# EFFICIENT EXTERNAL-MEMORY ALGORITHMS FOR GRAPH MINING

A Dissertation

by

YI CUI

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

| | |
|---|---|
| Chair of Committee, | Dmitri Loguinov |
| Committee Members, | Riccardo Bettati |
| | James Caverlee |
| | A. L. Narasimha Reddy |
| Head of Department, | Dilma Da Silva |

December 2017

Major Subject: Computer Science

ABSTRACT

The explosion of big data in areas like the web and social networks has posed big challenges to research activities, including data mining, information retrieval, security etc. This dissertation focuses on a particular area, graph mining, and specifically proposes several novel algorithms to solve the problems of triangle listing and computation of neighborhood function in large-scale graphs.

We first study the classic problem of triangle listing. We generalize the existing in-memory algorithms into a single framework of 18 triangle-search techniques. We then develop a novel external-memory approach, which we call Pruned Companion Files (PCF), that supports disk operation of all 18 algorithms. When compared to state-of-the-art available implementations MGT and PDTL, PCF runs 5-10 times faster and exhibits orders of magnitude less I/O.

We next focus on I/O complexity of triangle listing. Recent work by Pagh etc. provides an appealing theoretical I/O complexity for triangle listing via graph partitioning by random coloring of nodes. Since no implementation of Pagh is available and little is known about the comparison between Pagh and PCF, we carefully implement Pagh, 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 techniques in all graphs and under all RAM conditions.

We finally turn our attention to neighborhood function. Exact computation of neighborhood function is expensive in terms of CPU and I/O cost. Previous work mostly focuses on approximations. We show that our novel techniques developed for triangle listing can also be applied to this problem. We next study an application of neighborhood function

to ranking of Internet hosts. Our method computes neighborhood functions for each host as an indication of its reputation. The evaluation shows that our method is robust to ranking manipulation and brings less spam to its top ranking list compared to PageRank and TrustRank.

# DEDICATION

To my family

# ACKNOWLEDGMENTS

First of all, I would like to express my gratitude to my advisor, Dr. Dmitri Loguinov, without whom this dissertation would not have been possible. I learned from him not only the knowledge of the field, but also the attitude towards research. His insight into my research topics and continuous guidance have been of great importance to me. Under his supervision, I developed my skills in programming, writing research papers, and giving good presentations. I believe that these valuable skills would help me success in my future career.

I would also like to thank the other committee members, Dr. Riccardo Bettati, Dr. James Caverlee, and Dr. A. L. Narasimha Reddy, for their time and effort in serving my committee. I also appreciate Dr. Radu Stoleru's attendance to my dissertation defense due to the absence of Dr Riccardo Bettati. Their comments and suggestions provide meaningful insight into my work.

I also want to thank all the members of the Internet Research Lab whom I spent a great time with – Tanzir Ahmed, Xiaoyong Li, Siddhartha Mathiharan, Zain Shamsi, Sadhan Sood, Patrick Webster, Xiangzhou Xia, Di Xiao, Xiaoxi Zhang, Yue Zhuo. I'm always grateful for their companion during my studies.

Finally, I would like to express my deepest thanks to my parents for their endless love and support.

CONTRIBUTORS AND FUNDING SOURCES

## Contributors

This work was supported by a dissertation committee consisting of Professor Dmitri Loguinov, Riccardo Bettati, and James Caverlee of the Department of Computer Science and Engineering and Professor A. L. Narasimha Reddy of the Department of Electrical and Computer Engineering.

Di Xiao helped with modeling and simulation work in Triangle listing.

All other work conducted for the dissertation was completed by the student independently.

## Funding Sources

Graduate study was supported in part by NSF Grant CNS-1319984.

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

# 1. INTRODUCTION

## 1.1 Overview

With the explosion of big data in areas like the web and social networks, research activities including data mining, information retrieval, security etc. are now facing big challenges. This dissertation studies one particular research area – graph mining. In many areas, the data is represented as graphs that model the relationships between entities, e.g., linking structures between web pages, friend relationships between social network users, connections between Internet routers etc. Data mining in these graphs provides important insights into how they are structured and how they evolve, as well as other useful information. However, this task is facing increasing challenge as the graphs scale in orders of magnitude fast speed. For example, Yahoo disclosed crawling $150$B web pages and using graphs with $5$T edges [58] in late $2010$; Google was crawling $20$B pages/day and keeping track of $30$T unique URLs [72] in $2012$. Facebook keeps expanding its social network that now involves $1.94$B monthly active users [25].

One of the major challenges of data mining in such large-scale graphs is the tremendous computation cost, e.g., CPU cycles and disk I/O, needed to manage and process the graphs. Instead of devoting more computation resources (e.g., machines and CPU cores), our goals to deal with this challenge is to deeper our understanding of the underlying algorithms, build accurate models of their cost, and propose novel techniques that optimize the cost. With these goals in mind, this dissertation revisits two classic problems in graph theory that play a key role in graph mining – triangle listing and computation of neighborhood function. A set of novel algorithms in graph orientation, relabeling, and external-memory partitioning are developed that can be generally applied to solving these graph mining problems.

The next three chapters in this dissertation present algorithms for efficient triangle listing and computation of neighborhood function in large-scale graphs, and demonstrate an application of depth-2 neighborhood function in ranking of Internet host graphs. The rest of this chapter briefly introduces each of the topics in turn.

## 1.2 Pruned Companion Files (PCF)

Triangle listing is a classic problem that has been studied for over 35 years [41]. Its importance has long been recognized in various areas. In network analysis, many important measurements, including clustering coefficient [78], transitivity [55], and triangular connectivity [6], rely on the computation of triangles. In social networks, triangles help detect fake accounts [88], identify web spam [7], locate communities [11], and measure quality of content [7]. In bioinformatics, triangles are an important type of network motifs [53] that provide meaningful information in protein-protein, gene-regulatory, and metabolic networks [43]. Applications also emerged in other areas such as k-truss [20] and dense subgraph mining [77] in graph theory; query planning optimization in databases [5].

Despite a significant amount of effort [1], [3], [15], [17], [20], [29], [38], [41], [44], [48], [62], [63], [67], [69], [74] devoted into studying this problem, our understanding is still limited – little is know about the accurate CPU cost of triangle listing, its behavior under different acyclic orientations, and comparison across the different methods. Moreover, external-memory solutions remain largely unexplored. Most recent work [3], [17], [29], [38], [44] requires an I/O complexity that is quadratic to the graph size, which is difficult to scale when the graph size becomes large.

Our work [84] creates a unified framework that covers 6 possible triangle search orders and 18 vertex/edge iterator algorithms to actually perform the search. All previous work falls into the coverage of our framework. Furthermore, 4 CPU-cost equivalent classes are identified among the 18 options, each with its own optimal relabeling. We then propose

a single external-memory framework called Pruned Companion Files (PCF) that supports disk operation of all $18$ algorithms. PCF achieves deterministic and balanced graph partitions. Each partition is associated with a companion file that helps to discover triangles within the partition. By carefully pruning edges in the companion file and keep only the ones that potentially form a triangle, PCF is able to significantly reduce the I/O cost. In order to gain more insight into PCF's I/O performance, we next model its I/O cost and derive an upper bound of it. We show that in certain cases, PCF's I/O cost can be linear to graph size, which is the first report of linear I/O in triangle listing. Applying PCF to the $18$ algorithms, we find $3$ I/O equivalent classes. Combined with the $4$ CPU-cost equivalent classes, we eventually identify the optimal solution that minimizes both CPU and I/O cost.

We finish this chapter by providing a high-performance multi-threading C++ implementation of the optimal solution and evaluating its performance in various datasets. Besides several standard graphs that are normally used in previous work, we introduce new graphs from two web crawls: IRLbot [49] and ClueWeb [19] in order to test our algorithms in handling real large-scale graphs that contain billions of nodes and trillions of triangles. Armed with our novel techniques in graph relabeling, orientation, and external-memory partitioning, plus the introduction of SIMD (single instruction, multiple data) instructions into graph processing, our implementation runs $5 - 10$ times faster and exhibits order of magnitude less I/O compared to state-of-the-art competitors. To highlight some of the results, our implementation finds $1$T triangles in $237$ seconds using a desktop CPU, which translates to discovering over $4$B triangles per sec. On the largest graph in our evaluation with $8.2$B nodes and $102$B edges, we are able to finish triangles listing in only $1{,}747$ seconds.

## 1.3 Trigon

This chapter will focus on I/O cost of triangle listing as it is often the main bottleneck when dealing with large-scale graphs, especially with a small amount of memory or slow hard drives. Our method PCF is equipped with a highly optimized in-memory triangle-listing solution that includes optimal triangle search order and graph relabeling. However, its I/O performance can be suboptimal in certain cases. Although we have picked the optimal choice under PCF's graph partitioning scheme, it is not clear if there exists a novel graph partitioning scheme that can beat PCF's best configuration in I/O. Recent work by Pagh etc. [60] proposes to partition a graph with random coloring of nodes and proves an appealing theoretical I/O cost of triangle listing by using such graph partitioning. This motivates us to conduct a comprehensive I/O comparison between PCF and Pagh. Our goal is to understand the I/O behavior of both algorithms, under what conditions can one be better than the other, and whether it is possible to develop a new graph partitioning scheme that outperforms both of them in all cases.

The original Pagh algorithm only provides its I/O analysis at a high level with certain details omitted and no implementation available. Thus, this chapter starts by discussing our refinements to Pagh that make it practical. We next perform a comprehensive analysis of both Pagh and PCF by deriving accurate I/O models for them and pointing out their limitations, e.g., under what conditions one loses performance and what are the memory restrictions for them. While it is difficult to obtain a closed-form formula of PCF's I/O, we derive a series of strict upper bounds of its I/O and use them in comparison to the I/O of Pagh. Our asymptotic comparison of the two methods shows that each of them can beat the other in certain cases. PCF has the highest advantage when the graph is sparse, the variance of out-degree is small, and RAM is growing slowly with the graph size. Pagh wins when the conditions are reversed. When the number of nodes $n \to \infty$, PCF can beat

Pagh by a factor of $n$ in its best case and lose to Pagh by a factor of $\sqrt{n}$ in the worse case.

With the lessons learned from the two algorithms, we are now ready to deliver a novel method called Trigon that inherits advantages from both previous ones, overcomes their shortcomings, and consistently beats them in all cases. Assume graphs are stored as adjacency lists, each of them consists of a source node and a neighbor list. Two variations of PCF either partition by source nodes or destination nodes, which we call one dimensional (1D) partitioning. Trigon extends this to 2D partitioning that splits both source and destination nodes. Instead of random coloring in Pagh, Trigon adopts sequential coloring that reduces I/O and possesses several advantages in CPU processing, e.g., faster list compression, intersection, and hash table lookups. Besides these, we show that Trigon eliminates certain memory constraints from PCF and handles certain graphs that cause problems for Pagh.

We finish this chapter by evaluating the performance of Trigon. In order to test the ability of each algorithm to handle large-scale graphs, we set the available RAM size to be $3 - 4$ orders of magnitude smaller than the graph size. Since most real-world graphs in areas like the web and social networks are sparse, we find PCF outperforms Pagh by a factor up to $15$ in five out of the six cases. The only exception is a dense graph with average degree $1{,}030$. In the densest graphs, i.e., complete graphs, PCF loses to Pagh by a factor of $16$. This observation confirms our conclusion about the two methods. Our new method Trigon consistently beats the other two methods in all cases. In the best cases, it beats Pagh by a factor of $199$ and outperforms PCF by a factor of $32$.

## 1.4 Neighborhood Function

The next problem we study in this dissertation is neighborhood function. For each node, its neighborhood function computes the nodes reachable within $d$ steps in a graph. Normally, only a count of the number of neighborhood nodes is needed. In certain appli-

cations, more complex functions may be required, e.g., each node has some weigh and the goal is to collect the weights among the neighborhood. This function is important in data mining as to compute the effective diameter of a graph, measure network robustness, and check graph similarity [13], [61]. It is also used in ranking web graphs and detecting spam [8], [9], [73].

The exact computation of neighborhood function is costly, especially when the graph does not fit into RAM. Previous work [8], [9], [13], [61], [73] focuses on approximation of neighborhood function, which suffers from estimation errors and is counting only, e.g., cannot perform more complex functions. Motivated by recent work [73] that applies neighborhood function with $d = 2$ to ranking web graphs, we take an investigation into efficient computation of exact neighborhood function at depth $2$. We show that this is a similar but more difficult problem compared to triangle listing. Our developed techniques in graph relabeling and external-memory partitioning can be generally applied to solving this problem as well.

In the end, we demonstrate an application of neighborhood function at depth $2$ to ranking of Internet hosts. Ranking hosts is difficult since they can be infinitely generated for free by using automated scripts and DNS wildcard entries. Spammers can potentially generate any number of spam hosts and form any linking structures to manipulate ranking algorithms. To overcome this problem, our approach is to leverage finite Internet resources in ranking, e.g., domains, IP addresses, DNS nameservers. Controlling such resources in a large scale is difficult due to the financial cost involves in creating them. Additionally, we find that spam is prevalent inside web-hosting services, where people can purchase domain names and setup their websites. Spammers heavily use such services to host their spam content. Since websites within the same hosting services share the same network infrastructure, e.g., IP subnets and DNS nameservers, we propose the use of co-hosted domain density to combat spam.

We test our ranking algorithms in host graphs from two large-scale web crawls – IRLbot [49] and ClueWeb [19] and compare with classic methods including PageRank [59] and TrustRank [34]. Our first method ranks hosts by the number of domain supporters at depth $2$, which we call L2-D. Manual analysis shows that our method brings no spam hosts in its top-$10$ list; while both PageRank and TrustRank admits spam in the same range. Looking at a larger range, L2-D only has $0.2\%$ and $0.1\%$ spam in its top-1K in IRLbot and ClueWeb, respectively. This is a factor of $70$ and $20$ less spam compared to PageRank and TrustRank. L2-D keeps its advantage all the way to top-$100$K list. Another important metric we use in evaluation is Google Toolbar Rank (GTR) [66], which is a value from $0$ to $10$ that indicates Google's option about the quality of websites. L2-D is again the clear winner of this metric. Within the top-$100$K list, L2-D delivers the highest average GTR value and the least unwanted hosts, i.e., hosts with a GTR value $\leq 3$.

To further improve the ranking, we next study DNS co-hosting behaviors, which refers to multiple hosts and domains sharing the same IP addresses or DNS nameservers. Our analysis shows that a large fraction of the spam hosts identified during our manual inspection are inside web-hosting services. We first eliminate the inflation of ranking scores from the same co-hosted structure by computing the number of supporting DNS nameservers instead of domains. Then, we propose a method that utilizes domain density of each /$24$ IP subnet to accurately identify web-hosting services. This allows us to directly punish the hosts that come from web-hosting services. With all the effort, we are now able to create a spam-free top-1K list and only $0.2\%$ spam from 1K to 10K.

The rest of this dissertation is organized as the following. In Chapter 2 we present PCF for efficient external-memory solution of triangle listing. Chapter 3 discusses Trigon that further improves the I/O performance of PCF. Chapter 4 will address another problem – neighborhood function and illustrate its application in ranking of Internet hosts. We finish with Chapter 5 that summarizes the dissertation and discuss future work.

# 2. PRUNED COMPANION FILES (PCF)*

## 2.1 Introduction

Enormous size of modern datasets poses scalability challenges for a variety of algorithms and applications. One particular area affected by the explosion of big data is *graph mining* and, more specifically, motif discovery in large networks. Motifs are important building blocks of real-life networks in biology, physics, chemistry, sociology, and computer science [31], [37], [51], [53], [76], [81]. They capture *local* composition of graphs and allow reasoning about the underlying construction processes that result in the observed phenomena. Three-node cycles (i.e., triangles) have received the most attention, attracting research interest for over 35 years [41] and developing many applications in graph theory [6], [55], [77], [78], [82], bioinformatics [43], [53], computer graphics [28], databases [5], and social networks [7], [11], [20], [88].

Until recently [85], little was known about the CPU cost of triangle listing, its behavior under different acyclic orientations, and comparison across the different methods. Much of the previous work [2], [38], [48] utilized $O(.)$ bounds that were exactly the same for all involved methods (i.e., vertex/edge iterators). As it turns out [85], there are $18$ algorithms for traversing the nodes of a triangle and handling the neighbors, which can be reduced to four equivalence classes from the CPU-cost perspective, each with its own optimal orientation. However, *external-memory* triangle listing remains largely unexplored. Given the same $18$ options, how many different I/O classes are there, what node permutations do they require, and is it possible for some methods to simultaneously achieve optimal CPU and I/O complexity using the same orientation?

If $m$ is the number of edges and $M$ is RAM size, previous implementations [3], [29],

[38], [44] operate with a simple I/O model that requires reading the graph $m/M$ times, for a total overhead of $m^2/M$. In theoretical development, better bounds can be achieved using random coloring of the graph [39], [60]; however, there are no implementations that use this method and the constants inside its bound $O(m^{1.5}/\sqrt{M})$ are unknown. What makes these two approaches similar is that their performance does not depend on the traversal order within each triangle or preprocessing manipulations applied to the graph, which leaves little for additional investigation.

Instead, we show below that there exists a technique for graph partitioning that maps the $18$ triangle-listing algorithms into six distinct classes, each of which possesses different I/O performance characteristics that depend on the acyclic orientation of the original graph. We call this framework *Pruned Companion Files* (PCF) and demonstrate how all $18$ methods can be combined under an umbrella of a single algorithm. Taking into account both I/O and CPU cost [85], we discover $16$ unique ways to perform triangle listing in external memory, none of which were known before.

While accurate modeling of I/O complexity is difficult, we are still able to identify the best partitioning scheme, deduce its optimal permutation, and prove that the amount of data read from disk is $\min(m^2/M, O(m))$ in random graphs with Pareto degree sequences, where shape parameter $\alpha > 4/3$. Note that this is the first result with linear I/O bounds under constant memory size. In contrast, both of the previous techniques [38], [60] require $M$ to scale at least as fast as $m$ to achieve the same performance. We also demonstrate that our partitioning scheme keeps the number of list intersections and table lookups unchanged compared to RAM-only methods, which means that its runtime remains constant for all $M$ as long as I/O is not the bottleneck.

To test these developments in practice, we build an implementation that combines PCF with a novel application of SIMD to edge iterator. Our solution, which we call PaCiFier, is benchmarked on a variety of real-world graphs, including four new ones that have not

been examined for triangles before. Our densest graph contains over 1T triangles, while the largest has over 100B edges. Results show that PaCiFier is $1-2$ orders of magnitude faster than the best vertex iterator [38] and $5-10$ times faster than the best edge iterator [29]. More importantly, it achieves $10-50$ times lower I/O complexity when RAM size is small compared to $m$.

### 2.2 Generalized Iterators (GI)

Recent work [85] created a taxonomy of 18 vertex and edge iterators. They use figures to highlight the intuitive differences among the methods; however, the lacking formal treatment makes it difficult to extend these results to external-memory scenarios. We therefore introduce a new description framework, which we call *Generalized Iterators* (GI), that explicitly encodes the traversal order in each triangle. This allows us to parameterize a single algorithm to cover execution of all alternative methods.

#### 2.2.1 Redundancy Elimination

Naive triangle-listing algorithms do not enforce order among the neighbors, which results in extremely inefficient operation. Besides discovering each triangle $3! = 6$ times, there are serious repercussions stemming from the fact that the number of pairs checked at each node is a quadratic function of its degree. Even on relatively small graphs, this can lead to $1000\times$ more overhead than necessary [85].

The redundancy can be eliminated by converting the graph into a directed version, in which quadratic complexity applies only to the out-degree (or in-degree, depending on the method), whose second moments are kept significantly smaller than those of undirected degree. Assume the nodes are first shuffled using some algorithm and sequentially assigned IDs from sequence $(1, 2, \ldots, n)$. This creates a total order across the nodes and is often called *relabeling*. A directed graph is then created, where out-neighbors of each node have smaller labels and in-neighbors have larger. This step is called *acyclic orienta-*

*tion*. Finally, in the directed graph, triangles $\Delta_{xyz}$ are listed in ascending order of the new labels, i.e., $x < y < z$.

This procedure generalizes all previous efforts in the field, some of which perform only relabeling [48], [67], [69] and others only orientation [3], [29], [38], [44], [67], [71], [74]. The drawbacks of not doing both are discussed in [85].

### 2.2.2 Relabeling

Consider a simple (i.e., no self-loops) undirected graph $G = (V, E)$ with $n$ nodes and $m$ edges. Define $\theta$ to be a permutation of node IDs that starts with the ascending-degree order and re-writes the label of each node in position $i$ to $\theta(i)$. Among the $n!$ possibilities, there are several named permutations [85], which include *ascending-degree* $\theta_A(i) = i$, *descending-degree* $\theta_D(i) = n + 1 - i$, *round-robin*

$$\theta_{RR}(i) = \begin{cases} \lceil \frac{n+i}{2} \rceil & i \text{ is odd} \\ \lfloor \frac{n-i}{2} \rfloor + 1 & i \text{ is even} \end{cases}, \tag{2.1}$$

and *complementary round-robin* $\theta_{CRR}(i) = \theta_{RR}(n + 1 - i)$, each of which optimizes a different class of triangle-listing methods [85]. The difference in CPU cost between the best and worst permutations can be orders of magnitude. Even worse, this ratio may be unbounded as $n \to \infty$ [85]. For a given permutation $\theta$, define its *reverse* to be $\theta'(i) = n + 1 - \theta(i)$. This is a useful concept that allows detection of equivalence classes later in the chapter.

Suppose $G_\theta$ is the relabeled graph under permutation $\theta$. Its construction typically requires sorting the degree sequence of $G$ using $\theta$, re-writing the source nodes of each list, inverting the graph using external memory, and re-writing the source nodes again. It is also common during this process to drop all nodes with degree one since they cannot be part of a triangle.

### 2.2.3 Orientation

Define $N_i$ to be the adjacency list of node $i$ in $G_\theta$ and $d_i = |N_i|$ to be its undirected degree. In general, $i \notin N_i$ because the graph is simple. Suppose the neighbors within each $N_i$ are sorted ascending by their ID and $G_\theta$ is kept as a sequence of pairs $\{(i, N_i)\}_{i=1}^n$. Our next goal is to define notation that allows splitting arbitrary sets into values smaller/larger than a given pivot. The most immediate use is construction of in/out lists in the directed graph, but we will encounter other applications shortly.

Suppose $\mathbb{N}$ is the set of natural numbers and consider two finite sets $S, T \subseteq \mathbb{N}$. Then, let

$$(T, S)^+ = \{j \in S | j \leq \max(T)\} \tag{2.2}$$

be a subset of $S$ that is bounded from above by the largest value in $T$. When $T$ consists of a single element $i$, we simply write $(i, S)^+$. Similarly, define

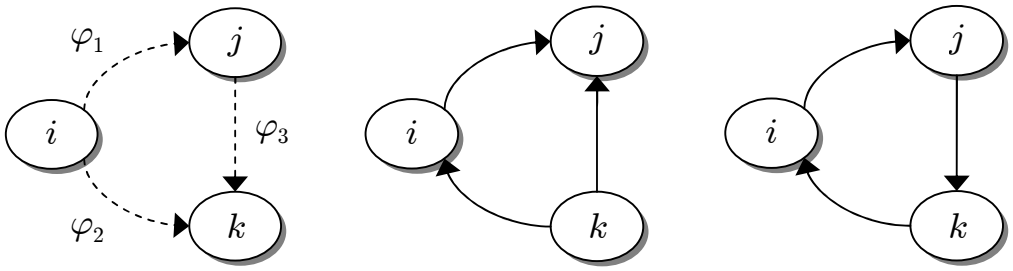
$$(T, S)^- = \{j \in S | j \geq \min(T)\} \tag{2.3}$$

to contain elements of $S$ no smaller than the minimum in $T$. Then, the out-list of $i$ in the oriented graph is given by $N_i^+ := (i, N_i)^+$, while the corresponding in-list by $N_i^- := (i, N_i)^-$.

When the $+/-$ operator is specified by a variable $\varphi$, i.e., $(T, S)^\varphi$, we say that $S$ *is $\varphi$-oriented by* $T$. This notation can be extended to other graph concepts. For example, $G_\theta^\varphi$ consists of tuples $\{(i, N_i^\varphi)\}$, where $i$ is the source node and $N_i^\varphi$ is its neighbor list, and $d_i^\varphi := |N_i^\varphi|$ is the corresponding degree in the directed graph. Define $1 - \varphi$ to be the *inverse* of operator $\varphi$, i.e., a plus becomes a minus and vice versa. It is then not difficult to see that $G_{\theta'}^\varphi$ is identical to $G_\theta^{1-\varphi}$, i.e., reversing the permutation is equivalent to inverting

the orientation.

### 2.2.4 Search Order




(a) general case (b) $+--$ (c) $+-+$

Figure 2.1: Search-order operators in triangle listing.

Given six different ways to permute the nodes of a triangle, we next show how $\varphi$ allows us to describe the various trajectories during search that result in exactly one listing of each triangle. Suppose $i$ is the first visited node by an algorithm, $j \in N_i$ is the second, and $k \in N_i$ is the last one. The larger/smaller relationship between these nodes is what differentiates the various traversal orders. All possible combinations are captured by Fig. 2.1(a), where each dashed arrow represents a $\varphi$-relationship between the two neighboring nodes. If labeled with a plus, a dashed arrow indicates that the source node is *larger* than the destination. The roles are reversed when the label is a minus. Note that unlike our earlier notation $\Delta_{xyz}$, where the order $x < y < z$ was fixed, the relationship between $(ijk)$ is fluid, i.e., changed by parameter $\bar{\varphi} = (\varphi_1, \varphi_2, \varphi_3)$.

Once the $\bar{\varphi}$ vector is chosen, the dashed arrows become oriented and are replaced with solid lines that specify greater-than relationships among the nodes. One example is shown in Fig. 2.1(b), where $k > i > j$. A simple rule to remember is that a $+$ keeps the direction of the dashed arrow, while a $-$ reverses it. Out of the $2^3 = 8$ possible $\bar{\varphi}$ vectors, two

produce loops, such as the one in Fig. 2.1(c). These are invalid because they lead to a contradiction, e.g., $k > i > j > k$. The remaining six combinations are studied next.

### 2.2.5 Algorithms

In Algorithm 1, we create the *generalized vertex iterator* (GVI) that can handle all valid $\bar{\varphi}$ vectors. The method starts by populating all directed edges from $G_\theta^{\varphi_3}$ into a hash table. The reason for using $\varphi_3$ is that the algorithm performs lookups of $(j, k)$ against $H$, which we know from Fig. 2.1(a) have relationship $\varphi_3$. Then, for each node $i$, GVI creates two sets – the *hit list* $X$, from which $j$ will be drawn, and the *local list* $Y$ consisting of neighbors $k$ that may complete a triangle. From Line 6, the algorithm examines every node $j \in X$, orients $Y$ using $\varphi_3$ with respect to $j$, and checks the resulting pairs $(j, k)$ against the hash table. Note that Line 7 is important for eliminating the possibility of redundancy.

**Algorithm 1:** Generalized vertex iterator

```
1 Function GVI (φ̄)
2     build hash table H with all directed edges from G_θ^{φ3}
3     for i = 1 to n do
4         X = (i, N_i)^{φ1}  ◁ neighbors of i in G_θ^{φ1} (hit list)
5         Y = (i, N_i)^{φ2}  ◁ same in G_θ^{φ2} (local list)
6         foreach j ∈ X do
7             Y' = (j, Y)^{φ3}  ◁ set Y φ3-oriented by j
8             foreach k ∈ Y' do
9                 if (j, k) ∈ H then report triangle Δ_{sort(ijk)}
```

The next technique is the *generalized lookup edge iterator* (GLEI) whose operation is presented in Algorithm 2. The main difference begins in line 5, where GLEI populates the local list $Y$ into a small hash table $H$. For each $j \in X$, the method constructs a *remote list* $Z$ consisting of $j$'s neighbors according to $\varphi_3$, orients it by $\varphi_2$ with respect to $i$, and checks its members against $H$. GLEI and GVI perform the same number of memory hits [85], with the only difference being the time needed to clear the hash table in Line 11.

**Algorithm 2:** Generalized lookup edge iterator

```
1  Function GLEI (φ̄)
2      for i = 1 to n do
3          X = (i, N_i)^φ1  ◁ neighbors of i in G_θ^φ1 (hit list)
4          Y = (i, N_i)^φ2  ◁ same in G_θ^φ2 (local list)
5          add elements of Y to hash table H
6          foreach j ∈ X do
7              Z = (j, N_j)^φ3  ◁ neighbors of j in G_θ^φ3 (remote list)
8              Z' = (i, Z)^φ2  ◁ set Z φ2-oriented by i
9              foreach k ∈ Z' do
10                 if k ∈ H then report triangle Δ_sort(ijk)
11         empty H
```

The last method is the *generalized scanning edge iterator* (GSEI), which is described by Algorithm 3. It relies on sequential traversal of neighbor lists to perform set intersection in Line 9. This is in contrast to GLEI that uses hash tables for this purpose. The rest of the algorithm is quite similar. Before intersecting local and remote lists $(Y, Z)$, the method orients them in Lines 7-8 to be consistent with Fig. 2.1(a). Note that the former is done by GVI and the latter by GLEI. In practice, orientation of the local list $Y$ imposes no additional overhead since $j$ monotonically increases within the loop, which is a consequence of $N_i^{\varphi_1}$ being sorted ascending. However, certain GSEI traversal orders require a binary search in the remote list $Z$ to locate $i$ [85].

**Algorithm 3:** Generalized scanning edge iterator

```
1  Function GSEI (φ̄)
2      for i = 1 to n do
3          X = (i, N_i)^φ1  ◁ neighbors of i in G_θ^φ1 (hit list)
4          Y = (i, N_i)^φ2  ◁ same in G_θ^φ2 (local list)
5          foreach j ∈ X do
6              Z = (j, N_j)^φ3  ◁ neighbors of j in G_θ^φ3 (remote list)
7              Y' = (j, Y)^φ3  ◁ set Y φ3-oriented by j
8              Z' = (i, Z)^φ2  ◁ set Z φ2-oriented by i
9              K = Intersect (Y', Z')
10             foreach k ∈ K do report triangle Δ_sort(ijk)
```

### 2.2.6 Taxonomy

Table 2.1: Taxonomy of Vertex/Edge Iterators

| GVI | GLEI | GSEI | Binary Search | Vector $\bar{\varphi}$ | $i$ | $j$ | $k$ |
|---|---|---|---|---|---|---|---|
| $T_1$ | $L_1$ | $E_1$ | No | $+++$ | $z$ | $y$ | $x$ |
| $T_2$ | $L_2$ | $E_2$ | No | $-++$ | $y$ | $z$ | $x$ |
| $T_3$ | $L_3$ | $E_3$ | No | $---$ | $x$ | $y$ | $z$ |
| $T_4$ | $L_4$ | $E_4$ | No | $++-$ | $z$ | $x$ | $y$ |
| $T_5$ | $L_5$ | $E_5$ | Yes | $+--$ | $y$ | $x$ | $z$ |
| $T_6$ | $L_6$ | $E_6$ | Yes | $--+$ | $x$ | $z$ | $y$ |

A combination of Algorithms 1-3 comprises our *Generalized Iterators* (GI) framework. Analysis above shows that each of the main algorithms (i.e., GVI, GLEI, GSEI) admits six traversal orders and that this classification is exhaustive (i.e., no other patterns are possible). Table 2.1 assigns names to all methods based on their $\bar{\varphi}$, specifying whether the edge iterators require a binary search and how to relate $(ijk)$ to $(xyz)$. In prior literature, $T_1$ can be found in [38], [44], [74], $E_1$ in [3], [29], [71], $E_2$ in [48], [67], $E_3$ in [15], [17], and $E_5$ in [69]. Methods $T_1$-$T_3$, $E_1$, $E_3$, $E_4$ are listed in [57].

While there are $18$ techniques total, their CPU cost can be reduced to just four non-isomorphic classes [85]; however, this may no longer hold when I/O is taken into account. What can be said for sure is that reversing $\theta$, or similarly inverting $\bar{\varphi}$, produces an identical method from the I/O standpoint. This allows reduction of scope to a subset of methods that cannot be converted into each other through inversion of $\bar{\varphi}$.

For example, keeping only methods that utilize $G_\theta^+$ for remote edges, i.e., $\varphi_3$ is the plus operator, would eliminate rows $(3, 4, 5)$ in Table 2.1. In that case, Fig. 2.2 shows the position of the remaining $9$ methods on a 2D plane, where the columns share the CPU cost, while the rows do the same for I/O. We use analysis from [85] to position the columns

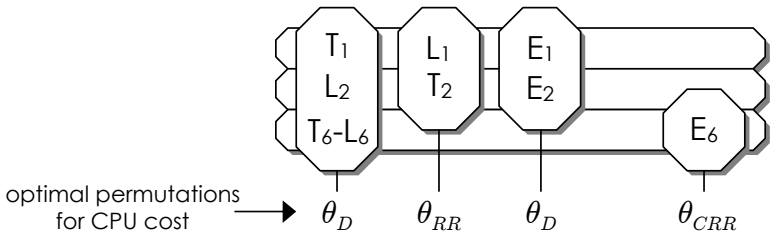



Figure 2.2: Four CPU and three I/O classes.

in order of increasing CPU complexity, with $T_1$ being the best and $E_6$ being the worst; however, it is currently unknown if the rows do in fact differ in cost, whether they can be split into multiple subrows depending on additional factors, and how their I/O relates to each other. This is our next topic.

## 2.3 Pruned Companion Files (PCF)

This section presents a general family of disk-based algorithms that supports all of the methods in Table 2.1. It also aims to achieve better I/O complexity than prior approaches.

### 2.3.1 Overview

It is important to discuss the performance objectives of external-memory algorithms before explaining our solution. There are four metrics that contribute towards the runtime of a method and its ability to handle large graphs. The first is the *triangle-identification time*, which consists of lookups against $H$ in GVI/GLEI and intersection in GSEI (i.e., Lines 9, 10, 9, respectively). For a method $\mathcal{M}$, suppose $c_n(\mathcal{M}, \theta)$ is the number of elementary operations, which we call the *CPU cost*, and $r(\mathcal{M})$ is the speed of these operations in nodes/sec. For a fixed pair $(i, j)$, the CPU cost equals $|Y'|$ for GVI, $|Z'|$ for GLEI, and $|Y'| + |Z'|$ for GSEI. Then, the triangle-identification time is given by $c_n(\mathcal{M}, \theta)/r(\mathcal{M})$.

The second metric is the amount of I/O performed. Because all reads are sequential, this overhead is measured by the length of adjacency lists across all graphs participating

in the algorithm. The third metric is the *number of lookups based on hit list* $X$ (i.e., Lines 6, 6, 5), which is generally a function of the partitioning scheme. This is in contrast to RAM-only operation, where this value is always fixed at $m$, i.e., the number of edges in $G_\theta$. Finally, the last parameter is the *minimum amount of RAM supported by the method*.

It is possible that some of these metrics are tradeoffs of each other; however, if an ideal algorithm exists, it would simultaneously beat the other methods in all four categories.

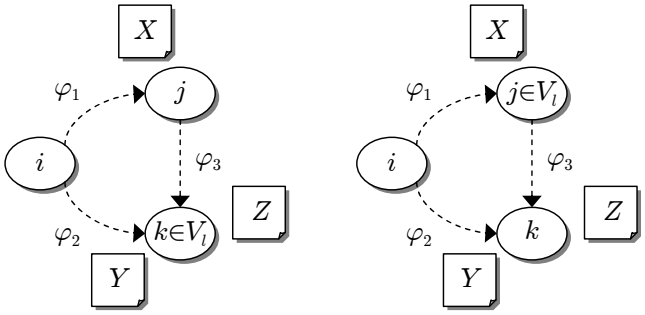
### 2.3.2 Graph Partitioning

Because GSEI explicitly maintains remote and local lists, both GVI and GLEI can be viewed as its special cases that replace one of the lists with a hash table. For example, GVI uses $H$ in place of scanning $Z$, while GLEI does the same for scanning of $Y$. As a result, any I/O partitioning scheme that handles GSEI can be adopted to work with the other two algorithms without incurring additional overhead. Therefore, our description of I/O techniques targets Algorithm 3.

In general, triangle-partitioning schemes work by placing one (or more) edges in some RAM buffer and then scanning the disk for discovery of the remaining edges that complete each triangle. Since node $j$ and its neighbors $k$ must be retrieved using random access, one crucial observation is that all methods require the *remote* edge $(j, k)$ to be present in RAM, while the other two lists $(X, Y)$ may be streamed from disk sequentially. This framework, coupled with general $\bar{\varphi}$ and the algorithms developed in this section, is what we call *Pruned Companion Files* (PCF).

Assume the set of nodes $V$ is divided into $p$ pair-wise non-overlapping and jointly exhaustive sets $\mathbf{V} = (V_1, \ldots, V_p)$. In a method we call PCF-A, we split $G_\theta^{\varphi_3}$ along the destination node of each pair $(j, N_j^{\varphi_3})$ to create a set of *remote-edge graphs*

$$G_\theta^r(l) = \{(j, N_j^{\varphi_3} \cap V_l)\}, \tag{2.4}$$




(a) PCF-A (b) PCF-B

Figure 2.3: Graph partitioning.

where $l = 1, 2, \ldots, p$. In a method we call PCF-B, we do the same along the source nodes

$$G_\theta^r(l) = \{(j, N_j^{\varphi_3}) | j \in V_l\}. \tag{2.5}$$

These technique are illustrated in Fig. 2.3 and their properties are given by the next result.

**Theorem 1.** *Algorithms 1-3 operating over PCF-A/B find each triangle exactly once. Furthermore, the triangle-identification cost $c_n(\mathcal{M}, \theta)$ remains constant for all $p$.*

*Proof.* First notice that every edge $(j, k)$ belongs to a unique partition $G_\theta^r(l)$. Then, replacing $G_\theta^{\varphi_3}$ with $G_\theta^r(l)$ in Algorithms 1-3 and repeating for all $l = 1, 2, \ldots, p$, we immediately obtain that no triangle is missed or counted more than once.

To show that the triangle-counting overhead remains constant, we focus on GSEI, with the other methods being similar. Fix a node $j$ and assume the length of its neighbor list $Z$ after orientation by node $i$ in Line 8 is given by $q_{ij}$. Note that list $Y'$ is independent of the partitioning scheme and can be ignored. For RAM-only operation, the intersection cost

related to $j$ can be expressed as

$$\sum_{(i,j)\in G_\theta^{\varphi 1}} q_{ij}. \tag{2.6}$$

In PCF-A, assume the length of $Z$ oriented by $i$ in partition $l$ is given by $q_{ij}(l)$. This leads to an overall cost for $j$

$$\sum_{l=1}^{p} \sum_{(i,j)\in G_\theta^{\varphi 1}} q_{ij}(l). \tag{2.7}$$

Since the partitions are mutually disjoint and exhaustive, it must be that for all $i$

$$\sum_{l=1}^{p} q_{ij}(l) = q_{ij}, \tag{2.8}$$

which yields the same cost in (2.7) as in (2.6) after changing the order of summations.

In PCF-B, the analysis is even simpler. Because $j$ appears as the source node in exactly one partition, it experiences the same overhead (2.6) in that partition and zero in all others.

□

This result shows that partitioning does not create any additional list-intersection operations, which allows us to focus on the remaining three objectives in the rest of the chapter.

### 2.3.3 Partition Balancing

Assume $M$ is the RAM size. To achieve the smallest $p$, each partition size $|G_\theta^r(l)|$ must equal $M$, which requires explicit balancing. Note that splitting the range $[1, n]$ into $p = m/M$ equal-size bins fails to accomplish this objective since permutation $\theta$ is degree-dependent. For example, with $\theta_D$, smaller node IDs indicate larger degree. Therefore,

nodes in the first bin may bring significantly more (or less depending on $\varphi_3$) edges into $G^r_\theta(l)$ than those in the last bin.

Balancing is accomplished by setting up boundaries $a_1, a_2, \ldots, a_{p+1}$ such that a node is included in $V_l$ if and only if it belongs to $[a_l, a_{l+1})$. While $a_1 = 1$ and $a_{p+1} = n + 1$ are obvious, the other values require more attention. For PCF-A in Fig. 2.3(a), notice that inclusion of $k$ into $V_l$ implies that all edges from list $N_k^{1-\varphi_3}$ are placed into $G^r_\theta(l)$. Therefore, we must select the boundaries such that

$$\sum_{k=a_l}^{a_{l+1}-1} d_k^{1-\varphi_3} = M, \tag{2.9}$$

which can be accomplished in one pass over $G_\theta^{1-\varphi_3}$. For PCF-B in Fig. 2.3(b), the roles of $j, k$ are reversed, which leads to

$$\sum_{j=a_l}^{a_{l+1}-1} d_j^{\varphi_3} = M. \tag{2.10}$$

Balancing in PCF-A and B is equally fast, except the former requires existence of an inverted version of $G_\theta^{\varphi_3}$.

### 2.3.4 Companion Files

The fastest previous implementations [3], [29], [38], [44] use a framework that would scan the entire file $G_\theta^{\varphi_1}$ to obtain hit lists $X$ and $G_\theta^{\varphi_2}$ for local lists $Y$. When $\varphi_1 = \varphi_2$, these files coincide, which cuts the overhead by half compared to other vectors $\bar{\varphi}$. Nevertheless, the amount of I/O produced by these schemes is still quite substantial, i.e., $mp = m^2/M$. Instead, our approach is to prune lists $X, Y$ to be optimally suited for each partition $l$ and write them into special *companion* files $G^c_\theta(l)$. Each of them, when paired with the corresponding remote-edge graph $G^c_\theta(l)$, allows identification of all triangles with either $k$

**Algorithm 4:** One-pass graph partitioning

```
1  Function PartitionGraph (method, φ̄, V)
2      for i = 1 to n do
3          X = (i, N_i)^{φ1}  ◁ hit list from G_θ^{φ1}
4          Y = (i, N_i)^{φ2}  ◁ local list from G_θ^{φ2}
5          Z = (i, N_i)^{φ3}  ◁ remote list from G_θ^{φ3}
6          for l = 1 to p do ◁ go through each partition
7              if method = PCF-A then
8                  X = (V_l, X)^{1−φ3}  ◁ hit list oriented by V_l
9                  Y = Y ∩ V_l  ◁ keep only nodes in V_l
10                 Z = Z ∩ V_l  ◁ keep only nodes in V_l
11             else
12                 X = X ∩ V_l  ◁ keep only nodes in V_l
13                 Y = (V_l, Y)^{φ3}  ◁ local list oriented by V_l
14                 Z = Z · 1_{i∈V_l}  ◁ Z if i ∈ V_l and ∅ otherwise
15             Y' = Y  ◁ local list to be written to G_θ^c(l)
16             if Z ≠ ∅ then
17                 write record (i, Z) into G_θ^r(l)
18                 if φ1 = φ3 then
19                     X = X\Z  ◁ further prune X
20                 if φ2 = φ3 then
21                     Y' = Y\Z  ◁ further prune Y
22             if X ≠ ∅ and Y ≠ ∅ and |X ∪ Y| ≥ 2 then
23                 write record (i, X, Y') to G_θ^c(l)
```

(PCF-A) or $j$ (PCF-B) in $V_l$.

Consider Algorithm 4, which is our one-pass solution to creating both companion and remote-edge files. If tuples $\{(i, N_i)\}$ are sorted by the source node $i$, Lines 3-5 simultaneously construct the three lists $(X, Y, Z)$ by scanning multiple files in parallel; otherwise, only methods with $\varphi_1 = \varphi_2 = \varphi_3$ are supported. In Lines 7-14, the algorithm prepares the necessary lists for each partition $l$. Among these, Line 8 can be explained with the help of Fig. 2.3(a). Notice that PCF-A can $(1 - \varphi_3)$-orient set $X$ with respect to $V_l$ without losing any relevant nodes $j$. Similarly Line 13 uses an observation from Fig. 2.3(b) that PCF-B can $\varphi_3$-orient $Y$ with respect to $V_l$ without omitting any essential nodes $k$.

In Lines 18-19, where $\varphi_1 = \varphi_3$ indicates that sets $X$ and $Z$ may overlap, the algorithm drops redundant edges from $X$. The same operation applies to $Y$ in Lines 20-21. Finally,

the companion file receives triple $(i, X, Y')$ if both hit list $X$ and local list $Y$ are non-empty, and there exist at least two nodes $j \in X$ and $k \in Y$ such that $j \neq k$.

Note that when $\varphi_1 = \varphi_2$, it is possible for $X$ to overlap with $Y$. An important aspect of these cases is that $Y$ is always $\varphi_3$-oriented against $X$. If additionally $Y' \neq \emptyset$, either $X \subseteq Y'$ or $Y' \subseteq X$ holds. Not only that, but the smaller list is always either at the bottom or top of the larger one. In such cases, only their union $X \cup Y'$ is written to disk, with an additional field indicating the offset that separates them. Algorithm 4 omits this detail to prevent clutter, but actual implementations should take it into account.

The main search function is shown in Algorithm 5. One noteworthy aspect is Line 8, which handles $X$ being in RAM for PCF-A, and Line 10, which does the same for PCF-B. In the latter case, only nodes $j \in V_l$ should be included in the hit list, which explains the need for additional pruning. Since $X$ being in RAM implies that $Y$ is too, Line 11 uses $N_i(l)$ as the local list. Processing of individual nodes is given by Algorithm 6, which is identical to the corresponding section of GSEI, except it finds $Z$ via the hash table rather than from the full graph $G_\theta^{\varphi_3}$.

## 2.4 Analysis

This section examines the introduced methods in comparison to each other. Our objective is to select a technique and its permutation so as to simultaneously maximize performance across all four criteria, if possible.

### 2.4.1 Overview

From this point on, we parameterize PCF with a specific $\bar{\varphi}$ from Table 2.1 by adding the corresponding row index. As before, we consider only rows $1, 2, 6$. When the A/B designation is non-essential, we omit it. For example, PCF-2 refers to $\bar{\varphi} = (-++)$ under both A/B, while PCF-2A narrows it down to the A partitioning scheme.

This creates the six I/O mechanisms in Table 2.2, where $i \to j$ signifies the out-list

**Algorithm 5:** Disk-based GSEI

```
1  Function FindTriangles (φ̄)
2      for l = 1 to p do
3          load G_θ^r(l) = {(i, N_i(l))} in RAM
4          build hash table H to map each i to its neighbor list N_i(l)
5          if φ_1 = φ_3 then ◁ possible for parts of X to be in RAM
6              foreach (i, N_i(l)) in RAM do
7                  if method = PCF-A then
8                      X = N_i(l) ◁ unrestricted hit list
9                  else
10                     X = N_i(l) ∩ V_l ◁ restrict hit list to V_l
11                 ProcessOneNode (φ̄, i, X, N_i(l))
12         while not EOF(G_θ^c(l)) do
13             read one record (i, X, Y) from companion G_θ^c(l)
14             if Y = ∅ then
15                 Y = H.find(i) ◁ local list must be in RAM
16             ProcessOneNode (φ̄, i, X, Y)
17         empty H
```

**Algorithm 6:** Modified GSEI intersection

```
1  Function ProcessOneNode (φ̄, i, X, Y)
2      foreach j ∈ X do
3          Z = H.find(j)  ◁ remote list is always in RAM
4          Y' = (j, Y)^{φ_3}  ◁ set Y φ_3-oriented by j
5          Z' = (i, Z)^{φ_2}  ◁ set Z φ_2-oriented by i
6          K = Intersect (Y', Z')
7          foreach k ∈ K do  report triangle Δ_{sort(ijk)}
```

neighbor relationship, i.e., $j \in N_i^+$, and $i \leftarrow j$ the opposite, i.e., $j \in N_i^-$. Note that PCF-1A and 2A place two edges in RAM and load the third one from disk. This explains why their local list $Y$ is always omitted from companion files. The remaining four techniques do the opposite – one edge is contained in $G_\theta^r(l)$ and two in $G_\theta^c(l)$. In three of these cases, edge direction is kept the same between $X$ and $Y$, which ensures that either $X \subseteq Y'$ or $Y' \subseteq X$, with only one of them actually written to disk. Method PCF-2B is the lone exception with its $X \cap Y' = \emptyset$.

Table 2.3 summarizes the pruning rules and specifies the contents of each companion

Table 2.2: Summary of PCF Algorithms Using Remote Graph $G_\theta^+$

| PCF | $G_\theta^r(l)$ | Condition | $X$ | $Y'$ |
|---|---|---|---|---|
| 1A | $(y,z) \to x$ | $x \in V_l$ | $z \to y$ | $\emptyset$ |
| 2A | $(y,z) \to x$ | $x \in V_l$ | $y \leftarrow z$ | $\emptyset$ |
| 6A | $z \to y$ | $y \in V_l$ | $x \leftarrow z$ | $x \leftarrow y$ |
| 1B | $y \to x$ | $y \in V_l$ | $z \to y$ | $z \to x$ |
| 2B | $z \to x$ | $z \in V_l$ | $y \leftarrow z$ | $y \to x$ |
| 6B | $z \to y$ | $z \in V_l$ | $x \leftarrow z$ | $x \leftarrow y$ |

list. Notice that PCF-1B uses stricter conditions for achieving $X, Y \neq 0$ than PCF-1A and its $X \cup Y'$ is the same or smaller, which indicates that it out-performs its counterpart. Assuming $\theta_D$, further scrutiny of companion lists in Table 2.3 reveals that PCF-1A produces less I/O than any of the remaining four methods, with PCF-6A/6B being essentially identical to each other.

### 2.4.2 Modeling I/O

Additional insight can be gleaned from bounding the size of companion files. Assume $u_{il}$ is the length of $i$'s hit list $X'$ in $G_\theta^c(l)$ and $v_{il}$ is that of $Y' \backslash X'$. Then, the total amount of companion I/O (in edges) is $H^c = H_X^c + H_Y^c$, where

$$H_X^c = \sum_{i=1}^{n} \sum_{l=1}^{p} u_{il}, \quad H_Y^c = \sum_{i=1}^{n} \sum_{l=1}^{p} v_{il}, \tag{2.11}$$

and that for remote-edge graphs is

$$H^r = \sum_{i=1}^{n} |G_\theta^r(l)| = m. \tag{2.12}$$

Since $H^r$ is constant for all $\bar{\varphi}$, comparison across the various approaches in Table 2.2 needs to involve only $H^c$. Closed-form derivation of accurate models for (2.11) currently appears intractable. Even ballparking the scaling rate is quite elusive for certain extremely

Table 2.3: Composition of Companion Lists in PCF

| PCF | $X$ | $Y$ | $Y'$ |
|---|---|---|---|
| 1A | $N_i^+ \cap [a_{l+1}, n]$ | $N_i^+ \cap V_l$ | $\emptyset$ |
| 1B | $(N_i^+ \cap V_l) \cdot \mathbf{1}_{i \geq a_{l+1}}$ | $N_i^+ \cap [1, a_{l+1})$ | $Y$ |
| 2A | $N_i^-$ | $N_i^+ \cap V_l$ | $\emptyset$ |
| 2B | $N_i^- \cap V_l$ | $N_i^+$ | $Y \cdot \mathbf{1}_{i \notin V_l}$ |
| 6A | $N_i^- \cap [a_l, n]$ | $N_i^- \cap V_l$ | $Y$ |
| 6B | $N_i^- \cap V_l$ | $N_i^- \cap [1, a_{l+1})$ | $Y$ |

heavy-tailed degree distributions [85]. Instead, we offer bounds achievable in two worst-case scenarios and leave more precise modeling for future work. Assume $H^c(k)$ refers to the companion overhead of PCF-$k$ and consider the next result.

**Theorem 2.** *The PCF I/O complexity (in edges) is upper-bounded by*

$$H^c(1) \leq \sum_{i=1}^{n} \min\Big(\frac{d_i^+ - 1}{2}, p-1\Big) d_i^+, \tag{2.13}$$

$$H^c(2) \leq \sum_{i=1}^{n} \min(d_i^+, p) d_i^-, \tag{2.14}$$

$$H^c(6) \leq \sum_{i=1}^{n} \min\Big(\frac{d_i^- + 1}{2}, p\Big) d_i^-, \tag{2.15}$$

*where $d_i^+$ is the out-degree of $i$ and $d_i^-$ is the in-degree.*

*Proof.* We only consider PCF-A since PCF-B uses similar arguments and produces the same bounds. It is not difficult to see that PCF-1A writes $H^c = H_X^c$ edges to companion files since its pruned hit lists $Y'$ are always empty. First, notice that a list cannot be split into more than $p$ chunks. Due to removal of overlap $X \cap Z$, we can do even better – the last partition $V_p$ produces a hit list $X$ only for neighbors $j \geq a_{p+1} = n + 1$. Since no label can exceed $n$, there are actually at most $p - 1$ partitions where $u_{il} \neq 0$. Therefore, $\sum_{l=1}^{p} u_{il} \leq (p-1) d_i^+$.

Our second observation is that an out-list cannot be split into more than $d_i^+$ files. Then, the worst case arises when each $V_l$ consists of a single node, where partition $l$ contains the largest $d_i^+ - l$ out-neighbors of $i$. Thus,

$$\sum_{l=1}^{p} u_{il} \leq \sum_{l=1}^{d_i^+} u_{il} \leq \sum_{l=1}^{d_i^+} (d_i^+ - l) = \frac{d_i^+(d_i^+ - 1)}{2}, \tag{2.16}$$

which combined with the first case yields (2.13).

For PCF-2A, the first case is very similar, except it uses the in-degree $d_i^-$ and fails to remove the overlap $X \cap Z$. The second case writes the full in-neighbor list exactly $d_i^+$ times, which yields the result in (2.14). Finally, PCF-6A operates similar to 1A, except it uses the in-degree and fails to prune the lists as efficiently. Due to these small differences, its bound (2.15) is not perfectly symmetrical to (2.13). □

Using [85], we obtain that the I/O bound of PCF-1 is minimized by the descending-degree permutation $\theta_D$, that of PCF-2 by round-robin $\theta_{RR}$, and that of PCF-6 by ascending-degree $\theta_A$. Furthermore, under their respective optimal permutations, (2.14) is strictly worse than (2.13). The bound of PCF-6 under $\theta_A$ rivals that of PCF-1 under $\theta_D$, although it is still slightly higher due to a less-efficient pruning of overlap between $(X, Y)$ and $Z$. The worst permutations corresponding to (2.13)-(2.15) are $\theta_A$, $\theta_{CRR}$, and $\theta_D$, respectively [85].

For the asymptotics, let $D_n \sim F_n(x)$ be the random degree of a node in a graph of size $n$. As $n \to \infty$, suppose $F_n(x) \to F(x)$ and let $D \sim F(x)$. Then, under $\theta_D$ and $E[D^{4/3}] < \infty$, the scaling rate of (2.13) is *no worse than linear* [85]

$$H^c(1) \leq \min\Big(\frac{m^2}{M}, O(m)\Big). \tag{2.17}$$

For example, Pareto distributions $F(x) = 1 - (1 + x/\beta)^{-\alpha}$ satisfy this requirement iff

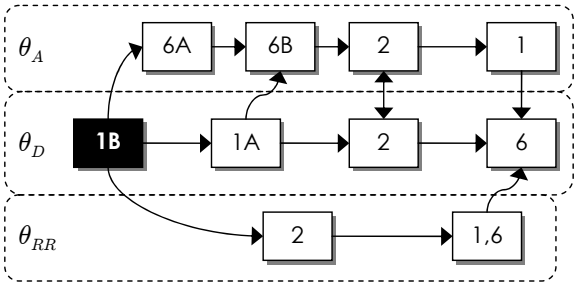



Figure 2.4: Better-than relationships across the I/O of various PCF methods.

$\alpha > 4/3$. For PCF-2 and $\theta_{RR}$, the rate (2.17) holds iff $\alpha > 1.5$ [85]. Note that (2.17) is strictly better than $m^2/M$ from prior implementations [3], [29], [38], [44]. When $M$ is a constant, it is also better than theoretical results of [39], [60] whose $O(m^{1.5}/\sqrt{M})$ bound cannot be linear unless $M$ grows at least as fast as $m$.

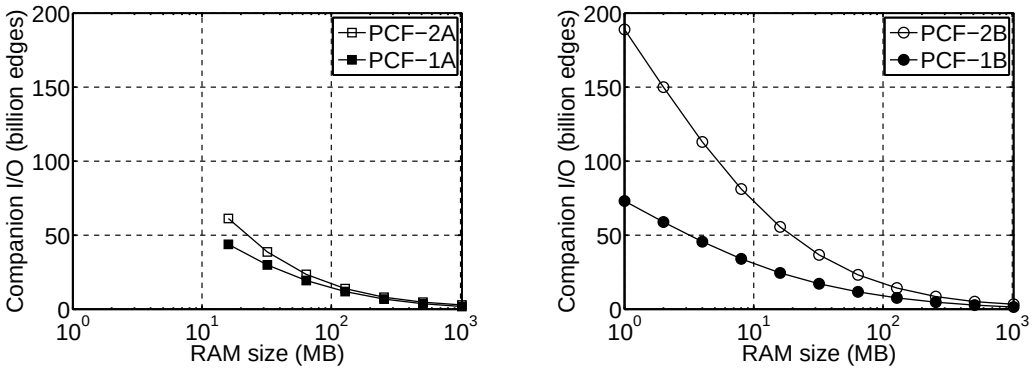
Based on Table 2.3, Theorem 2, and symmetry of PCF-1A/6B and 1B/6A, Fig. 2.4 places the I/O of the various methods in relationship to each other under different permutations. When we do not differentiate between the PCF variants A/B of a given method, it is usually because they have similar I/O. From the picture, it emerges that PCF-1B with $\theta_D$ is globally the most efficient technique.

### 2.4.3 I/O Comparison

For an illustration of the ideas presented earlier in this section, we employ the commonly considered Twitter graph [45] with $41$M nodes and $m = 1.2$B edges. The file occupies $9.3$ GB and its adjacency lists contain $2m = 2.4$B node IDs. We start with Table 2.4, which shows the size of companion files $H^c$. Observe that the predicted best-case permutations in each column (highlighted in gray) agree with earlier analysis. Additionally, notice that reversal of $\theta$ swaps PCF-A/B, switches PCF-1 to PCF-6, and maps PCF-2 back to itself. These effects were expected based on (2.13)-(2.15). Even though PCF-1 and PCF-6 are close under their optimal permutations, the former comes out ahead for the

Table 2.4: Twitter I/O (in Billion Edges) Under 16 MB of RAM

| Permutation | 1A | 1B | 2A | 2B | 6A | 6B |
|---|---|---|---|---|---|---|
| $\theta_D$ | 43.8 | 24.5 | 61.3 | 55.6 | 119.1 | 126.8 |
| $\theta_{RR}$ | 94.1 | 83.0 | 51.0 | 51.7 | 83.6 | 94.2 |
| $\theta_A$ | 125.7 | 118.4 | 54.8 | 61.7 | 25.5 | 44.2 |



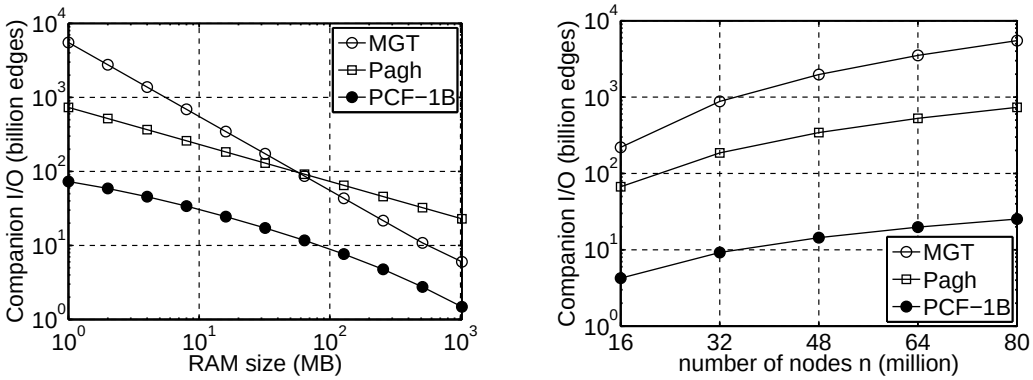

(a) PCF-A (b) PCF-B

Figure 2.5: Scaling rate of PCF-1 on Twitter under $\theta_D$.

reasons discussed above.

We now examine how the methods scale as $M \to 0$. We dismiss PCF-6 due to its similarity to PCF-1. We also fix $\theta_D$ since it achieves the best CPU cost among the methods in Fig. 2.2. We vary RAM size from 1 GB down to 1 MB and plot the result in Fig. 2.5, where PCF-A cannot go lower than 16 MB due to inability to fit the largest in-degree into RAM. Observe that not only is PCF-1 more efficient than PCF-2, but the gap between the two grows as $M$ decreases. As $M \to 0$ and $p \to \infty$, both methods converge towards their upper bounds, which are 150B in (2.13) and 360B in (2.14) [85]. The figure shows that PCF-1 is getting there at a slower pace than PCF-2.

We next analyze the scaling rate of our best method PCF-1B against the two previous models of I/O. Recall that the $m^2/M$ technique was proposed by MGT [38], while the




(a) Twitter (b) random graphs ($M = 1$ MB)

Figure 2.6: Comparison against prior methods.

$O(m^{1.5}/\sqrt{M})$ bound is due to Pagh *et al.* [60]. Since there is no actual implementation for the latter, it is difficult to assess the constants inside $O(.)$. We thus take some liberty in assuming how this method would work in practice. It randomly colors the nodes using $c = \sqrt{m/M}$ unique values and splits the edges into $c^2$ files based on the color of source/destination nodes. It then combines three files of colors $(ij, jk, ki)$ and runs MGT over the result. Since the size of each combined subgraph is $3m/c^2$, the I/O cost of the method is $9m^{1.5}/\sqrt{M}$, which accounts for all $c^3$ combinations of triplets $(ij, jk, ki)$. While [60] deals with undirected graphs, whose size is $\sum_{i=1}^{n} d_i = 2m$ edges, we assume the method can be applied to $G_\theta^+$. Thus, both MGT and Pagh use $m = 1.2$B in their respective models.

The result for Twitter and $M \to 0$ is shown in Fig. 2.6(a). After the initial jump, PCF-1B becomes parallel to Pagh's curve $1/\sqrt{M}$. Both of them scale significantly better than MGT's inverse linear function. In Fig. 2.6(b) we use random graphs with a Pareto degree distribution ($\alpha = 1.5$, $E[D] = 30$) to examine the scaling rate of I/O as $n \to \infty$. In this range, PCF-1B is roughly linear, while the other two methods grow significantly faster. As $n$ increases, the ratio of MGT to PCF-1B jumps from $51$ to $219$, while that for

Table 2.5: CPU-I/O Complexity Classes in Twitter under 16 MB of RAM

| Under CPU-optimal permutation | | | | Under I/O-optimal permutation | | | |
|---|---|---|---|---|---|---|---|
| Perm | GI | CPU | I/O | Perm | GI | CPU | I/O |
| $\theta_D$ | $T_1$ | 150B | 24B | $\theta_D$ | $T_1$ | 150B | 24B |
| | $L_2$ | 150B | 56B | | $L_1$ | 360B | 24B |
| | $T_6$-$L_6$ | 150B | 119B | | $E_1$ | 511B | 24B |
| | $E_1$ | 511B | 24B | $\theta_{RR}$ | $T_2$ | 255B | 51B |
| | $E_2$ | 511B | 56B | | $L_2$ | 63T | 51B |
| $\theta_{RR}$ | $L_1$ | 255B | 83B | | $E_2$ | 63T | 51B |
| | $T_2$ | 255B | 51B | $\theta_A$ | $T_6$-$L_6$ | 123T | 25B |
| $\theta_{CRR}$ | $E_6$ | 63T | 45B | | $E_6$ | 123T | 25B |

Pagh from 15.5 to 29.3. To put this in perspective, $n = 80$M nodes requires 25B edges of I/O for PCF-1B, 734B for Pagh, and 5.5T for MGT.

### 2.4.4 CPU-I/O Tradeoffs

As it turns out, Fig. 2.2 splits into 16 different CPU-I/O complexity classes, i.e., two (A/B) for each of the 8 unique GI methods, with $T_6$-$L_6$ being a single entity. In the past, it was believed that GVI and GLEI were functionally identical. However, this is not the case when I/O is taken into account. For example, $T_1$ shares the I/O cost with $L_1$, but at lower CPU complexity. Similarly, it shares the CPU cost with $L_2$-$L_6$, while imposing less I/O. In the same vein, it was unknown until now whether $E_1$ and $E_2$ were interchangeable. Results above confirm that they are not.

These observations are emphasized using Table 4.18, where each I/O cell reports the best number achieved by either PCF-A or B. Observe that the best GVI is $T_1$, which exhibits optimal CPU and I/O complexity under $\theta_D$. The decision is also easy for GSEI, where $E_1$ is the top contender. On the other hand, GLEI must choose which of the two objectives is more important – $L_1$ has the best I/O and $L_2$ the best CPU cost, both under $\theta_D$. Other GLEI combinations are much worse.

### 2.4.5 Lookups and Minimum RAM

Recalling that PCF-B prunes $X$ such that $X \subseteq V_l$ holds, while PCF-A does not, the next result follows immediately.

**Theorem 3.** *PCF-A issues $H_X^c$ hit-list lookups and requires $M \geq \max_i d_i^{1-\varphi_3}$. PCF-B performs exactly $m$ lookups and requires $M \geq \max_i d_i^{\varphi_3}$.*

In graphs with heavy-tailed degree and $M \ll m$, it is common that the hit list size $H_X^c \gg m$ (e.g., see Table 2.4). Therefore, for small RAM size, PCF-B should have a noticeably better CPU performance than PCF-A. In fact, its number of hash-table hits is optimal as it equals that in RAM-only algorithms.

In terms of restrictions on RAM, all considered methods PCF-1/2/6 have a plus for $\varphi_3$, which means that PCF-A lower-bounds $M$ by the largest *in-degree*, while PCF-B by the largest *out-degree*. It is well-known that $\theta_D$ keeps the latter no larger than $\sqrt{2m}$; however, its maximum in-degree equals $\max_i d_i$, which can be significantly higher, i.e., up to $n-1$. Therefore, PCF-B under $\theta_D$ is definitively less restrictive than PCF-A. When the permutation is reversed, the bounds on in/out degree are swapped and PCF-A becomes better than PCF-B. Finally, $\theta_{RR}$ has both maximum in/out degree equal to $\max_i d_i$, which makes this permutation equally bad in both PCF-A/B.

### 2.4.6 Summary

From the analysis above, two methods $\mathrm{T}_{1\mathrm{B}}$ and $\mathrm{E}_{1\mathrm{B}}$ emerge as clear winners within their respective classes (i.e., hash tables and scanning intersection). Among the $18$ methods, they achieve the smallest companion I/O, perform the minimal number of hit-list lookups, impose the lowest RAM requirements, do not need to invert $G_\theta^{\varphi_3}$ during creation of $\{V_l\}$, and obtain $(X, Y, Z)$ from a single file in Algorithm 4.

We next consider which of them has a smaller runtime. There are two aspects involved

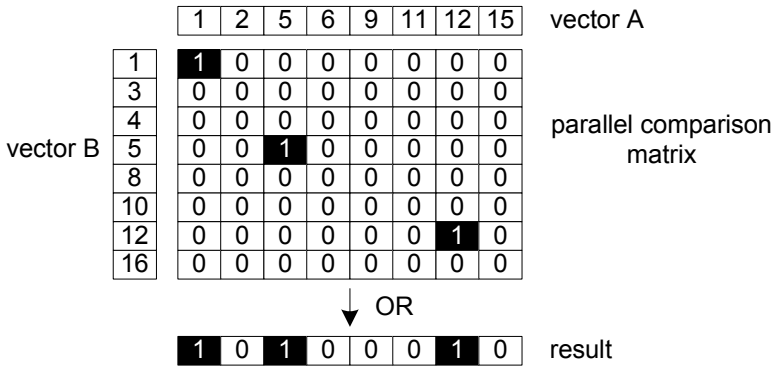



Figure 2.7: Parallel intersection with STTNI.

– the relative CPU cost

$$w_n := \frac{c_n(E_1, \theta_D)}{c_n(T_1, \theta_D)} \tag{2.18}$$

and the relative speed $s = r(E_1)/r(T_1)$. While [85] proves existence of random graphs where $w_n \to \infty$ as $n \to \infty$, ratio $w_n$ is only $2 - 3$ in real graphs commonly studied in this area. Given that $s$ is at least $20$ on modern CPUs, it is conclusive that scanning edge iterators will remain the best option until graphs are discovered with significantly larger $w_n$.

## 2.5 Implementation

We now build a fast implementation of $\mathrm{E}_{1\mathrm{B}}$ that takes advantage of SIMD for scanning the lists and PCF-B for I/O. We call this method PaCiFier and make it available in [22].

### 2.5.1 Intersection

Since $\mathrm{E}_{1\mathrm{B}}$ spends almost all of its CPU time on intersection, it is crucial to address this bottleneck first. With support for SIMD in modern CPUs, we can exploit data-level parallelism and achieve a significant speedup compared to traditional CPU-based methods. We adopt the technique from [68], which utilizes STTNI intrinsics from SSE 4.2. They

Table 2.6: Single-Core Speed (Intel i7-3930K @ 4.4 GHz)

| Implementation | Speed (M/sec) |
|---|---|
| Hash table | 19 |
| Naive scalar intersection | 264 |
| Branchless intersection | 416 |
| SIMD 32-bit intersection | 1,119 |
| SIMD 16-bit intersection | 1,801 |

Table 2.7: Dataset Properties

| Graph | Nodes ($n$) | Degree sum ($2m$) | Size |
|---|---|---|---|
| LJ | 4,846,609 | 85,702,474 | 364 MB |
| USRD | 23,947,347 | 57,708,624 | 403 MB |
| BTC | 164,660,997 | 772,822,094 | 4.1 GB |
| WebUK | 62,338,347 | 1,877,431,056 | 7.5 GB |
| Twitter | 41,652,230 | 2,405,026,390 | 9.3 GB |
| Yahoo | 720,242,173 | 12,869,122,070 | 53.3 GB |
| IRL-domain | 86,534,416 | 3,416,273,404 | 13.3 GB |
| IRL-host | 641,982,060 | 12,872,821,328 | 52.7 GB |
| IRL-IP | 1,588,925 | 1,636,848,800 | 6.1 GB |
| ClueWeb | 8,179,508,503 | 102,394,528,124 | 358 GB |

work on two 128-bit vector registers, treating them as four 32-bit or eight 16-bit integers. Fig. 2.7 shows how STTNI builds an all-to-all comparison matrix and outputs a vector of matches using just one instruction. While 32-bit intersection is fast, better results can be procured by compressing labels into 16-bit numbers. This is performed by grouping node IDs into chunks that share the same upper 16 bits. For each chunk, PaCiFier additionally keeps its length and a list of the lower two bytes from each original label. This works well because all vertices are sequentially relabeled and adjacency lists are kept in ascending order. Besides almost doubling intersection speed, this method reduces graph size by approximately 50%.

For lists that are shorter than some threshold (e.g., 16), both compression and 16-bit

Table 2.8: Dataset Triangle Properties

| Graph | Triangles | $w_n$ | $c_n(E_1, \theta_D)$ | $E[d_i]$ | $\max_i d_i$ | $\max_i d_i^+$ |
|---|---|---|---|---|---|---|
| LJ | 285,730,264 | 3.01 | 2.1B | 17.7 | 20,333 | 685 |
| USRD | 438,804 | 2.37 | 25M | 2.4 | 9 | 4 |
| BTC | 28,498,939 | 1.59 | 3.5B | 4.7 | 1,637,619 | 646 |
| WebUK | 179,076,331,071 | 1.99 | 364B | 30.1 | 48,822 | 5,692 |
| Twitter | 34,824,916,864 | 3.38 | 511B | 57.7 | 2,997,487 | 4,102 |
| Yahoo | 85,782,928,684 | 1.47 | 433B | 17.9 | 7,637,656 | 1,540 |
| IRL-domain | 112,797,037,447 | 3.63 | 1.4T | 39.5 | 2,948,635 | 4,481 |
| IRL-host | 437,436,899,269 | 2.85 | 2.6T | 20.1 | 5,475,377 | 4,516 |
| IRL-IP | 1,032,158,059,864 | 3.17 | 4.2T | 1,030 | 669,776 | 8,915 |
| ClueWeb | 879,280,163,294 | 2.00 | 3.0T | 12.5 | 44,383,637 | 1,747 |

intersection do not work well. In these cases, we keep the lists in 32-bit format and apply the branchless scalar (i.e., non-SIMD) intersection from [40]. A benchmark of these operations together with the Google Hash Table are shown in Table 2.6. With 1.8B operations/sec, PaCiFier's ratio $s$ is a whopping 94.7. This places even more doubt that $T_{1B}$ will be competitive in the near future, especially given that RAM bandwidth scales much faster than latency [65], i.e., $s$ will continue increasing.

### 2.5.2 Relabeling and Orientation

For degree-based permutations, prior work sorts pairs (degree, ID) to establish a total order. This becomes a major bottleneck in preprocessing, especially for large graphs where these tuples do not fit in RAM. In contrast, we use a novel approach that decides the new labels without sorting the nodes. We first accumulate a histogram of degree frequency in one pass over pairs $(i, d_i)$, which are kept separately from the adjacency lists $\{N_i\}$. Using a prefix sum of the histogram, we then establish the starting IDs for nodes of each unique degree value. Performing another scan of the tuples, we find the degree of each source node $i$ in the histogram and create a mapping from old labels to the corresponding new IDs. This is shown in Fig. 2.8. Frequently accessed parts of the histogram typically fit in

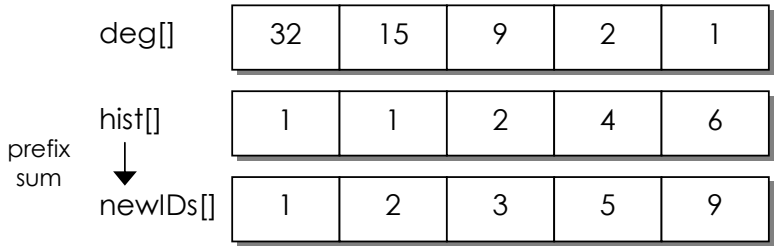



Figure 2.8: Descending-degree relabel with a histogram.

the L2 cache, which makes lookups against them extremely fast.

If the mapping fits in RAM, PaCiFier performs a scan over the adjacency lists and rewrites all edges in one pass. Otherwise, it changes the source nodes, inverts the graph, and updates the source nodes again.

### 2.5.3 Parallelization

Scaling PaCiFier to multiple cores is rather straightforward. In Algorithm 5, the processing of each record $(i, X, Y) \in G^c_\theta(l)$ is an independent job, which allows multiple threads to work on different lists without interfering with each other. The lookup table $H$ is read-only and can be safely shared by all worker threads without any locks. Assuming $c$ available cores and hyper-threading, we run $2c$ worker threads and set the affinity mask to bind each thread to a dedicated core. This configuration ensures $100\%$ CPU utilization for the entire execution and almost linear scalability with the number of cores (see below).

### 2.5.4 Evaluation Setup and Datasets

Experiments use a six-core Intel i7-3930K @ $4.4$ GHz, Asus Rampage IV Extreme motherboard, and quad-channel DDR3 RAM @ 2133 MHz. We compare PaCiFier against four methods with available implementations – RGP [17], DGP [17], MGT [38], and PDTL [29]. For the first three techniques, we use a multi-threaded binary shared by the authors of [38].

We employ all standard graphs in the field – Live Journal (LJ) [38], US road maps (USRD) [38], Billion Triples Challenge (BTC) [35], WebUK [38], Twitter [45], and Yahoo [87]. Note that the original Yahoo graph has $n = 1.4$B, which reduces to 720M after removing zero-degree nodes. To cover a wider variety of options, we add two web crawls: IRLbot [49] and ClueWeb [19]. Out of the former, we extract domain, host, and IP-level graphs. Assuming $I(x)$ is the IP address of an authoritative nameserver for domain $x$, graph IRL-IP contains edges $I(x) \to I(y)$ iff $x \to y$ in IRL-domain, which may be useful for spam detection and ranking. The original ClueWeb dataset published online [19] does not contain any dynamic links and is limited to 7.9B edges [63]. We remedy this problem by running our HTML parser over all pages, which yields a much larger graph with 102B links. The new files can be downloaded from [22].

Tables 2.7-2.8 summarize statistics of the graphs, where the old datasets require billion-scale intersection cost $c_n(E_1, \theta_D)$ and the new ones trillion-scale. The densest graph IRL-IP has an average degree 1,030, contains over 1T triangles, and requires 4.2T intersection operations. ClueWeb comes in at a hefty 358 GB, but neither its number of triangles nor CPU cost can top those of IRL-IP. Also note that the longest out-list in the table occupies just 35 KB of RAM, far smaller than the longest undirected neighbor set (i.e., 177 MB).

### 2.5.5 Preprocessing Time

RGP/DGP do not require preprocessing, while the other three methods manipulates the input graph $G$ into a suitable format prior to actual listing of triangles. It is common to time the two phases separately, especially since the former can be performed once and the latter repeated many times on the same preprocessed data. Table 2.9 shows the result using a RAID system capable of reads at 1 GB/s. Even though PaCiFier is the only one performing both relabeling and orientation, its usage of the histogram to avoid sorting makes it $2 - 8$ times faster than MGT and up to 20 times faster than PDTL.

Table 2.9: Preprocessing Time (Seconds)

| Graph | MGT | PDTL | PaCiFier |
|---|---|---|---|
| LJ | 2.2 | 1.0 | 1.7 |
| USRD | 2.0 | 1.4 | 2.0 |
| BTC | 18.8 | 11.6 | 8.9 |
| WebUK | 36.9 | 24.5 | 14.7 |
| Twitter | 88.9 | 38.4 | 24.5 |
| Yahoo | 295 | 276 | 149 |
| IRL-domain | 149 | 61.9 | 31.8 |
| IRL-host | 736 | 456 | 221 |
| IRL-IP | 33.9 | 19.1 | 8.5 |
| ClueWeb | 8,192 | 19,502 | 962 |

### 2.5.6 Triangle-Listing Time

We run the next set of tests using an 8-GB RAM constraint, which ensures that I/O is not a bottleneck for our RAID. As a result, Table 2.10 presents an evaluation of pure CPU efficiency of each algorithm. PaCiFier's performance is determined by the length of neighbor lists, i.e., efficiency of SIMD scanning. Compared to MGT, which implements $T_1$, its speedup varies from a factor of 13.6 on Yahoo to 78.6 on IRL-IP. In the latter graph, PaCiFier finds 1T triangles in 237 seconds, which translates into 17.7B neighbor checks/sec and 4.3B discovered triangles/sec using all six cores. Compared to PDTL, which is an optimized version of $E_1$ with MGT's partitioning scheme, PaCiFier achieves a $5 - 10\times$ faster runtime.

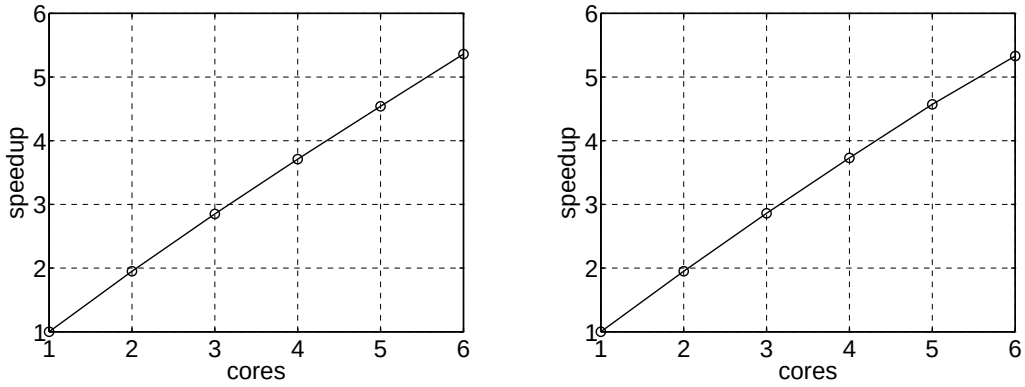
The number of found triangles is consistent across the methods, except RGP/DGP fail to finish within 12 hours on several graphs, which we indicate with a dash. Additionally, MGT quits with an unrealistically small number of triangles (i.e., 170M) after spending 24K seconds on ClueWeb, which we show with an asterisk. Its traces point toward early termination before processing all of the partitions; however, unavailability of the source code prevents further analysis.

Table 2.10: Runtime (Seconds) With 8 GB of RAM

| Graph | RGP | DGP | MGT | PDTL | PaCiFier |
|---|---|---|---|---|---|
| LJ | 22.3 | 22.2 | 11.2 | 2.8 | 0.7 |
| USRD | 12.3 | 12.3 | 1.2 | 6.2 | 0.3 |
| BTC | 111 | 110 | 11.4 | 12.1 | 2.1 |
| WebUK | 1,299 | 891 | 599 | 93.6 | 17.1 |
| Twitter | 10,300 | 9,814 | 2,238 | 327 | 63.4 |
| Yahoo | 31,945 | 13,990 | 1,080 | 619 | 79.2 |
| IRL-domain | 17,717 | 16,919 | 5,946 | 849 | 148 |
| IRL-host | – | – | 11,099 | 1,773 | 367 |
| IRL-IP | – | – | 18,617 | 2,358 | 237 |
| ClueWeb | – | – | * | 13,782 | 1,737 |

Table 2.11: Results from Prior Work

| Type | Algorithm | Runtime (sec) | | Cores or |
|---|---|---|---|---|
| | | Twitter | Yahoo | servers |
| RAM-only | [69] | 101 | – | 16 |
| | [71] | 55.9 | 77.7 | 40 |
| External | PATRIC [3] | 552 | – | 200 |
| | OPT [44] | 469 | 819 | 6 |
| MapReduce | [20] | 36,300 | – | 47 |
| | GP [74] | 28,980 | – | 1,636 |
| | TTP [62] | 12,780 | – | 47 |
| | CTTP [63] | 5,520 | 61,920 | 40 |




(a) Twitter (b) IRL-domain

Figure 2.9: Speedup vs. number of cores (8 GB of RAM).

To put these results in perspective, Table 2.11 cites the runtime from prior work on Twitter and Yahoo. We split the algorithms into several categories – RAM-only, external-memory, and MapReduce. We report the number of utilized cores for the former two groups and cluster size for the last one. The first two methods in the table [69], [71] produce comparable numbers to those of PaCiFier, but using $3-6$ times more late-model Xeon cores. Due to their RAM-only operation, we do not consider them competitors for PaCiFier. The next two techniques [3], [44] are extensions of RGP/DGP and MGT to multiple machines. They are generally faster than their respective predecessors, but still far slower than PaCiFier. The final four methods [20], [74], [62], [63] in the table are entirely disappointing – $87$ to $572$ times slower than PaCiFier while consuming substantially more resources.

### 2.5.7 Parallelization Efficiency

We now examine how PaCiFier scales with the number of cores, which indicates how well the algorithm benefits from additional CPU resources. As discussed earlier in section 2.5.3, PaCiFier's parallelization framework partitions the computation (i.e., triples

Table 2.12: Runtime (Seconds)

| Graph | RAM (MB) | MGT | PDTL | PaCiFier |
|---|---|---|---|---|
| Twitter | 8,192 | 2,238 | 323 | 63.3 |
| | 4,096 | 2,248 | 327 | 63.2 |
| | 2,048 | 2,260 | 327 | 61.9 |
| | 1,024 | 2,285 | 347 | 61.0 |
| | 512 | 2,354 | 464 | 61.4 |
| | 256 | 2,487 | 1,003 | 67.2 |
| IRL-domain | 8,192 | 5,947 | 849 | 148 |
| | 4,096 | 5,976 | 851 | 144 |
| | 2,048 | 6,020 | 853 | 143 |
| | 1,024 | 6,090 | 898 | 143 |
| | 512 | 6,252 | 995 | 145 |
| | 256 | 6,540 | 1,484 | 149 |

$(i, X, Y')$ from the companion file) into equal-sized jobs, which are processed lock-free by worker threads. As shown in Fig. 2.9, PaCiFier's runtime indeed scales almost linearly. The reason for a slightly suboptimal outcome is that certain auxiliary operations (e.g., indexing of $G^r_\theta(l)$ in Line 4 of Algorithm 5) are executed sequentially.

### 2.5.8 Effect of RAM: Bottlenecked by CPU

Next, we analyze the performance of each algorithm under varying RAM size. We showed earlier that PaCiFier's CPU cost was constant for all $M$. While the I/O complexity does increase as $M \to 0$, double buffering and prefetching can keep this overhead negligible until the disk becomes a bottleneck. Table 2.12 supports this discussion – using our RAID system, PaCiFier completes in virtually the same amount of time for all $M$ in the range between $256$ MB and $8$ GB. The initial drop in runtime can be explained by smaller lookup tables and better cache locality; however, as $M$ decreases further, SIMD becomes less efficient and this effect is reversed. While MGT is not bottlenecked by I/O either, PDTL increases its runtime by $49 - 116\%$ at $M = 256$ MB. More interesting cases where the disk can no longer keep up with the computation are studied next.

### 2.5.9 Effect of RAM: Bottlenecked by I/O

For comparison of disk activity, we use the exact model $m\lceil m/M \rceil$ for MGT/PDTL and compute the size of all companion files in PaCiFier by running Algorithm 4. Although DGP/RGP share the same $\Theta(m^2/M)$ asymptotic cost with MGT, these methods require two orders of magnitude more I/O due to slow convergence, which we omit from analysis. Instead, we contrast against MapReduce methods. The first one is GP [74], which uses at least $\rho = \lceil 3\sqrt{m/M} \rceil$ reducers and shuffles

$$\frac{30(\rho-1)(\rho-2)m}{\rho} \tag{2.19}$$

bytes of data [62]. A later method called TTP [62] reduces $\rho$ by a factor of $\sqrt{3}$ and improves the shuffle to $20(\rho-1)m$.

Table 3.2 shows the I/O in bytes on the two largest graphs under consideration. PaCiFier starts off beating GP/TTP by a factor of $32-78$ and MGT/PDTL by a factor of $3.7-9$. This advantage keeps accumulating as $M$ decreases. Eventually, PaCiFier develops a $58-195\times$ lead over the former and $34-64\times$ over the latter as $M$ reaches $256$ MB. In the last scenario, the I/O phase of MGT/PDTL would require $34.5$ hours to finish ClueWeb using our $1$ GB/s RAID. With a magnetic hard drive (i.e., $100$ MB/s read speed), this would take over two weeks. On the other hand, PaCiFier lowers these numbers to $32$ minutes and $5.3$ hours, respectively.

## 2.6 Conclusion

The chapter created a taxonomy of $18$ triangle-listing methods using a unifying framework called Generalized Iterators (GI), developed a new set of algorithms called Pruned Companion Files (PCF) for external-memory operation of GI, and showed that it possessed better complexity than current implementations in the field. It then determined which of

Table 2.13: I/O Comparison

| Graph | RAM (MB) | GP | TTP | MGT/PDTL | PaCiFier |
|---|---|---|---|---|---|
| Yahoo (in GB) | 8,192 | 2,099 | 1,066 | 88.8 | 40.4 |
| | 4,096 | 3,271 | 1,599 | 177.6 | 47.6 |
| | 2,048 | 5,247 | 2,132 | 355.1 | 55.5 |
| | 1,024 | 7,632 | 3,198 | 710.2 | 64.8 |
| | 512 | 11,219 | 4,531 | 1,420 | 74.6 |
| | 256 | 16,408 | 6,663 | 2,841 | 84.4 |
| ClueWeb (in TB) | 8,192 | 47.4 | 19.2 | 3.91 | 0.69 |
| | 4,096 | 68.4 | 27.9 | 7.82 | 0.87 |
| | 2,048 | 99.8 | 40.2 | 15.6 | 1.10 |
| | 1,024 | 141.7 | 55.9 | 31.3 | 1.36 |
| | 512 | 204.6 | 80.4 | 62.6 | 1.64 |
| | 256 | 291.1 | 113.6 | 125 | 1.93 |

the $18$ methods was the most efficient when both CPU and I/O objectives were taken into account and created a working solution that exhibited $5 - 10\times$ smaller runtime and orders of magnitude less I/O compared to the best previous technique.

# 3. TRIGON*

## 3.1 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 [5], [6], [7], [11], [20], [28], [55], [77], [78], [82], [88], including areas outside of computer science [31], [37], [43], [51], [53], [76], [81]. 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 finds 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 [3], [15], [17], [29], [38], [44], [48], [64], [67], [69], [71], [74] can be expressed under the umbrella of 18 vertex/edge-iterator algorithms [21], [84], where a single method $\mathrm{E}_1$ 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 [60] and the PCF framework from [21]. 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 chapter is to address these questions.

### 3.1.1 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.

This analysis shows how the scaling rate of average degree, memory size, and variance of out-degree affect which method is better. 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 $n \to \infty$, our results demonstrate that under the best scenario for PCF, it beats Pagh by a factor of $n$. In the worst case, it looses by a factor of $\sqrt{n}$. We also prove existence of graphs where PCF scales I/O no faster than Pagh for *all* memory sizes and vice versa.

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 indi-

vidual 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 [29], [38], are 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.

We implement Trigon in C++ and show its performance in real systems. In one configuration, we use a single computer, restricted to $1.9$ MB of RAM, and one magnetic hard drive. Given a $37$-GB complete graph, which is the best-case for Pagh, Trigon finds $167$ trillion triangles in $4$ hours. In contrast, PCF requires $31$ TB of I/O and an estimated $2.3$ days, while Pagh takes roughly $11$ hours.

### 3.2 Related Work

The issue of optimal speed for in-memory algorithms appears to be settled. In the last decade, the fastest techniques have come from the family of vertex/edge iterators [3], [29], [38], [44], [48], [67], [69], [71], [74]. The former methods typically rely on hash tables to perform neighbor checks, where the speed is limited by that of random memory access. On the other hand, scanning edge iterators can be implemented using vectorized CPU intrinsics, which are not bottlenecked by RAM latency. With $128$-bit SIMD and list compression, it is feasible to achieve two orders of magnitude faster neighbor verification [21]. In the taxonomy of $18$ vertex/edge iterators, method $\mathrm{E}_1$ [3], [29], [71] is by far the best technique [21], [84].

In external memory, early methods used a variety of techniques, including disk seeking [23], [52], MapReduce [20], [62], [74], general graph libraries [30], [47], and iterative

graph shrinkage [17], which are difficult to summarize here analytically. Due to their low efficiency, however, these approaches are not considered competitive today. In more recent development, algorithms have been streamlined to access the disk only sequentially, allowing comparison using just the amount of edges read from disk and RAM size $M$. In MGT [38], the graph is split into equal-size chunks. After each is loaded into RAM, the graph is scanned again to discover the missing edges that complete triangles with the portion already in RAM. Ignoring small terms, MGT reads $m^2/M$ edges.

This result was superseded by a method we call Pagh [60], which achieves a strictly better asymptotic bound $O(m^{1.5}/\sqrt{M})$. We review its operation in more detail below. A different approach is proposed in [21], where a set of six PCF algorithms covers all $18$ vertex/edge iterators in external memory. While there is no model for PCF I/O, upper bounds show that it is currently the only method that can achieve linear complexity under constant $M$.

### 3.3 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, whose complexity is normally a quadratic function of undirected degree. This overhead can be substantially reduced by performing an acyclic orientation on $G$, which makes cost depend on the much-smaller *directed* degree. In recent literature [84], orientation is modeled as some permutation $\theta$ 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 chapter, we use orientation $\theta_D$ that arranges the nodes in *descending*

**Algorithm 7:** Method $E_1$ processing source node $u$ in memory

```
1 foreach v ∈ N_u^+ do ◁ visit all out-neighbors
2     find N_v^+ using a hash table
3     W = Intersect(N_u^+, N_v^+) ◁ intersect two sorted out-lists
4     foreach w ∈ W do report Δ_uvw
```

order of undirected degree $d_u$. This permutation, also known as *largest-first* in graph theory [50], [80], is optimal for both the fastest edge iterator $\mathrm{E}_1$ and its corresponding PCF algorithms in [21], [84]. 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, $\mathrm{E}_1$ searches for all directed triangles $\triangle_{uvw}$, where $u > v > w$. This is done by calling Algorithm 7 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 [84]

$$\rho(n) = \sum_{u=1}^{n} \Big( \frac{X_u(X_u - 1)}{2} + X_u Y_u \Big). \tag{3.1}$$

To emphasized the importance of keeping track of lookups, consider the example of Twitter [46]. With $m = 1.2$B edges, Algorithm 7 requires $60$ seconds worth of lookups using one core of an Intel i7. The time to perform $511$B intersections is an additional $250$ seconds. Increasing the lookup cost just $5$ times shifts the bottleneck to the hash table and causes triangle search to increase the runtime from $310$ to $550$ seconds.

When $\mathrm{E}_1$ 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 [38], PCF-1B [21]), keeping two in RAM and streaming the third one (e.g., PCF-1A

[21]), or loading all three simultaneously [60], [64]. Because of the random lookup needed to obtain $N_v^+$, it does not currently appear feasible to stream all three edges.

Note that all methods require the same amount of I/O to store the found triangles. We therefore focus on the cost needed to create this list, which is what differentiates the various approaches.

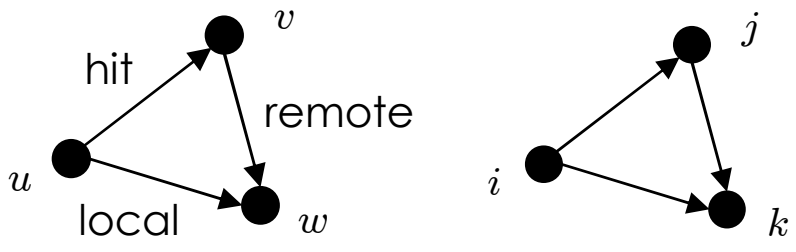
## 3.4 Analysis of Pagh

The original Pagh algorithm [60] 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 $O(.)$ notation [60], analytical comparison between the algorithms, as well as implementation, both require the missing constants. Additionally, since coupling of Pagh to $\mathrm{E}_1$ has not been discussed before, we perform this extension as well.

### 3.4.1 Algorithm

Pagh assigns to each node $u$ a uniformly random color $\phi_u$ drawn from a set $1, 2, \ldots, c$, where $c = \sqrt{m/M}$. Then, all nodes are split into $c$ subsets $V_1, \ldots, V_c$ such that

$$V_i = \{u \in V \mid \phi_u = i\}. \tag{3.2}$$




(a) notation (b) color assignment

Figure 3.1: Directed triangle ($u > v > w$).

**Algorithm 8:** Graph partitioning in Pagh

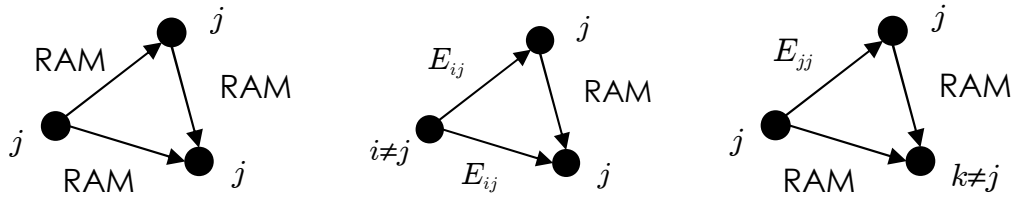
```
for u = 1 to n do
    i = φ_u  ◁ color of source node
    for j = 1 to c do ◁ color of destination nodes
        N⁺_uj = N⁺_u ∩ V_j ◁ out-neighbors of color j
        write (u, N⁺_uj) to subgraph E⁺_ij
```

This can be visualized with the help of Fig. 3.1. Part (a) shows a directed triangle $(uvw)$, as seen by Algorithm 7, with three uniquely-identifiable edges. While they have several different names in previous literature, we follow the notation of [21] for compatibility with $E_1$.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.,

$$E^+_{ij} = \{(u, v) \in E^+_\theta \mid \phi_u = i, \phi_v = j\}. \tag{3.3}$$

This is demonstrated in Algorithm 8, 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 [38] 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 $T_1$ [21], 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.




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

Figure 3.2: Special cases in Pagh.

**Algorithm 9:** 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 ∈ N+_uj do ◁ visit all neighbors in the hist list
6             find remote list N+_vk using the hash table
7             W = Intersect(N+_uk, N+_vk) ◁ local/remote lists
8             foreach w ∈ W do report Δ_uvw
```

### 3.4.2 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 9, which we call Pagh+, loads remote edges $E^+_{jk}$ into RAM and then scans the other two subgraphs. Since Algorithm 8 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 $E_1$. Note that we skipped discussing cases when some of the colors are duplicate; however, our implementation handles them efficiently (i.e., without reading unnecessary files).

**Theorem 4.** *Pagh+ needs $I_P(n) = (2c-1)m$ edges of I/O.*

*Proof.* First notice that Algorithm 9 loads each remote subgraph once, for a total I/O

cost of $m$. The remaining overhead comes from hit/local edges, which we consider next. While there are $c^3$ possible triples $(ijk)$, there are three special cases. The first one is shown in Fig. 3.2(a), where all three edges are in RAM. This results in no additional cost beyond $E^+_{jj}$. The second configuration in Fig. 3.2(b) has file $E^+_{jj}$ loaded in RAM and the remaining color $i$ is not equal to $j$. There are $c(c-1)$ such cases, each requiring $|E^+_{ij}|$ I/O. The last special case in Fig. 3.2(c) involves $c(c-1)$ files $E^+_{jk}$, each producing $|E^+_{jj}|$ I/O. The remaining scenarios are outside the scope of Fig. 3.2. There are $c(c-1)$ files $E^+_{jk}$ such that $j \neq k$, each of which can be coupled with $c-1$ values of $i \neq j$. This yields $c(c-1)(c-1)$ cases that load $2m/c^2$ edges each.

Combining the various terms, we get

$$m + \frac{m}{c^2}(0 + 2c(c-1) + 2c(c-1)(c-1)), \tag{3.4}$$

which simplifies to $2cm - m = (2c-1)m$. □

Since $(2c-1)m = 2m^{1.5}/\sqrt{M} - m$, Pagh+ has the best multiplicative constant in the literature. The closest alternative [64] 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 drawback to this approach is usage of undirected graphs, where $\mathrm{E}_1$ has to perform unnecessary intersections [84].

It is also simple to obtain the number of hash-table lookups in Pagh+. When they become a CPU bottleneck, $\mathrm{E}_1$ may essentially deteriorate into $\mathrm{T}_1$ and lose its advantages. The next result shows that this value is linear in $c$.

**Theorem 5.** *Pagh+ performs $\gamma_P(n) = cm$ lookups.*

*Proof.* Denote by $X_{uj} = |N^+_{uj}|$ the out-degree of $u$ with respect to neighbors of color $j$.

Now, notice that the size of hit lists processed by Algorithm 9 for all $(j, k)$ equals

$$\sum_{k=1}^{c}\sum_{j=1}^{c}\sum_{i=1}^{c}\Big(\sum_{u=1}^{n}\mathbf{1}_{\phi_u=i}\,X_{uj}\Big), \tag{3.5}$$

where $\mathbf{1}_A$ is an indicator of event $A$. Swapping the order of summations, this becomes

$$\begin{aligned}\sum_{k=1}^{c}\sum_{j=1}^{c}\sum_{u=1}^{n}\Big(\sum_{i=1}^{c}\mathbf{1}_{\phi_u=i}\,X_{uj}\Big) &= \sum_{k=1}^{c}\sum_{j=1}^{c}\sum_{u=1}^{n}X_{uj}\\ &= \sum_{k=1}^{c}\sum_{u=1}^{n}X_u. \end{aligned}\tag{3.6}$$

Leveraging the fact that $\sum_{u=1}^{n}X_u = m$, we get the statement of the theorem. □

### 3.4.3 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 8 produces $c$ partitions of size $m/c \gg M$. In fact, two of the three subgraphs involving color $k$ have size $m/c$. Pagh+ cannot be applied here, but MGT can be modified to handle any triple $(ijk)$ with I/O complexity $2(m/c)^2/M = 2m$. Repeating this $c^2$ times for all $(ij)$ produces a total of $2m^2/M$. Depending on $m$ and $M$, this result can be significantly worse than in Theorem 4.

Pagh [60] 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 [60], we get the usual $O(m^{1.5}/\sqrt{M})$;

**Algorithm 10:** PCF-1A graph partitioning

```
1 for u = 1 to n do ◁ iterate over all nodes
2     for i = 1 to p do ◁ go through each partition
3         H_ui = N_u^+ ∩ [a_{i+1}, n]  ◁ pruned hit list
4         L_ui = N_u^+ ∩ [a_i, a_{i+1})  ◁ local/remote list
5         if L_ui ≠ ∅ then
6             write (u, L_ui) to G_θ^r(i)  ◁ remote file i
7             if H_ui ≠ ∅ then
8                 write (u, H_ui) to G_θ^c(i)  ◁ companion file i
```

however, the hidden constants may be non-negligible. But more importantly, the CPU cost for sorting the graph $c$ times may be quite hefty.

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. However, when $c^3$ is large, Pagh has to read many small files and its I/O speed may be adversely affected by disk seeking.

### 3.5 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.

#### 3.5.1 Operation

PCF [21] 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. 3.1(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

**Algorithm 11:** PCF-1B graph partitioning

```
for u = 1 to n do ◁ iterate over all nodes
    for i = 1 to φ_u − 1 do ◁ go through each partition below u
        H_ui = N_u^+ ∩ [a_i, a_{i+1})  ◁ hit list
        L_ui = N_u^+ ∩ [1, a_{i+1})  ◁ local list
        if H_ui ≠ ∅ and |L_ui| ≥ 2 then
            write (u, L_ui) to G_θ^c(i)  ◁ save to companion file i
    if N_u^+ ≠ ∅ then ◁ out-degree non-zero?
        write (u, N_u^+) to G_θ^r(φ_u)  ◁ remote file of u's color
```

edges.

Method $\mathrm{E}_1$ 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 $V_1, \ldots, V_p$. 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 chapter, 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 $G_\theta^r(1), \ldots, G_\theta^r(p)$ 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 [21] 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 10-11.

### 3.5.2 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

$$R_{us} = |\{\phi_{ut} \mid t < s, \phi_{ut} \neq \phi_{us}\}|, \tag{3.7}$$

$$R'_{us} = |\{\phi_{ut} \mid t > s, \phi_{ut} \neq \phi_u\}|. \tag{3.8}$$

With this in mind, consider the next result.

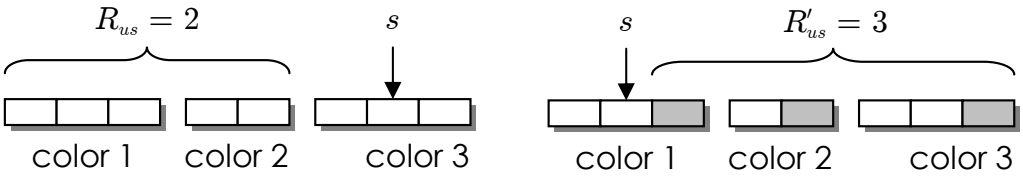
**Theorem 6.** *The companion I/O of PCF-1A is given by*

$$I_A(n) = \sum_{u=1}^{n} \sum_{s=1}^{X_u} R_{us} \tag{3.9}$$

*and that of PCF-1B by*

$$I_B(n) = \sum_{u=1}^{n} \Big[ R'_{u1} + \sum_{s=1}^{X_u} R'_{us} \Big]. \tag{3.10}$$

*Proof.* In PCF-1A, consider some source node $u$ and color $i$. As long as the local list $L_{ui} = N_u^+ \cap V_i \neq \emptyset$, all out-neighbors with labels at least $a_{i+1}$ are saved to disk in Algorithm 10. Therefore, from a perspective of some fixed position $s \in [1, X_u]$ in the out-list $N_u^+$, the number of times this node is written to disk equals the number of non-empty local lists in positions $[1, s-1]$, excluding those that contain $s$. An example is shown in Fig. 3.3(a), where $s$ is written twice. This happens to be the number of distinct colors, except $\phi_{us}$, among the nodes preceding $s$, which equals $R_{us}$ in (3.7). Taking a summation




(a) PCF-A (b) PCF-B

Figure 3.3: Colors among $N_u^+$ in PCF.

over all $u$ and $s$ yields (3.9).

For PCF-1B, we first have to remove neighbors of color $\phi_u$ from consideration since these edges are found in RAM (i.e., included in the remote graph). Once this is done, notice that Algorithm 11 writes a node in position $s$ into $R'_{us}$ files as part of some local list. Fig. 3.3(b) shows one such example. However, there is one exception for $s \geq 2$. The *last* node of each color (within $u$'s neighbor list) has overhead $R'_{us} + 1$, where the extra 1 accounts for $s$ being included in the hit list of companion file $G'^{r}_{\theta}(\phi_{us})$. The affected neighbors are shown in Fig. 3.3 using shading. Putting the pieces together,

$$I_B(n) = \sum_{u=1}^{n} \Big[ R'_{u1} + \sum_{s=2}^{X_u} \Big( R'_{us} + \mathbf{1}_{\phi_{u,s+1} \neq \phi_{us}} \Big) \Big], \tag{3.11}$$

where condition $\phi_{u,X_u+1} \neq \phi_{u,X_u}$ is always true (i.e., we always count an extra 1 for the very last node in $N_u^+$). Rearranging the terms, we get

$$I_B(n) = \sum_{u=1}^{n} \Big[ \sum_{s=1}^{X_u} R'_{us} + \sum_{s=2}^{X_u} \mathbf{1}_{\phi_{u,s+1} \neq \phi_{us}} \Big]. \tag{3.12}$$

Now notice that the sum of indicator variables yields the number of unique colors in positions $[2, X_u]$. Since this value is $R'_{u1}$, we obtain (3.10). □

Note that (3.9)-(3.10) 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 [21]. 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 7.** *The number of hash-table hits in PCF-1A is*

$$\gamma_A(n) = I_A(n) + m - \sum_{u=1}^{n} R_u, \tag{3.13}$$

*and that in PCF-1B is*

$$\gamma_B(n) = m - n. \tag{3.14}$$

*Proof.* PCF-1A writes only pruned hit-lists, which produce $I_A(n)$ lookups when they are loaded back to RAM. Additionally, a portion of each hit list is removed by Algorithm 10 and kept in RAM as part of the local list $L_{ui}$. In fact, the entire $L_{ui}$, except its first node, is part of the hit list for node $u$. Adding the two terms together yields (3.13).

For PCF-1B, every node in $N_u^+$ is part of the hit list, except the one in position $s = 1$. Writing

$$\gamma_B(n) = \sum_{u=1}^{n} (X_u - 1), \tag{3.15}$$

we immediately get (3.14). □

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.

### 3.5.3 Bounds

Computing the exact I/O formula (3.10) 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 8.** *For a given out-degree sequence $\{X_u\}$, the expected size of companion I/O in PCF-1B is bounded by*

$$E[I_B(n)] \leq \sum_{u=a_2}^{n} \zeta_u \Big[ X_u - \zeta_u + 1 + (\zeta_u - 2) q_u^{X_u - 1} \Big], \tag{3.16}$$

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

*Proof.* Note that uniformly random, rather than sequential, color assignment can only make $R'_{us}$ stochastically larger. Therefore, replacing $R'_{us}$ with some other variable $Q_{us}$ that uniformly draws from among $\zeta_u$ colors can yield only larger I/O in expectation. Since $Q_{us} = 0$ for $u < a_2$, we get

$$E[I_B(n)] \leq \sum_{u=a_2}^{n} \Big( E[Q_{u1}] + \sum_{s=1}^{X_u} E[Q_{us}] \Big). \tag{3.17}$$

To expand this, continue assuming random color choices and define

$$W_{usi} = \sum_{t=s+1}^{X_u} \mathbf{1}_{\phi_{ut}=i} \tag{3.18}$$

to be the number of $u$'s out-neighbors to the right of $s$ that have color $i$. Conditioning on the out-degree sequence, each $W_{usi}$ is Binomial$(X_u - s, 1/\zeta_u)$, where $E[W_{usi}] = X_u/\zeta_u$ and $P(W_{usi} \geq 1) = 1 - (1 - 1/\zeta_u)^{X_u - s}$. Then,

$$Q_{us} = \sum_{i=1}^{\zeta_u} \mathbf{1}_{W_{usi} \geq 1} \tag{3.19}$$

is the number of uniform colors to the right of $s$. Setting $q_u = 1 - 1/\zeta_u$, we get

$$E[Q_{us}] = \zeta_u P(W_{usi} \geq 1) = \zeta_u (1 - q_u^{X_u - s}). \tag{3.20}$$

Next, observe that

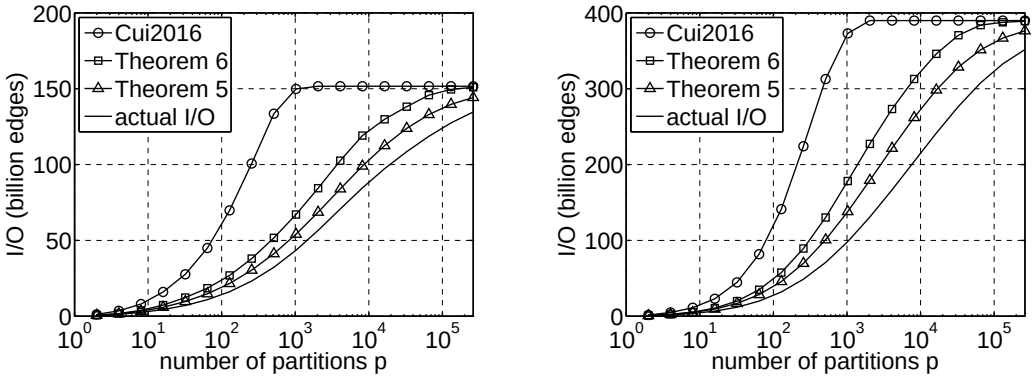
$$\begin{aligned}
\sum_{s=1}^{X_u} E[Q_{us}] &= \zeta_u \sum_{s=1}^{X_u} (1 - q_u^{X_u - s}) = \zeta_u \Big( X_u - \sum_{s=0}^{X_u - 1} q_u^s \Big) \\
&= \zeta_u \Big( X_u - \frac{1 - q_u^{X_u}}{1 - q_u} \Big) \\
&= \zeta_u \Big[ X_u - \zeta_u (1 - q_u^{X_u}) \Big].
\end{aligned} \tag{3.21}$$

Adding $E[Q_{u1}]$ to the last result, we get

$$\begin{aligned}
E[I_B(n)] &\leq \sum_{u=a_2}^{n} \zeta_u \Big[ X_u - \zeta_u + \zeta_u q_u^{X_u} + 1 - q_u^{X_u - 1} \Big] \\
&\leq \sum_{u=a_2}^{n} \zeta_u \Big[ X_u - \zeta_u + 1 + (\zeta_u - 2) q_u^{X_u - 1} \Big],
\end{aligned} \tag{3.22}$$

where we use the fact that $\zeta_u q_u = (\zeta_u - 1)$. □

Bound (3.16) should hold in most situations, but 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




(a) Twitter ($m = 1.2$B) (b) IRL-domain ($m = 1.7$B)

Figure 3.4: Model accuracy in PCF-1B.

moment of out-degree.

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

$$I_B(n) \leq \sum_{u=1}^{n} \min\Big(\frac{(X_u - 1)(X_u + 2)}{2}, X_u\zeta_u\Big). \tag{3.23}$$

*Proof.* Trivially, $R'_{us} \leq \min(X_u - j, \zeta_u)$. Thus, we get

$$\begin{aligned} I_B(n) &\leq \sum_{u=1}^{n} \Big[\min(X_u - 1, \zeta_u) + \sum_{s=1}^{X_u} \min(X_u - s, \zeta_u)\Big] \\ &= \sum_{u=1}^{n} \Big[\min(X_u - 1, \zeta_u) + \sum_{j=1}^{X_u - 1} \min(s, \zeta_u)\Big] \\ &\leq \sum_{u=1}^{n} \min\Big(X_u - 1 + \sum_{s=1}^{X_u - 1} s, X_u\zeta_u\Big), \end{aligned} \tag{3.24}$$

which becomes (3.23) after expanding the inner sum. □

Note that [21] also obtains an upper-bound on $I_B(n)$; however, they neglect the standalone term $R'_{u1}$ in (3.10). This issue notwithstanding, their bound is a special case of

(3.23) where $\zeta_u = \phi_u - 1$ is replaced by $p - 1$. Fig. 3.4 shows a comparison between that result and our models, where we use Twitter from [46] and IRL-domain from the authors of [21].

### 3.5.4 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 $n \to \infty$. 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 $\Delta_{uwv}$ 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. This effect is most pronounced on graphs with heavy-tailed degree. Second, due to sequential grouping of nodes into each color, splitting of neighbor lists in Algorithm 11 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. Third, placing similar node IDs into individual partitions allows better compression of neighbor lists. This can save up to $50\%$ on byte I/O. Similarly, [21] shows that SIMD intersection is $80\%$ faster on compressed lists.

## 3.6 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 $n \to \infty$, which should provide a qualitative assessment of each method's capabilities and types of graphs they are most suited for.

### 3.6.1 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*

$$\omega(f) = \lim_{n\to\infty} \log_n f(n), \tag{3.25}$$

*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

$$\omega(I_P) = \frac{3(1 + a) - r}{2}. \tag{3.26}$$

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

### 3.6.2 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

$$\pi_n = \sum_{u=1}^{n} X_u^2 \tag{3.27}$$

consider the next result.

**Theorem 10.** *The scaling rate of* (3.27) *is* $\omega(\pi) = 1 + 2a + \epsilon$*, where* $\epsilon \in [0, (1-a)/2]$.

*Proof.* Suppose $\pi_n \sim n^{1+2a+\epsilon_n}$, where $\epsilon_n$ is some unknown function. Our goal is to put bounds on it. Assuming $E[X_u] = m/n$ is fixed, it is obvious that minimizing the variance of set $X_1, \ldots, X_n$ yields the lowest $\pi_n$. Since this is achieved by constant $X_u = m/n$, we get

$$\pi_n \geq n\frac{m^2}{n^2} \sim n^{1+2a}. \tag{3.28}$$

This shows that $\epsilon_n \geq 0$ must hold. To arrive at the upper bound on $\pi_n$, first notice that $X_u$ cannot exceed the number of nodes preceding it (i.e., $u-1$). At the same time, $X_u$ must be no larger than $2m/u$; otherwise, the degree sum $\sum_{v=1}^{u} d_v$ of the largest $u$ nodes would exceed $2m$, which is impossible. As a result,

$$\begin{aligned}\pi_n &\leq \sum_{u=1}^{n} \min\Big(u-1, \frac{2m}{u}\Big)^2 \leq \sum_{u=1}^{\sqrt{2m}} u^2 + \sum_{u=\sqrt{2m}}^{n} \frac{(2m)^2}{u^2} \\ &\sim \frac{(2m)^{1.5}}{3} + 4m^2\Big(\frac{1}{\sqrt{2m}} - \frac{1}{n}\Big) \sim n^{3(1+a)/2}.\end{aligned} \tag{3.29}$$

Since we assumed that $\pi_n \sim n^{1+2a+\epsilon_n}$, we get that $\epsilon_n \leq (1-a)/2$. Letting $\epsilon_n \to \epsilon$ as $n \to \infty$, the statement of the theorem follows. □

Note that regular graphs (i.e., all degree equal to each other) yield $\epsilon = 0$ for all $a$. Another well-known case follows from [84]. 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 $n \to \infty$, these graphs also achieve $\epsilon = 0$. For more general cases, the family of dense-core graphs introduced next allows realization of any $\epsilon$.

**Theorem 11.** *For any $\epsilon \in [0, (1-a)/2]$, there exists a graph with $\omega(\pi) = 1 + 2a + \epsilon$.*

*Proof.* Assume a graph where the first $k_n$ nodes, each with degree $l_n \leq k_n$, link to nodes with labels $(1, 2, \ldots, l_n)$. All remaining nodes have degree two and link to nodes $(1, 2)$. Then, assuming $k_n \sim n^{z_1}$ and $l_n \sim n^{z_2}$, where $z_1 \geq z_2$ and $z_1 + z_2 \geq 1$, we get

$$E[D_n] = \frac{k_n l_n + 4(n - k_n)}{n} \sim n^{z_1 + z_2 - 1} \tag{3.30}$$

and

$$\pi_n \sim \sum_{u=1}^{l_n} u^2 + \sum_{u=l_n}^{k_n} l_n^2 + n - k_n \sim k_n l_n^2 \sim n^{z_1 + 2z_2}. \tag{3.31}$$

Assume $a$ is selected first and $\epsilon$ is selected second in the range $[0, (1-a)/2]$. Then, we can construct the system above using $z_1 = 1 - \epsilon$ and $z_2 = a + 1 - z_1$. Note that $z_1 + z_2 = a + 1 \geq 1$ is satisfied with any $a \geq 0$. Furthermore, condition $z_1 \geq z_2$ is equivalent to $2z_1 \geq a + 1$, or $\epsilon \leq (1-a)/2$, which is satisfied by any valid $\epsilon$. □

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

**Theorem 12.** *The rate of PCF-1B I/O is upper-bounded by*

$$\omega(I_B) \leq \min(1 + 2a + \epsilon, 2 + 2a - r). \tag{3.32}$$

*Furthermore, in the worst-case of* $\epsilon = (1 - a)/2$*, the graphs built in Theorem 11 reach* (3.32) *for all* $a$ *and* $r$*.*

*Proof.* From (3.23), it is clear that

$$I_B(n) \leq \min(\pi_n, (p-1)m) \leq \min\Big(\pi_n, \frac{m^2}{M}\Big). \tag{3.33}$$

Converting this into rates yields (3.32).

Next, for any $a$ and $\epsilon = (1 - a)/2$, the graphs introduced in Theorem 11 require $z_1 = z_2 = (1 + a)/2$. Then, we can set $k_n = 2l_n$ and obtain that out-lists of source nodes $u \in [l_n, 2l_n]$ have $X_u = l_n$ neighbors and roughly $\zeta_u = l_n^2/M$ colors. Converting this into asymptotics, it follows that the out-degree of these nodes scales as $z_2$ and the number of colors as $2z_2 - r$. Therefore, when $z_2 < 2z_2 - r$, or equivalently $r < 1 - \epsilon = (1 + a)/2$, PCF-1B has the same asymptotics as $\pi_n$. This makes $\omega(I_B) = 1 + 2a + \epsilon$. Otherwise, $I_B(n)$ scales as $k_n l_n \zeta_u \sim n^{2+2a-r}$. Both cases and the condition to switch between them are exactly the same as in (3.32). □

The graphs from Theorem 11 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 13.** *In bipartite graphs, PCF-1B has I/O overhead* $I_B(n) = m$ *for all* $a$ *and* $r$*, i.e.,* $\omega(I_B) = 1 + a$*.*

*Proof.* Suppose the nodes are divided into two sections, which we call $S_1$ and $S_2$, of size $k_n$ and $n - k_n$, respectively. Each node in $S_1$ connects to all nodes in $S_2$. We assume

that $k_n < n/2$, i.e., the first section is smaller, and $k_n \sim n^a$. Notice that this graph has an average degree proportional to $n^a$ and that none of the nodes in $S_1$ have any out-neighbors in $G_\theta^+$. Therefore, PCF-1B assigns them the same color 1. It then follows that the companion I/O from all nodes $u \in S_2$ is no more than $m$ since each out-neighbor list $N_u^+$ has nodes of one color and thus can only produce output to one companion file $G_\theta^c(1)$. Furthermore, this result holds for all $M_n$. □

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 13 reveals a non-trivial lower bound.

### 3.6.3 Analysis

We summarize the findings of this section using Fig. 3.5(a). The $x$-axis shows rate $r$ at which RAM increases as $n \to \infty$. 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 (3.26). 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 11. The lower boundary, shown by line $ZT$, is the bipartite graph from Theorem 13.

At $r = 0$, i.e., constant RAM, Pagh begins in point $X$ that is always no lower than PCF-1B's worst initial point $U$. This happens because $1.5 + 1.5a \geq 1 + 2a + \epsilon$ for all $\epsilon \leq (1 - a)/2$. 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, regardless of the actual $G$. Between points $W$ and $T$, some of the graphs are solved quicker by Pagh and others by PCF-1B.

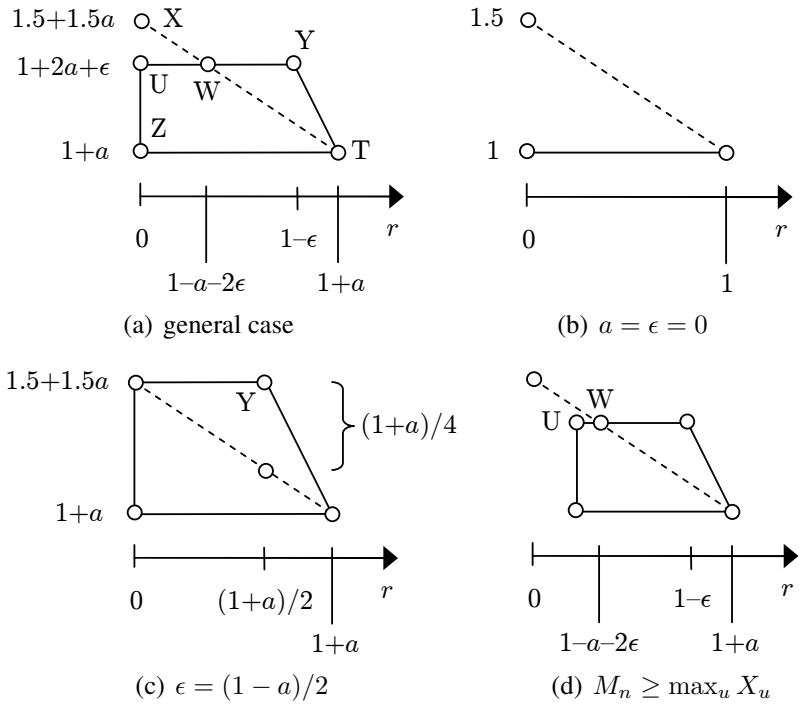
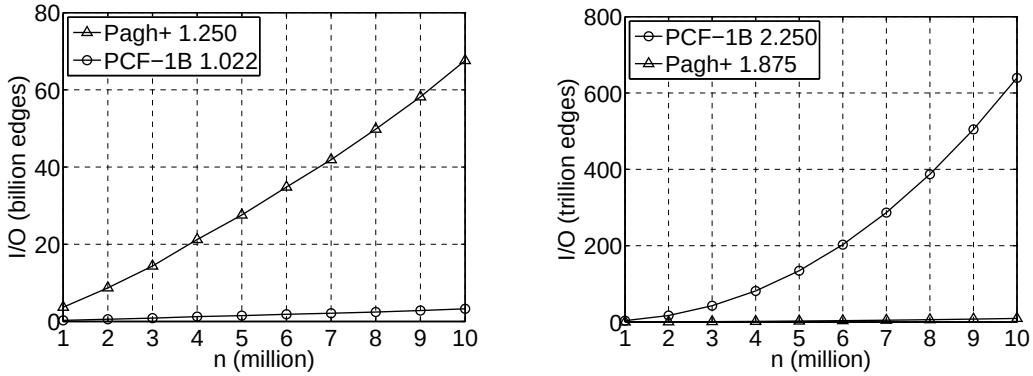



Figure 3.5: Comparison of scaling rates.

It can be seen from the figure that the largest gap between the two methods occurs at $r = 0$, where PCF-1B in point $Z$ beats 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. 3.5(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 11, 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




(a) $a = \epsilon = 0, r = 1/2$ (b) $a = 1/2, \epsilon = 1/4, r = 3/4$

Figure 3.6: Actual I/O with curve-fitted scaling rates.

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. 3.5(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. 3.6(a) shows the actual I/O of the two methods in a random graph with Pareto degree, where with shape $\alpha = 1.5$ and average degree is $30$. As predicted by our analysis and Fig 3.5(b), the asymptotic gap between the methods is $n^{1/4}$. Continuing to Fig. 3.6(b), we examine a dense-core graph from Theorem 11 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.

### 3.6.4 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.

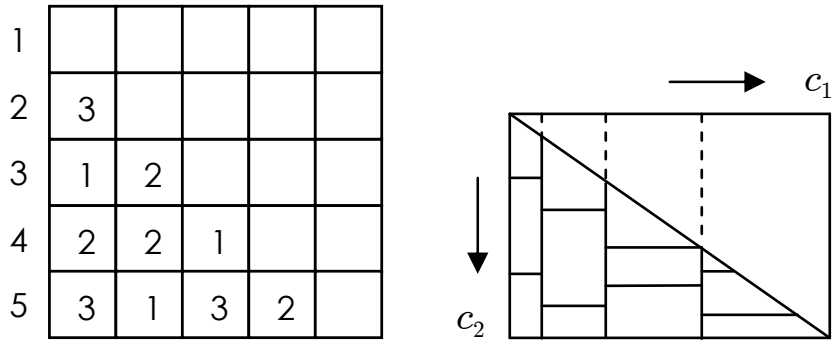
On the other hand, the main downside of PCF lies in one-dimensional color partitioning. This creates a large number of colors $p$ and causes unnecessary duplication of effort. Usage of 2D coloring could help reduce the number of files into which the out-neighbors must be written. This can be seen in (3.10), where making $R'_{us}$ pick out of $\sqrt{p}$ colors, rather than $p$, would be a noticeable improvement.

## 3.7 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.

### 3.7.1 Generalized Coloring

All vertex/edge iterators [21] 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




(a) general case (b) contiguous case

Figure 3.7: Heterogenous 2D partitioning of remote edges.

would allow the in-memory search to function properly. The framework developed below applies to all $18$ methods from [21]; however, to keep the notation to a minimum, we only describe how it works with $\mathrm{E}_1$.

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

$$B = \{(u, v) \in \mathbb{N}^2 \mid v < u \leq n\}, \tag{3.34}$$

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 $B_1, \ldots, B_p$ 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. 3.7(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. 3.7(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., $c_1 = 1$.

Once partitions are decided, the edges of $G_\theta^+$ must be separated into sets $E_1^+, \ldots, E_p^+$, where $E_\ell^+ = E_\theta^+ \cap B_\ell$ for $\ell = 1, 2, \ldots, p$ 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. 3.7(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 $\max_u X_u \leq M$.

### 3.7.2 Unified Partitioned Iterator

Assume the edges of $G_\theta^+$ have been separated into individual files. What remains is creation of companion files, which is done in a framework we call *Unified Partitioned Iterator* (UPI). Let $S_\ell = \{u \mid (u, v) \in B_\ell\}$ be the source nodes and $D_\ell = \{v \mid (u, v) \in B_\ell\}$ be the destination nodes in partition $\ell$. For the example in Fig. 3.7(a), $S_3 = \{2, 5\}$. Operation of UPI is summarized in Algorithm 12. For each node $u$ and its out-neighbor $v$, Line 3 finds all partitions $\ell$ where $v$ is a source. Next, recalling Fig. 3.1(a), observe that the local list needs to be customized to include only those neighbors $w$ of $u$ that are possibly neighbors of $v$ in $B_\ell$. This is done in Lines 4-5. If the local list is empty or contains only $v$, then $v$ cannot be $u$'s hit node for partition $\ell$ and the algorithm moves on in Line 6.

Line 7 checks if $u$ itself participates in $B_\ell$ as a source node. If so, the entire local list is already included in $E_\ell^+$, which Line 8 signals by emptying $L_{uv\ell}$. Additionally, it is possible that link $(u, v)$ is also contained in the remote graph, which happens if $v$ is a destination node in $B_\ell$. Line 9 takes care of this condition. Finally, Line 10 saves the triple $(u, v, L_{uv\ell})$ into the companion file, which is done even if $L_{uv\ell}$ was previously emptied in Line 8.

**Algorithm 12:** UPI creating companion files

```
1  for u = 1 to n do
2      foreach v ∈ N_u^+ do ◁ iterate through all out-neighbors
3          foreach partition ℓ where v ∈ S_ℓ do ◁ v is a source in B_ℓ
4              Z_vℓ = {w | (v, w) ∈ B_ℓ}  ◁ take neighbors of v in B_ℓ
5              L_uvℓ = N_u^+ ∩ Z_vℓ  ◁ local list for (u, v) in partition ℓ
6              if L_uvℓ \ {v} = ∅ then continue
7              if u ∈ S_ℓ then ◁ u is also a source in partition ℓ
8                  L_uvℓ = ∅  ◁ all local nodes in E_ℓ^+
9                  if v ∈ D_ℓ then continue ◁ (u, v) already in E_ℓ^+
10             write (u, v, L_uvℓ) to companion graph C_ℓ^+
```

**Algorithm 13:** UPI processing one partition $\ell$

```
1  load E_ℓ^+ = {(v, N_vℓ^+)} in RAM; set up hash table to source nodes
2  while companion file C_ℓ^+ not empty do
3      load (u, v, L_uvℓ) from C_ℓ^+
4      find remote list N_vℓ^+ using the hash table
5      W = Intersect(L_uvℓ, N_vℓ^+) ◁ local/remote lists
6      foreach w ∈ W do report Δ_uvw
```

Triangle search in UPI is shown in Algorithm 13. The only difference from Pagh+ is that each partition $\ell$ has its own companion file, from which nodes $u$, their hit neighbors $v$, and local lists $L_{uv\ell}$ are obtained.

**Theorem 14.** *UPI finds each triangle exactly once and exhibits no more intersection overhead than in-memory $E_1$.*

*Proof.* Because the edges are partitioned into non-overlapping and exhaustive sets, detecting the same triangle multiple times or missing some of them is impossible. This is a consequence of the fact that remote edge $(vw)$ belongs to exactly one partition $\ell$.

We now consider the intersection overhead of Algorithm 12. The local intersection

cost at node $u$ can be written as

$$\sum_{v \in N_u^+} \sum_{\ell=1}^{p} |L_{uv\ell}| = \sum_{v \in N_u^+} \sum_{\ell=1}^{p} |N_u^+ \cap Z_{vl}|. \tag{3.35}$$

Since $\{Z_{v1}, \ldots, Z_{vp}\}$ is a partition of $v$'s possible neighbor options $[1, v-1]$, we get that

$$\begin{aligned} \sum_{v \in N_u^+} \sum_{\ell=1}^{p} |N_u^+ \cap Z_{vl}| &= \sum_{v \in N_u^+} |N_u^+ \cap [1, v-1]| \\ &= \frac{X_u(X_u - 1)}{2}, \end{aligned} \tag{3.36}$$

which is exactly the same as in $\mathrm{E}_1$.

Now suppose $Y_{v\ell}$ is the in-degree of $v$ from hit lists in partition $\ell$ and let $X_{v\ell}$ be its out-degree in the remote graph $E_\ell^+$. Since node $v$ is hit $Y_{v\ell}$ times in $\ell$, each causing a scan over $X_{v\ell}$ neighbors, the remote intersection overhead for $v$ equals

$$\sum_{\ell=1}^{p} X_{v\ell} Y_{v\ell} \leq Y_v \sum_{\ell=1}^{p} X_{vz} = X_v Y_v. \tag{3.37}$$

Combining the upper bound in (3.37) with (3.36), we get the cost of $\mathrm{E}_1$ in (3.1). □

The proof of this theorem shows that intersection cost can actually *reduce* as $p$ increases. This happens because node $v$ participates in remote intersection only when there is a hit-list edge $(u, v)$ in the corresponding companion file. However, if $u$ has no other neighbors smaller than $v$ in partition $\ell$, Line 6 of Algorithm 12 discards $v$ as being ineligible. In practice, cost reduction only affects the $X_u Y_u$ term in (3.1) and happens only in partitioning schemes that break some of the out-lists $N_u^+$ across multiple $E_\ell^+$ (i.e., Pagh and PCF-1A).

### 3.7.3 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 (3.7)-(3.8). 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, the design in Fig. 3.7(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. 3.7(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

$$\sum_{u=a_k}^{a_{k+1}-1} Y_u = \frac{m}{c_1}, \tag{3.38}$$

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.,

$$\sum_{u=b_{kj}}^{b_{k,j+1}-1} |N_u^+ \cap [a_k, a_{k+1})| = M, \tag{3.39}$$

which is similar to PCF-1B. Note that if (3.38) 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, (3.39) 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. 3.7(b) is better.

The Trigon split technique is shown in Algorithm 14, where we continue using color $k$ for node $w$ and color $j$ for $v$ to maintain compatibility with Fig. 3.1(b). 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. 3.1, which explains their removal.

### 3.7.4 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 (3.7) still works for $R_{us}$. Similarly, $R_u$ counts the number of primary colors in $N_u^+$. To handle the

**Algorithm 14:** Trigon writing companion files

```
for u = 1 to n do
    for k = 1 to c_1 do ◁ run thru primary colors
        L_uk = N_u^+ ∩ [a_k, a_{k+1})  ◁ local list for color k
        if L_uk ≠ ∅ then ◁ work to be done?
            for j = 1 to c_2 do ◁ run thru secondary colors
                H_ukj = N_u^+ ∩ [b_kj, b_{k,j+1})  ◁ hit list for pair (kj)
                if H_ukj ≠ ∅ and |L_uk ∪ H_ukj| ≥ 2 then
                    L_uk = L_uk ∩ [1, max(H_ukj)]  ◁ prune
                    if φ_u(k) = j then ◁ local list in RAM
                        write (u, H_ukj \ L_uk) to companion C_kj^+
                    else
                        write (u, H_ukj ∪ L_uk) to C_kj^+
```

vertical dimension, 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, (3.8) is replaced with

$$R''_{us} = |\{\varphi_{ut}(\phi_s) \mid t > s, \varphi_{ut}(\phi_s) \neq \varphi_u(\phi_s)\}|, \tag{3.40}$$

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 15.** *The I/O complexity of Trigon is*

$$I_T(n) \approx \sum_{u=1}^{n} \Big[ R''_{u1} + \sum_{s=1}^{X_u} (R_{us} + R''_{us}) \Big] \tag{3.41}$$

*and the number of hash-table lookups is*

$$\gamma_T(n) = \sum_{u=1}^{n} \sum_{s=1}^{X_u} R_{us} + m - \sum_{u=1}^{n} R_u. \tag{3.42}$$

With the exception of minor terms related to overlapping local/hit lists, the result in (3.41) is exact. To perform a self-check, notice that PCF-1A (i.e., $c_1 = p$) has $R''_{us} = 0$,

which converts (3.41) into (3.9). 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 (3.10). Intuitively speaking, (3.41) can be viewed as a sum of I/O in PCF-1A running with $c_1$ colors and PCF-1B with $c_2$ colors, although this is approximate since $R''_{us}$ does not equal $R'_{us}$ unless $c_1 = 1$. Recalling (3.13), also observe that (3.42) is exactly the number of lookups in PCF-1A under $c_1$ partitions, where more primary colors always cause more CPU cost.

Following the proof of Theorem 9, there exists a simple bound on $I_T(n)$ that shows the impact of each color.

**Theorem 16.** *The Trigon I/O is upper-bounded by*

$$I_T(n) \leq \sum_{u=1}^{n} X_u h(X_u), \tag{3.43}$$

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

The first term of $h(x)$ represents the number of lookups, while the second models the size of local lists streamed from disk. Additionally, since $h(x) \leq c_1 + c_2 - 1$, usage of $c_1 = c_2 = \sqrt{p}$ in (3.43) yields a looser bound

$$I_T(n) \leq \sum_{u=1}^{n} X_u(c_1 + c_2 - 1) = (2\sqrt{p} - 1)m, \tag{3.44}$$

which is the I/O cost of Pagh+ in Theorem 4. Since colors are sequential, Trigon beats (3.44) even in complete graphs, where it comes the closest to this bound, by roughly a factor of 2. It is also clear that (3.42) is upper bounded by $c_1 m$. Recalling Theorem 5, this makes $\gamma_T(n)$ better than the corresponding metric in Pagh+ for all $c_1 \leq \sqrt{p}$.

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

### 3.7.5 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 (3.41) currently appears intractable, we only consider insight that might be gained from the upper-bound (3.43), 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

$$\xi(c) = E[X(\min(X, c) + \min(X, p/c))], \tag{3.45}$$

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

**Theorem 17.** *If $X$ has density $f(x)$,* (3.45) *is minimized by $c = 1$, $c = p$, or any solution to $g(c) = g(p/c)$, where*

$$g(y) = y \int_y^\infty x f(x) dx. \tag{3.46}$$

*Proof.* Suppose $X \sim F(x)$. Then, we can expand the expectation in (3.45) as

$$\begin{aligned}
\xi(c) &= \int_0^\infty x(\min(x, c) + \min(x, p/c)) dF(x) \\
&= \int_0^c x^2 dF(x) + c \int_c^\infty x dF(x) + \int_0^{p/c} x^2 dF(x) \\
&+ \frac{p}{c} \int_{p/c}^\infty x dF(x).
\end{aligned} \tag{3.47}$$

Differentiating with respect to $c$ and applying Leibnitz's integration rule four times,

$$\begin{aligned}\frac{d\xi(c)}{dc} &= c^2 f(c) - c^2 f(c) + \int_c^\infty x f(x) dx - \frac{p^3 f(p/c)}{c^4} \\ &\quad + \frac{p^3 f(p/c)}{c^4} - \frac{p}{c^2} \int_{p/c}^\infty x f(x) dx \\ &= \int_c^\infty x f(x) dx - \frac{p}{c^2} \int_{p/c}^\infty x f(x) dx \\ &= \frac{g(c) - g(p/c)}{c}. \end{aligned} \tag{3.48}$$

Optimal $c$ is either a solution to $g(c) = g(p/c)$ or lies on the boundary, i.e., $c = 1$ or $c = p$. □

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 18.** *If $X < 2\sqrt{p} - 1$ with probability 1, then $c = 1$ or $c = p$ is optimal in* (3.45)*. On the other hand, if $X > 2\sqrt{p} - 1$ with probability 1, then $c = \sqrt{p}$ is optimal.*

*Proof.* Because the objective function is symmetric in $c$, we only need to consider $c \in [1, \sqrt{p}]$. Any optimal solution $c$ has an optimal counterpart $1/c$. Suppose $X \sim F(x)$ is defined in $[1, n-1]$ and rewrite (3.45) as

$$\xi(c) = \int_1^{n-1} x \xi(c, x) dF(x), \tag{3.49}$$

where $\xi(c, x) = \min(x, c) + \min(x, p/c)$. First suppose that $x \leq \sqrt{p}$, in which case $\xi(c, x)$ becomes $\min(x, c) + x$. This is trivially minimized by $c = 1$. Second, suppose $x > \sqrt{p}$, in which case we get $\xi(c, x) = c + \min(x, p/c)$. There are two subcases here – 1) $x < p/c$

yields $c + x$, where $c = 1$ is optimal; and 2) $x \geq p/c$ produces $c + p/c$, where $c = \sqrt{p}$ is best. In the former subcase, the lowest cost is $x + 1$ and in the latter it is $2\sqrt{p}$. Therefore, $c = 1$ is better when $x < 2\sqrt{p} - 1$, worse when $x > 2\sqrt{p} - 1$, and the two are equal otherwise.

As a result, if $X$ is limited to $[1, 2\sqrt{p} - 1]$, we get that (3.49) minimized by $c = 1$. On the other hand, if $X$ is always larger than $2\sqrt{p} - 1$, the integral is minimized by $\sqrt{p}$. □

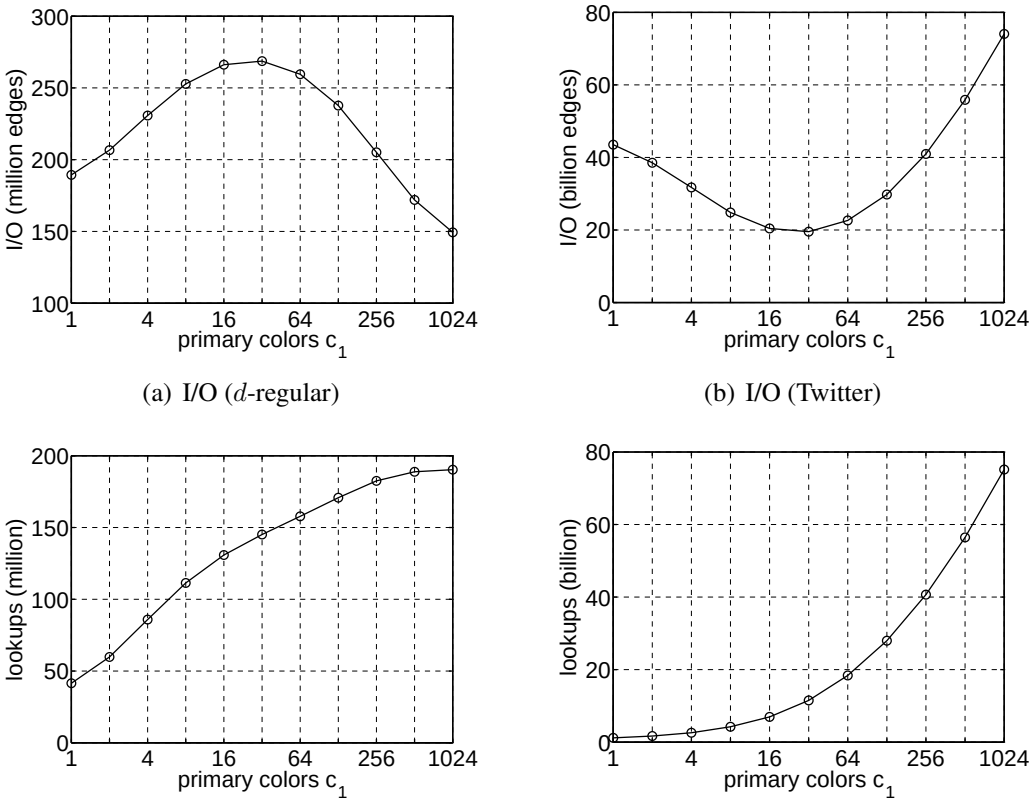
One example that falls under Theorem 18 are $d$-regular graphs. This is illustrated in Fig. 3.8(a) using a random graph with $d = 10$, $n = 10$M, and $p = 1024$. The I/O function of Trigon in this graph is an inverted cup, with the middle being the worst and the two boundaries being the best, which is one of the few cases where PCF-1A wins over PCF-1B. A more common scenario is given by Twitter in Fig. 3.8(b), where $c_1 = \sqrt{p} = 32$ is clearly optimal.

If the program has access to $G_\theta^+$, it can compute our models shown earlier in the chapter 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 19.** *Usage of $c = \sqrt{p}$ in* (3.45) *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 $\sqrt{p}$.*

### 3.7.6 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. 3.8(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.




(a) I/O ($d$-regular)

(b) I/O (Twitter)

(c) lookups ($d$-regular)

(d) lookups (Twitter)

Figure 3.8: Trigon tradeoffs between I/O and lookups ($p = 1024$).

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

$$r(c_1) = \max\left(\frac{I_T(n, c_1)}{S_D}, \frac{\rho(n)}{S_I} + \frac{\gamma_T(n, c_1)}{S_H}\right) \tag{3.50}$$

where $\rho(n)$ is the intersection cost from (3.1).

To obtain $I_T(n, c_1)$ and $\gamma_T(n, c_1)$, one can use (3.41)-(3.42). Direct computation of

these values may be costly; however, approximation (3.43), as well as its refinement using (3.16) or (3.23), work quite well. A binary search over $r(c_1)$ requires efficient computation of the models, i.e., without scanning the degree sequence $\{X_u\}$, which may not fit in RAM. Our approach is to create a short digest of the necessary information during construction of $G^+_\theta$, which summarizes the in/out degree sequences of the graph. Since $\{X_u\}$ typically contains many runs of similar values $(X_u, X_{u+1}, \ldots, X_{u+s-1})$, each of them can be compressed into one entry that keeps track of the count $s$ and the starting value $X_u$. As a result, minimization of (3.50) often takes negligible time.

### 3.8 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.

Table 3.1: Graph Properties

| Graph | Nodes $n$ | Edges $m$ | Size (GB) | Triangles |
|---|---|---|---|---|
| Twitter [46] | 41M | 1.2B | 9.3 | 35B |
| Yahoo [87] | 720M | 6.4B | 53.3 | 86B |
| IRL-domain [21] | 86M | 1.7B | 13.3 | 113B |
| IRL-host [21] | 642M | 6.4B | 52.7 | 437B |
| IRL-IP [21] | 1.6M | 818M | 6.1 | 1040B |
| ClueWeb [21] | 8.2B | 51B | 358 | 879B |
| Complete | 100K | 5.0B | 37.2 | 167T |
| Bipartite | 100K | 2.5B | 18.6 | 0 |

Out of the standard graphs used for triangle listing, we engage the six largest from

Table 3.2: 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 |

[21]. Their characteristics are shown in Table 3.1. In the last two rows, we add into the mixture best-case scenarios from Pagh and PCF.

### 3.8.1 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 experiment 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 3.2 shows that Pagh+ loses to PCF-1B in five out of 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.

### 3.8.2 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 3.3 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., $392$B 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 3.4 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 3.3, Pagh+ is $3-5$ times slower than Trigon. Usage of $10$K partitions for the complete graph creates a noticeable bottleneck in reading $c^3 = 1$M 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., $24\times$ on Yahoo).

Table 3.3: 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 3.4: 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 |

## 3.9 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. Our approach works by trading I/O cost for lookups, which makes the method adaptable to whatever bottlenecks triangle listing may be facing in a particular hardware configuration.

# 4. SHALLOW NEIGHBORHOOD FUNCTION

## 4.1 Introduction

In a given graph $G = (V, E)$, assume each node $v$ has some weight $w_v$ and let $P_d(v)$ be the set of paths of length $d$ that originate from $v$. The end points of the paths in $P_d(v)$ form a multiset $N_d(v)$ and $W_d(v) = \{w_u \mid u \in N_d(v)\}$ is the collection of the corresponding weights. A neighborhood function is a function that applies to the set of weights and paths in the vicinity of $v$ up to some distance $d$, i.e., $f(W_1(v), \ldots, W_d(v), P_1(v), \ldots, P_d(v))$. Note that the neighborhood function can work with duplicates or eliminate them.

In practice, we focus on a neighborhood distance $d \leq 2$ as most applications lie in this distance and the computation complexity grow exponentially for larger $d$. PageRank/TrustRank can be viewed as examples of neighborhood function at $d = 1$ since the score of each node relies on direct neighbors. Computation of supporters and four cycles is a neighborhood function at $d = 2$. Triangle listing requires information at both $d = 1$ and $d = 2$ since a triangle indicates that a node has both a $d = 1$ path and a $d = 2$ path to another node.

In this chapter, we discuss the problem of computing level-2 supporters. Given a directed graph, define $d(i, j)$ as the shortest distance from $j$ to $i$ along *out*-links. Our goal is that for each node $i$, compute the set of nodes whose shortest distance to $i$ is 2:

$$N(i, 2) = \{j \mid d(i, j) = 2\}. \tag{4.1}$$

For example, in Figure 4.1, we have $N(x_1, 2) = \{z_1, z_3\}$, $N(x_2, 2) = \{z_1, z_2, z_3\}$, and $N(x_3, 2) = \{z_1\}$. Note that a node $z$ may reach $x$ through different length-2 paths, i.e., via different middle nodes $y$. This requires us to detect duplicates when computing the unique

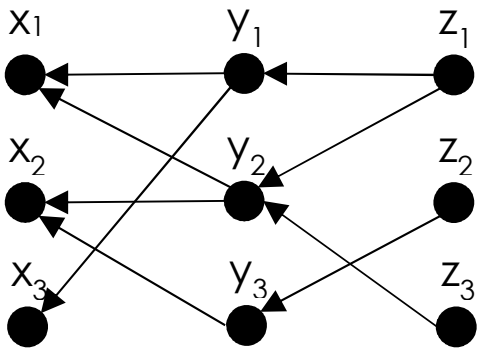



Figure 4.1: Level-2 supporters.

number of level-2 supporters for each $x$. Such duplicate elimination makes this problem more difficult than computing triangles.

The input of this problem is an out-graph $G^+$ and an in-graph $G^-$. An algorithm may choose to utilize both graphs or only one of them. When the graphs do not fit into memory, a graph partitioning scheme can be applied to partition the graphs either by $x$ or $z$. When partitioning by $x$, we compute the full supporter list for a subset of $x$ each time; when partitioning by $z$, we compute a partial supporter list of all $x$ each time. Both partitioning scheme guarantees no redundant counts. Partitioning by $y$ currently seems infeasible as the same $z$ may reach the same $x$ via different $y$ partitions. It thus requires to carry information across multiple partitions for possible duplicate elimination, which may be costly.

### 4.1.1 Counting vs. Listing

Due to high CPU and I/O cost, exact computation of neighborhood function has not been explored before. Previous work, including ANF (Approximate Neighborhood Function), BitVector, and TSE (Top Supporters Estimation), focuses on estimation of neighborhood function. However, these estimation methods only provide an estimated count of neighbors. Thus, they cannot produce a full list of neighbors or accomplish more advanced

**Algorithm 15:** SNF-A

```
1 Function SNF-A (G+, G−)
2     while G− is not finished do
3         load next chunk from G−, invert it into partial out-lists (y, OUT(y))
4         build hash table H that maps each y to OUT(y)
5         while G+ is not finished do
6             load next chunk of out-lists (z, N+(z)) from G+
7             foreach y ∈ N+(z) do
8                 MarkL2-A(z, y, H)  ◁ mark level-2 supporters
9             ClearL1-A(z, H)  ◁ clear level-1 supporters
```

**Algorithm 16:** Mark level-2 supporters

```
1 Function MarkL2-A (z, y, H)
2     locate OUT(y) from H
3     foreach x ∈ OUT(y) do
4         if lastSupporter[x] != z then
5             lastSupporter[x] = z  ◁ mark z as a level-2 supporter of x
6             cnt[x] += 1
```

functionalities beyond counting. Our study focuses the exact computation of neighborhood function, which guarantees us to actually visit every neighbor. Beside accurate counting, we can achieve more functionalities such as computing max/min weight among neighbors, summing up weights of neighbors, etc.

## 4.2 Algorithm

In this section, we introduce our algorithm that computes neighborhood function at depth 2, which we call Shallow Neighborhood Function (SNF). There exist two versions of this algorithm, i.e., SNF-A and SNF-B, that traverse the links in different directions.

### 4.2.1 SNF-A

SNF-A (Algorithm 15-17) partitions the graph by $x$. Assume the node set $V$ is partitioned into $p$ subsets $V_1, V_2, \ldots, V_p$, where $V_i \cap V_j = \emptyset$ $(1 \leq i \neq j \leq p)$ and $\cup_{i=1}^{p} V_i = V$. In the $i$-th iteration, SNF-A loads a chunk of $G^-$, which contains the in-neighbors of

**Algorithm 17:** Clear level-1 supporters

```
1 Function ClearL1-A (z, H)
2     locate OUT(z) from H
3     foreach x ∈ OUT(z) do
4         if lastSupporter[x] == z then
5             cnt[x] -= 1  ◁ z is a level-1 supporter of x, remove its count
```

$x \in V_i$. It then inverts the in-neighbor lists ($x \leftarrow y_1, y_2, \ldots$) into out-neighbor lists ($y \rightarrow x_1, x_2, \ldots$), denoted as $OUT(y)$. Note that $OUT(y)$ only contains the portion of $y$'s out-neighbors that belong to $V_i$, i.e., $OUT(y) = N^+(y) \cap V_i$. A hash table is built to map each $y$ to $OUT(y)$. Then, SNF-A scans $G^+$ from disk to obtain out-lists $(z, N^+(z))$. For each $y \in N^+(z)$, locate $OUT(y)$ via the hash table $H$. It then follows that each $x \in OUT(y)$ can be reached by $z$ in two hops along out-links.

In order to eliminate duplicates, i.e., $z$ may reach $x$ via different intermediate nodes $y$, we need to record the last supporter $z$ that reaches each $x$. Since we only process a small subset of nodes $V_i$ each time, we can keep a supporter count and the last supporter ID of each $x \in V_i$ in memory. After marking all unique $x$ that $z$ can reach in two hops, the last step is to clear the mark when $x$ is a direct out-neighbor of $z$, i.e., $d(x, z) = 1$. After finishing $G^+$, SNF-A has exhausted all possible level-2 supporters $z$ and obtained the final supporter count for each $x \in V_i$.

### 4.2.2 SNF-B

SNF-A works by starting from each $z$, walking along out-links for two hops, and reaching $x$. SNF-B partitions by $z$ and does the reverse by starting from $x$, walking along in-links for two hops, and finding its supporters $z$. SNF-B allows us to focus on a single $x$ each time, which has advantages including: 1) We only need to maintain a single counter, which can be held in register for efficiency; 2) Instead of keeping track of the last supporter ID for each $x$, we now only need to keep 1-bit information of whether each $z$ has been

**Algorithm 18:** SNF-B

```
1  Function SNF-B (G+, G−)
2      while G+ is not finished do
3          load next chunk from G+, invert it into partial in-lists (y, IN(y))
4          build hash table H that maps each y to IN(y)
5          setup a bitmap B of source nodes z ∈ V_i in this chunk
6          while G− is not finished do
7              load next chunk of in-lists (x, N−(x)) from G−
8              c = 0
9              foreach y ∈ N−(x) do
10                 MarkL2-B(y, H, B, c)  ◁ mark level-2 supporters
11             ClearL1-B(x, H, B, c)  ◁ clear level-1 supporters
12             cnt[x] += c
13             foreach y ∈ N−(x) do
14                 ClearL2-B(y, H, B)  ◁ clear level-2 supporters
```

**Algorithm 19:** Mark level-2 supporters

```
1  Function MarkL2-B (y, H, B, c)
2      locate IN(y) from H
3      foreach z ∈ IN(y) do
4          if B[z] == 0 then
5              B[z] = 1  ◁ mark the bit of z
6              c += 1
```

marked as a level-2 supporter. The main CPU bottleneck lies in marking supporters. By using a bit map, SNF-B significantly reduces the size of data structure needed for marking supporters. Depending on the size of each node partition, i.e., $|V_i|$, the bit map may be small enough to fit into CPU's L1 cache, which would significantly boost CPU processing speed.

The detail of SNF-B is illustrated in Algorithms 18-21. Compared to SNF-A, SNF-B reverses the processing order of $G^+$ and $G^-$. It scans $G^+$ in the outer loop. In the $i$-th iteration, SNF-B loads a chunk of $G^+$ that contains out-lists of source nodes $z \in V_i$, i.e., $(z \rightarrow y_1, y_2, \ldots)$. It then inverts it into partial in-lists of each $y$, i.e., $(y \leftarrow z_1, z_2, \ldots)$, denoted as $IN(y)$. Note that $IN(y) = N^-(y) \cap V_i$. Two auxiliary data structures are

**Algorithm 20:** Clear level-1 supporters

```
1 Function ClearL1-B (x, H, B, c)
2     locate IN(x) from H
3     foreach z ∈ IN(x) do
4         if B[z] == 1 then
5             B[z] = 0  ◁ z is a level-1 supporter of x, remove its count
6             c -= 1
```

**Algorithm 21:** Clear level-2 supporters

```
1 Function ClearL2-B (y, H, B)
2     locate IN(x) from H
3     foreach z ∈ IN(x) do
4         B[z] = 0  ◁ clear all set bits
```

needed in SNF-B: a hash table $H$ that maps each $y$ to $IN(y)$ and a bit map for all $z \in V_i$. The algorithm then traverses the graph along in-links by scanning each in-list from $G^-$. Starting from $x$, for each of its in-neighbors $y$, locate $IN(y)$ from $H$. For each $z \in IN(y)$, check it against the bit map. If the corresponding bit is not set, then $z$ is a new level-2 supporter of $x$ that has not been encountered before, we set the bit and increase the counter; otherwise, $z$ is deemed as a duplicate. Similarly, the algorithm has to eliminate level-1 supporters from the bit map.

In the last, SNF-B needs one additional step to reset the bit map. One possible approach is to reset the entire bit map memory space to zero, e.g., by calling function $memset()$. However, we find this approach extremely slow in practice as we need to repeat this for every node. In fact, we find that most positions of the bit map are $0$, which does not need to be touched. We only need to find the positions that are potentially set. Therefore, we repeat the process that is used to mark level-2 supporters and this time we reset all positions that we hit. This novel approach is illustrated in Algorithm 21.

## 4.3 CPU Complexity

In this section, we analyze the CPU complexity of SNF algorithms. The main CPU bottleneck is marking level-2 supporters. In both SNF-A/B algorithms, the number of attempts to mark level-2 supporters equals to the total number of nodes, including duplicates, that can be reached in two hops from each source node. This number is known as QVS (Quick-Visit Suppoters). Note that this number is the same regardless of traversing directions (i.e., along in-links or out-links) and can be computed by

$$QVS = \sum_{i=1}^{n} X_i Y_i. \tag{4.2}$$

Among these attempts, most of them will be deemed as duplicates and thus lead to no consequences. Such duplicate attempts only cause memory reads. Only the unique ones will cause an actual mark operation, i.e., a memory write. The total number of actual marks equals to the total number of supporters computed. Let $SUPP_2(i)$ be the number of level-2 supporters of node $i$, the number of marks is $\sum_{i=1}^{n} SUPP(i)$.

Eliminating level-1 supporters requires to check all direct neighbors of each node, which leads to a total of $m = \sum_{i=1}^{n} X_i = \sum_{i=1}^{n} Y_i$ checks. The number of checks that actually lead to a memory write which clears a bit depends on the number of times that a direct neighbor can also reach the source node in two hops. Finally, SNF-B requires an additional QVS number of memory writes to clear the bit map.

Table 4.1: CPU complexity of SNF

| Function | IRL domain | | ClueWeb domain | |
|---|---|---|---|---|
| | Read | Write | Read | Write |
| MarkL2 | 3.1T | 1.9T | 4.2T | 3.4T |
| ClearL1 | 1.8B | 1.3B | 415M | 340M |
| ClearL2 | – | 3.1T | – | 4.2T |

Table 4.2: Runtime (sec) in IRL domain

| $B$ (MB) | SNF-A | $B$ (MB) | SNF-B |
|---|---|---|---|
| 32 | 6,209 | 256 | 3,201 |
| 64 | 6,189 | 512 | 2,853 |
| 128 | 6,348 | 1,024 | 2,709 |
| 256 | 7,008 | 2,048 | 3,135 |

Table 4.1 illustrates the CPU cost of SNF on two graphs – IRL domain with $89$M nodes and $1.8$B edges, and ClueWeb domain with $31$M nodes and $415$M edges. As we can see from the table, in IRL domain graph, SNF issues $3.1$T attempts to mark level-$2$ supporters and among which $1.9$T lead to success (i.e., not duplicates). Such large number of memory operations may easily become the bottleneck and make the computation of neighborhood function CPU intensive. The other graph, ClueWeb domain, requires even more memory operations, $4.2$T reads and $3.4$T writes, despite the fact that this graph is smaller. This is because ClueWeb domain graph has higher density. The overhead of clearing level-$1$ supporters is relatively small compared to that of marking level-$2$ supporters and thus can be ignored. Note that SNF-B requires another $3.1$T and $4.2$T operations to clear the bit map in the two graphs; while SNF-A does not have this step. However, this does not necessarily mean that SNF-A is superior than SNF-B as the latter works on a bit map, which is a much smaller data structure and more cache efficient.

### 4.3.1 Runtime

Next, we analyze the runtime of each algorithm. Tables 4.2-4.3 illustrates the runtime using a six-core $4.4$ GHz Intel i7 CPU and a disk RAID system @ $1$ GB/sec. As we can see, the runtime is a function of the chunks size $B$. Instead of being a monotonic function, we notice that there exists an optimal $B$ for runtime. This is because of a tradeoff between CPU and I/O. In terms of CPU, cache efficiency is an important factor as it determines

Table 4.3: Runtime (sec) in ClueWeb domain

| $B$ (MB) | SNF-A | $B$ (MB) | SNF-B |
|---|---|---|---|
| 2 | 3,317 | 256 | 3,216 |
| 4 | 2,900 | 512 | 3,134 |
| 8 | 3,408 | 1,024 | 2,747 |
| 16 | 8,420 | 2,048 | 3,329 |

how fast the CPU operations can be done. When $B$ is small, the data structure maintained by SNF is small, which is able to greatly benefit from caching; however, a small $B$ also means SNF needs to scans the graphs more times, which causes a larger I/O. When $B$ is large, the effect reverses. The optimal $B$ in the IRL domain graph is $64$ MB and $1$ GB for SNF-A and SNF-B, respectively, and $4$ MB and $1$ GB in the ClueWeb case.

Note that SNF-A requires an array of last supporter IDs and counters, which is a much larger data structure than that of SNF-B, where the latter only requires a single counter and a bit map. This leads to two consequences: $1$) SNF-A is slower than SNF-B due to less cache efficiency, despite that SNF-B does more CPU operations as it needs an additional step to clear marks. Looking at the optimal runtime, SNF-B is $2.3$x and $1.1$x faster than SNF-A in the two graphs; $2$) In order to achieve the best performance, SNF-A has to operate with smaller $B$ as it needs to maintain larger data structures. It achieves best performance with the chunks size $64$ MB and $4$ MB; while SNF-B can work well with $1$ GB chunks. When the graph size scales, SNF-A may experience much larger I/O cost as it can only work with small chunks.

## 4.4 I/O Complexity

When the graphs do not fit into memory, external-memory operation is required to compute neighborhood functions. Our proposed algorithms SNF-A/B have already taken care of external-memory cases. They assume a simple I/O model that loads a chunk of

**Algorithm 22:** SNFD-A partition

```
1 Function SNFD-A Partition (G+, G−)
2     concurrently scan G+ and G− to obtain both out/in-list of each node (i, N+(i), N−(i))
3     for l = 1 to p do
4         X = N+(i) ∩ V_l
5         if X ≠ ∅ then
6             write (i, X) to subgraph G_l^+
7             write (i, N−(i)) to companion G_l^c
```

either $G^+$ or $G^-$ and scan the other graph from disk. Given the graph size $|G^+| = |G^-| = m$ and chunk size $B$, SNF-A/B requires a total I/O $m^2/B$. Increasing chunk size $B$ leads to fewer passes of scan over input graphs and thus less I/O. The maximum chunk size can be as large as memory size $M$. However, as we discussed in previous sections, larger chunk size may also lead to lower cache efficiency and thus slow down CPU operations.

As the graph size increases, the quadratic I/O complexity $m^2/B$ may soon become the bottleneck. In order to achieve more efficient I/O, we next propose a novel graph partitioning scheme called SNFD (Shallow Neighborhood Function on Disk) that can support disk operation of both SNF-A/B. Similar to SNF-A/B, SNFD starts by partition the nodes into $p$ mutually exclusive and jointly exhaustive subsets $V_1, \ldots, V_p$. Then, as shown in Algorithm 22, SNFD-A (which works similarly to SNF-A) partitions $G^+$ into $p$ subgraphs $G_1^+, \ldots, G_p^+$ by splitting the out-neighbors of each source node $i$ according to the node partition, i.e., the portion of out-neighbors $N^+(i) \cap V_l$ is written to $G_l^+$. We also create a companion file $G_l^c$ for each subgraph $G_l^+$ that contains in-neighbors of each source node $i$ written to $G_l^+$. SNFD-B (Algorithm 23) works similar to SNF-B, splits in-neighbors according to the node partition, and writes out-neighbors to the corresponding companion file. After partitioning, SNFD works on each pair of subgraph and companion file in the same way as SNF working on ($G^+$, $G^-$).

**Algorithm 23:** SNFD-B partition

```
Function SNFD-B Partition (G+, G−)
    concurrently scan G+ and G− to obtain both out/in-list of each node (i, N+(i), N−(i))
    for l = 1 to p do
        Y = N−(i) ∩ V_l
        if Y ≠ ∅ then
            write (i, Y) to subgraph G_l^−
            write (i, N+(i)) to companion G_l^c
```

### 4.4.1 I/O Upper-Bound

We next give a strict upper bound of the SNFD I/O cost. Note that the subgraph partitioning is non-redundant, i.e., $\sum_{l=1}^{p} |G_l^+| = m$. Therefore, we only need to focus on companion file size. Considering SNFD-A, for each source node $i$, suppose its out-neighbors are split into $c$ subgraphs, then we need to duplicate its in-neighbors into $c$ companion files. It is easy to see that $c \leq \min(X_i, p)$. Thus, the total companion size is upper-bounded by

$$H \leq \sum_{i=1}^{n} \min(X_i, p) Y_i. \tag{4.3}$$

Similarly, we can give the upper-bound of SNFD-B's I/O as

$$H \leq \sum_{i=1}^{n} \min(Y_i, p) X_i. \tag{4.4}$$

### 4.4.2 Load Balancing

Since SNFD requires to hold the entire subgraph in memory for processing, the subgraph size cannot exceed memory size $M$. In order to achieve the best I/O performance, load balancing is required to equalize the size of each subgraph to $M$, which produces the minimum possible $p$. Note that SNFD partitioning relies on node partitioning, i.e., how nodes are split into subsets $V_l$. We next introduce a deterministic node partitioning scheme

Table 4.4: I/O complexity in IRL domain

| RAM | SNF-A/B | SNFD-A | SNFD-B |
|---|---|---|---|
| 1,024M | 3.6B | 2.8B | 3.2B |
| 512M | 7.2B | 4.9B | 5.6B |
| 256M | 14.4B | 8.2B | 9.1B |
| 128M | 27.0B | 14.5B | 15.5B |
| 64M | 52.2B | 25.4B | 25.9B |
| 32M | 102.6B | 44.4B | 42.9B |

that controls the size of each subgraph and achieves load balancing.

We utilize *sequential* node partitioning that places nodes with consecutive IDs into the same partition. This brings the following advantages: 1) Instead of keeping track of which nodes belong to a particular partition $V_l$, we now just need to remember the partition boundaries $a_1, a_2, \ldots, a_{p+1}$, where $a_1 = 1$ and $a_{p+1} = n + 1$, such that node $i$ belongs to partition $V_l$ if $a_l \leq i < a_{l+1}$; 2) by grouping nodes with consecutive IDs together, it makes future memory operations, e.g., hash table lookups, more cache efficient.

In SNFD-A, the subgraph size $|G_l^+|$ depends on the total in-degree of nodes in $V_l$. Similarly, SNFD-B's subgraph size $|G_l^-|$ depends on the total out-degree of nodes in $V_l$. Combined with sequential node partitioning and the maximum subgraph size $M$, we can compute each node partition $V_l$ in SNFD-A by

$$\sum_{i=a_l}^{a_{l+1}-1} Y_i = M, \tag{4.5}$$

and that of SNFD-B by

$$\sum_{i=a_l}^{a_{l+1}-1} X_i = M, \tag{4.6}$$

Table 4.5: I/O complexity in ClueWeb domain

| RAM | SNF-A/B | SNFD-A | SNFD-B |
|---|---|---|---|
| 256M | 830M | 579M | 775M |
| 128M | 1.7B | 995M | 1.3B |
| 64M | 2.9B | 1.7B | 2.2B |
| 32M | 5.4B | 2.9B | 3.8B |
| 16M | 10.8B | 4.9B | 6.5B |
| 8M | 21.6B | 8.6B | 10.9B |

### 4.4.3 I/O Comparison

Finally, we compare the I/O performance of each algorithm in the IRL and ClueWeb domain graphs. Tables 4.4-4.5 show how the I/O cost scales with RAM size, where both I/O and RAM are measured by number of edges. As we can see from the tables, SNFD significantly outperforms SNF-A/B, where the latter requires quadratic I/O complexity. The smaller the RAM size, the more advantage SNFD has. On the other hand, SNFD-A tends to be better than SNFD-B in I/O.

We next present an important application of shallow neighborhood function in ranking Internet host.

## 4.5 Host Ranking

Search engines have become the primary mechanism for users to access the content of the Internet and for websites to attract visitor traffic. To achieve high ranking in search results, web spammers attempt to deceive search engines and manipulate their ranking algorithms by employing such tactics as content stuffing, link exchanges, and page hijacking [33]. In order for search engines to remain robust against these exploits, it is crucial to properly manage spam. Instead of doing this in the indexing or querying phase, we address the problem from another angle: *avoiding spam during data collection*.

A search engine typically relies on a distributed web crawler to download web pages

and maintain a copy of the web. In order to produce spam-free input to indexing and data-mining algorithms, web crawlers need a prioritization scheme to schedule the crawl order of pages in the frontier [73]. Ideally, pages from highly reputable websites achieve top priority and are crawled first. Similarly, spam pages are scheduled towards the end of the crawl or avoided altogether. Traditional crawlers [12], [36], [54], [70] do not implement spam avoidance and usually rely just on BFS. Simulations of page scheduling [4], [16] suggest ranking the frontier by PageRank [59]; however, these methods have not been used in non-commercial web crawls due to computational complexity and have remained largely untested.

Instead, we investigate a prioritization model that ranks websites and assigns *budgets* to them according to the ranking position. Budgets specify the amount of crawl resources (i.e., bandwidth, RAM, and disk space) allocated to each site. Generally, a crawler provides non-trivial budgets to the top-$R$ (e.g., $100$K) highest-ranked hosts* and crawls all others with a small fixed budget. This model is efficient since it ranks websites rather than pages and is concerned only with ordering the top-$R$ hosts.

The key to this model is a good host-ranking algorithm that achieves the following three goals. First, it must exclude spam from the top list. Since top-ranked hosts get a disproportionally high fraction of the budget, any spam at the top is likely to waste large amounts of crawl resources and pollute the final dataset. Second, the algorithm must be efficient as it needs to rank hosts in real-time and frequently update the ordering as more hosts and links are discovered. Finally, the method should not rely on human input (e.g., whitelisting/blacklisting) or complex training. The web evolves rapidly, which makes human and machine classification difficult to keep up-to-date.

Unfortunately, not much attention has been paid to this problem in the academic com-

*In this chapter, we refer to websites as hosts, which can be identified by hostnames. A hostname is a label assigned to a host computer and can be mapped to IP addresses by using a DNS resolver. For example, en.wikipedia.org, mail.google.com, and news.bbc.co.uk are hosts.

munity. Related work that attempts to rank hosts either simply applies PageRank to the host graph [24] or computes the sum of PageRank values inside each website [26]. Moreover, neither of these approaches produces detailed analysis of the obtained ranking list. This calls for more evaluation and possibly better methods for ordering websites.

### 4.5.1 Contributions

We study host ranking on two large web crawls: a 641M-host dataset IRLbot and a 118M-host dataset ClueWeb. These datasets are crawled with different seed pages and algorithms, providing complementary results.

We start by showing that existing techniques such as PageRank [59], TrustRank [34], in-degree (IN), and level-2 supporters ($SUPP_2$) [9], [73] leave much to be desired. For example, PageRank produces $14\%$ spam in the top-1K, while IN is even worse with $24\%$. TrustRank beats both of these by taking advantage of its whitelist, but $3.8\%$ spam within its top-1K is still far from ideal. $SUPP_2$ drops this number to $0.7\%$, but allows $8.4\%$ spam within the top-100K. These methods' main drawback lies in using hosts themselves as indication of endorsement. Since spammers can create any number of hosts *for free*, inflating the ranking of arbitrary targets becomes easy.

To overcome this issue, we propose a novel ranking framework that utilizes multiple graphs and ranks hosts with the endorsement from *finite* Internet resources. We first consider the use of domains[†]. Due to the financial cost involved, it is difficult for a spammer to get a large number of domains under control. We rank hosts by the number of level-2 domain supporters and call this algorithm L2-D. Our evaluation on IRLbot shows that compared to the best existing methods, L2-D produces $3.5$ times less spam among the top-1K list, $6.0$ times less spam from 1K to 10K, and keeps its advantage all the way up to 100K. Results with ClueWeb are similar.

[†]A domain must be purchased at a TLD or cc-TLD registrar (e.g., google.com, wikipedia.org, amazon.co.uk).

To further improve performance of L2-D, we incorporate the DNS infrastructure into our ranking framework. We replace level-2 domains with their authoritative DNS nameservers. In web-hosting services such as GoDaddy, there can be millions of domains co-hosted under a shared DNS cluster. Thus, by using authoritative DNS nameservers instead of domains, we eliminate inflation from a large number of low-quality domains within web-hosting services. Moreover, we propose simple ways to detect and remove highly co-hosted websites and links suspected of being hijacked. In our final method, we end up with no spam among the top-1K, less than $0.2\%$ among the top-10K, and only $2.0\%$ within the top-100K.

The rest of this chapter is organized as follows. Section 4.6 introduces related work. Section 4.7 formalizes the problem of topological ranking and proposes our multi-graph framework. We analyze two examples that use our framework to rank hosts with domains and authoritative DNS nameservers in section 4.8 and 4.9, respectively. Section 4.10 briefly compares the complexity of different multi-level supporter algorithms. Finally, section 4.11 concludes the chapter with our findings.

## 4.6 Related Work

### 4.6.1 Spam Detection

Web spam has become prevalent [33]. There have been numerous solutions to detect spam using the web topology. SpamRank [10] proposes to penalize a page if the PageRank score distribution of its in-neighbors does not follow a power law. Spam mass [32] of a page estimates the portion of PageRank that is contributed by spam nodes. A page is suspicious if the spam mass is larger than some threshold $\tau$. Study [83] starts with a set of blacklisted pages and propagates penalties in the neighborhood of blacklisted nodes. Spam farms have been found to create densely connected structures in the webgraph and have been detected by decomposing the graph into strongly connected components (SCC)

[18].

There has been earlier work [27], [79], [86] that suggests to leverage DNS information for spam detection. A high number of hostnames resolving to a single IP is considered indicative of spam in [27]. Spam domains are known to use wildcard DNS entries to generate hostnames with popular keywords. Similarly, IPs have been used as one of the features while classifying spam [79], and connected components in DNS queries have been used to find botnet client-server communication inside a network [86].

Other work [27], [42], [56], [75], relies on additional features for spam detection, including URL structure, HTTP headers, redirection, click-through data, and content analysis. These methods normally require human labeling and training with large amounts of prior knowledge.

While spam detection techniques can potentially help us to remove spam, they do not further distinguish the quality of the rest non-spam content (e.g., microsoft.com vs. some personal homepage). Such quality difference is important for web crawlers to decide the budgets. Thus, we are facing a different problem: *websites ranking and budgeting*. Our goal is not to develop a spam detection method, but a method to detect good websites and assign the majority of budgets to them.

### 4.6.2 Ranking

The problem of ranking websites has not been widely studied. Existing work [24], [26] is limited to simply applying PageRank on the host graph. Another method $\text{SUPP}_2$, which extends in-degree rank (IN) by counting the number of neighbors at distance 2, is found effective in eliminating spam when ranking domains [73]. Its performance in ranking hosts still needs to be examined.

Other methods that are originally created for page ranking, can be potentially applicable to websites. TrustRank [34] modifies PageRank by teleporting to trusted nodes only.

Trust scores then propagate through neighbors from a seed set of whitelist nodes. CredibleRank [14] assigns credibility to all pages by measuring the probability for random walks to end up on blacklisted nodes. However, selection of comprehensive whitelists or blacklists is difficult considering the size and dynamics of the Internet. As stated in introduction, one of our goals is to create an automatic algorithm that works on any underlying graph and keeps up with the changing web.

## 4.7 Topological Ranking

In this section, we formalize the problem of topological ranking and present a novel multi-graph ranking framework.

### 4.7.1 Single-Graph Ranking

The web is a huge collection of web pages linked to each other, which can be represented as a directed graph. In order to study host ranking, we first construct the host graph by condensing all pages found in the same website into a single node and adding an edge from node $i$ to node $j$ if any page in $i$ links to any page in $j$. To avoid ranking inflation from duplicate inter-host hyperlinks, the graph is *unweighted* (i.e., duplicate edges are removed).

Based on the linking structure of the graph, an importance score, which we call *reputation*, can be computed for each node. For example, if host $A$ has a hyperlink to host $B$, this implies that $A$ recommends the content of $B$ and can be viewed as an endorsement of $B$. Thus, the reputation of $B$ can be computed simply by the number of hosts that recommend it. This method is known as in-degree ranking (IN). We call the nodes that contribute to the reputation of a target its *supporters*.

In general, it may be argued that not all supporters are equally important. Consider two nodes with the same in-degree but one is pointed to by Yahoo and the other by a number of spam hosts. Thus, a *weight* may be assigned to each supporter to distinguish between

them. PageRank [59] decides the weights using the stationary probability for a random walker on the graph to be found at each node. Given a directed graph $G$ with $n$ nodes, at each step, the walker either chooses one of the out-links to follow with probability $\alpha = 0.85$ or teleports to a random node with probability $1 - \alpha$. In this case, the reputation of node $j$ is given by:

$$R_j = \alpha \sum_{(i,j)\in E} \frac{R_i}{d_{out}(i)} + \frac{1-\alpha}{n}, \tag{4.7}$$

where $d_{out}(i)$ is the out-degree of node $i$. The reputation is computed in multiple iterations, which can be viewed as a progressive adjustment to weights of supporters.

One problem of PageRank is that even nodes without any value gain some reputation from random teleportation. TrustRank [34] solves this problem by teleporting to trusted nodes only. Prior work [24] has shown that by using two selected reputable sites (i.e., www.microsoft.com and www.yahoo.com) as the whitelist ($W$), TrustRank produces much better top ranking lists than PageRank. In this chapter, we use www.google.com as the only trusted node since it is the most popular Internet service today. We also tried to use a larger $W$; however, it turned out to perform worse than www.google.com alone (we omit the results here for brevity). The lesson we learn is that TrustRank boosts neighbors of $W$, which for large $W$ may not always be desirable as reputable hosts are not required to link to other similarly ranked nodes.

Supporters can also be generalized to multiple levels. Defining $d(i, j)$ to be the shortest distance from node $i$ to node $j$, the reputation of $j$ can be given by:

$$R_j = |\{i : d(i, j) = D\}|. \tag{4.8}$$

We call $i$ a level-$D$ supporter of $j$. Note that this is equivalent to backwards BFS from each node to depth $D$. If $D = 1$, we get IN. With $D \geq 2$, we call the algorithm SUPP$_D$

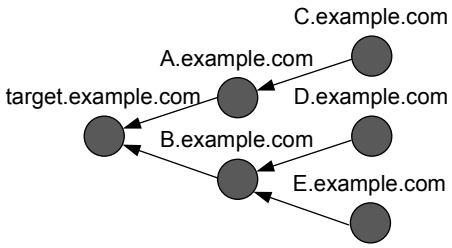



Figure 4.2: Host graph.

(supporters at level-$D$).

### 4.7.2 Multi-Graph Ranking

We define a graph to be an *infinite graph* if the nodes contained in the graph can be infinitely created for free; otherwise it is a *finite graph*. A host graph is infinite because once a spam domain is registered, a spammer can use automated scripts and DNS wild-card entries to create an infinite number of hosts under this domain. These hosts can be densely connected to each other and form link farms. Traditional algorithms that work on this single infinite graph are susceptible to trivial inflation. Figure 4.2 shows an example of manipulating the single-graph ranking techniques at the host level. A spammer who controls example.com creates a link farm with five dummy hosts. Two of them ($A$ and $B$) are boosting the IN ranking of the target. The other three hosts $C$, $D$ and $E$ support the target at level-2 and inflate its $\text{SUPP}_2$ count. Meanwhile, the target's PageRank score is boosted as well by receiving credit from dummy in-neighbors.

To address this issue, we propose a set of novel ideas for ranking purpose. First, we restrict supporters to come only from *finite* graphs, whose nodes cannot be controlled in a large number by spammers due to the financial cost involved. Two obvious examples of such nodes are domains and DNS nameservers. Second, we use *heterogenous* graphs that contain links between different types of nodes. This allows the reputation score of nodes

in infinite graphs to be computed using nodes in finite graphs. Finally, we use multiple graphs to create ranking methods that are more resistant to spam.

In order to rank on multiple graphs, we need a mechanism that allows reputation scores to flow between them. Given two directed graphs $G_1(V_1, E_1)$, $G_2(V_2, E_2)$ ($V_i$ is the set of nodes and $E_i$ is the set of edges) and two edges $(x_1 \leftarrow y_1) \in E_1$, $(x_2 \leftarrow y_2) \in E_2$, if the pair $(y_1, x_2)$ satisfies certain condition $\theta$ (i.e., $\theta(y_1, x_2) = 1$), then we can connect the two edges together to form a chain that allows one to traverse between the two graphs. Based on this concept, we define the *theta join* operation between two graphs to be:

$$\begin{aligned} G_1 \bowtie_\theta G_2 &= \{x_1 \leftarrow y_1 \leftarrow x_2 \leftarrow y_2 : x_1 \leftarrow y_1 \in E_1, \\ &\qquad x_2 \leftarrow y_2 \in E_2, \theta(y_1, x_2) = 1\}. \end{aligned}$$

If the $\theta$ condition is $y_1 = x_2$, we define the *natural join* of two graphs as:

$$\begin{aligned} G_1 \bowtie G_2 &= \{x_1 \leftarrow y_1 \leftarrow y_2 : x_1 \leftarrow y_1 \in E_1, \\ &\qquad x_2 \leftarrow y_2 \in E_2, y_1 = x_2\}. \end{aligned}$$

Note that one edge in a graph can be connected with multiple edges in the other graph. These joins are similar to their usage between tables in database systems. More generally, these operations can be applied to $n$ graphs:

$$\begin{aligned} G_1 \bowtie G_2 \cdots \bowtie G_n &= \{x_1 \leftarrow y_1 \leftarrow y_2 \cdots \leftarrow y_n : \\ &\qquad x_1 \leftarrow y_1 \in E_1, \ldots, x_n \leftarrow y_n \in E_n, \\ &\qquad y_1 = x_2, \ldots, y_{n-1} = x_n\}. \end{aligned}$$

In the chain above, nodes $y_1, \ldots, y_n$ are supporting the target $x_1$ at levels $1$ to $n$.

Note that $y_1, \ldots, y_n$ can be completely different types of nodes and there may be multiple chains supporting $x_1$. If $G_1 = G_n$, the chain becomes a cycle, which potentially allows PageRank-style iterative methods to be applied on multiple graphs. This new ranking framework allows us to compute the reputation score of $x_1$ using a variety of nodes (e.g., domains and DNS nameservers) from multiple graphs.

### 4.8 Domain Supporters

In this section, we show an example of incorporating domains into our ranking framework. A straightforward way is to create a heterogenous Host-Domain graph from the host graph by condensing in-neighbors into their domains and count the number of domain in-neighbors. Figure 4.3 shows such a graph constructed from Figure 4.2. Two in-neighbor hosts ($A$ and $B$) are merged into one domain node (example.com). This prevents inflation within direct neighbors. In our experiment, we find this technique outperforms IN and PageRank, but still falls behind TrustRank and $\text{SUPP}_2$. Since supporters at level-2 are found to produce good ranking lists [73], we extend domain supporters to level-2 (we call this method L2-D) by applying natural join on the Host-Domain graph and a Domain-Domain graph, which can be similarly constructed from the host graph. This produces Host-Domain-Domain chains. Figure 4.4 shows such a chain constructed from Figure 4.2. The original nodes $C$, $D$, and $E$ at level-2 are also merged into a single domain node. There are zero domain supporters at level-2 in the above case since the distance from example.com to the target is 1. Thus, L2-D completely eliminates the effect of such manipulation in Figure 4.2. Moreover, by condensing hosts into domains, the graph size and computational overhead are significantly reduced. The rest of this section compares the performance of L2-D with single-graph techniques.

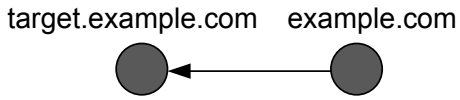



Figure 4.3: Host-Domain graph.

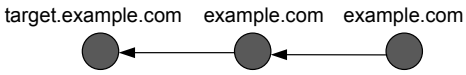



Figure 4.4: Host-Domain-Domain chain.

### 4.8.1 DataSets

Our first dataset IRLbot contains 6B HTML pages and reveals a subset of the web with 41B unique pages. The corresponding host graph consists of 641M sites and 6.8B edges. Our second dataset ClueWebhas 1B HTML pages. The corresponding host graph contains 118M nodes and 1.1B edges. In our experiment, we find ClueWebhas much less spam than IRLbot.

### 4.8.2 Manual Analysis

A crawler should assign most of the budget to top-ranked hosts, which will consume most of the crawl resources and contribute the major portion of the final crawl dataset. Thus, the most important goal for a host ranking algorithm is to exclude spam from the top list, which explains why our evaluation focuses on the quality of top-ranked hosts.

During our evaluation, we manually label each host by sampling a few pages (always including the home page) within it. We consider the following categories as spam: (1) *parked:* hosts primarily used to monetize web traffic by displaying ads, which mostly consist of previously expired domains and are kept alive for the value of incoming links; (2) *adult:* hosts that serve pornographic content; (3) *content spam:* hosts with no meaningful

Table 4.6: Top-10 ranked hosts in IRLbot by PageRank-style methods.

| PageRank | | TrustRank | |
|---|---|---|---|
| Site | GTR | Site | GTR |
| go.microsoft.com | 4 | www.google.com | 9 |
| www.blogger.com | 9 | go.microsoft.com | 4 |
| www.adobe.com | 9 | www.microsoft.com | 8 |
| www.microsoft.com | 8 | www.adobe.com | 9 |
| www.google.com | 9 | www.macromedia.com | 9 |
| searchportal.information.com | – | www.trafic.ro | 7 |
| www.macromedia.com | 9 | www.statcounter.com | 9 |
| www.sedoparking.com | – | searchportal.information.com | – |
| find-fm.com | 3 | validator.w3.org | 9 |
| validator.w3.org | 9 | www.apple.com | 9 |

information, excessive use of keywords and machine generated text; (4) *link spam:* hosts involved in link exchanges or affiliate marketing with a large number of revenue links.

We also use Google Toolbar Ranks (GTR) [66] of the home page to help us assess the quality of each websites. GTR is a value from 0 to 10 that indicates Google's opinion about the reputation of each page. Pages with a higher GTR value are more reputable. Google also returns a special "no GTR" response if the page has not been crawled, no longer exists, or has been removed from the index on purpose.

We first list the top-10 ranked hosts and their GTRs in Tables 4.6-4.9. In both datasets, $SUPP_2$ and L2-D produce reliable top-10 lists with reputable websites whose GTR values are at least 7, while PageRank suffers from significant amounts of spam. For example, this includes searchportal.information.com, find-fm.com and www.sedoparking.com. The first two are revenue-marketing search engines hosting predominantly spam; the last one provides parking services with highly questionable content. Also, we find that the entire domain information.com runs a large number of parked hosts that mostly serve ads.

One interesting case occurs in ClueWeb, where both PageRank and IN end up with 10

Table 4.7: Top-10 ranked hosts in IRLbot by degree-based methods.

| IN | | $SUPP_2$ | | L2-D | |
|---|---|---|---|---|---|
| Site | GTR | Site | GTR | Site | GTR |
| www.blogger.com | 9 | www.microsoft.com | 8 | www.microsoft.com | 8 |
| www.google.com | 9 | www.google.com | 9 | www.google.com | 9 |
| validator.w3.org | 9 | www.adobe.com | 9 | www.adobe.com | 9 |
| go.microsoft.com | 4 | en.wikipedia.org | 8 | www.macromedia.com | 9 |
| www.microsoft.com | 8 | groups.google.com | 8 | www.geocities.com | 7 |
| www.adobe.com | 9 | www.geocities.com | 7 | www.apple.com | 9 |
| wordpress.org | 9 | www.macromedia.com | 9 | en.wikipedia.org | 8 |
| www.yahoo.com | 9 | www.amazon.com | 8 | www.youtube.com | 9 |
| www.geocities.com | 7 | www.myspace.com | 8 | www.amazon.com | 8 |
| en.wikipedia.org | 8 | www.youtube.com | 9 | www.w3.org | 9 |

hosts from skyrock.com in their top-30 lists. Further analysis shows that skyrock.com is a social network where each user is assigned a unique hostname. Users frequently link to each other, which promotes their PageRank and in-degree values. Also, skyrock.com is massively crawled in ClueWeb (i.e., 4.2M pages downloaded), which brings a large number of inter-host links inside skyrock.com into the host graph. This gives a good example of how the classic methods are susceptible to inflation.

Next, we extend our manual analysis to a larger range and see how each ranking algorithm manages spam. We first compare the spam occurance within the top-1K list of each algorithm. Define $u_r$ to be the number of spam nodes in ranking position $[1, r]$. Figure 4.5 plots $u_r$ vs $r$ for each algorithm. Observe that $SUPP_2$ and L2-D produce almost spam-free top-1K lists. In the IRLbot case shown in Figure 4.5(a), we find only 7 spam cases from $SUPP_2$, with the first one being www.pathfinder.com in position 282. L2-D is even better with only 2, www.spywareinfo.com in position 463 and www.rense.com in position 865. In the ClueWeb case shown in Figure 4.5(b), we do not find any spam within the top-1K list of $SUPP_2$. For L2-D the only suspicious entry is www.pathfinder.com ranked at 825.

Table 4.8: Top-10 ranked hosts in ClueWeb by PageRank-style methods.

| PageRank | | TrustRank | |
|---|---|---|---|
| Site | GTR | Site | GTR |
| www.adobe.com | 9 | www.google.com | 9 |
| login.live.com | 8 | www.adobe.com | 9 |
| g.live.com | – | www.macromedia.com | 9 |
| g.msn.com | 6 | www.microsoft.com | 8 |
| go.microsoft.com | 4 | validator.w3.org | 9 |
| www.google.com | 9 | www.amazon.com | 8 |
| www.blogger.com | 9 | www.statcounter.com | 9 |
| www.skyrock.com | 6 | maps.google.com | 9 |
| uk.skyrock.com | 5 | www.youtube.com | 9 |
| es.skyrock.com | 3 | jigsaw.w3.org | 6 |

In sharp contrast, we discover spam throughout the entire top-1K lists of IN, PageRank, and TrustRank. In the IRLbot case, they end up with 238, 139, and 38, respectively. Although, the result is better in ClueWeb, these methods still amass 59, 67, and 23 hosts. Another indication of how poor traditional approaches are is that spam in their ranking lists starts showing up at the very top (e.g., within the top-10), which severely undermines their credibility.

While $SUPP_2$ and L2-D have comparable performance on top-1K lists, $SUPP_2$ starts to fall behind as we extend our analysis up to 100K. Since evaluating the entire list is quite labor intensive, we only sample 10% of the hosts from position 1K to 10K and 1% from position 10K to 100K. Table 4.10 lists the projected spam fraction. As before, PageRank and IN produce a significant amount of spam in all intervals. Additionally, their top-1K list contains *more* spam than the other intervals, which is the opposite from the desired situation where the spam probability is minimized near the top. TrustRank performs better than IN and PageRank, but it is still not able to match either of the supporters algorithms. While $SUPP_2$ does well in the top-1K, it delivers a significant amount of spam in 1K-10K

Table 4.9: Top-10 ranked hosts in ClueWeb by degree-based methods.

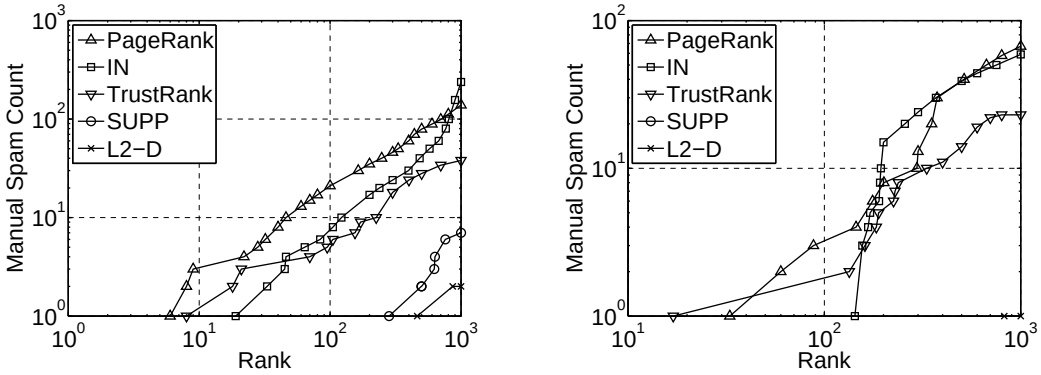
| IN | | $SUPP_2$ | | L2-D | |
|---|---|---|---|---|---|
| Site | GTR | Site | GTR | Site | GTR |
| login.live.com | 8 | www.google.com | 9 | www.google.com | 9 |
| g.live.com | – | www.youtube.com | 9 | www.adobe.com | 9 |
| go.microsoft.com | 4 | www.adobe.com | 9 | www.microsoft.com | 8 |
| g.msn.com | 6 | www.microsoft.com | 8 | en.wikipedia.org | 8 |
| www.skyrock.com | 6 | www.myspace.com | 8 | www.youtube.com | 9 |
| uk.skyrock.com | 5 | en.wikipedia.org | 8 | www.apple.com | 9 |
| es.skyrock.com | 3 | www.apple.com | 9 | www.geocities.com | 7 |
| fr.skyrock.com | 5 | www.amazon.com | 8 | www.macromedia.com | 9 |
| qc.skyrock.com | 5 | www.flickr.com | 9 | www.amazon.com | 8 |
| ca.skyrock.com | 1 | www.facebook.com | 9 | www.myspace.com | 8 |

| | IRLbot | | | ClueWeb | | |
|---|---|---|---|---|---|---|
| Algorithm | top-1K | 1K-10K | 10K-100K | top-1K | 1K-10K | 10K-100K |
| IN | 23.8% | 13.9% | 13.3% | 5.9% | 7.2% | 7.7% |
| PageRank | 14.0% | 11.9% | 8.8% | 6.7% | 4.8% | 5.9% |
| TrustRank | 3.8% | 7.8% | 7.6% | 2.3% | 2.2% | 5.6% |
| $SUPP_2$ | 0.7% | 8.4% | 6.0% | 0.0% | 2.3% | 2.9% |
| L2-D | 0.2% | 1.4% | 5.3% | 0.1% | 0.6% | 2.6% |

Table 4.10: Projected fraction of spam.

(8.4% and 2.3% respectively in the two datasets). In the same interval, L2-D manages to keep the fraction at 1.4% and 0.6%, which are respectively 6 times and 4 times less. Between 10K and 100K, we also see non-trivial improvements from L2-D.

Finally, we summarize the results of our manual inspection in Table 4.11. In total, we analyzed 25,398 unique hosts and marked 1,601 of them as spam. Parked hosts are the most common category. We identified several large groups of parked hosts (e.g., information.com), where sites in the same group display mostly identical content, which is ad links.




(a) IRLbot (b) ClueWeb

Figure 4.5: Manual spam count.

Table 4.11: Spam Categories

| | | | |
|---|---|---|---|
| Content Spam | 160 | Link Spam | 119 |
| Adult | 413 | Parked | 909 |
| Non-existent | 3,600 | Non-spam | 20,197 |

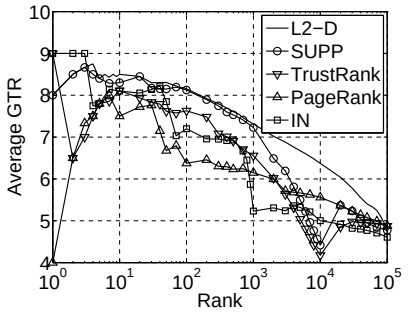
### 4.8.3 Automated Analysis

In this section, we rely on GTR to build an automated evaluation method for top-ranked hosts. We query the GTR values from Google for all top-100K hosts of each algorithm. After removing duplicates, we collect GTRs for $588,586$ unique websites.

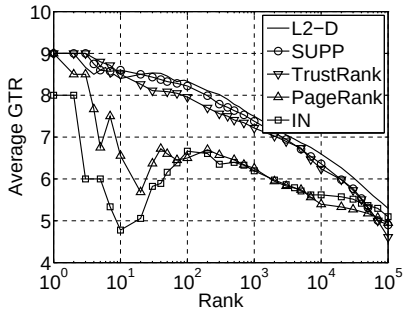
Figure 4.6 illustrates the results of GTR analysis. We first look at the running average of GTRs up to position 100K in subfigures (a) and (b). Define $g_i$ to be the GTR value of the host in ranking position $i$, the running average GTR up to position $K$ is given by:

$$avg_K = \frac{\sum_{i=1}^{K} g_i}{K}.$$

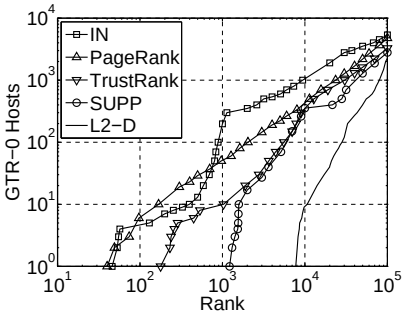
L2-D clearly maintains the highest average in both datasets. $\text{SUPP}_2$ initially stays close



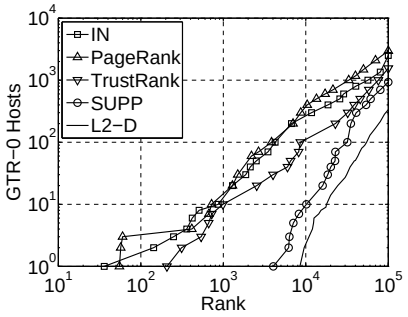

(a) IRLbot average GTR



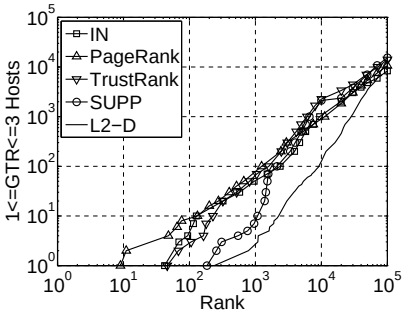

(b) ClueWeb average GTR



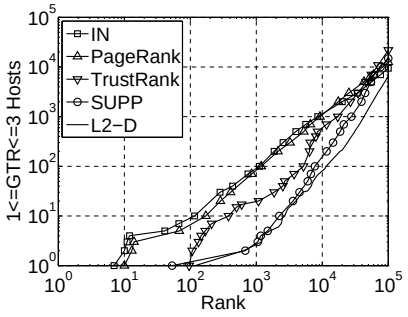

(c) IRLbot GTR-0



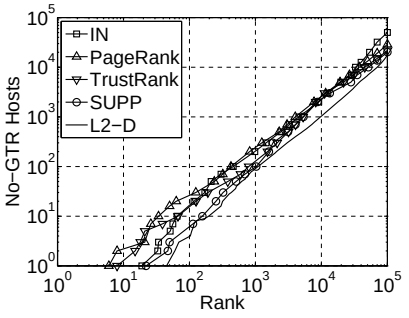

(d) ClueWeb GTR-0



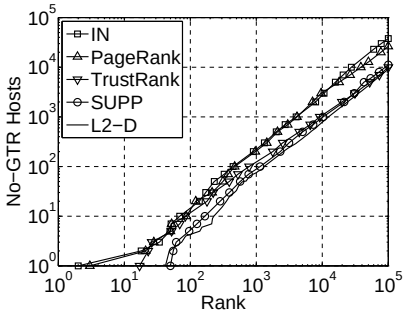

(e) IRLbot $1 \leq$GTR$\leq 3$




(f) ClueWeb $1 \leq$GTR$\leq 3$




(g) IRLbot no-GTR




(h) ClueWeb no-GTR

Figure 4.6: GTR analysis on top ranked hosts.

to L2-D, but falls behind after position 1K, where the average GTR of both $SUPP_2$ and TrustRank experiences a dramatic drop in IRLbot due to a large number of hosts with no or low GTR value. This agrees with our manual evaluation results. PageRank and IN trail in last place.

Parts (c) and (d) shows the cumulative distribution of hosts with a GTR value 0, which is a strong indication of poor quality and potential spam. L2-D is again the clear winner and maintains a significant separation from other methods. It admits almost no GTR-0 hosts into its entire top-10K list. $SUPP_2$ takes a second place. The other three techniques IN, PageRank, and TrustRank all contains GTR-0 hosts within the top-1K. When comparing the number of hosts with low (i.e., $1-3$) and no GTR in subfigures (e)-(h), L2-D still holds advantage over all other algorithms, and PageRank and IN lose in every aspect of the comparison. TrustRank splits the difference. While IN and PageRank are close to each other, the former requires much lower computational complexity, which leaves the latter even more at a disadvantage.

Note that although we rely on GTR in our evaluation, it cannot be used during crawls because this information for such a huge number of hosts (e.g., 641M in IRLbot) cannot be acquired in real-time without getting blocked by Google. Additionally, GTR data is not fine-granular enough to build a useful ranking. Instead, our goal is to design a framework that has comparable performance with Google GTR, but without resorting to hundreds of features, manual input, and complex machine-learning rules.

To summarize, the existing methods that utilize a single infinite graph (IN, PageRank, TrustRank, and $SUPP_2$) all produce a significant amount of spam and low-GTR hosts in the top list. If these algorithms were used to drive a web crawler, a large portion of crawl resources could be wasted on low-quality content. Our proposed algorithm L2-D works on multiple graphs and uses finite resources to compute reputation scores. It produces much better ranking, although still admits occasional outliers. This motivates us to seek even

better methods.

## 4.9 Nameserver Supporters

While domains cannot be infinitely proliferated, the spammers can register a certain number of them through web-hosting services and participate in link exchanges with other spammers to accumulate domains supporters. Additionally, web-hosting companies themselves are now more involved in parking activities, where millions of expired domains are stuffed with keywords and sponsored links. While providing easy access for web users to register domains and build websites, parking is mostly spam from the perspective of a search engine. We notice that such parking domains normally share common DNS infrastructures (see section 4.9.3 for more details). Thus, we seek to leverage DNS resources and incorporate them into our ranking framework.

### 4.9.1 DNS Resolution

To collect IP addresses and DNS nameservers, we traverse the DNS tree to perform lookups on all discovered hosts in both crawls using a custom C++ DNS resolver we developed for this purpose. Combining 641M hosts from IRLbot and 118M from ClueWeb, we obtain 721M total after removing duplicates (i.e., 39M hosts are in common). In order to collect comprehensive DNS information for each host, we implement an iterative resolver that starts from the root and fully traverses the tree to identify the authoritative DNS nameservers responsible for each domain. We end up with mapping these hosts to 4.9M unique IP addresses, and 809K different /24 subnets. This experiment also discovers 902K unique nameserver IPs.

### 4.9.2 IP Subnet Graph

Since a large number of companies normally own entire /24 subnets and a host may be assigned multiple IPs within the subnet, we group IPs into /24 subnets. Then, we build

| Rank | Subnet | WHOIS | in-degree |
|---|---|---|---|
| 1 | 64.4.11.0/24 | Microsoft MSN | 273,961 |
| 2 | 74.125.227.0/24 | Google | 210,068 |
| 3 | 82.98.86.0/24 | Sedo GmbH | 197,264 |
| 4 | 192.150.16.0/24 | Adobe | 175,862 |
| 5 | 208.87.35.0/24 | Secure Host | 171,096 |
| 6 | 184.168.221.0/24 | GoDaddy | 164,004 |
| 7 | 50.63.202.0/24 | GoDaddy | 159,354 |
| 8 | 208.73.211.0/24 | Oversee.net | 139,308 |
| 9 | 69.43.161.0/24 | Trellian Pty | 133,207 |
| 10 | 208.91.197.0/24 | Confluence Inc. | 130,893 |

Table 4.12: Top-10 subnets ranked by IN in IRLbot.

| Rank | Subnet | WHOIS | in-degree |
|---|---|---|---|
| 1 | 74.125.227.0/24 | Google | 167,249 |
| 2 | 192.150.16.0/24 | Adobe | 129,805 |
| 3 | 82.98.86.0/24 | Sedo GmbH | 122,082 |
| 4 | 184.168.221.0/24 | GoDaddy | 108,732 |
| 5 | 208.87.35.0/24 | Secure Host | 107,496 |
| 6 | 50.63.202.0/24 | GoDaddy | 106,038 |
| 7 | 64.4.11.0/24 | Microsoft MSN | 90,908 |
| 8 | 208.80.154.0/24 | Wikimedia | 89,747 |
| 9 | 208.73.211.0/24 | Oversee.net | 85,022 |
| 10 | 69.43.161.0/24 | Castle Access Inc. | 81,179 |

Table 4.13: Top-10 subnets ranked by IN in ClueWeb.

subnet graphs by replacing hosts with their subnets and removing duplicate edges. In order to find reputable hosts in highly ranked subnets, we apply IN on the subnet graphs of both datasets and list the top-10 nodes in Tables 4.12-4.13.

We arrive at a somewhat unexpected discovery that subnet ranking does not differentiate between parking sites and reputable domains. Although, several legitimate companies appear near the top, such as Microsoft, Google, Adobe, and Wikimedia, the others are mostly parking services. Since they are hosting millions of domains inside their subnets,

| Rank | Subnet | WHOIS | Hosts | Domains | Nameservers | Type |
|---|---|---|---|---|---|---|
| 1 | 184.168.221.0/24 | GoDaddy | 12,986,177 | 949,324 | 164 | Parking |
| 2 | 82.98.86.0/24 | Sedo GmbH | 10,857,325 | 763,298 | 449 | Parking |
| 3 | 198.202.143.0/24 | Level 3 Comm | 7,165,125 | 25,384 | 4 | Content Spam |
| 4 | 216.21.239.0/24 | Web.com | 7,054,488 | 59,931 | 853 | Parking |
| 5 | 74.125.227.0/24 | Google Inc. | 5,447,904 | 1,464 | 12 | Blogs |
| 6 | 91.203.184.0/24 | Skyrock | 4,711,435 | 68 | 13 | Social Network |
| 7 | 193.93.125.0/24 | Skyrock | 4,710,583 | 86 | 19 | Social Network |
| 8 | 157.55.96.0/24 | Microsoft MSN | 3,917,708 | 32 | 10 | Social Network |
| 9 | 87.118.118.0/24 | Keyweb | 3,791,882 | 873 | 112 | Parking |
| 10 | 74.63.153.0/24 | ViaWest | 2,982,248 | 5 | 4 | Affiliate Marketing |

Table 4.14: Top-10 subnets based on host density.

| Rank | Subnet | WHOIS | Hosts | Domains | Nameservers | Type |
|---|---|---|---|---|---|---|
| 1 | 184.168.221.0/24 | GoDaddy | 12,986,177 | 949,324 | 164 | Parking |
| 2 | 50.63.202.0/24 | GoDaddy | 1,345,709 | 841,660 | 163 | Parking |
| 3 | 82.98.86.0/24 | Sedo GmbH | 10,857,325 | 763,298 | 449 | Parking |
| 4 | 208.87.35.0/24 | Cable Bahamas | 2,434,091 | 588,432 | 47 | Parking |
| 5 | 81.169.145.0/24 | STRATO | 652,344 | 584,349 | 140 | Parking |
| 6 | 64.95.64.0/24 | Internap | 1,553,467 | 462,479 | 87 | Parking |
| 7 | 107.20.206.0/24 | Amazon Parking | 878,427 | 290,133 | 52 | Parking |
| 8 | 208.91.197.0/24 | Confluence Inc. | 749,258 | 238,794 | 414 | Parking |
| 9 | 208.73.211.0/24 | Oversee.net | 1,032,996 | 237,082 | 30 | Parking |
| 10 | 213.186.33.0/24 | OVH Systems | 302,838 | 233,546 | 1,555 | Parking |

Table 4.15: Top-10 subnets based on domain density.

links to any of those domains contribute to the in-degree of the subnet and boost its ranking position.

### 4.9.3 DNS Co-Hosting

In order to find ways to eliminate the ranking inflation from parking domains, we analyze in detail DNS co-hosting, which refers to multiple hosts and/or domains on the same IP. We merge DNS results of two datasets and compute the density of each /24 subnet. Table 4.14 lists the top-10 subnets based on host density. We find that a high host density does not necessarily indicate spam or low-quality content. For example, Google is hosting over 5M hosts within its subnet and most of them are from blogspot.com. Skyrock.com with over 4M hosts and Microsoft MSN with 3.9M hosts are also not spam. These blogs and social networks still provide valuable content.

There are also cases where the whole network appears to be spam. Inside Level 3 Communications, we find over 7M hosts from ustreasuryfunds.com, which is a parked domain. The spammers create millions of hosts with automatically generated keywords inside the hostnames, such as lumber-companies-okechobee-florida.ustreasuryfunds.com and gas-presure-washer.ustreasuryfunds.com. Another example is ViaWest, which is hosting 3M websites from the affiliate marketing company clickbank.net.

However, a high domain density is a strong indication of spam as shown in Table 4.15. We identify all of the top-10 subnets based on domain density as parking services. Moreover, as we manually go through the top-100 such subnets, 92% of them are classified as parking.

While the humongous amounts of hosts and domains inside parking services can potentially support each other and inflate their ranking positions, the size of the DNS infrastructures behind them is limited. Each parking service maintains a set of authoritative DNS nameservers that keep the zone records for all the domains inside the service and answer

| Rank | Subnet | WHOIS | Spam Hosts | Hosts | Domains | Type |
|---|---|---|---|---|---|---|
| 1 | 81.171.34.0/24 | Eweka Internet Services | 170 | 143,812 | 1,241 | Parking |
| 2 | 82.98.86.0/24 | Sedo GmbH | 82 | 10,857,326 | 763,298 | Parking |
| 3 | 208.91.197.0/24 | Confluence Inc. | 62 | 749,261 | 238,794 | Parking |
| 4 | 82.206.123.0/24 | TITAN-NETWORKS | 48 | 1,553,380 | 923 | Parking |
| 5 | 208.87.35.0/24 | Cable Bahamas | 45 | 2,434,091 | 588,432 | Parking |
| 6 | 208.73.211.0/24 | Oversee.net | 39 | 1,032,996 | 237,082 | Parking |
| 7 | 141.8.224.0/24 | Confluence Inc. | 34 | 643,170 | 76,750 | Parking |
| 8 | 184.168.221.0/24 | GoDaddy | 29 | 12,986,423 | 949,324 | Parking |
| 9 | 8.5.1.0/24 | eNom Inc | 19 | 521,408 | 100,536 | Parking |
| 10 | 62.116.143.0/24 | InterNetX GmbH | 17 | 1,400,191 | 170,592 | Parking |

Table 4.16: Top-10 subnets based on manual spam count.

| | IRLbot | | | ClueWeb | | |
|---|---|---|---|---|---|---|
| Algorithm | top-1K | 1K-10K | 10K-100K | top-1K | 1K-10K | 10K-100K |
| L2-D | 0.2% | 1.4% | 5.3% | 0.1% | 0.6% | 2.6% |
| L3-NS | 0.1% | 0.9% | 4.9% | 0.0% | 0.2% | 2.6% |
| L3-NS-filter | 0.0% | 0.2% | 2.0% | 0.0% | 0.1% | 0.7% |

Table 4.17: Projected fraction of spam.

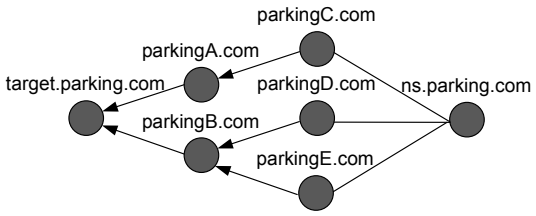



Figure 4.7: L3-NS example.

corresponding DNS queries. As illustrated in Tables 4.14-4.15, the number of nameservers is several orders of magnitude smaller than that of hosts and domains in parking services.

In order to use nameservers in our ranking framework, we first build a Domain-Nameserver DNS graph, where if a nameserver is responsible for the zone records of a domain, a link is added from the nameserver to that domain. We extend our previous method L2-D by adding the DNS graph. A natural join on three graphs, a Host-Domain graph, a Domain-Domain graph, and a Domain-Nameserver graph, gives us the chain Host-Domain-Domain-Nameserver that supports the target host. This allows us to compute the reputation score of a host by using the number of nameservers behind level-2 domains. We call this method nameserver supporters at level-3 (L3-NS). Figure 4.7 shows an example where parking domains parkingC.com, parkingD.com, and parkingE.com are created to support the target at level-2. However, since they share the same authoritative DNS nameserver ns.parking.com, the L3-NS value of the target is only boosted by one.

### 4.9.4 Spam Filter

To further remove spam hosts from the top list, we try to understand the features of top-ranked spam hosts. We briefly talk about some simple rules to filter spam since this is not our main contribution. First, we analyze if they come from any particular IP clusters. As we map all $1,601$ spam hosts manually identified in section 4.8.2 to their /$24$ subnets, we find that most of them come from a small number of sources. Table 4.16 lists the top-$10$ subnets with the most spam hosts. They are all parking services and are responsible for $545$ spam entries, or $34\%$ of the total.

Using high domain density as the main feature of parking services, we remove websites that are hosted on IPs with domain density exceeding some threshold $\alpha$. After careful analysis of the domain density on each IP, we set $\alpha = 1,000$. This removes $2,083$ IPs out of $4.9$M total. We set $\alpha$ to a relatively large number to reduce false positive since we notice that some reputable websites also register a large number of domains, which for example include typos of the original domain (e.g., amozon.com, verisgn.com) and domains under different TLDs (e.g., google.net, google.co.uk).

Another major feature we find is *hijacking*, which refers to spammers stealing links from reputable websites to support their targets. Through hijacking, in-neighbors of reputable sites become level-$2$ supporters of the target, which is a huge boost. However, hijacking does not inflate in-neighbors much since hijacking one website only boosts it by one. Thus, spam websites that benefit from hijacking reveal the following property: a small number of level-$i$ supporters generate a disproportionately large number of level-$i+1$ supporters. As an example, host www.areapropertiesllc.com is a small website for a real estate services company. Its GTR value is $0$, which indicates low reputation. However, it manages to enter the top-$100$K list of L$3$-NS with $245$K level-$3$ nameserver supporters. We find it has only $13$ domain in-neighbors with one of them being google.com, which

| | | IRLbot | | | ClueWeb | | |
|---|---|---|---|---|---|---|---|
| | | Links | Avg. deg | $C$ | Links | Avg. deg | $C$ |
| $SUPP_2$ | Host→Host | 6.8B | 69.4 | 2.54T | 1.1B | 38.0 | 627B |
| | Host←Host | 6.8B | 10.5 | | 1.1B | 9.4 | |
| L2-D | Domain→Host | 3.3B | 107.4 | 2.31T | 578M | 50.5 | 767B |
| | Domain←Domain | 1.8B | 20.8 | | 415M | 13.7 | |
| L3-NS | Domain→Host | 3.3B | 107.4 | 1.09T | 578M | 50.5 | 590B |
| | Domain←Nameserver | 1.25B | 32.7 | | 587M | 22.4 | |

Table 4.18: Comparison of computational complexity.

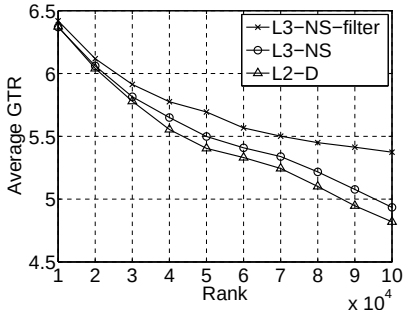
brings almost all the level-3 nameserver supporters for the target.

To filter out hijacked links, we set a minimal threshold on their number of level-2 nameserver supporters (L2-NS), which can be similarly computed by join a Host-Domain graph and a Domain-Nameserver graph. We first rank hosts by L2-NS to find the minimal number $\beta$ needed for a host to enter the top-100K list (in IRLbot, $\beta = 1099$; in ClueWeb, $\beta = 557$). Then, hosts whose L2-NS count is less than $0.5\beta$ will be filtered out from the top list of L3-NS.

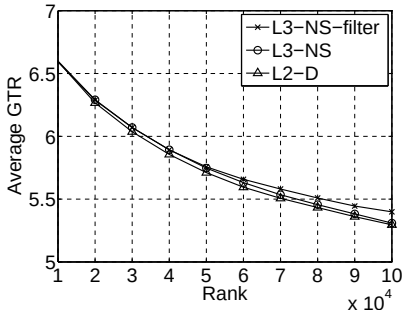
After applying the spam filter to the top list of L3-NS, we collect the new top-100K websites. We call this algorithm L3-NS-filter. While machine learning may produce other features and complex rules, we aim to keep both features and rules simple in order to maintain high efficiency since that is one of the goals for our host ranking algorithm. Note that the domain density of IPs and L2-NS counts can be easily collected during the crawl. Thus, the filter does not bring much additional overhead.

### 4.9.5 Evaluation

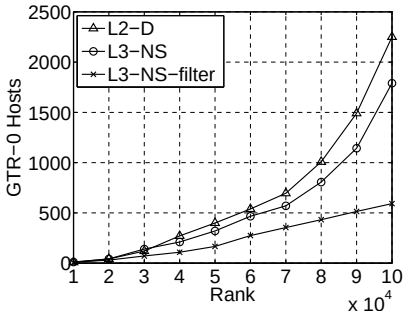
For brevity, we compare DNS based methods only with L2-D since the later is a clear winner in previous comparison. As shown in Table 4.17 for IRLbot, L3-NS finds only one spam node within its top-1K list, which is www.spywareinfo.com in position 474, while L3-NS-filter completely gets rid of spam in the same interval. From 1K to 10K, both DNS



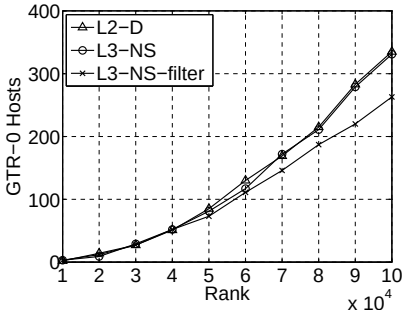

(a) IRLbot average GTR



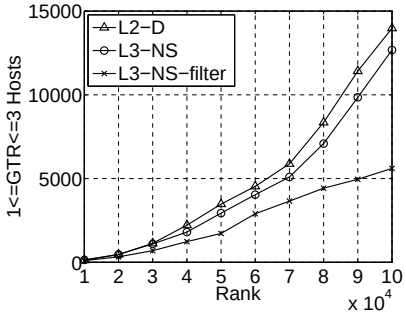

(b) ClueWeb average GTR



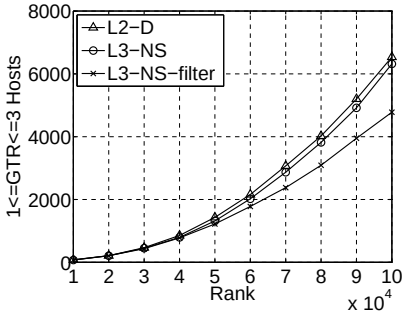

(c) IRLbot GTR-0



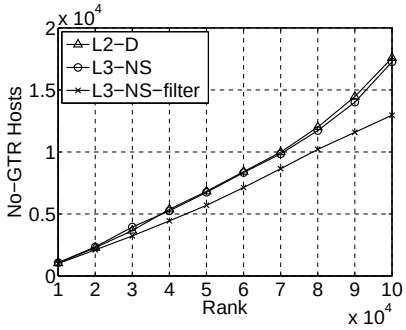

(d) ClueWeb GTR-0



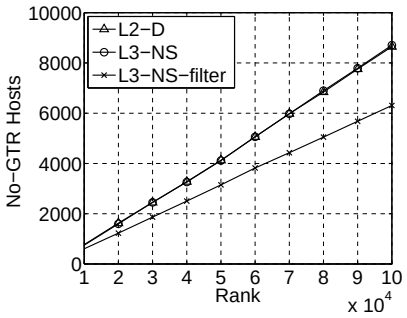

(e) IRLbot $1 <= \text{GTR} <= 3$




(f) ClueWeb $1 \leq \text{GTR} \leq 3$




(g) IRLbot no-GTR




(h) ClueWeb no-GTR

Figure 4.8: GTR analysis on top ranked hosts.

methods show improvement over L2-D, with the filter reducing spam to an impressive $0.2\%$, which is $7$ times less. From $10$K to $100$K, while the spam rate of both L2-D and L3-NS increases to around $5\%$, L3-NS-filter keeps its value at $2.0\%$, which is a $2.5$ times improvement. On ClueWeb, the results are similar.

On the GTR analysis, we find that all of the three algorithms have similar performance in their top-$10$K lists and the main difference appears between $10$K and $100$K. In order to see this clearly, Figure 4.8 plots the curves starting from $10$K and using a linear scale. In the IRLbot case, L3-NS is slightly better than L2-D, while L3-NS-filter shows a clear advantage. For the average GTR in Fig. 4.8(a), L2-D-filter ends up with an average of $5.37$ by position $100$K, while the other two algorithms all fall below $5$. In comparison of the avoidance of hosts with unreputable GTRs in (c)-(h), L3-NS-filter often shows $2$ times less unreputable hosts than the best remaining methods. In the ClueWeb case, the difference is smaller. However, L3-NS-filter is still the clear winner.

We finish the evaluation by briefly looking at the performance of the spam filter. As we check all the spam hosts manually identified in L3-NS against the top-$100$K list of L3-NS-filter, we find that over $60\%$ of them are filtered out. When looking at the GTR distribution of removed hosts, we find around $60\%$ of the them have a GTR $\leq 3$ or no GTR in IRLbot. In ClueWeb, this fraction is $33.09\%$, which is still high. While it is quite effective in removing spam and low-GTR hosts, we also see $2.22\%$ and $3.71\%$ filtered hosts with a GTR $\geq 7$, respectively on two datasets. While this potentially indicates the false positive of the spam filter, we find that many of these hosts are unpopular ones from reputable websites (e.g., jmlr.csail.mit.edu, blog.360.yahoo.com, featuresblogs.chicagotribune.com).

### 4.10 Computational Complexity

Besides the quality of top-ranked hosts, the computational complexity of the algorithm is another important aspect. Since the algorithm runs in real-time and periodically updates

the ranking list, high complexity is undesirable.

In this section, we quantify the overhead of different kinds of multi-level supporter algorithms (SUPP$_2$, L2-D and L3-NS) by the total number of link traversals in BFS. In practice, when computing L3-NS, we first transform the Domain-Domain graph to a Domain-Nameserver graph by replacing the domain in-neighbors with their nameservers (note the difference between this graph and the DNS graph we showed earlier). Then, L3-NS is computed by joining the Host-Domain graph and the Domain-Nameserver graph. Thus, its complexity is similarly computed as other level-2 methods. The general structure behind a level-2 supporter algorithm is $Host \leftarrow i \leftarrow j$, and its computation requires a reverse BFS to depth 2. Given $i$'s out-degree $A_i$ and its in-degree $B_i$, the total number of links to be visited through node $i$ is $A_iB_i$. Suppose there are $n$ nodes at level-1, the total overhead is $C = \sum_{i=1}^{n} A_iB_i$. Both $n$ and the average degree at level-1 affect complexity.

Table 4.18 lists the number of links in each single graph and the complexity $C$ of each algorithm. By grouping hosts into domains, we reduce the number of links; however, the average degree also increases. For example, in IRLbot, the number of links in the first graph reduces from $6.8$B to $3.3$B, while the average degree grows from $69.4$ to $107.4$. In the second graph, the number of links reduces from $6.8$B to $1.8$B, but the average degree doubles from $10.5$ to $20.8$. That is why Table 4.18 shows complexity $C$ staying the same (IRLbot) or slightly increasing (ClueWeb) from SUPP$_2$ to L2-D.

In L3-NS, we see improvement on both datasets. To be specific, L3-NS reduces the overhead by $57\%$ compared to SUPP$_2$ for IRLbot, which is quite significant. For ClueWeb, this improvement is $6\%$.

## 4.11 Conclusion

In this chapter, we formalized the problem of topological ranking and showed that traditional techniques that worked a single infinite graph often failed to produce good ranking

lists. We solved this problem from a novel angle and proposed a framework ranking on multiple and finite graphs. Based on our framework, a set of new techniques could be created. Our two examples using domains and nameservers were proved to be resistant to inflation. Additionally, by examining two sizable crawl datasets, we showed that our framework worked well in multiple diverse graphs. It is well known that PageRank has been manipulated by spammers ever since it was published. Spam farms are built to boost PageRank scores of target pages. Our method is arguably more resistant to manipulation than PageRank. Even if the algorithm is learned by spammers, it will still be difficult to get a large number of domains and nameservers under control for them to compete with highly reputable sites.

Future work involves using our ranking methods to drive a real web crawler.

# 5. SUMMARY AND FUTURE WORK

## 5.1 Summary

This dissertation was motivated by the growing needs of processing large-scale data in many research areas. Our approach is to analyze classic algorithms, understand their cost and shortcomings, and make them suitable for dealing with modern data size. To be specific, the dissertation addressed two problems, triangle listing and neighborhood function, which have been researched for many years and a lot of important applications have been explored.

### 5.1.1 PCF

Although plenty of research has been done to solve the triangle listing problem, our effort in modeling triangle listing cost has shown that much of the previous work involves significant amount of redundant computation and is far from optimal. Moreover, similarities between prior studies suggest that they can potentially be unified under a single framework. Our work creates such a framework that exhausts $6$ different triangle search orders and $18$ triangle listing algorithms. Then, a novel algorithm called PCF is proposed to handle the external-memory operation of all $18$ algorithms. Through efficient pruning of companion files, PCF achieves favorable I/O complexity compared to existing methods. Our I/O model suggests PCF as the first algorithm that can achieve linear I/O complexity in certain cases. In the evaluation, PCF beats its closest competitors by a significant margin. More importantly, we see PCF efficiently handles very large graphs, e.g., billions of nodes and hundreds of billions of edges, which demonstrates its huge potential to process even larger graphs as the data size keeps growing in the future.

### 5.1.2 Trigon

Motivated by a recent proposal from Pagh etc., we try to seek better graph partitioning schemes that can further optimize the I/O cost of existing methods Pagh and PCF. We first take an investigation into the properties of the two methods, model their I/O cost, understand their shortcomings, and shed light on the conditions under which each method defeats the other. Our analysis shows that there is no definite winner among these two methods. Their comparison results can be flipped over as conditions change. Learned from the comparison, we propose a novel algorithm called Trigon that consistently beats both previous methods in all cases. Trigon utilizes 2D graph partitioning scheme and sequential coloring, which enables better I/O complexity and faster CPU processing in list compress, intersection, and hash table lookups. Our evaluation shows that Trigon consistently beats both previous methods in I/O and runtime in all graphs that we use.

### 5.1.3 Neighborhood Function

We show that exact computation of neighborhood function at depth 2 is a similar problem to triangle listing. It is more difficult as it requires duplicate elimination among depth-2 neighbors. The good news is that our developed techniques for triangle listing, e.g., graph relabeling and external-memory partitioning, can be generally applied to solving this problem as well. We next demonstrate an application of neighborhood function in ranking of Internet hosts. We propose the use of finite Internet resources in ranking that efficiently eliminates trivial inflation by spam. We also look at web-hosting services and identify it as a main source of spam hosts. Then, domain density of IP subnets is used to detect web hosting and punish hosts originated from there. With all the techniques, we finally show that our ranking method brings much less spam into its top ranking list compared to PageRank and TrustRank.

## 5.2 Future Work

The work presented in this dissertation demonstrates how our techniques can be used in large-scale graph processing. There is still plenty of additional research that can be done from our current position.

### 5.2.1 Triangle Listing

Future work includes better models for the I/O complexity of PCF and Trigon. Since closed-form formulas are difficult to derive, we only show several I/O upper bounds. Experiments suggest that these upper bounds can be quite loose and there is still room for improvement. Also, we observe that curves of the upper bounds and the actual I/O often stay in parallel. The conjecture is that the actual I/O differs with the upper bounds by only a constant factor. Additional investigation is needed to verify the conjecture. In order to further test the scaling of our methods, we would like to build larger graphs. The largest graph available to us is the IRLbot web graph, which comes from our web crawler and consists of $43$B nodes and $390$B edges. Another direction is to study the applications of triangle listing. With this efficient tool to compute triangles, many data mining problems can be potentially better solved. For example, triangles can be potentially used in graph ranking, spam detection, and network measurement etc.

### 5.2.2 Neighborhood Function

This dissertation focuses on neighborhood function at depth $2$ as an effective metric for ranking purposes. In the future, more depth can be explored for many other applications. This problem then becomes similar to external-memory breath-first search (BFS). Both CPU and I/O cost will be extremely high when reaching out to more depth. While it is possible to repeat our depth-$2$ algorithm multiple times to compute neighborhood function at larger depth, more efficient algorithms are needed to make this problem actually feasi-

ble. As for host ranking, we initially study it for frontier prioritization in web crawlers. The ranking will help web crawlers to decide which content has high quality and should be explored first. Thus, the next step is to actually use our ranking methods in real crawlers.

Besides the two problems studied in this dissertation, our novel techniques in graph relabeling, orientation, and external-memory partitioning can be potentially applied to solving other graph-mining problems, e.g., listing four-node cycles and other structures in a graph, computing network coefficient, and measuring graph diameter etc.

# REFERENCES


[1] N. Alon, R. Yuster, and U. Zwick, "Finding and Counting Given Length Cycles," in *Proc. ESA*, 1994, pp. 354–364.

[2] N. Alon, R. Yuster, and U. Zwick, "Finding and Counting Given Length Cycles," *Algorithmica*, vol. 17, no. 3, pp. 209–223, Mar. 1997.

[3] 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.

[4] R. Baeza-Yates, C. Castillo, M. Marin, and A. Rodriguez, "Crawling a Country: Better Strategies than Breadth-First for Web Page Ordering," in *Proc. WWW*, May 2005.

[5] 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.

[6] V. Batagelj and M. Zaveršnik, "Short Cycle Connectivity," *Elsevier Discrete Mathematics*, vol. 307, no. 3-5, pp. 310–318, Feb. 2007.

[7] 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.

[8] L. Becchetti, C. Castillo, D. Donato, S. Leonardi, and R. Baeza-Yates, "Link-based Characterization and Detection of Web Spam," in *Proc. AIRWeb*, Aug. 2006.

[9] L. Becchetti, C. Castillo, D. Donato, S. Leonardi, and R. Baeza-Yates, "Using Rank Propagation and Probabilistic Counting for Link-Based Spam Detection," in *Proc. WebKDD*, Aug. 2006.

[10] A. A. Benczur, K. Csalogany, T. Sarlos, and M. Uher, "Spamrank – Fully Automatic Link Spam Detection," in *Proc. AIRWeb*, May 2005.

[11] J. W. Berry, B. Hendrickson, R. A. LaViolette, and C. A. Phillips, "Tolerating the Community Detection Resolution Limit With Edge Weighting," *Physical Review E*, vol. 83, no. 5, p. 056119, May 2011.

[12] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "UbiCrawler: A Scalable Fully Distributed Web Crawler," *Software: Practice & Experience*, vol. 34, no. 8, pp. 711–726, Jul. 2004.

[13] P. Boldi, M. Rosa, and S. Vigna, "HyperANF: Approximating the Neighbourhood Function of Very Large Graphs on a Budget," in *Proc. WWW*. ACM, Mar. 2011, pp. 625–634.

[14] J. Caverlee and L. Liu, "Countering Web Spam with Credibility-Based Link Analysis," in *Proc. ACM PODC*, Aug. 2007, pp. 157–166.

[15] N. Chiba and T. Nishizeki, "Arboricity and Subgraph Listing Algorithms," *SIAM J. Comput.*, vol. 14, no. 1, pp. 210–223, Feb. 1985.

[16] J. Cho, H. Garcia-Molina, and L. Page, "Efficient Crawling through URL Ordering," in *Proc. WWW*, Apr. 1998, pp. 161–172.

[17] S. Chu and J. Cheng, "Triangle Listing in Massive Networks and Its Applications," in *Proc. ACM SIGKDD*, Aug. 2011, pp. 672–680.

[18] Y.-J. Chung, M. Toyoda, and M. Kitsuregawa, "A study of link farm distribution and evolution using a time series of web snapshots," in *Proc. AIRWeb*, 2009, pp. 9–16.

[19] ClueWeb09, "ClueWeb09 Dataset," http://www.lemurproject.org/clueweb09/.

[20] J. Cohen, "Graph Twiddling in a MapReduce World," *Computing in Science & Engineering*, vol. 11, no. 4, pp. 29–41, Jul.-Aug. 2009.

[21] Y. Cui, D. Xiao, and D. Loguinov, "On Efficient External-Memory Triangle Listing," in *Proc. IEEE ICDM*, Dec. 2016.

[22] Y. Cui, D. Xiao, and D. Loguinov, "IRL Triangle Datasets and Code," Sep. 2016. [Online]. Available: http://irl.cs.tamu.edu/projects/motifs/.

[23] R. Dementiev, "Algorithm Engineering for Large Data Sets," Ph.D. dissertation, Universität des Saarlandes, 2006.

[24] N. Eiron, K. S. McCurley, and J. A. Tomlin, "Ranking the Web Frontier," in *Proc. WWW*, May 2004, pp. 309–318.

[25] Facebook, "Facebook stats," https://newsroom.fb.com/company-info/.

[26] G. Feng, T.-Y. Liu, Y. Wang, Y. Bao, Z. Ma, X.-D. Zhang, and W.-Y. Ma, "AggregateRank: Bringing Order to Web Sites," in *Proc. ACM SIGIR*, Aug. 2006, pp. 75–82.

[27] D. Fetterly, M. Manasse, and M. Najork, "Spam, Damn Spam, and Statistics: Using statistical analysis to locate spam web pages," in *Proc. WebDB*, 2004, pp. 1–6.

[28] 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.

[29] 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.

[30] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed Graph-parallel Computation on Natural Graphs," in *Proc. USENIX OSDI*, 2012, pp. 17–30.

[31] P. Gupta, V. Satuluri, A. Grewal, S. Gurumurthy, V. Zhabiuk, 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.

[32] Z. Gyöngyi, P. Berkhin, H. Garcia-Molina, and J. Pedersen, "Link Spam Detection Based on Mass Estimation," in *Proc. VLDB*, 2006, pp. 439–450.

[33] Z. Gyöngyi and H. Garcia-Molina, "Web Spam Taxonomy," in *Proc. AIRWeb*, May 2005, pp. 39–47.

[34] Z. Gyöngyi, H. Garcia-Molina, and J. Pedersen, "Combating Web Spam with TrustRank," in *Proc. VLDB*, Aug. 2004, pp. 576–587.

[35] A. Harth, "Billion Triples Challenge Data Set," 2009. [Online]. Available: http://km.aifb.kit.edu/projects/btc-2009/.

[36] J. Hirai, S. Raghavan, H. Garcia-Molina, and A. Paepcke, "WebBase: A Repository of Web Pages," in *Proc. WWW*, May 2000, pp. 277–293.

[37] T. Hocevar and J. Demsar, "A Combinatorial Approach to Graphlet Counting," *Bioinformatics*, vol. 30, no. 4, pp. 559–565, Feb. 2014.

[38] X. Hu, Y. Tao, and C. Chung, "Massive Graph Triangulation," in *Proc. ACM SIGMOD*, Jun. 2013, pp. 325–336.

[39] X. Hu, M. Qiao, and Y. Tao, "Join Dependency Testing, Loomis-Whitney Join, and Triangle Enumeration," in *Proc. ACM PODS*, May 2015, pp. 291–301.

[40] H. Inoue, M. Ohara, and K. Taura, "Faster Set Intersection with SIMD Instructions by Reducing Branch Mispredictions," *PVLDB*, vol. 8, no. 3, pp. 293–304, Nov. 2014.

[41] A. Itai and M. Rodeh, "Finding a Minimum Circuit in a Graph," *SIAM Journal on Computing*, vol. 7, no. 4, pp. 413–423, 1978.

[42] M. Ivory and M. Hearst, "Statistical Profiles of Highly-Rated Web Sites," in *Proc. CHI*, Apr. 2002, pp. 367–374.

[43] 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.

[44] 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.

[45] H. Kwak, C. Lee, H. Park, and S. Moon, "Twitter Graph," 2010. [Online]. Available: http://an.kaist.ac.kr/traces/WWW2010.html.

[46] 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.

[47] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale Graph Computation on Just a PC," in *Proc. USENIX OSDI*, 2012, pp. 31–46.

[48] 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.

[49] H.-T. Lee, D. Leonard, X. Wang, and D. Loguinov, "IRLbot: Scaling to 6 Billion Pages and Beyond," *ACM Trans. Web*, vol. 3, no. 3, pp. 1–34, Jun. 2009.

[50] 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.

[51] 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.

[52] B. Menegola, "An External Memory Algorithm for Listing Triangles," Universidade Federal do Rio Grande do Sul, Tech. Rep., 2010.

[53] 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.

[54] M. Najork and A. Heydon, "High-Performance Web Crawling," Compaq SRC, Tech. Rep. 173, Sep. 2001. [Online]. Available: http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-173.pdf.

[55] 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.

[56] A. Ntoulas, M. Najork, M. Manasse, and D. Fetterly, "Detecting spam web pages through content analysis," in *Proc. WWW*, 2006, pp. 83–92.

[57] M. Ortmann and U. Brandes, "Triangle Listing Algorithms: Back from the Diversion," in *Proc. ALENEX*, Jan. 2014, pp. 1–8.

[58] J. Padmanabhan, "Introduction to Grid Computing via Map-Reduce & Hadoop," http://internationalnetworking.iu.edu/sites/internationalnetworking.iu.edu/files/indo-us-preso.pdf, Dec. 2010.

[59] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web," Stanford Digital Library Technologies Project, Tech. Rep., Jan. 1998. [Online]. Available: http://dbpubs.stanford.edu:8090/pub/1999-66.

[60] R. Pagh and F. Silvestri, "The Input/Output Complexity of Triangle Enumeration," in *Proc. ACM PODS*, Jun. 2014, pp. 224–233.

[61] C. R. Palmer, P. B. Gibbons, and C. Faloutsos, "ANF: A Fast and Scalable Tool for Data Mining in Massive Graphs," in *Proc. ACM SIGKDD*, 2002, pp. 81–90.

[62] H. Park and C. Chung, "An Efficient MapReduce Algorithm for Counting Triangles in a Very Large Graph," in *Proc. ACM CIKM*, Oct. 2013, pp. 539–548.

[63] H. Park, F. Silvestri, U. Kang, and R. Pagh, "MapReduce Triangle Enumeration With Guarantees," in *Proc. ACM CIKM*, Nov. 2014, pp. 1739–1748.

[64] 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.

[65] D. A. Patterson, "Latency Lags Bandwidth," *Communications of the ACM*, vol. 47, no. 10, pp. 71–75, Oct. 2004.

[66] Google Toolbar Rank. [Online]. Available: http://toolbar.google.com/.

[67] 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.

[68] B. Schlegel, T. Willhalm, and W. Lehner, "Fast Sorted-Set Intersection using SIMD Instructions," in *Proc. ADMS*, Sep. 2011.

[69] 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.

[70] V. Shkapenyuk and T. Suel, "Design and Implementation of a High-Performance Distributed Web Crawler," in *Proc. IEEE ICDE*, Mar. 2002, pp. 357–368.

[71] J. Shun and K. Tangwongsan, "Multicore Triangle Computations without Tuning," in *Proc. IEEE ICDE*, Apr. 2015, pp. 149–160.

[72] A. Signhal, "Breakfast with Google's Search Team," https://www.youtube.com/watch?v=8a2VmxqFg8A, Aug. 2012.

[73] C. Sparkman, H.-T. Lee, and D. Loguinov, "Agnostic Topology-Based Spam Avoidance in Large-Scale Web Crawls," in *Proc. IEEE INFOCOM*, Apr. 2011, pp. 811–819.

[74] S. Suri and S. Vassilvitskii, "Counting Triangles and the Curse of the Last Reducer," in *Proc. WWW*, Mar. 2011, pp. 607–614.

[75] J. Tomlin, "A New Paradigm for Ranking Pages on the World Wide Web," in *Proc. WWW*, May 2003.

[76] N. H. Tran, K. P. Choi, and L. Zhang, "Counting Motifs in the Human Interactome," *Nature Communications*, vol. 4, p. 2241, Aug. 2013.

[77] 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.

[78] D. J. Watts and S. Strogatz, "Collective Dynamics of 'Small World' Networks," *Nature*, vol. 393, pp. 440–442, Jun. 1998.

[79] S. Webb, J. Caverlee, and C. Pu, "Predicting web spam with HTTP session information," in *Proc. CIKM*, 2008, pp. 339–348.

[80] 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.

[81] S. Wernicke and F. Rasche, "FANMOD: A Tool for Fast Network Motif Detection," *Bioinformatics*, vol. 22, no. 9, pp. 1152–1153, Feb. 2006.

[82] V. Willians and R. Williams, "Subcubic Equivalences Between Path, Matrix, and Triangle Problems," in *Proc. IEEE FOCS*, Oct. 2010, pp. 645–654.

[83] B. Wu and K. Chellapilla, "Extracting link spam using biased random walks from spam seed sets," in *Proc. AIRWeb*, 2007, pp. 37–44.

[84] D. Xiao, Y. Cui, D. Cline, and D. Loguinov, "On Asymptotic Cost of Triangle Listing in Random Graphs," in *Proc. ACM PODS*, May 2017.

[85] D. Xiao, Y. Cui, D. B. Cline, and D. Loguinov, "On Asymptotic Cost of Triangle Listing in Random Graphs," Texas A&M University, Tech. Rep. 2016-9-2, Sep. 2016. [Online]. Available: http://irl.cs.tamu.edu/publications/.

[86] S. Yadav, A. K. K. Reddy, A. N. Reddy, and S. Ranjan, "Detecting algorithmically generated malicious domain names," in *Proc. ACM IMC*. ACM, 2010, pp. 48–61.

[87] Yahoo Altavista Graph, 2002. [Online]. Available: http://webscope.sandbox.yahoo.com/catalog.php?datatype=g.

[88] 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.