NANYANG TECHNOLOGICAL UNIVERSITY

ENUMERATION OF SUBSTRUCTURE PATTERNS FROM LARGE GRAPHS

A thesis submitted
for a degree of Master of Engineering
by

CHU Shumo

School of Computer Engineering
Nanyang Technological University

May 2013

Abstract

Graph is a ubiquitous data model and can be used to represent complex entities and their relationships, such as bioinformatics and biochemistry networks, traffic networks, communication networks, social networks, etc. This thesis study the problem of enumeration of fundamental substructure patterns in graphs. The enumeration of substructure patterns of graphs plays an important role in network analysis, since these problems are crucial to the understanding of complex networks and integral parts of many more advanced techniques of graph analysis.

In recent years, we have seen a dramatic increase in the number, the size, and the variety of networks. For example, there are over 800 million active users in Facebook and there are over 1 billion webpages in WWW according to Google. How to handle big data becomes a very challenging problem for graph analysis. Due to the large volume, large graphs cannot fit into main memory of a single computer any longer. And due to the huge gap between the efficiency of accessing data from external memory and from main memory, (in terms of bandwidth and random access time), existing algorithms on enumerating substructures become inefficient or even infeasible for large graphs.

In this thesis, we study the problem of enumerating substructures of large disk-resident graph. We make the following major contributions.

- . We developed the first I/O-efficient algorithms for listing triangles in massive networks and applied them in the computation of clustering coefficient, transitivity, triangular connectivity, etc. For the same dataset, our algorithm outperform existing algorithm by orders of magnitude.

- We developed a general algorithm framework to guarantee performance in networks with different characteristics and worked out a non-trivial parallel version of this algorithm. This algorithm is up to 1,000 times faster than the state-of-the-art algorithms for clique enumeration in disk-resident networks with a single machine and shows near linear scale up given more machines.

Acknowledgments

I would like to express my sincere gratitude to my supervisor Dr. James Cheng for his unending patience and guidance. He not only shares his vision and insights with me on research, but also set an example of great personality. I can always be inspired by his unique perspective of challenging research problems and intense passion for pursuing answers.

I also want to thank my co-authors, Yiping Ke and Linhong Zhu. I enjoyed working with them. They are smart and knowledgable. Discussions with them help me a lot in research. I would also like to acknowledge my lab mates for their friendship and support-Zhiqiang Xu, Bin Li, Hao Xia, Tim Wu, Peilin Zhao, Jinfeng Zhuang, Diwen Zhu, Wenqing Lin, Xin Cao, Bosheng Zhang, Zongyang Ma, Min Wu, Miao Lin, Shaohua Li, Yong Liu, Chengliang Li and many great guys in DISCO Lab.

Lastly, I would like to thank my parents. They are the best parents in the world, always open minded and supportive. They are not only my parents but the best friends in my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This introduction gives a brief review of substructure enumeration of large graphs. First we introduce the background and motivations of the problems. Then we formulate the problem and set the research scope of our study. We conclude our contributions at last.

1.1 Background and Motivations

Graphs are used to represent complex entities and their relationships, such as bioinformatics and biochemistry networks, traffic networks, communication networks, social networks, etc. In recent years, we have seen a dramatic increase in the number, the size, and the variety of networks. For example, there are over 800 million active users in Facebook and there are over 1 billion webpages in WWW according to Google.

Substructure enumeration in large graphs is important because it is crucial to the understanding of complex networks and integral parts of many more advanced techniques of graph analysis. For example, an elementary substructure of graphs is the triangles, also known as the building blocks of a network. Triangles are closely related to a number of social theories such as social status, social balance, and structural hole. And triangle enumeration can be applied as to compute clustering coefficient, transitivity, triangular connectivity, etc.

However, the enumeration of substructure patterns in large graphs is very challenging. The volume of real world graphs becomes large and keeps increasing. This would make it is impossible to keep the whole graph in the main memory during computation. And in current computer architecture, the exchange of data between main memory (RAM) and external memory (usually hard disk) is very expensive in two ways.

- *Bandwidth*. If the data exchange (read or write) is sequential, the bandwidth of DDR3 SDRAM is from 6500 MB/s to 17000 MB/s [1], however the bandwidth of a modern 7200 rpm hard disk is about 80 MB/s [2], which is about 100 times slower comparing with the main memory.

- *Random Access Time*. If an algorithm need to randomly access data, due to data blocking in the computer systems, the computer need to find the right block to read or write. For main memory, the time overhead for each random data accessing is 10-15 ns. However, due to the mechanical limit of hard disk, the latency is about 4-5 ms for a typical 7200 rpm hard disk, this is more than 100,000 times slower.

This huge performance gap between main memory and hard disk, especially the difference in random accessing time, make almost all existing algorithms on substructure enumeration impractical if the graph need to be stored in hard disk. Our aim is to design I/O efficient algorithms for substructure pattern enumeration so that we can handle large, disk-resident graphs.

1.2 Problem and Research Scope

We study the enumeration of substructure patterns in real world large graphs, such as social networks, world wide web graphs and semantic graphs. The real world graphs we study are mostly sparse graphs. Thus, we can usually assume that the average degree of a graph is much less than the number of vertices of a graph.

There are many types of substructure patterns, such as cycles, paths, small subgraphs and k-core. This thesis focuses on the study of two fundamental substructure patterns, *triangles* and *maximal cliques*, since they are most widely recognized as the elementary substructures of a graph. We define the problems of triangle enumeration and maximal clique enumeration as follows.

- *Triangle Enumeration.* Given three distinct vertices of a graph, we say these three vertices form a *triangle* if there is an edge between each distinct pair of vertices among them. *Triangle Enumeration* is to list all triangles in a given graph without duplication.

- *Maximal Clique Enumeration.* A *clique* in a graph is a subset of vertices, such that the induced subgraph of this set of vertices is a complete subgraph of the original graph. A clique is called a *maximal clique* if there exists no clique in the original graph that is a proper superset of this clique. The problem of *Maximal Clique Enumeration* is that, given a graph, find the set of all maximal cliques in it.

1.3 Approach and Methodology

To solve this challenging problem, we investigate two scenarios. The first scenario is when the computing resource is limited. For example, the enumeration of substructures must be handled in a single computer or server. The second scenario is when the computing resource is sufficiently available. For example, we could use a cluster of servers which contains many computing node to do the enumeration.

Thus, there are two kinds of approaches and methods which we rely on to solve the problems.

- *External Memory Data Structure and I/O Efficient Algorithm design.* External memory data structures are the data structures that are designed for the case that the data is stored in external memory. One important objective for the external memory data structure is to reduce the number of I/Os per operation. Examples of external memory data structures include external linked lists, external stacks and B-trees, etc. These external data structures are usually used as building blocks for I/O efficient algorithms, which try to minimize the number of I/O over the input size. These approaches are extremely useful whenever the computing resource is limited or not, since I/O is easily to be a bottleneck if the data is stored in disk.

- *Parallelization* . When there is sufficient computing resource available, we consider parallelization as an important approach to speed up the computation. Ideally, the *scale up* of a parallel algorithm in terms of number of computing nodes given should be near linear. In the practice, due to the hardness of many graph problems, linear scale up cannot be always achieved.

1.4 Summary of Contributions

In this thesis, we make the following contributions.

(i) We propose the important problem of enumerating substructure patterns in large graphs. This problem is crucial for graph analysis however has not been well solved by existing work.

(ii) We developed the first I/O-efficient algorithms for listing triangles in massive networks and applied them in the computation of clustering coefficient, transitivity, triangular connectivity, etc. Our algorithm has remarkable performance: for the same dataset, a parallel algorithm developed by Yahoo! Research around the same period uses 5.33 minutes running on 1,636 machines, while our algorithm uses only half a minute on a single machine.

(iii) We developed a general algorithm framework for maximal clique enumeration to guarantee the performance in networks with different characteristics and devised a non-trivial parallel version of this algorithm. This algorithm is up to 1,000 times faster than the state-of-the-art algorithms for maximal clique enumeration in disk-resident graphs with a single machine and shows near linear scale up given more machines.

1.5 Thesis Organization

In this chapter, we introduce the problem we study and motivate it. In Chapter 2, we report our I/O efficient algorithm for triangle listing and its applications. In Chapter 3, we present the algorithm framework we designed for clique enumeration and its parallelization. In Chapter 4, we conclude the thesis.

Chapter 2

Triangle Enumeration in Massive Graphs

Triangle Enumeration is one of the fundamental algorithmic problems whose solution has numerous applications especially in the analysis of complex networks, such as the computation of clustering coefficients, transitivity, triangular connectivity, trusses, etc. Existing algorithms for triangle listing are mainly in-memory algorithms, whose performance cannot scale with the massive volume of today's fast growing networks. When the input graph cannot fit in main memory, triangle listing requires random disk accesses that can incur prohibitively huge I/O cost. Some streaming, semi-streaming, and sampling algorithms have been proposed but these are approximation algorithms. We propose an I/O-efficient algorithm for triangle listing. Our algorithm is exact and avoids random disk access. Our results show that our algorithm is scalable and outperforms the state-of-the-art in-memory and local triangle estimation algorithms.

2.1 Introduction

We study the problem of **triangle listing** in a simple undirected graph G, that is, *listing all triangles in* G, where a triangle is a complete subgraph of G that consists of three vertices. Our focus is to design an efficient algorithm for triangle listing when the input graph G is too large to fit in main memory and is disk resident.

Triangles are one of the fundamental types of small subgraphs most commonly used in the analysis of complex graphs/networks. In particular, a triangle is also the shortest non-trivial cycle (i.e., a cycle of length 3) and the smallest non-trivial clique (i.e., a clique of size 3). The concept of triangle is at the heart of the definitions of many important measures for network analysis, such as the clustering coefficients (of a single vertex and of the entire network) [69], transitivity [55, 68], triangular connectivity [12], etc. All these measures can be directly computed from the result of triangle listing.

The aforementioned triangle-centered measures have a large number of important applications. In addition, triangle listing also has a broad range of applications in other areas, such as the discovery of dense subgraphs [67], the computation of trusses (i.e., subgraphs of high connectivity) [27], the detection of spamming activities [13], the study of motif occurrences [52], the uncovering of hidden thematic relationships in the Web [31], etc. In all these applications, triangle listing plays a vital role in their computation.

Although many algorithms have been proposed for triangle listing, these existing algorithms [7, 11, 32, 43, 44, 50, 56, 57] all fall into the category of *in-memory algorithms*. The best existing in-memory algorithms require space that is asymptotically linear in the size of the input graph. However, many real-world networks have grown exceedingly large in recent years and are continuing to grow at a steady rate. For example, the Web graph has over 1 trillion webpages (by Google in 2008), most social networks (e.g., Facebook, MSN) have millions to billions of users, many citation networks (e.g., DBLP, Citeseer) have millions of publications, other networks such as phone-call networks, email networks, stock-market networks, etc., are also massively large and still growing fast.

For handling large graphs that cannot fit in main memory, a number of approximation algorithms have been proposed [8, 10, 13, 20, 28, 63]. However, all these algorithms are restricted to the approximation of *triangle counting*, i.e., estimating the number of triangles in a graph or that formed at each vertex. Algorithms for estimating the total number of triangles in a graph only are considerably accurate [8, 10, 20, 28, 63], but the range of their applications is significantly more limited than that of triangle listing. Algorithms for estimating the number of triangles formed at each vertex in a graph, also called *local triangle counting*, have a wider range of applications but the state-of-the-art algorithm for local triangle counting [13] is still not accurate enough. Moreover, the set of applications of triangle counting is only a small subset of that of triangle listing, as the result of triangle counting is directly obtainable from that of triangle listing.

We propose an I/O-efficient algorithm for exact triangle listing. Designing such an algorithm is difficult because triangle listing requires to access the neighbors of the neighbor of a vertex, which may appear arbitrarily in any position in the graph stored on disk. Thus, random access to the disk-resident graph is required, which incurs huge I/O cost.

Our algorithm iteratively partitions the input graph G into a set of subgraphs that can fit in main memory and processes triangle listing in each local subgraph in memory. To ensure the correctness and completeness of the final result computed iteratively from the local subgraphs, we categorize the triangles into three types. We devise an mechanism that lists all Type 1 and Type 2 triangles, and then converts the remaining Type 3 triangles into Type 1 and Type 2 by a new partition of a shrinking graph at the next iteration. To limit the total number of iterations, we show that we can remove all intra-partition edges at the end of each iteration, thus shrinking G until it becomes empty.

We propose three effective algorithms for graph partitioning for the task of triangle listing in our framework. The first algorithm sequentially scans the input graph only once to partition the graph, thus achieving high efficiency in practice. The sequential graph partitioning algorithm, however, does not have any theoretical guarantee on the number iterations required in the overall process of triangle listing. To this end, we propose another graph partitioning algorithm that requires two scans of the input graph and, by grouping neighboring vertices together based on the application of the dominating set, attains a theoretical upper bound on the total number of iterations needed. However, to apply the dominating set the algorithm requires $O(|V_G|)$ memory space, where $|V_G|$ is the number of vertices in the input graph G. To address this problem, we propose a randomized graph partitioning algorithm, which removes the memory space requirement, and with which we establish a bound on the total number of iterations with a high probability.

Many real-world networks undergo frequent updates. We propose a compact disk-based data structure for efficient update of the set of triangles when the input graph is updated. We discuss the operations for updating the data structure as well as for updating the set of triangles. The data structure is also useful in applications where the set of triangles needs to be materialized on disk.

We evaluated our algorithm on large real datasets with up to 106 million vertices and 1,877 million edges, by comparing with the state-of-the-art in-memory algorithm [50] and the approximation algorithm for local triangle counting [13]. Our algorithm achieves comparable performance with the in-memory algorithm when the graph can fit in main memory. For large graphs that cannot fit in main memory, our results show that our algorithm is superior to the approximation algorithm [13]: at comparable running time and memory usage, the approximation algorithm records a high error rate while ours returns the exact result. The results also show that our data structure supports efficient update of the set of triangles in a dynamic network.

Organization. Section 2.2 gives the notations and problem definition. Section 2.3 describes an in-memory algorithm. Section 2.4 discusses the I/O-efficient algorithm for triangle listing and Section 2.5 proposes the three graph partitioning algorithms. Section 2.6 presents an I/O-efficient algorithm for triangle listing in graphs with extremely high degree vertices. Section 2.7 discusses update in dynamic networks. Section 2.8 presents two applications of our algorithm. Section 2.9 reports the experimental results. Section 2.10 gives the related work. Section 2.11 concludes the paper.

2.2 Notations and Problem Definition

Let $G = (V_G, E_G)$ be a simple undirected graph, where V_G is the set of vertices and E_G is the set of edges. We define the set of *adjacent vertices* (or *neighbors*) of a vertex v in G as $adj_G(v) = \{u : (u, v) \in E_G\}$, and the *degree* of v in G as $deg_G(v) = |adj_G(v)|$.

We assume that the graph is stored in its adjacency list representation (whether in memory or on disk), which is a common data format used for graph storage. Each vertex in the graph is assigned a unique vertex ID. Given any two vertices u and v, we use $u < v$ or equivalently $v > u$ to denote that u is ordered before v according to the order of their vertex IDs. In the adjacency list representation, the vertices are ordered in the ascending order of their vertex IDs.

Given three distinct vertices, $u, v, w \in V_G$, we say that u, v and w form a *triangle* in G if $(u, v), (u, w), (v, w) \in E_G$. We use $\triangle_{uvw}$ to denote the triangle formed by the vertices u, v and w.

The set of triangles that consist of a vertex v, denoted by $\triangle(v)$, is defined as

$$\triangle(v) = \{\triangle_{uvw} : u, w \in adj_G(v), (u, w) \in E_G\}. \qquad \text{(Eq. 2.1)}$$

The *triangle number* of v, denoted by $N_\triangle(v)$, is defined as $N_\triangle(v) = |\triangle(v)|$.

Let $\triangle(G)$ be the set of all triangles in G. Then, $\triangle(G)$ is given by

$$\triangle(G) = \bigcup_{v \in V_G} \triangle(v). \qquad \text{(Eq. 2.2)}$$

The number of triangles in G, denoted by $N_\triangle(G)$, is defined as $N_\triangle(G) = |\triangle(G)|$, which is also given as follows

$$N_\triangle(G) = \frac{1}{3} \sum_{v \in V_G} N_\triangle(v). \qquad \text{(Eq. 2.3)}$$

Equation Eq. 2.3 holds because every triangle $\triangle_{uvw}$ is counted three times, once for each of the three vertices u, v and w.

Given a vertex $v \in V_G$, we say that u, v and w form an *open triangle* centered at v if $u, w \in adj_G(v)$. An open triangle is considered as a potential triangle. A triangle $\triangle_{uvw}$ may be regarded as a *closed triangle* and by definition, $\triangle_{uvw}$ contains three open triangles, centered at u, v, and w, respectively.

The number of open triangles centered at v, denoted by $N_\vee(v)$, is defined as

$$N_\vee(v) = \frac{1}{2} deg_G(v)(deg_G(v) - 1). \tag{Eq. 2.4}$$

Intuitively, $N_\vee(v)$ defines the maximum number of triangles that can be potentially formed from v.

The following example illustrates the concepts.

EXAMPLES 1 *Let G be the graph given in Figure 2.1. Consider the vertices b and e, we have $\triangle(b) = \{\triangle_{abc}, \triangle_{bcg}, \triangle_{bgi}\}$ and $\triangle(e) = \{\triangle_{dej}, \triangle_{efh}\}$. Thus, $N_\triangle(b) = 3$ and $N_\triangle(e) = 2$. By Equation* Eq. 2.4*, $N_\vee(b) = 6$ and $N_\vee(e) = 6$ since $deg_G(b) = 4$ and $deg_G(e) = 4$. From Figure 2.1, we can also easily find that $\triangle(G) = \{\triangle_{abc}, \triangle_{bcg}, \triangle_{bgi}, \triangle_{dej}, \triangle_{efh}, \triangle_{jkl}\}$ and $N_\triangle(G) = |\triangle(G)| = 6$.* □

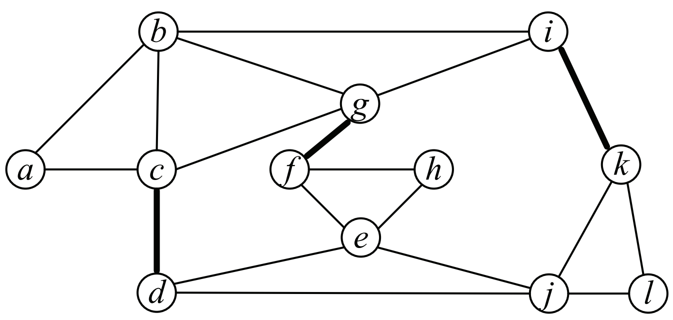

Figure 2.1: A Graph G

Table 2.1: Frequently Used Notations

Notation	**Description**
$G = (V_G, E_G)$	A simple undirected graph
$adj_G(v)$	The set of adjacent vertices of v in G
$deg_G(v)$	The degree of v in G
$\triangle_{uvw}$	A triangle formed by u, v and w
$\triangle(v)$	The set of triangles that contains the vertex v (Eq. Eq. 2.1)
$N_\triangle(v)$	The triangle number of v, $N_\triangle(v) = \|\triangle(v)\|$
$\triangle(e)$	The set of triangles that contains the edge e (Section 2.7.2)
$\triangle(G)$	The set of all triangles in G (Eq. Eq. 2.2)
$N_\triangle(G)$	The number of triangles in G (Eq. Eq. 2.3)
$N_\vee(v)$	The number of open triangles centered at v (Eq. Eq. 2.4)
M	The available main memory size
B	The disk block size
$scan(N)$	$\Theta(N/B)$
$sort(N)$	$\Theta(\frac{N}{B}\log_{M/B}\frac{N}{B})$

Problem Definition. This chapter studies the problem of **triangle listing** defined as follows. Given a graph $G = (V_G, E_G)$, output $\triangle(G)$. In particular, we design I/O-efficient algorithms when G cannot fit in main memory, i.e., $(|V_G| + |E_G|) > M$, where M is the size of the available main memory.

For the complexity analysis of I/O-efficient algorithms, we use the standard I/O model [5] with the following parameters: M is the available main memory size and B is the disk block size, where $1 \ll B \leq M/2$.

We also use the following standard I/O complexity notations: $scan(N)$ I/Os $=\Theta(N/B)$ I/Os and $sort(N)$ I/Os $= \Theta(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os, where N is the amount of data being read or written from/to disk.

Table 2.1 gives the frequently-used notations in the paper.

2.3 In-Memory Triangle Listing

In this section, we first present an in-memory algorithm for triangle listing and use it to explain the difficulties of triangle listing in the case when main memory is insufficient to hold the input graph.

We sketch the algorithm in Algorithm 1. The algorithm intersects the adjacency list of each vertex u with the adjacency list of each neighbor v of u. Clearly, each vertex w as the result of the intersection is a neighbor of both u and v, and as u and v are also neighbors, we obtain a triangle $\triangle_{uvw}$.

A naive algorithm for triangle listing processes every neighbor $v \in adj_G(u)$, and intersects the entire $adj_G(u)$ and $adj_G(v)$. This involves much redundant processing and also outputs duplicate triangles. In Algorithm 1, we only process a neighbor v that is ordered after u (Line 3), because if v is ordered before u, i.e., $v < u$, then v has been processed before u and hence the triangle $\triangle_{vuw}$ must have been already listed (note that $\triangle_{vuw} = \triangle_{uvw}$). For the intersection between $adj_G(u)$ and $adj_G(v)$ (Line 4), we also skip those vertices that are ordered before v (and hence also u) in $adj_G(u)$ and $adj_G(v)$. The above process is similar to the state-of-the-art in-memory algorithm for triangle listing [50], except that their work assumes that the adjacent list of each vertex is sorted in the non-increasing order of the vertex degree, which requires costly pre-processing for the sorting for a large graph. Note that it is not common to store a graph with adjacency lists sorted by the vertex degree, since update to such a storage data format is expensive.

Algorithm 1 is efficient when the input graph can fit in main memory. However, when the input graph G cannot fit in main memory, the algorithm requires huge I/O cost due to random disk access. Most existing in-memory algorithms [7, 11, 32, 43, 44, 50, 56] require random access to each $adj_G(v)$ for each $v \in adj_G(u)$ (as in Line 4 of Algorithm 1 for the intersection). Note that each $adj_G(u)$ in Algorithm 1 is read sequentially as we read G, but $adj_G(v)$ can be in an arbitrary position on disk where G is stored. Other existing in-memory algorithms [57] use an additional array for each vertex in G and the total size of these arrays is in the order of the size of the input graph; thus these arrays need to be stored on disk and random access is again inevitable.

Algorithm 1 *In-Memory Triangle Listing*

Input: A graph $G = (V_G, E_G)$
Output: $\triangle(G)$

```
1.   $\triangle(G) \leftarrow \varnothing$;
2.   for each $u \in V_G$ do
3.       for each $v \in adj_G(u)$, where $v > u$, do
4.           for each $w \in (adj_G(u) \cap adj_G(v))$, where $w > v$, do
5.               $\triangle(G) \leftarrow (\triangle(G) \cup \{\triangle_{uvw}\})$;
             end
         end
     end
6.   return $\triangle(G)$;
```

When G cannot fit in main memory, Algorithm 1 requires $O(|E_G| \cdot scan(d_{max}))$ I/Os in the worst case, where d_{max} is the maximum vertex degree in G, since we need to randomly access $adj_G(v)$ for each edge $(u, v) \in E_G$ and $deg_G(v) = O(d_{max})$. This I/O cost can be prohibitively large especially when G is large.

2.4 I/O-Efficient Triangle Listing

In this section, we present an I/O-efficient algorithm for triangle listing in a large graph when main memory is not sufficient to hold the entire graph. We first sketch the framework of our algorithm and then present the details of the algorithm.

2.4.1 Algorithm Framework

When the input graph G cannot fit in main memory, we can only load a portion (i.e., a subgraph) of G that can fit in main memory each time. Thus, our algorithm iteratively performs triangle listing in a subgraph of G that fits in main memory. We outline the framework of our algorithm as follows.

- Each iteration:

  - Partition G into a set of subgraphs, $\mathcal{P} = \{G_1, \ldots, G_i, \ldots, G_p\}$, such that each G_i can fit in main memory;
  - Load each G_i in main memory and perform triangle listing in G_i;
  - Remove from G those edges of G_i that can no longer contribute to triangle listing.

- Repeat the above iteration until G becomes empty.

The main idea of our algorithm is to iteratively partition the graph and perform triangle listing in each local subgraph G_i separately, as to avoid random access to arbitrary vertices (and their adjacency lists) in the graph.

The concept is simple but there are a number of technical challenges: (1) ensuring the correctness and completeness of the final result obtained from the iterative local computations; (2) an effective and efficient partitioning algorithm for triangle listing; and (3) bounding the overall I/O complexity of the algorithm (i.e., the I/O complexity at each step and the number of iterations). We discuss the above three issues in each of the following subsections.

2.4.2 Correctness and Global-Completeness of Local Triangle Listing

We first propose an algorithm that ensures the correctness of triangle listing in each local subgraph of G as well as the completeness of the global result obtained from all local computations.

The design of our algorithm is based on the following Lemma.

Lemma 2.1 *Let $\mathcal{P} = \{G_1, \ldots, G_i, \ldots, G_p\}$ be a partition of a graph $G = (V_G, E_G)$, where $\cup_{1 \le i \le p} V_{G_i} = V_G$ and $V_{G_i} \cap V_{G_j} = \varnothing$ for $1 \le i < j \le p$. Then, $\triangle(G) = \triangle 1 \cup \triangle 2 \cup \triangle 3$, where $\triangle 1$, $\triangle 2$, and $\triangle 3$ are disjoint sets defined as follows.*

- $\triangle 1 = \cup_{1 \le i \le p} \{\triangle_{uvw} : u, v, w \in V_{G_i}\}$.

- $\triangle 2 = \cup_{1 \leq i,j \leq p \,\wedge\, i \neq j}\{\triangle_{uvw} : u, v \in V_{G_i}, w \in V_{G_j}\}$.

- $\triangle 3 = \cup_{1 \leq i < j < k \leq p}\{\triangle_{uvw}\colon u \in V_{G_i}, v \in V_{G_j}, w \in V_{G_k}\}$.

Proof: First, $(\triangle 1 \cup \triangle 2 \cup \triangle 3) \subseteq \triangle(G)$, since the elements in $\triangle 1$, $\triangle 2$, and $\triangle 3$ are triangles in G.

Next we show $\triangle(G) \subseteq (\triangle 1 \cup \triangle 2 \cup \triangle 3)$. For any triangle $\triangle_{uvw} \in \triangle(G)$, there are only three cases where u, v, and w can be located in the subgraphs in $\mathcal{P}$:

(i) u, v, and w are all in the same subgraph G_i.

(ii) Two of them are in the same subgraph G_i while the other in another different subgraph G_j; that is, without the loss of generality, we may assume that $u, v \in V_{G_i}, w \in V_{G_j}, i \neq j$.

(iii) u, v, and w are in three different subgraphs G_i, G_j, and G_k; that is, without the loss of generality, we may assume that $u \in V_{G_i}, v \in V_{G_j}, w \in V_{G_k}, i < j < k$.

The above three cases correspond to the three types $\triangle 1$, $\triangle 2$, and $\triangle 3$, and thus $\triangle(G) \subseteq (\triangle 1 \cup \triangle 2 \cup \triangle 3)$.

The triangles in $\triangle 1$, $\triangle 2$, and $\triangle 3$ are also called *Type 1*, *Type 2*, and *Type 3* triangles, respectively. The following example illustrates the concept of the three types of triangles.

EXAMPLES 2 *Figure 2.2 shows a partition, $\mathcal{P} = \{G_1, G_2, G_3\}$, of the graph G shown in Figure 2.1. In the figure, $\triangle_{abc}$, $\triangle_{efh}$ and $\triangle_{jkl}$ are Type 1 triangles because all the three vertices in each of these three triangles are in the same subgraph. We only have one Type 2 triangle, $\triangle_{bcg}$, because its vertices are in two subgraphs, G_1 and G_2, in $\mathcal{P}$. We have two Type 3 triangles, $\triangle_{bgi}$ and $\triangle_{dej}$, because all the three vertices of each of the two triangles are in three different subgraphs in $\mathcal{P}$.* $\square$

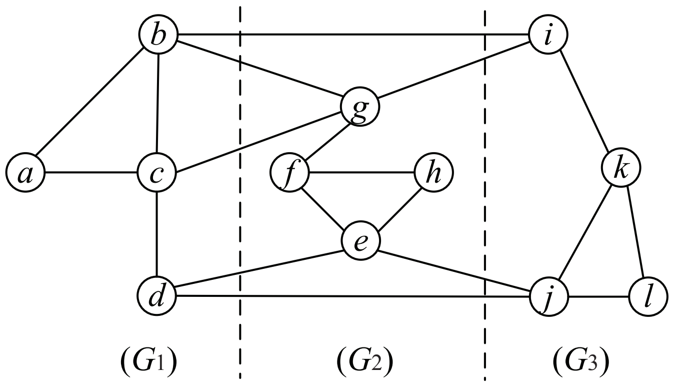

Figure 2.2: A Partition of the Graph G in Figure 2.1: $\mathcal{P} = \{G_1, G_2, G_3\}$

According to Lemma 2.1, a triangle $\triangle_{uvw}$ can be listed by searching a subgraph G_i alone if and only if u, v and w are all in G_i (i.e., Type 1 triangles). However, the number of Type 1 triangles may be quite limited. More critically, we cannot remove any edge (and hence any vertex) of G_i from G even after we list all Type 1 triangles, because an edge (u, v) in $\triangle_{uvw}$ may form another triangle $\triangle_{uvx}$ with a vertex x in another subgraph G_j.

To enable the removal of edges after all triangles containing these edges are listed, and at same time to ensure the completeness of the global result, we introduce the notion of *extended subgraph* as follows.

Definition 2.1 (Extended Subgraph) *Let $H = (V_H, E_H)$ be a subgraph of $G = (V_G, E_G)$. An* extended subgraph *of H in G, denoted by H^+, is a* directed *graph defined as $H^+ = (V_{H^+}, E_{H^+})$, where $V_{H^+} = V_H \cup \{v : u \in V_H, v \in V_G \backslash V_H, (u, v) \in E_G\}$ and $E_{H^+} = \{(u, v) : (u, v) \in E_G, u \in V_H\}$.*

Intuitively, an extended subgraph of H is a subgraph obtained by adding (to H) those directed edges from the vertices in H to those vertices not in H. Note that $\forall v \in V_{H^+} \backslash V_H$, $adj_{H^+}(v) = \varnothing$. For now, we assume that if $V_H = \{u, v\}$, for any $(u, v) \in E$, then the corresponding H^+

fits in main memory, which is a realistic assumption with the available memory size of a commodity PC today. We remove this assumption in Section 2.6.

We give an example of an extended subgraph as follows.

EXAMPLES 3 *Figure 2.3 depicts the extended subgraphs of G_1, G_2, and G_3 in Figure 2.2, i.e., G_1^+, G_2^+, and G_3^+. The shaded vertices are the vertices in each G_i, while the directed edges (i.e., those with an arrow) show the extension to vertices outside each G_i. All the other edges that are without an arrow are considered as the bi-directional edges in G_1^+, G_2^+, and G_3^+.* □

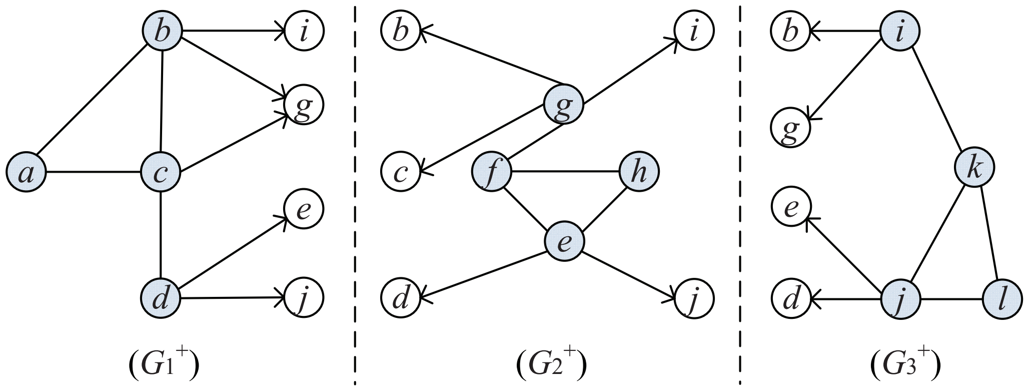

Figure 2.3: The Extended Subgraphs of G_1, G_2, G_3 in Figure 2.2

Based on Definition 2.1, we have the following lemma for triangle listing in an extended subgraph.

Lemma 2.2 *Let H^+ be an extended subgraph of a subgraph H of G. Then:*

- *Let $\triangle 1(H^+) = \{\triangle_{uvw} : \triangle_{uvw} \in \triangle 1,\ \ u, v, w \in V_H\}$. Then, $\forall \triangle_{uvw} \in \triangle 1(H^+)$, $\triangle_{uvw}$ can be listed by searching H^+ alone.*

- *Let $\triangle 2(H^+) = \{\triangle_{uvw} : \triangle_{uvw} \in \triangle 2,\ \ u, v \in V_H\}$. Then, $\forall \triangle_{uvw} \in \triangle 2(H^+)$, $\triangle_{uvw}$ can be listed by searching H^+ alone.*

Algorithm 2 *I/O-Efficient Triangle Listing*

Input: A graph $G = (V_G, E_G)$
Output: $\triangle(G)$

```
1.    while($G$ is not empty)
2.        Partition $G$ into $\mathcal{P} = \{G_1, \ldots, G_i, \ldots, G_p\}$;
3.        for each extended subgraph $G_i^+$ of $G_i \in \mathcal{P}$ do
4.            List all triangles in $\triangle 1(G_i^+)$ and $\triangle 2(G_i^+)$ (by Algorithm 3);
5.            Remove all edges in $G_i$ from $G$;
          end
      end
```

In addition, for any edge $(u, v) \in E_H$*,* (u, v) *does not exist in any triangle in* $\triangle(G) \backslash (\triangle 1(H^+) \cup \triangle 2(H^+))$.

Proof: First, $\forall \triangle_{uvw} \in \triangle 1(H^+)$, $\triangle_{uvw}$ can be listed by searching H^+ alone because all the three edges (u, v) (u, w), and (v, w) of $\triangle_{uvw}$ are in H and hence also in H^+. Second, $\forall \triangle_{uvw} \in \triangle 2(H^+)$, $\triangle_{uvw}$ can be listed by searching H^+ alone because $u, v \in V_H$, $w \in adj_{H^+}(u)$ and $w \in adj_{H^+}(v)$, which means that $\triangle_{uvw}$ can be found by intersecting $adj_{H^+}(u)$ and $adj_{H^+}(v)$.

Finally, for any edge $(u, v) \in E_H$, (u, v) does not exist in any triangle in $\triangle(G) \backslash (\triangle 1(H^+) \cup \triangle 2(H^+))$ because any triangle containing (u, v) must be in either $\triangle 1(H^+)$ or $\triangle 2(H^+)$.

Lemma 2.2 implies that we can list all Type 1 and Type 2 triangles from the extended subgraph G_i^+ of each subgraph G_i in the partition $\mathcal{P}$ of G. More importantly, after listing the two types of triangles in each G_i^+, we can remove all edges in G_i (i.e., those bi-directed edges in G_i^+), since all triangles containing these edges have been already listed.

Listing all Type 1 and Type 2 triangles alone is not enough since we still miss all Type 3 triangles. We devise an efficient algorithm that iteratively converts Type 3 triangles into Type 1 and Type 2 triangles so that all triangles can be listed, while at the same time reducing the size of the graph to reduce the I/O cost as well as the search space for subsequent iterations of triangle listing. We outline our algorithm in Algorithm 2.

Algorithm 3 *Triangle Listing in Extended Subgraph*

Input: An extended subgraph $H^+ = (V_{H^+}, E_{H^+})$
Output: $\triangle1(H^+)$ and $\triangle2(H^+)$

```
1.  for each $u \in V_H$ do
2.      for each $v \in adj_H(u)$, where $v > u$, do
3.          for each $w \in (adj_{H^+}(u) \cap adj_{H^+}(v))$ do
4.              if($w > v$ or $w \notin V_H$)
5.                  List $\triangle_{uvw}$;
                end
            end
        end
    end
```

Algorithm 2 is essentially an iterative computation of $\triangle1(G_i^+)$ and $\triangle2(G_i^+)$ from the extended subgraph G_i^+ of each $G_i \in \mathcal{P}$, as defined in Lemma 2.2, where $\mathcal{P}$ is the new partition of the new graph G at each iteration (note that some edges of G are deleted at the end of each iteration, i.e., Step 5 of Algorithm 2). Note that Algorithm 2 deletes the original graph G, but we can first make a copy of G on disk and disk copy is relatively much cheaper compared with triangle listing. For the removal of edges in Step 5 of Algorithm 2, we simply remove all vertices in V_{G_i} from the adjacency list of each vertex in G_i and write the new adjacency lists back to disk.

At the end of each iteration, we remove all edges in each G_i in the current partition $\mathcal{P}$ and obtain a shrunk new graph. Then, at the beginning of the next iteration, we re-partition the new shrunk graph. The new partition $\mathcal{P}$ defines new sets of $\triangle1(G_i^+)$ and $\triangle2(G_i^+)$ for the extended subgraph G_i^+ of each $G_i \in \mathcal{P}$. Thus, Algorithm 2 iteratively converts the old set of Type 3 triangles at the previous iteration into Type 1 and Type 2 triangles with respect to the new partition at the current iteration. This process continues until all edges in G are removed.

Another purpose of graph partition is to make sure that each subgraph in $\mathcal{P}$ is small enough to fit in main memory, so as to avoid random disk access for triangle listing. Meanwhile, we also want to take full utilization of the available memory and hence each $G_i \in \mathcal{P}$ should be as big as possible under the condition that $(|V_{G_i}| + |E_{G_i}|) \leq M$. Thus, if the shrinking graph G becomes small enough to fit in main memory at any iteration, Step 2 of Algorithm 2 computes a partition consisting of only one subgraph, i.e., $\mathcal{P} = \{G\}$, after which the algorithm terminates since all edges in G will be removed at the end of the iteration.

Step 4 of Algorithm 2 invokes Algorithm 3 to compute $\triangle 1(G_i^+)$ and $\triangle 2(G_i^+)$ from G_i^+, i.e., H^+ in Algorithm 3. The extended subgraph G_i^+ can be easily obtained along with the computation of the partition $\mathcal{P}$, which we discuss in Section 2.5.

Algorithm 3 is an in-memory algorithm similar to Algorithm 1. The only difference is that the extended graph H^+ contains two sets of vertices, V_H and $V_{H^+} \backslash V_H$. Algorithm 3 only intersects the adjacency lists of those vertices in V_H. Let w be a vertex found in $(adj_{H^+}(u) \cap adj_{H^+}(v))$, where $u, v \in V_H$. If $w \in V_H$, we also require $w > v$ (and hence also $w > u$) in order to avoid duplicate listing of the triangle $\triangle_{uvw}$. If $w \notin V_H$, then we simply list $\triangle_{uvw}$ since $\triangle_{uvw}$ cannot be listed elsewhere.

Finally, although in many applications the output of our algorithm is pipelined as the input of another algorithm, there are also applications where the set of triangles needs to be materialized on disk. In the worst case, the size of the output for any general graph is $O(|V_G|^3)$, that is, in the case of a complete graph. More precisely, the worst case output size of a graph is given by $O(\frac{1}{3}\sum_{v \in V_G} N_{\vee}(v))$, which depends on the degree of a vertex (see Equation Eq. 2.4). Although the average case output size may be much smaller, it can still be considerably large and should be stored on disk. We present a compact data structure for storing the set of triangles in Section 2.7.1.

We now prove the correctness and completeness of Algorithm 2.

Theorem 2.1 *Given a graph $G = (V_G, E_G)$, Algorithm 2 lists all triangles in G and no false or duplicate triangle is listed.*

Proof: Lemma 2.2 ensures that (1) all triangles containing a removed edge are listed, (2) all edges of any triangle not yet listed are still in the current graph G, and (3) the already listed triangles will not be listed again at any future iterations because at least one of their edges has been removed from G. By (1) and (2), all triangles in G are listed because G becomes empty when Algorithm 2 terminates. Since Algorithm 3 does not list any duplicate triangle due to the enforced vertex ordering, by (3) Algorithm 2 does not list any duplicate triangle. Finally, since all triangles listed by Algorithm 3 are real triangles in G, Algorithm 2 does not list any false triangle.

The overall complexity of triangle listing by Algorithm 2, however, also depends on the graph partitioning algorithm being used. Therefore, we give the overall complexity analysis of Algorithm 2 in Section 2.5.4 after the discussion of the graph partitioning algorithms.

2.5 Graph Partitioning Algorithms for Triangle Listing

The objectives of graph partitioning for the task of triangle listing are: (1) each subgraph in the partition should fill the available memory as much as possible; and (2) each subgraph should contain as many intra-partition edges (i.e., edges within the same subgraph) as possible. The first objective is to fully utilize memory, while the second objective is to remove as many edges as possible at each iteration of Algorithm 2 in order to reduce the search space at each iteration and the number of total iterations.

Graph partitioning that fulfills the above two objectives, however, is known to be APX-hard [9] when the number of subgraphs in the partition is more than 2. There have been a number of approximation algorithms proposed [3, 37–39, 45, 46], but they are in-memory algorithms that are not suitable for triangle listing in massive networks that cannot fit in memory. Other graph preprocessing techniques may be applied to improve the quality of graph partitioning, and hence the efficiency of triangle listing. For example, by discovering vertices that share many common neighbors through a frequent itemset mining process on the adjacency lists [19], and then grouping them into the same subgraphs in the partition. Though useful for many other graph computations, many of these algorithms are either in-memory algorithms or too costly as a preprocessing step, which can considerably increase the overall cost of triangle listing.

For the task of triangle listing in a large graph that cannot fit in memory, we need an efficient algorithm that partitions the graph with limited memory consumption. We propose three efficient graph partitioning algorithms, two of them are streaming algorithms that require only one scan of the input graph, while the other scans the graph only twice. All three algorithms have linear CPU time complexity. Moreover, all the three partitioning algorithms output the extended subgraphs required in the triangle listing algorithm.

2.5.1 Sequential Graph Partitioning

Sequential graph partitioning is a simple, efficient streaming algorithm, which works as follows: we sequentially read the input graph G from disk, whenever the available memory is filled up, the portion of G being read in memory forms a subgraph in the partition. Note that this subgraph is actually an extended subgraph, because each vertex v being read is associated with its adjacency list $adj_G(v)$.

After we scan G once, we obtain a partition with approximately $(|V_G|+|E_G|)/M$ subgraphs in it. Since the algorithm scans G only once, it requires only $O(scan(|V_G| + |E_G|))$ I/Os and $O(|V_G| + |E_G|)$ CPU time. Furthermore, since the subgraphs in the partition is obtained sequentially one after another as we read G, it allows pipelining such that we can process triangle listing in each subgraph as soon as it is obtained, rather than starting triangle listing until the entire partitioning process finishes.

Sequential graph partitioning is effective when the input graph exhibits high locality, i.e., vertices are naturally clustered according to the sequential order by which the graph is stored. For example, in a road network, proximate vertices are assigned consecutive vertex IDs and they are stored sequentially in nearby positions in the adjacency list graph representation. In social network graphs, local communities may also be stored together.

Algorithm 4 *DominatingSet*

Input: A graph $G = (V_G, E_G)$
Output: A dominating set, D, of G

```
1.    $D \leftarrow \varnothing$;
2.    Create a bit array $A$ of size $|V_G|$ and set each bit in $A$ to $0$;
3.    for each $v \in V_G$, where $A[v] = 0$, do
4.        $D \leftarrow D \cup \{v\}$;
5.        Set $A[v]$ and $A[u]$, for all $u \in adj_G(v)$, to 1;
      end
6.    return $D$;
```

2.5.2 Dominating-Set-based Graph Partitioning

Sequential graph partitioning may be efficient in practice but it gives no theoretical guarantee on the number of iterations Algorithm 2 may take. To this end, we propose another graph partitioning algorithm based on the concept of *dominating set*. The partitioning algorithm takes two scans of the input graph, one for the computation of the dominating set and one for graph partitioning.

A dominating set of a graph G is a subset of vertices $D \subseteq V_G$ such that every vertex in G is either in D or a neighbor of some vertex in D. Computing the minimum dominating set is known to be NP-hard. However, for our purpose of graph partitioning, we do not require a minimum dominating set. Thus, we devise an efficient one-pass algorithm to compute a dominating set for G as shown in Algorithm 4.

In Algorithm 4, we first initialize a bit array A of size $|V_G|$ and set all bits in A to 0. Then, we read G from disk sequentially and for each vertex v (together with $adj_G(v)$) read, we add v to D only if $A[v] = 0$. If v is added to D, then we also set v and all v's neighbors to 1 in A. Thus, all vertices in $V_G \backslash D$ are neighbors of some vertex in D.

We then use D to compute a partition of G, as outlined in Algorithm 5. For triangle listing, we want the vertices in the same subgraph in the partition to be highly connected with each other. We divide D into $p = \frac{|E_G|}{M}$ subsets and create p initial subgraphs in $\mathcal{P}$. Then, we use

Algorithm 5 *Dominating-Set-based Graph Partitioning*

```
Input: A graph $G = (V_G, E_G)$
Output: A partition of $G$, $\mathcal{P} = \{G_1, \ldots, G_i, \ldots, G_p\}$

1.    Compute a dominating set $D$ of $G$ by Algorithm 4;
2.    Divide $D$ into $p$ disjoint subsets of roughly the same size;
3.    Create $p$ subgraphs for $\mathcal{P}$ out of the $p$ subsets of $D$;
4.    for each $v \in V_G$, where $v \notin D$, do
5.        Add $v$ and $adj_G(v)$ to the smallest subgraph in $\mathcal{P}$ that has
              at least $(deg_{\mathcal{P}}(v)/p)$ neighbors of $v$;
      end
6.    return $\mathcal{P}$;
```

the dominating vertices in each subset as seeds to grow each of the p subgraphs by attracting their neighbors. Again, we read G sequentially from disk. For each vertex v (together with $adj_G(v)$) read, let $deg_{\mathcal{P}}(v)$ be the *current* total number of neighbors of v in all the subgraphs in the *current* partition $\mathcal{P}$, which can be easily obtained by scanning $adj_G(v)$ and checking which subgraph in $\mathcal{P}$ each neighbor of v belongs to. We choose the subgraph that has at least $(deg_{\mathcal{P}}(v)/p)$ neighbors of v currently, and add v to that subgraph. If there are more than one such subgraph, we add v to the subgraph with the smallest size so far. Upon adding v, we also add $adj_G(v)$ to the subgraph, so that the resultant subgraph is an extended subgraph ready for triangle listing in Algorithm 2.

Whenever the size of a subgraph becomes greater than B (i.e., the block size), we write a block of the subgraph to disk. Thus, we need extra I/Os to first write all subgraphs in $\mathcal{P}$ to disk and then read each subgraph in $\mathcal{P}$ into memory for triangle listing in Algorithm 2. However, the asymptotic I/O complexity of Algorithm 5 is still $O(scan(|V_G| + |E_G|))$. The CPU time complexity is $O(|V_G| + |E_G|)$ since we only need to scan $adj_G(v)$ for each v. But to compute $deg_{\mathcal{P}}(v)$ efficiently we need a look-up table to keep which subgraph in $\mathcal{P}$ a vertex in V_G belongs to. The look-up table requires $(|V_G| \log_2 p)$ bits, which is a problem if $(|V_G| \log_2 p) > M$.

Dominating-set-based graph partitioning has two advantages for the task of triangle listing. First, the method groups neighborhood vertices together, which leads to a larger number of

Type 1 and Type 2 triangles to be listed in each local subgraph in the partition and hence also a larger number of edges to be deleted at the end of each iteration. Second, the method gives a guaranteed lower bound on the number of intra-partition edges, i.e., edges that can be removed at the end of each iteration of Algorithm 2, as we prove in the following lemma.

Lemma 2.3 *Let $\mathcal{P} = \{G_1, \ldots, G_i, \ldots, G_p\}$ be a partition of G computed by Algorithm 5. Then, the number of intra-partition edges of $\mathcal{P}$ (i.e., edges that are incident on vertices within the same subgraph in $\mathcal{P}$) is at least $\frac{|E_G|}{p}$.*

Proof: Let $\mathcal{P}_{t(v)}$ be the current partition $\mathcal{P}$ at the time when a vertex v is added to $\mathcal{P}$ at Step 5 of Algorithm 5. For each vertex v, v is added to the smallest subgraph in $\mathcal{P}_{t(v)}$ that has at least $((deg_{\mathcal{P}_{t(v)}}(v))/p)$ neighbors of v. First, there must exist such a subgraph in $\mathcal{P}_{t(v)}$ when v is being added, because v is the neighbor of at least one vertex in D. Thus, the total number of intra-partition edges is at least $\sum_{v \in V_G \backslash D}((deg_{\mathcal{P}_{t(v)}}(v))/p)$. We have $\sum_{v \in V_G \backslash D}(deg_{\mathcal{P}_{t(v)}}(v)) = |E_G|$ because each edge is counted once by one of its end vertices and no edge exists between any two vertices in D according to Algorithm 4. The result thus follows.

The significance of Lemma 2.3 is further shown when we apply it to obtain an upper bound on the total number of iterations required in Algorithm 2 in Section 2.5.4.

2.5.3 Randomized Graph Partitioning

Dominating-set-based graph partitioning has a theoretical guarantee on the number of iterations performed in Algorithm 2, but it requires $O(|V_G|)$ memory space. For a very large graph, it is possible that $|V_G| > M$. To address this problem, we devise another graph partitioning algorithm that not only has a theoretical guarantee on the number of iterations required in Algorithm 2, but also does not have the memory space problem.

The algorithm is very simple and efficient. It is a streaming algorithm that scans the input graph only once, as shown in Algorithm 6. The algorithm uniformly at random maps each

Algorithm 6 *Randomized Graph Partitioning*

Input: A graph $G = (V_G, E_G)$
Output: A partition of G, $\mathcal{P} = \{G_1, \ldots, G_i, \ldots, G_p\}$

```
1.    Create $p$ empty subgraphs for the initial partition $\mathcal{P}$;
2.    Let $h(v)$ be a function that maps a vertex $v \in V_G$ into $[1..p]$ uniformly at random;
3.    for each $v \in V_G$, where $v \notin D$, do
4.        Add $v$ and $adj_G(v)$ to $G_{h(v)} \in \mathcal{P}$;
      end
5.    return $\mathcal{P}$;
```

vertex $v \in V_G$ to one of the $p = \frac{|E_G|}{M}$ subgraphs in $\mathcal{P}$. We can create p buffers in memory, one for each subgraph in $\mathcal{P}$. When we map a vertex v to a subgraph $G_i \in \mathcal{P}$, we add v and $adj_G(v)$ to G_i's buffer. Whenever a buffer is filled with a block (of size B) of data, we write the block of data to disk and clear it from the buffer.

The following lemma gives the expected number of intra-partition edges of a partition computed by the randomized graph partitioning algorithm.

Lemma 2.4 *Let $\mathcal{P} = \{G_1, \ldots, G_i, \ldots, G_p\}$ be a partition of G computed by Algorithm 6. Let E_{intra} be the set of intra-partition edges of $\mathcal{P}$, i.e., edges that are incident on vertices within the same subgraph in $\mathcal{P}$. Then, the expectation of $|E_{intra}|$ is given by $\mathbf{E}[|E_{intra}|] = \frac{|E_G|}{p}$.*

Proof: For each vertex $v \in V_G$, the probability that v is mapped to a subgraph $G_i \in \mathcal{P}$ is given by $\mathbf{Pr}[h(v) = i] = \frac{1}{p}$. Given two vertices, u and v, the event that u is mapped to a subgraph in $\mathcal{P}$ and the event that v is mapped to a subgraph in $\mathcal{P}$ are independent. Thus, given an edge (u, v), we have $\mathbf{Pr}[(u, v) \in E_{intra}] = p \times \frac{1}{p} \times \frac{1}{p} = \frac{1}{p}$. As a result, the expectation of $|E_{intra}|$ is given by $\mathbf{E}[|E_{intra}|] = \sum_{(u,v) \in E_G} \mathbf{Pr}[(u, v) \in E_{intra}] = \frac{|E_G|}{p}$.

Lemma 2.4 shows that the expected number of intra-partition edges obtained by the randomized graph partitioning is equal to the lower bound on the number of intra-partition edges obtained by the dominating-set-based graph partitioning, but without requiring $O(|V_G|)$ memory space as does the dominating-set-based graph partitioning algorithm. In Section 2.5.4, we also apply Lemma 2.4 to obtain a bound on the total number of iterations required in Algorithm 2 by applying Algorithm 6 with a high probability.

2.5.4 Bounding I/O Complexity by Graph Partitioning

We now analyze the overall complexity of our algorithm for triangle listing. For all the three graph partitioning algorithms, only $O(scan(|V_G| + |E_G|))$ I/Os are required. More precisely, only one scan of the input graph is required for the sequential and the randomized graph partitioning algorithms, while two scans are required for the dominating-set-based graph partitioning algorithm.

At each iteration of Algorithm 2, we read each extended subgraph in the partition into main memory only once. Thus, the overall I/O complexity for each iteration is $O(scan(|V_G|+|E_G|))$, but for a shrinking graph $G = (V_G, E_G)$.

From the above analysis, the overall I/O complexity of Algorithm 2 depends on the total number of iterations. Thus, we analyze the number of iterations required in Algorithm 2 when each of the three graph partitioning algorithms is applied.

Sequential Graph Partitioning

If we apply sequential graph partitioning in Algorithm 2, the number of iterations depends largely on the locality of the graph data, which varies for different datasets and is difficult to analyze. There is no graph model that characterizes this property for real-world graphs. Thus, we use synthetic graph dataset to investigate the behavior of the partitioning technique, as we show in our experiments.

Dominating-Set-based Graph Partitioning

If we apply dominating-set-based graph partitioning, then we can obtain an upper bound on the total number of iterations required in Algorithm 2, as shown in Lemma 2.5.

Lemma 2.5 *If Algorithm 2 partitions G by Algorithm 5, then the total number of iterations required in Algorithm 2 is* $O(\frac{|E_G|}{M})$.

Proof: According to Lemma 2.3, the number of intra-partition edges of $\mathcal{P}$ is at least $\frac{|E_G|}{p}$, where $p = \frac{|E_G|}{M}$. This means that $\frac{|E_G|}{p} = \frac{|E_G|}{|E_G|/M} = M$ edges are removed at the end of each iteration in Algorithm 2. Thus, the total number of iterations required in Algorithm 2 is $O(\frac{|E_G|}{M})$.

With the result of Lemma 2.5, we give the overall I/O complexity of Algorithm 2, when dominating-set-based graph partitioning is applied, as follows.

Theorem 2.2 *If Algorithm 2 partitions G by Algorithm 5, then Algorithm 2 requires* $O(\frac{|E_G|}{M} scan(|V_G|+|E_G|) - scan(\frac{|E_G|^2}{M}))$ *I/Os, where* $(|V_G| + |E_G|)$ *is the size of the original input graph G.*

Proof: According to Lemma 2.5, at least M edges are removed from G at the end of each iteration of Algorithm 2. Thus, at the start of the i-th iteration, $(i-1)M$ edges have been removed from the original graph G. Summing up, the overall I/O complexity of Algorithm 2 is given by $O(\sum_{i=1}^{|E_G|/M}(scan(|V_G| + |E_G| - (i-1)M))) = O(\frac{|E_G|}{M}\ scan(|V_G| + |E_G|) - scan((\frac{|E_G|}{M})^2 M)) = O(\frac{|E_G|}{M} scan(|V_G| + |E_G|) - scan(\frac{|E_G|^2}{M}))$ I/Os.

Although applying dominating-set-based graph partitioning in Algorithm 2 gives a bounded I/O complexity, the overall process requires at least $(|V_G| \log_2 p)$ bits of memory space. The randomized graph partitioning removes this requirement on memory.

Randomized Graph Partitioning

If randomized graph partitioning is applied, we can establish a bound on the total number of iterations required in Algorithm 2 with a high probability as follows.

Lemma 2.6 *Let* $X = |E_{intra}|$ *be the number of intra-partition edges of a partition computed by Algorithm 6. Then, at each iteration of Algorithm 2, we have* $Pr[X \geq (1-\epsilon)M] \geq 1 - \frac{1}{\epsilon^2 M}$, *where* $0 < \epsilon < 1$.

PROOF. Let $|E_{G^i}|$ be the number of edges in the i-th iteration. Let X_k be the indicator random variable for each edge $e_k \in V_{G^i}$, where $X_k = 1$ if $e_k \in E_{intra}$, and $X_k = 0$ otherwise. Then, we have $X = \sum X_k$.

Recall in Algorithm 6, we uniformly at random pick each vertex to be added to one of the $p_i = \frac{|E_{G^i}|}{M}$ subgraphs. Thus, we obtain the variance of X as follows:

$$\begin{aligned}
Var[X] &= Var[\sum X_k] \\
&= E[(\sum X_k)^2] - (E[\sum X_k])^2 \\
&= E[\sum X_k^2 + \sum_{j \neq k} X_j X_k] - (E[\sum X_k])^2 \\
&= |E_{G^i}|\frac{1}{p_i} + |E_{G^i}|(|E_{G^i}| - 1)\frac{1}{p_i^2} - (\frac{|E_{G^i}|}{p_i})^2 \\
&= \frac{(p_i - 1)|E_{G^i}|}{p_i^2} .
\end{aligned}$$

Let $\mu = E[X] = M$, where $E[X] = M$ is obtained in Lemma 2.4. We first obtain $Pr[X < (1-\epsilon)M]$ as follows:

$$\begin{aligned}
& Pr[X < (1-\epsilon)M] \\
= \; & Pr[\, \mu - X > \mu - (1-\epsilon)M \,] \\
\leq \; & Pr[\, |\mu - X| > \epsilon M \,] \\
= \; & Pr[\, |X - \mu| \geq \epsilon \frac{|E_{G^i}|}{p_i} \,] . \qquad \text{(Eq. 2.5)}
\end{aligned}$$

By applying Chebyshev's inequality, we have:

$$\begin{aligned} Pr[X < (1-\epsilon)M] &\leq Pr[\,|X-\mu| \geq \epsilon\frac{|E_{G^i}|}{p_i}\,] \\ &\leq \frac{Var[X]}{(\frac{\epsilon|E_{G^i}|}{p_i})^2} \\ &= \frac{p_i - 1}{\epsilon^2|E_{G^i}|}\,. \end{aligned} \qquad \text{(Eq. 2.6)}$$

Thus, we obtain:

$$\begin{aligned} Pr[X \geq (1-\epsilon)M] &> 1 - \frac{p_i - 1}{\epsilon^2|E_{G^i}|} \\ &> 1 - \frac{p_i}{\epsilon^2|E_{G^i}|}\,. \end{aligned} \qquad \text{(Eq. 2.7)}$$

Since $p_i = \frac{|E_{G^i}|}{M}$, from Equation Eq. 2.7 we obtain the desired result:

$$Pr[X \geq (1-\epsilon)M] \;>\; 1 - \frac{1}{\epsilon^2 M}\,.$$

□

With the result of Lemma 2.6, the following theorem establishes a bound on the total number of iterations required in Algorithm 2 with a high probability.

Theorem 2.3 *Let n be the total number of iterations required in Algorithm 2 by applying Algorithm 6. Then, $n \leq \frac{|E_G|}{(1-\epsilon)M}$ with a probability of at least $(1-\frac{1}{\epsilon^2 M})^{\frac{|E_G|}{(1-\epsilon)M}}$, where $0 < \epsilon < 1$.*

PROOF. Let A_i be the event that at i-th iteration of Algorithm 2, $|E_{intra}| \geq (1-\epsilon)M$. Then, we have:

$$\begin{aligned} Pr[\cap_{i=1}^{k} A_i] &= Pr[A_1] \times Pr[A_2|A_1] \ldots Pr[A_k| \cap_{i=1}^{k-1} A_i] \\ &\geq (1 - \frac{1}{\epsilon^2 M})^k . \end{aligned} \qquad \text{(Eq. 2.8)}$$

Let $k = \frac{|E_G|}{(1-\epsilon)M}$, then $\cap_{i=1}^{k} A_i$ implies that Algorithm 2 terminates in at most k iterations, since at each iteration, the number of intra-partition edges is at least $(1-\epsilon)M$ according to Lemma 2.6. By substituting k with $\frac{|E_G|}{(1-\epsilon)M}$ in Equation Eq. 2.8, the proof follows.

□

The following example gives an idea how tight the bound obtained in Theorem 2.3 may be.

EXAMPLES 4 *Let $\epsilon = 0.1$ and $M = 10^9$. If $|E_G| = 10^{10}$, then by Theorem 2.3, Algorithm 2 runs at most 11 iterations with a probability of greater than 0.99999. In fact, with this reasonable setting of $\epsilon = 0.1$ and $M = 10^9$, for any graph with $|E_G| \leq (9 \times 10^{13})$ edges, we obtain a probability greater than 0.99 that Algorithm 2 terminates within the bound specified in Theorem 2.3, i.e., $n \leq \frac{|E_G|}{(1-\epsilon)M}$. For a graph of a size with more than (9×10^{13}) edges, it is certainly reasonable to assume a larger available memory size M, with which we can establish the bound again with a high probability.* □

With the result of Theorem 2.3, we obtain a bound on the overall I/O complexity of Algorithm 2 with a high probability as follows.

Theorem 2.4 *If Algorithm 2 partitions G by Algorithm 6, then with a probability of at least $(1-\frac{1}{\epsilon^2 M})^{\frac{|E_G|}{(1-\epsilon)M}}$, the I/O complexity of Algorithm 2 is $O(\frac{|E_G|}{(1-\epsilon)M} scan(|V_G|+|E_G|) - scan(\frac{|E_G|^2}{(1-\epsilon)M}))$ I/Os, where $0 < \epsilon < 1$ and $(|V_G| + |E_G|)$ is the size of the original input graph G.*

Proof: The proof is similar to that of Theorem 2.2, by replacing M in the proof of Theorem 2.2 with $(1-\epsilon)M$.

Finally, we remark that the CPU time complexity for triangle listing at each iteration of Algorithm 2 is similar that of the counter-part in-memory algorithm with the same input graph. We thus refer the readers to the related work [50] for details.

2.6 Triangle Listing for High Degree Vertices with Limited Memory

In the previous sections, we assume that if $V_H = \{u, v\}$, for any $(u, v) \in E$, then the corresponding extended subgraph H^+ fits in main memory (the assumption is given after Definition 2.1 in Section 2.4.2). In other words, we assume that $(deg_G(u) + deg_G(v)) < M$. This assumption is required because if for some $u, v \in V_G$, $adj_G(u)$ and $adj_G(v)$ cannot fit in memory but they appear in the same extended subgraph, then it incurs extra I/O cost when we apply the in-memory triangle listing algorithm in this extended subgraph. Note that to list any triangles in a local subgraph in a partition, V_H must consist of at least two vertices in the corresponding extended subgraph $H^+ = (V_{H^+}, E_{H^+})$; otherwise, either we have listed all triangles in the graph or we need to re-partition the graph.

The above assumption is realistic with the available memory size of a commodity PC today because for a machine with 4GB of memory, $adj_G(u)$ and $adj_G(v)$ should consist of 10^9 adjacent vertices in order to use up the memory. Even with very large power-law graphs [35, 54] that can consist of few vertices with large degree, it is still extremely rare that we have such a large real graph with this extreme vertex degree. Note that we are not referring to the number of vertices in a graph but the degree of a vertex in a graph.

The above estimation shows that our I/O-efficient algorithm can handle most of the real-world large graphs. For those extremely large graphs for which the assumption is not valid, we propose an algorithm to handle this extreme case as follows.

Let S be the set of high degree vertices that violate the above-mentioned assumption. We first remove S, together with $adj_G(v)$ for each $v \in S$, from the input graph G, which gives a

Algorithm 7 *I/O-Efficient Triangle Listing for High Degree Vertices*

Input: A set of high degree vertices S, and $adj_G(v)$ for each $v \in S$
Output: The set of triangles with two end vertices in S, i.e., $\triangle(S) = \{\triangle_{uvw} : u, v \in S\}$

```
1.   for each $u \in S$ do
2.       for each block, $B_1$, of $adj_G(u)$ read from disk do
3.           for each $v$ in $B_1$, where $v \in S$ and $v > u$, do
4.               for each block, $B_2$, of $adj_G(v)$ read from disk do
5.                   for each $w \in (B_1 \cap B_2)$ do
6.                       if($w > v$ or $w \notin S$)
7.                           List $\triangle_{uvw}$;
                         end
                     end
                 end
             end
         end
     end
```

new graph G'. Then, we apply Algorithm 2 to list all triangles in G'. Finally, we list the rest of the triangles in G as shown in Algorithm 7.

Algorithm 7 is similar to Algorithm 3, except that now each $adj_G(u)$ and $adj_G(v)$ cannot fit in main memory. Therefore, to avoid random disk access, we use a *block nested-loop join* to allow sequential disk scans described as follows.

In the block nested-loop join, both the outer relation and the inner relation are the adjacency lists $adj_G(u)$ and $adj_G(v)$ for each pair of vertices $u, v \in S$, where $(u, v) \in E_G$ and $u < v$. The outer relation reads $adj_G(u)$ from disk, which is then joined with $adj_G(v)$ in the inner relation to find the common neighbor w of u and v, which form the triangle $\triangle_{uvw}$.

The I/O complexity analysis of Algorithm 7 is similar to that of the standard block nested-loop join. The number of disk blocks occupied by the outer relation is given by $\alpha = \sum_{u \in S} scan(adj_G(u))$, since the outer relation is scanned only once. For each $u \in S$ processed in Algorithm 7, the number of disk blocks occupied by the inner relation is given by $\beta(u) = \sum_{v \in adj_G(u) \wedge v \in S \wedge v > u} scan(adj_G(v))$. If $adj_G(u)$ can fit in memory, then the algorithm requires $O(\alpha + \sum_{u \in S} \beta(u))$ I/Os. Otherwise, the algorithm requires $O(\alpha + (\alpha/(M/B-2)) \cdot (\sum_{u \in S} \beta(u)))$ I/Os.

The I/O cost can be high if S is large, i.e., there are many extremely high degree vertices in the graph. However, according to [35, 54], there are only a very small number of high degree vertices in a large real world graph, which can be estimated as follows.

According to [35], large real-world graphs follow a power law degree distribution given by the following equation:

$$deg_G(v) = \frac{1}{|V_G|^r}(rank_G(v))^r, \qquad \text{(Eq. 2.9)}$$

where $rank_G(v)$ is the *degree rank* of a vertex v in G, i.e., v is the $(rank_G(v))$-th highest degree vertex in G, and r is the *rank exponent*, where $r < 0$ is a *constant* with a typical value between -0.8 and -0.7 for most real-world networks [35].

According to Equation Eq. 2.9, if $r = -0.7$ and $(deg_G(u) + deg_G(v)) \approx 10^9$, where u and v are the two highest degree vertices in G, then $|V_G| \approx (4 \times 10^{12})$. In other words, for a power-law graph G that has 4 trillion vertices, it only has two vertices u and v in G such that $adj_G(u)$ and $adj_G(v)$ would use up 4GB of memory, i.e., $|S| = 2$ in Algorithm 7.

2.7 Update in Dynamic Networks

Many real-world networks undergo frequent updates. When the network graph is updated, the set of triangles of the graph also needs to be updated. The challenge, however, is that a single edge insertion or deletion can trigger a series of updates to the set of triangles. This can be a costly operation because the set of triangles may be large and stored on disk, and thus the updates require random disk accesses to read and write the triangles.

In this section, we present a compact data structure for efficient update of the set of triangles when the input graph undergoes frequent updates. We consider two types of updates: edge insertion and edge deletion. Vertex insertion/deletion can be considered as a sequence of edge insertions/deletions preceded/followed by the insertion/deletion of an isolated vertex. Note that it is trivial to handle the insertion/deletion of an isolated vertex, since it does not trigger an update to the set of triangles.

2.7.1 A Compact Data Structure for Storage of Triangles

We first present the data structure that supports efficient update of the set of triangles. Apart from supporting efficient update, the data structure is also useful in applications where the set of triangles needs to be materialized on disk.

The set of triangles can be naturally represented as a prefix tree structure. We define the data structure, called *triangle-tree*, as follows.

Definition 2.2 (Triangle-Tree) *Let $\triangle(G)$ be the set of all triangles in a graph G. The* triangle-tree *of G is a prefix tree defined as follows.*

- *The root of the tree represents an empty set.*
- *Each $\triangle_{uvw} \in \triangle(G)$, where $u < v < w$, is represented by a path in the tree as follows: u is a child of the root, v is a child of u, and w is a child of v.*
- *The prefix tree order follows the order of the IDs of the vertices in the triangles.*

Excluding the root, the triangle-tree has three levels, corresponding to the three vertices in a triangle. The compactness comes from the sharing of prefixing vertices among the triangles. Further compression is possible to reduce the space for storage purpose; however, our focus here is to support efficient update operations on disk.

To support the update of the set of triangles in a dynamic graph, we need to support efficient insertion and deletion of nodes in the triangle-tree. Both node insertion and deletion can be done in logarithmic time if the operations are performed in main memory. However, the triangle-tree data structure is stored on disk and the update operations need to be reflected on disk. If the tree is represented as pointers, then random disk accesses to the tree nodes are obviously too expensive. If the nodes in the triangle-tree is stored sequentially on disk, then a single node insertion may cause all the data following the node to be shifted backwards on disk in order to maintain the sequential order, which is prohibitive if the tree is large or the update is frequent.

We propose an effective disk storage scheme for the triangle-tree as follows. The storage scheme consists of three levels, corresponding to the three levels (excluding the root) in the triangle-tree. The nodes at each level of the triangle-tree are stored, according to their level-order traversal sequence, in a linked list of disk blocks (note that only the blocks are stored as a linked list, not the data items inside each block). To enable access to their children, each internal node (i.e., nodes at the first two levels) of the triangle-tree is also associated with a pointer to the block where its children are stored.

To avoid the overflow of a disk block due to node insertions or the under-utilization of disk space due to node deletions, we leave some free space in each of the disk blocks. Then, the following invariant is used to maintain the tree storage for efficient update: *at least $\frac{2}{3}B$ of storage space is utilized for every pair of neighboring disk blocks*. Note that it does not mean that we only use $\frac{2}{3}$B out of every 2B of storage space, the $\frac{2}{3}$B is only an invariant to be maintained during the update process but we can use much more space than that.

With the above invariant, a node insertion requires a single I/O if the insertion does not cause an overflow. Let Y be the disk block where the node is to be inserted, and X and Z be the two disk blocks at either end of Y. Now suppose that we have an overflow, i.e., the block Y is full. Then, if either X or Z still has free space to insert the node, we simply shift the data within the two blocks Y and X (or Z), and then insert the node there. If both X and Z are full, then we split Y into two new blocks, Y_1 and Y_2, each taking approximately $\frac{1}{2}$B of the data from Y. Then, we insert the node into either Y_1 or Y_2 according to its order. It is easy to see that now the three pairs of neighboring blocks, i.e., (X, Y_1), (Y_1, Y_2), and (Y_2, Z), utilize $\frac{3}{2}$B, 1B, and $\frac{3}{2}$B of storage space, respectively. Therefore, the invariant is always maintained for an insertion operation, which takes $O(1)$ I/Os in the worst case.

Now we discuss a node deletion. Let Y be the disk block where the node is deleted, and X and Z be the two disk blocks at either end of Y. To maintain the invariant, we need to check if the pair of blocks (X, Y) or (Y, Z) now use less than $\frac{2}{3}$B of storage space. If both pairs still utilize at least $\frac{2}{3}$B of storage space, then the variant is still maintained. But assume that, without the loss of generality, X and Y now uses less than $\frac{2}{3}$B of storage space, then we simply merge X and Y into one single block. Clearly, the merging re-establishes the invariant. Thus, a node deletion takes $O(1)$ I/Os in the worst case.

There is another type of update that we also need to consider, that is, the update of the pointer of a node to its children's disk block. For an insertion/deletion of a triangle into/from the triangle-tree, we access the tree from parent to child by loading their respective disk blocks into memory; thus, the pointer information to the child's block can be easily updated in memory before it is written back to disk. However, in case of a disk block split/merge due to a node insertion/deletion at the child level, it may trigger an update of the pointer information of some nodes at the parent level. Let X be the block at the parent level currently loaded in memory. Now if the pointer information to be updated at the parent level is within the block X, it can be done directly in X in memory and then write X back to disk. However, if the parent block in which the pointer information needs to be updated is not the block X in memory, then it must be a neighboring block of X on disk, because the nodes in the triangle-tree are stored according to their level-order traversal sequence. Thus, we can read the corresponding neighboring block of X into memory, update the pointer information to the child blocks, and then write the block back to memory. In the worst case, such an update also takes $O(1)$ I/Os only.

2.7.2 Edge Insertion and Deletion

Having discussed the data structure that supports efficient update of triangles, we now discuss how the update to the set of triangles is performed when a new/old edge is inserted/deleted into/from the input graph.

Edge Insertion

We first consider edge insertion. The insertion of an edge, $e = (u, v)$, may create a number of new triangles, which are to be inserted into the triangle-tree. Let $\triangle(e)$ be the set of new triangles created with the insertion of e. Assume that, without the loss of generality, $u < v$. We can divide $\triangle(e)$ into the following three categories:

- $\triangle_1(e) = \{\triangle_{uvw} : \triangle_{uvw} \in \triangle(e), u < v < w\}$,
- $\triangle_2(e) = \{\triangle_{uvw} : \triangle_{uvw} \in \triangle(e), u < w < v\}$,
- $\triangle_3(e) = \{\triangle_{uvw} : \triangle_{uvw} \in \triangle(e), w < u < v\}$.

To insert the triangles in $\triangle_1(e) = \{\triangle_{uvw_1}, \cdots, \triangle_{uvw_k}\}$ into the triangle-tree, we process as follows. We first try to locate u (by binary search) among the children of the root. If u is not a child of the root, then we create a new node u to be inserted as a child of the root. Note that even if u is already a child of the root in the triangle-tree, v cannot be a child of u in the tree because the edge (u, v) is a new edge. Thus, in either case, we create a new node v to be inserted among the children of u or as the only child of u if u is newly created. Then, we create new nodes, w_1, $\cdots$, and w_k, to be inserted as the children of v in the triangle-tree.

The insertion of a triangle $\triangle_{uvw}$ in the second category, $\triangle_2(e)$, into the triangle-tree is processed as follows. We first try to locate u among the children of the root and then w among the children of u. If u and w are already there in the triangle-tree, then we create a new node v to be inserted among the children of w. If u and/or w are not in the triangle-tree, we create u and/or w to be inserted into the tree, followed by the creation of a new node v to be inserted as a child of w.

The insertion of a triangle $\triangle_{uvw}$ in the third category, $\triangle_3(e)$, into the triangle-tree is processed in a similar way as the insertion of a triangle in $\triangle_2(e)$, except that the orders of u and w are now reversed.

The insertion of a new node into the triangle-tree and its cost/complexity is discussed in Section 2.7.1. The extra cost required in the above process of inserting a triangle is the cost of locating a node among the children or determining the position where a node should be inserted among the children, both of which can be done by a binary search in memory.

Edge Deletion

Next, we consider edge deletion. The deletion of an edge, $e = (u, v)$, may invalidate a number of existing triangles, which thus need to be deleted from the triangle-tree. Let $\triangle(e)$ be the set of triangles that are originally in G but are no longer triangles after the removal of e from G. As in Section 2.7.2, we assume that $u < v$ and divide $\triangle(e)$ into three categories.

To delete the triangles in $\triangle_1(e) = \{\triangle_{uvw_1}, \cdots, \triangle_{uvw_k}\}$ from the triangle-tree, we process as follows. First, we locate u among the children of the root and then v among the children u. We delete all the children of v, which are exactly the set $\{w_1, \cdots, w_k\}$, and then also delete v from among the children of u. If after the deletion of v, u has no other child, then we also delete u from the triangle-tree.

The deletion of a triangle $\triangle_{uvw}$ in the second category, $\triangle_2(e)$, from the triangle-tree is processed as follows. First, we locate u among the children of the root and then w among the children u. Then, we locate v among the children of w and delete v. We also delete w if after the deletion of v, w has no other child. Similarly, we delete u if w is the only child of u.

The deletion of a triangle $\triangle_{uvw}$ in the third category, $\triangle_3(e)$, from the triangle-tree is processed in a similar way as the deletion of a triangle in $\triangle_2(e)$, except that the orders of u and w are now reversed.

The deletion of a node from the triangle-tree and its cost/complexity is discussed in Section 2.7.1. The extra cost required in the above process of deleting a triangle is the cost of locating a node among the children, which can be done by a binary search in memory.

2.8 Applications of Triangle Listing

Triangle listing has many important applications. Here, we focus on a few popular ones and demonstrate how our algorithm can be applied to benefit these applications in massive networks that cannot fit in main memory.

2.8.1 Triangle Counting, Clustering Coefficients, and Transitivity

Our algorithm can be readily applied to compute triangle counting, clustering coefficients, and transitivity. For completeness, we also give the definitions of these concepts here.

We first define *clustering coefficient* [69], which is a popular index for network analysis, as follows.

Definition 2.3 (Clustering Coefficient) *The clustering coefficient of a vertex* $v \in V_G$*, where* $deg_G(v) > 1$*, denoted by* $\mathcal{C}(v)$*, is defined as*

$$\mathcal{C}(v) = \frac{N_{\triangle}(v)}{N_{\vee}(v)}. \qquad \text{(Eq. 2.10)}$$

When $deg_G(v) \leq 1$*, we define* $\mathcal{C}(v) = 0$*.*

The clustering coefficient of G, denoted by $\mathcal{C}(G)$*, is defined as*

$$\mathcal{C}(G) = \frac{1}{|V_G|} \sum_{v \in V_G} \mathcal{C}(v). \qquad \text{(Eq. 2.11)}$$

Intuitively, the clustering coefficient of a vertex v, also called the *local clustering coefficient* or *neighborhood density* of v, defines the probability that two vertices are also neighbors of each other if they are both the neighbors of v. In a social network setting, local clustering coefficient implies how likely the two friends of a person are also themselves friends of each other.

The clustering coefficient of a network/graph G is the average of the clustering coefficient of all the vertices in G. G is a *small-world* network if $\mathcal{C}(G)$ is high (with respect to that of a random graph constructed on the same set of vertices) and G also has short average path length [69].

Next we define *transitivity* [55, 68] of a network as follows.

Definition 2.4 (Transitivity) *The transitivity of a graph G, denoted by $\mathcal{C}(G)$, is defined as*

$$\mathcal{T}(G) = \frac{N_{\triangle}(G)}{\frac{1}{3}\sum_{v \in V_G} N_{\vee}(v)}. \qquad \text{(Eq. 2.12)}$$

The transitivity of a network/graph G is a measure of the degree to which the vertices in G tend to cluster together. It is often known as the *global clustering coefficient*, because it is an indication of clustering in the whole network, i.e., globally as contrast to the local clustering coefficient $\mathcal{C}(v)$. Note that $\mathcal{T}(G) \neq \mathcal{C}(G)$ in general.

When the input graph is too large to fit in main memory, our algorithm is far more efficient than the existing algorithms for computing these network measures. Algorithm 2 can be pipelined to compute all the above measures as shown in Algorithm 8, i.e., we do not need to first compute all triangles and perform a post-processing. The algorithm is self-explanatory.

Steps 5-9 of Algorithm 8 requires another scan of G, but the asymptotic I/O complexity of the algorithm is the same as Algorithm 2. Updating $N_{\triangle}(v)$ for all $v \in V_G$ may require $O(|V_G|)$ space, but this can be avoided by writing $N_{\triangle}(v)$ after processing each extended subgraph and then merging the results of all extended subgraphs. The asymptotic I/O complexity still remains the same since the size of $N_{\triangle}(v)$ for all v in an extended subgraph is bounded by the size of the subgraph.

2.8.2 Triangular Vertex Connectivity

Triangular vertex connectivity (also called *3-gonal connectivity*) defines stronger connectivity in a network than single link connectivity, since edges that are in short cyclic component (e.g.,

Algorithm 8 *Triangle Counting, Clustering Coefficients, and Transitivity*

Input: A graph $G = (V_G, E_G)$
Output: $N_\triangle(v)$ and $\mathcal{C}(v)$ for each $v \in V_G$, $\mathcal{C}(G)$, and $\mathcal{T}(G)$

```
1.   $\forall v \in V_G, N_\triangle(v) \leftarrow 0$;
2.   $\mathcal{C}(G) \leftarrow 0$;  $N_\triangle(G) \leftarrow 0$;  $N_\vee(G) \leftarrow 0$;
3.   for each triangle $\triangle_{uvw}$ listed by Algorithm 2 do
4.       $N_\triangle(x) \leftarrow N_\triangle(x) + 1$, for all $x \in \{u, v, w\}$;
     end
5.   for each $v \in V_G$ do
6.       $\mathcal{C}(v) \leftarrow \frac{2N_\triangle(v)}{deg_G(v)(deg_G(v)-1)}$;
7.       $\mathcal{C}(G) \leftarrow (\mathcal{C}(G) + \mathcal{C}(v))$;
8.       $N_\triangle(G) \leftarrow (N_\triangle(G) + N_\triangle(v))$;
9.       $N_\vee(G) \leftarrow (N_\vee(G) + \frac{1}{2} deg_G(v)(deg_G(v) - 1))$;
     end
10.  $\mathcal{C}(G) \leftarrow \frac{\mathcal{C}(G)}{|V_G|}$;
11.  $\mathcal{T}(G) \leftarrow \frac{3N_\triangle(G)}{N_\vee(G)}$;
```

triangles) are considered as strong ties [12, 42]. Triangular vertex connectivity is formally defined as follows [56].

Definition 2.5 (Triangular Vertex Connectivity) *Two vertices u and v are* triangularly vertex-connected *if there exists a sequence of triangles $\langle \triangle_1, \dots, \triangle_n \rangle$ such that u is in $\triangle_1$, v is in $\triangle_n$, and either (1) $n = 1$, or (2) for $1 \le i < n$, $\triangle_i$ and $\triangle_{i+1}$ share at least one common vertex.*

Intuitively, if u and v are connected by a single path, then they become disconnected if any edge on the path is removed. On the contrary, if u and v are connected by a sequence of triangles, then removing any edge does not disconnect them.

Triangular vertex connectivity is important in many applications [31, 40, 61, 69]. It defines an equivalence relation $\mathcal{V}^\triangle$ on V_G. Two vertices u and v belong to the same equivalence class if they are triangularly vertex-connected. The following example further explains the concept.

EXAMPLES 5 *In the graph G given in Figure 2.1, there are two equivalence classes defined by triangular vertex connectivity, $C_1 = \{a, b, c, g, i\}$ and $C_2 = \{d, e, f, h, j, k, l\}$, which can* *be obtained by removing the three edges (in bold lines) that are not part of any triangle. Let $G[C_1]$ and $G[C_2]$ be induced subgraph of G by C_1 and C_2, respectively. Removing any edge from $G[C_1]$ or $G[C_2]$ does not disconnect the vertices in $G[C_1]$ or $G[C_2]$, which demonstrates that the connectivity between the vertices within the graphs are stronger.* $\square$

The existing algorithm for computing the equivalence classes of $\mathcal{V}^{\triangle}$ is based on triangle listing [12]. However, the algorithm assumes that all computations, both equivalent class computation and triangle list, are processed in main memory, which is thus impractical when the input graph is too large and disk resident.

We show that the result of Algorithm 2 can be easily pipelined to compute the equivalence classes of $\mathcal{V}^{\triangle}$ as follows. Upon processing each extended subgraph, we first mark the directed edges (u, v), (u, w), (v, w), where $u < v < w$, for each triangle $\triangle_{uvw}$ listed. Then, we write these marked edges to disk. When Algorithm 2 terminates, we read the marked edges one by one to find which equivalent class each $v \in V_G$ belongs to. This can be done by using two lookup tables, C and A, where $C[j] = i$ indicates that i is the smallest class ID that Class j is connected to, and $A[v] = i$ indicates v belongs to Class i. Initially, $C[i] = i$ and $A[v] = \infty$. We keep a counter c which is initialized to 0. For each marked edge (u, v) read, we do the following: (1) if $A[u] = A[v] = \infty$, we set $A[u] = A[v] = c$ and increment c; (2) if $A[u] \neq A[v]$, without the loss of generality, assume that $A[u] < A[v]$, we set $C[A[v]] = min(C[A[v]], A[u])$ and $A[v] = A[u]$; (3) otherwise, do nothing. Finally, we scan C once to update each $C[i]$ to the smallest class ID that Class i is connected to, and update $A[v] = C[A[v]]$, which indicates the class v belongs to.

Since the number of marked edges written to disk for each extended subgraph cannot exceed the size of the subgraph, the asymptotic I/O complexity of computing the equivalence classes of $\mathcal{V}^{\triangle}$ by the above algorithm is the same as that of Algorithm 2.

2.9 Experimental Results

We compare our algorithms with the state-of-the-art in-memory triangle listing algorithm (denoted by **In-Mem**) [50] and the semi-streaming local triangle estimation algorithm (denoted by **Semi-Stream**) [13]. We ran all experiments on a machine with an Intel Xeno 2.67GHz CPU and 4GB RAM, running CentOS 5.4.

Dataset. We use four real datasets: *LiveJournal* (**LJ**), *U.S. road network* (**USRD**), *World Wide Web of UK* (**WebUK**), and *Billion Triple Challenge* (**BTC**). LJ is a social network (http://www.live-journal.com, http://snap.stanford.edu), where vertices are members and edges represent friendship between members. USRD is the road network of United States, where vertices represent intersections and endpoints, and edges represent the roads connecting these intersections or road endpoints. WebUK is obtained from the YAHOO webspam dataset (http://barcelona.research.yahoo.net), where vertices are pages and edges are hyperlinks. BTC is a semantic graph converted from the Billion Triple Challenge 2009 RDF dataset (http://vmlion25.deri.ie), where each vertex represents an object such as a person, a document, and an event, and each edge represents the relationship between two vertices such as "has-author", "links-to", and "has-title".

To examine the behavior of the graph partitioning algorithms, we also use a synthetic dataset, which is denoted as **Synthetic** in our discussion. We give the number of vertices and edges, and the storage size on disk, of all the datasets in Table 2.2.

Table 2.2: Datasets (M=10^6, G=10^9)

	LJ	**USRD**	**WebUK**	**BTC**	**Synthetic**
$\lvert V_G \rvert$	4.8M	24M	106M	165M	604M
$\lvert E_G \rvert$	69M	58M	1,877M	773M	1,208M
Disk size	809.1MB	969.6MB	20.3GB	10.0GB	21.4GB

2.9.1 Effectiveness of Graph Partitioning Algorithms

We first show the effectiveness of the three graph partitioning algorithms, *sequential graph partitioning* (**Sequential**), *dominating-set-based graph partitioning* (**Dominating**), and *randomized graph partitioning* (**Randomized**). We set the available memory size M for partitioning to be $256MB$, $512MB$, $1GB$, $2GB$, and $4GB$, respectively.

Tables 2.3 and 2.4 reports the total number of iterations Algorithm 2 takes by applying Sequential, Dominating, or Randomized, on the two large datasets, WebUK and BTC. The theoretical upper bound on the total number of iterations by applying Dominating and that by applying Randomized (with a probability of at least 0.99, where $\epsilon = 0.1$) are also given as reference.

Table 2.3: The Number of Iterations of Algorithm 2 on the WebUK Dataset

	M=256MB	M=512MB	M=1GB	M=2GB	M=4GB
Sequential	3	3	2	2	1
Dominating	–	3	2	2	1
Dominating (Upper Bound)	–	16	8	4	2
Randomized	33	16	8	3	2
Randomized (Upper Bound)	35	18	9	5	3

Table 2.4: The Number of Iterations of Algorithm 2 on the BTC Dataset

	M=256MB	M=512MB	M=1GB	M=2GB	M=4GB
Sequential	$\to \infty$	$\to \infty$	7	2	1
Dominating	–	6	3	2	1
Dominating (Upper Bound)	–	9	5	3	2
Randomized	18	9	4	2	2
Randomized (Upper Bound)	19	10	5	3	2

The results show that for both WebUK and BTC, Dominating is most effective in all cases when the available memory is large enough, which is an expected result since dominating-vertex-based graph partitioning groups neighboring vertices together. These neighboring vertices tend to form Type 1 and Type 2 triangles, resulting in more intra-partition edges being deleted at the end of each iteration and hence less total number of iterations. However, Dominating becomes infeasible when the available memory is smaller than $O(|V_G| \log_2 p)$ bits, which happens to both WebUK and BTC when $M = 256$MB.

The results show that Sequential is effective in practice for both WebUK and BTC. In fact, the number of iterations of Algorithm 2 by applying Sequential is as small as that by applying Dominating in most cases. Thus, as shown in Table 2.6, applying Sequential in Algorithm 2 can be more efficient than applying Dominating since Dominating requires two more scans of the input graph at each iteration of Algorithm 2.

However, when the available memory becomes smaller, adopting Sequential shows no sign of termination since there is a point that all or almost all triangles are Type 3 triangles with respect to the partition. Thus, no or very few edges are removed at the end of an iteration. Such a situation happens to the BTC dataset when $M = 256$MB or $M = 512$MB, since BTC has a lower locality than the WebUK dataset.

In this case, Dominating shows its advantage as it gives a guaranteed upper bound on the number of iterations, while our result shows that it indeed has consistent performance. The result also shows that in all cases, the actual number of iterations needed by applying Dominating is always less than its theoretical upper bound.

The results also show that Randomized is not as effective as Sequential and Dominating in some cases, mainly because the randomized process destroys the locality that exists in the real datasets. However, when both Sequential and Dominating fail, the results show that Randomized is still effective, which demonstrates the advantage of applying Randomized in Algorithm 2. We also show that the total number of iterations of Algorithm 2 by applying Randomized almost matches its corresponding upper bound in all cases, for both WebUK and BTC.

We also remark that when $M = 4$GB, applying either Sequential or Dominating in Algorithm 2 requires only 1 iteration, for both WebUK and BTC. This result may seem to be be impossible since both datasets cannot fit in the 4GB memory and therefore one may expect that at least 2 iterations are needed. We examined the details and found that there are only two subgraphs in the partition, each of them (with the more compact binary format than the ascii format in disk storage) can fit in the 4GB memory (just fit for WebUK). In this case, there is no Type 3 triangles (since there are only two subgraphs in the partition) and therefore all triangles, either Type 1 or Type 2, are listed at the first iteration.

To further analyze the behaviors of the graph partitioning algorithms, we generate the Synthetic dataset as follows. Let G be the Synthetic graph. We divide G sequentially into three parts, G_1, G_2, and G_3. The graph G is generated in such a way that there is no triangle within each G_i ($i \in \{1, 2, 3\}$) and across any G_i and G_j ($i \neq j$); that is, there is no Type 1 and Type 2 triangles with respect to the partition $\{G_1, G_2, G_3\}$ of G. The size of each G_i, when loaded in memory, is larger than 2GB but smaller than 4GB.

Table 2.5: The Number of Iterations of Algorithm 2 on the Synthetic Dataset

	M=256MB	M=512MB	M=1GB	M=2GB	M=4GB
Sequential	∞	∞	∞	∞	3
Dominating	–	–	–	2	2
Dominating (Upper Bound)	–	–	–	5	3
Randomized	45	23	10	5	2
Randomized (Upper Bound)	40	20	10	5	3

We show the results for the three graph partitioning algorithms in Table 2.5. For Sequential, the algorithm obviously enters a dead loop when the available memory is less than 4GB, since there is no Type 1 and Type 2 triangles in these cases. For Dominating, the algorithm fails when the available memory is smaller than $O(|V_G| \log_2 p)$ bits, but is very effective when more memory ($M \geq$ 2GB) is available. For Randomized, it works in all settings of available memory and thus the result again demonstrates its advantage over Sequential and Dominating. However, for the cases when Dominating works, Randomized is not as effective as Dominating, since Dominating groups neighboring vertices together even though the neighboring vertices may be widely scattered over the Synthetic graph. In the case when Sequential also works (M =

4GB), Randomized is more effective than Sequential, mainly because Sequential requires more on data locality which is lacking in the Synthetic dataset.

2.9.2 Performance Comparison with Existing Algorithms

We now report the performance of Algorithm 2 by applying Sequential (denoted by **TL-Sequential**), Dominating (denoted by **TL-Dominating**), and Randomized (denoted by **TL-Randomized**), compared with In-Mem [50] and Semi-Stream [13]. We set the available memory size to be 2GB.

Table 2.6 reports the running time (wall-clock time in seconds) of all the algorithms. We do not report the memory consumption since the smaller datasets LJ and USRD can fit in memory and all algorithms use roughly the same memory (less than 1GB), while our algorithms and Semi-Stream use all available memory for the larger datasets BTC and WebUK. In-Mem is extremely slow on the larger datasets BTC and WebUK due to too many I/O swaps as a result of insufficient memory.

Table 2.6: Running Time of Our Algorithms Compared with the Existing Algorithms

	LJ	**USRD**	**BTC**	**WebUK**
TL-Sequential	29.63	6.24	350	2411
TL-Dominating	29.43	12.46	412	2503
TL-Randomized	20.60	29.03	465	2991
In-Mem	32.98	6.68	–	–
Semi-Stream ($\overline{\sigma}(N_{\triangle}(v)) \approx 0.8$)	306	321	3402	7032
Semi-Stream ($\overline{\sigma}(N_{\triangle}(v)) \approx 0.5$)	1275	1683	13711	34722

The result shows that the performance of our algorithm, whichever of the three graph partitioning algorithms is applied, is very competitive. When the graphs can fit in memory, our algorithm has comparable performance with In-Mem. When the graphs are too large to fit in memory, our algorithm demonstrates significant advantage over Semi-Stream, which makes multiple passes over the original input graph (in contrast to our algorithm that makes multiple iterations over a shrinking graph).

Let $\overline{\sigma}(N_{\triangle}(v))$ be the *average approximation error rate* for $N_{\triangle}(v)$, defined as (|approximate value $-$ exact value|/exact value) averaged over all vertices. We find that Semi-Stream takes significantly longer time in order to obtain a low $\overline{\sigma}(N_{\triangle}(v))$. Table 2.6 reports the running time for Semi-Stream by setting $\overline{\sigma}(N_{\triangle}(v)) \approx 0.8$ and $\overline{\sigma}(N_{\triangle}(v)) \approx 0.5$. The result shows that Semi-Stream is many times slower than our algorithm in the case of $\overline{\sigma}(N_{\triangle}(v)) \approx 0.8$ and up to orders of magnitude slower in the case of $\overline{\sigma}(N_{\triangle}(v)) \approx 0.5$. Note that our algorithm is an exact algorithm.

Table 2.7: Error Rate of Semi-Stream at Comparable Running Time with TL-Sequential

LJ	**USRD**	**BTC**	**WebUK**
97.6%	133.6%	115.4%	95.0%

From another angle of comparison, we choose a setting for Semi-Stream that it takes slightly longer time than our algorithm TL-Sequential, and we report the error rate of Semi-Stream in Table 2.7. The result shows that at comparable running time with TL-Sequential, the error rate $\overline{\sigma}(N_{\triangle}(v))$ of Semi-Stream increases significantly.

2.9.3 Performance on Update in a Dynamic Network

In this subsection, we assess the performance of updating the set of triangles when the input graph is updated. We use a real dynamic network, the *Technorati blog* (**blog**) dataset, whose edges are associated with a time stamp when they were created. The Technorati blog network was collected from the top-15 popular queries published by Technorati (technorati.com) every three hours over a period of over a year. For each query, the top-50 results are retrieved. In the blog network, vertices are blogs and edges indicate that two blogs appear in the search result of the same query.

Update in the blog network is performed as follows. We first choose a starting point, that is the blog network obtained by the end of February 2007. Then, to test the update performance on edge insertion, we insert the edges that were created in the months of March and April

2007, in the order of their time stamp. To test the update performance on edge deletion, we delete the edges that were created in the months of February and January 2007, in the reverse order of their time stamp. For each edge insertion and deletion, we update the set of triangles accordingly.

Table 2.8 reports the size of the blog network at the starting point T, i.e., the end of February 2007, two months before T, denoted by T_-, and two months after T, denoted by T_+. We also report the number of triangles in the blog network at T_-, T, and T_+, respectively. The result shows that with the deletion of 1.63 million edges created in the period from T_- to T, we need to delete 5.41 million triangles. This implies that for each edge deletion, on average 3.32 triangles need to be deleted. On the other hand, the insertion of 1.91 million edges from T to T_+ triggers the insertion of 6.81 million triangles. On average, we have 3.57 triangles that need to be inserted for each edge insertion.

Table 2.8: Triangle Number, Storage Size, Triangle-Tree Sizes, and Disk Utilization (for Update)

	$\lvert V_G\rvert$	$\lvert E_G\rvert$	$N_\triangle(G)$	Storage Size of $\triangle(G)$	Triangle-Tree Size for Storage	Triangle-Tree Size for Update	Disk Utilization (Update Only)
T_-	0.09M	1.47M	5.06M	58MB	25MB	67MB	42.97%
T	0.17M	3.10M	10.47M	120MB	49MB	91MB	65.63%
T_+	0.27M	5.01M	17.28M	198MB	78MB	123MB	79.69%

Table 2.8 also reports the disk storage size of the set of triangles, the size of the triangle-tree (for storage purpose only), and that of the triangle-tree (for update purpose), at T_-, T, and T_+, respectively. The result shows that the triangle-tree saves the storage space considerably, especially when the tree is used for storage purpose only, i.e., there is not pointer information maintained for update purpose and every disk block is fully used. The disk utilization rate is considerably lower at T_- than at T_+. This is mainly because the update is performed as a series of continuous edge deletions or edge insertions, respectively, from the starting point T. As a result, the utilization rate at T_- is lower than at T while that at T_+ is higher than at T, which

results in the greater gap between T_- and T_+. However, a continuous load of edge deletions and that of edge insertions stand for the two severest update conditions. And the result shows that in all cases, the invariant given in Section 2.7 is always maintained.

Table 2.9 reports the update time and the number of I/Os used for the update, averaged over all edges or triangles. The result shows that inserting or deleting triangles is highly efficient, which is only about 0.01 milliseconds per triangle update. The number of I/Os required is only slightly more than 1 I/Os for each of the three vertices in a triangle. For edge insertion and edge deletion, they are more costly because an edge insertion/deletion may trigger the insertion/deletion of a number of triangles.

Table 2.9: Average Update Time and Average Number of I/Os for Update

	Insertion (Per Edge)	Insertion (Per $\triangle$)	Deletion (Per Edge)	Deletion (Per $\triangle$)
Avg. Update Time (msec)	0.11	0.015	0.07	0.011
Avg. Number of I/Os	33.43	4.69	26.45	4.00

In summary, the results show that the triangle-tree is efficient for storage as well as for frequent updates in dynamic networks.

2.9.4 Performance on Applications

The experimental results in Section 2.9.2 have demonstrated that our algorithm has significant advantages over the existing algorithms when the input graphs are too large to fit in main memory.

Table 2.10 reports the error rates of clustering coefficient and transitivity of a network approximated by Semi-Stream, denoted by $\sigma(\mathcal{C}(G))$ and $\sigma(\mathcal{T}(G))$, respectively. The result shows that, although the error rate of Semi-Stream is not too large for those global measures, our algorithm is able to obtain the exact results (the running time is almost the same as that shown in Table 2.6).

Table 2.10: Error Rate of Semi-Stream for $\sigma(\mathcal{C}(G))$ and $\sigma(\mathcal{T}(G))$

BTC ($\sigma(\mathcal{C}(G))$)	**BTC** ($\sigma(\mathcal{T}(G))$)	**WebUK** ($\sigma(\mathcal{C}(G))$)	**WebUK** ($\sigma(\mathcal{T}(G))$)
26.5%	40.2%	12.7%	15.3%

We also assess the performance of using our algorithm for computing triangular-vertex-connectivity-based equivalence classes, compared with the state-of-the-art in-memory algorithm [12] (also denoted by **In-Mem** here for simplicity).

Table 2.11: Triangular Vertex Connectivity

	LJ	**USRD**	**BTC**	**WebUK**
TL-Sequential	173.1	11.5	380.2	3670.3
In-Mem	138.1	9.0	N.A.	N.A.

Table 2.11 shows that the running time of our algorithm is comparable with that of In-Mem on the smaller datasets LJ and USRD. But on the larger datasets, BTC and WebUK, In-Mem becomes infeasible due to insufficient memory while our algorithm still records high efficiency. The result again demonstrates that our algorithm is I/O-efficient for processing large graphs.

2.10 Related Work

The algorithms for triangle listing or counting can be categorized into *exact* algorithms and *approximation* algorithms.

The first non-trivial exact algorithm was a spanning-tree-based algorithm [43, 44], which achieves a running time of $O(|E_G|^{1.5})$. This is the optimal worst-case time complexity for an in-memory algorithm for triangle listing because there are $O(|V_G|^3) = O(|E_G|^{1.5})$ triangles in a graph in the worst case. However, a number of practical fast algorithms have been proposed that use vertex ordering and efficient data structures such as lookup tables to facilitate the intersection of the adjacency lists of the neighboring vertices [50, 56, 57]. For triangle counting, the number of triangles can be counted in $O(|E_G|^{1.41})$ time with a fast matrix multiplication algorithm [7]. The basic triangle listing algorithm was also extended to count triads (directed subgraph with three vertices) in directed graph [11]. Maintaining the number of triangles in a dynamic graph was discussed in [32]. All the aforementioned algorithms are in-memory algorithm and require at least $O(|V_G| + |E_G|)$ memory space.

Approximation algorithms have been proposed for triangle counting in large graphs that cannot fit in main memory. Accurate streaming algorithms [8, 10, 20, 28] and sampling algorithm [63] have been proposed to estimate the total number of triangles in a graph. More closely related to our work is the semi-streaming algorithm that estimates the number of triangles formed locally at each vertex in a graph [13]. All these algorithms, however, cannot handle triangle listing, which has a broader range of applications.

For triangle counting in a large graph that cannot fit in main memory, parallel algorithms that apply the MapReduce framework were proposed recently [60]. Their algorithms are exact and do not require to keep the entire input graph in main memory at each individual machine. Our approach is orthogonal to their approach of parallelization for triangle counting.

2.11 Conclusions

We presented an I/O-efficient algorithm for exact triangle listing. To avoid random disk access, our algorithm partitions the input graph and only process one subgraph in the partition each time. By carefully extracting the subgraphs, we proved that triangle listing in those local subgraphs gives globally correct and complete result. We devised three effective partitioning strategies, one achieving high efficiency in practice while the other two bounding the I/O complexity theoretically.

Our experimental results on large graphs with up to 106 million vertices and 1,877 million edges show that our algorithm is either significantly more efficient or more accurate than the state-of-the-art approximation algorithm for local triangle counting [13]. Our results on a dynamic network show that the triangle-tree is able to support frequent updates efficiently. The results also demonstrate the efficiency of applying our algorithm on computing various network measures such as clustering coefficients and transitivity, as well as equivalence classes of the graph based on triangular vertex connectivity. Thus, we believe that our work can benefit many other applications in processing large graphs.

Chapter 3

Maximal Clique Enumeration with Limited Memory

Maximal clique enumeration (*MCE*) is a long-standing problem in graph theory and has numerous important applications. Though extensively studied, most existing algorithms become impractical when the input graph is too large and is disk-resident. We first propose an efficient partition-based algorithm for MCE that addresses the problem of processing large graphs with limited memory. We then further reduce the high cost of CPU computation of MCE by a careful nested partition based on a cost model. Finally, we parallelize our algorithm to further reduce the overall running time. We verified the efficiency of our algorithms by experiments in large real-world graphs.

3.1 Introduction

Maximal clique enumeration (**MCE**) [6, 18] is one of the fundamental problems in graph theory. It is closely related to many other important graph problems, such as maximal independent sets (or minimal vertex covers), graph coloring, maximal common induced subgraphs, maximal common edge subgraphs, etc. Besides graph theory, MCE has numerous other applications, such as social network analysis [36], hierarchy detection through email networks [29], study of structures in behavioral and cognitive networks [14], statistical analysis of financial networks

[16], clustering in dynamic networks [59], the detection of emergent patterns in terrorist networks [15], etc. Moreover, MCE is also closely tied to various applications in computational biology [4], including the detection of protein-protein interaction complex and clustering protein sequences.

The problem of MCE is NP-hard theoretically but because of its significance in real applications, many practical algorithms [4, 6, 18, 21, 33, 34, 41, 48, 49, 51, 53, 59, 62, 64, 65] have been proposed. However, many of these algorithms have become impractical today due to the fast growing size of graphs in the real world. For example, the Web graph has over 1 trillion webpages (by Google in 2008); most social networks (e.g., Facebook, LinkedIn, Twitter, Google+) and communication networks (e.g., MSN, phone, SMS, and email networks) have up to billions of users; other networks such as transportation networks, citation networks, stock-market networks, etc., are also massively large.

The existing algorithms are *in-memory* algorithms, which require space that is asymptotically linear in the size of the input graph. However, the massive size of many graphs has outpaced the advance in the memory available on commodity hardware. MCE computation accesses vertices in a rather arbitrary manner and this results in random access. When memory is insufficient and the graph has to be resident on disk, random disk access incurs high I/O cost.

To process such large graphs, Cheng et al. proposed efficient algorithms [22, 24] which enumerate maximal cliques in local subgraphs that fit in main memory as to eliminate the high I/O cost due to random access. However, MCE intrinsically has a high CPU time complexity and thus reducing the I/O cost alone does not fully address the problem.

To address the intensity of MCE computation, several parallel algorithms [30, 58] have been proposed. However, they still require a copy of the entire input graph to be resident in memory at each computing node, which ends at the same predicament faced by the existing in-memory sequential MCE algorithms.

MapReduce has been popularly used to handle massive data. A MapReduce algorithm [70] was proposed recently for MCE. The MapReduce model, however, may not be efficient for enumeration tasks such as MCE and triangle listing [25, 26] in large graphs due to the huge amount of intermediate data produced in the shuffling phase between Mappers and Reducers.

In this paper, we propose new algorithms for MCE that achieve the following three objectives.

(i) We reduce the I/O cost by a partition-based strategy, which avoids random access for MCE in large graphs that cannot fit in memory.

(ii) We reduce the cost of CPU computation by investigating a neglected (by existing work) but non-negligible cost in MCE; that is, most of the operations at each step of MCE require only constant time but some set intersections require linear time. We devise a cost model to reduce this cost which significantly speeds up the entire MCE process.

(iii) We further reduce the overall running time by parallelizing MCE based on our partitioning strategy. Instead of requiring to keep the entire input graph at each computing node as in the existing parallel algorithms [30, 58], our algorithm requires only limited memory at any computing node.

Extensive experiments on large real-world datasets verify the effectiveness of our algorithms in reducing both the I/O cost and CPU cost. The results show that our algorithms significantly outperform the state-of-the-art MCE algorithms [22, 24, 33, 34, 62], especially in the case when the input graph is too large to fit in main memory.

Organization. The remaining of the paper is organized as follows. Section 3.2 defines the problem and basic notations. Section 3.3 describes a conventional algorithm for MCE and points out its weaknesses. Section 3.4 discusses the details of our algorithms. Section 3.5 reports the experimental results. Section 3.6 discusses the related work and Section 3.7 concludes the paper.

3.2 Notations and Problem Definition

Let $G = (V, E)$ be a simple undirected graph. We define the *size* of G, denoted by $|G|$, as $|G| = (|V| + |E|)$. Given a subset of vertices $S \subseteq V$, we define the *induced subgraph* of G by S as $G_S = (V_S, E_S)$, where $V_S = S$ and $E_S = \{(u, v) : u, v \in S, (u, v) \in E\}$. We define the set of *adjacent vertices* of a vertex v in G as $adj(v) = \{u : (u, v) \in E\}$, and the *degree* of v in G as $deg(v) = |adj(v)|$. Similarly, we define $adj(v, G_S) = \{u : (u, v) \in E_S\}$ and $deg(v, G_S) = |adj(v, G_S)|$.

We assume that all graphs are stored (whether in memory or on disk) in adjacency list representation, where vertices are assigned unique IDs and ordered according to their IDs.

A *clique* C in a graph G is a subset of vertices, where $C \subseteq V$, such that G_C is a complete subgraph of G. C is called a *maximal clique* in G if there exists no clique C' in G such that $C' \supset C$.

Problem definition. The problem of **MCE** is: *given a graph G, find the set of all maximal cliques in G.*

We focus on *sparse graphs*, for which $|E| \ll B \cdot |V|$, and B is the disk block size. Most large and many fast growing real-world networks are sparse. For processing these large sparse graphs, random disk access in the process of MCE is prohibitively expensive since data transferred from/to disk is in blocks, while the exact amount of data used in each step of MCE is much less than the amount of data transferred for each random access, since $deg(v) \ll B$ for most $v \in V$.

Algorithm 9 *IM-MCE*

Input: a graph $G = (V, E)$
Output: all maximal cliques in G

```
1. IM-MCE-Step($\varnothing, V, \varnothing$);
```

3.3 Conventional MCE Algorithm

We first discuss the conventional algorithms for MCE and their shortcomings. Algorithm 9 and Procedure 10 sketch a general algorithm framework for MCE, which is adopted by most existing MCE algorithms [18, 48, 62]. The state-of-the-art parallel MCE algorithm [58] is also developed based on this framework. This framework is also used in later sections to explain our algorithms.

Algorithm 9 calls Procedure 10. In Procedure 10, we use the notations ***todo***, ***doing***, and ***done*** to represent the set of vertices *to be processed*, *being processed*, and *already processed*, respectively.

The algorithm starts from the set of all vertices in G, i.e., $todo = V$, and $done = \varnothing$, by calling Procedure 10. Procedure 10 recursively calls itself to grow a clique C in a depth-first manner until C becomes maximal; that is, when $todo = \varnothing$, i.e., there is no more vertex for C to grow with, and $done = \varnothing$, i.e., C has not been enumerated before (if $done \neq \varnothing$, then $\exists C' \supset C$ such that C' has been found as a maximal clique).

The depth-first MCE process essentially constructs a search tree, or more precisely a search forest joined by a (virtual) root. At each call of Procedure 10, we first pick a *pivot* vertex v_p, which is used to prune all the neighbors of v_p, because any maximal clique containing a vertex $u \in adj(v_p)$ can be enumerated from either v_p or another vertex $v \in adj(u)$, where $v \notin adj(v_p)$. Therefore, all the search subtrees rooted at each $u \in adj(v_p)$ can be pruned.

We then construct $doing$, which is the set of vertices being processed in the current call of Procedure 10. For each vertex $v \in doing$, we grow the current clique C by adding v to C. To further grow $(C \cup \{v\})$ to a bigger clique in the subsequent recursive call, we need to make

Procedure 10 *IM-MCE-Step*$(C, todo, done)$

```
1. if($todo = \varnothing$ and $done = \varnothing$)
2.     output $C$ as a maximal clique;
3. else
4.     choose a pivot vertex $v_p$ from $(todo \cup done)$;
5.     $doing := todo \backslash adj(v_p)$;
6.     for each $v \in doing$ do
7.         $todo := todo \backslash \{v\}$;
8.         IM-MCE-Step$(C \cup \{v\}, todo \cap adj(v), done \cap adj(v))$;
9.         $done := done \cup \{v\}$;
```

sure that every vertex in $todo$ must be a neighbor of v. After the recursive call returns, we add v to $done$ to indicate that v has been processed in the current search subtree.

Shortcomings. Algorithm 9 requires $O(3^{|V|/3})$ time in the worst case, but has shown to be the optimal worst-case complexity [62]. In practice, the existing algorithms that adopt the framework of Algorithm 9 are reasonably fast for processing relatively small real-world graphs. However, these algorithms are in-memory algorithms and require $\Omega(|V| + |E|)$ memory space. If memory is not sufficient or the input graph is disk-resident, the process immediately becomes impractical. As we observe in Line 8 of Procedure 10, the MCE process requires accesses to the neighbor set of each vertex $v \in doing$, i.e., $adj(v)$, which may scatter in different locations in the graph resident on disk. Thus, random disk access leads to huge I/O cost and severely degrades the performance of the in-memory algorithm. In addition, some branches of the search tree constructed by Algorithm 9 can be potentially parallelized. Intuitively, the search subtrees rooted at each of the siblings of the search tree can be constructed in parallel with careful design. We address these issues in Section 3.4.

3.4 Faster MCE Algorithms

In this section, we address the shortcomings of the conventional algorithm for MCE. In particular, we have the following three main objectives: *(1) reducing the I/O cost, (2) reducing the CPU cost, and (3) further reducing the overall running time by parallelization.*

3.4.1 Reducing the I/O Cost

When the input graph cannot fit in main memory, reducing the I/O cost by avoiding random disk accesses becomes the key to the efficiency of MCE. The main idea of our algorithm is to repeatedly extract a subgraph that fits in memory and compute the maximal cliques locally from the subgraph. However, we should also preserve both the *global* correctness and completeness of the results computed from the *local* subgraphs.

We first define the subgraph to be extracted for MCE as follows.

Definition* 1 (Seed Vertices/Subgraph)** *Given a graph $G = (V, E)$, denote S to be a set of* ***seed vertices *selected from V, where $S \subseteq V$. The* ***seed subgraph*** *of G, denoted by $G_S = (V_S, E_S)$, is defined as the induced subgraph of G by S.*

Using the seed subgraph G_S alone is not sufficient to ensure the completeness of MCE since some seed vertices may form a clique with vertices not in G_S. To address this, we extend G_S as follows.

Definition* 2 (Extended Vertices/Subgraph)** *Given a set of seed vertices S of a graph $G = (V, E)$, the set of* ***extended seed vertices*, or simply* ***extended vertices****, denoted by S^+, is defined as $S^+ = S \cup \{v : v \in adj(u), u \in S\}$. The* ***extended seed subgraph****, or simply* ***extended subgraph****, denoted by G_{S^+}, is defined as the induced subgraph of G by S^+.*

Note that the above definition of subgraph is different from the one used in [22, 24], where they do not include the edges among the vertices in $(S^+ \backslash S)$. This difference renders the design of the algorithm totally different, since the subgraphs in [22, 24] still need to access the input graph for MCE while ours do not. Compared with the algorithms in [22, 24], our algorithm is simpler, more efficient, and natural to be parallelized.

The following example illustrates the concepts.

EXAMPLES 6 *Consider the graph G shown in Figure 3.1. If we choose $S = \{0, 1, 2\}$ as the set of seed vertices, we obtain the seed subgraph G_S as shown in Figure 3.2 (left). It is easy to see that the seed subgraph alone cannot produce any maximal clique of G. So we extend the seed vertices to be $S^+ = \{0, 1, 2, 3, 4, 5, 7\}$ and obtain the extended subgraph G_{S^+}, as given in Figure 3.2 (right). The extended subgraph makes possible the local enumeration of global maximal cliques, i.e., $\{0, 1, 2, 3\}, \{1, 4, 5\}, \{2, 7\}$, since G_{S^+} captures fully the neighborhood information of seed vertices.* □

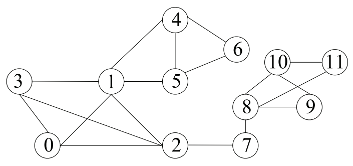

Figure 3.1: An Example of Input Graph, G

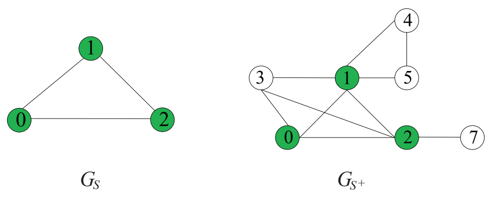

Figure 3.2: Seed Subgraph and Extended Subgraph

With the above formulation of extended subgraph, we first prove that the maximal cliques computed locally in each extended subgraph are also globally maximal.

Let $\mathcal{M}(G)$ and $\mathcal{M}(G_{S^+})$ be the set of maximal cliques in G and G_{S^+}, respectively. Let $\mathcal{M}_S(G) = \{C : C \in \mathcal{M}(G), C \cap S \neq \varnothing\}$ and $\mathcal{M}_S(G_{S^+}) = \{C : C \in \mathcal{M}(G_{S^+}), C \cap S \neq \varnothing\}$ be the set of maximal cliques, respectively in G and G_{S^+}, that contain at least one vertex in S.

Lemma 3.7 $\mathcal{M}_S(G_{S^+}) = \mathcal{M}_S(G)$.

Proof:

We first prove $\mathcal{M}_S(G_{S^+}) \subseteq \mathcal{M}_S(G)$. We begin by proving that $\forall C \in \mathcal{M}_S(G_{S^+})$, $C \in \mathcal{M}(G)$. Suppose $C \notin \mathcal{M}(G)$, then $\exists C' \in \mathcal{M}(G)$ such that $C' \supset C$. Then $\exists v \in C'$ and $v \notin S^+$ (otherwise $C' \in \mathcal{M}_S(G_{S^+})$ and hence $C \notin \mathcal{M}_S(G_{S^+})$). However, $v \notin S^+$ implies that $(v, u) \notin E$ for some vertex $u \in (C \cap S)$, which contradicts that C' is a clique. Thus, $C \in \mathcal{M}(G)$ and it follows that $C \in \mathcal{M}_S(G)$ since $C \cap S \neq \varnothing$.

We then prove $\mathcal{M}_S(G) \subseteq \mathcal{M}_S(G_{S^+})$. $\forall C \in \mathcal{M}_S(G)$, we have $C \in \mathcal{M}(G)$ and $C \cap S \neq \varnothing$. Let $u \in (C \cap S)$. Then $\forall v \in C \backslash \{u\}$, we have $(u, v) \in E$ and thus $v \in S^+$. Therefore, $C \in \mathcal{M}(G_{S^+})$ and it follows that $C \in \mathcal{M}_S(G_{S^+})$.

Lemma 3.7 implies that all maximal cliques in G that contain at least one seed vertex in S can be computed from G_{S^+} alone. This leads to the design of an algorithm that repeatedly extracts a subgraph G_{S^+} that can fit in main memory, computes $\mathcal{M}_S(G_{S^+})$ from G_{S^+} in memory, and then continues to select another set of seed vertices from $(V \backslash S)$, until all vertices in V are processed. The following theorem guarantees the correctness and completeness of the set of maximal cliques so computed.

THEOREM 1 *Let $\mathcal{S} = \{S_1, \ldots, S_k\}$, where $\bigcup_{1 \leq i \leq k} S_i = V$ and $S_i \cap S_j = \varnothing$ for $i \neq j$. Let $\mathcal{M}_{\mathcal{S}} = \bigcup_{1 \leq i \leq k} \mathcal{M}_{S_i}(G_{S_i^+})$. Then, $\mathcal{M}_{\mathcal{S}} = \mathcal{M}(G)$.*

Proof:

First, $\forall S_i$, $\mathcal{M}_{S_i}(G_{S_i^+}) \subseteq \mathcal{M}(G)$ according to Lemma 3.7. Thus, $\mathcal{M}_{\mathcal{S}} \subseteq \mathcal{M}(G)$. Next, $\forall C \in \mathcal{M}(G)$, $\exists S_i$ such that $C \in \mathcal{M}_{S_i}(G_{S_i^+})$, where $C \cap S_i \neq \varnothing$ (note that the existence of S_i is because $\bigcup_{1 \leq i \leq k} S_i = V$). Thus, $\mathcal{M}(G) \subseteq \mathcal{M}_{\mathcal{S}}$.

Theorem 1 immediately leads to the design of an efficient partition-based algorithm, as shown in Algorithm 11.

The algorithm sequentially scans the input graph G to select a set of seed vertices S, extracts G_{S^+} by another scan of G, computes $\mathcal{M}_S(G_{S^+})$ from G_{S^+}, and then continues to select another set of seed vertices until all vertices in V are processed as seed vertices.

The algorithm determines the size of G_{S^+} in Line 4, where a parameter ϕ_{deg} is used to estimate the size of G_{S^+}, and $0 < c < 1$ is to ensure the estimated size is less than M[1], where M is the available memory size. Here, ϕ_{deg} can be the average or maximum vertex degree in G. If ϕ_{deg} is the average vertex degree, it is possible that $|G_{S^+}| > M$; however, this can be easily fixed by further splitting G_{S^+} into smaller extended subgraphs with smaller seed vertex sets. Since only some selection of S can lead to the case "$|G_{S^+}| > M$", in practice the total number of extended subgraphs to be extracted remains relatively stable. If ϕ_{deg} is the maximum vertex degree, we can guarantee that $|G_{S^+}| \leq M$. However, this may lead to under-utilization of the available memory. This is also true for splitting a large G_{S^+} into smaller extended subgraphs in the case of setting ϕ_{deg} as the average vertex degree. However, we shall show in Section 3.4.2 that fully utilizing the available memory may actually lead to an even higher cost in the in-memory computation of MCE.

After computing the set of local maximal cliques in G_{S^+}, i.e., $\mathcal{M}(G_{S^+})$, the algorithm outputs only those cliques containing at least one seed vertex to ensure the correctness according to Lemma 3.7. However, it is possible that a maximal clique C is enumerated in more than one G_{S^+}, if the vertices in C are in different sets of seed vertices, i.e., $\exists u, v \in C$ such that $u \in S_i$ and $v \in S_j$, where $u \neq v$ and $i \neq j$. To avoid duplicate output of a maximal clique, Line 8 sets another condition, ($\nexists u \in C$ s.t. $u \prec S$). Here, $u \prec S$ means that u is selected as a seed vertex (in an earlier iteration of Lines 5-10), before any vertex in S is selected (in the current iteration). Thus, Lines 7-9 do not output a maximal clique C if C contains a vertex that has already been selected as a seed vertex before S is selected as the seed vertex set.

[1]We assume that if $|S| = 1$, then $|G_{S^+}| \leq M$. In the case of a high-degree vertex v, $|G_{S^+}|$ with $S = \{v\}$ can be large. For sparse real-world graphs, we may effectively reduce $|G_{S^+}|$ by sorting the vertex set V in ascending order of the vertex degree, since we actually do not need to include into G_{S^+} any edge (u, w) if u and w are ordered before v, for all $u, w \in S^+$ (as implied by Line 8 of Algorithm 11).

Algorithm 11 *Partition-Based MCE*

Input: a graph $G = (V, E)$

Output: all maximal cliques in G

```
1.  $S := \varnothing$, $S^+ := \varnothing$;
2.  for each $v \in V$ do
3.      $S := (S \cup \{v\})$, $S^+ := (S^+ \cup \{v\} \cup adj(v))$;
4.      if$((\phi_{deg} \cdot |S^+|) \geq cM)$
5.          scan $G$ once to extract $G_{S^+}$;
6.          apply in-memory MCE to $G_{S^+}$;
7.          for each $C \in \mathcal{M}(G_{S^+})$ do
8.              if$((C \cap S) \neq \varnothing$ and $(\nexists u \in C$ s.t. $u \prec S))$
9.                  output $C$ as a maximal clique;
10.         $S := \varnothing$, $S^+ := \varnothing$;
```

Greater Pruning. The condition "$(\nexists u \in C$ s.t. $u \prec S)$" also implies that we do not need to include into G_{S^+} any edge (u, w), $\forall u, w \in S^+$ and $u, w \prec S$. Furthermore, we can push the condition into the in-memory MCE process to achieve greater pruning as follows. Recall that the search tree (implicitly) constructed by Algorithm 9 is a prefix tree. During the MCE process, we can process the seed vertices in S before the vertices in $S^+ \backslash S$. Then we do not grow the current clique C by any vertex $v \prec S$ in Line 6 of Procedure 10. In this way, we avoid duplicate enumeration of maximal cliques as well as achieve greater pruning effect for MCE.

Complexity Analysis. The following theorem gives the complexity of Algorithm 11. We use the following standard I/O complexity notations [5] in the complexity analysis: M is the main memory size, B is the disk block size, $scan(N) = \Theta(N/B)$ I/Os, where $1 \ll B \leq M/2$ and N is the amount of data being read/written from/to disk.

THEOREM 2 *Algorithm 11 requires $O(k \cdot scan(|G|))$ I/Os, $O(kT)$ CPU time, and $O(M)$ memory space, where $k = min\{\frac{|V|(\phi_{deg})^2}{M}, |V|\}$, and T is the CPU time complexity of the in-memory algorithm for computing MCE in an extended subgraph G_{S^+} with $|G_{S^+}| \leq M$.*

Proof:

Algorithm 11 makes $(k+1)$ scans of G, once in Line 1 and k times in Line 5, where k is the total number of extended subgraphs G_{S^+} extracted. From Line 5 we have $(\phi_{deg} \cdot |S^+|) \geq cM$, but we can easily move the last vertex in S to the next round of seed vertex selection to obtain $|G_{S^+}| \leq (\phi_{deg} \cdot |S^+|) = O(M)$. Thus, we have $((\phi_{deg})^2 \cdot |S|) = O(M)$ since $|S^+| \simeq (\phi_{deg} \cdot |S|)$. Since V is divided into k sets of seed vertices, we have $k = O(\frac{|V|}{|S|}) = O(\frac{|V|(\phi_{deg})^2}{M})$. In the worst case, each S contains only one vertex and thus $k \leq |V|$.

The selection of seed vertices and extraction of extended subgraphs require only linear time; thus, the CPU time is bounded by the number of times the in-memory algorithm is invoked to compute MCE on an extended subgraph G_{S^+}, where the maximum size of any G_{S^+} is M (we refer the readers to [62] for the details on the CPU time complexity analysis on the in-memory algorithm). The memory space requirement is $|G_{S^+}| = O(M)$.

3.4.2 Reducing the CPU Cost

In processing large graphs that cannot fit in main memory, the bottleneck of the in-memory algorithms is the huge I/O cost due to random disk access. However, by processing these large graphs using Algorithm 11, the bottleneck may no longer be the I/O cost, but rather the cost of the in-memory computation of MCE in each extracted extended subgraph.

A Neglected but Non-Negligible Cost

The cost of in-memory MCE computation can be broken down into two main types: *the cost of constructing the search tree* and *the cost of set intersection*.

Conventional algorithms focus on reducing the cost of constructing the search tree by employing various pruning techniques to reduce the size of search tree. Although these pruning techniques are vital for efficient MCE, we have observed that we can further reduce the cost of CPU computation as follows.

Among all individual operations in the process of MCE, the most expensive one is the set intersection: two in Line 8 of Procedure 10 (and the set subtraction in Line 5 may also be considered as one). These set intersections can be significantly more costly to process than the other operations that require only constant time.

There are approximation algorithms for set intersection, but for MCE we require an exact answer. To compute $X \cap Y = Z$, if exact set intersection is required, then the cost of the intersection is at least $(|X| + |Y|)$. However, if $|Z|$ is significantly smaller than $|X|$ and $|Y|$, then the majority of the processing is "wasted" on processing *non-contributing* elements in X and Y, i.e., elements in $X \backslash Z$ and $Y \backslash Z$. Thus, the objective here is to reduce the number of non-contributing elements in X and Y.

Reducing the number of non-contributing elements in X and Y in Procedure 10 means removing non-contributing elements in $todo$, $done$, and in particular $adj(v)$ since $adj(v)$ is often significantly larger than $todo$ and $done$ (except in the first call of Procedure 10).

Cost Reduction of Set Intersection

To reduce the portion of the non-contributing elements in $todo$, $done$, and $adj(v)$, we transform the search space from G to G_{S^+}. In this way, the search space is immediately reduced from V to S^+ for $todo$ and $done$, and from $adj(v, G)$ to $adj(v, G_{S^+})$. However, which S^+ and hence G_{S^+} should we select in order to minimize the cost?

To avoid expensive random disk access, G_{S^+} should have size smaller than M. For this purpose, Algorithm 11 naturally fulfills the requirement. However, Algorithm 11 attempts to fully utilize the available memory, i.e., $|G_{S^+}|$ is close to M. But the analysis in Section 3.4.2 suggests that a smaller G_{S^+} gives a smaller cost of set intersection.

Extracting a smaller G_{S^+} in Algorithm 11 can be easily done by setting a smaller c in Line 4 of Algorithm 11. However, doing so proportionally increases the number of scans of G in Line 5 and hence the overall I/O cost. The following lemma enables us to totally avoid any extra I/O in extracting smaller G_{S^+}.

Lemma 3.8 *Given two sets of seed vertices, S_{big} and S_{small}, if $S_{small} \subseteq S_{big}$, then $G_{S^+_{small}}$ is a subgraph of $G_{S^+_{big}}$.*

Proof:

Since $S_{small} \subseteq S_{big}$, by Definition 2, $S^+_{small} \subseteq S^+_{big}$. Then we also have $E_{S^+_{small}} = \{(u, v) : u, v \in S^+_{small}, (u, v) \in E\} \subseteq E_{S^+_{big}} = \{(u, v) : u, v \in S^+_{big}, (u, v) \in E\}$. Thus, $G_{S^+_{small}}$ is a subgraph of $G_{S^+_{big}}$.

Lemma 3.8 shows that it is possible to extract a smaller extended subgraph $G_{S^+_{small}}$ from another bigger extended subgraph $G_{S^+_{big}}$ (which is resident in memory) instead of from the original input graph G (which is resident on disk). However, we still need to determine how small this $G_{S^+_{small}}$ should be.

A Cost Model

We propose a cost model for choosing the right size for the extended subgraph. Suppose that we now have a set of extended subgraphs $\mathcal{G}_\beta$ resident in main memory for MCE computation. Each subgraph $G_{S^+_\beta}$ in $\mathcal{G}_\beta$ is roughly of the same size. Initially, $\mathcal{G}_\beta$ contains only one extended subgraph G_{S^+} extracted in Lines 4-5 of Algorithm 11. We determine (iteratively) whether extracting smaller extended subgraphs from $\mathcal{G}_\beta$ can achieve better efficiency for in-memory MCE computation.

For each $G_{S^+_\beta} \in \mathcal{G}_\beta$, we consider to split $G_{S^+_\beta}$ into a set of smaller extended subgraphs $\{G_{S^+_\alpha}\}$, each of them is roughly of the same size. Let $\mathcal{G}_\alpha$ be the set of all such smaller extended subgraphs extracted from $\mathcal{G}_\beta$.

First, having a smaller $adj(v, G_{S^+_\alpha})$, instead of $adj(v, G_{S^+_\beta})$, in the process of MCE leads to a lower cost of set intersection. The *gain* obtained by preferring $\mathcal{G}_\alpha$ over $\mathcal{G}_\beta$ can be estimated as follows.

$$\begin{aligned} Gain &= \sum_{G_{S_\beta^+} \in \mathcal{G}_\beta} |T_{S_\beta^+}| \cdot \overline{deg}(v, G_{S_\beta^+}) \\ &\quad - \sum_{G_{S_\alpha^+} \in \mathcal{G}_\alpha} |T_{S_\alpha^+}| \cdot \overline{deg}(v, G_{S_\alpha^+}) \\ &= \sum_{G_{S_\beta^+} \in \mathcal{G}_\beta} |T_{S_\beta^+}| \cdot (|E_{S_\beta^+}|/|S_\beta^+|) \\ &\quad - \sum_{G_{S_\alpha^+} \in \mathcal{G}_\alpha} |T_{S_\alpha^+}| \cdot (|E_{S_\alpha^+}|/|S_\alpha^+|), \qquad \text{(Eq. 3.1)} \end{aligned}$$

where $|T_{S_\alpha^+}|$ is the number of nodes in the search tree $T_{S_\alpha^+}$ constructed by Algorithm 9 with $G_{S_\alpha^+}$ as the input, and $\overline{deg}(v, G_{S_\alpha^+})$ is the average degree of a vertex v in $G_{S_\alpha^+}$; and similarly for $|T_{S_\beta^+}|$ and $\overline{deg}(v, G_{S_\beta^+})$.

In Equation (Eq. 3.1), the gain obtained from the reduction in the cost of set intersection with a smaller $adj(v, G_{S_\alpha^+})$ instead of $adj(v, G_{S_\beta^+})$ is reflected by the average degree $\overline{deg}(v, G_{S_\alpha^+})$ and $\overline{deg}(v, G_{S_\beta^+})$. And we multiply the average degree by the number of nodes in the search tree because at each node in the search tree, we need to perform the set intersection as shown in Line 8 of Procedure 10.

If choosing $\mathcal{G}_\alpha$ instead of $\mathcal{G}_\beta$ only results in a gain for MCE, then the optimal choice is by setting $S_\alpha = \{v\}$ for each single vertex $v \in V$. However, preferring $\mathcal{G}_\alpha$ over $\mathcal{G}_\beta$ may also result in some loss due to the extraction of a large number of smaller extended subgraphs and the construction of their search trees during the process of MCE.

First, if we further split $G_{S_\beta^+}$ into a set of smaller $G_{S_\alpha^+}$, we have extra extraction costs. Second, the size of the search tree constructed from $G_{S_\beta^+}$ is smaller than the total size of the search trees constructed from the set $\{G_{S_\alpha^+}\}$ extracted from $G_{S_\beta^+}$. The *loss* of preferring $\mathcal{G}_\alpha$ over $\mathcal{G}_\beta$ is given as follows.

$$\begin{aligned}
Loss &= \Big(\sum_{G_{S_\beta^+} \in \mathcal{G}_\beta} \sum_{G_{S_\alpha^+} \subset G_{S_\beta^+}} (|G_{S_\alpha^+}| + |G_{S_\beta^+}|) \Big) \\
&\quad + \Big(\sum_{G_{S_\alpha^+} \in \mathcal{G}_\alpha} |T_{S_\alpha^+}| - \sum_{G_{S_\beta^+} \in \mathcal{G}_\beta} |T_{S_\beta^+}| \Big). \qquad \text{(Eq. 3.2)}
\end{aligned}$$

In Equation (Eq. 3.2), the first half gives the cost of extracting all $G_{S_\alpha^+}$ from each $G_{S_\beta^+} \in \mathcal{G}_\beta$. Since we scan each $G_{S_\beta^+}$ at most once to extract each $G_{S_\alpha^+}$, we add $|G_{S_\beta^+}|$ to the total cost. The second half is the difference in the total size of the search trees constructed from all $G_{S_\alpha^+}$ and from all $G_{S_\beta^+}$.

Equation (Eq. 3.1) and Equation (Eq. 3.2) represent a tradeoff: we trade off *(1) the cost of set intersection* with *(2) the cost of subgraph extraction and that of search tree construction*; that is, using smaller extended subgraphs reduces (1) but increases (2), while using larger extended subgraphs results in the opposite. We quantify this tradeoff by the following cost model.

$$TotalGain = Gain - Loss. \qquad \text{(Eq. 3.3)}$$

Finally, in Equation (Eq. 3.1) and Equation (Eq. 3.2), $|G_{S_\alpha^+}|$ and $|G_{S_\beta^+}|$ are known after we extract the subgraphs, while $|T_{S_\alpha^+}|$ and $|T_{S_\beta^+}|$ can be estimated by a variation of Knuth's method [47] for estimating the size of a backtracking tree of MCE proposed in [22].

An Improved Algorithm for MCE

With the results of the previous subsections, we propose an improved partition-based algorithm for MCE, as shown in Algorithm 12. We name this algorithm as **SeqMCE** (for **Seq**uential **MCE**), to distinguish from the parallel algorithm proposed in Section 3.4.3.

```
Algorithm 12 SeqMCE
Input: a graph $G = (V, E)$
Output: all maximal cliques in $G$

Replace Lines 6-9 of Algorithm 11 by: MCE-Step($G_{S+}$);
```

As shown in Algorithm 12 and Procedure 13, the extraction of extended subgraphs is processed at two levels, one aiming to minimize the I/O cost while the other aiming to reduce the CPU cost.

The first level is to extract bigger extended subgraphs, which is done in a similar way as in Algorithm 11 but by replacing Lines 6-9 with a call to Procedure 13. At this level, the size of each G_{S+} should be close to M to minimize the number of scans of G and hence the I/O cost.

Then, at the second level as shown in Procedure 13, we extract smaller extended subgraphs from each bigger extended subgraph H obtained at the first level. Whether or not to further extract smaller extended subgraphs is determined by the cost model given by Equation (Eq. 3.3).

Computing the optimal $\mathcal{G}_\alpha$ to maximize $TotalGain$ is similar to finding the optimal graph partition from H, which is APX-hard [9]. Thus, this process alone would dominate the overall cost of MCE. However, if we do not select the seed vertices from H randomly but rather select them sequentially, the selection process can be made much more efficient.

Another challenge of applying Equation (Eq. 3.3) is that Equation (Eq. 3.3) consists of Equations (Eq. 3.1) and (Eq. 3.2), both of which require the entire sets of extended subgraphs, $\mathcal{G}_\alpha$ and $\mathcal{G}_\beta$, to be known. Even if we select the seed vertices sequentially, there are still exponentially many different permutations of $\mathcal{G}_\alpha$ and $\mathcal{G}_\beta$.

To avoid the costly intermediate process of subgraph extraction, we consider the search space as a binary tree, where the set of nodes at each level of the tree is a set of extended subgraphs. We construct the binary tree level-wise (Lines 1-10). The first level consists of only one extended subgraph, i.e., H itself. The two children of any internal node in the tree are constructed by dividing the set of seed vertices of the parent sequentially into two sets of

Procedure 13 *MCE-Step*$(H = (V_H, E_H))$

```
 1. $\mathcal{G}_\alpha := \{H\}$;
 2. $TotalGain := \infty$;
 3. while($TotalGain > 0$)
 4.         $\mathcal{G}_\beta := \mathcal{G}_\alpha, \mathcal{G}_\alpha := \varnothing$;
 5.         for each $G_{S_\beta^+} \in \mathcal{G}_\beta$ do
 6.                 divide $S_\beta$ into two equal halves: $S_{\alpha 1}$ and $S_{\alpha 2}$;
 7.                 extract $G_{S_{\alpha 1}^+}$ and $G_{S_{\alpha 2}^+}$ from $G_{S_\beta^+}$;
 8.                 $\mathcal{G}_\alpha := \mathcal{G}_\alpha \cup \{G_{S_{\alpha 1}^+}, G_{S_{\alpha 2}^+}\}$;
 9.         compute $TotalGain$ from $\mathcal{G}_\alpha$ and $\mathcal{G}_\beta$;
10. $\mathcal{G}_\alpha := \mathcal{G}_\beta$;
11. for each $G_{S_\alpha^+} \in \mathcal{G}_\alpha$ do
12.         apply in-memory MCE to $G_{S_\alpha^+}$;
```

seed vertices with equal size, and then extract the corresponding extended subgraphs from the parent (the correctness of the extraction is guaranteed by Lemma 3.8). Then at each level, we compute $TotalGain$. We stop constructing a new level of the binary tree when there is no longer a gain. Then (in Lines 10-12), we take the corresponding $\mathcal{G}_\alpha$ as the set of extended subgraphs, and process MCE in each $G_{S_\alpha^+} \in \mathcal{G}_\alpha$ as is done before in Algorithm 11.

The above scheme for subgraph extraction, though heuristic, is effective and efficient for the following important reasons: first, it fits seamlessly into the design of the partition-based algorithm; second, it supports effective parallelization; lastly, it avoids extra overhead in re-assigning the vertex order or relabeling the vertices, which is necessary for the task of MCE when vertices are selected out of order into $G_{S_\alpha^+}$.

3.4.3 Parallel Maximal Clique Enumeration

Existing parallel algorithms for MCE [30, 58] are in-memory algorithms that require the entire graph to be resident in main memory of each computing node, and thus are not scalable as the graph size increases and main memory is not sufficient to hold the graph.

We adopt our partition-based MCE framework and propose a parallel algorithm for MCE that requires only limited memory at each computing node. We describe the algorithm, named

Algorithm 14 *ParMCE*

Input: a graph $G = (V, E)$
Output: all maximal cliques in G

```
 1. $S := \varnothing$, $S^+ := \varnothing$;
 2. for each $v \in V$ do
 3.     $S := (S \cup \{v\})$, $S^+ := (S^+ \cup \{v\} \cup adj(v))$;
 4.     if$((\phi_{deg} \cdot |S^+|) \geq cM)$
 5.         scan $G$ once to extract $G_{S^+}$;
 6.         while(no idle computing node)   wait;
 7.         distribute the task MCE-Job$(G_{S^+})$
                to any idle computing node;
 8.         $S := \varnothing$, $S^+ := \varnothing$;
```

Procedure 15 *MCE-Job*$(H = (V_H, E_H))$

```
 1. $\mathcal{G}_\alpha := \{H\}$;
 2. $TotalGain := \infty$;
 3. while$((TotalGain > 0)$ or
            (# of idle computing nodes $> |\mathcal{G}_\beta|$))
 4.     $\mathcal{G}_\beta := \mathcal{G}_\alpha$, $\mathcal{G}_\alpha := \varnothing$;
 5.     for each $G_{S_\beta^+} \in \mathcal{G}_\beta$ do
 6.         divide $S_\beta$ into two equal halves: $S_{\alpha 1}$ and $S_{\alpha 2}$;
 7.         extract $G_{S_{\alpha 1}^+}$ and $G_{S_{\alpha 2}^+}$ from $G_{S_\beta^+}$;
 8.         $\mathcal{G}_\alpha := \mathcal{G}_\alpha \cup \{G_{S_{\alpha 1}^+}, G_{S_{\alpha 2}^+}\}$;
 9.     compute $TotalGain$ from $\mathcal{G}_\alpha$ and $\mathcal{G}_\beta$;
10. $\mathcal{G}_\alpha := \mathcal{G}_\beta$;
11. for each $G_{S_\alpha^+} \in \mathcal{G}_\alpha$ do
12.     while(no idle computing node)   wait;
13.     distribute $G_{S_\alpha^+}$ to an idle computing node, $X$, and
        compute in-memory MCE in $G_{S_\alpha^+}$, further parallelizing
        the task among multiple cores or threads within $X$,
        or any other idle computing node(s), if any;
```

as **ParMCE** (for **Par**allel **MCE**), in Algorithm 14 and Procedure 15.

Algorithm 14 uses one computing node as the *master node*. The master node reads the input graph G from disk, extracts extended subgraphs from G, and distributes them to the idle nodes, if any. To utilize parallelism (e.g., multi-cores, hyper threading, etc.) within the master node, we create two threads in the master node. One thread extracts extended subgraphs and pushes it into the *pool* of un-computed subgraphs, and the other one takes the subgraphs from the *pool* and sends them to any available idle nodes. In case that the total number of available computing nodes becomes too large so that the computation at the master node might become a bottleneck, our algorithm easily allows us to split the input graph and create multiple master nodes to serve the increasing number of computing nodes.

Procedure 15 then describes that when a computing node receives an extended subgraph H (distributed by Algorithm 14), it further extracts smaller extended subgraphs to improve the efficiency of MCE, as done in Procedure 13. If we have extra idle computing nodes, we may further divide extended subgraphs into smaller ones to maximize parallelization (note that the bottleneck is at the MCE computation rather than at the subgraph extraction, both of which are now performed in-memory). This task itself can also be easily parallelized, by simply distributing the extended subgraphs in $\mathcal{G}_\beta$ to new computing nodes and call Procedure 15. But since this process is not the bottleneck, it should be parallelized only when computing nodes are excessive.

After we obtain the set of extended subgraphs $\mathcal{G}_\alpha$, we distribute each small extended subgraph $G_{S_\alpha^+} \in \mathcal{G}_\alpha$ to an idle computing node, X. Then at X, we apply the in-memory algorithm to process MCE in $G_{S_\alpha^+}$. The fine-grained parallelization of MCE proposed in [58] can be further applied if X has multiple cores or threads, or if there are more idle computing nodes available.

3.5 Experimental Evaluation

In this section, we evaluate the performance of our algorithms. We compared with the algorithm that has the optimal time complexity for MCE in memory proposed by Tomita et al. [62] (denoted by *TomitaTT*), a recent algorithm for MCE in d-degenerate graphs proposed by Eppstein et al. [33, 34] (denoted by *EppsteinLS*), and the I/O-efficient MCE algorithm by Cheng et al. [24] (denoted by *ChengKFYZ*).

For all the sequential programs, we ran the experiments on a machine with an Intel Xeon 2.67GHz CPU and 4GB RAM, running CentOS 5.4 (Linux). For the parallel program (using the MPI), we ran the experiments on a cluster, with each computing nodes having 2.93GHz CPU and 4GB RAM.

Datasets. We use the following four datasets: `blog`, `LJ`, `Web`, and `BTC`. The `blog` network is collected from the top-15 popular queries published by Technorati (technorati.com) every three hours from Nov 2006 to Mar 2008. For each query, the top-50 results are retrieved. In the `blog` network, vertices are blogs and edges indicate that two blogs appear in the search result of the same query. `LJ` is the free online community called *LiveJournal*, where vertices are members and edges represent friendship between members. The `LJ` dataset is available from snap.stanford.edu. The `Web` graph is obtained from the YAHOO webspam dataset (barcelona.research. yahoo.net/webspam), where vertices are webpages and edges are hyperlinks. The `BTC` dataset is a semantic graph converted from the *Billion Triple Challenge 2009 RDF* dataset (http://vmlion25.deri.ie), where each vertex represents an object such as a person, a document, and an event, and each edge represents the relationship between two vertices such as "as-author", "links-to", and "has-title". We list some details of the datasets (number of vertices and edges, physical storage size) in Table 3.1.

Table 3.1: Datasets (M = 1,000,000)

	`blog`	`LJ`	`Web`	`BTC`
$\|V\|$	1M	4.8M	52.9M	165M
$\|E\|$	6.5M	43M	274.8M	773M
Storage size	186MB	1310MB	5GB	14.5GB

Among the four datasets, `blog` and `LJ` are two relatively smaller graphs, while `Web` and `BTC` are two larger graphs. We use the smaller graphs to compare the performance of our algorithm with the in-memory algorithms, while we use the larger graphs to evaluate the performance of our algorithms when memory of a single machine is insufficient to hold the input graph.

3.5.1 Effectiveness of Cost Model

We first examine whether the reduction of set intersection cost by extracting smaller extended subgraphs, rather than extracting extended subgraphs that fill the available memory, can indeed improve the efficiency of MCE computation. We also assess the effectiveness of our cost model, given in Section 3.4.2, for choosing the right size for the extended subgraphs to be extracted. We test the settings on *SeqMCE*, i.e., Algorithm 12.

Figure 3.3 reports the running time of *SeqMCE* for `blog` and `Web`. We extract extended subgraphs of size from 0.001% to 10% of the available memory size at the second level of Algorithm 12, i.e., each $G_{S_\alpha^+} \in \mathcal{G}_\alpha$ in Lines 11-12 of Procedure 13 is of size xM, where $x \in \{0.001\%, 0.01\%, 0.1\%, 1\%, 10\%\}$ and M is the available memory size, which is set to 1 GB in this experiment. We also report the running time of *SeqMCE* that extracts extended subgraphs by our cost model, represented by "**CM**" on the x-axis. From Figure 3.3, the sizes of the extended subgraphs extracted by "**CM**" are around 2% for `blog` and 0.05% for `Web`.

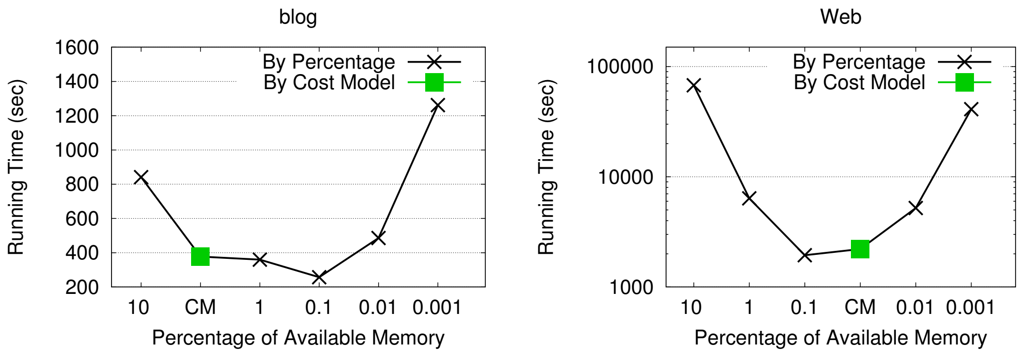

Figure 3.3: Running time of *SeqMCE* for MCE with varying sizes of extended subgraphs

The results show that for both graphs, the efficiency of MCE is first improved significantly when smaller extended subgraphs are used at the second level of the algorithm, but there is an optimal point after which further decreasing the size of the extended subgraphs reports a significantly deteriorated performance. The trend of the running time for different sizes of extended subgraphs can be explained by our analysis of the gain and loss, i.e., Equation (Eq. 3.1) and Equation (Eq. 3.2), in Section 3.4.2, which represent a tradeoff between *(1) the cost of set intersection* and *(2) the cost of subgraph extraction and search tree construction*. The same tradeoff can be also be clearly observed from the `LJ` and `BTC` graphs (details omitted due to lack of space).

Having shown the tradeoff, we show that the results obtained by the cost model is near the optimal point. Relatively speaking, the running time reported for the cost model for `blog` is further away from the optimal point than that for `Web`. This is because the `blog` graph is small and hence determining the right subgraph size by the cost model takes a considerable portion of the total time. When the input graph is large, the cost of computation involving the cost model becomes negligible compared to the cost of MCE.

In the subsequent experiments, the algorithm *SeqMCE* always extracts extended subgraphs by the cost model.

3.5.2 Performance of Sequential Algorithms

We now compare our algorithm *SeqMCE* with other state-of-the-art sequential algorithms for MCE.

Table 3.2: Running time (wall-clock time in seconds)

	`blog`	`LJ`	`Web`	`BTC`
TomitaTT [62]	11,876	150,122	–	–
EppsteinLS [33]	68	67	–	–
ChengKFYZ [24]	329	6,941	22,840	134,951
SeqMCE	378	175	2,209	28,410

Table 3.2 reports the running time of the four algorithms. For the smaller datasets, `blog` and `LJ`, *EppsteinLS* is the fastest. The *EppsteinLS* algorithm uses a d-degeneracy ordering[2]

[2]A graph is d-degenerate if it has a maximum k-core number of d [34].

Table 3.3: Peak memory consumption (in GB)

	blog	LJ	Web	BTC
TomitaTT [62]	0.2	0.9	–	–
EppsteinLS [33]	1.3	4.0	–	–
ChengKFYZ [24]	0.1	1.0	1.0	1.0
SeqMCE	0.1	1.0	1.0	1.0

to limit the depth of recursive calls in the *TomitaTT* algorithm to d, where d is not large for many real-world sparse graphs. Our algorithm *SeqMCE* can effectively reduce the cost of set intersection as well as limit the depth of recursive calls; however, it is a semi-streaming algorithm designed for processing large datasets, while *EppsteinLS* is an in-memory algorithm. Thus, *SeqMCE* needs to scan the input graph many times, which explains why it is slower than *EppsteinLS*. However, *EppsteinLS* uses significantly more memory than *SeqMCE* and ran out of memory for the larger graphs, `Web` and `BTC`. On the contrary, with the available memory set at only 1 GB, *SeqMCE* still processes MCE efficiently on both large graphs, and is up to an order of magnitude faster than the state-of-the-art I/O-efficient MCE algorithm, *ChengKFYZ*.

Compared with the in-memory algorithm *TomitaTT*, which also ran out of memory for `Web` and `BTC`, *SeqMCE* is up to two orders of magnitude faster for processing `blog` and `LJ`. Since the smaller datasets can be processed in memory, the main difference between *SeqMCE* and *TomitaTT* for in-memory processing is the new adoption of cost reduction on set intersection, we conclude that the performance improvement of *SeqMCE* over *TomitaTT* is mainly due to our proposal of cost reduction on set intersection. This result thus further demonstrates the effectiveness of the CPU cost reduction method proposed in Section 3.4.2.

3.5.3 Performance of Parallel Algorithm

We now evaluate the performance of the parallel algorithm (i.e., Algorithm 14), denoted by *ParMCE*. There are some recent algorithms for parallel MCE [30, 58, 70], but we were not able to obtain their code for comparison.

We report the running time of *ParMCE* in Figure 3.4. To show that our method is truly effective in handling graphs that cannot fit in memory, we set the available memory to only 64 MB for `blog` and `LJ`, and 256 MB for the two large graphs `Web` and `BTC`.

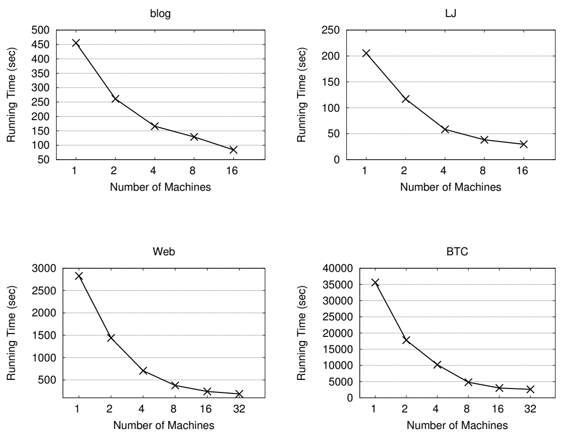

Figure 3.4: Running time (wall-clock time in seconds) of *ParMCE*

The figures show that when two machines are used, the running time is nearly halved for all the datasets. The efficiency continues to improve when more machines are used, although the speed-up becomes milder when more machines are used. With only eight machines, we can finish computing MCE in the largest dataset in about an hour.

3.6 Related Work

Maximal clique enumeration has been studied extensively and comprehensive reviews can be found in [21] and [17] (the latter also discusses maximum clique finding). The first algorithms were the *backtracking* method [6, 18] that use $O(|V|^2)$ memory space. Then effective pruning strategy by selecting good *pivots* was employed to further reduce the search space [21, 48, 62]. Algorithm for d-degenerate graphs was also proposed [33, 34], which achieves a time complexity of $O(d|V|3^{d/3})$ by utilizing a d-degeneracy ordering. However, all these studies did not focus on reducing the memory complexity and require $\Omega(|V| + |E|)$ memory space, which may not be practical for large disk-resident graphs.

Recently, several parallel algorithms were proposed for MCE. Most of these parallel algorithms [30, 58] still require the entire input graph to be resident in main memory of each computing node, and thus are not practical for processing large graphs that cannot fit in memory. A MapReduce algorithm [70] was also proposed for MCE. The MapReduce model, however, is slow in practice for enumeration tasks such as MCE and triangle listing in large graphs, mainly because of the huge amount of intermediate data produced in the shuffling phase between Mappers and Reducers.

Algorithm for output-sensitive MCE was also introduced which is based on reverse search [64]. The time delay was reduced to $O(d_{max}^4)$ for sparse graphs using matrix multiplication [51], where d_{max} is the maximum degree of a graph; but the algorithm requires $O(|V| \cdot |E|)$ preprocessing time. Other algorithms, such as computing a k-clique by joining two $(k-1)$-cliques [49], by making use of triangles [65], and enumerating maximal cliques of size larger than a threshold [53], were also studied. However, all these algorithms require memory space at least $\Omega(|V| + |E|)$.

Stix [59] proposed a streaming algorithm that updates the set of maximal cliques upon each edge insertion, but the update is expensive and it also requires to keep all maximal cliques in memory, which has been verified in [22].

Recently, Cheng et al. [22, 24] proposed an efficient algorithm that recursively extracts a core part of the input graph for local MCE computation. This idea is similar to the algorithm described in Section 3.4.1, but a fundamental difference in the formulation of the subgraphs to be extracted, i.e., their subgraphs still need to access the input graph for MCE while ours do not, making it nontrivial to parallelize their algorithm as what we do. We have also verified in Section 3.5 that our algorithm is significantly more efficient.

Other related works that also avoid random disk access by extracting small subgraphs include core/truss decomposition [23, 66], triangle listing [25, 26]. However, the subgraphs used in these works cannot guarantee the maximality of the cliques computed and their algorithm frameworks are also not suitable for the task of MCE.

3.7 Conclusions

We presented efficient algorithms to reduce both the I/O cost and CPU cost of MCE in massive networks. We verified the performance of our algorithms comparing with the state-of-the-art algorithms for MCE [22, 24, 33, 34, 62]. Our results first demonstrate that, by reducing the cost of set intersection in MCE, we are able to achieve significant speed-ups in both cases when the input graph can fit in main memory or cannot fit in main memory. Then, we also showed that our parallel algorithm for MCE significantly speeds up the MCE computation compared with the sequential algorithm.

Chapter 4

Conclusions and Future Directions

We studied enumerating substructure patterns in large graphs. These problem are important since they are fundamental problems in graph analysis and are widely used in various graph analysis applications. However, as the graph cannot fit into main memory, these problems become challenging due to the huge performance gap between main memory and external memory.

We focused on two most fundamental substructures, triangles and cliques, and proposed efficient I/O efficient or parallel algorithms for enumerating them in large graphs efficiently. Our contributions can be summarized as the following.

- We developed the first I/O-efficient algorithm for listing triangles in disk resident massive graphs. We also showed that our algorithm can be pipelined and thus can be applied to the computation of clustering coefficient, transitivity, triangular connectivity, etc. The performance of our algorithms is impressing. On same dataset, the running time of this algorithm on a single computer is even faster than state-of-the-art parallel algorithm on a cluster of more than 1,000 computer.

- We developed a general algorithm framework to guarantee the performance in networks with different characteristics and worked out a non-trivial parallel version of this algorithm to better utilize the computing resource available. Our algorithms are up to 1,000 times faster than the state-of-the-art algorithms for clique enumeration in disk-resident networks with a single machine and shows near linear scale up given more machines.

In the future, we would like to study more fundamental substructures, such as sterner tree, k-truss and cycles. We also plan to use the algorithms we developed to higher level applications such as link prediction, community detections and abnormal subgraph detections, etc.

Publication List:

- **Shumo Chu** and James Cheng. Triangle Listing in Massive Network. To appear in *ACM Transcation on Knowledge Discovery from Data* (**TKDD**), 2012.

- James Cheng, Linhong Zhu, Yiping Ke and **Shumo Chu**. Fast Algorithms for Maximal Clique Enumeration with Bounded Memory. In proceedings of *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (**KDD**), Pages 1240-1248, 2012.

- James Cheng, Yiping Ke **Shumo Chu** and Carter Cheng. Efficient Processing of Distance Queries in Large Graphs: A Vertex Cover Approach. In proceedings of *ACM SIGMOD Conference* (**SIGMOD**),Pages 457-468, 2012.

- **Shumo Chu** and James Cheng. Triangle Listing in Massive Networks and Its Applications. In proceedings of *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (**KDD**), pages 672-680, 2011.

- James Cheng, Yiping Ke, **Shumo Chu** and Tamer Öszu. Core Decomposition in Massive Networks. In proceedings of *IEEE International Conference on Data Engineering*(**ICDE**), Pages 51-62, 2011.

References

[1] . URL http://en.wikipedia.org/wiki/DDR3_SDRAM.

[2] . URL http://en.wikipedia.org/wiki/Hard_disk_drive_performance_characteristics.

[3] A. Abou-Rjeili and G. Karypis. Multilevel algorithms for partitioning power-law graphs. In *IPDPS*, 2006.

[4] F. N. Abu-Khzam, N. E. Baldwin, M. A. Langston, and N. F. Samatova. On the relative efficiency of maximal clique enumeration algorithms, with applications to high-throughput computational biology. In *International Conference on Research Trends in Science and Technology*, 2005.

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

[6] E. A. Akkoyunlu. The enumeration of maximal cliques of large graphs. *SIAM J. Comput.*, 2(1):1–6, 1973.

[7] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):354–364, 1997.

[8] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.*, 58(1):137–147, 1999.

[9] K. Andreev and H. Racke. Balanced graph partitioning. In *SPAA*, pages 120–124, 2004.

[10] Z. Bar-Yossef, R. Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *SODA*, pages 623–632, 2002.

[11] V. Batagelj and A. Mrvar. A subquadratic triad census algorithm for large sparse networks with small maximum degree. *Social Networks*, 23(3):237–243, 2001.

[12] V. Batagelj and M. Zaveršnik. Short cycle connectivity. *Discrete Mathematics*, 307(3-5): 310 – 318, 2007.

[13] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *KDD*, pages 16–24, 2008.

[14] H. R. Bernard, P. D. Killworth, and L. Sailer. Informant accuracy in social network data iv: a comparison of clique-level structure in behavioral and cognitive network data. *Social Networks*, 2(3):191–218, 1979.

[15] N. M. Berry, T. H. Ko, T. Moy, J. Smrcka, J. Turnley, and B. Wu. Emergent clique formation in terrorist recruitment. In *The AAAI-04 Workshop on Agent Organizations: Theory and Practice*, 2004.

[16] V. Boginski, S. Butenko, and P. M. Pardalos. Statistical analysis of financial networks. *Computational Statistics & Data Analysis*, 48(2):431–443, 2005.

[17] I. M. Bomze, M. Budinich, P. M. Pardalos, and M. Pelillo. The maximum clique problem. In *Handbook of Combinatorial Optimization*, pages 1–74. Kluwer Academic Publishers, 1999.

[18] C. Bron and J. Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Commun. ACM*, 16(9):575–577, 1973.

[19] G. Buehrer and K. Chellapilla. A scalable pattern mining approach to web graph compression with communities. In *WSDM*, pages 95–106, 2008.

[20] L. S. Buriol, G. Frahling, S. Leonardi, A. Marchetti-Spaccamela, and C. Sohler. Counting triangles in data streams. In *PODS*, pages 253–262, 2006.

[21] F. Cazals and C. Karande. A note on the problem of reporting maximal cliques. *Theor. Comput. Sci.*, 407(1-3):564–568, 2008.

[22] J. Cheng, Y. Ke, A. W.-C. Fu, J. X. Yu, and L. Zhu. Finding maximal cliques in massive networks by h*-graph. In *SIGMOD Conference*, pages 447–458, 2010.

[23] J. Cheng, Y. Ke, S. Chu, and M. T. Özsu. Efficient core decomposition in massive networks. In *ICDE*, pages 51–62, 2011.

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

[25] S. Chu and J. Cheng. Triangle listing in massive networks. *To appear in ACM Transactions on Knowledge Discovery from Data, 2012.*

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

[27] J. Cohen. Graph twiddling in a mapreduce world. *Computing in Science and Engineering*, 11:29–41, 2009. ISSN 1521-9615. doi: http://doi.ieeecomputersociety.org/10.1109/MCSE.2009.120.

[28] D. Coppersmith and R. Kumar. An improved data stream algorithm for frequency moments. In *SODA*, pages 151–156, 2004.

[29] G. Creamer, R. Rowe, S. Hershkop, and S. J. Stolfo. Segmentation and automated social hierarchy detection through email network analysis. In *WebKDD/SNA-KDD*, pages 40–58, 2007.

[30] N. Du, B. Wu, L. Xu, B. Wang, and P. Xin. Parallel algorithm for enumerating maximal cliques in complex network. In *Mining Complex Data*, pages 207–221. 2009.

[31] J.-P. Eckmann and E. Moses. Curvature of co-links uncovers hidden thematic layers in the world wide web. *PNAS*, 99:5825–5829, 2002.

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

[33] D. Eppstein and D. Strash. Listing all maximal cliques in large sparse real-world graphs. In *SEA*, pages 364–375, 2011.

[34] D. Eppstein, M. Löffler, and D. Strash. Listing all maximal cliques in sparse graphs in near-optimal time. In *ISAAC (1)*, pages 403–414, 2010.

[35] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In *SIGCOMM*, pages 251–262, 1999.

[36] K. Faust and S. Wasserman. Social network analysis: Methods and applications. *Cambridge University Press*, 1995.

[37] U. Feige and R. Krauthgamer. A polylogarithmic approximation of the minimum bisection. In *FOCS*, pages 105–115, 2000.

[38] U. Feige, R. Krauthgamer, and K. Nissim. Approximating the minimum bisection size (extended abstract). In *STOC*, pages 530–536, 2000.

[39] C. M. Fiduccia and R. M. Mattheyses. A linear time heuristic for improving network partitions. In *IEEE Design Automation Conference*, 1982.

[40] B. Fritzke. A self-organizing network for unsupervised learning. *TR-03-026*, 42, 1993.

[41] K. Gouda and M. J. Zaki. Efficiently mining maximal frequent itemsets. In *ICDM*, pages 163–170, 2001.

[42] M. Granovetter. The strength of weak ties. *American Journal of Sociology*, 78(6):1360–1380, 1973.

[43] A. Itai and M. Rodeh. Finding a minimum circuit in a graph. In *STOC*, pages 1–10, 1977.

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

[45] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning for irregular graphs. *SIAM Review*, 41(2):278–300, 1999.

[46] B. W. Kernigham and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, pages 291–308, 1970.

[47] D. E. Knuth. Estimating the efficiency of backtrack programs. *Mathematics of Computation*, 29(129):121–136, 1975.

[48] I. Koch. Enumerating all connected maximal common subgraphs in two graphs. *Theor. Comput. Sci.*, 250(1-2):1–30, 2001.

[49] F. Kose, W. Weckwerth, T. Linke, and O. Fiehn. Visualizing plant metabolomic correlation networks using clique-metabolite matrices. *Bioinformatics*, 17(12):1198–1208, 2001.

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

[51] K. Makino and T. Uno. New algorithms for enumerating all maximal cliques. In *SWAT*, pages 260–272, 2004.

[52] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network Motifs: Simple Building Blocks of Complex Networks. *Science*, 298(5594):824–827, 2002.

[53] N. Modani and K. Dey. Large maximal cliques enumeration in sparse graphs. In *CIKM*, pages 1377–1378, 2008.

[54] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, Vol. 45, No. 2:167–256, 2003.

[55] M. E. J. Newman, D. J. Watts, and S. H. Strogatz. Random graph models of social networks. *PNAS*, 99:2566–2572, 2002.

[56] T. Schank. Algorithmic aspects of triangle-based network analysis. *Ph.D. Dissertation, Universität Karlsruhe, Fakultät für Informatik*, 2007.

[57] T. Schank and D. Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *WEA*, pages 606–609, 2005.

[58] M. C. Schmidt, N. F. Samatova, K. Thomas, and B.-H. Park. A scalable, parallel algorithm for maximal clique enumeration. *J. Parallel Distrib. Comput.*, 69(4):417–428, 2009.

[59] V. Stix. Finding all maximal cliques in dynamic graphs. *Computational Optimization and applications*, 27:173–186, 2004.

[60] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *WWW*, pages 607–614, 2011.

[61] G. Taubin and J. Rossignac. Geometric compression through topological surgery. *ACM Trans. Graph.*, 17(2):84–115, 1998.

[62] E. Tomita, A. Tanaka, and H. Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theor. Comput. Sci.*, 363(1):28–42, 2006.

[63] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos. Doulion: counting triangles in massive graphs with a coin. In *KDD*, pages 837–846, 2009.

[64] S. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM J. Comput.*, 6(3):505–517, 1977.

[65] L. Wan, B. Wu, N. Du, Q. Ye, and P. Chen. A new algorithm for enumerating all maximal cliques in complex network. In *ADMA*, pages 606–617, 2006.

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

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

[68] S. Wasserman and K. Faust. Social network analysis: Methods and applications. *Cambridge University Press*, 1994.

[69] D. J. Watts and S. H. Strogatz. Collective dynamics of ‘small-world’ networks. *Nature*, 393(6684):440–442, 1998.

[70] B. Wu, S. Yang, H. Zhao, and B. Wang. A distributed algorithm to enumerate all maximal cliques in mapreduce. In *Proceedings of the Fourth International Conference on Frontier of Computer Science and Technology*, pages 45–51, 2009.