

# Improving I/O Complexity of Triangle Enumeration

Yi Cui, Di Xiao, Daren B.H. Cline, and Dmitri Loguinov  
Texas A&M University, College Station, TX 77843 USA  
{yicui, di}@cse.tamu.edu, dcline@stat.tamu.edu, dmitri@cse.tamu.edu

**Abstract**—In the age of big data, many graph algorithms are now required to operate in external memory and deliver performance that does not significantly degrade with the scale of the problem. One particular area that frequently deals with graphs larger than RAM is *triangle listing*, where the algorithms must carefully piece together edges from multiple partitions to detect cycles. In recent literature, two competing proposals (i.e., Pagh and PCF) have emerged; however, neither one is universally better than the other. Since little is known about the I/O cost of PCF or how these methods compare to each other, we undertake an investigation into the properties of these algorithms, model their I/O cost, understand their shortcomings, and shed light on the conditions under which each method defeats the other. This insight leads us to develop a novel framework we call *Trigon* that surpasses the I/O performance of both previous techniques in all graphs and under all RAM conditions.

## I. INTRODUCTION

Triangle listing is a field of graph mining that aims to identify all three-node cycles in undirected graphs  $G$ . This problem has many applications in theory and practice [2], [3], [4], [7], [10], [22], [30], [31], [34], [37], including areas outside of computer science [12], [13], [15], [20], [21], [29], [33]. Due to the scale of modern graphs (i.e., billions/trillions of edges) and anticipated emergence of even bigger datasets in the future, reducing I/O complexity during graph manipulation has become an important topic.

Triangle listing involves two components – *in-memory search*, whose purpose is to find all relevant motifs (i.e., triangles) within portions of the graph loaded in RAM, and *graph partitioning*, whose responsibility is to chunk  $G$  into such pieces that ensure no triangle is missed or discovered more than once. In-memory search entails verification of neighboring relationships between all pairs of candidate nodes. The majority of these solutions [1], [5], [6], [11], [14], [16], [18], [24], [25], [26], [27], [28] can be expressed under the umbrella of 18 vertex/edge-iterator algorithms [8], [35], where a single method  $E_1$  has emerged as a clear winner.

In graph partitioning, however, the situation is more interesting. As of this writing, the two most-efficient approaches to splitting the graph are a coloring scheme called Pagh [23] and the PCF framework from [8]. The main caveat is that the former has lower I/O bounds on complete graphs, while the latter on sparse, i.e., neither one is better than the other. Besides I/O, execution time also depends on the amount of hash-table lookups, which is a function of the partitioning algorithm. This raises a possibility that some methods might exhibit less I/O, but require more CPU cost.

It currently remains unclear under what specific conditions Pagh is better than PCF in terms of I/O, which of them should

be chosen for a particular  $G$ , why one approach may have inherent advantages over the other, and whether it is possible to design a single algorithm that can perform better than both of these techniques. If so, how does one decide on its parameters in order to achieve the smallest runtime? Our goal in the paper is to address these questions.

## A. Overview of Results

We start by analyzing the asymptotics of I/O in Pagh and PCF, aiming to achieve an understanding of their strengths and weaknesses. While the former has a simple model, the overhead of the latter is a complex function of the acyclic orientation  $\theta$ , the resulting directed graph  $G_\theta$ , and specific traversal order of nodes in each triangle. We derive the exact overhead of PCF; however, this formula proves difficult for closed-form analysis. We therefore obtain tight upper/lower bounds on its growth rate, which are then used in the comparison against Pagh. In general, PCF has the highest advantage when the graph is sparse, the variance of out-degree is small, and RAM is growing slowly with the number of edges  $m$ . Pagh wins when these conditions are reversed. As the number of nodes  $n \rightarrow \infty$ , our results demonstrate that under the best scenario for PCF, it beats Pagh by a factor of  $n$ . In the worst case, it loses by a factor of  $\sqrt{n}$ .

Our investigation reveals that each method brings a significant amount of redundant edges into RAM, but *they do so under different conditions*. This gives hope that a single method can combine the strengths of these techniques and simultaneously avoid their individual drawbacks. To this end, we first generalize graph partitioning to cover all possible ways to execute vertex/edge iterators in external memory. Not surprisingly, both Pagh and PCF, as well as previous techniques based on MGT [11], [14], are all special cases of this unifying framework. Under its umbrella, we then create a particular scheme, which we call *Trigon*, that leverages the lessons learned from the preceding analysis. We show that *Trigon's* I/O is never worse, and in many cases much better, than either of its predecessors. Not only that, but it is also the first method that allows balancing between I/O and CPU cost in order to achieve the smallest runtime.

## II. PRELIMINARIES

Assume a simple undirected graph  $G = (V, E)$  with  $n$  nodes and  $m$  edges. Detection of triangles requires a large number of neighbor checks. This overhead can be substantially reduced by performing an acyclic orientation on  $G$ . In recent literature [35], orientation is modeled as some permutation  $\theta$

**Algorithm 1:** Method  $E_1$  processing source node  $u$  in memory

---

```

1 foreach  $v \in N_u^+$  do  $\triangleleft$  visit all out-neighbors
2   | find  $N_v^+$  using a hash table
3   |  $W = \text{Intersect}(N_u^+, N_v^+) \triangleleft$  intersect two sorted out-lists
4   | foreach  $w \in W$  do report  $\Delta_{uvw}$ 

```

---

that decides the direction of each edge. Specifically, each node  $u$  is placed into a new location  $\theta(u)$ , the permuted sequence of nodes is relabeled from 1 to  $n$ , and all edges are directed from larger to smaller node IDs. This splits each neighbor list  $N_u$  into out-neighbors  $N_u^+$  and in-neighbors  $N_u^-$ , with the corresponding graphs  $G_\theta^+$  and  $G_\theta^-$ . Note that adjacency lists are sorted by the new node label.

Throughout the paper, we use orientation  $\theta_D$  that arranges the nodes in *descending* order of undirected degree  $d_u$ . This permutation, also known as *largest-first* in graph theory [19], [32], is optimal for both the fastest edge iterator  $E_1$  and its corresponding PCF algorithms in [8], [35]. Since Pagh’s performance is independent of  $\theta$ , this choice does not affect its I/O. Letting  $Y_u = |N_u^-|$  and  $X_u = |N_u^+|$  be the respective in/out-degrees of  $u$  in directed  $G_\theta$ , it follows that  $X_u + Y_u = d_u$  and  $\sum_{u=1}^n X_u = \sum_{u=1}^n Y_u = m$ .

After orientation,  $E_1$  searches for all directed triangles  $\Delta_{uvw}$ , where  $u > v > w$ . This is done by calling Algorithm 1 for each source node  $u$  in  $G_\theta^+$ . The CPU cost consists of the number of hash-table lookups to retrieve  $N_v^+$  and the size of intersection in Line 3. For in-memory operation, the former is just  $\gamma(n) = m - n$ , while the latter is given by [35]

$$\rho(n) = \sum_{u=1}^n \left( \frac{X_u(X_u - 1)}{2} + X_u Y_u \right). \quad (1)$$

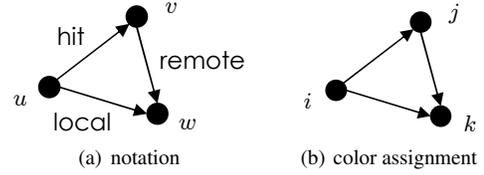
When  $E_1$  is used in external memory, the partitioning scheme must ensure that all three edges of a triangle are eventually present in RAM *at the same time*. This can be accomplished by holding one of them in RAM and streaming the other two from disk (e.g., MGT [14], PCF-1B [8]), keeping two in RAM and streaming the third one (e.g., PCF-1A [8]), or loading all three simultaneously [23], [24]. Because of the random lookup needed to obtain  $N_v^+$ , it does not currently appear feasible to stream all three edges.

### III. ANALYSIS OF PAGH

The original Pagh algorithm [23] has certain details omitted from the paper, while others are sketched at a high level. While I/O complexity of this method has known bounds in the  $O(\cdot)$  notation [23], numerical comparison between the algorithms, as well as implementation, both require the missing constants. Additionally, since coupling of Pagh to  $E_1$  has not been discussed before, we perform this extension as well. Note that the proofs omitted from the paper can be found in [9].

#### A. Algorithm

Pagh assigns to each node  $u$  a uniformly random color  $\phi_u$  drawn from a set  $1, 2, \dots, c$ , where  $c = \sqrt{m/M}$  and  $M$  is

Fig. 1. Directed triangle ( $u > v > w$ ).**Algorithm 2:** Graph partitioning in Pagh

---

```

1 for  $u = 1$  to  $n$  do
2   |  $i = \phi_u \triangleleft$  color of source node
3   | for  $j = 1$  to  $c$  do  $\triangleleft$  color of destination nodes
4   |   |  $N_{uj}^+ = N_u^+ \cap V_j \triangleleft$  out-neighbors of color  $j$ 
5   |   | write  $(u, N_{uj}^+)$  to subgraph  $E_{ij}^+$ 

```

---

RAM size in edges. Then, all nodes are split into  $c$  subsets  $V_1, \dots, V_c$  such that

$$V_i = \{u \in V \mid \phi_u = i\}. \quad (2)$$

This can be visualized with the help of Fig. 1. Part (a) shows a directed triangle  $(uvw)$ , as seen by Algorithm 1, with three uniquely-identifiable edges. From  $u$ ’s perspective, edge  $(uv)$  results in a *hit* on the hash table,  $(uw)$  participates in *local* intersection at  $u$ , and  $(vw)$  is part of *remote* intersection. The mapping to colors is shown in part (b) of the figure, where  $i$  refers to the color of the largest node,  $j$  to that of the middle, and  $k$  to that of the smallest.

The edges of  $G_\theta^+ = (V, E_\theta^+)$  are partitioned into  $c^2$  subsets  $\{E_{ij}^+\}$  according to the color of source/destination nodes, i.e.,

$$E_{ij}^+ = \{(u, v) \in E_\theta^+ \mid \phi_u = i, \phi_v = j\}. \quad (3)$$

This is demonstrated in Algorithm 2, which splits the out-graph into tuples  $(u, N_{uj}^+)$ , where  $N_{uj}^+$  contains  $u$ ’s out-neighbors of color  $j$ . Note that the expected size of each  $V_i$  is  $n/c$  and that of  $E_{ij}^+$  is  $m/c^2$  edges. After this preprocessing step, Pagh suggests using MGT [14] to find triangles in each of the  $c^3$  triples  $(E_{ij}^+, E_{jk}^+, E_{ik}^+)$ , where the remote edge belongs to  $E_{jk}^+$ . MGT relies on vertex iterator  $T_1$  [8], which is 15 – 80 times slower than  $E_1$  on real graphs. Additionally, it does not by default handle heterogeneous partitions (i.e., hit/remote/local edges all being stored separately). To create a fully working system, we need a few refinements.

#### B. Pagh+

Assuming partitions are well-balanced, i.e., all have size  $M$  within some tolerance, MGT can be combined with  $E_1$  to efficiently solve the problem. Algorithm 3, which we call Pagh+, loads remote edges  $E_{jk}^+$  into RAM and then scans the other two subgraphs. Since Algorithm 2 writes source nodes in the same order for all subgraphs, Pagh+ can obtain both hit and local lists of each  $u$  by concurrently reading  $E_{ij}^+$  and  $E_{ik}^+$ . The resulting system detects each triangle once and performs no more intersections than in-memory  $E_1$ .

**Theorem 1.** *Pagh+ needs  $I_P(n) = (2c - 1)m$  edges of I/O.*

**Algorithm 3:** Pagh+ handling one remote graph

```

1 load  $E_{jk}^+ = \{(v, N_{vk}^+)\}$  in RAM; set up hash table to source nodes
2 for  $i = 1$  to  $c$  do
3   while file  $E_{ij}^+$  not empty do
4     load  $(u, N_{uj}^+)$  from  $E_{ij}^+$  and  $(u, N_{uk}^+)$  from  $E_{ik}^+$ 
5     foreach  $v \in N_{uj}^+$  do  $\triangleleft$  visit all neighbors in the hist list
6       find remote list  $N_{vk}^+$  using the hash table
7        $W = \text{Intersect}(N_{uk}^+, N_{vk}^+) \triangleleft$  local/remote lists
8       foreach  $w \in W$  do report  $\Delta_{uvw}$ 

```

**Algorithm 4:** PCF-1A graph partitioning

```

1 for  $u = 1$  to  $n$  do  $\triangleleft$  iterate over all nodes
2   for  $i = 1$  to  $p$  do  $\triangleleft$  go through each partition
3      $H_{ui} = N_u^+ \cap [a_{i+1}, n] \triangleleft$  pruned hit list
4      $L_{ui} = N_u^+ \cap [a_i, a_{i+1}] \triangleleft$  local/remote list
5     if  $L_{ui} \neq \emptyset$  then
6       write  $(u, L_{ui})$  to  $G_\theta^r(i) \triangleleft$  remote file  $i$ 
7       if  $H_{ui} \neq \emptyset$  then
8         write  $(u, H_{ui})$  to  $G_\theta^c(i) \triangleleft$  companion file  $i$ 

```

Since  $(2c - 1)m = 2m^{1.5}/\sqrt{M} - m$ , Pagh+ has the best multiplicative constant in the literature. The closest alternative [24] uses the undirected graph  $G$ , assigns direction to *colors* rather than edges, and increases  $c$  to  $\sqrt{5m/M}$  such that certain combinations of subgraphs fit in RAM. This leads to  $\sqrt{5}m^{1.5}/\sqrt{M} \approx 2.2m^{1.5}/\sqrt{M}$  total I/O, which is worse than the result above. Another potential drawback to this approach is usage of undirected graphs, where  $E_1$  has to perform unnecessary intersections [35].

It is also simple to obtain the number of hash-table lookups in Pagh+. When they become a CPU bottleneck,  $E_1$  may essentially deteriorate into  $T_1$  and lose its advantages. The next result shows that this value is linear in  $c$ .

**Theorem 2.** *Pagh+ performs  $\gamma_P(n) = cm$  lookups.*

### C. Discussion

Slightly unbalanced partition sizes  $|E_{ij}^+|$  due to randomness of color assignment are a minor issue in practice. However, when the graph contains nodes with large degree, Pagh requires a different algorithm. One example is the star graph, where all nodes connect to a center node of some color  $k$ . To avoid optimizations that discard  $(ijk)$  if any of the subgraphs is empty, the star graph can be augmented with  $c^2$  random edges between the leaf nodes. Neglecting small terms, Algorithm 2 produces  $c$  partitions of size  $m/c \gg M$ . In fact, two of the three subgraphs involving color  $k$  have size  $m/c$ .

Pagh [23] handles this case by isolating nodes of degree larger than  $\sqrt{mM}$  into a separate category. Each of them requires sorting up to  $m$  edges on disk. Since there are no more than  $c$  such nodes, the I/O can be bounded by  $c \cdot \text{sort}(m) \sim cm \log m / \log M$  edges. If RAM scales as some power of  $m$ , as assumed in [23], we get the usual  $O(m^{1.5}/\sqrt{M})$ ; however, the hidden constants may be non-negligible. But more importantly, the CPU cost for sorting the graph  $c$  times may be quite hefty.

**Algorithm 5:** PCF-1B graph partitioning

```

1 for  $u = 1$  to  $n$  do  $\triangleleft$  iterate over all nodes
2   for  $i = 1$  to  $\phi_u - 1$  do  $\triangleleft$  go through each partition below  $u$ 
3      $H_{ui} = N_u^+ \cap [a_i, a_{i+1}] \triangleleft$  hit list
4      $L_{ui} = N_u^+ \cap [1, a_{i+1}] \triangleleft$  local list
5     if  $H_{ui} \neq \emptyset$  and  $|L_{ui}| \geq 2$  then
6       write  $(u, L_{ui})$  to  $G_\theta^c(i) \triangleleft$  save to companion file  $i$ 
7   if  $N_u^+ \neq \emptyset$  then  $\triangleleft$  out-degree non-zero?
8     write  $(u, N_u^+)$  to  $G_\theta^r(\phi_u) \triangleleft$  remote file of  $u$ 's color

```

On the bright side, Pagh does not impose much restriction on minimum RAM or disk size. Setting  $c = n$ , it is possible to create subgraphs that contain just one edge each, resulting in  $O(1)$  memory consumption. Furthermore, its disk-space requirement is only  $m$  edges.

## IV. ANALYSIS OF PCF

The I/O complexity of PCF is quite peculiar due to the dependency on the underlying graph. This section develops the methodology and insight that not only sheds light on PCF, but also helps later with comparison against Pagh+ and design of our new method.

### A. Operation

PCF [8] is a suite of six algorithms 1A, 1B, 2A, 2B, 6A, 6B. All of them partition the graph along the remote edge of the corresponding in-memory algorithm (i.e.,  $E_1$ ,  $E_2$ , and  $E_6$ ). In the notation of Fig. 1(a), these are  $(vw)$  for 1A/1B,  $(uw)$  for 2A/2B, and  $(uv)$  for 6A/6B. The A variants split based on the destination node of the remote edge, while the B versions do the same on the source node. After preprocessing, PCF sequentially loads chunks of  $G_\theta^+$  in RAM and scans so-called *pruned companion files* to obtain the missing edges.

Method  $E_1$  requires PCF-1A/1B, which we review and analyze next. Both of them start by dividing the set of nodes  $V$  into  $p = m/M$  non-overlapping subsets  $V_1, \dots, V_p$ . PCF utilizes *sequential* partitions such that  $u \in V_i$  iff  $u \in [a_i, a_{i+1})$ , where boundaries  $\{a_i\}$  are determined by load-balancing either the in-degree (1A) or out-degree (1B) of each partition to equal memory size  $M$ . To be consistent with other parts of the paper, we say that nodes in  $V_i$  have color  $i$ . We also use the same function  $\phi_u$  to map  $u$  to its color.

File  $G_\theta^+$  is split into  $p$  disjoint subgraphs  $G_\theta^r(1), \dots, G_\theta^r(p)$  that contain all remote edges  $(vw)$  matching the corresponding color. Specifically,  $(vw)$  is written into  $G_\theta^r(i)$  iff  $w \in V_i$  in PCF-1A and  $v \in V_i$  in PCF-1B. The corresponding companion files  $G_\theta^c(i)$  contain nodes  $u$  and their hit/local lists, but only if they are relevant to partition  $i$ . For example, PCF-1B skips node  $u$  unless it has at least one neighbor of color  $i$  and another neighbor with a smaller ID. While [8] has a comprehensive algorithm that covers all six methods, it may be difficult to parse. We therefore find it useful to show the minimal versions of PCF-1A and 1B using Algorithms 4-5.

### B. Model

Since  $\sum_{i=1}^p |G_\theta^r(i)| = m$  is fixed, the main open question is companion I/O, i.e.,  $\sum_{i=1}^p |G_\theta^c(i)|$ . For a source node  $u$ ,

suppose  $\phi_{us}$  is the color of its  $s$ -th out-neighbor in sorted order. For a given list  $N_u^+$ , denote by  $R_{us}$  the number of colors to the left of position  $s$ , excluding the color of  $s$ , and by  $R'_{us}$  the number to the right, but not counting  $u$ 's own color

$$R_{us} = |\{\phi_{ut} \mid t < s, \phi_{ut} \neq \phi_{us}\}|, \quad (4)$$

$$R'_{us} = |\{\phi_{ut} \mid t > s, \phi_{ut} \neq \phi_u\}|. \quad (5)$$

With this in mind, consider the next result.

**Theorem 3.** *The companion I/O of PCF-1A is given by*

$$I_A(n) = \sum_{u=1}^n \sum_{s=1}^{X_u} R_{us} \quad (6)$$

and that of PCF-1B by

$$I_B(n) = \sum_{u=1}^n \left[ R'_{u1} + \sum_{s=1}^{X_u} R'_{us} \right]. \quad (7)$$

Note that (6)-(7) are exact. While  $R_{us}$  and  $R'_{us}$  appear symmetric to each other, there is a subtle difference. PCF-1A load-balances using in-degree, while PCF-1B using out-degree. Hence, their color assignments are not directly comparable to each other. However, on real graphs, PCF-1B commonly demands less I/O [8]. Additionally, it requires a lot fewer lookups. For the next result that shows this, define  $R_u = R_{u, X_u} + 1$  to be number of colors in  $N_u^+$ .

**Theorem 4.** *The number of hash-table hits in PCF-1A is*

$$\gamma_A(n) = I_A(n) + m - \sum_{u=1}^n R_u, \quad (8)$$

and that in PCF-1B is

$$\gamma_B(n) = m - n. \quad (9)$$

Note that  $\gamma_A(n)$  can be orders of magnitude larger than  $m$ , while  $\gamma_B(n)$  is always optimal (i.e., the same as in-memory  $E_1$ ). Further problems of PCF-1A include a requirement that RAM size be no smaller than the largest in-degree  $\max_u Y_u$ , which can be as large as  $n-1$ . In contrast, PCF-1B only needs  $M \geq \max_u X_u$ , whose largest value under descending-degree permutation  $\theta_D$  stays bounded by  $\sqrt{2m}$ . While PCF-1A can be dismissed for now as being inferior, we later come back to it and explain what features the new method shares with it.

### C. Bounds

Computing the exact I/O formula (7) requires processing the entire  $G_{\theta}^+$  and splitting all  $m$  edges into colors. In certain cases, this may be too expensive, especially if repeated many times (e.g., in an iterative search for optimal parameters). To overcome this issue, we derive simple upper bounds that require one pass over the out-degree sequence  $\{X_u\}$ .

**Theorem 5.** *For a given out-degree sequence  $\{X_u\}$ , the expected size of companion I/O in PCF-1B is bounded by*

$$E[I_B(n)] \leq \sum_{u=a_2}^n \zeta_u \left[ X_u - \zeta_u + 1 + (\zeta_u - 2)q_u^{X_u-1} \right], \quad (10)$$

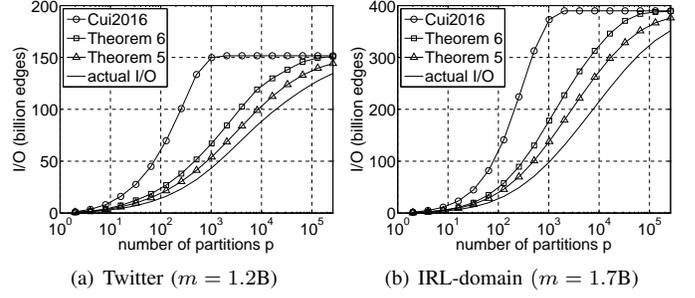


Fig. 2. Model accuracy in PCF-1B.

where  $q_u = 1 - 1/\zeta_u$  and  $\zeta_u = \phi_u - 1$ .

Bound (10) holds in expectation; however, there are adversarial graphs and color assignments that may violate it. Therefore, our second bound is deterministic, but somewhat looser in sparse graphs. It shows a more clear dependency of I/O on the second moment of out-degree.

**Theorem 6.** *The companion I/O of PCF-1B is bounded by*

$$I_B(n) \leq \sum_{u=1}^n \min\left(\frac{(X_u - 1)(X_u + 2)}{2}, X_u \zeta_u\right). \quad (11)$$

Note that [8] also obtains an upper-bound on  $I_B(n)$ ; however, they neglect the standalone term  $R'_{u1}$  in (7). This issue notwithstanding, their bound is a special case of (11) where  $\zeta_u = \phi_u - 1$  is replaced by  $p - 1$ . Fig. 2 shows a comparison between that result and our models, where we use Twitter from [17] and IRL-domain from the authors of [8].

### D. Discussion

PCF-1B requires that the longest out-list fit in memory, i.e.,  $M \geq \max_u X_u$ . While much better than in PCF-1A, this condition is stricter than in Pagh, which can work with constant  $M$  as  $n \rightarrow \infty$ . Additionally, PCF-1B needs enough disk space to write all companion files. In some cases, the read-only operation of Pagh may be preferable. Furthermore, it is common to exclude the preprocessing stage from comparison, because triangle enumeration can run multiple times over the same input (e.g., feeding the found  $\Delta_{uvw}$  to different consumers on the fly). However, if this is not the case, all I/O of PCF-1B needs to be doubled. This is of no consequence to asymptotics, but we benchmark both stages separately in the experimental section.

On the positive side, PCF achieves deterministic load-balancing and its sequential color assignment brings many benefits compared to random colors in Pagh. First, contiguous coloring produces stochastically smaller  $R'_{us}$  because  $u$ 's neighbors are more drawn towards colors with a large mass of degree. Since such colors are concentrated at the start of the range  $[1, n]$ , neighbor lists contain more duplicate colors than would be possible under uniform assignment. Second, due to sequential grouping of nodes into each color, splitting of neighbor lists in Algorithm 5 does not require a hash-table lookup for each edge. Similarly, when PCF-1B loads

the remote graph into RAM, it can use an array of offsets instead of a hash table to perform retrieval of remote edges.

## V. ASYMPTOTIC COMPARISON

We are now interested in the conditions that cause each of the candidate methods to be better than the other. Deciding this for finite  $n$  requires a specific graph and computation of the various models/bounds from the previous section. Instead, we study cases of  $n \rightarrow \infty$ , which should provide a qualitative assessment of each method's capabilities and types of graphs they are most suited for.

### A. Definitions

Suppose the average directed degree of the graph, i.e.,  $m/n$ , grows proportionally to  $n^a$ , where  $a \in [0, 1]$  is a constant. In general, we write  $f \sim g$  to mean that  $f(n) = O(g(n))$  and  $g(n) = O(f(n))$ . Similarly, assume memory size  $M_n \sim n^r$ , where  $r \in [0, 1+a]$  is also fixed. To ignore contribution from constants and slowly growing terms, we have the following definition.

**Definition 1.** The scaling rate of a function  $f(n)$  is given by

$$\omega(f) = \lim_{n \rightarrow \infty} \log_n f(n), \quad (12)$$

as long as the limit exists and is finite.

For example,  $f(n) = 5n^{2.3}/\log(n)$  has  $\omega(f) = 2.3$ . Since the scaling rate of  $m$  is  $1+a$ , Pagh has a very simple result

$$\omega(I_P) = \frac{3(1+a) - r}{2}. \quad (13)$$

However, the corresponding model for PCF is less obvious. We therefore perform a separate investigation into it.

### B. Dynamics of PCF

We start with an upper bound on  $\omega(I_B)$ , which requires studying the second moment of out-degree. To this end, define

$$\pi_n = \sum_{u=1}^n X_u^2 \quad (14)$$

and consider the next result.

**Theorem 7.** The scaling rate of (14) is  $\omega(\pi) = 1 + 2a + \epsilon$ , where  $\epsilon \in [0, (1-a)/2]$ .

Note that regular graphs (i.e., all degree equal to each other) yield  $\epsilon = 0$  for all  $a$ . Another well-known case follows from [35]. Specifically, for a sequence of graphs  $\{G_n\}$ , define  $D_n$  to be a random variable with the same distribution as undirected degree in  $G_n$ . Then, assuming  $E[D_n^{4/3}]$  converges to a finite constant as  $n \rightarrow \infty$ , these graphs also achieve  $\epsilon = 0$ . For more general cases, the family of dense-core graphs introduced next allows realization of any  $\epsilon$ .

**Theorem 8.** For any  $\epsilon \in [0, (1-a)/2]$ , there exists a graph with  $\omega(\pi) = 1 + 2a + \epsilon$ .

Leveraging the last two theorems finally produces a usable upper bound on the scaling rate of  $I_B(n)$ .

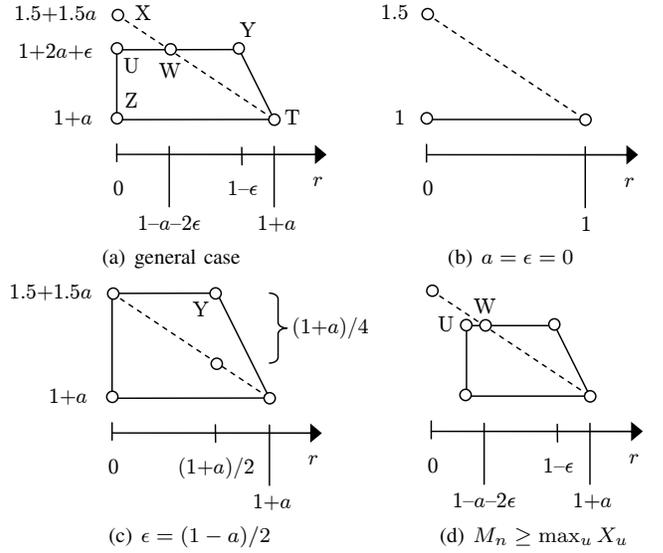


Fig. 3. Comparison of scaling rates.

**Theorem 9.** The rate of PCF-1B I/O is upper-bounded by

$$\omega(I_B) \leq \min(1 + 2a + \epsilon, 2 + 2a - r). \quad (15)$$

Furthermore, in the worst-case of  $\epsilon = (1-a)/2$ , the graphs built in Theorem 8 reach (15) for all  $a$  and  $r$ .

The graphs from Theorem 8 bring out the worst in PCF, to which we come back shortly. In the mean time, we show that it has a pretty impressive best-case as well.

**Theorem 10.** In bipartite graphs, PCF-1B has I/O overhead  $I_B(n) = m$  for all  $a$  and  $r$ , i.e.,  $\omega(I_B) = 1 + a$ .

Because PCF-1A load-balances partitions on the in-degree, rather than the out-degree, it fails to achieve the same benefits on bipartite graphs. Since PCF-1B cannot have less I/O than  $m$ , Theorem 10 shows that this bound is tight.

### C. Analysis

We summarize the findings of this section using Fig. 3(a). The  $x$ -axis shows rate  $r$  at which RAM increases as  $n \rightarrow \infty$ . This value ranges from zero (i.e., constant  $M_n$ ) to  $1+a$  (i.e., the entire graph fits in memory). On the  $y$ -axis, we have Pagh's scaling rate  $\omega(I_P)$ , represented by a dashed line, and the PCF-1B rate  $\omega(I_B)$ , given by the  $UYTZ$  trapezoid. Pagh's curve is a straight line that comes from (13). On the other hand, the rate of PCF-1B is contained somewhere in the trapezoid, with each interior point possibly corresponding to some graph  $G$ . The upper boundary, delineated by segments  $UY$  and  $YT$ , is produced by graphs from Theorem 8. The lower boundary, shown by line  $ZT$ , is the bipartite graph from Theorem 10.

At  $r = 0$ , i.e., constant RAM, Pagh begins in point  $X$  that is always no lower than PCF-1B's worst initial point  $U$ . As  $r$  increases, Pagh descends and eventually intersects with the upper bound of PCF-1B in point  $W$ . Therefore, in the range  $[0, 1-a-2\epsilon)$ , Pagh has no chance of beating PCF-1B,

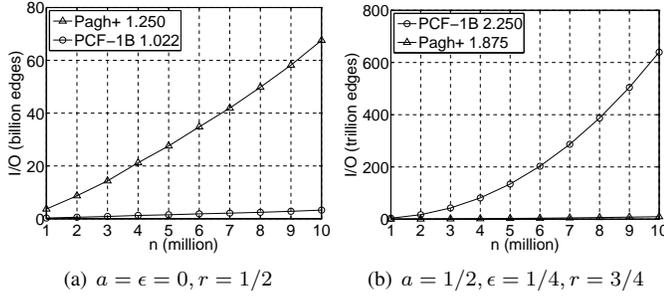


Fig. 4. Actual I/O with curve-fitted scaling rates.

regardless of the actual  $G$ . Between points  $W$  and  $T$ , some of the graphs are solved quicker by Pagh and others by PCF-1B.

It can be seen from the figure that the largest gap between the two methods occurs at  $r = 0$ , where PCF-1B in point  $Z$  vanquishes Pagh in point  $X$  by  $(1 + a)/2$ . Using a complete bipartite graph with  $a = 1$ , this yields a factor of  $n$  improvement in favor of PCF-1B. Outside of this custom-tailored graph, a more realistic best-case scenario for PCF-1B consists of graphs with a constant average degree and  $\epsilon = 0$ . This is depicted in Fig. 3(b), where PCF-1B collapses the trapezoid into a single line and defeats Pagh for all  $r$ . The biggest gap occurs at  $r = 0$ , where PCF-1B has a factor of  $\sqrt{n}$  less I/O.

On the other hand, the best case for Pagh is  $\epsilon = (1 - a)/2$ , which is shown in part (c) of the figure. In this situation, it beats the upper-bound of PCF-1B for all memory sizes. Consequently, knowing that  $G$  has a dense core similar to the graphs in Theorem 8, Pagh is the method of choice. The largest improvement is achieved in  $r = (1 + a)/2$ , where Pagh undercuts the scaling rate of PCF-1B by  $(1 + a)/4$ . Since  $a \leq 1$ , point  $Y$  causes the most damage to PCF in complete graphs, i.e., when  $a = 1$ . On these, Pagh has smaller cost by a factor of  $\sqrt{n}$ .

The final caveat is shown in Fig. 3(d), where the trapezoid has its left boundary moved forward to reflect the fact that  $M_n \geq \max_u X_u$  must hold for PCF-1B to work. While it is hard to predict how far point  $U$  shifts without access to the actual graph, we know it is no further than  $r = (1 + a)/2$  since  $\max_u X_u \leq \sqrt{2m}$ . This may be to the left of  $W$ , as show in the picture, or to the right. In either case, Pagh wins by default for all  $r$  where PCF-1B is unable to execute.

To see some of these cases in practice, Fig. 4(a) shows the actual I/O of the two methods in a random graph with Pareto degree, where shape  $\alpha = 1.5$  and average degree is 30. As predicted by our analysis and Fig 3(b), the asymptotic gap between the methods is  $n^{1/4}$ . Continuing to Fig. 4(b), we examine a dense-core graph from Theorem 8 whose average degree scales as  $\sqrt{n}$ , RAM size  $M_n = n^{3/4}$ , and  $\epsilon = 1/4$ . This puts the graph on the upper-bound of PCF, where the model suggests Pagh should win by  $n^{3/8}$ . Indeed, it does.

#### D. Discussion

We can now summarize the insight gained from dissecting both methods. Pagh's main pitfall is that it fails to exclude

nodes  $u$  that obviously cannot be in any triangles of relevant color. For example, if  $u$  has out-neighbors of color  $j$ , but none of color  $k$ , it should not be used in conjunction with remote edges  $E_{jk}^+$ . This leads to epic redundancy when the graph is sparse, i.e., there are few colors among the neighbors. On the other hand, this strategy works well for dense graphs where little pruning is necessary in the first place. The number of hash-table lookups proportional to  $c$  is also a concern.

On the other hand, the main downside of PCF lies in one-dimensional color partitioning. This creates a large number of colors  $p$  and causes unnecessary duplication of effort. Usage of 2D coloring could help reduce the number of files into which the out-neighbors must be written. This can be seen in (7), where making  $R'_{us}$  pick out of  $\sqrt{p}$  colors, rather than  $p$ , would be a noticeable improvement.

## VI. TRIGON

Our investigation discovered that an ideal algorithm should prune unnecessary edges, be able to utilize  $\sqrt{p}$  colors, deterministically load-balance partitions, leverage sequential colors for faster compression/intersection/lookups, handle star graphs without exorbitant overhead, operate with  $O(1)$  RAM, and post lower I/O numbers than either of the current techniques. We offer such an approach next.

### A. Generalized Coloring

All vertex/edge iterators [8] require the remote edge of enumerated triangles to be retrievable using random lookup in RAM. Therefore, for such methods to operate in external memory, the oriented graph must be split into at least  $p = m/M$  chunks. For now, we ignore the issue of *how* partitioning should be done and focus on the general concepts that would allow the in-memory search to function properly. The framework developed below applies to all 18 methods from [8]; however, to keep the notation to a minimum, we only describe how it works with  $E_1$ .

Since  $G_\theta^+$  is oriented and without self-loops, only the lower half of the adjacency matrix has non-zero entries. Therefore, any edge partition can be viewed as a subset of

$$B = \{(u, v) \in \mathbb{N}^2 \mid v < u \leq n\}, \quad (16)$$

which is a collection of all integer pairs  $(u, v)$  such that  $u > v$  and both numbers are no larger than  $n$ . Now suppose there exist sets  $B_1, \dots, B_p$  that form a partition on  $B$ , i.e.,  $B_\ell \subseteq B$  for all  $\ell$ ,  $B_i \cap B_j = \emptyset$  for  $i \neq j$ , and  $\cup_{\ell=1}^p B_\ell = B$ . This is illustrated using Fig. 5(a), where a  $5 \times 5$  adjacency matrix is split into three subgraphs. The number in each cell specifies the partition  $\ell$  it belongs to.

Note that all previous methods are special cases of this formalization. For example, Pagh uses  $B_{(j-1)c+k} = \{(u, v) \mid u \in V_j, v \in V_k\}$ , where  $c = \sqrt{p}$  is the number of colors. Both PCF methods utilize contiguous partitions shown in Fig. 5(b), where destinations are split into  $c_1$  colors and sources nodes into  $c_2 = p/c_1$ . PCF-1A uses  $c_1 = p$ , while PCF-1B does the opposite, i.e.,  $c_1 = 1$ .

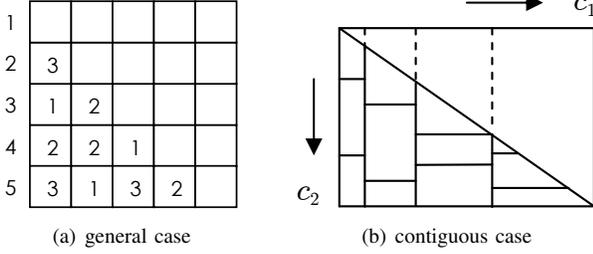


Fig. 5. Heterogeneous 2D partitioning of remote edges.

Once partitions are decided, the edges of  $G_\theta^+$  must be separated into sets  $E_1^+, \dots, E_p^+$ , where  $E_\ell^+ = E_\theta^+ \cap B_\ell$  for  $\ell = 1, 2, \dots, p$  and the following condition enforced.

**Definition 2.** A partition  $\{B_\ell\}$  is called admissible with respect to  $G_\theta^+$  if it guarantees that  $|E_\ell^+| = m/p$  for all  $\ell$ .

As discussed earlier, Pagh fails to produce admissible partitions on star graphs and similar structures. PCF-1A attempts to split the destinations into  $c_1 = p$  colors in Fig. 5(b) and runs into the same problem. On the other hand, PCF-1B is able to produce admissible partitions in all  $G$  as long as  $\max_u X_u \leq M$ .

### B. Trigon

We next decide how to achieve the best admissible partition within the general framework above. On one hand, it is theoretically possible to customize set  $\{B_\ell\}$  to a particular  $G_\theta^+$  in order to achieve the absolute minimum I/O for that graph. However, this solution is expensive (i.e., NP-hard) as it requires steam-rolling through all possible subsets of  $m$  edges. Instead, we are interested in alternative approaches that can be computationally reasonable.

To this end, recall our discussion of PCF and Pagh, where random assignment of nodes into colors would have produced stochastically larger  $R_{us}$  and  $R'_{us}$  in (4)-(5). The best technique, which comes from PCF, is to group nodes of the same color together. This forces members of  $N_u^+$  to pick color from a smaller range of options (i.e., those contained in  $[1, u-1]$ ). Additionally, continuous colors simplify preprocessing, remove redundancy between local lists of different hit nodes  $v$ , and improve intersection/compression performance. At the same time, Pagh's lowering of  $c$  to  $\sqrt{p}$  is appealing as well. Combining these ideas, it turns out that the design in Fig. 5(b) is the most sensible solution.

We call this approach *Trigon* and discuss its operation next. Since there are two colors involved (i.e., along the source and destination nodes), we call the one whose partitions are decided first *primary* and the other *secondary*. One option is to use  $c_1$  primary and  $c_2$  secondary colors, which is the case in Fig. 5(b). This approach starts by selecting vertical boundaries such that the number of edges contained in each primary color equals  $m/c_1$ . This is done by computing set  $\{a_k\}_{k=1}^{c_1}$  such that

$$\sum_{u=a_k}^{a_{k+1}-1} Y_u = \frac{m}{c_1}, \quad (17)$$

Algorithm 6: Trigon writing companion files

```

1 for u = 1 to n do
2   for k = 1 to c1 do ◁ run thru primary colors
3     Luk = Nu+ ∩ [ak, ak+1) ◁ local list for color k
4     if Luk ≠ ∅ then ◁ work to be done?
5       for j = 1 to c2 do ◁ run thru secondary colors
6         Hukj = Nu+ ∩ [bkj, bk,j+1) ◁ hit list for pair (kj)
7         if Hukj ≠ ∅ and |Luk ∩ Hukj| ≥ 2 then
8           Luk = Luk ∩ [1, max(Hukj)] ◁ prune
9           if φu(k) = j then ◁ local list in RAM
10            write (u, Hukj \ Luk) to companion Ckj+
11          else
12            write (u, Hukj ∪ Luk) to Ckj+

```

where  $Y_u$  is the in-degree of  $u$ . Note that this is exactly how PCF-1A begins and that the  $\{Y_u\}$  sequence is available during orientation of  $G$ , i.e., at no extra cost.

Then, for each primary color  $k$ , suppose boundaries  $\{b_{kj}\}_{j=1}^{c_2}$  specify the corresponding ranges of secondary colors. This is accomplished by load-balancing the out-degree within each partition  $(kj)$ , i.e.,

$$\sum_{u=b_{kj}}^{b_{k,j+1}-1} |N_u^+ \cap [a_k, a_{k+1})| = M, \quad (18)$$

which is similar to PCF-1B. For Fig. 5(b), this means each vertical column has size  $m/c_1$  and each rectangle fits in RAM. Note that if (17) fails to create enough partitions of primary color, e.g., on star-like graphs, the value of  $c_1$  is lowered to match the particulars of  $G_\theta^+$ . To compensate for the lack of vertical partitions, (18) automatically increases the number of secondary colors such that  $c_1 c_2 = p$  continues to hold.

The second option is to reverse this process, i.e., use source nodes for primary colors. However, it is not difficult to see that this procedure offers no I/O benefits due to symmetry, but at the same time has a major drawback in inability to adapt  $c_1$  to  $G_\theta^+$ . Therefore, the configuration in Fig. 5(b) is preferred.

The Trigon split technique is shown in Algorithm 6. The algorithm is pretty much self-explanatory, with the only caveat being Line 8. Under Trigon's coloring model, it is now possible for local list  $L_{uk}$  to contain nodes  $w$  larger than any hit node in  $H_{ukj}$ . They can never complete directed triangles in Fig. 1, which explains their removal.

### C. Analysis

Suppose  $\phi_u$  and  $\phi_{us}$  are defined as before, except they now refer to respectively the *primary* color of  $u$  and that of its  $s$ -th out-neighbor. In this notation, expression (4) still works for  $R_{us}$ . Similarly,  $R_u$  counts the number of primary colors in  $N_u^+$ . To handle the vertical dimension with  $c_2$  colors, let  $\varphi_u(k)$  be the *secondary* color of node  $u$  with respect to primary color  $k$  and assume  $\varphi_{us}(k)$  is the same for  $u$ 's out-neighbor  $s$ . Then, (5) is replaced with

$$R''_{us} = |\{\varphi_{ut}(\phi_s) \mid t > s, \varphi_{ut}(\phi_s) \neq \varphi_u(\phi_s)\}|, \quad (19)$$

which counts the number of secondary colors to the right of  $s$ , again excluding the color of  $u$ . Using the analysis of PCF-1B, the next result follows immediately.

**Theorem 11.** *The I/O complexity of Trigon is*

$$I_T(n) \approx \sum_{u=1}^n \left[ R''_{u1} + \sum_{s=1}^{X_u} (R_{us} + R''_{us}) \right] \quad (20)$$

and the number of hash-table lookups is

$$\gamma_T(n) = \sum_{u=1}^n \sum_{s=1}^{X_u} R_{us} + m - \sum_{u=1}^n R_u. \quad (21)$$

With the exception of minor terms related to overlapping local/hit lists, the result in (20) is exact. To perform a self-check, notice that PCF-1A (i.e.,  $c_1 = p$ ) has  $R''_{us} = 0$ , which converts (20) into (6). For PCF-1B (i.e.,  $c_1 = 1$ ), we get  $R_{us} = 0$  and  $R''_{us} = R'_{us}$ , which makes the Trigon model identical to (7). Recalling (8), also observe that (21) is exactly the number of lookups in PCF-1A under  $c_1$  partitions, where more primary colors cause more CPU cost.

Following the proof of Theorem 6, there exists a simple bound on  $I_T(n)$  that shows the impact of each color.

**Theorem 12.** *The Trigon I/O is upper-bounded by*

$$I_T(n) \leq \sum_{u=1}^n X_u h(X_u), \quad (22)$$

where  $h(x) = \min(x/2, c_1) + \min(x/2, c_2 - 1)$ .

The first term of  $h(x)$  bounds the size of hit lists (and the number of lookups), while the second models the size of local lists. Additionally, since  $h(x) \leq c_1 + c_2 - 1$ , usage of  $c_1 = c_2 = \sqrt{p}$  in (22) yields a looser bound

$$I_T(n) \leq \sum_{u=1}^n X_u (c_1 + c_2 - 1) = (2\sqrt{p} - 1)m, \quad (23)$$

which is the I/O cost of Pagh+ in Theorem 1. Since colors are sequential, Trigon beats (23) even in complete graphs, where it comes the closest to this bound, by roughly a factor of 2. It is also clear that (21) is upper bounded by  $c_1 m$ . Recalling Theorem 2, this makes  $\gamma_T(n)$  better than the corresponding metric in Pagh+ for all  $c_1 \leq \sqrt{p}$ .

Under an appropriately-chosen  $c_1$ , Trigon is no worse than either of the previous methods; however, the best choice for the number of primary colors remains far from obvious.

#### D. Minimizing I/O

One big question is whether deploying  $c_1 = \sqrt{p}$  is optimal for achieving the lowest I/O. This seems logical as it reduces the number of colors in each direction to their minimum. Because analysis of the accurate model (20) currently appears intractable, we only consider insight that might be gained from the upper-bound (22), which can be written as  $E[Xh(X)]$  for some random variable  $X$ .

Since  $c_1$  and  $c_2$  are almost interchangeable in  $h(x)$ , it makes sense to study the following simplified problem. Suppose we are interested in minimizing

$$\xi(c) = E[X(\min(X, c) + \min(X, p/c))], \quad (24)$$

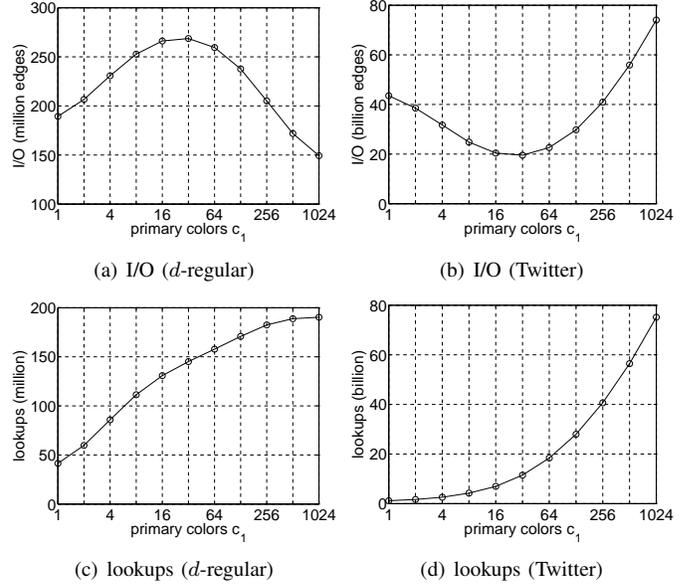


Fig. 6. Trigon tradeoffs between I/O and lookups ( $p = 1024$ ).

where  $X$  is a random variable that represents the out-degree of  $G_\theta^+$  and  $c$  is the number of primary colors.

**Theorem 13.** *If  $X$  has density  $f(x)$ , (24) is minimized by  $c = 1$ ,  $c = p$ , or any solution to  $g(c) = g(p/c)$ , where*

$$g(y) = y \int_y^\infty x f(x) dx. \quad (25)$$

Notice that  $c = \sqrt{p}$  is a trivial solution to  $g(c) = g(p/c)$ . Furthermore, it is the *only* solution if  $g(x)$  is monotonic. Outside of certain esoteric cases, this result shows that the optimal Trigon configuration is PCF-1A, PCF-1B, or  $c_1 = \sqrt{p}$ . However, there is no clear winner for all graphs  $G$ . The next example shows one such case.

**Theorem 14.** *If  $X < 2\sqrt{p} - 1$  with probability 1, then  $c = 1$  or  $c = p$  is optimal in (24). On the other hand, if  $X > 2\sqrt{p} - 1$  with probability 1, then  $c = \sqrt{p}$  is optimal.*

One example that falls under Theorem 14 are  $d$ -regular graphs. This is illustrated in Fig. 6(a) using a random graph with  $d = 10$ ,  $n = 10M$ , and  $p = 1024$ . The I/O function of Trigon in this graph is an inverted cup, with the middle being the worst and the two boundaries being the best. This is one of the few cases where PCF-1A wins over PCF-1B. A more common scenario is given by Twitter in Fig. 6(b), where  $c_1 = \sqrt{p} = 32$  is clearly optimal and PCF-1B beats PCF-1A.

If the program has access to  $G_\theta^+$ , it can compute our models shown earlier in the paper and always make the right decision. However, if graph  $G_\theta^+$  cannot be examined before choosing  $c_1$ , the next result explains which choice would always be safer.

**Theorem 15.** *Usage of  $c = \sqrt{p}$  in (24) yields at most double the optimal I/O. On the other hand,  $c = 1$  or  $c = p$  can be worse than optimal by a factor of  $\sqrt{p}$ .*

TABLE I  
GRAPH PROPERTIES

Graph	Nodes $n$	Edges $m$	Size (GB)	Triangles
Twitter [17]	41M	1.2B	9.3	35B
Yahoo [36]	720M	6.4B	53.3	86B
IRL-domain [8]	86M	1.7B	13.3	113B
IRL-host [8]	642M	6.4B	52.7	437B
IRL-IP [8]	1.6M	818M	6.1	1040B
ClueWeb [8]	8.2B	51B	358	879B
Complete	100K	5.0B	37.2	167T
Bipartite	100K	2.5B	18.6	0

TABLE II  
I/O (BILLION EDGES)

Graph	$p$	Pagh+	PCF-1B	Trigon	RAM
Twitter	1,024	75.6	43.5	19.5	4.5 MB
Yahoo		392.3	25.5	25.5	23.2 MB
IRL-domain		104.8	98.4	33.8	6.2 MB
IRL-host		386.5	137.9	59.7	22.9 MB
IRL-IP		51.5	145.7	23.4	3.0 MB
ClueWeb		2,869.9	457.1	326.2	169.7 MB
Complete	10,000	995.0	15,742	493	1.9 MB
Bipartite		497.0	2.5	2.5	1.0 MB

### E. Minimizing Runtime

When achieving the quickest execution time is a priority, the choice of optimal  $c_1$  may involve balancing conflicting objectives. This is exemplified by Fig. 6(c)-(d), where optimal points  $c_1$  do not coincide with those in plots (a)-(b). Note that the  $x$ -axis is on a  $\log_2$  scale and lookup growth is sublinear. On the  $d$ -regular graph, Trigon increases  $\gamma_T(n)$  by 3.5 times between  $c_1 = 1$  and  $\sqrt{p}$ . On Twitter, the number of lookups goes up by a factor of 9.8. As predicted earlier, both values are much smaller than Pagh’s linear (i.e., 32-fold) increase.

With overlapped operation between CPU and I/O, the runtime is determined by the maximum of disk read time and in-memory operations. Define  $S_D$ ,  $S_I$ , and  $S_H$  to be respectively the speed of the disk, intersection, and lookups (in edges/sec), which can be easily benchmarked on startup. Parameterizing  $I_T(n)$  and  $\gamma_T(n)$  with  $c_1$ , an objective might be to minimize

$$r(c_1) = \max\left(\frac{I_T(n, c_1)}{S_D}, \frac{\rho(n)}{S_I} + \frac{\gamma_T(n, c_1)}{S_H}\right) \quad (26)$$

where  $\rho(n)$  is the intersection cost from (1).

To obtain  $I_T(n, c_1)$  and  $\gamma_T(n, c_1)$ , one can use (20)-(21). Direct computation of these values may be costly; however, approximation (22), as well as its refinement using (10) or (11), work quite well.

## VII. EVALUATION

We finally come to the stage of putting the ideas developed in the previous section to work. To enable a fair comparison, we use C++ to implement Trigon and Pagh+ as separate modules that share the same in-memory and disk components (i.e., multi-threading, overlapped I/O, SIMD intersection). Setting  $c_1 = 1$  in Trigon, we obtain PCF-1B. Therefore, the only difference between the three methods lies in their partitioning scheme. As PCF-1A is not competitive on most real-world graphs, we do not consider it here.

Out of the standard graphs used for triangle listing, we engage the six largest from [8]. Their characteristics are shown in Table I. In the last two rows, we add into the mixture best-case scenarios from Pagh and PCF.

### A. I/O

Performance of triangle listing depends on the ratio between graph size and available RAM, i.e.,  $p = m/M$ . Since our I/O methods are quite efficient, this affords us an opportunity to examine scenarios where graphs are substantially larger

than memory. In fact, this is the first paper that runs an actual implementation with RAM size that is 3 – 4 orders of magnitude smaller than the oriented graph  $G_\theta^+$ .

On real-world graphs, Table II shows that Pagh+ loses to PCF-1B in five out of the six cases, sometimes by as much as a factor of 15. The only graph where it wins is IRL-IP, which is quite dense (average degree 1,030). This is not surprising given our earlier analysis. If we consider preprocessing to be part of triangle listing and double the PCF-1B result, it becomes worse that Pagh+ in three cases. On the other hand, Trigon beats both previous methods on each of the graphs. Furthermore, even if its I/O is doubled, it still stays below Pagh+, in some cases by a wide margin.

On the complete graph and 10K partitions, Pagh+ has 15 times less I/O than PCF-1B. However, its overhead is still double that of Trigon, which follows from the dichotomy of sequential vs random coloring discussed earlier. On the bipartite graph, PCF-1B and Trigon both annihilate Pagh+ by issuing 200 times less I/O, which also agrees with our analysis.

### B. Runtime

For the experiments, we use one machine with a six-core Intel i7-3930K (desktop CPU released in 2011). We equip this computer with a single 3-TB magnetic hard drive (Hitachi Deskstar 7K3000) that is capable of reads at 160 MB/s. We omit PCF-1B since slow I/O makes it predictably worse than Trigon. Instead, we compare against Pagh+ to investigate the impact of non-sequential colors, lookup load, and disk seeking. Furthermore, we consider the total delay, which includes the partitioning phase, as one of the measures of performance.

Table III shows the result. In the first four rows, Trigon completes triangle search 15 – 60 times quicker than Pagh+. One notable example is Yahoo, where purely sequential I/O in Pagh+ would have been responsible for only 163 minutes (i.e., 392B edges, four bytes each, read at 160 MB/s). Instead, Pagh+ spends an additional 1,132 minutes (i.e., 18 hours) on lookups. A similar scenario occurs with ClueWeb in row six, where Pagh+ gets bogged down for 5 days just checking the hash table. Table IV confirms that Pagh+ requires substantially more random memory access than Trigon. The larger the hash-table size, the worse the lookup speed, which explains the huge runtime gap between the two methods on ClueWeb.

In dense-graph scenarios of Table III, Pagh+ is 3 – 5 times slower than Trigon. Usage of 10K partitions for the complete graph creates a noticeable bottleneck in reading  $c^3 = 1M$

TABLE III  
PREPROCESSING AND ENUMERATION TIME (MINUTES)

Graph	Pagh+			Trigon		
	pre	run	total	pre	run	total
Twitter	3.3	144.0	147.4	14.8	10.0	24.8
Yahoo	27.8	1,296.4	1,324.2	35.5	19.1	54.6
IRL-domain	3.5	191.4	194.9	21.0	14.8	35.8
IRL-host	26.2	1,070.3	1,096.5	52.7	32.0	84.7
IRL-IP	0.2	31.7	31.9	12.1	8.7	20.8
ClueWeb	181.8	8,331.1	8,512.9	426.8	254.3	681.1
Complete	2.5	1,050.7	1,053.2	624.2	238.6	862.8
Bipartite	8.8	629.5	638.3	6.6	2.3	9.9

TABLE IV  
NUMBER OF LOOKUPS (BILLION)

Graph	Pagh+	Trigon	Ratio
Twitter	38.4	11.5	3.5
Yahoo	199.2	19.9	10.0
IRL-domain	53.2	19.4	2.7
IRL-host	196.3	34.3	5.8
IRL-IP	26.2	13.2	2.0
ClueWeb	1,457.8	205.3	7.1
Complete	500.0	252.0	2.0
Bipartite	250.0	2.5	100.0

combinations of files. Analysis of the total delay, i.e., both preprocessing and triangle listing, shows a more favorable outcome for Pagh+; however, Trigon is still faster in all graphs, sometimes by a wide margin (e.g., 24× on Yahoo).

## VIII. CONCLUSION

We analyzed I/O complexity of the best methods in the literature, compared their asymptotics, identified their inherent strengths and weaknesses, and developed a novel framework that surpassed the existing efforts in all performance measures relevant to triangle listing.

## REFERENCES

- [1] S. Arifuzzaman, M. Khan, and M. Marathe, "PATRIC: A Parallel Algorithm for Counting Triangles in Massive Networks," in *Proc. ACM CIKM*, Oct. 2013, pp. 529–538.
- [2] Z. Bar-Yossef, R. Kumar, and D. Sivakumar, "Reductions in Streaming Algorithms, with an Application to Counting Triangles in Graphs," in *Proc. ACM-SIAM SODA*, Jan. 2002, pp. 623–632.
- [3] V. Batagelj and M. Zaveršnik, "Short Cycle Connectivity," *Elsevier Discrete Mathematics*, vol. 307, no. 3-5, pp. 310–318, Feb. 2007.
- [4] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis, "Efficient Semi-streaming Algorithms for Local Triangle Counting in Massive Graphs," in *Proc. ACM SIGKDD*, Aug. 2008, pp. 16–24.
- [5] N. Chiba and T. Nishizeki, "Arboricity and Subgraph Listing Algorithms," *SIAM J. Comput.*, vol. 14, no. 1, pp. 210–223, Feb. 1985.
- [6] S. Chu and J. Cheng, "Triangle Listing in Massive Networks and Its Applications," in *Proc. ACM SIGKDD*, Aug. 2011, pp. 672–680.
- [7] J. Cohen, "Graph Twiddling in a MapReduce World," *Computing in Science & Engineering*, vol. 11, no. 4, pp. 29–41, Jul.-Aug. 2009.
- [8] Y. Cui, D. Xiao, and D. Loguinov, "On Efficient External-Memory Triangle Listing," in *Proc. IEEE ICDM*, Dec. 2016, pp. 101–110.
- [9] Y. Cui, D. Xiao, D. B. Cline, and D. Loguinov, "Improving I/O Complexity of Triangle Enumeration," Texas A&M University, Tech. Rep. 2017-8-3, Aug. 2017. [Online]. Available: <http://irl.cse.tamu.edu/publications>.
- [10] I. Fudos and C. M. Hoffmann, "A Graph-Constructive Approach to Solving Systems of Geometric Constraints," *ACM Transactions on Graphics*, vol. 16, no. 2, pp. 179–216, Apr. 1997.
- [11] I. Giechaskiel, G. Panagopoulos, and E. Yoneki, "PDTL: Parallel and Distributed Triangle Listing for Massive Graphs," in *Proc. IEEE ICPP*, Sep. 2015, pp. 370–379.
- [12] P. Gupta, V. Satuluri, A. Grewal, S. Gurumurthy, V. Zhabiyuk, Q. Li, and J. Lin, "Real-Time Twitter Recommendation: Online Motif Detection in Large Dynamic Graphs," *PVLDB*, vol. 7, no. 13, pp. 1379–1380, Aug. 2014.
- [13] T. Hocevar and J. Demsar, "A Combinatorial Approach to Graphlet Counting," *Bioinformatics*, vol. 30, no. 4, pp. 559–565, Feb. 2014.
- [14] X. Hu, Y. Tao, and C. Chung, "Massive Graph Triangulation," in *Proc. ACM SIGMOD*, Jun. 2013, pp. 325–336.
- [15] Z. R. Kashani, H. Ahrabian, E. Elahi, A. Nowzari-Dalini, E. S. Ansari, S. Asadi, S. Mohammadi, F. Schreiber, and A. Masoudi-Nejad, "Kavosh: A New Algorithm for Finding Network Motifs," *Bioinformatics*, vol. 10, no. 318, Oct. 2009.
- [16] J. Kim, W. Han, S. Lee, K. Park, and H. Yu, "OPT: A New Framework for Overlapped and Parallel Triangulation in Large-Scale Graphs," in *Proc. ACM SIGMOD*, Jun. 2014, pp. 637–648.
- [17] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, A Social Network or a News Media?" in *Proc. WWW*, Apr. 2010, pp. 591–600.
- [18] M. Latapy, "Main-memory Triangle Computations for Very Large (Sparse (Power-law)) Graphs," *Elsevier Theor. Comput. Sci.*, vol. 407, no. 1-3, pp. 458–473, Nov. 2008.
- [19] D. W. Matula and L. L. Beck, "Smallest-Last Ordering and Clustering and Graph Coloring Algorithms," *Journal of the ACM*, vol. 30, no. 3, pp. 417–427, Jul. 1983.
- [20] L. A. A. Meira, V. R. Maximo, A. L. Fazenda, and A. F. D. Conceicao, "Acc-Motif: Accelerated Network Motif Detection," *IEEE/ACM Trans. Computational Biology and Bioinformatics*, vol. 11, no. 5, pp. 853–862, Apr. 2014.
- [21] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon, "Network Motifs: Simple Building Blocks of Complex Networks," *Science*, vol. 298, no. 5594, pp. 824–827, Oct. 2002.
- [22] M. E. Newman, D. J. Watts, and S. H. Strogatz, "Random Graph Models of Social Networks," *Proceedings of the National Academy of Sciences*, vol. 99, no. Suppl. 1, pp. 2566–2572, Feb. 2002.
- [23] R. Pagh and F. Silvestri, "The Input/Output Complexity of Triangle Enumeration," in *Proc. ACM PODS*, Jun. 2014, pp. 224–233.
- [24] H.-M. Park, S.-H. Myaeng, and U. Kang, "PTE: Enumerating Trillion Triangles On Distributed Systems," in *Proc. ACM SIGKDD*, Aug. 2016, pp. 1115–1124.
- [25] T. Schank and D. Wagner, "Finding, Counting and Listing All Triangles in Large Graphs, an Experimental Study," in *Proc. WEA*, May 2005, pp. 606–609.
- [26] M. Sevenich, S. Hong, A. Welc, and H. Chafi, "Fast In-Memory Triangle Listing for Large Real-World Graphs," in *Proc. ACM SNA-KDD*, Aug. 2014, pp. 1–9.
- [27] J. Shun and K. Tangwongsan, "Multicore Triangle Computations without Tuning," in *Proc. IEEE ICDE*, Apr. 2015, pp. 149–160.
- [28] S. Suri and S. Vassilvitskii, "Counting Triangles and the Curse of the Last Reducer," in *Proc. WWW*, Mar. 2011, pp. 607–614.
- [29] N. H. Tran, K. P. Choi, and L. Zhang, "Counting Motifs in the Human Interactome," *Nature Communications*, vol. 4, p. 2241, Aug. 2013.
- [30] N. Wang, J. Zhang, K.-L. Tan, and A. K. Tung, "On Triangulation-Based Dense Neighborhood Graph Discovery," *PVLDB*, vol. 4, no. 2, pp. 58–68, Nov. 2010.
- [31] D. J. Watts and S. Strogatz, "Collective Dynamics of 'Small World' Networks," *Nature*, vol. 393, pp. 440–442, Jun. 1998.
- [32] D. J. Welsh and M. B. Powell, "An upper bound for the chromatic number of a graph and its application to timetabling problems," *Comput. J.*, vol. 10, no. 1, pp. 85–86, Jan. 1967.
- [33] S. Wernicke and F. Rasche, "FANMOD: A Tool for Fast Network Motif Detection," *Bioinformatics*, vol. 22, no. 9, pp. 1152–1153, Feb. 2006.
- [34] V. Williams and R. Williams, "Subcubic Equivalences Between Path, Matrix, and Triangle Problems," in *Proc. IEEE FOCS*, Oct. 2010, pp. 645–654.
- [35] D. Xiao, Y. Cui, D. Cline, and D. Loguinov, "On Asymptotic Cost of Triangle Listing in Random Graphs," in *Proc. ACM PODS*, May 2017, pp. 261–272.
- [36] Yahoo Altavista Graph, 2002. [Online]. Available: <http://webscope.sandbox.yahoo.com/catalog.php?datatype=g>.
- [37] Z. Yang, C. Wilson, X. Wang, T. Gao, B. Zhao, and Y. Dai, "Uncovering Social Network Sybils in the Wild," in *Proc. ACM IMC*, Nov. 2011, pp. 259–268.