UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
DEPARTAMENTO DE INFORMÁTICA TEÓRICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

BRUNO MENEGOLA

An External Memory Algorithm for Listing Triangles

Trabalho de Graduação

Luciana Salete Buriol
Orientadora

Marcus Ritt
Co-orientador

Porto Alegre, 2 de Julho de 2010

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

In this work, we propose a new external memory algorithm for counting and listing triangles in huge graphs. Another major contribution is a better analysis of the external memory algorithm for listing triangles proposed by Roman Dementiev [8]. Besides that, there is a review of solutions (in internal and external memory) for the problems of finding, counting and listing triangles.

Many real world applications need to process a large amount of data. Dealing with massive data is a challenge for algorithm design. The goal for external memory algorithms, or I/O-efficient algorithms, is to reduce the number of reads and writes (I/Os) to external devices, typically a hard disk, wich has a huge cost per I/O when compared to another made for internal memory. The external device is used to store informations that the main memory, usually a RAM (Random Access Memory), can not deal because it lacks of space.

In a graph, a triangle is a set of three vertices such that each possible edge between them is present. Usually, the number of triangles, for example, is not an useful information by itself. However, they can be used for other purposes like the calculation of graph properties, for example, the clustering coefficient and transitivity coefficient; analysis of complex networks; finding special subgraphs, for example, in protein interaction networks; and also intrusion detection, spamming or community detection.

In a graph with m edges, the I/O complexity of our proposed algorithm is $O(\mathrm{Scan}(m^{\frac{3}{2}}))$. With this algorithm is possible to count the number of triangles in a graph with 800 million edges in about 9 hours using only 1.5 GiB of main memory.

Keywords: Algorithms, external memory, triangles, triangle problems, counting triangles, listing triangles.

Um Algoritmo de Memória Externa para Listagem de Triângulos

RESUMO

Este trabalho propõe um novo algoritmo de memória externa para contagem e listagem de triângulos em grafos massivos. Outra grande contribuição é uma melhor análise do algoritmo de listagem de triângulos de memória externa proposto por Roman Dementiev [8]. Além disso, há uma revisão bibliográfica das soluções (tanto para memória interna, quanto para memória externa) para os problemas de busca, contagem e listagem de triângulos.

Muitas aplicações atuais precisam processar uma quantidade imensa de dados. Lidar com esse problema é um desafio para a área de projeto de algoritmos. Algoritmos de memória externa, ou algoritmos I/O-eficientes, objetivam reduzir o número de leituras e escritas (I/Os) na mídia externa, tipicamente um disco rígido, pois o custo de um I/O para um dispositivo esterno é muito maior que um I/O realizado em memória interna. A mídia externa é utilizada para armazenar as informações que a memória principal, normalmente uma RAM (Random Access Memory), não consegue lidar por falta de espaço.

Em grafos, Triângulos são conjuntos de 3 vértices tal que cada possível aresta entre eles está presente. Usualmente, o número ou a lista de triângulos em um grafo não é uma informação útil por si só. Entretanto ela pode ser utilizada para outros propósitos como o cálculo de propriedades do grafo, por exemplo, o coeficiente de clustering ou o coeficiente de transitividade; análise de redes complexas; busca de outros subgrafos especiais, por exemplo, em redes de interação entre proteínas; e também detecção de intrusão, de comunidades e de atividades de spam.

Em um grafo com m arestas, a complexidade de I/O do algoritmo que propomos é $O(\mathrm{Scan}(m^{\frac{3}{2}}))$. Com o algoritmo proposto, é possível calcular o número de triângulos em um grafo com 800 milhões de arestas em pouco mais de 9 horas usando apenas 1.5GiB de memória principal.

Palavras-chave: Algoritmos, memória externa, triângulos, contagem de triângulos, listagem de triângulos.

1 INTRODUCTION

Many real world applications need to process a large amount of data. Examples of such applications are indexing of web pages, books, images, videos and all the terabytes of media produced every year; processing information from sensor networks with ambiental parameters in a production process control and storing, compressing and searching in biological databases for study of proteins. However, if a large amount of data cannot be properly explored, there is no reason for storing it. Dealing with massive data is a challenge for algorithm design.

In the last few years it became common, and in many cases mandatory, to design algorithms based on machine models that were proposed for dealing with large data sets. This is the case of external memory algorithms (also referred to I/O-efficient algorithms) [23, 22], cache oblivious algorithms [18], and streaming algorithms [1].

The first two models explore the memory hierarchy of computers. *External memory (EM) algorithms* are the ones that explicitly manage data placement and transfer from and to EM devices. Since the main memory is not enough to store massive data, one must use a secondary storage – a hard disk, for example – to store it. However, all systems require that the information is processed in main memory. Thus, the main memory may serve as a cache for the external device. When some data is required, it is read from disk to main memory. When altered data is no longer needed, it is stored back on disk again. The goal for an EM algorithm is to reduce the number of reads and writes to external devices. The inputs and outputs (I/Os) cost a lot more time than a read and write to a Random Access Memory, for example. EM algorithms design aims to reduce those I/Os by increasing locality of processing. This way the main memory can be efficiently used. Increased locality means that, once data is loaded into main memory, it should be processed as much as possible with no other or few I/Os.

External memory algorithms store massive data in external devices. There is a special case, *semi-external memory algorithms*, that classifies algorithms that uses some data structure in main memory for potential massive data. For example, in a graph processing algorithm, very often some structure of size proportional to the number of vertices can be stored in main memory, but not one proportional to the number of edges.

Cache oblivious algorithms aim to minimize communication between RAM and cache. External memory and cache oblivious algorithms are very similar, since both handle two levels of memory, but usually the later use a recursive approach for problem solving.

Streaming algorithms, on the other hand, estimate properties of data that is so large that it cannot be stored, and that arrives continually element by element.

Many problems can be modeled by graphs, and if the application requires massive processing, the algorithms have to be designed using a proper model. Among the three cited models, external memory algorithms are the ones which has frequently been used on graphs. Many EM algorithms for graphs have low I/O complexity, while maintaining the time complexity of the best algorithm known for internal memory. Particularly in the last decade, the research in this field brought many important contributions. We refer to [16] for data structures and basic graph properties in external memory. Moreover, more advanced graph problems were efficiently solved in external memory, such as graph search [15], minimum spanning trees [3, 10], single source shortest paths [3], among others.

Some algorithms have not been studied much in external memory yet. This is the case of the basic triangles problems. In a graph, a triangle is a set of three vertices such that each possible edge between them is present. The basic triangles problems are *finding*, *counting* and *listing*. They compute, respectively, if there is a triangle in the graph, how many triangles exist and which nodes compose each triangle.

Usually, the number of triangles, for example, is not an useful information by itself. But they are the basis for other graph informations and properties. Their computation plays a key role in the analysis of complex networks. Important statistical properties such as the clustering coefficient and the transitivity coefficient of graphs depend on the number of triangles. Triangles are also used to find special subgraphs in large graphs, like in protein interaction networks. Tsourakakis [21] cites other applications, like finding patterns in large networks for intrusion detection, spamming or community detection.

The best known methods for counting the number of triangles in a general graph use matrix multiplication ($O(n^\omega)$ in time complexity, where ω is the exponent for the best known algorithm for matrix multiplication. Currently, $\omega = 2.376$). Even on graphs of medium size this is not computationally feasible because of the large main memory usage required to store the whole graph (an adjacency matrix requires $O(n^2)$ in space). Schank and Wagner [19] gave an experimental study of algorithms in main memory for counting and listing triangles and computing the clustering coefficient in small and medium size graphs. Their algorithms work in main memory and report results for graphs up to 668 thousand nodes. More recently, Latapy [12] presented new algorithms for finding, counting and listing triangles, which perform better on power-law-graphs. Computational results are presented for graphs up to 39.5 million nodes.

Among the three cited models for dealing with massive data, the data stream model is applied for estimating the number of triangles [6]. We are not aware of any cache oblivious algorithm for this problem. An external memory algorithm was proposed by Dementiev [7] as part of his doctoral thesis [8].

In this thesis we propose a new external memory algorithm for listing all triangles in an undirected graph. Our algorithm is based on the main memory algorithm named *compact-forward* proposed by Latapy [12].

The main contributions of this thesis are:

1. A better analysis of Dementiev's external memory algorithm for listing triangles.

2. A new external memory algorithm for counting and listing triangles in a graph. With an implementation of the algorithm, we also present an experimental comparison with Dementiev's algorithm and Latapy's *compact-forward* main memory algorithm as a baseline for the tests.

1.1 Notation

Throughout the paper, we consider a graph $G = (V, E)$ with a set of vertices V and a set of edges E. We denote the number of vertices by $n = |V|$ and the number of edges by $m = |E|$. G is unweighted, has no self-loops and we make no distinction between two edges (u, v) and (v, u), so the graph is undirected. We denote by $N(u) = \{v \in V | (u, v) \in E\}$ the neighborhood of vertex $u \in V$ and by $d(u) = |N(u)|$ the node degree. The maximum degree of the graph is represented by Δ.

A bipartite graph is a graph G whose vertices can be partitioned into two sets: $V_1 \subseteq V$ and $V_2 \subseteq V$. The set of edges E must fullfil the following criterion: $\forall (u, v) \in E : u \neq v \implies u \in V_1 \wedge v \in V_2$.

The clustering coefficient is defined as the normalized sum of the fraction of neighbor pairs of a vertex v of the graph that form a triangle with v, while the related transitivity coefficient of a graph is defined as the ratio between three times the number of triangles and the number of paths of length two in the graph.

1.2 Overview

This thesis is organized as follows. In Chapter 2 we review the state of the art for external memory algorithms analysis and development. We also review the current solutions for main and external memory algorithms for listing triangles, including Roman Dementiev's EM algorithm in Section 2.3.2. In Chapter 3 we present the first part of the above mentioned contributions: Dementiev's algorithm analysis and our new triangles listing algorithm. In Chapter 4 we expose the experimental results of our algorithm and the comparison with the others. In Chapter 5 is concludes and discusses future works.

2 STATE OF THE ART

In this chapter we will briefly review key subjects, that will be used in the construction of our algorithm, such as the Parallel Disk Model (Section 2.1) and libraries of external memory data structures (Section 2.2). Also, the current solutions for the triangles problems will be presented in Section 2.3.

2.1 A Model For I/O Complexity Analysis

External memory algorithms usually make use of hard disks (HDs) to store the data that does not fit in main memory. It is important for algorithm designers to have a model of those disks such that different algorithms can be compared and optimized properly. The model must capture the essence of the media to create a simple but accurate way to analyze algorithms. To understand the source of the I/O problem that external memory algorithms attempt to address, Section 2.1.1 remembers the functional aspects of hard disks. The model used for I/O complexity analysis will be presented in Section 2.1.2.

2.1.1 Hard Disks Aspects

As shown in Figure 2.1, a hard disk is composed of several platters rotating at constant angular velocity and movable arms, which hold heads to read and write data. A circle of sectors on each platter is called a track. The virtual set of tracks from different platters but at the same distance from the center is called a cylinder. Usually more platters are used to increase available disk space, not to improve the transfer rate through read/write parallelism, since only one track is read or written at a time [22].

When some data is requested from or written to the HD, the device must locate the correct place to load or store it. The movement of the arm to the correct cylinder is called a seek. The time waiting for the correct sector to pass below the arm head is called *rotational latency*. The average latency is half of the time needed for a full disk revolution.

The latency of accessing data is the combination of seek time and rotational latency. A seek time is on the order of 3 to 10 milliseconds. The rotational latency depends on the rotational speed of the disk. For example, a 7.200 rpm disk makes a revolution every 8.34 milliseconds and so the rotational latency is about 4.17 milliseconds. It is important to note that, while the latency of accessing data in a hard disk is in the order of milliseconds, the access time in a Random Access Memory (RAM) is in the order of nanoseconds. So, HD accesses are a million times

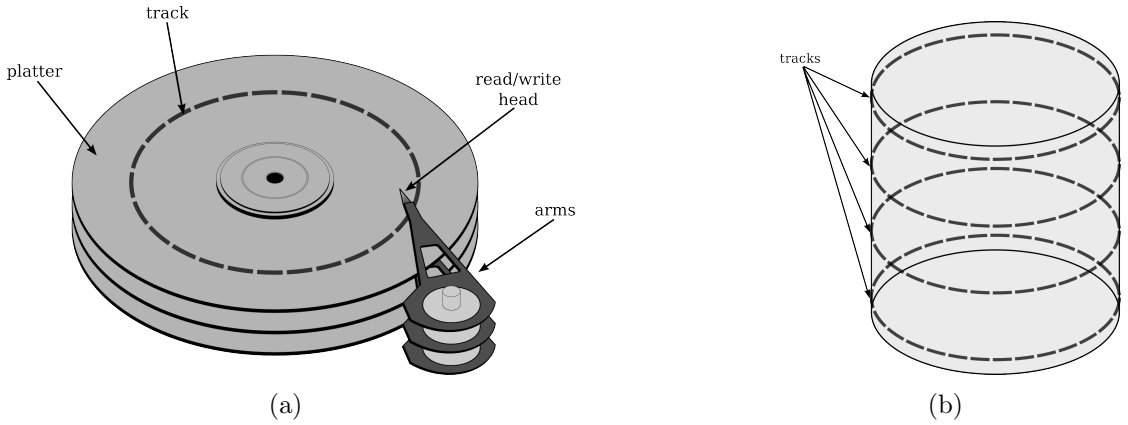

Figure 2.1: Hard Disk drive schema [22]. (a) Platters spin synchronously while data is read/written by arms' heads from sectors in each track. (b) A cylinder is a virtual set of concentric tracks, one track per platter.

slower than in main memory [22].

To complete a data transfer from the hard disk to main memory, as soon as the initial sector is found, the data is retrieved as fast as the disk rotates. For state-of-the-art HDs, this speed is about 70 to 100 megabytes per second, this is also slower than in main memory which achieves transfer rates up to 52.8 GB/s.

To amortize the initial delay of seeking, one can transfer a large contiguous group of data at a time. A block is the amount of data transferred to or from the disk in a single I/O operation. The block size may vary, but since I/Os are in units of blocks, the algorithms that explore locality in item accesses, as opposed to a random distribution, will perform better.

2.1.2 Parallel Disk Model

The goal of external memory algorithms is to minimize the number of I/Os. As seen in Section 2.1.1, the number of I/Os is directly related to the number of transferred blocks. This is one key aspect of the Parallel Disk Model (PDM), the most used model for design and analysis of EM algorithms. The PDM has the following main properties:

- N = problem size (in units of data items);
- M = internal memory size (in units of data items);
- B = block transfer size (in units of data items);
- D = number of independent disk drives;
- P = number of CPUs.

Locality of reference and parallel disk access are important mechanisms for I/O efficient algorithms. Locality is achieved partially through the concept of block transfers. Processors and disks are independent and the model assumes that, if $P \leq D$, a CPU can drive about D/P disks; if $D < P$, each disk is shared by about P/D processors. The disks run in parallel, so D blocks can be transferred in one operation. It is also common to measure the size of the data in units of blocks

and not in units of data items. Data size in blocks is expressed using a lower case notation:

$$n = \frac{N}{B}, \quad m = \frac{M}{B}.$$

The main performance measure for the EM algorithms analysis is the number of I/Os it performs rather than its time complexity, since the transfer of a block from a disk, for example, is millions times slower compared to main memory. This critical point is the basis of EM algorithms design because they aim to reduce those I/Os.

Table 2.1 shows I/O bounds for some of the fundamental external memory operations: scan, sort and search. They perform, respectively, data read, sorting and a search for a specific element in a sorted array. These operations are often used and the complexity of most EM algorithms can be expressed in the number of Scan, Sort and Search operations.

Table 2.1: I/O bounds for some of the fundamental operations [22].

	I/O bound	
Operation	$D = 1$	$D \geq 1$
Scan(N)	$\Theta\left(\frac{N}{B}\right) = \Theta(n)$	$\Theta\left(\frac{N}{DB}\right) = \Theta\left(\frac{n}{D}\right)$
Sort(N)	$\Theta\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right) = \Theta(n \log_m n)$	$\Theta\left(\frac{N}{DB} \log_{M/B} \frac{N}{B}\right) = \Theta\left(\frac{n}{D} \log_m n\right)$
Search(N)	$\Theta(\log_B N)$	$\Theta(\log_{DB} N)$
Output(Z)	$\Theta\left(\max\left\{1, \frac{Z}{B}\right\}\right) = \Theta(\max\{1, z\})$	$\Theta\left(\max\left\{1, \frac{Z}{DB}\right\}\right) = \Theta\left(\max\left\{1, \frac{z}{D}\right\}\right)$

There exist other models, besides PDM, for working not only with EM, but all kinds of algorithms that make use of a memory hierarchy. Those models, however, add or remove some parameters that are important from their perspective. For a complete listing of models and their characteristics, we refer the reader to [22].

2.2 Libraries For External Memory Algorithms

For EM implementations, one must care about how the I/Os will be made. The implementation of a generic library that handles the transfer of data between main and external memory is very time consuming. One public available library is the Standard Template Library for Extra Large Data Sets (STXXL) [20], initially developed by Roman Dementiev [9].

2.2.1 STXXL

As the C++ Standard Library (STL), STXXL implements data structures like vectors, maps, stacks, priority queues, and also algorithms like sort, merge and others. It also has transparent support for parallel disk accesses.

In the algorithm that we propose in Section 3.2, the only used structure are vectors. It is important for the reader unfamiliar with STXXL to know how the

library handles data. This way it is possible to understand the mapping to the PDM that is the basis for the complexity analysis. The next paragraphs will briefly explain the available parameters for vectors and how data is swapped between main and external memory.

STXXL splits data in blocks and allows the user to define their size in bytes. Blocks are loaded in pages, which can consist of one or more blocks. Only pages are loaded from or stored in external memory. Vectors have these two parameters plus the type of the elements, the maximum number of pages that can be held in main memory at once and the type of the pager. The pager is the object responsible for replacing pages in main memory by those requested from external memory.

The pager works with the value of the maximum number of pages in memory. Every page in memory fills a *slot*. When a page is requested and it is not in main memory, the pager loads it from external memory to main memory. If all slots are filled, the pager replaces some of them with the new page. The default pager for vectors of STXXL implements the Least Recently Used (LRU) strategy, which drops the page that had its last access earlier in time.

The minimum block size is 4096 bytes. At least one page must be loaded in main memory. The number of I/Os done by STXXL is the number of blocks read and written. The number of I/Os is always a multiple of the page size.

2.3 Current Algorithms For Counting And Listing Triangles

We will summarize main memory algorithms for counting and listing triangles in Section 2.3.1, and external memory algorithms in Section 2.3.2.

2.3.1 Main Memory Solutions

Most exact algorithms for counting and listing triangles have been designed for main memory. Schank and Wagner [19] studied experimentally almost all available algorithms. Latapy [12] reviewed those algorithms, presented new ones and paid much attention to their memory consumption, analyzing their space complexities. Both papers expose the solutions for triangles counting and listing. The later is also a great guide for triangle problem solving methods.

For counting purposes, the use of matrix multiplication is a simple approach. Although it has poor efficiency with the trivial method, fast matrix multiplication can be used to speed it up. The most efficient triangles counting algorithm is, following Latapy's nomenclature, the *ayz-counting*, proposed by Alon et al. [2]. It requires fast matrix multiplication and has time complexity $O(n^{2.376})$. However, in practice, this algorithm is not used because of its space complexity, since it requires an adjacency matrix with size $\Theta(n^2)$.

One can note that there is a lower bound for time complexity in triangle listing in the general case. There may be $\binom{n}{3} \in \Theta(n^3)$ triangles in G. This is the same to say that there are $\Theta(m^{\frac{3}{2}})$ triangles since G may be a clique of $\Theta(\sqrt{m})$ vertices. With this, Lemma 1 can be stated.

Lemma 1. *([12, 19]). Listing all triangles in G is in* $\Omega(n^3)$ *and* $\Omega(m^{\frac{3}{2}})$ *time.*

However, $\Theta(m^{\frac{3}{2}})$ is much better in sparse graphs. The most common algorithms that explore this property are *vertex-iterator* (Algorithm 1) and *edge-iterator* (Algorithm 2).

Algorithm 1: vertex-iterator

Input: adjacency list E of G and its adjacency matrix A

```
1 foreach v ∈ V do
2     foreach u ∈ N(v) do
3         foreach w ∈ N(v) do
4             if A_uw = 1 then output triangle {u, v, w}
```

Algorithm 2: edge-iterator

Input: sorted adjacency list E of G

```
1 foreach (u, v) ∈ E do
2     foreach w ∈ N(u) ∩ N(v) do
3         output triangle {u, v, w}
```

The *vertex-iterator* has time complexity $\Theta(\sum_{v \in V} d(v)^2)$, $\Theta(m\Delta)$, $\Theta(mn)$ and $\Theta(n^3)$ [12]. The *edge-iterator* has time complexity equals to $\Theta(m\Delta)$, $\Theta(mn)$ and $\Theta(n^3)$ [12]. Note that all complexities are the same in the worst case, i.e., when G is a clique.

These two algorithms are the basis for triangle listing. Many algorithms reported in literature are variations of them. Besides them, there exists *tree-listing*, *new-vertex-listing*, *new-listing* and *ayz-listing*. We will not cover in depth all of them because it is not the scope of this work, but observe that all have the same time complexity but different space complexities. A summary of their complexities is presented in Table 2.2. The most efficient triangles listing algorithms – and with simplified implementations – are *forward* and *compact-forward*. The later is also memory efficient, compared to the former.

Table 2.2: Main memory algorithms for several triangles problems and their time and space complexities. More proofs can be found in [12]. σ is a constant that represents the size of the type that encodes a vertex.

Algorithm	Time Complexity	Space Complexity
vertex-iterator	$\Theta\big(\sum_v d(v)^2\big)$, $\Theta(m\Delta)$, $\Theta(mn)$, $\Theta(n^3)$	$\Theta(n^2)$
edge-iterator	$\Theta(m\Delta)$, $\Theta(mn)$, $\Theta(n^3)$	$\Theta(2m+n) \subseteq \Theta(m)$
tree-listing	$\Theta(m^{\frac{3}{2}})$	$\Theta(n^2)$
ayz-listing	$\Theta(m^{\frac{3}{2}})$ if $K \in \Theta(\sqrt{m})$	$\Theta(n^2)$
forward	$\Theta(m^{\frac{3}{2}})$	$\Theta(3m+2n) \subseteq \Theta(m)$
compact-forward	$\Theta(m^{\frac{3}{2}})$	$\Theta(2m+2n) \subseteq \Theta(m)$
new-listing	$\Theta(m^{\frac{3}{2}})$	$\Theta(2m+n+\frac{n}{\sigma}) \subseteq \Theta(m)$

We present the *compact-forward* algorithm in detail (Algorithm 3) and demonstrate its correctness and complexity.

The *compact-forward* algorithm is derived from *edge-iterator* but it aims to be

Algorithm 3: compact-forward [12]

Input: adjacency list E of G

```
1   number the vertices with an injective function η() such that
    ∀u, v ∈ V : d(u) > d(v) ⟹ η(u) < η(v)
2   sort E according to η()
3   foreach v taken in increasing order of η() do
4       foreach u ∈ N(v) with η(u) > η(v) do
5           let u' be the first neighbor of u, and v' the one of v
6           while there remain untreated neighbors of u and v and
            η(u') < η(v) ∧ η(v') < η(v) do
7               if η(u') < η(v') then set u' to the next neighbor of u
8               else if η(u') > η(v') then set v' to the next neighbor of v
9               else
10                  output triangle {u, v, u'}
11                  set u' to the next neighbor of u
12                  set v' to the next neighbor of v
```

memory efficient, or compact, since it does not require an adjacency matrix but simply an edge list.

To prove that its time complexity achieves the triangle listing lower bound $\Theta(m^{\frac{3}{2}})$, described in Lemma 1, a property of vertices is stated in Lemma 2.

Lemma 2. *Any vertex $v \in V$ has at most $O(\sqrt{m})$ neighbors of higher degree.*

Proof. Assume v has H neighbors $n_1, ..., n_H$ of higher degree. We have $d(n_i) \geq d(v) \geq H$, and therefore they induce at least

$$\sum_{1 \leq i \leq H} d(n_i)/2 \geq Hd(v)/2 \geq H^2/2$$

edges. Since $H^2/2 \leq m$, i.e., $H \leq \sqrt{2m}$ we have $H \in O(\sqrt{m})$. □

Theorem 1. *([12]) Given the adjacency list of G,* compact-forward *lists all triangles in $\Theta(m^{\frac{3}{2}})$.*

Proof. For each vertex x, let $N^+(x) = \{y \in N(x) | d(y) \geq d(x)\}$ be the set of neighbors of x with equal or higher degree to the one of x itself. For any triangle $t = \{u, v, w\}$, one can suppose without loss of generality that $\eta(w) < \eta(v) < \eta(u)$. One may then discover t by discovering that w is in $N^+(u) \cap N^+(v)$.

If the adjacency structures encoding the neighborhoods are sorted according to $\eta()$, then we have that $N^+(v)$ is the beginning of $N(v)$, truncated when we reach a vertex v' with $\eta(v') > \eta(v)$. Likewise, $N^+(u)$ is $N(u)$ truncated at u' such that $\eta(u') > \eta(v)$. Lines 1 and 2 of Algorithm 3 ensure the appropriate sorting for the edge list E, which also represents the neighborhoods for all vertices. These lines have time complexity $O(m \log m)$.

Lines 3 and 4 iterate through all edges on the graph. The condition $\eta(u) > \eta(v)$ in line 4 ensures that edges do not get repeated. Time complexity is $O(m)$ for those.

Lines 6 to 12 compute the intersection $N^+(u) \cap N^+(v)$, explained later. The intersection's matches output triangles (Line 10). The time complexity for this internal loop is $O(\sqrt{m})$ as Lemma 2 ensures.

Thus, the final complexity of the *compact-forward* algorithm is $O(m\sqrt{m}) = O(m^{\frac{3}{2}})$.

□

2.3.2 External Memory Solutions

The main memory solutions for triangle problems are not prepared for dealing with massive graphs. Until now there was only one algorithm I/O-efficient for listing triangles. It was proposed by Roman Dementiev in his doctoral thesis [8] and we reproduce it in Algorithm 4.

Algorithm 4: Dementiev's External memory vertex-iterator [8]

Input: An adjacency list E of G, assuming that $\forall v, u \in V : (v, u) \in E \implies (u, v) \in E$; A value k as the maximum size of queries list

```
1  create empty lists D, F, L, N, Q, R
2  sort E lexicographically
3  compute degree list D = [(v, d(v)), ...] based on E
4  sort D by d(v)
5  for i ∈ {0, 1, ..., n − 1} do
6      let (v, d(v)) be the iTh element in D
7      append (v, i) to R
8  sort R lexicographically
9  let R be [(v, p(v)), ...]
10 scan E and R producing F = [(p(v), v, u), ...]
11 sort F by u
12 scan F and R producing F = [(p(v), p(u), v, u), ...]
13 sort F by p(v), p(u)
14 c = −1
15 foreach (p(v), p(u), v, u) ∈ F do
16     if p(v) < p(u) then
17         if p(v) ≠ c then
18             c = p(v)
19             N = [p(u)]
20         else
21             foreach w ∈ N do
22                 append (w, p(u), v) to Q
23                 if Q is longer than km then
24                     sort Q lexicographically
25                     foreach (w, p(u), v) ∈ Q do
26                         if (w, p(u)) ∈ E then output triangle {w, p(u), v}
27                     clear Q
28             append p(u) to N
```

Dementiev's algorithm is based on *vertex-iterator*. However, *vertex-iterator* requires an adjacency matrix of the graph such that the adjacency test can be made in $O(1)$ (line 4 of Algorithm 1). To avoid the adjacency matrix, Dementiev's uses an edge list E and the adjacency tests, or queries, are buffered in batches of size km

(where k is some constant; e.g. Dementiev uses $k = 4$ [7]). The buffer is important so the queries can be checked with only one scan of E per batch. This is more efficient than full random access, that would be required if no buffer were used. Thus, the final I/O complexity can be reduced.

Besides the buffered approach, Dementiev observes in [8] that the number of edge queries can be reduced heuristically if the nodes are processed in order of increasing degrees. Then, while examining the neighbors of a vertex v, a query is made only for neighbors u and w if $d(v) < d(u) < d(w)$. This heuristic is implemented I/O-efficiently by using an injective mapping function, built with scans and sorts, represented by lines 1 to 13 in Algorithm 4.

Dementiev assumed that the adjacency list of a vertex, represented by N in Algorithm 4, must fit in main memory. This makes Algorithm 4 as a semi-external memory algorithm. However, the list can be implemented with an external memory structure, to obtain an external memory algorithm.

3 EXTERNAL MEMORY TRIANGLES LISTING

In this chapter we present our main contributions: an analysis of Dementiev's *external memory vertex-iterator* in Section 3.1; and a new external memory algorithm for listing triangles in Section 3.2, including a discussion of its I/O complexity and how its performance can be improved.

3.1 External Memory Vertex-Iterator Analysis

We first analyze the Dementiev's *external memory vertex-iterator* (Algorithm 4) without answering queries, i.e., ignoring lines 23 to 27.

Lemma 3. *If a single adjacency list fits in main memory, and without answering queries, the external memory vertex-iterator in lines 15 to 28 has worst case I/O complexity*

$$\Theta\Big(\text{Output}\Big(\sum_{v\in V} d(v)^2\Big)\Big) \subseteq \Theta(\text{Output}(m\Delta)) \subseteq \Theta(\text{Output}(mn)) \subseteq \Theta(\text{Output}(n^3)).$$

Proof. The upper limits follow from the observation that the loop in lines 21 to 27 is limited by $d(v)$ and is executed at most $d(v)$ times for a given vertex v and the fact that $d(v) \leq \Delta \leq n$ and $m \leq n^2$. This way, there are at most $\sum_{v\in V} d(v)^2$ queries that are output to buffer Q. The case of a clique of size n shows all the upper limits are tight, since each iteration of the loop outputs a triangle. □

Theorem 2. *The worst case I/O complexity of Algorithm 4 is*

$$O\left(\frac{\sum_{v\in V} d(v)^2}{m}\text{Sort}(m)\right).$$

Proof. Dementiev's implementation assumes that the adjacency list fits into main memory. If it does not, we output and scan at most another $d(v)^2$ items per vertex v, for an additional total cost of $O(\text{Scan}(\sum_{v\in V} d(v)^2))$, which is already dominated by the cost without answering queries.

As observed in Lemma 3, there are at most $\sum_{v\in V} d(v)^2$ queries to answer. These queries are processed in batches of size km (line 23). Each batch requires a sorting of the list of queries ($O(\text{Sort}(km))$ followed by a parallel scan of the edge list ($O(\text{Scan}(m))$) and the list of queries ($O(\text{Scan}(km))$). Thus, answering those queries

in batches of size km costs

$$\Theta\left(\frac{\sum_{v\in V} d(v)^2}{km}\big(\mathrm{Sort}(km)+\mathrm{Scan}(km)\big)\right) \subseteq$$
$$\Theta\left(\frac{\sum_{v\in V} d(v)^2}{km}\mathrm{Sort}(km)+\mathrm{Scan}\Big(\sum_{v\in V} d(v)^2\Big)\right) \subseteq$$
$$O\left(\frac{\sum_{v\in V} d(v)^2}{m}\mathrm{Sort}(m)\right) \quad (3.1)$$

The pre-processing in lines 1 to 13 costs $O(\mathrm{Sort}(m))$. The output cost observed by Lemma 3 is overlapped by the Scan cost in Equation 3.1. The cost for answering queries yields the total cost claimed above. □

As noted previously in Section 2.3.2, Dementiev proposed an heuristic to reduce the number of edge queries: his algorithm can be improved querying only vertices of higher degree. This improvement is made by replacing line 22 in Algorithm 4 by

if $p(v) < w \wedge p(v) < p(u)$ **then** append $(w, p(u), v)$ to Q

With an improved implementation, one can skip the low-index w's. In this case, the output complexity reduces to

$$O\Big(\sum_{v\in V} d^+(v)^2\Big) \leq O\Big(\sqrt{m}\sum_{v\in V} d^+(v)\Big) \leq O(m\sqrt{m}) = O(m^{\frac{3}{2}})$$

where $d^+(v)$ represents the number of neighbors of v with higher or equal degree than itself. With this observation, we can state the following corollary.

Corollary 1. *Based on Theorem 2, the worst case I/O complexity of Algorithm 4 with the above alteration is*

$$O\Big(m^{\frac{3}{2}}\mathrm{Sort}(m)\Big).$$

3.2 External Memory Compact-Forward

Our approach adapts the *compact-forward* algorithm for listing triangles in external memory. We choose *compact-forward* because it does not depend on adjacency matrices, as *vertex-iterator* does; or adjacency arrays for each vertex, as *forward* is based. Although it requires an offset array for locating the neighbors of a vertex, since we use an edge list for the graph representation. The proposal is described in Algorithm 5.

In lines 3 to 5, a new list F of edges with duplicate edges is built. Each element has a special structure, a pair of pairs, because it will later be altered with new indices which will guide the proper sorting of the list.

Line 6 is a notation to help our explanation, in which we name each edge (or element of F) by a source vertex v, that has a mapped index $m(v)$, to a target vertex u, that has a mapped index $m(u)$. The mapped indices are defined later, in lines 10 to 13, and set in lines 15 and 16.

Algorithm 5: External Memory Compact-Forward

Input: list of edges E of G

create empty lists O, D and F
$B =$ a buffer
foreach $(v, u) \in E$ **do**
    append $((v, v), (u, u))$ to F
    append $((u, u), (v, v))$ to F
let F be $[((v, m(v)), (u, m(u))), ...]$
sort F by v and u lexicographically
compute degree list $D = [(v, d(v)), ...]$ based on F
sort D by $d(v)$ decreasingly
for $i \in \{0, 1, ..., n-1\}$ **do**
    let $(v, d(v))$ be the iTh element in D
    set $O[i]$ with $O[i-1] + d(v)$
    set $(v, d(v))$ in D with (v, i)
sort D by v lexicographically
set all $m(v)$ in F with i where $(v, i) \in D$
set all $m(u)$ in F with i where $(u, i) \in D$
sort F by $m(v)$ and $m(u)$ lexicographically
foreach $((v, m(v)), (u, m(u))) \in F$ **do**
    if $m(v) < m(u)$ **then**
        insert query $((v, m(v)), (u, m(u)))$ in B
        if $|B| \geq k$ **then**
            sort B appropriately
            foreach $((v, m(v)), (u, m(u))) \in B$ **do**
                access neighbors of v with $F[O[m(v)]]$
                access neighbors of u with $F[O[m(u)]]$
                let v' be the first neighbor of v, and u' the one of u
                let $m(v')$ be the mapped index of v', and $m(u')$ the one of u'
                while *there remain untreated neighbors of* u *and* v
                $\wedge\ m(u') < m(v) \wedge m(v') < m(v)$ **do**
                    if $m(v') < m(u')$ **then** set v' to the next neighbor of v
                    else if $m(v') > m(u')$ **then** set u' to the next neighbor of u
                    else
                        output triangle (u, v, u')
                        set v' to the next neighbor of v
                        set u' to the next neighbor of u

Line 8 computes the degree of each node, iterating through F, and stores it in a pair $(v, d(v))$. This information is used to sort the vertices by non-increasing degrees. This processing order is guaranteed with lines 9 to 17.

Lines 18 to 34 count the triangles. They are very similar to Latapy's *compact-forward* (Algorithm 3).

The lines 18 to 20 iterate over all not repeated edges on the graph and insert them in a buffer B. When the buffer is full (line 21), i.e., when its size reaches a value k, the queries holded by it get processed.

The processing order of the buffer's queries is defined by the sorting applied in line 22. The appropriate sorting method will be discussed in Section 3.2.2. Let us imagine that there is no sorting for now.

The neighborhood of v and u is accessed with lines 24 and 25, respectively. The random access of F is helped by an offset list O that keep track of the position of the first neighbors for all vertices, thus $|O| = n$.

Finally, the intersection of the neighbors of u and v, as explained for Algorithm 3, is made with lines 28 to 34 in Algorithm 5.

The IO complexity of Algorithm 5 is established with Theorem 3.

Theorem 3. *Algorithm 5 lists all triangles in* $O(Scan(m^{\frac{3}{2}}))$

Proof. The I/O complexity of the lines 1 to 17 is $O(\text{Sort}(m))$. Line 18 starts an iteration over all edges in F ($O(\text{Scan}(m))$). By Theorem 1, the loop of lines 28 to 34 has time complexity $O(\sqrt{m})$, establishing an I/O complexity of $O(\text{Scan}(\sqrt{m}))$ since each loop iteration executes only a constant number of I/O operations. Random access of the offset list O has I/O complexity $O(1)$. Thus, it does not alter the algorithm's complexity. In the worst case, it will perform one operation for each edge, which is already taken into account for the loop in line 18. With this, lines 18 to 34 and EMCF algorithm I/O complexity is

$$O(\text{Scan}(m)\text{Scan}(\sqrt{m})) \subseteq O(\text{Scan}(m^{\frac{3}{2}}))$$

□

3.2.1 I/O Costs

In this section we discuss the source of I/Os in Algorithm 5 based on observations of our implementation. These observations were important to propose a performance optimization that will be explained in this and Section 3.2.2. In this section we will introduce the lines of the algorithm that may produce I/Os. This will require some assumptions so we can isolate those lines and will lead to the conclusion that a specific processing order of the edges would reduce those I/Os.

With an implementation of the algorithm one can find out that the number of input and outputs depends on the block size of the main data structure and how blocks are managed in main memory.

The edge list F in Algorithm 5 is implemented as an STXXL vector. Throughout this and next sections, we assume the block size for F to 64KiB; page size to 4, leading to a 256KiB page; the number of pages depends of the available main memory space; and the pager is the default one, LRU.

Since the edge list F requires four integers per page and each page has 256KiB, with 32 bits integers, 16,384 edges can be stored within it. If the maximum degree

of a graph is lower than this, all edges adjacent to a vertex can be stored in a single page. Therefore at most two pages will be loaded if an iteration through its edges is required. Let us suppose that pages are not shared by the edges of two vertices. Under this assumption, lines 28 to 34 does not make any input or output if we consider that we do not output triangles but only count them. Therefore, the lines that can do I/Os are 18, in u, mu, v and mv assignment; 24 or 25. The offset structure O is an STXXL vector, but let us assume from now on that it does not do any I/O, i.e, it fits in main memory.

Supposing that the buffer B from Algorithm 5 has no effect (for example, if its size is one), line 24 does not make any I/O because the page that holds the neighbors of v was already loaded in main memory by line 18. However, the access to neighbors of u (line 25) almost *always* loads a page. This happens because of the forward behavior of the algorithm. For example, imagine a program that has some limited main memory space available; in this space, the program loads a maximum of 1,000 pages at once; supposing that for some graph, a vertex v_i has 2,000 neighbors and all of them are in different pages; even if v_{i+1}, the next analyzed vertex, had the same neighbors of v_i, all pages would be loaded again. In the example, this happens because the pages $1, ..., 1000$, loaded by v_i, would be replaced at all by $1001, ...2000$, also loaded by v_i, and then loaded again through v_{i+1} by the same motive.

If buffer size is more than one and supposing that main memory is clean, i.e. no pages were previously loaded; when answering buffer's queries, the access of neighbors of v and u (lines 24 and 25) are the only places that an I/O can happen.

These simplifications lead to a more precise definition of the I/O problem: each edge (u, v) in B has a cost of $c_{uv} \in \{0, 1, 2\}$ which represents the number of pages that is needed to be loaded. The real value of c_{uv} depends on the state of the main memory, i.e., which pages are already loaded. The cost for the edges in a buffer is $C = \sum_{(u,v)\in B} c_{uv}$. A graph would be split in r buffers, since the number of generated queries appended to the buffer could be greater than its size. Thus, for the whole graph, the cost is

$$C_G = \sum_{i\in\{1,..,k\}} C_i.$$

If the buffer is not sorted, the use of a buffer has no effect either, because the same problem of page replacements will happen. Therefore, to minimize IO complexity, we have to find a permutation of edges which minimizes C_G. This optimization is done through an appropriately sorting of B.

3.2.2 Sorting the buffer

In this section we will exemplify how I/Os can be reduced when a sorting is applied to the buffer from Algorithm 5. A bipartite graph will be used as example because it will help the understanding about how data replacement is made between main and external memory. We will use the concept of *page*, seen in Section 2.2.1, to explain this replacements. The observations will be the basis for a proposal of sorting algorithm.

Keep the assumption from Section 3.2.1 that all adjacent edges of a vertex fits in a page and pages are not shared by the edges of two different vertices.

Imagine that one uses, as input of the Algorithm 5, a bipartite graph (Figure 3.1). Obviously, the number of triangles in it is zero. However, vertices may represent

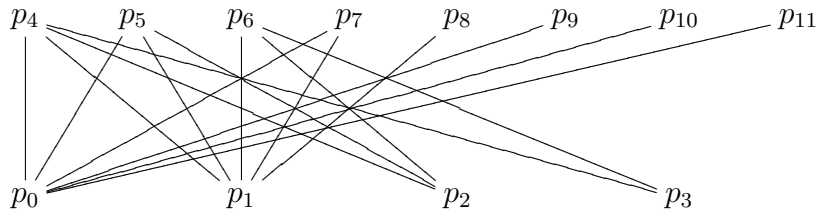

Figure 3.1: A bipartite graph

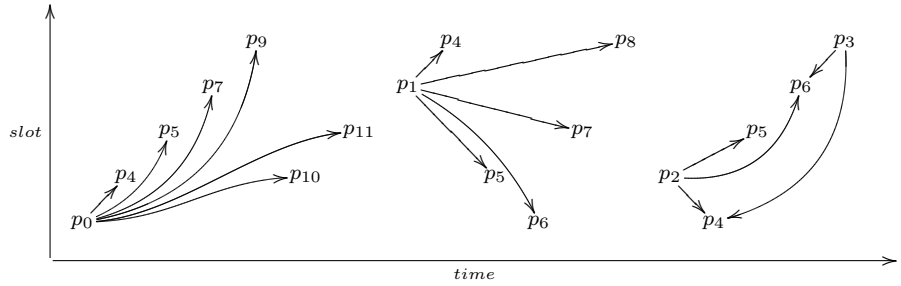

Figure 3.2: Page loading scheme. Each page loaded more than once is represented the same amount of times. There are only 5 slots in main memory for loading pages. Arcs represent the processed edges: naturally, edges have two vertices; since the vertices represents pages, two pages can be loaded for an edge. A requested source vertex or page points to a target page. 18 page loads are made in total. Observe that edges pointing to a page that is back in time of its source indicate reused pages.

pages (because its adjacent edges are not in different pages) and, so, edges represents the relations between pages. This is why the bipartite graph is interesting to illustrate the sorting. Since $\forall u \in V_1 : u \notin V_2$, there is this direct relation between the graph and the external memory structure. When one wants to access the neighborhood of a vertex, it means that only one page will be loaded.

Now, suppose that one can load only a maximum of 5 pages at once, i.e., the main memory has 5 slots for pages. Following Algorithm 5, some of the pages will be loaded more than once because a page can be source and target of multiple vertices. What in fact happens is shown in Figure 3.2. Each reloaded page is represented as a duplicated vertex.

The edges of the example graph are processed at the usual way. Some pages are loaded a lot and 18 page loads are made, as shown in Figure 3.2. This can be improved if the edges are processed in a different order. For example, Figure 3.3 shows the page load scheme for one possible permutation of the edges. It requires only 14 page loads.

The improvement is made using a sorting heuristic, defined by Algorithm 6. It groups target pages that can fit in memory and permits the iteration of all edges related to them but also grouping source pages secondarily. Everything related to the sorting is done in main memory so no I/O is necessary. In line 11, $q - 2$ is the divisor because, if there are q slots in main memory, the iteration through the sorted buffer edges will load the page for a source vertex and $q-2$ pages for target vertices. When a new source comes up, a free slot will be replaced by its page. If it was used $q - 1$ in line 11, for example, potentially a new page would replace a target page and this is not the desired behavior.

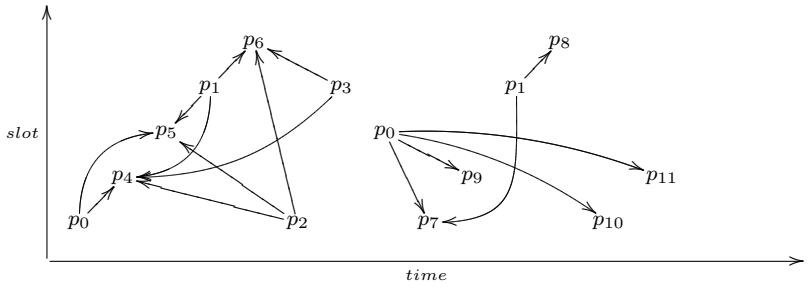

Figure 3.3: Page loading scheme after sorting the buffer. 14 page loads are made.

Algorithm 6: Sort Buffer

Input: The list of edges of buffer B; The offsets O from Algorithm 5

```
1  P = [ ]
2  A = [ ]
3  foreach ((u, mu), (v, mv)) ∈ B do
4      p = the page of v identified through mv and O
5      insert p in P
6      insert (p, (u, mu), (v, mv)) in A
7  sort P lexicographically and remove duplicates
8  let i_p be the index of page p ∈ P
9  let q be the number of page slots in main memory
10 foreach (p, (u, mu), (v, mv)) ∈ A do
11     update p with ⌊i_p/(q − 2)⌋
12 sort A lexicographically
13 map A to B, dropping the first element of A
```

Since Algorithm 6 is an heuristic, there is no guarantee that the result will be improved. In fact, it can be worse than running the algorithm with no special sorting. For example, the proposed algorithm is based on the assumption that the buffer size is a lot smaller than the total number of edges. With this, the number of queries will be small enough so only a few number of different source vertices will be inside it and the number of neighbors is great enough so they cover many pages of the graph. Thus, there are potentially more reuse of the target pages than the source ones. This is very common for most graphs. Although, if there is a graph that all target vertices do not share source vertices, the edges could be split in such a way that source pages would be loaded several times, increasing the total number of I/Os.

One can think in another heuristic that groups pages most frequently used. But, experimental tests show results a bit worst than the one from Algorithm 6.

The *optimal sorting* should be the one that minimizes the cost C_G, described previously. Unfortunately it is hard to define the correct order because of the dynamic cost of the edges.

4 EXPERIMENTAL RESULTS

In this chapter we will expose some experimental results based on the implementation of our algorithm, *external memory compact-forward* (EMCF). We compared the results with the ones obtained from Dementiev's *external memory vertex-iterator* (EMVI) implementation [7] and Latapy's *compact-forward* (CF) [11]. Although the later is a main memory algorithm, it is our baseline for time tests.

4.1 Dataset

The dataset used in the experiments is a subset of Latapy's [12, 14]. The graphs represent real situations:

- **ip** [13] — nodes are IP addresses with a link between two addresses if the router on which the measurement is conducted routed a packet from one of them to the other.
- **notre-dame** [13] — a graph in which nodes are the web pages at the university of Notre-Dame, with links between them.
- **actor-2002** — the nodes are movie actors found in IMDB, with a link between two actors if they appear in a same movie.
- **p2p** [13] — nodes are the users of an eDonkey server, and two nodes are linked if they exchanged a file during the measurement
- **uk-2005** [13] — graph provided by the WebGraph project [5, 4]. Nodes are pages in the .uk domain and edges are links between them.

In Table 4.1 is shown some graph properties for each instance. The number of triangles were counted by our algorithm and compared to the result of Dementiev's.

4.2 Results

The results that will be presented in this section are structured as follows. We will show results for a set of tests for EMCF with different memory configurations. This will expose the performance impact on the use of a buffer. Next, we will show the results for CF and EMVI so we can compare them with the best results of EMCF.

In this first set of tests we experimented EMCF with different buffer and block size configurations. EMCF_1 represents the tests where buffer size is one and block

Table 4.1: The instances of graphs used at the experiments and some of their properties.

Instance name	Vertices	Edges	Max Degree	Triangles
ip	467,273	1,744,214	81,756	22,197,777
notre-dame	701,654	1,935,518	5,331	5,334,389
actor-2002	382,220	15,038,083	3,956	346,813,199
p2p	6,235,399	159,870,973	15,420	4,350,587,128
uk-2005	39,459,925	822,487,051	3,984,658	21,860,421,554

Table 4.2: Time results for EMCF with different buffer and block size configurations. Preproc. is a shortening for preprocessing and represents the time of processing lines 1 to 17 in Algorithm 5.

	$EMCF_1$		$EMCF_2$		$EMCF_3$	
Instance	Preproc.	Total	Preproc.	Total	Preproc.	Total
ip	6 s	10.1 s	6 s	10.5 s	6 s	10.5 s
notre-dame	7 s	7.5 s	7.1 s	8.8 s	6.9 s	8.5 s
actor-2002	1 min 7 s	2 min 1 s	1 min 8 s	2 min 7 s	1 min 7 s	2 min 4 s
p2p	26 min	8 days 19.6 hrs	17 min	1 hr 5 min	16 min	44 min
uk-2005	—	—	7 hrs 8 min	17 hrs 25 min	7 hrs 6 min	9 hrs 25 min

size is 64KiB. In $EMCF_2$, the buffer size corresponds to 10% of main memory's available space and block size is also 64KiB; In $EMCF_3$, the buffer size corresponds to 30% of main memory and block size is 2MiB. The time results are shown in Table 4.2 and I/O results are in Table 4.3.

Tests of $EMCF_1$ run in a Core 2 Q8200@2.33GHz with 4GB DDR2@800MHz Dual Channel RAM and a SATA 7200rpm HD. Tests $EMCF_2$ and $EMCF_3$ run in a Core i7 930@2.80GHz with 12GB DDR3@1333MHz Tripple Channel RAM and a SATA 7200rpm HD. All three tests in this set had the main memory limited to 1.5GiB.

The exposed results are useful to analyze how a different processing order of edges, produced through the sorting of the buffer, affects the performance of the algorithm. $EMCF_1$ took almost 9 days to get complete and transferred 20 TiB of data. On the other hand, the increase of buffer size leads to an understanding of the edges and optimization of the processing order. As the experiments show, $EMCF_2$ and $EMCF_3$ reduce sensitively the amount of data transferred and, thus, Algorithm 6 has been proved to be effective, at least with these instances.

Table 4.3: Transfer amount results for EMCF with different buffer and block size configurations. Preproc. is a shortening for preprocessing and represents the transfer during the processing of lines 1 to 17 in Algorithm 5.

	$EMCF_1$		$EMCF_2$		$EMCF_3$	
Instance	Preproc.	Total	Preproc.	Total	Preproc.	Total
ip	0.5 GiB	0.5 GiB	0.5 GiB	0.5 GiB	0.5 GiB	0.5 GiB
notre-dame	0.5 GiB	0.5 GiB	0.5 GiB	0.5 GiB	0.5 GiB	0.5 GiB
actor-2002	5.1 GiB	5.1 GiB	5.1 GiB	5.1 GiB	5.1 GiB	5.1 GiB
p2p	82.7 GiB	21,317.0 GiB	77.9 GiB	139.3 GiB	78.2 GiB	116.2 GiB
uk-2005	—	—	1,135.5 GiB	2,386.4 GiB	1,135.8 GiB	1,939.9 GiB

The next set of tests was to experiment Dementiev's EMVI and compare the results with EMCF's, shown in Tables 4.2 and 4.3. EMVI run on the same machine as $EMCF_3$ and also got its main memory limited to 1.5GiB. The comparison is represented through Table 4.4.

Table 4.4: Results for the comparison of Latapy's main memory triangles listing *compact-forward* (CF), our *external memory compact-forward* in its best configuration for huge graphs ($EMCF_3$) and Dementiev's *external memory vertex-iterator* (EMVI).

	Time			I/O	
Instance	CF	$EMCF_3$	EMVI	$EMCF_3$	EMVI
ip	< 1 s	10.5 s	27.6 s	0.5 GiB	1.9 GiB
notre-dame	< 1 s	8.5 s	7.9 s	0.5 GiB	0.5 GiB
actor-2002	14 s	2 min 4 s	4 min 36 s	5.1 GiB	20.5 GiB
p2p	6 min 16 s	44 min	2 hrs 20 min	116.2 GiB	467.7 GiB
uk-2005	5 min 33 s	9 hrs 25 min	6 hrs 9 min	1,939.9 GiB	1,296.8 GiB

The results of EMCF were, except for the graph *uk-2005*, best than EMVI. However, Dementiev's EMVI had less I/Os than EMCF in *uk-2005*. Although EMVI has a worst case I/O complexity greater than EMCF, we were not capable to keep the performance with this relation. The best explanation for why this happens comes from the analysis of EMCF. With no buffer, the algorithm almost always gets in worst case. This happens because of the random access in edge list while a neighborhood is accessed. The buffer was then introduced to rearrange the edges and reduce the I/Os produced. Despite the fact that the buffer is effective, we still do not have the knowledge about a better sorting or even the optimum sorting method. Therefore, for now, the average complexity for EMVI is better than EMCF for those instances.

5 CONCLUSIONS

In this thesis, we analyzed some of the existent algorithms for triangles listing in main memory and also the only one in external memory. Our main contributions were the following ones:

- A better analysis of *external memory vertex-iterator* (EMVI) algorithm for triangles listing proposed by Roman Dementiev [8]. We concluded that its worst case I/O complexity is

$$O\left(\frac{\sum_{v \in V} d(v)^2}{m} \mathrm{Sort}(m)\right).$$

  We also explained an optimization for evaluating only queries with higher degree neighbors and this leads to the worst case I/O complexity $O(m^{\frac{3}{2}}\mathrm{Sort}(m))$.

- We proposed a new algorithm for listing triangles called *external memory compact-forward* (EMCF) that has worst case I/O complexity $O(\mathrm{Scan}(m^{\frac{3}{2}}))$. Some performance optimizations were also proposed and experimental tests were run so we could compare it to Dementiev's in practice.

Although the worst case I/O complexity of our algorithm is better than EMVI, with an implementation we made experimental tests which were important to observe that our algorithm still require optimizations to get better performance in practice that the Dementiev's algorithm. The performance could be improved with a different arrangement of the graph's edges as noted in Section 3.2.1, but we still do not have knowledge of how to produce the optimal arrangement yet.

The average performance of EMCF was understood to be related to the block size of the external memory data structures and how the edges of the graph are processed. This was proved to affect performance through the introduction of a buffer that could rearrange the edges before they were processed. The buffer was sorted with a proposed algorithm and tested in different configurations. However, we still have some open questions about if there exists an optimal sorting and if it can be constructed without affecting to much the time complexity.

Although the proposed algorithm does not have better performance in practice than its predecessor, there is still margin for optimizations. The research of a better sorting of the buffer is necessary for further improvements on *external memory compact-forward*. We also observe that the research for solutions of the triangles problems with external memory are not stagnate and new proposals may appear – maybe using I/O efficient matrix multiplications [17].

REFERENCES

[1] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58:137–147, 1999.

[2] Noga Alon, Raphael Yuster, and Uri Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, March 1997.

[3] Lars Arge, Gerth Stølting Brodal, and Laura Toma. On external-memory MST, SSSP and multi-way planar graph separation. In *7th Scandinavian Workshop on Algorithm Theory*, volume 1851 of *Lecture Notes in Computer Science*, pages 433–447. Springer, 2000.

[4] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.

[5] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[6] Luciana Salete Buriol, Gereon Frahling, Stefano Leonardi, Christian Sohler, and Alberto Marchetti-Spaccamela. Counting triangles in data streams. In *25th ACM Symposium on Principles of Database Systems (PODS2006)*, pages 253–262, 2006.

[7] Roman Dementiev. Pipelined external memory triangle counting/listing algorithm. `http://algo2.iti.kit.edu/dementiev/tria/algorithm.shtml`, 2005. [Online; accessed 3-December-2009].

[8] Roman Dementiev. *Algorithm Engineering for Large Data Sets.* PhD thesis, Saarland University, 2006.

[9] Roman Dementiev, Lutz Kettner, and Peter Sanders. STXXL: Standard template library for XXL data sets. In *13th Annual European Symposium on Algorithms (ESA2005)*, pages 640–651, 2005.

[10] Roman Dementiev, Peter Sanders, Dominik Schultes, and Jop Sibeyn. Engineering an external memory minimum spanning tree algorithm. In *3rd IFIP International Conference on Theoretical Computer Science*, pages 195–208, 2004.

[11] Matthieu Latapy. Triangle computations. `http://www-rp.lip6.fr/~latapy/Triangles`. Source code; Accessed 16-March-2010.

[12] Matthieu Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical Computer Science*, 407(1-3):458–473, 2008.

[13] Clemence Magnien and Matthieu Latapy. Massive graphs dataset. `http://data.complexnetworks.fr/Diameter`. Used as dataset for [14].

[14] Clémence Magnien, Matthieu Latapy, and Michel Habib. Fast computation of empirically tight bounds for the diameter of massive graphs. *Journal of Experimental Algorithmics*, 13:1.10–1.9, 2009.

[15] Ulrich Meyer. On dynamic breadth-first search in external-memory. In *Proc. 25th Annual Symposium on Theoretical Aspects (STACS)*, pages 551–560, 2008.

[16] Ulrich Meyer, Peter Sanders, and Jop F. Sibeyn, editors. *Algorithms for Memory Hierarchies – Advanced Lectures*, volume 2625 of *Lecture Notes in Computer Science*. Springer, 2003.

[17] Sraban Kumar Mohanty. *I/O Efficient Algorithms for Matrix Computations*. PhD thesis, Indian Institute of Technology Guwahati, 2009.

[18] Harald Prokop. *Cache-oblivious algorithms*. PhD thesis, MIT, 1999.

[19] Thomas Schank and Dorothea Wagner. *Finding, Counting and Listing All Triangles in Large Graphs, an Experimental Study*, volume 3503/2005 of *Lecture Notes in Computer Science*, pages 606–609. Springer Berlin / Heidelberg, 2005.

[20] STXXL – Standard Template Library for Extra Large Data Sets. `http://stxxl.sourceforge.net`.

[21] Charalampos E. Tsourakakis. Fast counting of triangles in large real networks without counting: Algorithms and laws. In *ICDM '08: Proceedings of the 2008 Eighth IEEE International Conference on Data Mining*, pages 608–617, Washington, DC, USA, 2008. IEEE Computer Society.

[22] Jeffrey Scott Vitter. *Algorithms and Data Structures for External Memory*, volume 2 of *Foundations and Trends® in Theoretical Computer Science*. Now Publishers, 2006.

[23] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2–3):110–147, 1994.