywords
Graph, triangle, I/O-efficient algorithm

1. INTRODUCTION

In this paper, we revisit the classic *triangle listing problem*. The input is a graph[1] $G = (V, E)$, where V (E) is the set of vertices (edges). A *triangle* is a clique of 3 vertices u, v, w in G, and is denoted as Δ_{uvw}. The goal of triangle listing is to output all the triangles in G. For instance, if G is the graph in Figure 1, the set of triangles is $\{\Delta_{123}, \Delta_{234}, \Delta_{346}, \Delta_{368}, \Delta_{456}, \Delta_{568}\}$.

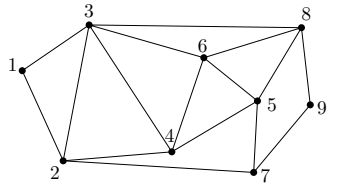


Figure 1: The input graph in our running example

The importance of triangle listing has long been recognized in the literatures of database, network analysis, knowledge discovery, and graph theory:

Dense Subgraph Mining. Given a graph G, a *dense neighborhood graph* (DN-graph) [23] is a subgraph of G, where each pair of connected vertices share at least a number of common neighbors. The most efficient known algorithm [23] 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* [4] 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, [19]], 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 [19] (hence, fast triangle listing is again the key).

***k*-truss.** Given a graph G, its *k-truss* ($k \geq 3$) [10] 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 [10, 22]. Not surprisingly, the state-of-the-art algorithm [22] for k-truss computation deploys triangle listing as an initial step.

Network Measurement. A popular approach in studying networks is to interpret the measurements on certain key aspects. A well-known measurement is the (local) *clustering coefficient* [24]. Given a vertex v in a graph G, 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. A large clustering coefficient indicates high density around v (i.e., many edges among the neighbors of v). The calcu-

*This work was supported in part by the WCU (World Class University) program under the National Research Foundation of Korea, and funded by the Ministry of Education, Science and Technology of Korea (Project No: R31-30007). Xiaocheng Hu and Yufei Tao were also supported in part by projects GRF 4166/10, 4165/11, and 4164/12 from HKRGC. Chin-Wan Chung was also supported in part by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MEST) (No. 2012-0000182), and in part by Microsoft Research Asia through KAIST-Microsoft Research Collaboration Center.

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'13, June 22–27, 2013, New York, New York, USA.
Copyright 2013 ACM 978-1-4503-2037-5/13/06 ...$15.00.

lation of such coefficients requires computing $t(v)$ for all vertices v. Interestingly, as shown in [9], the most I/O-efficient algorithm for this purpose resorts to triangle listing, after which $t(v)$ can be obtained by simple counting.

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 interests (see [6, 9, 13] 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 [3]. 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 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) [17]. 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) [11]. 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 [8, 9]. 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 [9] 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 [9] presented two algorithms 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.

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 [9]. Somewhat surprisingly, we show that the term $|E|^2/(MB)$ is inevitable, namely, it is impossible to achieve $o(|E|^2/(MB))$ I/Os for all inputs even if $M \ll |E|$.[2] This stands in sharp contrast to triangle listing in memory, where $o(|E|^2)$-time algorithms are well known (e.g., the algorithm of [7] 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 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 [14], and is much smaller for graphs in reality [7, 16]. 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.

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 [9] 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.

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. We are the first to put all the known algorithms into a direct cross comparison (the papers [8, 9] 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 [9] 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 [24] and the modern popular *recursive matrix* (R-MAT) model [5]. 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 ana-

[2]The inevitability of $E^2/(MB)$ is trivial for $M = \Omega(|E|)$.

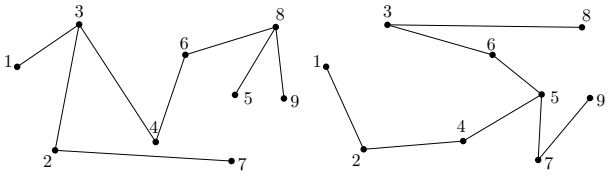


Figure 2: Two edge-disjoint trees covering all the edges of the graph in Figure 1

lyzes 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 an extensive experimental evaluation to demonstrate the superiority of the proposed technique over the existing solutions. Finally, Section 7 concludes the paper with a summary of our findings.

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 apparently no single forest can cover all the edges.

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 1 ([18]). *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 1 ([7]). *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.*

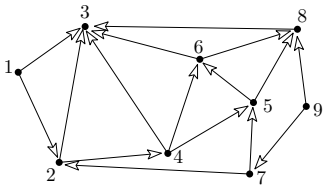


Figure 3: An oriented version of Figure 1

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 [7, 16]).

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:

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

Chiba and Nishizeki [7] observed a delicate connection between the above and arboricity:

PROPOSITION 2 ([7]). *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 1. 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.

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 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

The triangle listing problem has been extensively studied in internal memory, yielding a large number of algorithms [7, 12, 14,

15, 20]. 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 this section, we extend the description in Section 1 about the existing I/O-efficient algorithms. Focus will be devoted to the solutions of [9] since they are the state of the art.

3.1 EM-CF

External memory compact forward (EM-CF) [17] 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 [17] 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) [11] 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 $(u, w) \in E^\star$.
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 $L = \{(1, \{2, 3\}), (2, \{3, 4\}), (4, \{3, 6\}), (4, \{5, 6\}), (4, \{3, 5\}), (5, \{6, 8\}), (6, \{3, 8\}), (7, \{2, 5\}), (9, \{7, 8\})\}$, 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 [11] showed how to perform the two steps in $O(|E|/B + |L|/B \cdot \log_{M/B}(|E|/B))$ I/Os. Menegola [17] 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 [8, 9] 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 ($1 \leq i \leq p$) 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.

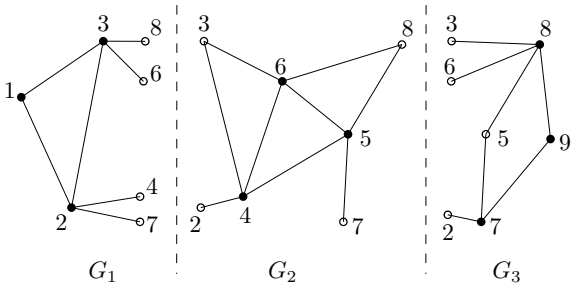


Figure 4: Extended subgraphs

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

$\boldsymbol{A_1}$**:** 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:

$$\boldsymbol{A_2}\text{:}\ |V| \leq M.$$

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 [9] 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

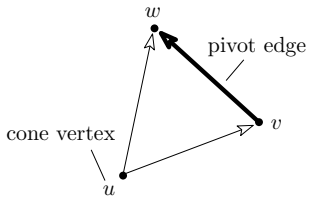


Figure 5: A triangle in $G^\star$ and its pivot edge ($u \prec v \prec w$)

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 [9], 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 [9] showed that under the assumption:

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

DGP and RGP ensure I/O complexity[3] $O(|E|^2/(MB) + K/B)$, 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 $O(|E|^2/(MB) \cdot \log_{M/B}(|E|/B) + K/B)$.

4. A NEW 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.

[3]The complexity is expected for RGP.

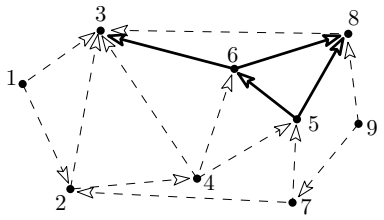


Figure 6: Illustration of Step 2 (solid edges are in E_{mem})

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

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}

```
1 obtain V_mem from E_mem
2 for each vertex u ∈ V do
3     read N+(u) from disk to acquire N_mem(u) in memory
4     S ← the set of edges from u to the vertices in N_mem(u)
5     find in S ∪ E_mem all the triangles where u is the cone vertex
6     release N_mem(u) and S from memory
7 return
```

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:

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

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

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

Hence, by setting c appropriately[4], 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

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

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 $O(|E|^2/(MB) + K/B)$.

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 ← ∅
for each vertex v ∈ V whose edges have never entered E_mem do
    if |N+(v)| + |E_mem| ≤ 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}

```
1  obtain V_mem and V+_mem from E_mem
2  build hash structures on V_mem, V+_mem, and E_mem
3  for each vertex u ∈ V do
4      read N+(u) from disk to acquire N_mem(u) in memory
5      obtain N+_mem(u) from N_mem(u) in memory
6      for each v ∈ N+_mem(u) do
7          for each w ∈ N_mem(u) do
8              if v ≠ w and edge (v, w) ∈ E_mem then
9                  output Δ_uvw
10     release N_mem(u) and N+_mem(u) from memory
11 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 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:

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

□

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

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

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

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

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

LEMMA 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:

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

Hence:

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

Therefore:

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

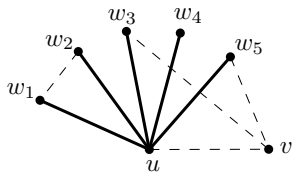


Figure 7: Illustration for the conversion algorithm (S consists of the solid edges)

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 [1]) 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[5] 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:

- **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.

[5]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.

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 1. *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.*

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, implying that the I/O complexity in Theorem 1 is already optimal up to a constant factor. 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 1.

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 2. *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:

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

which is $O(\alpha|E|)$ by Lemma 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 2 is separated from the old result whenever $\alpha = o(\sqrt{|E|})$. Moreover, Theorem 2 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 1), in which case EM-NI finishes in $O(SORT(|E|))$ I/Os.

Theorem 2 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 $M \geq |V|$*. To see this, first notice that:

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

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 $M \geq |V|$,

$$\alpha > |E|/|V| \geq |E|/M$$

which makes

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

namely, the I/O complexity in Theorem 2 is never better than that in Theorem 1.

5.2 RGP

Recall from Section 3 that the graph partition framework [9], which is the state of the art, relies on a key assumption $\boldsymbol{A_1}$ to attain its I/O efficiency. Chu and Cheng [9] 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 3 ([2]). *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 3. *Under the condition:*

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

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 3) 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, 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 3 shows that $\boldsymbol{A_1}$ holds with extremely high probability as long as the memory is not too small. 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 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. 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 6.2 (6.3) evaluates the efficiency of alternative solutions on real (synthetic) datasets.

6.1 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 [1], 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.

6.2 Performance on Real Data

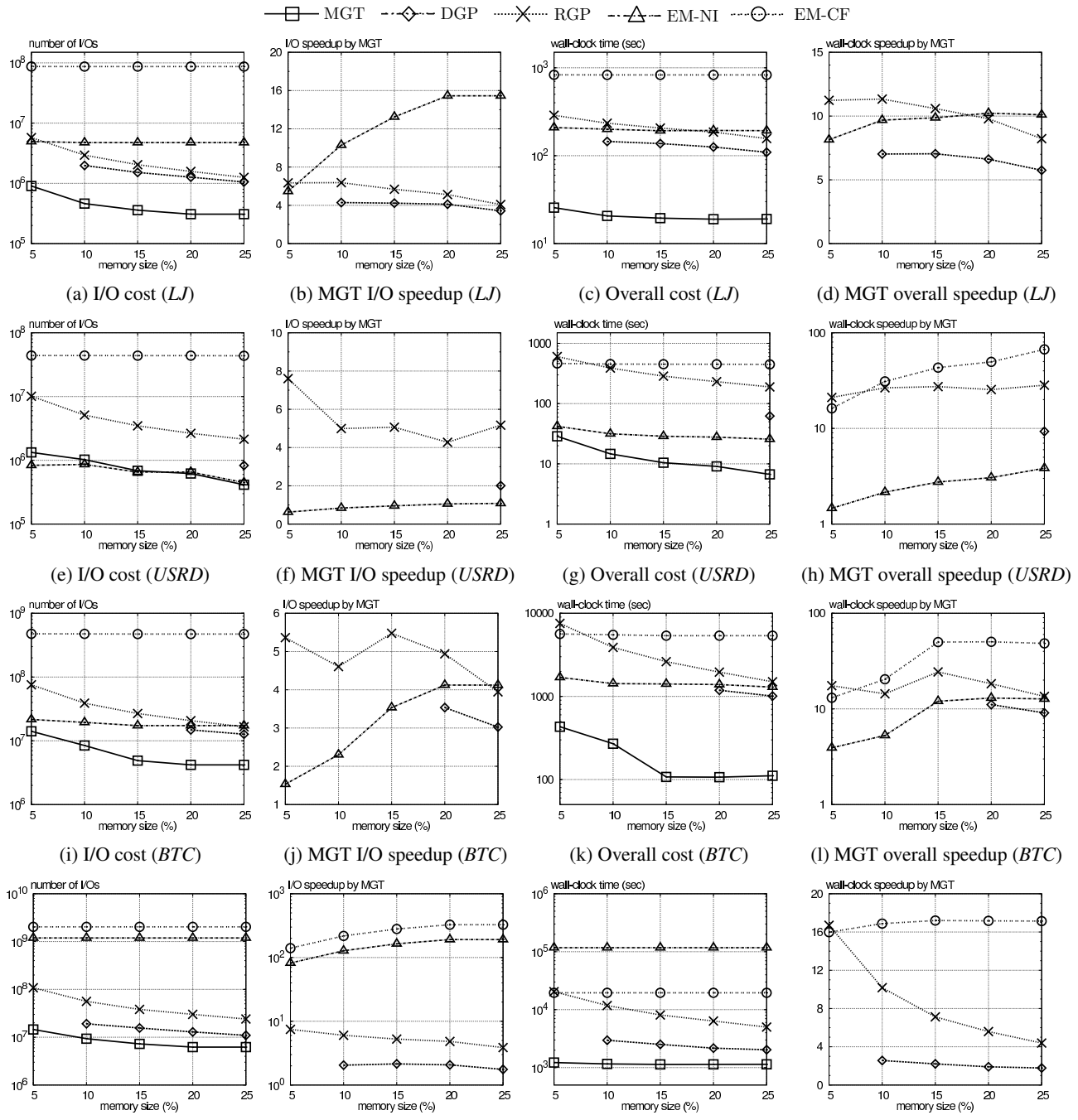
Datasets and Methodology. We deployed exactly the same real datasets as were used in [9] where state-of-the-art DGP and RGP were developed. These datasets are named *LJ*, *USRD*, *WebUK* and *BTC* respectively, whose meta information is displayed as below[6]:

	LJ	*USRD*	*BTC*	*WebUK*
$\|V\|$	4,846,609	23,947,347	164,660,997	62,338,347
$\|E\|$	42,851,237	28,854,312	386,411,047	938,715,528
$\|E\|/\|V\|$	8.84	1.20	2.35	15.06
disk size	364 M	403 M	4.1 G	7.5 G

Table 1: Meta data of real graphs

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

[6]The graph sizes listed here are different from those provided in [9], which, however, is due to the errors in [9], as has been verified by our communication with the authors of [9].



(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*)

Figure 8: Efficiency comparison on real graphs

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*). Finally, *WebUK* captures the hyperlinks (i.e., edges) among a set of web pages (i.e., vertices) gathered for investigation of web spams (*http://barcelona.research.yahoo.net*).

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 of 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 5% of the graph's disk size (see Table 1) to 25%, and in the meantime, compared the performance of different algorithms.

Results. Figure 8 presents all the results of the experiments on the real graphs. In the first row, Figure 8a 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 8b shows the I/O speedups of MGT over the other algorithms as the memory grows. Figures 8c and 8d 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, and last rows of Figure 8 present the outcome of the same experiments on *USRD*, *BTC* and *WebUK*, respectively.

In Figure 8, 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 8b 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 at least one setting on every dataset: specifically, $M = 5\%, \leq 20\%$, $\leq 15\%$ and $= 5\%$ for *LJ*, *USRD*, *BTC* and *WebUK*, 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, on the 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 8 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 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 8o). 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.

6.3 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. [5], 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 [21, 25]). Finally, duplicate edges were removed by a clean-up process.
- **Small World** (*S-WORLD*): This classic model was first described by Watts and Strogatz [24]. 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[7]. 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 [24].

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):

$\|V\|$	16×2^{20}	32×2^{20}	48×2^{20}	64×2^{20}	80×2^{20}
$\|E\|$	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

Table 2: Meta data of synthetic graphs (all distributions)

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.

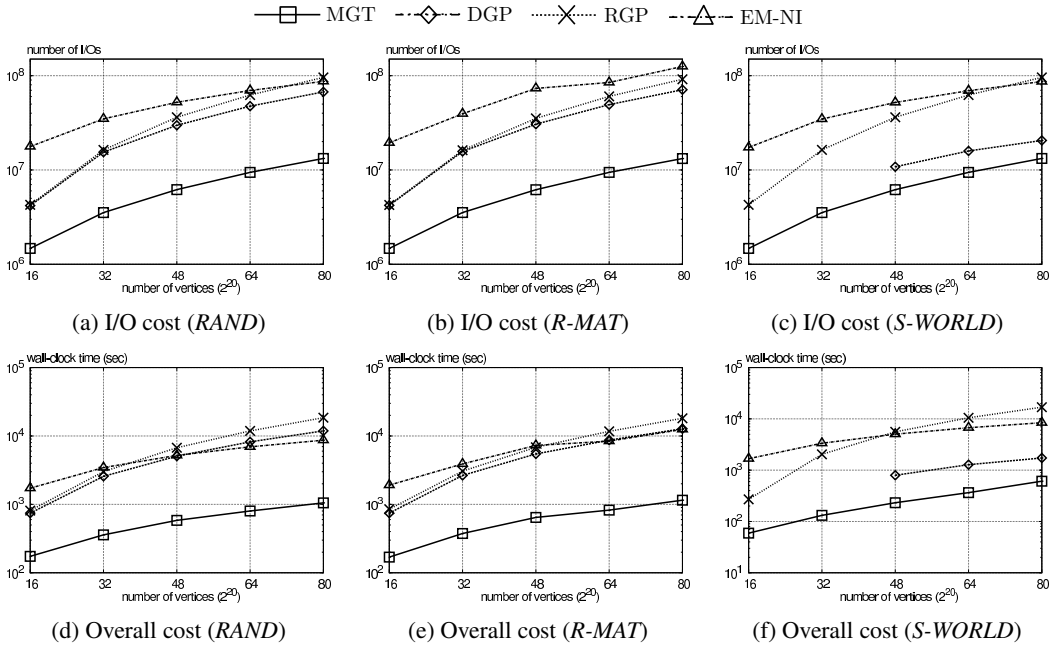
Results. Figure 9 demonstrates the comparison results of MGT, DGP, RGP and EM-NI on synthetic graphs, by focusing on the I/O and wall-clock time in the first and second rows, respectively. DGP has no results on *S-WORLD* graphs when $|V| \leq 32 \times 2^{20}$ because it failed due to the violation of $\boldsymbol{A_1}$.

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.

7. CONCLUSIONS

Triangle listing is an important classic problem on graphs that has numerous applications in different domains. Although it has been well studied in internal memory, solving it 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.

[7]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.



(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*)

Figure 9: Efficiency comparison on synthetic graphs

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 both optimal in the worst case.

8. REFERENCES

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *CACM*, 31(9):1116–1127, 1988.

[2] N. Alon and J. H. Spencer. *The Probabilistic Methods*. Wiley, New York, 2nd edition, 2000.

[3] E. Bakshy, I. Rosenn, C. Marlow, and L. A. Adamic. The role of social networks in information diffusion. *CoRR*, 2012.

[4] V. Batagelj and M. Zaversnik. Short cycle connectivity. *Discrete Mathematics*, 307:310–318, 2007.

[5] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *SDM*, 2004.

[6] J. Cheng, Y. Ke, A. W.-C. Fu, J. X. Yu, and L. Zhu. Finding maximal cliques in massive networks. *TODS*, 36(4):21, 2011.

[7] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. of Comp.*, 14(1):210–223, 1985.

[8] S. Chu and J. Cheng. Triangle listing in massive networks and its applications. In *SIGKDD*, pages 672–680, 2011.

[9] S. Chu and J. Cheng. Triangle listing in massive networks. *TKDD*, 6(4):17, 2012.

[10] J. Cohen. Graph twiddling in a mapreduce world. *Computing in Science and Engineering*, 11(4):29–41, 2009.

[11] R. Dementiev. *Algorithm engineering for large data sets hardware, software, algorithms*. PhD thesis, Saarland University, 2006.

[12] D. Eppstein and E. S. Spiro. The h-index of a graph and its application to dynamic subgraph statistics. In *WADS*, pages 278–289, 2009.

[13] J. Hellings, G. H. L. Fletcher, and H. J. Haverkort. Efficient external-memory bisimulation on dags. In *SIGMOD*, pages 553–564, 2012.

[14] A. Itai and M. Rodeh. Finding a minimum circuit in a graph. *SIAM J. of Comp.*, 7(4):413–423, 1978.

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

[16] M. C. Lin, F. J. Soulignac, and J. L. Szwarcfiter. Arboricity, h-index, and dynamic algorithms. *TCC*, 426:75–90, 2012.

[17] B. Menegola. An external memory algorithm for listing triangles. Technical report, Universidade Federal do Rio Grande do Sul, 2010.

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

[19] T. Schank. *Algorithmic Aspects of Triangle-Based Network Analysis*. PhD thesis, Universitat Karlsruhe, Fakultat fur Informatik, 2007.

[20] T. Schank and D. Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *Workshop on Experimental Algorithms (WEA)*, pages 606–609, 2005.

[21] C.-H. Tai, P. S. Yu, D.-N. Yang, and M.-S. Chen. Privacy-preserving social network publication against friendship attacks. In *SIGKDD*, pages 1262–1270, 2011.

[22] J. Wang and J. Cheng. Truss decomposition in massive networks. *PVLDB*, 5(9):812–823, 2012.

[23] N. Wang, J. Zhang, K.-L. Tan, and A. K. H. Tung. On triangulation-based dense neighborhood graph discovery. *PVLDB*, 4(2):58–68, 2010.

[24] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, 1998.

[25] P. Zhao, C. C. Aggarwal, and M. Wang. gsketch: On query estimation in graph streams. *PVLDB*, 5(3):193–204, 2011.