EFFICIENT AND SCALABLE LISTING OF FOUR-VERTEX SUBGRAPHS

A Thesis

by

XIANGZHOU XIA

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

MASTER OF SCIENCE

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

May 2016

Major Subject: Computer Science

ABSTRACT

Identifying four-vertex subgraphs has long been recognized as a fundamental technique in bioinformatics and social networks. However, listing these structures is a challenging task, especially for graphs that do not fit in RAM. To address this problem, we build a set of algorithms, models, and implementations that can handle massive graphs on commodity hardware. Our technique achieves $4 - 5$ orders of magnitude speedup compared to the best prior methods on graphs with billions of edges, with external-memory operation equally efficient.

To my mother and wife

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr Loguinov, for all help and guidance that he has given me over the past two years. He has set an example of excellence as a researcher, a mentor and instructor. From him, I learned how to do critical thinking, how to be proud of myself and the importance of passion. All those things have shaped my life. The experience of working with him will guide me in my future career. I would also like to thank Dr. Bettati and Dr. Reddy for serving on my committee.

I would like to extend my thanks to my colleagues at Internet Research Lab, to Yi Cui for inspiring this work, to Xiaoyong Li and Zain Shamsi for encouraing me keep move forward, to Tanzir Ahmed, who helped me fix server issues. It is my fortune to work with these wonderful guys.

Finally, I gratefully acknowledge my mother and wife for their everlasting love and support.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

CHAPTER I

INTRODUCTION AND LITERATURE REVIEW

1.1 Introduction

Assume an undirect graph $G = (V, E)$ have $n = |V|$ vertices and $m = |E|$ edges. A subgraph of G is a graph whose vertices are a subset of V, and whose edges are a subset of the E. Suppose a k-vertex subgraph have exact k vertices. Then, a small subgraph, also known as graphlet and motif, is a subgraph of G with a small k (usually $1 < k < 10$). For bioinformatics people, the interest of counting small subgraphs start from [34]. The report [34] defined "network motifs" to be small subgraphs that more likely to occur in real world graph than randomized graph. The authors of this report find "network motifs" in networks from biochemistry, neurobiology, ecology, and engineering. After this report, this technique have been widely used in biology [32], [41] include protein function prediction, network alignment, and phylogeny [27], [33], [53]. The study of graphlets counting in social network have a long history [10], [13], [18]. Other application of motif listing include Computing network [3], [8], [16], Chemoinformatics [24], [43], Image segmentation [64], [65], Machine learning [11], [35], [45], [46], [60].

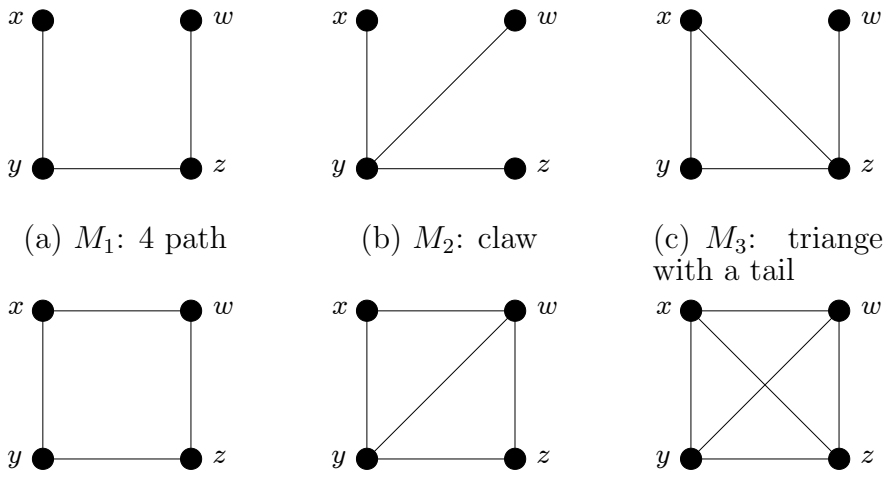
There are two directions in the research of small subgraphs listing. One direction is listing general small subgraphs with a vary k like paper mentioned above. Another direction is listing subgraphs with a fix k. Small subgraphs listing with $k = 3$(triangle listing) and $k = 2$(densest subgraph) are well studied problems. Both of those two problems have efficient algorithms and fluent applications. The importance of 4-vertex subgraphs listing in social media is introduced by [59] which find social network have

Graph	Nodes n	Edges m
Twitter	41,652,230	2,405,026,390
Yahoo	720,242,173	12,869,122,070
PLD	86,534,416	3,416,273,404
WebUK	62,338,347	1,877,431,056

Table I. Real world graph.

more motifs than random graph. They also find that in event classification problem, 4-vertex subgraph is a better feature than triangle. [59] shows that use 4-vertex motif as feature achieve higher precision than triangle in event classification problem. 4-node graphlets frequency is used as a feature in Twitter recommendation system [15]. The ego 3-profile problem introduced by [7] which have applications on spam detection and generative models can be transfer to 4-vertex subgraph counting. As 4-vertex subgraph become more and more important, many algorithms have been proposed to improve the performance of listing 4-vertex subgraphs. However, there is no efficient and scalable solution exist. With single thread, the state-of-the-art exact listing algorithm [1] takes nearly 6 hours to finish a graph with 4.7 millions edges. Since real world graph always have billions of edges as present on Table I, current solution need years to finish process those graphs. Another problem of real world graph is that they can not fit in RAM. For instance, the size of PLD graph is 16 GB and the size of Yahoo graph is 53 GB. All previous work assume the graph is fit in RAM and there is no external solution exist. So, there is no algorithms that able to handle those real world graphs.

A graph is connected when there is a path between every pair of vertices. Previous research [30], [22] only consider 4-vertex subgraphs that are connected. Recently, the paper [1] investigated the general 4-vertex subgraph listing algorithms which include unconnected 4-node patterns. The results of [1] shows that the overhead of listing unconnected 4-vertex subgraph is minor compared to connected 4-vertex sub-



(a) M_1: 4 path (b) M_2: claw (c) M_3: triange with a tail

(d) M_4: 4 cycle (e) M_5: 4 cycle with a chord (f) M_6: 4 clique

Fig. 1. Types of 4-vertex subgraph.

graph listing. So we focus on connected 4-vertex subgraph. As shown in Figure 1, there are in total 6 patterns of connected 4-vertex subgraph. There are two types of counting strategies which called induced and non-induced subgraph counting. The difference between these two strategy occur when there are some overlap between patterns. For example, a 4-clique will also contains a 4-cycle. There are two ways to handle this problem. An induced subgraph is a subset of V together with any edges whose endpoints are both in this subset. For a 4-clique, we count it only as a 4-clique if we are counting induced subgraphs. If we count a 4-clique both as one 4-clique and six 4-cycle, we are counting non-induced subgraphs. Our algorithm should be able to count both of them. There exist a formula to transfer non-induced 4-vertex subgraphs counts to induced 4-vertex subgraphs counts. So we first listing non-induced subgraphs and transfer it to induced counts later. We denote the number of induced ith subgraph by C_i (Figure 1). We also denote the number of non induced ith subgraph by N_i. The formula is given by:

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 2 & 4 \\ 0 & 1 & 2 & 4 & 6 & 12 \\ 0 & 0 & 1 & 0 & 4 & 12 \\ 0 & 0 & 0 & 1 & 1 & 3 \\ 0 & 0 & 0 & 0 & 1 & 6 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{bmatrix} = \begin{bmatrix} N_1 \\ N_2 \\ N_3 \\ N_4 \\ N_5 \\ N_6 \end{bmatrix}$$

1.1.1 Contribution

Our first contribution is applied Relabeling and Orientation which are well known techniques in triangle listing to 4-vertex subgraph listing. After apply those two techniques, we avoid calculate redundancy and decreased the operation number in our algorithm. Our second contribution is proposed a series of algorithm that highly improved the bottleneck of previous work. We achieved over 50,000x speedup in a real world graph with 3.6 billion edges. Our third contrition is build the first external memory solution for 4-vertex subgraph counting. Our external-memory solution has basically the same CPU overhead with our in-memory algorithms while it require minor IO. Our fourth contrition is build a set of models to estimate the overhead of our algorithm. Our model match with the experiment results.

1.2 Related Work

Most previous research is interest in general cases of motif counting. Paper [61], [62] propose a system called Fanmod which use BFS to counting small subgraphs. Fanmod need 574 minutes to compute a graph with 75k edges. The paper [17] propose a exact counting algorithm called ORCA for counting small subgraphs. To listing four-node graphlets, Orca will first listing all 3-vertex subgraphs and search any

possible vertex that can form 4-node graphlets with them. Orca is several hundred times faster than Fanmod in graphs with thousand of vertices. [31] propose a exact counting algorithm called acc-Motif. To counting k-vertex subgraph, Acc-Motif will listing $(k-2)$-vertex subgraph and form k-vertex subgraph by them. Acc-Motif achieved 40x speedup from Fanmod in small graphs. The algorithm proposed by [40] is called SGIA-MR which is basically DFS on Map-Reduce. They use a cluster have 64 machines. For a graph with 394,552 edges and 11,920 vertices, SGIA-MR require 260 sec. [29] propose a subgraph listing mapreduce algorithm called TwinTwigJoin. In experiment, they use a cluster with 15 nodes. Each of the computing nodes has one 3.47 GHz Intel Xeon CPU with 6 cores and 12 GB memory. For a graph with 3 million vertices and 12 million edges, TwinTwigJoin require 10 minutes to counting 4-cycles and 1 minutes to counting 4-cliques. There are also many motif listing research in bioinformatics area [5], [6], [14], [23], [25], [28], [36], [44], [49], [63]. The paper [9] proposed a algorithm to counting k-clique using mapreduce. They method require $O(m^{3/2})$ total space and $O(m^{k/2})$ work. However, those algorithms only scale to thousands of vertices.

Some research focus on graphlets that involve 3 vertices. There are fluent research target to estimating triangle numbers [20], [50], [51]. There are also fluent research aim to find exact triangle listing algorithm [37], [38], [39], [47], [52], [54], [57]. Early research on triangle listing only work for graph that fit in memory. Recent years, some work make triangle listing for massive graph possible. Among those work, Trigon is the state-of-the-art triangle listing algorithm. Trigon investigated the impact of total order (degree order or vertex id order). Trigon also proposed external-memory solution for triangle listing.

There are some research target at exact counting 4-vertex motifs. The paper [30] present a series algorithms called RAGE to exact 4-vertex subgraph. In RAGE,

the time complexity of counting 4-cycle and 4-clique are $O(d|E|) + O(|E|^2)$ while other motifs at most need $O(d|E|)$ (d is the average degree and $|E|$ is the number of vertex). So The bottleneck of RAGE are 4-cycle and 4-clique counting. In our test, RAGE 4-cycle algorithm have higher time complexity than RAGE 4-clique. RAGE is being treat as baseline in following 4-vertex exact counting papers. Authors of this paper only test their algorithms on one graph that have 26,561 nodes and 92,584 edges, RAGE finish this graph in 40 minutes while Fanmod need nearly 3 hours. In our experiment, we implement RAGE and achieved at least 10x speedup compare to it's original implementation. We do not investigate the reason for the speedup. [1] extend the patterns of 4-vertex subgraphs. They aim to listing the general case of 4-vertex include 4-vertex subgraphs that are not connected. [1] also proposed the state-of-the-art 4-vertex listing algorithm. However, the system proposed by [1] do not improve the bottleneck of RAGE since it's 4-cycle and 4-clique counting methods are exactly the same with RAGE. The authors of [1] claim their algorithm is 460 times faster than RAGE. A important reason for the improvement is that PGD have parallelization while RAGE only have one thread. Another improvement of PGD is sort the graph by degree order which is a standard technique in triangle listing. [1] do not investigate how the performance of 4-vertex motif listing is effect by Relabeling order. So PGD keep the order choice to be a open question. We tried different graph order in our tests. The results shows that the PGD can at most be $2\times$ faster no matter which order we choose. This indicate that the improve of PGD that contribute by relabeling is small. In our tests, PGD have similar performance with RAGE when PGD run with original graph order and only have single thread.

Since exact counting 4-node graphlets is very time consuming, some research start focus on investigate estimation algorithm. Algorithms in [4], [12], [42], [55], [56], [58] aims to estimate frequencies of general small subgraphs. For specific 4-vertex sub-

graph estimation, Path Sampling proposed by [21] has the best performance. There are several drawbacks of estimation algorithm. The first drawback is that sampling algorithm always have high error bounds. The second drawback is that only motif counting is possible for estimation algorithm while there is no sampling algorithm that able to listing motifs.

1.3 Overview

In following section, we propose our in-memory and external-memory solutions for counting or listing 4-vertex subgraphs. Since M_1, M_2, and M_3 are very to count, we just use the algorithm from RAGE. The algorithm to listing M_1, M_2, and M_3 have maximum time complexity that is equal to triangle listing. Our work is focus on M_4, M_5, and M_6 which are discussed on following sections. We use M_i to represent ith type of motif in Figure 1. We present both in-memory and external-memory solution for each motif. We also derive a series of accurate models for our algorithms. Suppose v_i have X_i out-neighbors, Y_i in-neighbors, and d_i undirect neighbors. We also assume that T_i to be number of triangles belong to v_i. We investigate the techniques to improve basic operation performance. We focus on merging and intersection since they are majority operations in our algorithms. Table II shown the benchmark of every operation appear in our algorithms.

1.3.1 Preprocessing

For each type of 4-vertex subgraph $xyzw$, there may be k nodes that are equivalent. For instance, $k = 4$ in 4-cycle. Therefore, it is possibly listed $k!$ times according to different permutations of the node sequence (x, y, z, w). In order to eliminate such duplicates and improve efficiency, our goal is to list all 4-vertex subgraph *uniquely*

	Implementation	Speed (M/s)
Random Access, 32 bit key		135
Sorting, 64 bit key	STL Sort	6
Intersection, 32 bit key	Scalar(CPU)	246
	SIMD	1119
	SIMD with compression	1801
	SIMD with counters	1001
Merging, 32 bit key	Scalar(CPU)	221
	SIMD	1110
	SIMD with compression	1720

Table II. Benchmark of operation speed.

in certain order $\mathcal{O}$. This goal can be achieved by standard techniques from triangle listing [2], [3], [26], [52]. There are 3 steps in such techniques. The first step, which we call *relabeling*, will sort the nodes by $\mathcal{O}$ and sequentially assign IDs from sequence (1,2,...,n). The most common relabel order in triangle listing is random and descending-degree. Trigon proves that descending order have the best performance in triangles listing. In our experiment, descending order have best performance in our algorithms. Follow the definition in Trigon, we use G_θ to denote the relabeled graph. The second step, which we call *orientation*, will scan G_θ and split each undirect neighbor lists $N(y)$ into in-neighbor lists $N^-(y)$ and out-neighbor lists $N^+(y)$. Suppose in-graph G_θ^- and out-graph G_θ^+ is generated after orientation. Also suppose node i have out-degree X_i, in-degree Y_i, and total degree $d_i = X_i + Y_i$ in directed graph $G_\theta^* = (V, E_\theta^*)$. The third step will list triangles with node IDs in either ascending or descending order. Due to it's good performance, *relabeling* and *orientation* is default in our algorithms.

CHAPTER II

PROBLEM AND SOLUTIONS

2.1 Four-Cycle

Listing 4-cycles is the bottleneck in prior work [1], [30]. To address this problem, we propose a series of novel algorithms that significantly improve upon the existing methods.

2.1.1 RAGE

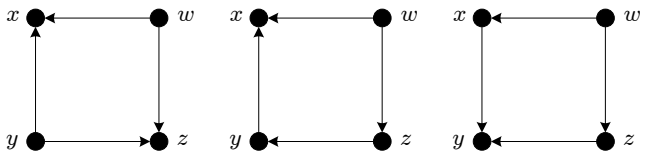
To find 4-cycle in Figure 1, RAGE will first iterates over each edge. Then, for a edge (y, z) $(y > z)$, RAGE will store all $z's$ neighbors in a hashtable. Third, RAGE will iterates over $y's$ neighbors. For $y's$ neighbor x, RAGE will check whether $x's$ neighbors in hashtable, each hit find a 4-cycle. RAGE will count 4-cycle 4 times Since same process will apply to every edge of a 4-cycle.

There are two major type of operations in this algorithm. The first type is hash table insert and the second type is hash table lookup. Since the undirect neighbors of the smaller vertex of a edges will insert to hash table, $v_i's$ undirect neighbors will be insert into hash table Y_i times. So the overhead of hash table insert is:

$$\sum_{i=1}^{n}(Y_i d_i) \tag{1}$$

For every out-neighbors v_i have, all the neighbors of neighbors of v_i will lookup hash table once. So the hash table lookup is:

$$\sum_{i=1}^{n}(X_i(\sum_{j=1}^{d_i} d_{ij})) \tag{2}$$



(a) $x < z < y$ (b) $x < y < z$ (c) $y < x < z$

Fig. 2. Types of directed 4-cycles.

Algorithm 1 Intersection-Based 4-Cycle(V, E)

```
1: for w ∈ V do
2:     for x ∈ N⁺(w) and z ∈ N⁺(w) and x < z do
3:         for y ∈ N(x) ∩ N(z) and y < w do
4:             output 4-cycle (x, y, z, w)
```

The final overhead is given by:

$$c_n(RAGE, \theta_n) = \sum_{i=1}^{n} (Y_i d_i + X_i (\sum_{j=1}^{d_i} d_{ij}^{+})) \tag{3}$$

2.1.2 Intersection-Based (IB)

For each 4-cycle $\square_{xyzw}$, its four nodes are equivalent. Therefore, it is possibly listed $4! = 24$ times according to different permutations of the node sequence (x, y, z, w). In order to eliminate such duplicates and improve efficiency, our goal is to list all 4-cycles *uniquely* in certain order $\mathcal{O}$, which is achieved with relabeling and orientation. After orientation, the directions of edges in a 4-cycle is shown in Figure 2. By fixing w as the pivot node, we list all 4-cycles that have w as the largest node. The directions of edges (w, x) and (w, z) are determined because $w > x$ and $w > z$. Without loss of generality, assume $x < z$, we discover 3 possible orders of the node sequence (x, y, z), which correspond to the three types of 4-cycles in Figure 2.

Based on the above observations, we propose our M_4-listing algorithm based on intersection. As shown in Algorithm 1, each source node is taken as w, candidate

pairs (x, z) are picked from $N^+(w)$ and follow the order $x < z < w$. Finally, the intersection discovers possible nodes y in all three cases. In order to enforce w as the largest node, the intersection should stop at w; otherwise duplicate 4-cycles are listed. This implies the intersection involves all out- neighbors and a portion of the in-list up to hash y. The overhead is then

$$C_n(IB, \theta_n) = \sum_{i=1}^{n} \sum_{j=1}^{(X_i-1)} \sum_{k=j+1}^{X_i} (d_{ij}^+(i) + d_{ik}^+(i)). \tag{4}$$

which can be simplified to

$$\begin{aligned} C_n(IB, \theta_n) &= \sum_{i=1}^{n} \sum_{j=1}^{(X_i-1)} \left(d_{ij}^+(i)(X_i - j) + \sum_{k=j+1}^{X_i} d_{ik}^+(i) \right) && (5) \\ &= \sum_{i=1}^{n} \left(\sum_{j=1}^{(X_i-1)} d_{ij}^+(i)(X_i - j) + \sum_{j=2}^{X_i} (d_{ij}^+(i)(j-1) \right) && (6) \\ &= \sum_{i=1}^{n} \left(\sum_{j=2}^{(X_i-1)} d_{ij}^+(i)(X_i - 1) + d_{i1}^+(i)(X_i - 1) + d_{i,X_i}(X_i - 1) \right) && (7) \\ &= \sum_{i=1}^{n} (X_i - 1) \sum_{j=1}^{X_i} d_{ij}^+(i) && (8) \end{aligned}$$

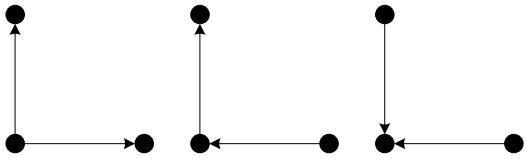
Theorem 1. *the overall complexity of IB is given by:*

$$C_n(IB, \theta_n) = \sum_{i=1}^{n} (X_i - 1) \sum_{j=1}^{X_i} d_{ij}^+(i). \tag{9}$$

Since every common node in the intersection discovers only one 4-cycle, the number of intersection operations is no less than the number of 4-cycles. Moreover, many intersection scans are useless because no common nodes are found. Table III shown that IB only achieve 2.3x speedup compare to RAGE. We next seek for algorithms that break the limit by performing fewer operations and discovering multiple 4-cycles per operation.

Method	Operations
RAGE	30 Q
IB	12.9 Q
UWT	123 T
DWT without relabeling	10.7 T
DWT	0.51 T

Table III. Operation numbers of different algorithm in Twitter.



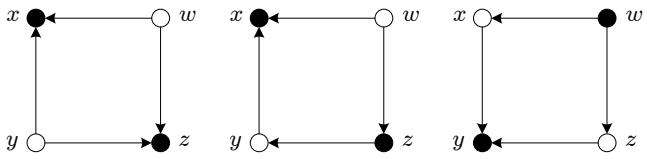
(a) T_1 (b) T_2 (c) T_3

Fig. 3. Types of directed wedges.

2.1.3 Undirected Wedge Traveler (UWT)

Nodes x, y, z form a wedge pivoting at y if $(x, y), (y, z) \in E$. We call (x, z) end nodes. We notice that if two wedges (x, y, z) and (x, w, z), where $y \neq w$, exist in G, a 4-cycle $\square_{xyzw}$ is discovered. So, 4-cycle can be listed by match wedges. Besides choosing y and w as the pivots, $\square_{xyzw}$ can be also discovered by match (y, x, w) and (y, z, w). Therefore, every 4-cycle will be listing twice if we listing 4-cycle by match wedges. Based on above observations, we propose a new algorithm called undirected wedge traveler (UWT).

UWT first iterates over each node and generates all pairs of end nodes. Then, UWT match the wedges by sorting them and grouping end nodes together. If k wedges have the same end nodes, any pair of wedges choose from those k wedges can form 4-cycle. So, we can count $\binom{k}{2}$ 4-cycles when k duplicate is detected. Finally, since there are two ways to decompose a 4-cycle, its total number should be divided by 2. For 4-cycle counting, we only need to sort end-node pair. However, the pivot nodes need to be attached with the end-node pairs if we want to list each 4-cycle. For



(a) T_1+T_1 (b) T_1+T_2 (c) T_2+T_2

Fig. 4. Directed 4-cycles with T_1 and T_2 wedges.

each source node i with degree d_i, a total number of $\binom{d_i}{2}$ wedges pivoting at i can be generated.

Theorem 2. *the wedges generated by UWT is given by:*

$$w_n(UWT) = \sum_{i=1}^{n} \frac{d_i(d_i - 1)}{2}. \tag{10}$$

Since wedges need to be sorted and linear scanned to match. the actual complexity of this algorithm is:

$$c_n(UWT) = w_n(UWT) \log_2 w_n(UWT) + w_n(UWT) \tag{11}$$

UWT is able to update multiple 4-cycles in a single operation. This potentially improves the performance upon IB. As shown on Table III, the operations of UWT is 100x less than IB. Although the improvement of UWT is significant, sorting 123 trillion wedges is unpractical. Based on the sorting speed on Table II, UWT need 237 days to finish Twitter graph.

2.1.4 Directed Wedge Traveler (DWT)

In According to Figure 2, there are three types of oriented wedges (x, y, z), which we call T_1-T_3. Their directions are illustrated in Figure 3. Note that wedge (x, w, z) is always T_1 since w is the largest node. Therefore, the three types of 4-cycles can be viewed as the combinations of T_1+T_1, T_1+T_2, and T_1+T_3. However, if we consider

x and z as the pivot nodes, there is another way to decompose these three types of 4-cycles: T_3+T_3, T_2+T_3, and T_2+T_2.

After analyzing the patterns of directed 4-cycles, we find that T_1 and T_2 wedges are enough to construct all possible cases. We illustrate this finding in Figure 4, where the pivot nodes are marked as hollow cycle. For each source node i as the pivot node, the number of T_1 wedges generated by UWT is:

$$c(T_1, \theta) = \sum_{i=1}^{n} \frac{X_i(X_i - 1)}{2}. \tag{12}$$

The number of T_2 pairs generated by UWT is:

$$C(T_2, \theta) = \sum_{i=1}^{n} X_i Y_i. \tag{13}$$

Therefore, after dropping T_3 wedges, we get the following result.

Theorem 3. *The total number of wedges generated by DWT is given by:*

$$w_n(DWT, \theta_n) = c(T_1, \theta) + C(T_2, \theta). \tag{14}$$

We found that C_3 is exactly the same with the CPU complexity of edge iterator in Trigon. The authors of Trigon investigate the impact of permutation to operation numbers and prove that descending-degree order is indeed optimal for edge iterator. Therefore, descending-degree order is the best permutation for DWT. Skipping T_3 wedges significantly reduces the overhead for two reasons: 1) under descending-degree permutation, it is possible to transform the majority of wedges to T_3 and avoid them altogether. Since the total number of wedges C_2 is fixed, the rest T_1 and T_2 wedges are minor; 2) by using only two types of wedges, every 4-cycle is constructed in an unique way and thus listed only once.

Consider pair sorting and matching, the final complexity is:

$$c_n(DWT, \theta_n)c(T_1, \theta)(\log_2 c(T_1, \theta) + 1) \\ +c(T_2, \theta)(\log_2 c(T_2, \theta) + 1) \tag{15}$$

Table III shows that DWT achieve 12x speedup by dropping T_3 wedges. DWT achieve another 20x speedup by apply optimal permutation.

2.1.5 Implementing Directed Wedge Traveler

Now we discuss the implementation of DWT. The obvious method (DWT1) is to store all wedges (x,z) in RAM and then sort them using std::sort. The problem is that the number of wedges in real world graph is too large to fit in RAM. For instance, Twitter have around 510 billion T_1 and T_2 wedges. Suppose a wedge need 8 byte, DWT1 will require 4TB space to store wedges. To solve this problem, we can split wedges into blocks and only store the wedges of one block in RAM.

We split the nodes into ω disjoint sets $V_1, V_2, \ldots, V_\omega$, where

$$\bigcup_{k=1}^{\omega} V_k = V \text{ and } V_i \cap V_j = \emptyset \text{ for } i \neq j. \tag{16}$$

For each set V_j, we only store wedges whose smallest end-point falls into V_j. The number of wedges generated from V_j is:

$$W_j = \sum_{i=1}^{n} \sum_{k \in V_j} (X_i - k + Y_i) \tag{17}$$

When $k = 1$ and $\max |V_j| = 1$, W_j achieve the minimum

$$\sum_{i=1}^{n} (X_i - 1 + Y_i) = m - n \tag{18}$$

Therefore, the minimum RAM require by DWT1 is $2m - n$. To achieve load

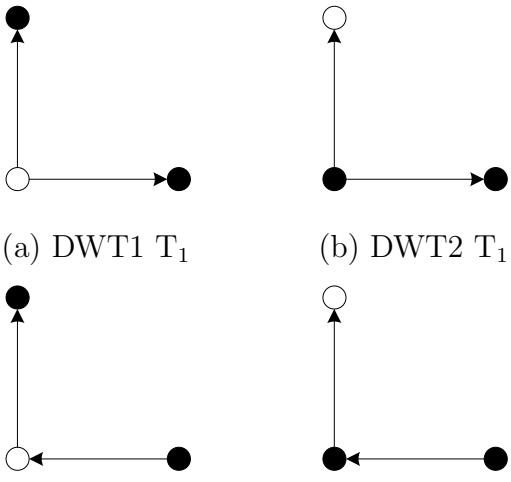
Algorithm 2 VertexSplitting(V, E^*)

Initialize array W to 0
for $i = 1$ **to** n **do**
  for x_{ij} as the j-th out-neighbor of i **do**
    $W[x_{ij}] = W[x_{ij}] + X_i - j + Y_i$
$S = 0$
$j = 1$
for $i = 1$ **to** n **do**
  if $S + W[i] > M$ **then**
    $j = j + 1$
    $S = 0$
  $S = S + W[i]$
  $V_j = V_j \cup \{i\}$

balance in vertex splitting, we propose Algorithm 2.

The next method (DWT2) generates wedges from the smallest node instead of the one in the middle. Assume x is the smallest node in a 4-cycle, Figure 5 compares wedges generated from DWT1 and DWT2, where the pivot node is marked in hollow cycle. Again, we only consider T_1 and T_2.

Fixing the smallest node x as the pivot, we only need to generate the end point z instead of the pair (y, z) for each wedge, which saves 50% memory. Moreover, this allows us to process each pivot node x separately, which only requires the memory to hold all end points z for a given x. DWT2 operates by holding G in RAM, scan G_θ^-, visiting each node x, jumping to its in-neighbors y_{x1},y_{x2},..., grab it's undirected lists of neighbors larger than x as the candidate end points z. Note that the undirected neighbors $N(y)$ are sorted already. Therefore, instead of globally sorting all end



(a) DWT1 T_1 (b) DWT2 T_1

(c) DWT1 T_2 (d) DWT2 T_2

Fig. 5. Comparison of DWT1 vs DWT2.

Method	Operations
Actual merge overhead	7 T
search without start point	151 B
search with start point	1.2 B

Table IV. Benchmark of start point in Twitter.

points for x, DWT2 just needs to merge Y_x sorted lists. Finally, in the sorted list of end points, a node z that appears k times indicates $\binom{k}{2}$ 4-cycles. The total number of end points that need to be processed is the same as C_3.

DWT2 requires a hash table that keeps for each $y \in V$ the following: a) the offset in the G buffer of its neighbor list; b) out-degree; c) in-degree, and d) the start point from which we search for x in y's neighbors. The search can be linear, binary, or interpolation. Note that when the start point exceeds the out-degree, there is no need to perform the search - all y's in-neighbors are automatically larger than x. Since x are processed in increasing order, the start point moves forward monotonically.

We now examine how fast DWT2 works on Twitter with and without the start point. In the latter case, the number of nodes scanned should be (focusing on the

middle node y)

$$w_n(DWT_2^{'}, \theta_n) = \sum_{i=1}^{n} \sum_{j=1}^{Y_i} d_{ij}^{-} = \sum_{i=1}^{n} \sum_{j=1}^{X_i} d_i = \sum_{i=1}^{n} (X_i Y_i + X_i^2) \quad (19)$$

and in the former

$$w_n(DWT_2, \theta_n) = \sum_{i=1}^{n} \sum_{j=1}^{Y_i} (d_{ij}^{-} - d_{ij}^{-}(i)) = \sum_{i=1}^{n} \sum_{j=1}^{X_i} (d_i - j) = \sum_{i=1}^{n} (X_i Y_i + \frac{X_i(X_i - 1)}{2}) \quad (20)$$

On Twitter, (19) yields 0.66T and (20) produces 0.51T(i.e., they differ by $\sum X_i(X_i+1)/2$=151B). Converting cost to binary comparisons, these become

$$c_n(DWT_2^{'}, \theta_n) = \sum_{i=1}^{n} \log_2 Y_i \sum_{j=1}^{Y_i} d_{ij}^{-} \quad (21)$$

and

$$c_n(DWT_2, \theta_n) = \sum_{i=1}^{n} \log_2 Y_i \sum_{j=1}^{Y_i} (d_{-}^{ij} - d_{ij}^{-}(i)) \quad (22)$$

which are 8.7T and 7T, respectively. This means that scanning for x improves total comparison cost by 20% in this graph. *Note that this model fails to account for elimination of duplicates in the merge tree* (the current implementation does not do this anyway due to difficulties in coding the counter into SIMD merge).

Searching for x sequentially from the beginning of each list requires 151B extra

scalar comparisons. Note that this number is much smaller than the un- necessary merge overhead (i.e., 1.7T) because the latter is repeated at every level of the merge tree. If we use the start position, the scan overhead drops to 1.2B (Table IV). It appears that none of the suggested searches (i.e., SSE, binary, interpolation) would help improve the runtime due to the already negligible number of comparisons (i.e., 1.2B/7T = 0.017%). To save RAM, we could also eliminate the start position and live with a 151B/7T = 2.1% increase in cost. Since scalar search is slower than SSE merge, this will likely lead to a 6-8% increase in runtime.

For DWT2, RAM must be large enough to store two copies of the longest set of wedges for a given node x. This would be the sum of partial out-degree across all in-neighbors of x. Specially, suppose the j-th in-neighbor of i has $X_{ij}^{-}(i)$ out-neighbors with labels larger than i. Then, RAM usage is

$$2 \max_{i} \sum_{j=1}^{Y_i} (d_{ij}^{-} - d_{ij}^{-}(i)). \tag{23}$$

For Twitter, the largest list occupies 1.4 GB and thus the method requires 2.8 GB to be the merge buffer.

2.1.6 SIMD Merging

We also explored SIMD merging to increase merging speed. Similar to intersection, SIMD merging achieved 4x speedup compared to CPU-based merging. The SIMD merging algorithm used in our implementation is from [19]. The paper [19] use 128-bit vector registers to perform 4x4 and 8x8 32-bit integer merging. We adopt 8x8 merging since 8x8 merging out-perform 4x4 merging in our experiments.

We try to improve merging speed by applying graph compression. However, our experiments shows that the graph compression do not increase the speed of merging.

	RAGE	UWT	DWT
PLD	140 Q	1.4 T	1.4 T
Twitter	30 Q	0.51 T	0.51 T
Yahoo	39.7 T	0.43 T	0.43 T

Table V. Operation numbers of M_4 listing algorithms.

Algorithm	Operation Type
RAGE	Random Access
DWT1	Sorting
DWT2	Merging

Table VI. Operation types of M_4 listing algorithms.

For the merging order, the naive method is binary merging which is easy to implement. However, we can improve merging performance by apply optimal merging. Optimal merging always merge two lists with the shortest length. This algorithm is performed by maintaining a min heap with sorted lists as nodes. The key of the min heap is the length of each sorted list. After merging two lists popped from min heap, we re-insert the result list into the heap. The improvement of optimal merging is depend on graph. Optimal merging achieve 10% speedup compare to binary merge in PLD graph. In PLD graph, optimal merging have 17.5 trillion operations while binary merging have 19 trillion operations.

2.1.7 Runtime Estimation

The models derived in previous section is able to compute the operation number of each algorithm. Table V shown the operation number of DWT and RAGE in PLD, Twitter, and Yahoo graph. Since PGD have the same CPU complexity with RAGE,

	RAGE	DWT1	DWT2
PLD	33 years	2.7 days	5 hours
Twitter	7 years	1 day	2 hours
Yahoo	9.3 years	20 hours	1.7 hours

Table VII. Estimated runtime of M_4 listing algorithms.

Algorithm 3 G^*_θ Partition

```
1: size = 0
2: k = 1
3: for y = 1 to n do
4:     if d_y + size > M then
5:         k = k + 1
6:         size = 0
7:     size+ = d_y
8:     V[k].add(y)
```

we only use RAGE as the baseline for compare. Based on operation type in Table VI and operation speed in Table II, we can estimate the runtime of RAGE and Falcon (Table VII). From Table VII, we can see that DWT have significant speedup from RAGE. DWT1 is 2-4 magnitude faster than RAGE. DWT2 achieve 12-13x speedup compare to DWT1.

2.1.8 Partitioning

We build external solution for 4-vertex subgraph listing. One fundamental technique in our external solution is graph partition. Since we need to partition G^+_θ and G^*_θ in following section. We present the algorithm to partition G^+_θ and G^*_θ.

The partitioning algorithm in Falcon is similar to graph partition algorithm of

Trigon. We start from split G_θ^+ and then extend to split (G_θ^-, G_θ^+). The first step of split G_θ^+ is divide V into w set $V_1..., V_w$ that satisfy 16.

Each direct edge in G_θ^+ that point to a vertex in V_k will be put into $G_\theta^+(k)$. Based on this strategy, we will iterate over vertices y in G_θ^+ and divide it's out-neighbor list to s_y^+ chunks.

The strategy to split G_θ^* is basically the same with G_θ^+ except we consider both in-neighbor lists and out-neighbor lists. Each direct edge in G_θ^+ that point to a vertex in V_k and each direct edge in G_θ^- that start from a vertex in V_k will be put into $G_\theta^*(k)$. Every vertex $y's$ direct neighbor list will divide into s_y^* chunks.

One problem of graph is load balancing. To maximum the usage of RAM, each subgraph generated by partitioning algorithm should as close to available RAM size as possible. So, given available RAM to be M, the optimal partition strategy should produce subgraphs that all have size M. Simply even distribute vertex to V_i will not satisfy this requirement. Since graph is sort by degree, most edges are belong to small source nodes. So, even distributed vertex will lead to unbalance of subgraph size. To achieve optimal partition for G_θ^*, we propose Algorithm 3. The basic idea of this algorithm is iterate over vertex y in increase (or decrease) order, check whether $X_y + Y_y$ still allow current partition to fit in RAM. If fit, put y in V_k; If not, put y in V_{k+1} and increment k. For G_θ^+ partition, we only need replace d_y with Y_y.

2.1.9 External Memory

The external solution is straightforward - partition G (using techniques present in section) as in Trigon into subgraphs $G(1)$, $G(2)$,..., then run DWT2 using pairs $(G_\theta^-, G_\theta(k))$, where the latter is stored in RAM and the former is scanned from disk. The amount of information in the hash table can be 25% less: a) the offset each neighbor list in the $G(k)$ buffer; b) total degree; and c) the start point. The hash

	8.0 GB	4.0 GB	2.0 GB	1.0 GB
PLD	19.9 GB	26.5 GB	39.7 GB	59.5 GB
Twitter	16 GB	22.2 GB	34.4 GB	58.9 GB
Yahoo	274.1 GB	489.4 GB	920.1 GB	1781.6 GB

Table VIII. IO complexity of DWT on real world graph.

table determine the minimum RAM require by our external solution. In our partition algorithm, the largest $|G_\theta(k)|$ will be at least $\max d_i$. Since hash table need 12 byte for keys and 4 byte for value, each vertex require at least 16 bytes. In Yahoo, the minimum RAM size is 122 MB which is small enough for most situations.

The I/O overhead is $O(m^2)$ using simplified analysis. Ignoring the wedge array, there are $= m/M$ partitions, where M is RAM size in nodes. Since G_θ^- whose size is $m/2$ will be read w times, the read overhead is $m^2/(2M)$. Combine with the read overhead of $\sum |G_\theta(k))|$=m, we get the following result.

Theorem 4. *The I/O overhead of DWT is given by:*

$$H(DWT, \theta) = \frac{m^2}{2M} + m. \tag{24}$$

Table VIII shows the I/O complexity of DWT works on real graphs.

2.1.10 Experiment Setup

We use a single server to perform our experiments. This server has a six-core Intel i7-3930K 4.4 GHz processor, Asus Rampage IV Extreme motherboard, quad-channel DDR3 RAM @ 2133 MHz, and a RAID systerm capable of I/O at 1 GB/s. Falcon is compare to previous solutions include RAGE and PGD. RAGE do not have parallelization while Falcon and PGD have. To make experiments more fair, all the algorithms are running on a single thread.

Graph	n	m	M_4	RAGE-O	RAGE-N	PGD *reg*	PGD *des*	Falcon 1.0	Falcon 2.0
Amazon0302	260,000	1,240,000	2.5 M		0.84	1.08	0.6	0.5	0.08
Amazon0505	410,000	3,360,000	36 M		9.8	6.48	4.26	2.2	0.4
Cit-Patents	3,370,000	16,500,000	342 M		55	84.3	48.54	24.8	3.6
web-Google	880,000	5,100,000	540 M		118	53.64	68.82	4.2	0.76
web-Stanford	280,000	2,300,000	13 B		605	476.7	500.94	2.9	0.42
web-WikiTalk	2,390,000	5,020,000	2 B		20,532	4069.74	2127.84	32	3.6
web-Youtube	1,140,000	2,990,000	469 M		453	144.6	136.68	6.7	0.9

Table IX. Benchmark of M_4 for small graphs.

	Motifs	RAGE-O	PGD *des*	DWT1	FDWT2
PLD	518,205,999,844,451	-	-	3.8 days	4.8 hours
Twitter	466,222,645,290,726	-	-		2.1 hours
Yahoo	6,402,519,185,064,717	-	-		1.9 hours

Table X. Benchmark of M_4 on large graphs.

The first set of graphs we use are standard datasets from previous works. We download all those graphs from SNAP (Stanford Network Analysis Project) website. Since the graphs in first set have max millions of edges, we call them small graphs. Small graphs can be load on RAM in commodity servers. To verify our solution can scale to billions of edges, we also used Twitter, Yahoo and a new graph called PLD graph. The vertex and edge number of those graph can be found on Table I.

2.1.11 DWT Runtime

Table IX presents the runtime of 4-cycle counting on small graphs. We assume that every small graphs can load into RAM. We can see that the performance of PGD and RAGE are very close on every graph. It prove that PGD is RAGE with parallelization. We notice that despite the fact that all those graphs have similar amount of vertices and edges, the runtime of 4-cycle counting is quite different. The

	8.0 GB	4.0 GB	2.0 GB	1.0 GB
PLD	52 min	55 min	1 hour	1.3 hour

Table XI. Runtime of DWT2 when RAM is limit.

reason is that the structure of a graph decide the time complexity of 4-cycle counting. More dense the graph, more run-time need for 4-cycle counting. The speedup rate is increase when the runtime of a graph is rise. This prove that falcon have lower time complexity than PGD and RAGE. Falcon 1.0 is several hundred times faster than PGD *des* in densest graph (Wikitalk). DWT2 achieve 6-11x speedup compare to DWT1. M_4 counting is the biggest bottleneck in PGD and RAGE. However, in our solution, the runtime M_4 counting is close to M_6 which is a huge improvement.

Table X presents the runtime of DWT and RAGE on PLD graph. RAGE can not finish 4-cycle counting. The runtime of DWT is close to our estimation which proves that our estimation mechanism is reasonable. Due to our estimation, RAGE need 33 years to finish 4-cycle counting. For DWT2, we have around 360,000x speedup of M_4 counting on PLD graph. Our solutions is able to efficient counting 4-vertex motifs on graphs with billions of edges.

Since there is no previous external solution for 4-vertex counting exist, we only compare our external solution with our in memory solution. Table XI presents the runtime of our external solution on different RAM limit. We can see that runtime is increase when available RAM is decrease. However, the performance decrease is on a reasonable range. When we only have 1 GB RAM, the runtime is not double for all the motifs. The performance of Falcon-D prove that our external solution is efficient.

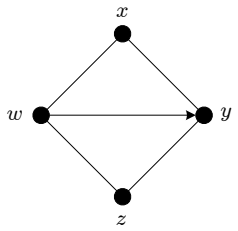


Fig. 6. 4-cycle with a chord.

2.2 Four-Cycle With a Chord

Although M_5 has one more edge compared to M_4, it is actually much easier to list M_5 in terms of complexity because the additional edge (i.e., the chord) serves as the pivot edge that helps locate the corresponding M_5. In contrast, without the chord, M_4 requires BFS explore to at least depth 2, which is quite expensive.

2.2.1 RAGE

This method takes each edge (x, y) and intersects $N(x)$ with $N(y)$, both undirected lists. If $|N(x)N(y)| = c$, RAGE increments the motif counter for (x, y) by $c(c1)/2$. To enforce a single visit through each edge, we can use implicit orientation $y > x$. Note that RAGE and PGD both use array-based hash tables of size n. In that case, the cost is

$$c_n(RAGE, \theta_n) = \sum_{i=1}^{n} d_i^2, \tag{25}$$

2.2.2 Streaming Intersector (SI)

We use Trigons approach to intersection (i.e., SSE) and utilize a directed graph with some θ_n. M_5 is easily located by its chord. Without loss of generality, assume $y < w$, the direction of the chord is illustrated in Figure 6. Fixing the chord, any two common neighbors, e.g., x and z, of the chord's incident nodes y and w indicate

Algorithm 4 M_5 Counting(V, E)

```
C = 0
for y ∈ V do
    for w ∈ N⁻(y) do
        k = |N(y) ∩ N(z)|
        C = C + (k choose 2)
return C
```

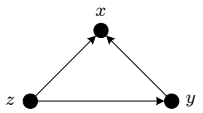


Fig. 7. Directed triangle with $x < y < z$.

a M_5, where the common neighbors can be easily computed with an intersection. Since there are no specific requirements of the directions of the other four edges, the intersection is applied to the undirected neighbors of y and w. This leads to our first version of M_5 counting algorithm as shown in Algorithm 4. The intersection at each chord (y, w) can be done in $d_y + d_w$ time.

Theorem 5. *This intersection overhead in Algorithm 4 is given by:*

$$C_n(SI) = \sum_{(y,w)\in E^*} (d_y + d_w) = \sum_{i=1}^{n} d_i^2. \tag{26}$$

which is 246T on Twitter.

2.2.3 Edge Loader (EL)

The complexity $C_n(SI)$ is a bottleneck in large graphs. To overcome this, we notice that listing M_5 is similar to listing triangles. The idea here is to use Trigon to find all cycles in G and attach a counter to every edge that indicates the number of

Algorithm 5 M_5 Counting 2.0(V, E^*)

setup a counter $C_{xy} = 0$ for each edge (x, y)
$C = 0$
for $z \in V$ **do**
    for $y \in N^+(z)$ **do**
        for $x \in N^+(y) \cap N^+(z)$ **do**
            $C_{xy} = C_{xy} + 1$
            $C_{xz} = C_{xz} + 1$
            $C_{yz} = C_{yz} + 1$
for $(x, y) \in E^*$ **do**
    $C = C + \binom{C_{xy}}{2}$
return C

triangles the edge belongs to. After Trigon finishes, we iterate over edges and compute the number of motifs they participate in. Specifically, if the counter is $c >= 2$, the number of motifs for the edge is $c(c1)/2$; otherwise, it is zero. Based on this idea, we next leverage the state-of-art triangle listing algorithm to facilitate our M_5 listing process. After orientation, the directions of edges in a triangle is illustrated as Figure 7. Without loss of generality, assume $x < y < z$, we set the edge $y \leftarrow z$ as the pivot edge (marked in bold). The other node x is then discovered by intersection the out-neighbors of y and z. Once a triangle is detected, update the counter of its three edges. This process is illustrated in Algorithm 5. By applying intersection only on the out-neighbors, the overhead is significantly reduced. The intersection overhead of this algorithm is the same with E_3 in Trigon.

	RAGE	SI	EL
PLD	83 T	83 T	1.4 T
Twitter	246 T	246 T	0.51 T
Yahoo	56.4 T	56.4 T	0.43 T

Table XII. Operation numbers of M_5 listing algorithms.

Theorem 6. *This intersection overhead in Algorithm 5 is given by:*

$$c_n(EL) = \sum_{i=1}^{n} \left(\frac{X_i(X_i - 1)}{2} + X_i Y_i \right). \tag{27}$$

2.2.4 SIMD Intersection

SIMD intersection can achieve a significant speedup compare to CPU-based scalar intersection. We use the technique in [48] which use 128-bit vector registers to intersect 32-bit or 16-bit integers. In our experiment, SIMD intersection is 4 times faster than CPU-based intersection.

While the vertex ID in our graph is 32-bit ID, we can achieve better intersection performance by compress them to 16-bit. The compression is done by group vertices into chunks by the upper 16 bits. For each chunk, we store the length and a list of lower 16-bit integers. Compression will double the performance of SIMD intersection and decrease the graph size by 50%. So, our SIMD intersection in total achieve 8x speedup compare to CPU-based Scalar intersection. This feature is apply on FR_5^2.

In FR_5^2, we need intersect lists that have counter attach to each element. Since there is no previous work have this feature, we proposed a novel SIMD intersect algorithm that able to solve this problem. While counters double the size of lists, our SIMD-intersection-with-counters achieved 59% performance of original SIMD intersection in experiment. Figure 6 shown the code of SIMD intersection with counters.

Algorithm	Operation Type
RAGE	Intersection & Random Memory Access
SI	Intersection
EL	Intersection

Table XIII. Operation types of M_5 listing algorithms.

	RAGE	SI	EL
PLD	3.9 days	15 hours	36 min
Twitter	11.5 days	55 hours	11.2 min
Yahoo	2.6 days	12.6 hours	9.4 min

Table XIV. Estimated runtime of M_5 listing algorithms.

2.2.5 Runtime Estimation

The models derived in previous section is able to compute the operation number of each algorithm. Table XII shown the operation number of SI, EL, and RAGE in PLD, Twitter, and Yahoo graph. Since PGD have the same CPU complexity with RAGE, we only use RAGE as the baseline for compare. Based on operation type in Table XIII and operation speed in Table II, we can estimate the runtime of SI, EL, and RAGE (Table XIV). From Table XIV, we can see that our solution have significant speedup from RAGE. SI is 2-4x faster than RAGE. EL achieve 30-250x speedup compare to SI.

2.2.6 External Memory

The external solution of M_5 listing is the same with Trigon-D except each vertex in our algorithm have a counter. This counter will double both intersection overhead and I/O overhead of Trigon-D. Following Trigon-D, the intersection overhead of our

	8.0 GB	4.0 GB	2.0 GB	1.0 GB
PLD	66.2 GB	105.8 GB	198.2 GB	
Twitter				
Yahoo	107.6 GB	121.8 GB	139.8 GB	163 GB

Table XV. IO complexity of EL on real world graph.

algorithm is:

$$H_{cpu}^{intersect} = 2 * \sum_{i=1}^{n} (\frac{X_i(X_i - 1)}{2} + X_i Y_i)$$
$$= \sum_{i=1}^{n} (X_i(X_i - 1) + 2X_i Y_i) \quad (28)$$

The hash table lookup overhead is the same with Trigon-D:

$$H_{cpu}^{lookup} = \sum_{i=1}^{n} (s_i^* X_i) \quad (29)$$

I/O complexity in our algorithm is different with Trigon-D. The first step of our algorithm is triangle listing. When we process $(G_\theta^c(k), G_\theta^+(k))$, C_{xy} and C_{xz} is in RAM while C_{yz} not. Therefore, we need write the C_{yz} to disk. A counter file that have the

Graph	n	m	M_5	RAGE-O	RAGE-N	PGD *reg*	PGD *des*	SI	EL
Amazon0302	260,000	1,240,000	3 M		0.84	0.84	0.42	0.09	0.05
Amazon0505	410,000	3,360,000	45 M		1.90	3	1.62	0.41	0.24
Cit-Patents	3,370,000	16,500,000	84 M		11.99	25.14	11.4	3.83	2.00
web-Google	880,000	5,100,000	621 M		13.26	11.76	13.02	1.56	0.27
web-Stanford	280,000	2,300,000	9 B		43.22	63.18	73.44	3.98	0.07
web-WikiTalk	2,390,000	5,020,000	1.4 B		172.81	459.66	451.92	3.82	0.86
web-Youtube	1,140,000	2,990,000	252 M		17.85	15.72	14.4	2.37	0.28

Table XVI. Benchmark of M_5 in small graphs.

	Motifs	RAGE-O	PGD *des*	SI	EL
PLD	336 T	5.2 days		17 hours	355 sec
Twitter	103,341,853,863,470				201.5 sec
Yahoo	1,920,147,871,975,643				

Table XVII. Benchmark of M_5 on large graphs.

	8.0 GB	4.0 GB	2.0 GB	1.0 GB
PLD	388 sec	410 sec	443 sec	525 sec
Twitter				
Yahoo				

Table XVIII. Runtime of EL when RAM is limit.

same size with $G^c_\theta(k)$ will created. Then, the I/O complexity in this step is:

$$\sum_{k=1}^{w} 2|G^c_\theta(k)| = \sum_{i=1}^{n} 2s_iY_i \tag{30}$$

After we finish listing triangles, we merge all the counter files. The I/O complexity of in this step is:

$$\sum_{k=1}^{w}(|G^c_\theta(k)| + |G^+_\theta(k)|) = \sum_{i=1}^{n} s_i(2Y_i + X_i) \tag{31}$$

Table XV shows the I/O complexity of EL works on real graphs.

2.2.7 SI and EL Runtime

The benchmark of M_5 counting are show on Table XVI. We can see that the performance of PGD and RAGE are also very close on every graph. For both M_5 counting, SI achieved several hundred times speedup from PGD *des* in densest graph (Wikitalk). Also, EL is several times faster than SI in all graphs. Table XVII presents

the runtime of Falcon and RAGE on PLD graph. Since RAGE do not have external memory solution, we load the whole PLD graph into RAM. We finished RAGE on M_5 counting, the runtime is close our estimation. It proves that our estimation mechanism is reasonable. In Twitter, EL achieve 3750x speedup compare to RAGE. Table XVIII presents the runtime of external EL on different RAM limit. We can see that runtime is increase when available RAM is decrease. However, the performance decrease is on a reasonable range. The performance of our external memory solution prove that our external solution is efficient.

2.3 Four-Clique

2.3.1 RAGE

For each found $\square_{yzwx}$, where $y \to z$ is the pivot edge, RAGE checks existence of $(w, x) \in E$. Specifically, it pushes $S'_{yz} = N(y) \cap N(z)$ into a table and then for all $x \in S'_{yz}$ performs a lookup for all $w \in N(x)$. The push adds 1 extra memory hit per triangle, each of which is processed three times, leading to

$$c_n(RAGE) = \sum_{i=1}^{n}(d_i^2 + \sum_{j=1}(d_i + d_j + d_k + 3). \tag{32}$$

This number is 32Q on Twitter. RAGE lists each clique six times, treating each possible edge as the chord. Duplicate elimination for this algorithm is still a major headache.

2.3.2 Triangle Composer (TC)

Similar to M_5, M_6 can also be listed using triangle techniques. Since the nodes (x, y, z, w) in a M_6 are identical to each other, we can assume $w < x < y < z$ without losing generality. After orientation, the directions of edges in a 4-clique is shown in

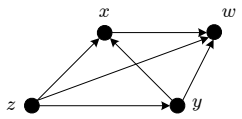


Fig. 8. Directed 4-clique with $w < x < y < z$.

Figure 8, where the largest node z has 3 out-links, the smallest node w has 3 in-links, and the other two nodes y and x have 2 out-links, 1 in-link and 1 out-link, 2 in-links. Note that the case is different from M_4, where the four nodes are actually *not* identical to each other. For the example in Figure 1(d), in the perspective of w, nodes x and z are its neighbors, whereas y is not. Therefore, we cannot simply assume the order between y and x, z, which leads to the three cases in Figure 2. From this point of view, we claim that M_6 is also a simpler problem than M_4.

As shows on Algorithm 7, the first idea is to use E_1 to discover all triangle $\triangle_{xyz}$, where y is the pivot node and $z > y$ is fixed. This produces an intersection $S_{yz} = N^+(y) \cap N^+(z)$, which we save into another array. We then continue through all nodes x in S_{yz} and compute the intersection of their out-neighbors $N^+(x)$ against S_{yz}. We are looking for nodes w, which are the smallest in the clique. Note that the intersection is automatically limited to the range [1, x] within S_{yz}. The overhead can then be expressed using

$$c_n(TC, \theta_n) = c_n(E_1, \theta_n) + \sum_{i=1}^{n} \left(X_i |\triangle_{i**}| + \sum_{j=1}^{Y_i} \frac{s_{ij}(s_{ij} - 1)}{2} \right), \tag{33}$$

where $\triangle_{i**}$ is the set of all triangles with i being the smallest node and $s_{ij} = |S_{ij}|$. The rationale for this formula comes from Trigon there is local overhead related to x and remote related to S_{yz}. In the former case, we scan all out- neighbors of x exactly the number of times it appears in some S_{yz}, which is the number of triangles it participates in as the smallest node. The latter case arises when we scan S_{yz} in the

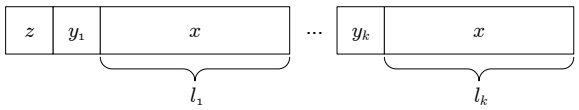


Fig. 9. List of triangles with $z > y > x$.

range [1, x], for all $x \in S_{yz}$. Note that

$$\sum_{i=1}^{n} X_i |\triangle_{i**}| = \sum_{i=1}^{n} \sum_{j=1}^{Y_i} \sum_{k \in S_{ij}} X_k, \tag{34}$$

which might be simpler to compute in practice. Another way to represent overhead is to iterate over all triangles

$$c_n(TC, \theta_n) = c_n(E_1, \theta_n) + \sum_{\triangle_{ijk} \in G^*_\theta} (|S_{jk}(i)| + X_i), \tag{35}$$

where $S_{yz}(x)$ is the set S_{yz} restricted to nodes smaller than x. The total cost on Twitter is 43T and the extra RAM usage is minimal since $|S_{xz}| <= \max_i X_i$.

Theorem 7. *This intersection overhead in Algorithm 7 can be present by (33) or (35).*

2.3.3 Triangle Director (TD)

This is a slight improvement over TC. Observe that a clique can be uniquely identified using three directed triangles $\triangle_{xyz}$, $\triangle_{wxz}$, $\triangle_{wyz}$, all ending with z.We first use E_3, pivoting on the largest node z, to discover all triangles $T_z = \triangle_{xyz_x} y$, where $x < y < z$. The only caveat is that we enumerate the ys backwards, starting at the end of $N^+(z)$. We dump the result into a set $S_z = (y, S_{yz})y$. Note that the ys are in descending order and $x \in S_{yz}$ are in ascending. Fix pairs (y, S_{yz}) and (x, S_{xz}) such that $y > x$. Then, suppose we scan S_{yz} from the start position backwards to discover x. This indicates the presence of triangle $\triangle_{xyz}$. Now, taking an intersection of S_{yz}

	RAGE	TC	TD
PLD	8 Q	106 T	76 T
Twitter	4.8 Q	43 T	17 T

Table XIX. Operation numbers of M_6 listing algorithms.

Algorithm	Operation Type
RAGE	Intersection & Random Access
TC	Intersection
TD	Intersection

Table XX. Operation types of M_6 listing algorithms.

in [1, *start*] and S_{xz} detects all w that participate in additional triangles $\triangle_{wyz}$ and $\triangle_{wxz}$. After the pair (y, S_{yz}) and (x, S_{xz}) is processed, we move to the next x. Once all x are processed for a given y, we move to the next y, reset the start pointer to 1, and repeat. The code of this algorithm is show on Algorithm 8.

Theorem 8. *This intersection overhead in Algorithm 8 is given by:*

$$c_n(TD, \theta_n) = c_n(E_1, \theta_n) + \sum_{\triangle_{ijk} \in G^*_\theta} (|S_{jk}(i)| + |S_{ik}|), \tag{36}$$

where we ignore the terms related to scanning from the start position to discover x. This cost is simply $\sum_i |S_i|$. Note that TD is faster (17T operations on Twitter) than TC because it intersects a partial set S_{yz} with S_{xy} rather than $N^+(x)$. However, memory consumption of TD is higher around 6M nodes in the largest set S_z on Twitter.

	RAGE	TC	TD
PLD	1.9 years	16.4 hours	11.7 hours
Twitter	1.1 years	6.6 hours	2.6 hours

Table XXI. Estimated runtime of M_6 listing algorithms.

	8.0 GB	4.0 GB	2.0 GB	1.0 GB
PLD	0	141.7 GB	384.8 GB	1424.3 GB
Twitter	0 GB	69.4 GB	153.5 GB	549.5 GB

Table XXII. IO complexity of TD on real world graph.

2.3.4 Runtime Estimation

The models derived in previous section is able to compute the operation number of each algorithm. Table XIX shown the operation number of TC, TD, and RAGE in PLD, Twitter, and Yahoo graph. Since PGD have the same CPU complexity with RAGE, we only use RAGE as the baseline for compare. Based on operation type in Table XX and operation speed in Table II, we can estimate the runtime of TC, TD, and RAGE (Table XXI). From Table XXI, we can see that our algorithms have significant speedup from RAGE. TC is 80-100x than RAGE. TD achieve 30%-60% speedup compare to TC.

2.3.5 External Memory

The idea is rather simple. We load in RAM all possible pairs of graphs $G_\theta^+(k)$ and $G_\theta^+(j)$ for all $j < k$. Note that x is detected in the k-th partition and w in the j-th. The streaming portion reads companion files G_c operation. We do not need the companion files of $G_\theta^+(j)$ since it contains all the relevant edges already. Note that this methods also works for TC, after an appropriate adjustment to the companion

files.

Suppose $w = m/M$ is the number of partitions, each containing M edges. Then, the I/O involved in the out-graphs is $Mw(w+1)/2m^2/(2M)$; however, a bigger issue is that we have to read $G^c_\theta(k)$ exactly wk times. Assuming all companion files have the same size H_3/w, this I/O becomes $G^c_\theta(1)w^2/2 = H_3w/2 = H_3m/(2M)$. Since H_3 can be substantially larger than m, this may become a potential problem. In the worst case, $H_3 = mw$ and our I/O is $O(m^3/M^2)$. For constant degree, $H_3 = O(n)$ and TD has $O(n^2/M)$ complexity.

Table XXII shows the I/O complexity of TD works on real graphs.

2.3.6 TC and TD Runtime

The benchmark of M_6 counting are show on Table XXIII. We can see that the performance of PGD and RAGE are also very close on every graph. For both M_6 counting, TC achieved 400 hundred times speedup from PGD *des* in densest graph (Wikitalk). Also, TD is 30%-60% faster than TC in all graphs. RAGE can not finish process any real graphs. In Twitter, TD achieve 2000x speedup compare to RAGE. Table XXV presents the runtime of external TD on different RAM limit. We can see

Graph	n	m	M_6	RAGE-O	RAGE-N	PGD *reg*	PGD *des*	TC	TD
Amazon0302	260,000	1,240,000	304 K		0.46	0.96	0.48	0.05	0.04
Amazon0505	410,000	3,360,000	4.3 M		2.7	4.44	2.4	0.31	0.21
Cit-Patents	3,370,000	16,500,000	3.5 M		16.25	30.24	12.78	2.47	2.08
web-Google	880,000	5,100,000	40 M		30.7	50.22	32.4	0.51	0.36
web-Stanford	280,000	2,300,000	79 M		148.49	394.08	365.64	0.22	0.21
web-WikiTalk	2,390,000	5,020,000	65 M		391.02	982.2	959.58	1.84	0.80
web-Youtube	1,140,000	2,990,000	5 M		37.55	60.72	51.96	0.40	0.27

Table XXIII. Benchmark of M_6 for small graphs.

	Motifs	RAGE-O	PGD *des*	Falcon 1.0	Falcon 2.0
PLD	32,646,228,464,854	-	-	16 hours	11.4 hours
Twitter	6,622,234,180,319	-	-		2.9 hours

Table XXIV. Benchmark of M_6 on large graphs.

	8.0 GB	4.0 GB	2.0 GB	1.0 GB
PLD	1.9 hours	2.1 hours	2.4 hours	3.3 hours
Twitter				

Table XXV. Runtime of TD when RAM is limit.

that runtime is increase when available RAM is decrease. However, the performance decrease is on a reasonable range.

Algorithm 6 SIMD intersection with counters

```
1: count = 0, i_a = 0, i_b = 0
2: sta = (sa / 4) * 4, stb = (sb / 4) * 4
3: while i_a < st_a && i_b < st_b do
4:   v_a = _mm_loadu_si128(A[i_a]), v_b = _mm_loadu_si128(B[i_b])
5:   a_max = _mm_extract_epi32(v_a, 3)
6:   b_max = _mm_extract_epi32(v_b, 3)
7:   i_a += (a_max <= b_max) * 4
8:   i_b += (a_max >= b_max) * 4
9:   cmp_mask1 = _mm_cmpeq_epi32(v_a, v_b)
10:   v_b = _mm_shuffle_epi32(v_b, SHIFT)
11:   cmp_mask2 = _mm_cmpeq_epi32(v_a, v_b)
12:   v_b = _mm_shuffle_epi32(v_b, SHIFT)
13:   cmp_mask3 = _mm_cmpeq_epi32(v_a, v_b)
14:   v_b = _mm_shuffle_epi32(v_b, SHIFT)
15:   cmp_mask4 = _mm_cmpeq_epi32(v_a, v_b)
16:   cmp_maski = _mm_or_si128(
17:         _mm_or_si128(cmp_mask1, cmp_mask2),
18:         _mm_or_si128(cmp_mask3, cmp_mask4))
19:   cmp_mask = _mm_castsi128_ps(cmp_maski)
20:   mask = _mm_movemask_ps(cmp_mask)
21:   p = _mm_shuffle_epi8(v_a, sh_32_mask[mask])
22:   _mm_storeu_si128(C[count], p)
23:   count += _mm_popcnt_u32(mask)
```

Algorithm 7 M_6 Counting(V, E^*)

```
C = 0
for y ∈ V do
    for z ∈ N⁻(y) do
        S = N⁺(y) ∩ N⁺(z)
        for x ∈ S do
            C = C + |N⁺(x) ∩ S|
return C
```

Algorithm 8 M_6-Counting2.0$(z, y_1, l_1, \ldots, y_k, l_k)$

```
C = 0
setup a variable StartP_i for each triangle chunk l_i
for i = 1 to k − 1 do
    for j = i + 1 to k do
        pos = BinarySearch(l_{i,StartP_i}, y_j)
        if pos ≥ 0 then
            C = C + |l_{i,pos+1} ∩ l_j|
        StartP_i = pos + 1
return C
```

Algorithm 9 Triangle(V, E^*)

```
for z ∈ V do
    for y_i ∈ N⁺(z) do
        l_i = N⁺(z) ∩ N⁺(y_i)
```

CHAPTER III

CONCLUSION

Our algorithms are the most scalable and efficient solution for 4-vertex subgraph exact listing. Previous work only focus on graphs with millions of edges while our solution can extend to billions of edges. Despite the fact that we achieved the best performance, there still be several open issues. The first one is to find an 4-clique listing algorithm that able to break the lower bound. The second one is to extend our algorithm to 5-vertex subgraph. The third one is to find an external solution that have linear I/O complexity.

REFERENCES

[1] N. Ahmed, J. Neville, R. Rossi, and N.Duffield, "Efficient Graphlet Counting for Large Networks," in *Proc. IEEE ICDM*, Nov. 2015.

[2] 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.

[3] 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.

[4] M. Bhuiyan, M. Rahman, and M. A. Hasan, "Guise: Uniform sampling of graphlets for large graph analysis," in *Proc. IEEE ICDM*, Dec. 2012, pp. 91–100.

[5] J. Chen, W. Hsu, M. L. Lee, and S.-K. Ng, "NeMoFinder: Dissecting genome-wide protein-protein interactions with meso-scale network motifs," in *Proc. ACM SIGKDD*, Aug. 2006, pp. 106–115.

[6] G. Ciriello and C. Guerra, "A review on models and algorithms for motif discovery in protein–protein interaction networks," *Briefings in functional genomics & proteomics*, vol. 7, no. 2, pp. 147–156, Apr. 2008.

[7] E. R. Elenberg, K. Shanmugam, M. Borokhovich, and A. Dimakis, "Beyond Triangles: A Distributed Framework for Estimating 3-profiles of Large Graphs," in *Proc. ACM SIGKDD*, Aug. 2015, pp. 229–238.

[8] D. Feldman and Y. Shavitt, "Automatic large scale generation of internet pop level maps," in *Proc. IEEE GLOBECOM*, Dec. 2008, pp. 1–6.

[9] I. Finocchi, M. Finocchi, and E. G. Fusco, "Clique counting in MapReduce: algorithms and experiments," *Journal of Experimental Algorithmics (JEA)*, vol. 20, no. 1, pp. 1–7, Oct. 2015.

[10] O. Frank, "Triad count statistics," *Annals of Discrete Mathematics*, vol. 38, pp. 141–149, Dec. 1988.

[11] L. Getoor, *Introduction to statistical relational learning*, 1st ed. MIT press, 2007.

[12] M. Gonen and Y. Shavitt, "Approximating the number of network motifs," *Internet Mathematics*, vol. 6, no. 3, pp. 349–372, Feb. 2011.

[13] M. Granovetter, "The strength of weak ties: A network theory revisited," *Sociological theory*, vol. 1, no. 1, pp. 201–233, Jan. 1983.

[14] J. A. Grochow and M. Kellis, "Network motif discovery using subgraph enumeration and symmetry-breaking," in *Proc. Research in Computational Molecular Biology*, April 2007, pp. 92–106.

[15] 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.

[16] D. Hales and S. Arteconi, "Motifs in evolving cooperative networks look like protein structure networks," *NHM*, vol. 3, no. 2, pp. 239–249, Jun. 2008.

[17] T. Hočevar and J. Demšar, "A combinatorial approach to graphlet counting," *Bioinformatics*, vol. 30, no. 4, pp. 559–565, Feb. 2014.

[18] P. W. Holland and S. Leinhardt, "Local structure in social networks," *Sociological methodology*, vol. 7, pp. 1–45, Jan. 1976.

[19] H. Inoue and K. Taura, "SIMD-and cache-friendly algorithm for sorting an array of structures," *PVLDB*, vol. 8, no. 11, pp. 1274–1285, Jul. 2015.

[20] M. Jha, C. Seshadhri, and A. Pinar, "A Space Efficient Streaming Algorithm for Triangle Counting Using the Birthday Paradox," in *Proc. ACM SIGKDD*, Aug. 2013, pp. 589–597.

[21] M. Jha, C. Seshadhri, and A. Pinar, "Path Sampling: A Fast and Provable Method for Estimating 4-Vertex Subgraph Counts," in *Proc. WWW*, May 2015, pp. 495–505.

[22] M. Jha, C. Seshadhri, and A. Pinar, "A Space-Efficient Streaming Algorithm for Estimating Transitivity and Triangle Counts Using the Birthday Paradox," *ACM TKDD*, vol. 9, no. 3, pp. 15:1–15:21, Feb. 2015.

[23] 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," *BMC Bioinformatics*, vol. 10, no. 1, p. 318, Oct. 2009.

[24] H. Kashima, H. Saigo, M. Hattori, and K. Tsuda, "Graph kernels for chemoinformatics," *Chemoinformatics and Advanced Machine Learning Perspectives: Complex Computational Methods and Collaborative Techniques*, p. 1, Nov. 2010.

[25] N. Kashtan, S. Itzkovitz, R. Milo, and U. Alon, "Efficient sampling algorithm for estimating subgraph concentrations and detecting network motifs," *Bioinformatics*, vol. 20, no. 11, pp. 1746–1758, Mar. 2004.

[26] 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.

[27] O. Kuchaiev, T. Milenković, V. Memišević, W. Hayes, and N. Pržulj, "Topological network alignment uncovers biological function and phylogeny," *Journal of the Royal Society Interface*, vol. 7, no. 50, pp. 1341–1354, Mar. 2010.

[28] M. Kuramochi and G. Karypis, "Finding frequent patterns in a large sparse graph*," *Data mining and knowledge discovery*, vol. 11, no. 3, pp. 243–271, Nov. 2005.

[29] L. Lai, L. Qin, X. Lin, and L. Chang, "Scalable Subgraph Enumeration in MapReduce," *PVLDB*, vol. 8, no. 10, pp. 974–985, Jun. 2015.

[30] D. Marcus and Y. Shavitt, "Efficient counting of network motifs," in *Proc. IEEE ICDCS Workshops*, Jun. 2010, pp. 92–98.

[31] L. A. A. Meira, V. R. Máximo, A. L. Fazenda, and A. F. D. Conceiç, "Acc-Motif: Accelerated network motif detection," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 11, no. 5, pp. 853–862, Apr. 2014.

[32] T. Milenkoviæ and N. Pržulj, "Uncovering biological network function via graphlet degree signatures," *Cancer informatics*, vol. 6, p. 257, Apr. 2008.

[33] T. Milenković, W. L. Ng, W. Hayes, and N. Pržulj, "Optimal network alignment with graphlet degree vectors," *Cancer informatics*, vol. 9, p. 121, Apr. 2010.

[34] 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.

[35] C. C. Noble and D. J. Cook, "Graph-based anomaly detection," in *Proc. ACM SIGKDD*, Aug. 2003, pp. 631–636.

[36] S. Omidi, F. Schreiber, and A. Masoudi-Nejad, "MODA: an efficient algorithm for network motif discovery in biological networks," *Genes & genetic systems*, vol. 84, no. 5, pp. 385–395, Oct. 2009.

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

[38] 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.

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

[40] T. Plantenga, "Inexact subgraph isomorphism in MapReduce," *Journal of Parallel and Distributed Computing*, vol. 73, no. 2, pp. 164–175, Oct. 2013.

[41] N. Pržulj, D. G. Corneil, and I. Jurisica, "Modeling interactome: scale-free or geometric?" *Bioinformatics*, vol. 20, no. 18, pp. 3508–3515, Jul. 2004.

[42] M. Rahman, M. A. Bhuiyan, and M. A. Hasan, "Graft: An efficient graphlet counting method for large graph analysis," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 10, pp. 2466–2478, Jan. 2014.

[43] L. Ralaivola, S. J. Swamidass, H. Saigo, and P. Baldi, "Graph kernels for chemical informatics," *Neural Networks*, vol. 18, no. 8, pp. 1093–1110, Oct. 2005.

[44] P. Ribeiro, F. Silva, and M. Kaiser, "Strategies for network motifs discovery," in *Proc. IEEE Conference on e-Science*, Oct. 2009, pp. 80–87.

[45] R. Rossi, N. K. Ahmed *et al.*, "Role discovery in networks," *IEEE Transaction on Knowledge and Data Engineering*, vol. 27, no. 4, pp. 1112–1131, Jun 2015.

[46] S. E. Schaeffer, "Graph clustering," *Computer Science Review*, vol. 1, no. 1, pp. 27–64, May 2007.

[47] 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.

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

[49] F. Schreiber and H. Schwöbbermeyer, "MAVisto: a tool for the exploration of network motifs," *Bioinformatics*, vol. 21, no. 17, pp. 3572–3574, Jul. 2005.

[50] C. Seshadhri, A. Pinar, and T. G. Kolda, "Triadic measures on graphs: The power of wedge sampling," in *Proc. Proceedings of the SIAM Conference on Data Mining (SDM)*, Jul. 2013.

[51] C. Seshadhri, A. Pinar, and T. G. Kolda, "Wedge sampling for computing clustering coefficients and triangle counts on large graphs," *Statistical Analysis and Data Mining: The ASA Data Science Journal*, vol. 7, no. 4, pp. 294–307, Aug. 2014.

[52] 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, p. 2.

[53] N. Shervashidze, T. Petri, K. Mehlhorn, K. M. Borgwardt, and S. Vishwanathan, "Efficient graphlet kernels for large graph comparison," in *Proc. AISTATS*, April 2009, pp. 488–495.

[54] J. Shun and K. Tangwongsan, "Multicore triangle computations without tuning," in *Proc. IEEE ICDE*, Apr. 2015, pp. 149–160.

[55] G. M. Slota and K. Madduri, "Complex network analysis using parallel approximate motif counting," in *Proc. IEEE IPDPS*, May 2014, pp. 405–414.

[56] G. M. Slota and K. Madduri, "Fast approximate subgraph counting and enumeration," in *Proc. IEEE ICPP*, Oct. 2013, pp. 210–219.

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

[58] N. H. Tran, K. P. Choi, and L. Zhang, "Counting motifs in the human interactome," *Nature Communications*, vol. 4, p. 2241, Aug. 2013.

[59] J. Ugander, L. Backstrom, and J. Kleinberg, "Subgraph frequencies: Mapping the empirical and extremal geography of large graph collections," in *Proc. WWW*, Apr. 2013, pp. 1307–1318.

[60] S. V. N. Vishwanathan, N. N. Schraudolph, R. Kondor, and K. M. Borgwardt, "Graph kernels," *JMLR*, vol. 11, pp. 1201–1242, Apr. 2010.

[61] S. Wernicke, "Efficient Detection of Network Motifs," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 3, no. 4, pp. 347–359, Oct. 2006.

[62] S. Wernicke and F. Rasche, "FANMOD: A tool for fast network motif detection," *Bioinformatics*, vol. 22, no. 9, pp. 1152–1153, Feb. 2006.

[63] E. Wong, B. Baur, S. Quader, and C.-H. Huang, "Biological network motif detection: principles and practice," *Briefings in bioinformatics*, vol. 13, no. 2, pp. 202–215, Mar. 2012.

[64] L. Zhang, Y. Han, Y. Yang, M. Song, S. Yan, and Q. Tian, "Discovering discriminative graphlets for aerial image categories recognition," *IEEE Image Processing*, vol. 22, no. 12, pp. 5071–5084, Dec. 2013.

[65] L. Zhang, M. Song, Z. Liu, X. Liu, J. Bu, and C. Chen, "Probabilistic graphlet cut: Exploiting spatial structure cue for weakly supervised image segmentation," in *Proc. IEEE CVPR*, Jun. 2013, pp. 1908–1915.