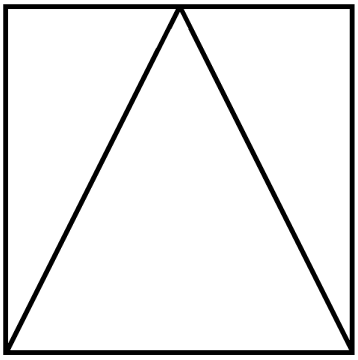
UNIVERSITÉ LIBRE DE BRUXELLES

Faculty of Sciences

Departement of Computer Science



MEMO-F524 - Master thesis

Counting rapidly triangles and 4-cycle subgraphs

Author :

Oruç Kaplan

000540662

Promoter :

Prof. Gwenaël JORET

May 27, 2024

M-INFOS

Preface

This thesis was written as part of my studies in Computer Science at the Université Libre de Bruxelles. It marks the culmination of my academic journey as well as the knowledge I gained during this program.

I chose this topic because it is part of one of the two compulsory specialisation modules I took during my studies. The first being Computational Intelligence and the second being Algorithms focusing on graph theory, data structures and computational geometry. As I was more interested in graphs and algorithms, the choice was straightforward.

In terms of writing, I decided to write this thesis in English, even though it is not my native language, so that it would be open to a wider audience. As a result, I had to use tools such as Deepl and Reverso to correct certain errors and improve the formulation.

Abstract

We live in an age where social networks are predominant in the interaction between individuals. This has resulted in the creation of groups with diverse and varied structures. Today, it has become a major asset to understand these networks in order to take advantage of them for a whole host of applications such as server optimisation, individual influences, advertising, etc. But it's a complex task requiring a great deal of computing power and time-consuming algorithms. This is why, in this thesis, we studied triangle counting and listing algorithms (C_3/K_3) as well as 4-cycles (C_4) which can be useful in understanding clusters, one of the indicators of which is the clustering coefficient. We therefore started with a few theoretical points about graphs and explored a few matrix multiplication techniques in order to find one that is suitable for concrete implementation and experimentation, in this case the use of Basic Linear Algebra Subprograms (BLAS) [33]. Then, after testing and implementing the algorithms, some of which use matrix multiplication, we realised that some of the algorithms dedicated to counting C_3 and C_4 were much more efficient, due to the fact that the data structures used and the input graphs could greatly influence performance. One of the main factors is graph's clustering coefficient. Finally, we tried to parallelise some of the simpler algorithms to see if it was possible to make considerable gains.

The implementations were carried out in C++ using the Standard Library [34] and the OpenBLAS Library [35]. The code for the experiments is on Github, and more information is available in the dedicated section *Repository* (see 8.2).

Acknowledgements

I would like to express my gratitude to my promoter, Gwenaël Joret, for providing me with the opportunity to work on this topic and for his invaluable advice and assistance throughout the completion of this thesis.

I am also thankful to my family for their unconditional support during the writing of this thesis and throughout my university career.

Finally, I would also like to thank my classmates Maxime, Thomas, Vincent and Ali for our mutual support and companionship throughout the completion of this thesis and my studies.

Contents

1 Introduction

In today's society, information technology has become so omnipresent that it enables human activities to be carried out on a much larger scale. In the past, you had to wait several days or even weeks to send a letter. The telephone made it possible to communicate more quickly, but this still came at a considerable cost, which severely limited people's ability to communicate and thus create big groups and communities of people. Indeed, with the development of the Internet, and subsequently of social networks, people were able to carry out tasks such as sharing information, working and communicating, much more widely, more easily and at a reduced cost. This has had the impact of creating larger communities and groups.

Numerous social networks have emerged since the early 2000s, including Facebook, Twitter, Whatsapp, Instagram, WeChat and more. Some are quite simple, allowing only the transfer of simple messages, while others enable much more massive data sharing. But in any case, they all participate in the creation of groups, and their users are becoming more and more numerous as the years are passing [7], as shown by the graph below showing the number of active users on the respective social networks between 2004 and 2018.

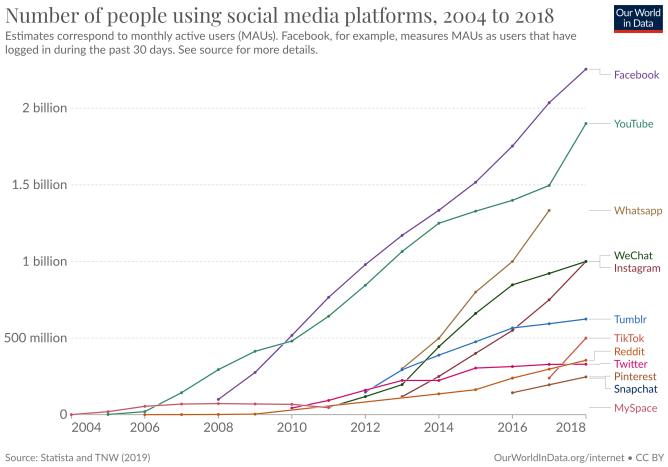


Figure 1: Evolution of the number of monthly active users over the years on every social media platform

The growth of networks has become such that it has become interesting to study their structures. But first it was necessary to model a way of representing the networks and the most common approach is to model the individuals by points and their interaction by lines. In other words, using a mathematical object widely used in computer science called *Graphs*.

In order to carry out these studies, many researchers or even companies call on algorithms that manipulate social network graphs to highlight certain characteristics in order to meet their various needs, whether for optimization, analysis or functionality. That's why, in this thesis, our objective is to study algorithms for counting the number of triangles, 4-cycles and more, present in a graph. Indeed, these various counts are useful for understanding the structure of social networks, and in particular the clustering coefficient, which is calculated on the basis of the number of triangles, which we'll discuss in the dedicated section (see 2.5) and which is an indicator of clustering and thus inter-connectivity between people.

So this thesis is divided into four parts. Firstly, the *Graph* section, which covers all the purely theoretical aspects that will help you understand what follows. We will talk about *Graphs In Social Networks*, *Graph Notations* and some more technical aspects like *Clustering Coefficient*.

Then the *Matrix Multiplication Techniques* section, which is a key mathematical operation in many graph-manipulating algorithms. There are several more or less complex ways of solving it.

Followed by the most important part, our *Algorithms and Implementations*, which contains the various counting and listing algorithms for triangles and 4-cycles that we studied and experimented in practice. But also our technical choices and some generalization for larger cycles.

Finally, before going to the *Conclusion*, we'll take a small look at whether it is possible to have *Improvement With Parallel Programming*. We'll see if we can enhance things with simple parallelisation.

2 Graph

In this first section, we will explain all the aspects related to the graphs that will be used in the algorithms that will be developed in this thesis. First, we'll see *Graphs In Social Networks*, from there see how the *Graph Notations* works. We'll also define *Graph Homomorphism*. Then we'll develop the concept of *Graph Density*, which can be expressed in terms of *Arboricity* or *Average Degeneracy*, and which will be useful for defining the complexity of certain algorithms. Then we'll talk about the *Clustering Coefficient*, a measure that defines the interconnection of the vertices of a graph, and therefore of the people in a network.

2.1 Graphs In Social Networks

In discrete mathematics and computer science, there are a wide range of data structures that can be stored and manipulated to perform calculations. There are, of course, primitive types such as integers, real numbers (in the form of floats or doubles) and characters. These can be grouped in 1, 2, 3 or even more dimensions to form a string, a list, an array or a matrix, and so form what is known as an ordered data structure. On the other hand, we have what are known as unordered data structures, made up of sets, maps and hash tables.

Among all these, there are graphs, which are a rather special structure because, although many people are tempted to put them in the unordered group, in reality they offer so many possibilities that they can form their own category. This is based on the simple concept of nodes, often called vertices, which are interconnected using links called edges. This makes it possible to build a structure in the form of a network that can represent a whole range of things.

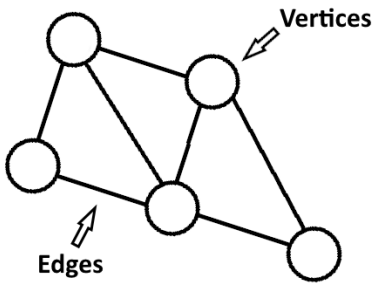


Figure 2: Example of a simple graph

These things include, for example, structure of people (Family trees, Social networks, Groups), road systems, computer networks, cell interactions. But then, we're only interested in social networks, many studies like [8] have already shown that there are different structures of groups among social networks, whose geometries can vary. Among these, we can cite four of them:

Assortative Where the different highly connected groups are also connected between them through only some individual. Theses kind of group formations is frequent on classic social networks such as Facebook or Twitter.

Disassortative Where the individuals of different groups are highly connected between them. This can exist in networks where the aim is to connect as many people as possible, whatever their group.

Ordered Where the different groups form some hierarchy. And thus the communication between two far away groups is only done by an intermediate group. This exist in social network like Linkedin, since the relations between the individual follows the hierarchy within the companies.

Core-periphery Where groups form a dense cluster at the center and sparse at the periphery. This is particularly true in scientific communities, where, for example, the best-known researchers talk to each other a lot, but the new and lesser-known ones are a bit on the sidelines.

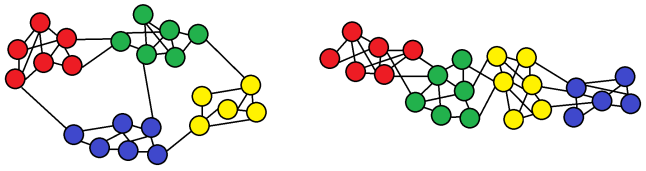


Figure 3: Example of assortative groups on the left and ordered on the right

The interest in studying the formation of these structures within social networks can be numerous. Firstly, from a purely functional point of view, companies can use these descriptions to optimize the way data is exchanged and stored for the smooth running of their social networks. It's easy to imagine dedicating specific servers to each community, in order to limit the amount of data in transit between servers and thus conserve valuable bandwidth. Secondly, it could also enable us to detect influencers who have a positive or negative impact, and thus moderate the

social network to ensure there are no abuses. Another widely-used application is the advertising dimension, where you can easily customize and assign ads to each group to make them more relevant, and so on.

That's why, we we'll talk later about clusters (see 2.5) and how we can define them in a graph. But first, we need to look at some graph notations in order to standardize the concept.

2.2 Graph Notations

In this thesis, we'll be talking mainly about graphs, explaining the various concepts involved. But also the different algorithms that will have to manipulate them. It goes without saying that we'll need to adopt a few necessary notations.

Consider a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges. Sometimes, to designate the set of a particular graph, we'll also use the notation $V(G)$ and $E(G)$. The cardinality of V, denoted $n = |V(G)|$, is the number of vertices in the graph G, while $m = |E(G)|$ represents the number of edges in G. Next, we can define the neighborhood of a vertex $v \in V(G)$ as the set of vertices that share an edge with v (in the case of a Multigraph, there could be several), this is denoted $\Gamma(v) = \{w \in V \mid \{v, w\} \in E\}$. Finally, the degree of a vertex v, denoted by d_v, represents the number of edges incident on v and therefore the number of neighbors of v (at least in simple graphs). We can therefore say that $d_v = |\Gamma(v)|$.

This defines the basis in terms of graph notations, although other concepts will be developed in the next sections, as well as some algorithm-specific notations. More details can be found about *Graph Theory* in books such as this one [22].

2.3 Graph Homomorphism

The reason we are interested in homomorphism is because we are going to reuse this concept to generalise an aspect much later in this thesis (see 4.5).

In concrete terms, two graphs are homomorphic if they share the same relationships with their vertices and edges. More formally, if we define our graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, we need a function $f : V_1 \cup E1 \to V_2 \cup E_2$ that satisfies these two conditions:

- For every vertex $v \in V_1$, $f(v) \in V_2$.

- For every edge $e = \{u, v\} \in E_1$, $f(e) = \{f(u), f(v)\} \in E_2$.

If the homomorphism works in both directions, in other words it is a bijection, then it is called an isomorphism.

2.4 Graph Density

In this section, we present three concepts for measuring graph density (sparsity). This will allow us to express the complexity of algorithms not only on the basis of the number of vertices and edges, but also based on the density of a graph. There are many ways of measuring the density of a graph like the general density based on the number of possible edges, but also the ones we're most interested in are arboricity and average degeneracy.

2.4.1 General Edge Density

One of the first ways of defining the density of a graph is to calculate the ratio between the number of edges and the maximum number of possible edges. Even if this measurement doesn't provide much information, it is already a first indication of the general density of a graph. It can be defined as follows:

$$d(G) = \frac{m}{n(n-1)/2}$$

Where m represents the number of edges and $n(n-1)/2$ the maximum number of edges in an undirected graph. In the case of a directed graph, the division by 2 is removed.

2.4.2 Arboricity

Arboricity represents the minimum number of disjoint forests (acyclic graphs), i.e, the minimum number of disjoint sets of trees, to which the edges of a graph can be partitioned. This allows us to represent the density of a graph, in other words, the higher the arboricity, the denser it is, or the denser a specific part of the graph will be, and therefore it must contain a dense subgraph. Because the arboricity of a graph G will be the maximum value among a subgraphs $H \subseteq G$. This article [9] takes up the definition of Tutte and Nash-Williams (1961) who show that if k represents the arboricity of a graph G, this number will be equal to:

$$k = \max_{H \subseteq G} \left\lceil \frac{|E(H)|}{|V(H)| - 1} \right\rceil$$

The following Figure 4 shows the arboricity for the K_6 graph, which is a complete graph of size 6. We can clearly see that this is three and cannot be less, otherwise one of the subsets will form a cycle. The arboricity of a graph G is noted $a(G)$ and we can write $a(K_6) = 3$.

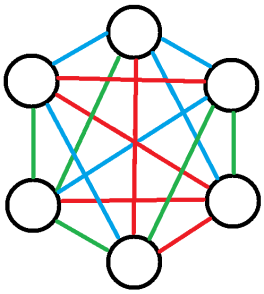


Figure 4: K_6 Arboricity example

2.4.3 Average Degeneracy

Not to be confused with graph degeneracy, which is the smallest k such that the graph is k-degenerate [10], which means that for a graph G and all its subgraphs, there exists a vertex of degree k at most. The average degeneracy [11, 1] is a finer measure of the density of a graph, represented by the average of the smallest degree among the vertices of each edge. This can be defined by the following formula,

$$\overline{\delta}(G) = \frac{1}{m} \sum_{uv \in E} \min\{d_u, d_v\}$$

2.5 Clustering Coefficient

2.5.1 What Is A Cluster ?

The concept of cluster in a graph allows us to determine the extent to which vertices form interconnected groups across different edges. In concrete terms, vertices in a cluster will be more connected to each other than those outside. Clustering allows you to see how vertices are organized within a graph, and to identify the different sub-groups.

This can be useful in the context of social networks, where each person is represented by vertices and edges represent their relationships. We can, for example, keep track of the different communities to which a person belongs, as well as his or her centers of interest, which comes in handy when targeting advertising campaigns.

Another example is communication networks. Here, each vertex represents a router/-computer and the edges the links. It's easy to imagine that clustering could be used to optimize the network by detecting different patterns in data traffic, or to detect certain frauds or anomalies.
There are many other possible applications, such as in the medical field, in the case of virus propagation, to detect the different clusters of the disease, or perhaps in neurology, etc.

2.5.2 Definition

Before going further, there is a distinction to make between what we call a local clustering coefficient, an average clustering coefficient and a global one. The local one is focused on a specific vertex and the interconnection between its neighbors, the average one is an average of all the local clustering coefficient of a graph meanwhile the global one will give us ratio between the number of triangle and the total possible ones.

For the (local) clustering coefficient, as mentioned in theses articles [2, 12], let $G = (V, E)$ be an unweighted, undirected simple graph. The (local) clustering coefficient for a node $v \in V$ in an undirected graph is defined by this formula,

$$cc(v) = \frac{|\{\{u, w\} \in E \mid u \in \Gamma(v) \text{ and } w \in \Gamma(v)\}|}{\binom{d_v}{2}}$$

The formula represents the ratio between the number of connections between v's neighbors and the maximum possible number of connections. Hence, the numerator represents the number of edges between v's neighbors, where u and w must be the ends of an edge and must be neighbors of v. This is the same as calculating the number of different $\{u, v, w\}$ triangles that include vertex v. And the denominator represents the maximum possible number of edges between neighbors, which we can simply reduce to calculating the number of combinations of size 2 from d_v, the degree of vertex v.

The big difficulty in a concrete case is calculating the numerator. Calculating the clustering coefficient is therefore a direct application of counting the number of triangles in a graph, (see 4.3).

As shown in the following Figure 5, the possible values of the clustering coefficient of a given vertex are between 0 and 1.

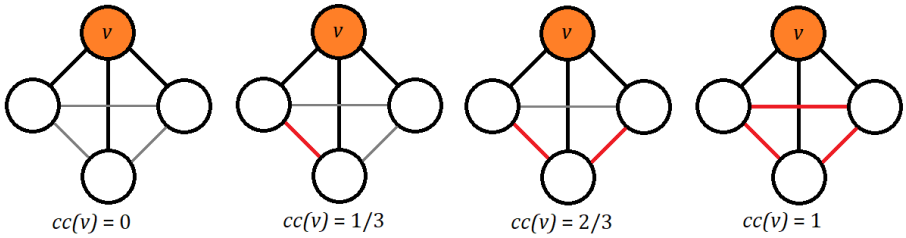


Figure 5: Clustering coefficient example

We can clearly see that when none of v's neighbors are interconnected, $cc(v)$ is 0. When they are all interconnected, $cc(v)$ is 1, and in all other cases a value in between.

The (average) clustering coefficient can therefore be expressed as follows, by taking a simple average of the local clustering coefficient,

$$cc_{avg} = \frac{1}{n} \sum_{v \in V} cc(v)$$

And finally the (global) clustering coefficient is defined as follows,

$$cc = \frac{3 \times \text{number of triangles}}{\text{number of triplets}}$$

We can thus see how important the number of triangles can be in detecting and analyzing clusters in a graph representing networks of different topics mentioned before. Since it is involved with the various types of clustering coefficients, namely local, average, and global.

3 Matrix Multiplication Techniques

Matrix multiplication is a fundamental operation in mathematics and computer science. The principle is to multiply and perform operations efficiently on data modeled as a matrix. These can range from simple Booleans to far more complex mathematical objects. This tool is used in many fields and has many applications [24], not least in,

Linear algebra Matrix multiplication can be used to solve systems of linear equations and calculate eigenvalues and vectors for various transformations.

Cryptography Some cryptographic algorithms and techniques, such as AES or matrix-based RSA, use matrix operations to achieve secure encryption and decryption.

Computer Graphics Since our screens are represented by a matrix of pixels, our GPUs spend their time performing operations on matrices, sometimes including matrix multiplication, for visual effects and rendering.

Machine Learning Matrices are widely used in linear regression and neural networks. Markov Chains use transition matrices for the various states, and sometimes need to be multiplied.

Graphs Of course, what we're interested in are graphs. One of the most common representations are adjacency matrices, and in order to detect paths of specific length we can directly call upon matrix multiplication. And also detect and count some sub-graphs as cycles.

Since we were working with graphs and our objective is to count cycles like triangles and 4-cycles, and since we'll at some point work with adjacency matrices. Because some algorithms require the application of matrix multiplication in order to square or cube the adjacency matrix. So, in this section, we'll look at several techniques for performing this multiplication, also see the history of the improvements done from the basic method to the last discoveries and finally what is actually used nowadays.

In the following Figure 6, we will find the curve of the evolution [25] for the complexity over time. Of course, we're not going to review all the discoveries, but rather the two major ones, that of the first sub-cubic algorithm by Strassen and the last great leap by Coppersmith and Winograd.

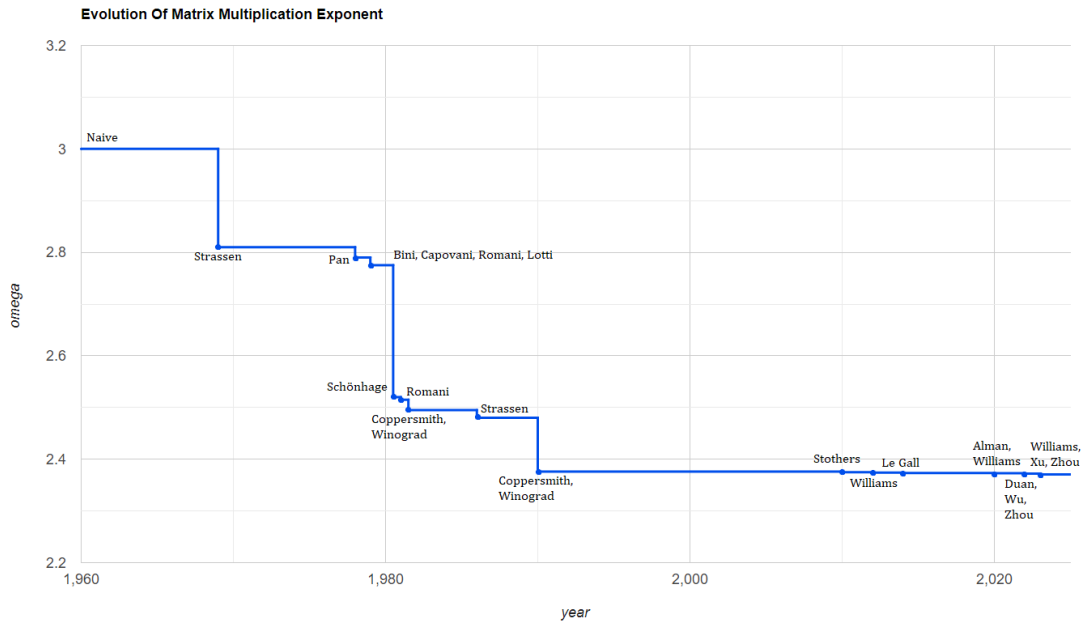


Figure 6: Evolution of the matrix multiplication algorithms time complexity given by the value of ω for $O(n^{\omega})$

3.1 Naive Iterative Way

To calculate a matrix multiplication [26], the first way to go is the naive one. The idea is quite simple: simply apply the basic equation defining matrix multiplication word for word. For matrices $C = AB$ where A is of dimension $n \times m$, B is of dimension $m \times p$, C must be $n \times p$. The equation represents the calculation of each element C_{ij} of the final matrix. To do this, we simply sum the multiplication of each element in the corresponding row and column of A and B to obtain the total sum in C_{ij}.

$$C_{ij} = A_{i1} \times B_{1j} + A_{i2} \times B_{2j} + \cdots + A_{im} \times B_{mj} = \sum_{k=1}^{m} A_{ik} \times B_{kj}$$

Obviously, if we want to apply this equation in an algorithm, we'll have to iterate over each i and j of the C matrix. To do this, we'll need to build an algorithm with two nested loops (one for i and the other for j) that will go up to n and p respectively. Then add another loop inside to represent our equation. In short, we'll end up with an algorithm of cubic complexity, at least in the worst case. The worst case occurs when $n = p = m$ (square matrices), which is the case when working with graphs and adjacency matrices that are always square. So if a counting algorithm uses this method, it will automatically have at least $O(n^3)$ time complexity.

3.2 Divide And Conquer

The idea behind divide-and-conquer algorithms is to find an identity between a simple case and easily achievable operations. Then reuse that case recursively to solve more complex ones. In this section, we'll look at the simplest identity relation. That by calculating a 2x2 matrix represented by a certain number of simple arithmetic operations (polynomials). Then apply this recursively to larger matrices. Of course, there are identity relations with much larger matrices or more complex operations, but the following two examples are the most straightforward ones.

3.2.1 Base Method

As it turns out, there's a divide-and-conquer technique that recursively calculates each region of a matrix independently [23, p75–79]. To do this, however, the matrix must be square, which is not a problem when manipulating graphs, but it must also have a shape of $2^n \times 2^n$. This already limits the field of application. But it's still worth studying for future improvements based on this method.

The idea is based on the following statement which divide the result matrix C into 4 distinct polynomials based on matrices A and B , which we can apply recursively to calculate the final result of our algorithm,

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{bmatrix}$$

As you can see, this procedure will require 8 recursive calls (each multiplication) and each recursion will process $n/2$ part of the previous matrix. Don't forget that combining and dividing the matrix will require $\Theta(n^2)$. As a result, if we write the complexity as $T(n)$, we end up with a time complexity of $T(n) = 8T(n/2) + \Theta(n^2)$ which correspond to $O(n^{\log_2 8}) = O(n^3)$ if we apply the Master Theorem [13]. We can see that this approach doesn't do much in terms of time complexity. Nevertheless, it can be useful if you want to distribute the different iterations of the recursion over different machines, so as to perform the calculation in parallel.

3.2.2 Strassen Algorithm

As its name suggests, this algorithm, developed by Volker Strassen [14]('69), it's also a divide and conquer algorithm but it reduces the number of multiplications for each recursive call from 8 to 7. While this doesn't sound like much, it does make

the complexity sub-cubic.
To achieve this, he defined the following 7 products:

- $P_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22})$
- $P_2 = A_{22} \times (B_{21} - B_{11})$
- $P_3 = (A_{11} + A_{12}) \times B_{22}$
- $P_4 = (A_{12} - A_{22}) \times (B_{21} + B_{22})$
- $P_5 = A_{11} \times (B_{12} - B_{22})$
- $P_6 = (A_{21} + A_{22}) \times B_{11}$
- $P_7 = (A_{11} - A_{21}) \times (B_{11} + B_{12})$

We can then reuse theses 7 products the following way to compute the different parts of the matrix C by building 4 new and more complex polynomials as follow:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} P_1 + P_2 - P_3 + P_4 & P_5 + P_3 \\ P_6 + P_2 & P_5 + P_1 - P_6 - P_7 \end{bmatrix}$$

Finally we got this in algorithmic way:

Algorithm 1 Strassen Matrix Multiplication

Require: Square matrices A and B of size $n \times n$
Ensure: Matrix C as the result of $A \times B$
if `len`$(A) <= 2$ **then**
    return $A \times B$
Split A and B into four, (a,b,c,d) and (e,f,g,h), each of size $n/2 \times n/2$
Compute the following seven products:
$P_1 \leftarrow$ `Strassen`$(a + d, e + h)$
$P_2 \leftarrow$ `Strassen`$(d, g - e)$
$P_3 \leftarrow$ `Strassen`$(a + b, h)$
$P_4 \leftarrow$ `Strassen`$(b - d, g + h)$
$P_5 \leftarrow$ `Strassen`$(a, f - h)$
$P_6 \leftarrow$ `Strassen`$(c + d, e)$
$P_7 \leftarrow$ `Strassen`$(a - c, e + f)$
Compute the four blocks of the resulting matrix C:
$C_{11} \leftarrow P_1 + P_2 - P_3 + P_4$
$C_{12} \leftarrow P_5 + P_3$
$C_{21} \leftarrow P_6 + P_2$
$C_{22} \leftarrow P_5 + P_1 - P_6 - P_7$
return C

If we were to calculate the complexity of this algorithm. It turns out that the lines 3 and 13 to 16 perform in $\Theta(n^2)$ and the lines 5 to 11 perform in $T(n/2)$, since it executes recursively for this same portion of the matrix. This gives the following complexity for $T(n)$,

$$T(n) = 7T(n/2) + \Theta(n^2)$$

Again, by using the Master Theorem [13] we end up with the following time complexity, which is finally sub-cubic: $O(n^{\log_2 7}) \approx O(n^{2.8074})$. In the following sections, we'll look at techniques for reducing this complexity even further. But to this day, this algorithm remains one of the most practical to implement.

3.3 Laser Method

The laser method uses a rather indirect technique which is very complex. This will be a rough summary explained in details here [15, 27]. The idea is to reduce the matrix multiplication problem MM to an algebraic problem P. Basically, with a polynomial representing matrix multiplication, trying to construct a new set of polynomials used in a known problem and which is known to have an efficient algorithm for solving it. To do this, we're going to use what are known as Tensors, which represent, among other things, matrix multiplication. These same Tensors will possess a Rank that will enable us to define ω, the matrix multiplication exponent that will define the complexity of the matrix multiplication algorithm as $O(n^\omega)$.

3.3.1 Tensor And Matrix Multiplication Tensor

To do this, we use what's known as a Tensor. This is a representation used to indicate which inputs to use from the input matrices to obtain the output matrix. The simplest example is a 2×2 matrix. For example, as $c_2 = a_1b_2 + a_2b_4$, in the following Figure 7, the entries $\langle a_1, b_2, c_2\rangle$ and $\langle a_2, b_4, c_2\rangle$ are set to 1.

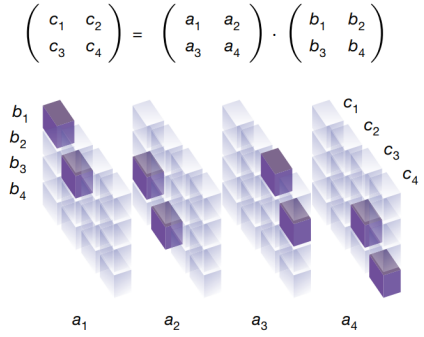


Figure 7: Tensor representing the computation of a 2×2 matrix

As a result, we end up with a polynomial of the following form, which represents a matrix multiplication in order to transform it into another polynomial.

$$\langle a, b, c \rangle = \sum_{i \in [a]} \sum_{j \in [b]} \sum_{k \in [c]} x_{ij} y_{jk} z_{ki}$$

It turns out that a Tensor T is most commonly represented by the following form, in a field $\mathbb{F}$ and variables x_i, y_i, z_i belong to X, Y, Z where a_{ijk} will act as a coefficient from a field $\mathbb{F}$. In a tensor where a_{ijk} can only be equal to 0 or 1, $x_i y_j z_k$ will be non-zero in T if $a_{ijk} \neq 0$. There is a decomposed form of a_{ijk} into $\alpha_i, \beta_j, \gamma_k$ if T has a rank 1. So we end up with both representations,

$$T = \sum_{i=1}^{|X|} \sum_{j=1}^{|Y|} \sum_{k=1}^{|Z|} a_{ijk} \cdot x_i y_j z_k = \left(\sum_{i=1}^{|X|} \alpha_i \cdot x_i \right) \cdot \left(\sum_{j=1}^{|Y|} \beta_j \cdot y_j \right) \cdot \left(\sum_{k=1}^{|Z|} \gamma_k \cdot z_k \right)$$

This will allow us to define the rank of a Tensor and therefore ω.

3.3.2 Tensor Rank And Matrix Multiplication Exponent

For a Tensor T, it has rank 1 only if the second representation is possible, for example, if $a_{ijk} = \alpha_i \cdot \beta_j \cdot \gamma_k$ for all i, j, k. So if $R(T)$ represents the rank of the Tensor T. $R(T)$ represents the number of rank 1 sub-Tensors to recompose the original tensor T.
It can be decomposed as follows: $T_1 + T_2 + ... + T_{R(T)} = T$.
Another representation exists, the asymptotic rank, where n is the number of sub-tensors and $T^{\otimes n}$ the sum of n sub-tensors,

$$\tilde{R}(T) = \lim_{n \to \infty} \left(R(T^{\otimes n}) \right)^{\frac{1}{n}}$$

So to define the matrix multiplication exponent ω, we can return to our matrix multiplication Tensor written as $\langle a, b, c\rangle$ and a matrix multiplication of size $O(n^{\log_q(r)})$. For any positive integer q and r, $R(\langle q, q, q\rangle) = r$, so we can define ω as,

$$\omega := \inf_{q\in\mathbb{N}} \log_q R(\langle q, q, q\rangle) \text{ or } \omega = \log_q \tilde{R}(\langle q, q, q\rangle)$$

For example, Strassen [14]('69) showed that $R(\langle 2, 2, 2\rangle) \leq 7$, which implied $\omega \leq \log_2(7) < 2.81$.

3.3.3 General Idea

After Strassen, it wasn't until the end of 1980s that the $\omega < 2.4$ mark was reached. And to do this, the main technique used since then and still used today is the Laser Method or similar principles. Since this is an indirect method, an intermediate Tensor T must be chosen to approach the original matrix multiplication Tensor.

To do this, we need to do two things: The first is to prove that T has a low asymptotic rank, which shows us that T can be computed efficiently. The second is that T has a great chance of computing a matrix multiplication by restricting T to a sum of several matrix multiplication Tensors. In order to achieve this restriction, the idea is to zero out certain variables in the intermediate Tensor T in order to simplify it. There are several techniques and steps to achieve this, which are explained here [15].

The most commonly used approach is to use the Tensor introduced by Coppersmith-Winograd [16]('87), denoted CW_q where q represents the size of the Tensor. This is given by the following formula,

$$CW_q := x_0y_0z_{q+1} + x_0z_{q+1}y_0 + x_{q+1}y_0z_0 + \sum_{i=1}^{q}(x_0y_iz_i + x_iy_0z_i + x_iy_iz_0)$$

CW_q has been used extensively up to the present day, the main change being the value of q which in the original work was equal to 2, but today variants of this Tensor are used with a size of up to $q = 32$.

3.4 Basic Linear Algebra Subprograms (BLAS)

The previous techniques have enabled us to see the progress made in terms of complexity in matrix multiplication. Indeed, we have seen the extent to which matrix exponent multiplication has decreased with the different types of solution proposed. But in practice, one might ask which of these solutions are implementable and are there more efficient solutions ?

In practice, although the naive iterative way and the divide and conquer solutions are obviously implementable. The first is just not efficient enough because of its complexity and the second is difficult to apply to real cases with real graphs of any size since it only works with graphs of size $2^n \times 2^n$ unless you make adaptations such as resizing the matrix or slicing it in more intelligent ways. As far as the laser method is concerned, this is a much more theoretical approach which does indeed prove that ω can be reduced, but in practice it is difficult to implement.

That's why we're going to talk about a final solution called BLAS for Basic Linear Algebra Subprograms. This is a widely used solution whose highly optimised implementations already exist in many programming languages.

3.4.1 Description of BLAS

BLAS [33] was originally a standardised interface describing routines for solving linear algebra problems. These include dot products, operations involving matrices such as decomposition or multiplication, etc. BLAS was originally developed for the Fortran language, included to LAPACK [36] (linear algebra package) and has become a standard in the field of linear algebra. Then, several implementations were made, notably in C with CBLAS, resulting in highly optimised libraries such as OpenBLAS [35], which is optimised for many architectures by taking advantage of certain features such as SIMD [37] (single instruction multiple data) instructions, caches and the usage of divide and conquer technique with for more efficient memory usage. But there are obviously other versions, such as Jblas for Java, pyblas for Python and versions optimised for GPU computing.

BLAS is organised into 3 levels, each corresponding to linear algebra functions of increasing complexity. Each level is made up of different operations, categorised as simple *S*, double *D* precision, complex *C* and double complex *Z*, to meet different user needs.

Level 1 This first level corresponds to relatively simple operations on vectors such as dot products, Euclidean norms and additions of the form: $y = \alpha x + y$. These include *SDOT*, *SNRM2* and *SAXYP*, which are the corresponding single-precision functions where the first letter indicates the precision. The *S* can therefore be replaced to meet the required precision.

Level 2 This one corresponds to matrix-vector operations such as multiplication $y = \alpha Ax + \beta y$. There are therefore different functions depending on whether we

are dealing with a simple, symmetrical or triangular matrix such as *SGEMV*, *SSYMV*, *STRMV* etc.

Level 3 This last corresponds to matrix-matrix operations, in this case matrix multiplication, again with simple, symmetrical or triangular matrices such as *SGEMM*, *SSYMM*, *STRMM*, etc.

It's precisely the BLAS level 3 functions that we're interested in for performing matrix multiplication as $C = \alpha AB + \beta C$. So we have *SGEMM* and *DGEMM*, which correspond to the multiplication of general matrices with elements in single and double precision, respectively.

(S/D)GEMM(TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB, BETA, C, LDC)

Where:

- **TRANSA, TRANSB**: Specify whether to transpose matrices A and B.
- **M, N, K**: Dimensions of matrices A, B, and C.
- **ALPHA**: Scalar multiplier for matrices A and B.
- **A, LDA**: Matrix A and leading dimension of A.
- **B, LDB**: Matrix B and leading dimension of B.
- **BETA**: Scalar multiplier for matrix C.
- **C, LDC**: Matrix C and leading dimension of C.

But since we're going to work with graphs, our matrices, which correspond to the adjacency matrices of the graphs, will be symmetrical. This is why we can also consider using level 3 functions *SSYMM* and *DSYMM*.

(S/D)SYMM(SIDE, UPLO, M, N, ALPHA, A, LDA, B, LDB, BETA, C, LDC)

Where:

- **SIDE**: Specifies whether matrix B appears on the left or right side of matrix A.
- **UPLO**: Specifies whether the lower or upper triangle of matrix C is used.
- **M, N**: Dimensions of matrix C.

3.5 Summary of Matrix Multiplication Methods

In our explanation of the different techniques of matrix multiplication, in our context where we are implementing algorithms for counting triangles and 4-cycles, we have explored several important methods which have different applications. Starting with the *Naive Iterative Way*, we have seen the basis of matrix multiplication and the fact that it is fundamentally a cubic-complexity solution problem. But this gave us a reference point.

Then we saw a first improvement thanks to the *Divide And Conquer* strategy. In particular, Strassen's Algorithm shows how a slightly more elaborate decomposition can significantly reduce complexity. These are the last algorithms that can still be implemented manually.

Next, we turned to the *Laser Method*, a more theoretical approach which seeks to reduce matrix multiplication to an algebraic reduction problem. Even if this isn't very useful in practice, it has enabled us to highlight the matrix multiplication exponent ω.

Finally, on the more practical side, we explained *Basic Linear Algebra Subprograms (BLAS)*, which provide highly optimized routines that take into account hardware specificities for matrix multiplication. These are perfectly suited to our purpose, which is the development of algorithms for fast triangle and C_4 counting.

4 Algorithms and Implementations

In order to discuss the different algorithms and their implementation, in this section we'll first talk about *Programming Language Choice*, which is the first thing to do before starting any implementation. Then we'll talk briefly about our *Experimental Setup And Methodology* to show how we worked. Finally, we'll get to the heart of the matter, the algorithms for *Triangles (C_3/K_3)* and *4-Cycles (C_4)* followed by a *Generalization For k-Cycles (C_k)* to have a common method for any cycle length.

4.1 Programming Language Choice

In this section, we will cover the way how we choose our programming language for a project like in this master thesis where the goal is to implement and study algorithms in an efficient way. To do that, we will first talk about the *Types Of Programming Languages*, then we will do *Languages Comparisons* to see that the one which is the most suitable is C++ and finally check the *Common Array "Like" C++ Data Structures* to be able to do the right choices during an implementation.

4.1.1 Types Of Programming Languages

When it comes to choosing a programming language for a project, one of the first things to know is which type to go for. There are hundreds of programming languages, all of which work in different ways, and this has a direct impact on performance, portability, ease of development and security. But in reality, languages can be broadly classified into 3 main categories, each of which has more or less the same consequences. Firstly, compiled languages, then interpreted languages and finally hybrid languages.

Compiled

Compiled languages are corresponding to those in which the source code is transformed to machine code so that it can be executed by the processor. This is done by what is known as a compiler, whose task is divided into 2 main parts: the analysis phase and the generation phase. Without going into too much detail, during these phases, the symbols are analysed and ordered in such a way as to form what is known as a syntax tree. This tree corresponds to the logic of the program and will then be read to generate code, first intermediate code and then final code. So there

will be management of the symbols encountered, as well as management of errors if the code is incorrectly written.

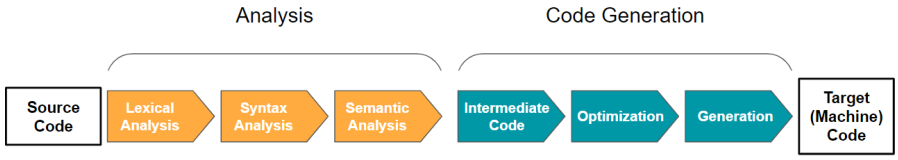


Figure 8: Compilation phases

The main advantage of compiled languages is their performance: thanks to the optimisation phase, the program can be optimised for the architecture on which it will run. Also, the source code is not needed by the user which is a sort of security. On the other hand, compilation is time-consuming, especially if the code is long. And the second problem is that machine code is specific to each machine. More precisely, to each architecture and operating system, which is why there are different versions of compilers.

For example, among the compiled languages we can cite C, C++, Rust, Fortran, etc.

Interpreted

Unlike compilers, which generate optimized code for subsequent execution, interpreted languages use an interpreter to read the source code, translate it and execute it in real time. In other words, it does this line by line or instruction by instruction as the program runs. You might think that this would be more efficient, as compilation is a long process. But in the case of multiple executions of interpreted code, the interpreter will have to repeat the same task each time. Whereas with compiled code, all you have to do is re-execute the optimized machine code. In addition, the user side needs the source code, which can be dangerous for sensitive projects. The only advantages would be flexibility, since you don't have to recompile every time, and portability, since there's no machine code specific to each platform. In the case of large-scale projects requiring multiple execution of the same task, or large-scale performance analysis (such as this one), interpreted languages are not really suitable.

We can cite mainly web focused languages, such as Javascript, PHP or Ruby. But one of the main exception is Python which is also used in analytic and machine

learning despite being an interpreted language.

Hybrid

Finally, there's a compromise between the two, hybrid languages. They try to combine both compilation and interpretation. How this works depends strongly from one language to another, but the idea remains the same. Generally speaking, there's always a compiler that transforms the code into intermediate code, like Java bytecode (.class). But what's really going to change is that hybrid languages call on a virtual machine. The role of this virtual machine is to choose what will be compiled into machine code and what will remain to be interpreted.

In Java, there's the Java Virtual Machine (JVM), which has an interpreter but also a compiler called JIT for the compilation part. We can also mention C# with the Common Language Runtime (CLR), which is the same equivalent of the virtual machine. Here, in most cases, the compiler will remain similar whatever the platform, since the intermediate bytecode is standardized. But it's the Virtual Machine that can change from one platform to another. For example, Android uses Android Run Time (ART) and not the JVM to run Java and Kotlin programs.

This delivers better performance than 100% interpreted languages, while retaining a certain degree of flexibility.

4.1.2 Languages Comparisons

For this thesis, we need a high-performance and efficient language for implementing and executing triangle-counting algorithms. This will enable us to make comparisons as accurately as possible. But also a language that's popular enough, for code comprehension and the availability of different libraries.

Performance and Efficiency

As far as performance is concerned, numerous studies have been carried out, such as this article [17] comparing the performance and efficiency of tens of languages. They made comparisons on a whole set of operations from The Computer Language Benchmarks Game [28] (CLBG), such as *n-body*, *regex-redux* or *binary-trees* as follow:

binary-trees				
	Energy (J)	Time (ms)	Ratio (J/ms)	Mb
(c) C	39.80	1125	0.035	131
(c) C++	41.23	1129	0.037	132
(c) Rust $\Downarrow_2$	49.07	1263	0.039	180
(c) Fortran $\Uparrow_1$	69.82	2112	0.033	133
(c) Ada $\Downarrow_1$	95.02	2822	0.034	197
(c) Ocaml $\downarrow_1$ $\Uparrow_2$	100.74	3525	0.029	148
(v) Java $\uparrow_1$ $\Downarrow_{16}$	111.84	3306	0.034	1120
(v) Lisp $\downarrow_3$ $\Downarrow_3$	149.55	10570	0.014	373
(v) Racket $\downarrow_4$ $\Downarrow_6$	155.81	11261	0.014	467
(i) Hack $\uparrow_2$ $\Downarrow_9$	156.71	4497	0.035	502
(v) C# $\downarrow_1$ $\Downarrow_1$	189.74	10797	0.018	427
(v) F# $\downarrow_3$ $\Downarrow_1$	207.13	15637	0.013	432
(c) Pascal $\downarrow_3$ $\Uparrow_5$	214.64	16079	0.013	256
(c) Chapel $\uparrow_5$ $\Uparrow_4$	237.29	7265	0.033	335
(v) Erlang $\uparrow_5$ $\Uparrow_1$	266.14	7327	0.036	433
(c) Haskell $\uparrow_2$ $\Downarrow_2$	270.15	11582	0.023	494
(i) Dart $\downarrow_1$ $\Uparrow_1$	290.27	17197	0.017	475
(i) JavaScript $\downarrow_2$ $\Downarrow_4$	312.14	21349	0.015	916
(i) TypeScript $\downarrow_2$ $\Downarrow_2$	315.10	21686	0.015	915
(c) Go $\uparrow_3$ $\Uparrow_{13}$	636.71	16292	0.039	228
(i) Jruby $\uparrow_2$ $\Downarrow_3$	720.53	19276	0.037	1671
(i) Ruby $\Uparrow_5$	855.12	26634	0.032	482
(i) PHP $\Uparrow_3$	1,397.51	42316	0.033	786
(i) Python $\Uparrow_{15}$	1,793.46	45003	0.040	275
(i) Lua $\downarrow_1$	2,452.04	209217	0.012	1961
(i) Perl $\uparrow_1$	3,542.20	96097	0.037	2148
(c) Swift	n.e.			

Figure 9: Comparison of the Time, Energy and Memory usage for allocating, traverse and deallocating binary-trees from this article [17]

In this *binary-trees* operation comparison, where (c) represent the compiled languages, (i) the interpreted and (v) the hybrid languages (because they have a virtual machine). We can clearly see that the interpreted languages such as PHP, Python or Javascript have generally poor performance and efficiency comparing to hybrid languages in the middle and finally the compilied languages such as Fortran, Rust and C/C++ on the top. The results are relatively the same for the others operations. The only non-general behavior is the memory consumption where for example Java seems to be more greedy than few interpreted languages.

It's now clear that we need a **compiled** language to perform our implementation. Indeed, if we do the implementation with an interpreted language, the loss of time is not negligible since the running times are longer in a 10 or even a 100 factor.

Popularity

All that remains is to find out which compiled languages stand out among the most popular languages. To do this, there are a number of rankings, including that of the TIOBE organisation called the TIOBE Index [29]. This organisation specialises in measuring the quality of software used by numerous companies. To produce their index, they calculated the rating of each language by counting its occurrence on the first 100 pages of several search engines with the `"<language> programming"` query.

March 2024 Rank	Programming Language	Rating
2	C	11.17%
3	C++	10.70%
8	Go	1.59%
14	Fortran	1.22%
16	Swift	1.08%
17	Rust	1.03%

Table 1: Compiled programming language in the top 20 TIOBE Index

To choose our programming language, we can start by removing languages that don't offer much flexibility, in other words C and Fortran. These have a procedural paradigm, which limits modularity, particularly if we want to encapsulate attributes and functions in an instance in order to create generic implementations. This is something that multi-paradigm or object-oriented languages generally allow. Go has relatively poor performance, Swift is specific to Apple and Rust, the first stable version is from 2015, is the most recent language and may not be mature and may lack certain libraries. Which leaves us with C++.

4.1.3 Common Array "Like" C++ Data Structures

Since the choice of programming language is C++, we also need to be aware of the different data structures that exist. Here we're not going to give a complete review of what C++ is and all its special features. Instead, we'll talk about a few common data structures that can be useful for building more complex data structures that can represent a **Graph**. In particular, how they work and their complexity.

std::vector are an evolution of std::array, which is the object-oriented version of C legacy arrays. The only difference with std::array is that vectors are dynamic arrays, meaning that their size can be modified. This is made possible by the fact that vectors are allocated in the heap and not in the stack. The elements are stored contiguously and if there isn't enough space, the vector is relocated. This is only penalising in algorithms that regularly modify the size of a vector, which will not be our case.

std::list are doubly linked lists. In practical terms, this means that each element has a pointer to its next and previous elements. This allows non-contiguous allocation in memory, so it not have to relocate anything if you add an element, only the pointers change. On the other hand, there is a fragmentation of the array in memory, which can be costly if long iteration is required.

std::set are used to store unique and sorted elements. This is implemented via a balanced binary search tree. All operations are performed in logarithmic complexity.

std::unordered_set it's like a std::set but the elements don't need to be sorted. It is implemented using a hash table. This allows constant-time operations.

Operation	`std::vector`	`std::list`	`std::set`	`std::unordered_set`
Insert	$O(n)$ or $O(1)$	$O(1)$	$O(log\ n)$	$O(1)$, $O(n)$ (worst)
Delete	$O(n)$	$O(1)$	$O(log\ n)$	$O(1)$, $O(n)$ (worst)
Search (by value)	$O(n)$	$O(n)$	$O(log\ n)$	$O(1)$, $O(n)$ (worst)
Access (by index)	$O(1)$	/	/	/

Table 2: Array "Like" C++ data structures complexities

It now is possible to choose how to build more complex data structures such as matrices or adjacency lists to represent our graphs, so they can be well suited to the algorithms.

4.2 Experimental Setup And Methodology

For this thesis, our implementations depend on two fundamental libraries: the C++ Standard Library [34] and OpenBLAS [35]. The C++ Standard Library is the foundation for this programming language, providing the essential data structures, algorithms and components for manipulating data and creating a program written in C++. It contains many useful headers, some of which implement the data structures discussed above, such as *<algorithm>*, *<list>*, *<vector>*, *<thread>*, *<chrono>*, *<set>*, and the list goes on with more than a hundred. To complete this, we're going to use OpenBLAS which is one of the open-source implementations of BLAS routines. There are many others, such as FLAME and Intel MKL, but the advantage of OpenBLAS is that, as its name suggests, it is open-source and, in terms of performance, it is optimised for many different architectures and operating systems. We're going to use it to perform matrix multiplication operations quickier, with the implementation of the 4 BLAS functions from level 3 mentioned earlier (see 3.4). They are called *cblas_dgemm()*, *cblas_dsymm()*, *cblas_sgemm()* and *cblas_ssymm()*. But there is one last function, *openblas_set_num_threads()*, which allows you to specify to OpenBLAS the number of threads with which you want it to perform operations. By default, OpenBLAS uses several threads, which is why we're going to start by explicitly specifying the use of a single thread, but this can be useful for more advanced analyses involving parallelism.

Concerning the used versions. For C++, we use the g++ 13.2.0 (rev3) compiler from MinGW-w64 and for OpenBLAS the version 0.3.26. For the compiling, the option *-lopenblas* for linking the OpenBlas library, the option *-O3* for high optimisation and we specify the C++ version 17 by using *-std=c++17*.

The graph datasets we will be using are those provided by the Stanford Network Analysis Platform (SNAP) [38]. Developed by the SNAP group at Stanford University, they provide a whole range of tools and libraries for studying graphs. But what interests us most of all is their repository, which contains various graphs representing social networks, web graphs, citation networks, communication networks, etc. These graphs are derived from real data and are documented with a range of information, including the number of triangles, the average clustering coefficient and the density. But since some of them are directed graphs, we undirected them to be able to use them with our algorithms. In the past, we have already had the opportunity to use some of these graphs in algorithmic courses.

There are several possible scenarios for the correctness of the algorithms. Either the result is provided by SNAP, as in the case of the number of triangles present in the graphs, then we have run the algorithm in question and checked whether the result obtained is what was expected. If the result is not provided by SNAP, such as the number of C_4 in a graph, then we either compared several algorithms and checked that the results were the same or, to be sure, we created smaller instances using a graph editor such as the one provided on CS Academy [30] and checked visually whether the result was correct.

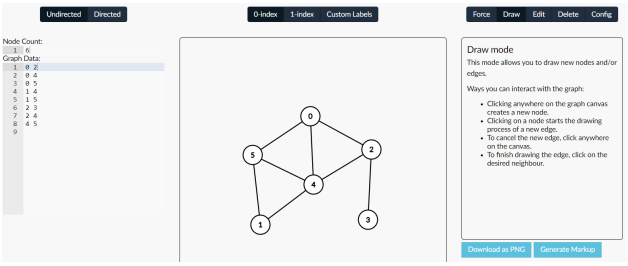


Figure 10: CS Academy's Graph Editor

In order to carry out the various computational experiments for this master's thesis. The tests were carried out on a laptop equipped with an Intel Core i7 -7700HQ @ 2.80Ghz (Max 3.5Ghz in Turbo Mode), with 4 cores and 8 logical processors. The machine is equipped with 16GB @ 2400Mhz of memory and runs Windows 10 Home. The results were generated by averaging 10 runs. The computer was plugged

to the socket, in order to avoid random fluctuations due to power saving but since the tests were done on multiple days, some fluctuations can occurs the day to day, so for each plot we tried to record all the data the same day. The results are shown on plots generated using Matplotlib in Python.

4.3 Triangles (C_3/K_3)

In this section, we'll look at algorithms for counting the number of triangles T in a graph. Counting triangles, noted (C_3/K_3), is very interesting indeed, as it enables us, for example, to calculate what we call the clustering coefficient (see 2.5). As we said earlier, this enables us to analyze communities and groups within a graph. To do this, there are a multitude of algorithms of varying complexity, as cited in these article [2, 3, 4, 31].

In concrete terms, we're going to start with *Matrix Multiplication Based* algorithms designed to work exclusively with adjacency matrices. Then we'll compare two algorithms, *NodeIterator Vs NodeIterator++* , the first being a naive triangle counting algorithm and the second an optimised version. Finally, we'll take a look at some *Algorithms Designed For Listing* that we'll also adapt to a counting version in order to understand how much listing can cost.

Each time, we will try to understand what impacts the performances, whether in terms of data structure or graph specificity.

4.3.1 Matrix Multiplication Based

One way of counting the number of triangles in a graph would be to use matrix multiplication to find the number of paths of length 3 starting and ending at the same vertex and then divide this number by 6 (by 3 for each vertex and by 2 for each direction), which will in effect give us the number of triangles. This principle is based on the fact that $\sum_i (A^p)_{ii}$ gives us the number of paths of length p, starting and ending at the same vertex. Where A represents the adjacency matrix of the graph, n the size of it (in the following algorithm) and thus also the number of vertices and p the size of the path we want to count.

Algorithm 2 Counting Triangles by Matrix Cube

```
1: T ← 0
2: compute A^3
3: for i = 1 to n do
4:     T ← T + A_ii
5: return T/6
```

Obviously, computing the cube of an adjacency matrix isn't going to help at all. Indeed, this will require to do two matrix multiplication instead of one. This is why there's an other algorithm [31] . This algorithm is still based on matrix

multiplication, the principle of this algorithm is that $\sum_{i,j} A_{ij} \cdot (A^2)_{ij}$ also gives us the number of triangles in the graph by avoiding of cubing the matrix.

Algorithm 3 Counting Triangles by Matrix Squaring

1: $T \leftarrow 0$
2: compute A^2
3: **for** $i = 1$ to n **do**
4: **for** $j = 1$ to n **do**
5: $T \leftarrow T + A_{ij} \cdot (A^2)_{ij}$
6: **return** $T/6$

Clearly, the part of the algorithm from line 3 to line 5 can be executed in $O(n^2)$. The question is, how fast can we calculate the matrix multiplication ? However, we can already say that the complexity is $O(n^2) + O(n^\omega)$ where $O(n^\omega)$ represents the time at which we can calculate this multiplication and ω the matrix multiplication exponent. Since, the upper bound is the multiplication, we can affirm that this algorithm has a complexity of $O(n^\omega)$. Because already talked about the different techniques in the dedicated section (see 3) and conclude that the best known bound is $\omega < 2.371866$ [18].

In order to implement these algorithms, we need to store the graph in the form of an adjacency matrix. Otherwise it would be impossible to perform these manipulations. We'll now see how these two algorithms behave in practice and why it's not a good idea to count triangles in a graph in this way.

Graph Sample and Data Structure

Matrix multiplication is notoriously slow on computers. As mentioned previously, this is why there are optimized libraries such as OpenBLAS. We have therefore chosen six reasonably sized graphs, with increasing n, available on SNAP to avoid extremely long execution times. These graphs have been undirected since a lot of them are directed by default.

Graph Name	n	m (undirected)	cc_{avg}	$d(G)$	T
email-eu-core	1005	25571 (16064)	0.3994	3.184%	105461
ego-facebook	4039	88234 (88234)	0.6055	1.082%	1612010
wiki-Vote	7115	103689 (100762)	0.1409	0.398%	608389
ca-HepPh	12008	118521 (118489)	0.6115	0.164%	3358499
p2p-Gnutella25	22687	54705 (54705)	0.0106	0.021%	806
cit-HepTh	27770	352807 (352285)	0.3120	0.091%	1478735

Table 3: 6 reasonably small graph with n increasing

These graphs are stored in a graph class called GraphAdjMatrixVV with an implementation based on an adjacency matrix, in this case in a 2D integer vector. In C++, this corresponds to a `vector<vector<int>>` structure. The reason for this choice is that, since we're going to be performing operations on matrices, we can't store them with Booleans (1 byte). We could just as easily use **short int** (2 bytes), capable of storing 65,536 different values. But if we're dealing with a very dense graph, say a complete one, we'll be limited to 256 vertices. The reason is as follows: if A is a 256x256 matrix filled with 1, A^3 will be a 256x256 matrix filled with 65,536. In reality, this won't happen with sparse graphs, but it's possible that with larger graphs it will exceed for few cells. This is why we have chosen **int** (4 bytes) as the type whose positive limit is 2147483647 when signed and 4294967295 when unsigned, which is more than enough.

Based on this structure, we can already see why we have limited ourselves to graphs of this size. In fact we can already calculate that **cit-HepTh** will take $(27770^2) \times 4$ bytes which already corresponds to more than 3GB in memory.

Using Naive Matrix Multiplication

One of the first comparisons we can make is to compare the two algorithms using a naive matrix multiplication algorithm (see 3.1). It goes without saying that the performance is likely to be extremely poor, but this will give us a basis for seeing what can be achieved with BLAS.

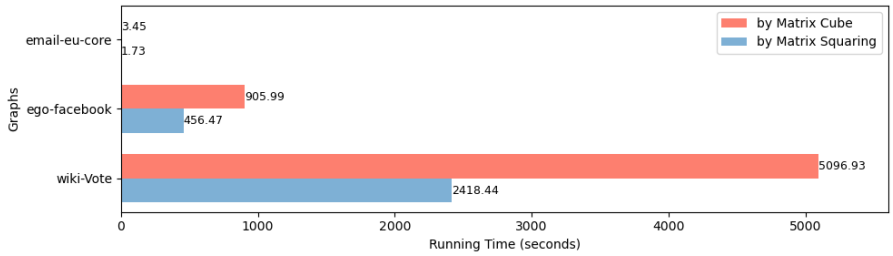


Figure 11: Counting triangles matrix cube vs matrix squaring using a naive matrix multiplication approach

There are a few things we can see from this first plot. Firstly, the extremely long execution time of the algorithms when n increase. This is the reason why only the first three graphs have been tested, because if we had to test the others, we'd probably have to run them for several hours or even days. Despite the fact that it is difficult to demonstrate complexity with only 3 graphs, we can still see the cubic

nature of naive matrix multiplication. Secondly, we can see that the difference in terms of execution between the Matrix Cube algorithm and Matrix Squaring is a factor of 2. Which makes sense, because in one we're doing two multiplication and in the other we're doing only one.

In terms of memory usage, this approach of using adjacency matrices clearly shows its limitations, with the following results:

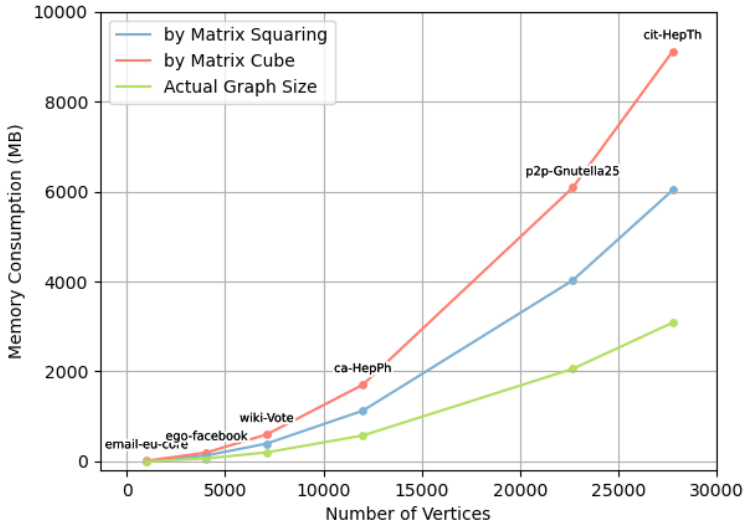


Figure 12: Memory Consumption of matrix cube vs matrix squaring

First of all, even before looking at the graph, if we think about it a little we can already predict the amount of ram that will be used. In fact, in order to carry out matrix multiplication, we need to allocate space for the input matrix (from the graph) as well as for the result. So it's normal to have a usage two times greater than what the graph is supposed to take up in terms of space. We can also see this in the difference in memory usage between the two algorithms, since to calculate the cube of the graph matrix we need to store the squared intermediate result in extra. Plus, we can see that memory usage increases with complexity squared, which makes sense since we are working with $n \times n$ matrices.

What we can already conclude is that there is clearly nothing we can do in terms of memory use. An adjacency matrix is just not a good data structure for counting triangles. But it would still be interesting to know the performance gains with BLAS and maybe it will be worth having such a high space complexity.

Using BLAS Matrix Multiplication

As mentioned previously (see 3.4), BLAS has several functions for matrix-matrix multiplication, SGEMM, DGEMM, SSYMM and DSYMM. Since we were working with int (4 bytes), the ideal would be to work with functions taking an integer matrix as a parameter, which BLAS does not offer. So what we did was to transform our matrices into floats (4 bytes) so that we could use the functions in simple precision. There's technically no point in using the double-precision functions, which take doubles (8 bytes) as parameters. All that will happen is that more memory will be used and, as the plot below shows, execution time will be longer. Despite that, the performances are still very impressive since the executions times decreased from several tens of minutes with the naive method to only few seconds with theses BLAS fonctions.

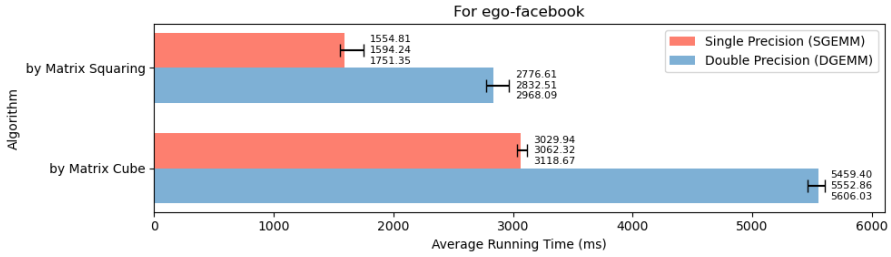


Figure 13: Execution times single vs double precision (10 runs average)

The last two things we can ask ourselves. The first is how the algorithms behave in terms of execution time depending on the size of the input graph. And the second is whether we can observe a performance gain by using the SSYMM method instead of SGEMM, because since we're working with graphs, our matrices are symmetrical.

We can see several things on the following plot, which compares the execution time of the two BLAS functions used by the cube and square counting algorithm, on the graphs cited above. Firstly, the execution time are still very impressive compared with the naive method, but unfortunately still seems to increase by a cubic factor. In addition, we can see that the difference in execution time between the functions dedicated to classical and symmetrical matrices (SGEMM and SSYMM) is not significant. At least this is not the case for our machine with the OpenBLAS library implementations with graph matrices. Perhaps there is a configuration where the impact of using SYMM functions is greater, but as this article [19] benchmarking Level-3's BLAS functions shows, this doesn't really seem to be the case either.

Lastly, there is still the same difference between the two algorithms, squaring and cube, since the multiplication operation is performed twice instead of once.

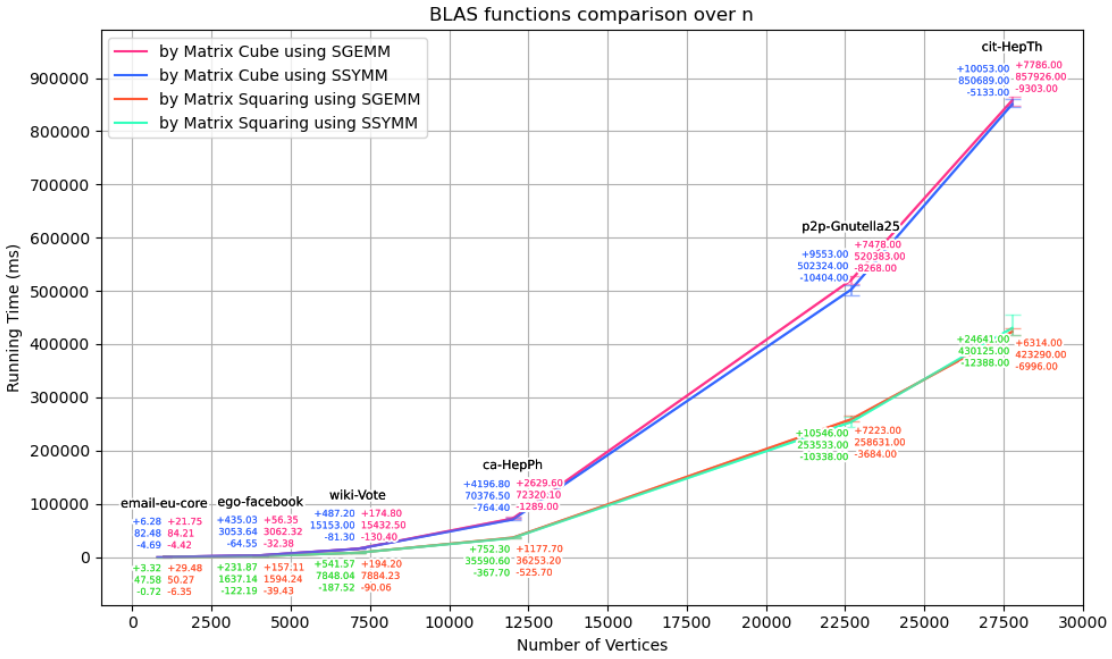


Figure 14: Execution times of the different BLAS functions over n (10 runs average)

If we have to summarise the triangle counting algorithms based on matrix multiplication. We have seen that the complexity in space and the amount of memory required is extremely high because of the data structure we use to model the graph, which is a matrix. In terms of time, even if the complexity is cubic, or at least it's theoretically impossible to go below $O(n^{\omega})$, the time savings made by using BLAS are enormous. It remains to be seen exactly how these times compare with the following algorithms.

4.3.2 NodeIterator Vs NodeIterator++

Another approach is that there are also naive ways of counting the number of triangles in a graph. This first algorithm is called NodeIterator [2], see algorithm (Alg:4), and, as its name suggests, it will iterate over each vertex v and check whether for each neighbor u of v another neighbor w of v is adjacent. This is equivalent to closing two paths leaving v and thus forming a triangle adjacent to v. Obviously, if you apply this procedure to each vertex, the total number of triangles counted will be multiplied by 6, 3 times per triangle and in each direction. That's why the counting is done by 1/2 and the total number is divided by 3.

Algorithm 4 NodeIterator(V,E)

```
1: T ← 0
2: for v ∈ V do
3:     for u ∈ Γ(v) do
4:         for w ∈ Γ(v) do
5:             if u, w ∈ E then
6:                 T ← T + 1/2
7: return T/3
```

If we try to analyze the time complexity of this algorithm, we can clearly see that it is based on vertex degree. The worst case is when our graph is complete, so the complexity is $O(n^3)$, which is uncommon in the real world. But it's still possible that in the real world, there are some nodes with a very high degree, for example with famous people in social networks. Which can rapidly have as a consequence to the algorithm to be quadratic ($O(n^2)$) in running time. Thus, the complexity can be summarized as follows: $O(\sum_{v \in V} d_v^2)$.

This second algorithm (Alg:5) is designed to correct NodeIterator's biggest problem. That of counting each triangle 6 times. To do this, the idea is not only to choose a responsible vertex among the 3 to count a triangle, but also to take the one with the lowest degree. If you choose the lowest degree, the iteration will be shorter and there will be fewer unnecessary checks. To do this, we consider $\succ$ as the total order of all vertices where $v \succ u$ means that $d_v \geq d_u$ (if equals the vertex id will order). Thus in the following algorithm, v will be responsible to count if its neighbors (in the corresponding triangle) have a higher degree.

Algorithm 5 NodeIterator++(V,E)

```
1: T ← 0
2: for v ∈ V do
3:     for u ∈ Γ(v) and u ≻ v do
4:         for w ∈ Γ(v) and w ≻ u do
5:             if u, w ∈ E then
6:                 T ← T + 1
7: return T
```

The running time of this algorithm is $O(m^{3/2})$. To illustrate this upper bound, the author of the article [2] takes the example of a lollipop graph. This is a graph with a clique of size $\sqrt{n}$, and the remaining $n - \sqrt{n}$ vertices are placed in a line starting from the clique. This example shows that this is the best possible case, since the number of triangles present in this type of graph is always $\binom{\sqrt{n}}{3} = \Theta(n^{3/2})$.

Which is in line with the stated running time because at each iteration we will at most count 1 triangle by treating 1 edge.

For the proof, if we go back to the original article describing this algorithm [5], they explain it in a slightly different way, but the principle remains the same. Basically, at each iteration they use a rooted spanning tree and verify that it is inside a triangle. If it is inside, this confirms the existence of the triangle and they remove this tree (this takes $O(m)$). This will decompose the graph into several connected components c, either c is smaller than $n - m^{1/2}$, or it is larger. In both cases the number of remaining iterations will be $m^{1/2}$, which is why at the end the algorithm takes $O(m^{1/2})O(m) = O(m^{3/2})$.

These algorithms can be implemented by storing the graphs in various ways. We no longer work with matrix multiplication, so we can obviously continue to store graphs using an adjacency matrix, or we could also see what happens if we use an adjacency list.

Graph Sample and Data Structures

However, to test these NodeIterator algorithms, since their complexity is no longer entirely related to n, we're going to use several graph samples. This will allow us to carry out a more in-depth analysis. We're obviously going to re-use the previous graph set (table 3), but its big problem is that these graphs have different average clustering coefficients, densities and numbers of triangles. To get a more accurate analysis, we're going to use SNAP's Internet peer-to-peer networks, which have the advantage of having relatively similar average clustering coefficients and densities.

Graph Name	n	m	cc_{avg}	$d(G)$	T
p2p-Gnutella08	6301	20777	0.0109	0.1046%	2383
p2p-Gnutella09	8114	26013	0.0095	0.0790%	2354
p2p-Gnutella06	8717	31525	0.0067	0.0829%	1142
p2p-Gnutella05	8846	31839	0.0072	0.0814%	1112
p2p-Gnutella04	10876	39994	0.0062	0.0676%	934
p2p-Gnutella25	22687	54705	0.0053	0.0212%	806
p2p-Gnutella24	26518	65369	0.0055	0.0186%	986
p2p-Gnutella30	36682	88328	0.0063	0.0131%	1590
p2p-Gnutella31	62586	147892	0.0055	0.0075%	2024

Table 4: Peer-to-peer network graphs

But that's not all, later on we'll be using other graphs to help us answer some questions about the differences in performance between NodeIterator and NodeIterator++.

As for the data structure we used. For an implementation with an adjacency matrix, we reused the GraphAdjMatrixVV class as explained in the previous section (see 4.3.1), which is based on a two-dimensional vector in C++. But for the implementation using an adjacency list, we had to choose a data structure that was as suitable as possible for the algorithms described. In concrete terms, we need to satisfy four requirements:

1. The ability to iterate over all the vertices $V(G)$.
2. Iterate over the neighbours of a vertex $\Gamma(v)$.
3. The ability to quickly retrieve the degree of a vertex d_v.
4. Check the existence of an edge.

This is why we have chosen to use a `vector<unordered_set<int>>` as the data structure to store our graphs which is defined in a class called GraphAdjListVUS.

Indeed, the choice of a vector as the global list for the vertices is explained by the need to iterate over all the vertices of the graph. Vector arrays are dynamic, which means that the elements are stored contiguously and are relocated if there's no more room, but since we're not changing the size of our graph and the elements are next to each other, iteration will be done in the most efficient way. This is not the case, for example, with list arrays, which are not contiguous and use pointers to form a double-linked list. The advantage of list is insertion and deletion in $O(1)$ but what we need is contiguity and reading.

Then, when it came to the unordered_set as a list of neighbors of a v vertex, the choice was a little more complicated, since we needed something that would satisfy requirements 2, 3 and 4. In fact, vector arrays already satisfy requirements 2 and 3, the problem is the number 4. Lookups by index are done in $O(1)$ for vectors, but by value we have to iterate and therefore it is in $O(n)$, which considerably slows down the verification of the existence of an edge. The same problem exists with list, but it's even worse because storage is not contiguous.

Finally, as far as unordered_set is concerned, it allows us to store unique things, which doesn't bother us since we're working with simple graphs, but what could bother us is the non-contiguity, for point 2. Fortunately, our graphs have a rather low density $d(G)$ and so, compared with the size of the graphs, the vertices have only a few neighbors. For point 3, the size is recovered in $O(1)$ as for vectors and for point 4, since unordered_set uses a hash table, the average lookup by value is $O(1)$.

One last thing, the choice of using **int** (4bytes) instead of **short int** (2bytes) is explained by the number of edges.

Adjacency Matrix vs Adjacency List

Before making detailed comparisons with the various samples, it's interesting to see how the NodeIterator and NodeIterator++ algorithms behave with an adjacency matrix and an adjacency list.

In the case of NodeIterator, with an adjacency list, the algorithm can be implemented in a straight-forward way, but for the adjacency matrix, since we can't iterate over the neighbors on lines 3 and 4 of the algorithm, we'll have to iterate over the n vertices and check the existence of the $\{u, v\}$ and $\{w, v\}$ edges. For NodeIterator++, with an adjacency list, for lines 3 and 4 the total order is checked in $O(1)$, since all you have to do is retrieve the size of the unordered_set for the degree. But with an adjacency matrix, things get more complicated, as you can't perform total order operations in $O(1)$. There are therefore two choices: Either we calculate d_v, d_u and d_w each time, which corresponds to a complexity of $O(n^3)$, since we iterate over n on lines 2,3,4. Or we sort the vertices by degree to calculate the total order and then run the algorithm, which corresponds to calculating all the degrees in a matrix ($O(n^2)$), sorting ($O(n\ log\ n)$) and running the algorithm ($O(m^{3/2})$).

We end up with the following 5 cases, here in a comparison for the ego-facebook graph:

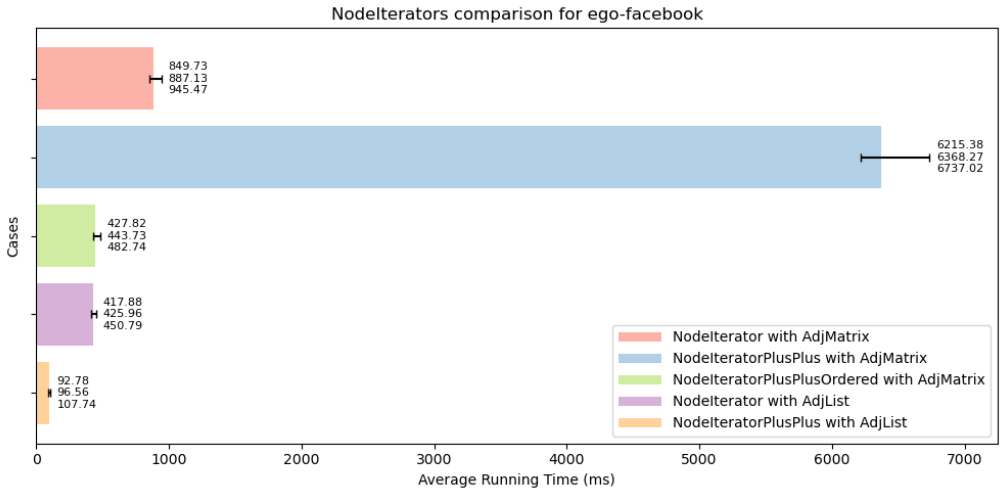


Figure 15: Comparison of NodeIterator algorithms on ego-facebook with an adjacency matrix and adjacency list (10 runs average)

Following this plot, we can already notice 3 things. The first and most obvious is the advantage of calculating the total order before running NodeIterator++ in the case of an adjacency matrix. In fact, it's just a waste of time to calculate the degree while the algorithm is running, especially as we'll probably be calculating it several times for the same vertex. Secondly, we can see that algorithms running on an adjacency list are faster, which is logical since we only iterate on neighbors with fewer conditions. Finally, the difference in performance between NodeIterator and NodeIterator++. We can clearly see the advantage of counting each triangle only once. The gain is even greater with an adjacency list, since we don't have to manage the sorting of vertices by degree.

With the simple example of the ego-facebook graph, we saw that in terms of performance, the implementation with an adjacency list seemed faster, but what would be interesting to know is the memory usage. We saw earlier that adjacency matrices were very resource-hungry, since their size grew as $\Omega(n^2)$. In the case of an adjacency list, memory usage should therefore be much lower, as we can see from the following plot, which compares usage with the two types of implementation for peer-to-peer networks.

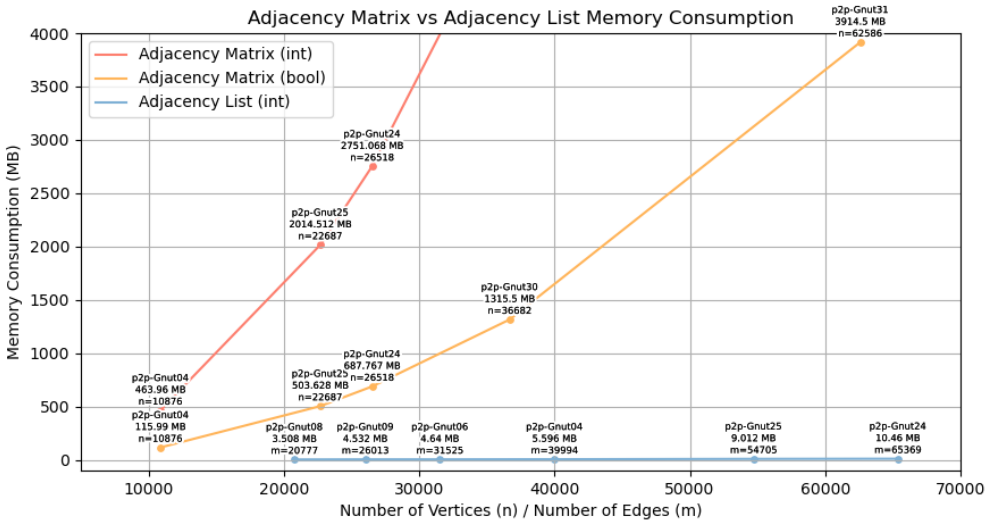


Figure 16: Memory consumption of the different peer-to-peer network graphs (table 4) with an adjacency matrix or an adjacency list

In fact, as we can see, with an adjacency list memory usage increases very little. To be exact, it increases linearly proportional to m and therefore has a space complexity of $\Omega(m)$.

We could already conclude that NodeIterator algorithms implemented with an adjacency list are much faster and more efficient than their counterparts with an adjacency matrix. But it would be a shame to miss out on further analysis by comparing execution times in different scenarios.

Why a further analysis ?

To answer this question, a first analysis we can make is simply to use the first graph set (table 3) to see if anything unusual is happening. Indeed, despite the fact that this first set has graphs ordered by n in ascending order, they have different average clustering coefficients, densities and numbers of triangles, which could have a major impact on the runtime. This is why we also chose a second graph set (table 4) where n and m are increasing but the values of the average clustering coefficient, density and number of triangles are very close to each other and finally we will also do other specific tests.

Here we have two plots on the first graph set (table 3) with the adjacency matrix and adjacency list implementations:

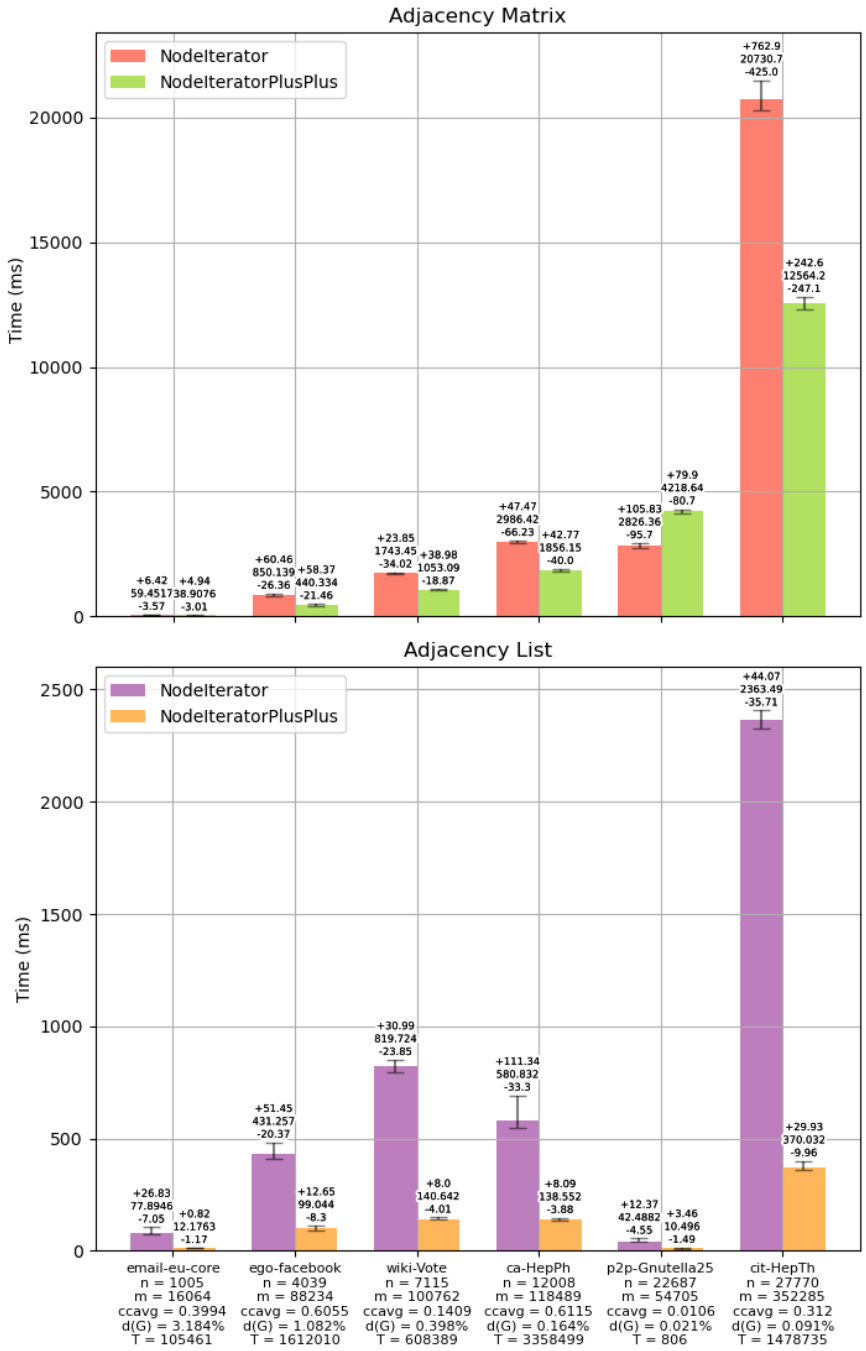


Figure 17: NodeIterators on the first graph set (table 3) (10 runs average)

For the first plot with the implementation based on an adjacency matrix. We saw earlier (see page 47) that this implementation depended much more on n in terms of complexity. We pointed out that NodeIterator required n to be traversed for lines 3 and 4 of the algorithm. For NodeIterator++ it was worse, because the total order had to be calculated, which led to the addition of $O(n^2)$ and $O(n\ log\ n)$ complexity. Indeed, this seems to be the case, except for p2p-Gnutella25 ($n = 22687|m = 54705$), with which NodeIterator seems more efficient than its improved

variant. This means that calculating the total order and counting the number of triangles once with NodeIterator++ takes longer than counting the 806 triangles 6 times with NodeIterator. This can be maybe explained by the low number of triangles and the number of edges relative to the number of vertices in this graph. If we compare it with ca-HepPh ($n = 12008|m = 118489$), the number of vertices has doubled but the number of edges has been cut in half. Thus they have a similar NodeIterator time.

To summarize, in the case of an implementation with an adjacency matrix, NodeIterator is more dependent on n, m. NodeIterator++, on the other hand, is mainly dependent on n in terms of complexity because of its implementation, which requires the total order to be precalculated. While for the implementation with an adjacency list, if we look at the second plot, it's almost impossible to conclude anything because the running times seem to be greatly influenced by something else than n. We can only see that the times are shorter with this type of implementation.

This explains why we need to do further analysis to better understand the differences in behaviour between NodeIterator and NodeIterator++ in particular in the context of an implementation with an adjacency list.

Increasing n and m with fixed parameters

Thus, here is the plots of the second set (table 4) with m on the x axis.

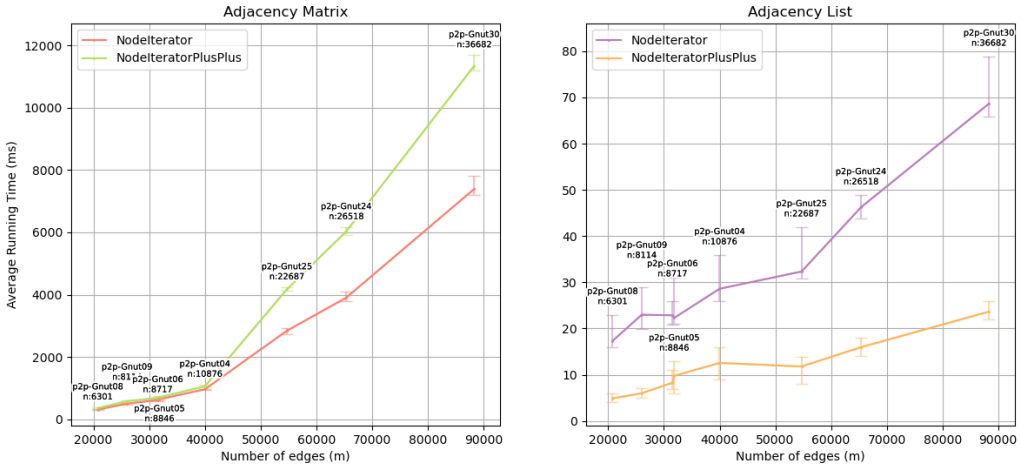


Figure 18: NodeIterators on the second graph set (table 4)

Having a graph set with similar characteristics might allows us to check whether the complexities are respected.

First of all, concerning the plot on the left, NodeIterator takes less time than NodeIterator++. As explained earlier with p2p-Gnutella25, the graphs in this set have few triangles and few edges, which is why NodeIterator++ is penalised by the calculation of the total order, which takes up a large part of the running time. In addition, the exponential aspects of the two curves can be seen, due to the way the implementations have been made.

Then, for the plot on the right, with the implementation using an adjacency list. The results are finally interpretable because all the graphs have similar densities, average clustering coefficients and numbers of triangles. The times are shorter, but above all the curves are more flat, particularly for NodeIterator++. This can be explained by the respective complexities of $O(\sum_{v \in V} d_v^2)$ and $O(m^{3/2})$. But we can see that the delta Δ between NodeIterator and NodeIterator++ stays relatively the same around x3 in practice, while based on the complexities, it should increase:

Graph Name	n	m	$\sum_{v \in V} d_v^2$	$m^{3/2}$	Δexpected	Δreal
p2p-Gnutella08	6301	20777	733620	2994844	x4.08	x3.53
p2p-Gnutella09	8114	26013	874720	4195518	x4.79	x3.83
p2p-Gnutella06	8717	31525	908184	5597352	x6.16	x2.76
p2p-Gnutella05	8846	31839	941810	5681187	x6.03	x2.28
p2p-Gnutella04	10876	39994	1117376	7998200	x7.16	x2.27
p2p-Gnutella25	22687	54705	1175652	12795007	x10.88	x2.74
p2p-Gnutella24	26518	65369	1573150	16713128	x10.624	x2.89
p2p-Gnutella30	36682	88328	2024168	26251105	x12.97	x2.90

Table 5: Expected differences in running time between NodeIterator and NodeIterator++ based on their complexity

What we can say is that since we're dealing with graphs of the same category with nearly fixed parameters (*avgcc* and density), NodeIterator and NodeIterator++ behave in the same way. Thus, the difference in execution time between the two remains approximately the same, that can explain why the results are different from the expected complexities, after all complexites represent the worst case scenarios. We should also not forget that we have fluctuations due to theses small instances and we also have optimisation done with the -O3 flag in the g++ compiler. In the following part, we will see what is causing actually the delta between NodeIterator and NodeIterator++.

The number of tested paths

As said just before, we were working with real graph networks, since they are very different from one another and there are a whole range of graph categories. It's going to be difficult to establish a relationship between the characteristics of the graphs and the complexity of the algorithms in order to explain the obtained running times, even by focusing in a specific category. Another approach would be to count the number of different paths tested by the two algorithms and see if there is a big difference which will redefine our Δexpecteds that we can re-compare with our Δreals and have better conclusions relatives to different graphs.

To do this, simply call up modified versions of NodeIterator and NodeIterator++ and count the number of paths of length two $\{u, v, w\}$ tested just before confirming the existence of a triangle which is done with the $\{u, w\}$ testing. We will therefore note $|nipaths|$ and $|nipppaths|$ for the number of paths tested by the respective algorithms.

Graph Name	$\|nipaths\|$	$\|nipppaths\|$	Δexpected	Δreal
p2p-Gnutella08	7.336×10^5	1.837×10^5	x3.99	x3.53
p2p-Gnutella09	8.747×10^5	2.260×10^5	x3.87	x3.83
p2p-Gnutella06	9.082×10^5	2.839×10^5	x3.19	x2.76
p2p-Gnutella05	9.418×10^5	2.802×10^5	x3.36	x2.28
p2p-Gnutella04	11.17×10^5	3.639×10^5	x3.07	x2.27
p2p-Gnutella25	11.76×10^5	3.658×10^5	x3.21	x2.74
p2p-Gnutella24	15.73×10^5	4.543×10^5	x3.46	x2.89
p2p-Gnutella30	20.24×10^5	6.235×10^5	x3.24	x2.90

Table 6: Expected differences between NodeIterator and NodeIterator++ based on the number of paths tests

Using the same graphs, Δexpected now represents the difference between $|nipaths|$ and $|nipppaths|$. We can thus see that theses are now much closer to the Δreal runtimes between NodeIterator and NodeIterator++. But ideally, we should also test on other graphs with larger and larger Δ to see if we have coherent results and above all if we can draw conclusions. That's why we ran both algorithms on 40 graphs from SNAP (table 10). These are the main graphs available on the platform, for which we have the number of triangles as well as other additional information such as the average clustering coefficient, and of course graphs with less than 10 million edges, so as not to have to deal with graphs that are too big. In the following plot, we have placed Δexpected on the X axis and Δreal on the Y axis in logarithmic scale.

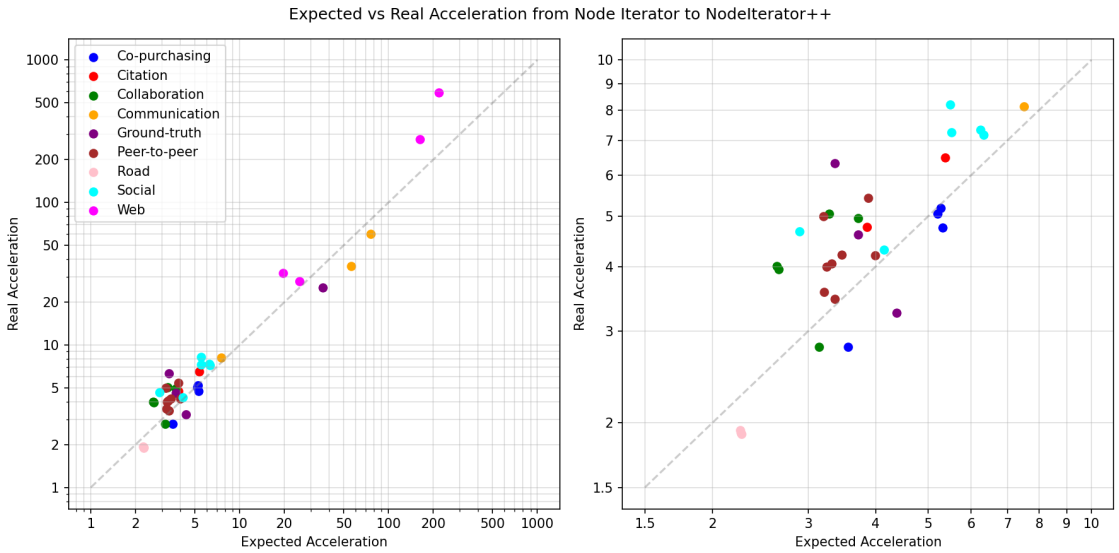
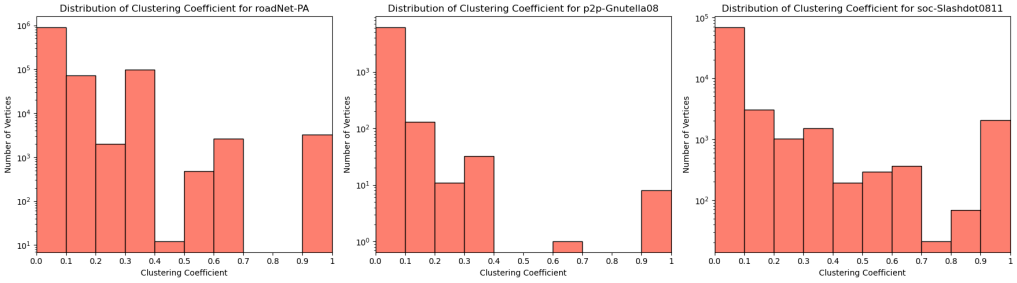


Figure 19: Δreal over Δexpected for 40 graphs from SNAP

By looking at this scatter plot, we can see that the points are approximately close to the diagonal, which clearly means that the ratio between the difference in execution time between NodeIterator and NodeIterator++ and the ratio between the number of paths tested are clearly correlated. Of course, not everything is perfect, the majority of points are slightly above the diagonal, which means that the actual acceleration with NodeIterator++ is better than expected, which may be due to the optimisation flag.

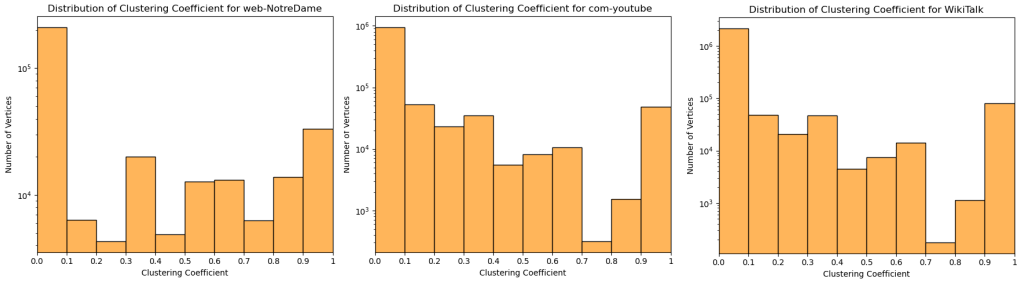
Now we need to examine what distinguishes graphs with acceleration below x10, those above x200 and those in between. In this case, it's the distribution of the clustering coefficient of the different graphs:

- With low acceleration (Less than $\times 10$):



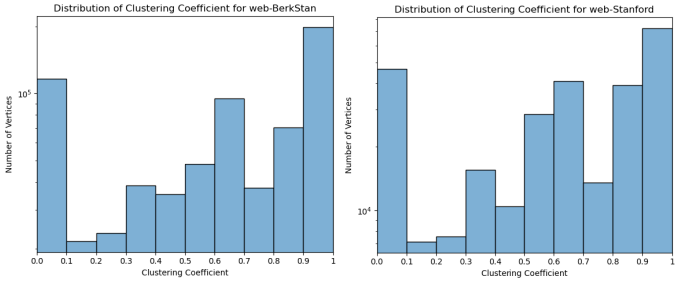
If we look at *roadNet-PA*, *p2p-Gnutella08* and *soc-Slashdot0811*, most of the vertices have a very low clustering coefficient. Actually a majority of them have a clustering coefficient of 0 because they only have 1 neighbor.

- With low-medium acceleration (×10 - ×100):



In this case with *web-NotreDame*, *com-youtube* and *WikiTalk*, we begin to have some vertices with higher clustering coefficient but still with most of them close to 0.

- With high acceleration (×200 or greater):



Finally here, with *web-BerkStan* and *web-Stanford*, the vertices have in majority a high clustering coefficient, actually a lot of them close to 1.

What we can conclude is that the distribution of the clustering coefficent has a major impact on the acceleration provided by NodeIterator++. In fact, in graphs with vertices with low clustering coefficients, NodeIterator won't waste as much time testing paths because the number of possible paths for counting triangles is already quite low. On the other hand, in a graph with high clustering coefficients, the number of possible paths to count the triangles will be much higher and so it will waste a lot of time compared to NodeIterator++, which takes care to count the triangles only once, testing as few paths as possible thanks to the total order of the vertices.

4.3.3 Algorithms Designed For Listing

There are a number of algorithms for listing the triangles in a graph. We can of course adapt NodeIterator++ so that it returns the triangles, but there are other suitable algorithms, such as the Forward algorithm [3].

This algorithm is in fact already an improvement on an algorithm called EdgeIterator [3] where the idea is to iterate over all the edges $\{u, w\}$ and compare their neighbours $\Gamma(u)$ and $\Gamma(w)$ in order to loop the triangles $\{u, v, w\}$. The idea with Forward is to do the same thing except that instead of comparing all the neighbours of u and w, we'll only compare a portion of a size delimited by the total order of u. To do this, an array will be used to store the vertices with a degree smaller than d_u in the set $A(u)$.

Algorithm 6 Forward(V)

Require: V sorted by increasing degrees ($v_i \prec v_{i+1}$)
Require: an empty array for each vertex v: $A(v)$

```
1: for w ∈ V do
2:     for u ∈ Γ(w) do
3:         if w ≺ u then
4:             for v ∈ A(w) ∩ A(u) do
5:                 output triangle {u, v, w}
6:             A(u) ← A(u) ∪ w
```

Thanks to the fact that the vertices are ordered in increasing degree and with the Handshake Lemma, the author [3] explains that the complexity of this algorithm is $O(m^{3/2})$ as NodeIterator++ (Alg:5). In order to do this, he bound the degree (actually the in-degree in a directed graph) of a vertex v by $d_v \in O(\sqrt{m})$ because the running time to compute the intersection $A(w) \cap A(u)$ is $\sum_{\{u,w\}\in E}(d_u + d_w)$.

He base his proof on k being the largest index of a vertex such that $d_k > \sqrt{m}$ but $d_{k+1} \leq \sqrt{m}$. And show the two bounds: for $i \leq k$, $d_i \leq 2\sqrt{m}$ and for $i > k$, $d_i \leq \sqrt{m}$. Which will ends with the following statement:

$$\sum_{\{u,w\}\in E} (d_u + d_w) \leq \sum_{i\leq k} 2\sqrt{m} \cdot 2m + \sum_{i>k} \sqrt{m} \cdot m = O(m^{3/2})$$

The advantage of this algorithm is that it does not modify the base graph, which is not the case with the listing algorithm developed by Chiba and Nishizeki [6], which operates differently.

Algorithm 7 ListingByChibaNishizeki(V)

Require: V sorted by decreasing degrees ($v_i \succ v_{i+1}$)
Require: $false$-initialized array M of size n indexed by V

```
1: for i = 0 to n − 3 do
2:     for u ∈ Γ(v_i) do
3:         M(u) ← true
4:     for u ∈ Γ(v_i) and M(u) = true do                ▷ For finding triangles
5:         for w ∈ Γ(u) and M(w) = true do
6:             output triangle {v_i, u, w}
7:         M(u) ← false
8:     V ← V \ v_i                                      ▷ To avoid duplication
```

This algorithm is different in the sense that it attempts to complete the $\{v, u, w\}$ triangle using a marking system. In concrete terms, we start by choosing a vertex v and marking all its neighbours u, then we look to see if there is a vertex $w \in \Gamma(u)$ which would also be marked, which confirms the existence of a triangle $\{v, u, w\}$. But to avoid counting the triangle several times, we need to remove v from the graph once all the $\{v, u, w\}$ cycles have been found. This algorithm has a complexity of $O(a(G)m)$ [6].

Graph Sample and Data Structures

For this part, we're going to reuse the graphs we discovered and used in the previous section. These are varied graphs with different distributions of clustering coefficients, but still with certain points in common at times. Or vice versa, with a very similar distribution but with other parameters that change. This will allow us to compare which factors are having what effects.

Graph Name	n	m (undirected)	cc_{avg}	$d(G)$	T
roadNet-PA	1088092	1541898	0.0464	0.0003%	67150
p2p-Gnutella08	6301	20777	0.0109	0.1047%	2383
soc-Slashdot0811	77360	469180	0.0555	0.0157%	551724
web-NotreDame	325729	1090108	0.2346	0.0020%	8910005
com-youtube	1134890	2987624	0.0808	0.0005%	3056386
WikiTalk	2394385	4659565	0.0526	0.0001%	9203519
web-BerkStan	685230	6649470	0.5967	0.0028%	64690980
web-Stanford	281903	1992636	0.5976	0.0050%	11329473

Table 7: Graphs selected to compare the listing algorithms

As for the data structures used, for the graphs themselves, we reuse the GraphAdjListVUS class which uses a `vector<unordered_set<int>>` as an adjacency list, that

for the same reasons as the NodeIterators algorithms (see 4.3.2).

On the other hand, for the data structures internal to the algorithms, certain choices had to be made. To begin with, for Forward, for the A arrays, we used a `vector<unordered_set<int>>` in order to get good performance when searching for the intersection of two A arrays. To have V sorted by total order, the function *sort* with a `vector<int>` is used. Regarding the ChibaNishizeki algorithm, the same thing is done to have the same V sorted by total order and for the M array a `vector<bool>` is used. But as this algorithm also has to remove a vertex at each iteration, a copy of the adjacency list is made only at the start and modified as it goes along. This operation has no significant impact as we use an adjacency list that takes up very little memory space and still uses `unordered_set<int>`, which is very efficient in terms of complexity.

Adapted algorithms for more comparisons

At the start of this section, we thought that in order to list triangles in a graph, we could simply adapt NodeIterator++ to do the same thing as the dedicated Forward and ChibaNishizeki algorithms. This might be interesting to compare in terms of performance with those two.

While we're at it, doing the opposite could be just as interesting, meaning adapting Forward and ChibaNishizeki into a triangle-counting algorithm instead of a listing one. To make these adaptations, we simply need to replace the line where we output the triangle by the incrementation of a counter and vice versa to adapt an algorithm for counting the output triangle with the indices of the for loops. This is what we did to adapt NodeIterator++ for the listing:

Algorithm 8 ListingNodeIterator++(V,E)

```
1: for v ∈ V do
2:     for u ∈ Γ(v) and u ≻ v do
3:         for w ∈ Γ(v) and w ≻ u do
4:             if u, w ∈ E then
5:                 output triangle {v, u, w}
```

So, all in all, we end up with the following 6 algorithms to compare, so we'll have to pay close attention to which performs well and which poorly in different cases and with different types of graphs.

Listings:

- NodeIterator++ adapted for listing
- Forward
- ChibaNishizeki

Counting:

- NodeIterator++
- Forward adapted for counting
- ChibaNishizeki adapted for counting

Impact of the clustering coefficient and its distribution on the algorithms and the adapted ones

In order to analyze the 6 algorithms just mentioned, we ran them on the same 8 graphs used at the end of the section on the comparison between NodeIterator and NodeIterator++, with different clustering coefficient distributions. For each graph, we have a bar graph comparing the average running time of the 6 bars, that is, NodeIterator++, Forward, ChibaNishizeki in their counting and listing variants. We can therefore do several observation based on the impact of the average clustering coefficient and the distribution of it.

- Impact of the clustering coefficient distribution:

  To observe the impact of the clustering coefficient distribution, we can look at the general appearance of the plots, basically the order in which the algorithms are ranked in terms of running time. Therefore, we compared the following 5 graphs: *roadNet-PA*, *p2p-Gnutella08*, *soc-Slashdot0811*, *com-youtube* and *WikiTalk*, all of which have a low average clustering coefficient but have different distributions.

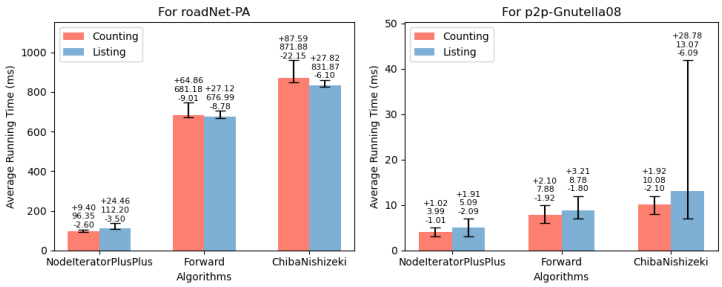


Figure 20: Times for roadNet-PA and p2p-Gnutella08

For *roadNet-PA* and *p2p-Gnutella08* it's firstly the NodeIterator++ algorithms, then Forward, then ChibaNishizeki that follow in terms of increasing

execution time. Whereas for *soc-Slashdot0811*, *com-youtube* and *WikiTalk* the order of execution times from lowest to highest is NodeIterator++, ChibaNishizeki and lastly Forward.

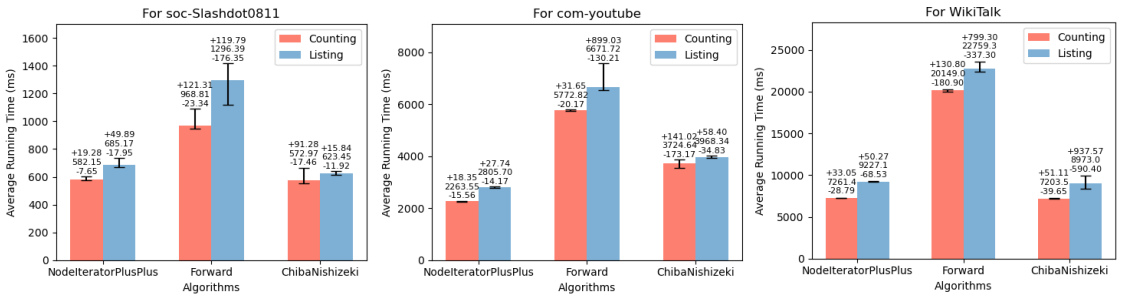


Figure 21: Times for soc-Slashdot0811, com-youtube and WikiTalk

What we can see is that the algorithm that seems to be impacted by the clustering coefficient distribution is ChibaNshizeki. We can explain this behavior by the fact that the last 3 graphs have more vertices with high clustering coefficients, as their distribution suggests. This means there are more interconnected vertices. Since we're dealing first with the higher-degree vertices that are even more interconnected, these vertices will be removed from the graph more quickly, which will speed up the algorithm. Incidentally, we find the same ranking for the 3 remaining graphs, *web-NotreDame*, *web-BerkStan* and *web-Stanford*, at least for the listing.

- Impact of the average clustering coefficient:

If we want to observe the impact of the average clustering coefficient, we can focus on the difference between execution times for counting and listing algorithms. To do this, we can look on the 3 graphs *web-NotreDame*, *web-BerkStan* and *web-Stanford* which have high average clustering coefficients of 0.2346, 0.5967 and 0.5976 respectively.

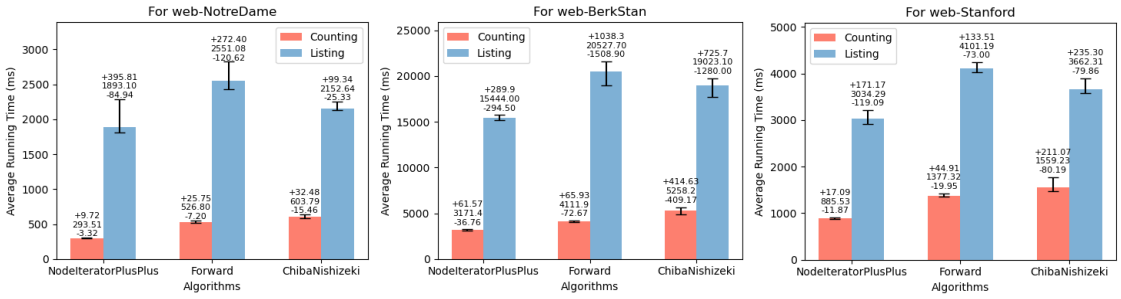


Figure 22: Times for web-NotreDame, web-BerkStan and web-Stanford

What we can see is that, indeed, listing algorithms are 3× to 5× slower than their counting counterparts. Whereas for graphs with a much lower average clustering coefficient, there was virtually no difference, in the worst case listing algorithms are only 30% slower.

To explain this phenomenon, we need to remember that to list our triangles we need to create and store them. This means creating a vector at the beginning to store and saving the triplet in question for each triangle found. What will happen in the inner loop of the algorithms with a high average clustering coefficient is that new triangles will be found more regularly. This means that our vector will be updated as regularly with more frequent write operations to the heap. Not to mention the fact that our vector will have to update its maximum size by doubling it each time it is exceeded. We could say that we don't need to store them and just output the triangles in the standard output, but that doesn't change anything because the standard output also passes through the heap.

Here, it's really the frequency induced by the large average clustering coefficient that causes this difference. We might think that it's perhaps the type of graph, because we're dealing with three web network-based graphs. But no, this is also the case, for example with *ego-facebook*, a social network graph which has an $avg_{cc} = 0.60$ and for the algorithm Forward, counts in around 91.58ms and lists in 460ms. This also amounts to a slowdown of order 5×.

If we were to draw a conclusion about listing algorithms and their performance. We have seen the significant impact that the clustering coefficient of a graph can have, and in particular its distribution across the vertices. Having analysed the Forward algorithm, ChibaNishizeki and a adapted version of NodeIterator++, it turns out that ChibaNishizeki does rather well when the distribution of the clustering coefficient is homogeneous. But when dealing with graphs with a high average clustering coefficient, listing becomes much heavier than counting because of the frequency of the operations, regardless of the number of triangles. Finally, from a general point of view, it's better to use NodeIterator++ to get good and consistent performance, whatever the graph used.

4.4 4-Cycles (C_4)

In the previous section, several triangle-counting algorithms are presented. It turns out that other types of subgraphs can also be counted. In this section, we'll look at algorithms for counting 4-cycles (C_4). The article [1] puts forward 4 theorems on the time and space complexity of algorithms counting 4-cycles in general sparse graphs. Which have the particularity of requiring $O(m\overline{\delta}(G))$ time and $O(n)$ space.

In concrete terms, once again we're going to start talking about a *Matrix Multiplication Based* algorithm. Then we're going to compare 3 *Counting Algorithms*, a general count, a vertex-local and an edge-local. Finally, we'll take a brief look at a *Listing Algorithm* that we won't be able to analyse in depth due to the limitations of our machine. In general, graphs have way more 4-cycles than triangles, which can be a limiting factor in terms of memory.

In each case, as before, we will try to see what influences performance in terms of data structure or graph specificity.

4.4.1 Matrix Multiplication Based

In order to count the number of C_4 in a graph, the first method would again be to use matrix multiplication as with the Algorithm 2. We could simply use the formula $\sum_i (A^p)_{ii}$ with $p = 4$. But this gives us the paths of length 4 starting and ending at the same point, not the number of C_4. To do this, we need to subtract 4 times the paths of length 2 and 2 times the paths of length 1. This gives us the formula $\frac{1}{8}(\sum_i (A^4)_{ii} - 4\sum_i \binom{d_i}{2} - 2m)$ which corresponds to the number of C_4 and can be translated to the following algorithm. The number of 4-cycles in the graph G being stored in $\diamond(G)$.

Algorithm 9 Counting 4-cycles by matrix to power 4

1: $\diamond(G) \leftarrow 0$
2: $chooseSum \leftarrow 0$
3: compute A^4
4: **for** $i = 1$ to n **do**
5: $\quad \diamond(G) \leftarrow \diamond(G) + A_{ii}$
6: $\quad chooseSum \leftarrow chooseSum + \binom{d_i}{2}$
7: $\diamond(G) \leftarrow \diamond(G) - 4chooseSum - 2m$
8: **return** $\diamond(G)/8$

For the algorithmic complexity, as explained in the chapter about matrix multiplication (see 3.3) and seen with the triangle counting version, in line 3, A^4 is calculated

in $O(n^\omega)$. The lines from 4 to 6 are calculated in $O(n)$ unless the degree d_i has to be calculated by traversing n, as is the case with an adjacency matrix implementation, so it's more likely to be $O(n^2)$. The final complexity is also $O(n^2) + O(n^\omega)$, and since the upper bound is the matrix multiplication (because $\omega < 2.371866$ [18]), the complexity is actually $O(n^\omega)$.

Using BLAS SSYMM Matrix Multiplication

For this algorithm, we will not go into too much detail. Using the graph sample with n increasing (table 3). We can expect that the results will be almost identical to the algorithm (Alg:2) which consisted in finding the number of triangles by calculating A^3. Indeed, since the main operation is to calculate the multiplied adjacency matrix and in both cases, the number of matrix multiplications to be performed is identical. In other words, for A^3 and A^4 we go through the calculation of A^2.

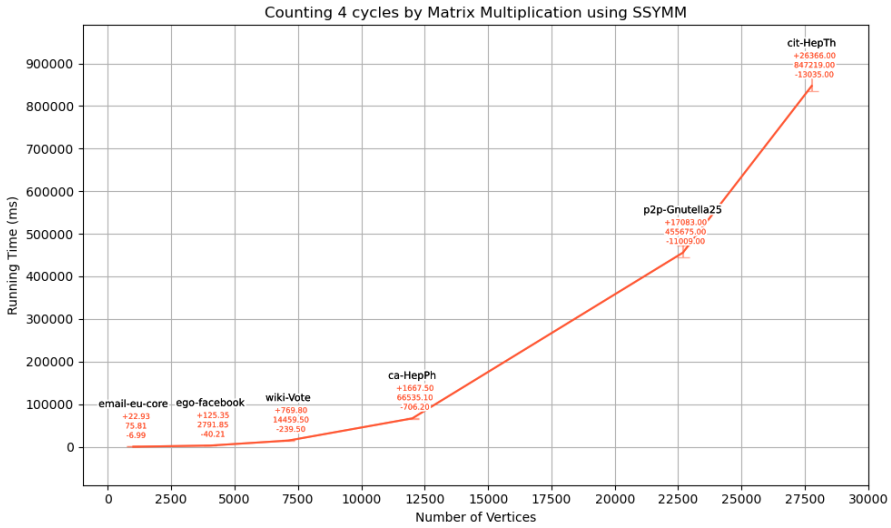


Figure 23: Counting 4-cycles with matrix multiplication with n increasing

In fact, as you can see from this plot, the times are almost the same as if you were counting the number of triangles with the cube matrix. Actually, this algorithm is the same as applying the generalisation we will see later (see 4.5). We'll take this opportunity to make a few more comparisons.

4.4.2 Counting Algorithms

The idea behind this algorithm (Alg:10) is similar to that used for triangles with NodeIterator++ (Alg:5), where you designate a vertex in order to count the triangle just once. In fact, we're dealing with the same problem here, except that if we were

to use a naive algorithm, we'd count the 4-cycles 8 times, with a starting point on each vertex and in each direction. Which will be even worse if we do it naively for 4-cycles. But the difference here, is that we will use a L list to store the occurrence of the "neighbors of neighbors", that way if for a specific vertex, there is 2 paths of length 2, this will be considered as a 4-cycle. This avoid the usage of a "if" statement and the checking of the existence of a specific edge like in NodeIterators algorithms, which can be advantageous with the correct data structure.

Algorithm 10 General Counting 4-cycles

Require: zero-initialized array L of size n indexed by V

1: **for** $v \in V$ **do**
2: **for** $u \in \Gamma(v) : u \prec v$ **do**
3: **for** $y \in \Gamma(u) : y \prec v$ **do**
4: set $\diamond(G) := \diamond(G) + L(y)$ ▷ Count of 4-cycles
5: set $L(y) := L(y) + 1$
6: **for** $u \in \Gamma(v) : u \prec v$ **do**
7: **for** $y \in \Gamma(u) : y \prec v$ **do**
8: set $L(y) := 0$ ▷ Reset $L(y)$ for the next iteration

A very detailed explanation is already provided by the author [1]. But if we had to summarize, in a concrete example where we have a unique C_4 with $1 \prec 2 \prec 3 \prec 4$. Here is an execution table which trace the evolution of the different variables while the algorithm is running,

	Lines	v	u	y	L	$\diamond(G)$
Vertex 1 1.	1-3	1	/	/	{0,0,0,0}	0
2.	6-7	1	/	/	{0,0,0,0}	0
Vertex 2 1.	1-3	2	1	/	{0,0,0,0}	0
2.	6-7	2	1	/	{0,0,0,0}	0
Vertex 3 1.	1-3	3	2	1	{0,0,0,0}	0
2.	4-5	3	2	1	{1,0,0,0}	0
3.	6-7	3	2	1	{1,0,0,0}	0
4.	8	3	2	1	{0,0,0,0}	0
Vertex 4 1.	1-3	4	1	2	{0,0,0,0}	0
2.	4-5	4	1	2	{0,1,0,0}	0
3.	2-3	4	3	2	{0,1,0,0}	0
4.	4-5	4	3	2	{0,2,0,0}	1
5.	6-7	4	1	2	{0,2,0,0}	1
6.	8	4	1	2	{0,0,0,0}	1
7.	6-7	4	3	2	{0,0,0,0}	1
8.	8	4	3	2	{0,0,0,0}	1

Table 8: Execution Table for Algorithm 10 on C_4 with $1 \prec 2 \prec 3 \prec 4$

The execution can vary depending on which 4-cycle is taken from (4, 2, 1, 3), (4, 1, 2, 3) or (4, 1, 3, 2). But the reason why the algorithm works is that it will try to find a responsible vertex with the highest "local" order, in this case *Vertex 4*. And try to "trace" two distinct paths of length 2 to a vertex of lower order, in this case *Vertex 2*, passing for each path also through vertices of lower order from the original one. We can clearly see the $L(2)$ increasing when the path pass though *Vertex 1*, line 2 of the table and the same for *Vertex 3*, line 4 of the table. The second nested loop lines 6-8 clearly allows to reset the tries for the past candidate vertices. This algorithm has a time complexity of $O(m\overline{\delta}(G))$ and space $O(n)$, since this is the size of our list L.

Sometimes, we not only need the total number of 4-cycles, but we also need the number of 4-cycles to which a vertex v belongs, noted $\diamond(v)$. Once we know how to do this, it's very easy to find the total number of 4-cycles in graph G, since the sum of $\diamond(v)$ will be equal to four times the number of 4-cycles in G (because each cycle will be counted four times). The same principle applies to edges if we have an algorithm capable of calculating $\diamond(uv)$. This is particularly useful when we want to divide the task in order to count the total number of cycles in parallel or we want to get the information for a specific group of vertices.

Algorithm 11 Vertex-local 4-cycle counting

Require: zero-initialized arrays L, L', and $\diamond$ of size n indexed by V

```
1:  for v ∈ V do
2:      for u ∈ Γ(v) : u ≺ v do
3:          for y ∈ Γ(u) : y ≺ v do
4:              set ◇(v) := ◇(v) + L'(y)
5:              set ◇(y) := ◇(y) + L'(y)
6:              set L(y) := L'(y)
7:              set L'(y) := L'(y) + 1
8:      for u ∈ Γ(v) : u ≺ v do
9:          for y ∈ Γ(u) : y ≺ v do
10:             set ◇(u) := ◇(u) + L(y)
11:             set L'(y) := 0
```

In any case, the author of the article [1] presents this algorithm (Alg:11) still based on the previous (Alg:10), which increments the 4-cycle counters of vertex v and y at the same time on lines 4 and 5. What's more, it uses two lists instead of one, List L and List L'. The L' replace the role of L from the first algorithm, but we still need a copy which is actually L and will be useful for calculating $\diamond(u)$ at line 8, while still being able to reset the L' list like L previously.

Since this algorithm is based on the first, the time complexity of $O(m\overline{\delta}(G))$ remains the same, while the space complexity is always proportional to the number of vertices n, since L, L' and $\diamond$ are of this size.

For the variant for the edge local counting, we use a M and a T array. In concrete terms, the array M will act as an edge list, and in order to know where all edges belonging to a vertex v in the direction $\{v, u\}$ are stored, the array T will act as an index. So M will be of size $2m$. Here's an example with a graph G showing how the number of 4-cycles incident on each edge will be stored,

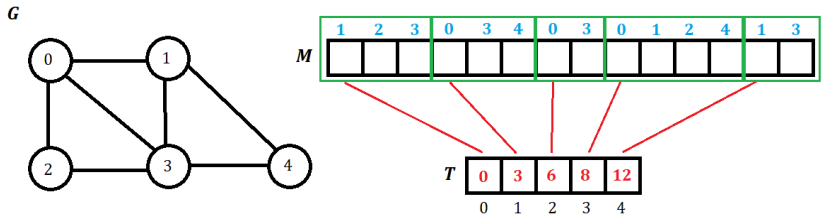


Figure 24: Example of how the array M will be for the graph G

Algorithm 12 Edge-local 4-cycle counting

Require: zero-initialized array L, L' of size n indexed by V
Require: zero-initialized array $\diamond$ of size m
Require: zero-initialized array M of size $2m$
Require: array T of size n defined by $T(v) = \sum_{v'<v} d_{v'}$

for $v \in V$ **do**
  for $u \in \Gamma(v) : u \prec v$ **do**
    for $y \in \Gamma(u) : y \prec v$ **do**
      Set $L(y) := L'(y)$
      Set $L(y) := L(y) + 1$
  for $i = 0$ to $d_v - 1$ **do**
    Set u as the ith neighbor of v
    if $u \prec v$ **then**
      for $j = 0$ to $d_u - 1$ **do**
        Set y as the jth neighbor of u
        if $y \prec v$ **then**
          Set $M(T(v) + i) := M(T(v) + i) + L(y)$
          Set $M(T(u) + j) := M(T(u) + j) + L(y)$
          Set $L'(y) := 0$
for $v \in V$ **do**
  for $i = 0$ to $d_x - 1$ **do**
    Set u as the ith neighbor of v
    if $u \prec v$ **then**
      Find the index j of v in $\Gamma(u)$
      Set $\diamond(vu) := M(T(v) + i) + M(T(u) + j)$

This algorithm works like the vertex-local, but the direct counting is removed from the first nested loop, and only the updating of L and L' is done. It's in the second nested loop (line 6-14) that the M array is updated. Counting for the v, u and u, y edges is done at the same time, somewhat as before. Access to the counters is quite intuitive, as shown in Figure 24. At the end of the algorithm, to obtain the value of $\diamond(vu)$, the counters of the respective vertices are summed. The complexity remains at $O(m\overline{\delta}(G))$ and for the space complexity is $O(n+m)$ due to the array M.

Graph Sample and Data Structure

As far as the graphs are concerned, we're going to reuse the graphs we've seen in previous sections. In this case the set of peer-to-peer networks (table 4), which have the advantage of having fixed parameters with only n and m increasing. But also the sample (table 7) used to do the clustering coefficient analysis.

Concerning the data structure used, for the General counting algorithm and the Local-Vertex algorithm we can firstly use the same adjacency list from the class GraphAdjListVUS based on a `vector<unordered_set<int>>`. But for the local edge counting algorithm, since it uses the vertex indexes of the adjacency list, it is impossible to use any set as a data structure for the graph (because sets are not index based). To do this, we simply use a graph based on a `vector<vector<int>>` as the adjacency list in the class called GraphAdjListVV. Thus, to do a fair comparison between the three algorithms, we will also use this second implementation for the General Counting and Local-Vertex algorithm.

Indeed, comparatively to NodeIterator++, the 4-cycles algorithms don't need to check the existence of an edge. So, only the following three requirements needs to be satisfied.

1. The ability to iterate over all the vertices $V(G)$.
2. Iterate over the neighbours of a vertex $\Gamma(v)$.
3. The ability to quickly retrieve the degree of a vertex d_v.

With only theses three requirement, the choice of a `vector<vector<int>>` as the adjacency list can be better for long iterations over a `vector<unordered_set<int>>` as explained previously (see 4.1.3). Since the hash list of the unordered_set was useful only for the access by value for the edge existence check.

For the structures in the algorithms themselves, the arrays L, L', M, T use simply a `vector<int>`. But for the returned array $\diamond$, for the general algorithm, we use a integer counter, for the vertex local a `vector<int>` and for the edge local a `vector<vector<int>>` where the array is structured as follow :
$\{\{v, u, VUcounter\}, \{x, y, XYCounter\}, ...\}$

Unordered set vs Vector

Before going any further, it's a good idea to choose the ideal data structure for the graphs used in these algorithms. This is to highlight the fact that using a `vector<vector<int>>` could improve performance, since no access by value is required. Concretely, we have 3 algorithms, the first two of which can be implemented in the two data structures. We therefore end up with five possible executions which we have carried out on the ego-facebook graph as follows.

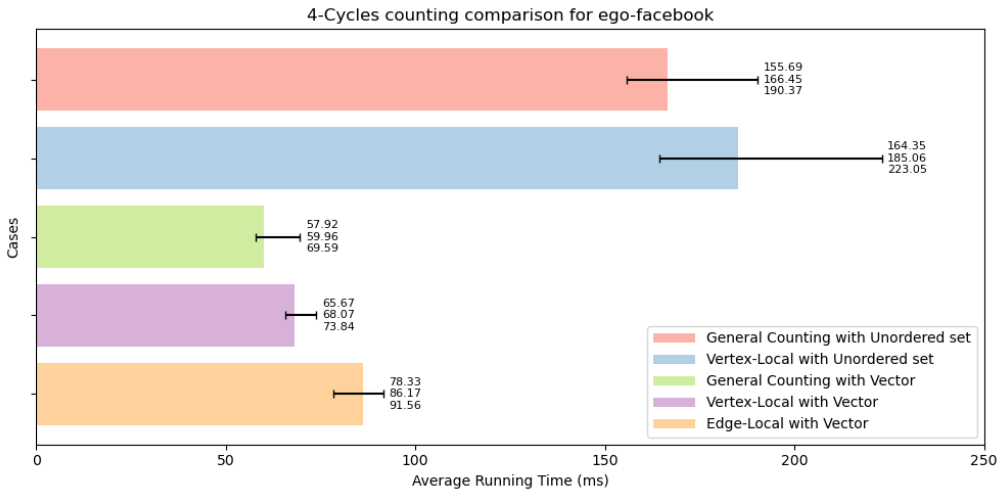


Figure 25: Comparison of 4-cycles counting algorithms on ego-facebook with graph based on std::unordered_set and std::vector (10 runs average)

We can see two things, the first is that if we don't need to access by value as with our 4-cycle counting algorithms. An adjacency list based on a `vector<vector<int>>` is preferable. In fact, the times are much shorter than those based on an unordered_set. The second is that General counting is the fastest, followed by Vertex-Local counting and finally Edge-Local counting. Which makes sense, but it would be better to do a more detailed analysis with other graphs to be sure.

We also compared the two data structures on other graphs and we noticed that using a graph based on an adjacency list with vector is always more efficient. We

will therefore focus on this data structure for the future comparisons.

Increasing n and m with fixed parameters

Since we're dealing with 3 very similar algorithms with the same basic structure, we might think that they'll behave in the same way whatever the graph. But the difference between the General algorithm, the Vertex-Local algorithm and the Edge-local Algorithm is that, from the most basic to the most complex, they use more and more arrays. Even if the space complexity remains similar, except for Edge-Local which has a complexity of $O(n + m)$ instead of $O(n)$, the execution time curves should remain the same since all three algorithms have a complexity of $O(m\overline{\delta}(G))$.

That's why, to find out for sure, we ran the 3 algorithms on peer-to-peer networks which, as a reminder, are graphs with very similar characteristics but increasing size. And we end up with the following plot.

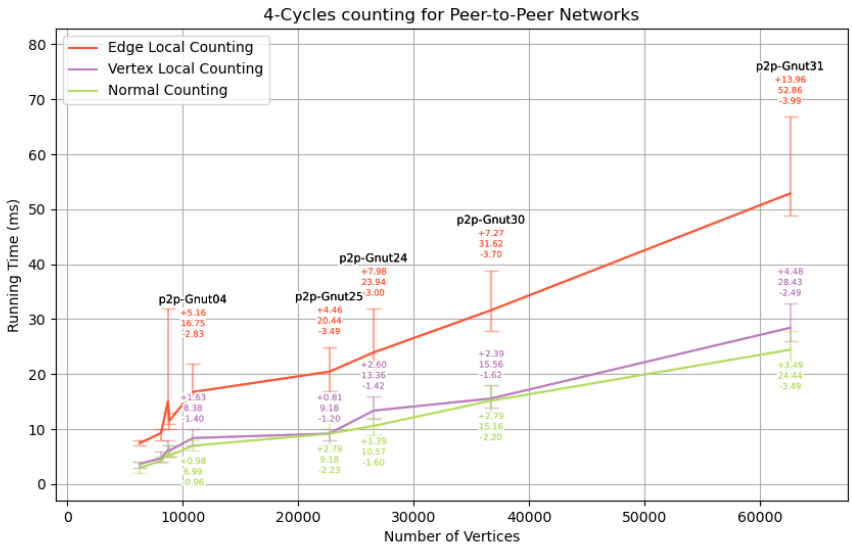


Figure 26: Comparison of 4-cycles counting algorithms on peer-to-peer networks (10 runs average)

Looking at this plot, we can see that the execution times for the General algorithm and Vertex-Local are very close. Vertex-Local is a little slower, but this can be explained by the use of different arrays L and L'. The overall pattern remains the same. However, for Edge-Local Counting, the execution time is around twice as long, so the difference is constantly increasing (linearly). This can be explained by

the fact that an array T of size n has to be defined. However, this can be defined in one pass because $T(v) = T(v-1) + d_{v-1}$, so avoiding $O(n^2)$ complexity.

In the next plot, we also measured memory usage to see if it was consistent across the three algorithms, both in terms of the different arrays used and the space complexity.

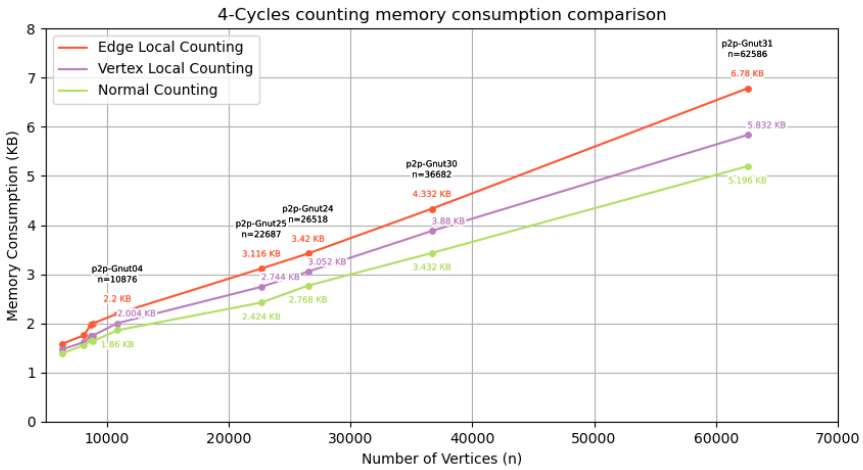


Figure 27: Memory consumption comparison of 4-cycles counting algorithms on peer-to-peer networks (10 runs average)

What we can see is that we end up with the same order. In fact, it's logical that the General algorithm should be the least greedy in terms of memory, since it only uses an array L of size n. Then comes Vertex-Local with L and L' and finally Edge-Local with L, L', M and T.

We can see that the space complexity is $O(n)$ but for Edge-Local it is difficult to see that the complexity is $O(n+m)$ because we are dealing with graphs of the same density where n and m increase at the same rate. However, if we compare two graphs with different m, p2p-Gnutella24($n = 26518$, $m = 65396$) and Cit-HepTh($n = 27770$, $m = 352285$), the memory usage is very different, respectively 3,420KB and 9,380KB for Edge-Local.

Impact of the clustering coefficient distribution

If we do the same kind of analysis as we did with the Triangle algorithms on the impact of the clustering coefficient distribution on the 4-cycle counting algorithms. We notice that there are indeed similarities in the performance differences

between the algorithms for each type of distribution. But overall the ranking of the algorithms will remain the same since the 3 algorithms General, Vertex-Local and Edge-Local are based on the same nested loop. The General algorithm will always be the fastest.

- For graphs whose distribution contains vertices with a small clustering coefficient, as with *roadNet-PA* and *p2p-Gnutella08*. The pattern is similar, the algorithm General and Vertex-Local have very similar performances, then comes Edge-Local with an execution time more than twice as slow. This can be explained by the fact that we are dealing with a lot of weakly interconnected vertices and so the L and L' arrays (in the case of Vertex-Local) will not be updated very much. On the other hand, Edge-Local always has to initialise T, the time of which depends solely on the number of vertices, since we calculate the degree of a vertex based on the size of its adjacency list.

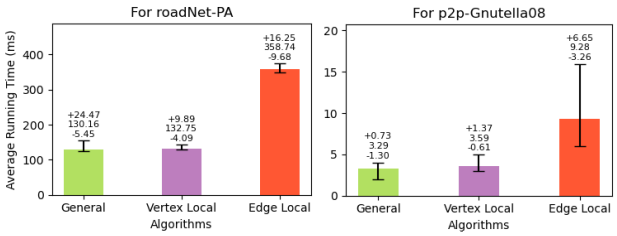


Figure 28: Times for roadNet-PA and p2p-Gnutella08

- Regarding graphs with a balanced distribution such as *com-youtube*, *soc-Slashdot0811* and *WikiTalk*. As we start to get vertices with higher clustering coefficients, the General and Vertex-Local algorithms seem to slow down, but this is even more the case for Vertex-Local. This makes sense because Vertex-Local has to update two arrays, L and L', whereas General just updates L.

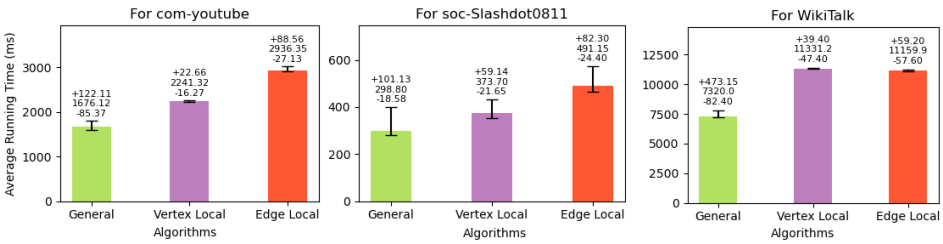


Figure 29: Times for soc-Slashdot0811, com-youtube and WikiTalk

- Finally with graphs having a distribution with high coefficient clustering vertices such as *web-NotreDame*, *web-BerkStan* and *web-Stanford*. The difference

between General and Vertex-Local is still visible but Edge-Local seems to have been greatly impacted by the presence of vertices with high clustering coefficients. This can be explained by the M array which has to be updated even more in the nested loop from line 6 to 14. But also by the recording of cycles when M is read on line 20. If we have a lot of vertices with a high clustering coefficient, this means that we have a lot of edges and therefore we will have to update as much at the end.

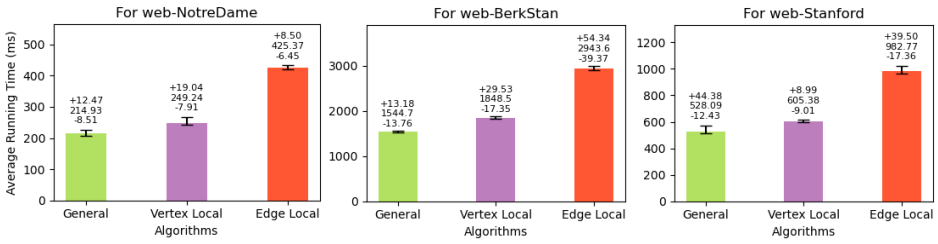


Figure 30: Times for web-NotreDame, web-BerkStan and web-Stanford

In a few words, when it comes to overall performance and memory usage, we're dealing with the General, Vertex-Local and Edge-local algorithms, which share the same principle. In terms of time and memory complexity, we end up with almost identical behaviour. However, the initialisation and the use of additional arrays can have a linear impact on execution time. As with what we saw in the subsections *NodeIterator Vs NodeIterator++* and *Algorithms Designed For Listing* on the triangle algorithms analysis, the distribution of the clustering coefficient can have an impact on performance. In our case, this influences execution time over that linear impact of using extra arrays.

4.4.3 Listing Algorithm

If we want to enumerate the 4-cycles, The following algorithm is a modified version of (Alg:10). The time complexity increases to $O(m\bar{\delta}(G) + \#C_4) <= O(m^{3/2} + \#C_4)$ [1] and the space complexity remains at $O(n)$. This is due to the use of a list for each vertex instead of a simple counter, and especially to the addition of an extra loop on line 4-5 to return each 4-cycle.

Algorithm 13 4-cycles listing

Require: list $L(v)$ for each vertex $v \in V$, all initialized to be empty

```
1: for v ∈ V do
2:     for u ∈ Γ(v) : u ≺ v do
3:         for y ∈ Γ(u) : y ≺ v do
4:             for each vertex x ∈ L(y) do
5:                 output (v, u, y, x)
6:             add u to the end of list L(y)
7:     for u ∈ Γ(v) : u ≺ v do
8:         for y ∈ Γ(u) do
9:             set L(y) := ∅
```

Graph Sample and Data Structure

For the graphs used, we will unfortunately not be able to analyse graphs with a distribution containing vertices with a very high coefficient of clustering. For example with *web-Standford* and *web-BerkStan* because these graphs contain 13 and 127 billion 4-cycles respectively. If we were to store all these 4-cycles in memory, it would be equivalent to storing 208 and 2032GB respectively, which is just inconceivable. And even if, instead of storing them, we just displayed them or stored them in a (very big) file, we'd still have the same problem with the $L(v)$ lists. The only solutions would be either to use a supercomputer, or to adapt the algorithm to be able to divide the task into smaller instances so that it could run on a distributed system, for example.

However, that's not going to stop us from drawing conclusions about the impact of the distribution and the average clustering coefficient. There is *soc-Slashdot0811* which has a relatively balanced distribution with only 50 million 4 cycles and *ego-Facebook* which has a high average clustering coefficient. And we can also draw some conclusion with the peer-to-peer networks.

Finally for the data structures used, it's the same adjacency list of `vector<vector<int>`. Plus for the lists in the algorithm and for the output 4-cycles list, it will simply be a 2D vector of integer.

Increasing number of 4-cycles with fixed parameters

In order to verify the time complexity of $O(m\overline{\delta}(G) + \#C_4)$. One solution is to compare similar graphs with an increasing number of 4-cycles, between the counting and listing algorithms. To do this, we can use the peer-to-peer networks again, but

we'll sort them in ascending order of number of 4-cycles. If all goes well, an increasing gap between the counting and listing algorithms should become noticeable. So, we end up with the following plot.

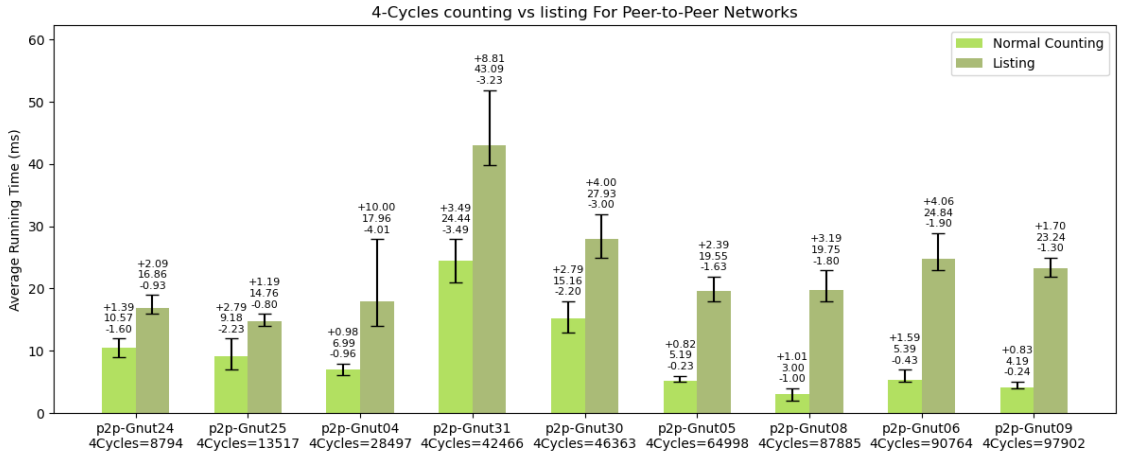


Figure 31: Comparison of 4-cycles General counting and listing algorithm on peer-to-peer networks

Indeed, this is what we can see on the plot. For the graph *p2p-Gnutella24* with only 8794 4-cycles, the listing algorithm is only 1.6× slower than the counting algorithm. Whereas for *p2p-Gnutella24* which has 97902 4-cycles, the listing algorithm is 5.5× slower.

Impact of the average clustering coefficient and its distribution

Finally, as regards the impact of the clustering coefficient and its distribution. As mentioned previously, we were unable to perform a listing test on certain graphs due to memory limitations. But with *com-DBLP* and *ego-Facebook*, which are graphs with a high average clustering coefficient and a generous distribution, we were able to run the algorithm because they are not so large and do not have so many 4-cycles.

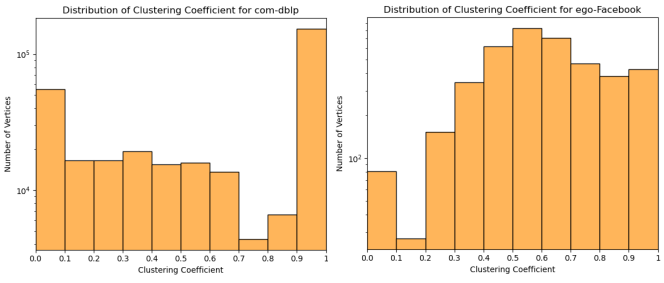


Figure 32: com-dblp and ego-facebook clustering coefficient distribution

Firstly, if we look at the following table, between *roadNet-PA* and *p2p-Gnutella08*, the first one is much larger than the second one and has more 4-cycles. The second graph is much more affected by listing, which is due to the fact that the road network graph has a large number of vertices with a clustering coefficient of zero or close to zero.

Then, on the other hand, for *soc-Slashdot0811*, despite its average clustering coefficient being similar to the two previous ones, its distribution and its number of 4-cycles means that we end up with a running time for listing of $\times 32.03$ instead of $\times 1.99$ and $\times 7.30$. If we were to compare it to a graph with a similar number of 4-cycles. There is *com-DBLP* which has a distribution with a fairly similar profile but with more vertices with a clustering coefficient close to 1. This explains its high average clustering coefficient. We end up with a slowdown increase to $\times 43.76$.

Finally, with *ego-Facebook*, which is a much smaller graph than *com-DBLP*, has almost 3 times as many 4-cycles and has a distribution with a lot of high clustering coefficient verticies but still with similar high average clustering coefficient. We end up with a slowdown from the counting algorithm to the listing algorithm of $\times 504.93$.

Graph Name	n	m	cc_{avg}	$\diamond(G)$	Counting	Listing	Δ
roadNet-PA	1088092	1541898	0.0464	157802	130.155 ms	259.11 ms	$\times 1.99$
p2p-Gnutella08	6301	20777	0.0109	87885	3.29192 ms	24.037 ms	$\times 7.30$
soc-Slashdot0811	77360	469180	0.0555	49965153	298.801 ms	9570.8 ms	$\times 32.03$
com-DBLP	317080	1049866	0.6324	55107655	233.36 ms	10212 ms	$\times 43.76$
ego-Facebook	4039	88234	0.6055	144023053	59.9568 ms	30274 ms	$\times 504.93$

Table 9: Time difference between counting and listing for graphs ordered by increasing Δ between counting and listing

To conclude, we have seen that listing 4-cycles can be more or less costly. The more there are, the longer it will take to count them. For sparse graphs, this is not such a big problem, but for graphs with a high average clustering coefficient and/or with an unfavourable clustering coefficient distribution (a relatively dense graph), listing can become a much more time-consuming task. Not to mention the fact that graphs generally have many more 4-cycles than triangles, which makes things complicated in terms of memory.

4.5 Generalization For k-Cycles (C_k)

In order to generalise a way of knowing the number of k-cycles C_k in a graph G. We first need to count the number of *simple closed path* of length k because if we divide it by $2k$, for each direction and for k starting points, to obtain the number of C_k in G. The number of simple closed paths (for a particular k) actually corresponds to :

$$|\textit{simple closed paths}| = |\textit{total closed paths}| - |\textit{non-simple closed path}|$$

We can thus find the number of C_k in a graph G, which can be written $|G(C_k)|$ based on the following syntax $|G(H)|$. This is done as follows:

$$|G(C_k)| = \frac{1}{2k} \times (|\textit{total closed paths}| - |\textit{non-simple closed paths}|)$$

This is why we'll explain how to get the number of *total closed paths* and the *non-simple closed paths* of length k.

4.5.1 Total Closed Paths Of Length k

There are two different ways of counting the total number of closed paths of length k. The first is to use the adjacency matrix A of our graph G. If we set our matrix to the power of k to obtain A^k and sum its diagonal by doing $\sum_i (A^k)_{ii}$, we obtain the trace $tr(A^k)$ which corresponds to the total number of closed paths of length k. For the second way, let's imagine that we have a k-cyclic graph H which is a homomorphic image of a cycle C_k (see 2.3), we can define the different number of homomorphisms by $C_k(H)$. And if we also define the number of subgraphs of a graph G isomorphic to H by $|G(H)|$. The total number of paths of length k in G can be defined as follows:

$$|\textit{total closed paths}| = tr(A^k) = \sum_H C_k(H) \times |G(H)|$$

4.5.2 Non-simple Closed Paths Of Length k

In order to find the number of non-simple closed paths of length k, we need to know which are the different homomorphisms of a cycle C_k in order to know which are non-simple closed paths so that we can count them and remove them from our final formula.

Unfortunately, there is no miracle solution for this, it's a case-by-case affair. For example, when $k = 3$, the only possible homomorphism is C_3 itself, which is a simple closed path. This is the only case where $|\textit{simple closed paths}| = |\textit{total closed paths}|$, which we multiply by $\frac{1}{2\times 3}$ to obtain $|G(C_3)|$, hence the algorithm (Alg:2). In the case where $k > 3$, i.e. for example with C_4, C_5 or even more, there are indeed homomorphisms in *non-simple closed paths* form.

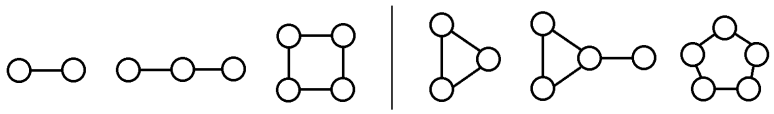


Figure 33: Graphs homomorphic to C_4 on the left and to C_5 on the right

This explains why we have removed 2 times the paths of length 1 and 4 times the paths of length 2 in the C_4 counting algorithm based on matrix multiplication (Alg:9). The same can be applied to C_5. For C_6 the number of homomorphisms rises to 10 and even 12 for C_7, as shown in the article by Alon, Yuster and Zwick [4].

$$|\textit{non-simple closed path}| = \sum_{V(H)<k} C_k(H) \times |G(H)|$$

In concrete terms, the non-simple closed paths for a k-cyclic graph correspond to homomorphisms where $V(H) < k$. And of course, we need to consider $C_k(H)$ for each homomorphismes.

4.5.3 Final Formula And Its Complexity

If we put all the pieces of the puzzle together, we end up with the following formula for counting C_k for $k \geq 3$. This shows and confirms the generalisation.

$$|G(C_k)| = \frac{1}{2k}\left(\operatorname{tr}(A^k) - \sum_{V(H)<k} C_k(H) \times |G(H)|\right)$$

As for complexity, for $3 \leq k \leq 7$, it is $O(n^\omega)$, in fact for all these cases, the author [4] shows how to obtain the different homomorphisms (15 in total). Each time, in the worst case, we can use the adjacency matrix or one of its multiples produce a formula to count the occurrences of these homomorphisms in a graph. The problem starts at $k = 8$, since in order to count the number of octagons in a graph, we need

to count the number of complete graphs K_4, as this is a homomorphism of C_8. And we don't know how to do this in $O(n^\omega)$.

4.5.4 Comparisons Between C_3, C_4 and C_5

As for C_3 and C_4, we've already seen how to implement them in the sections dedicated to matrix multiplication, in this case the (Alg:2) and (Algo:9) algorithms.

Regarding C_5, it has two other homomorphisms, C_3 and a C_3 with an extra edge. The first is simply the number of triangles and the second corresponds to the formula $\frac{1}{2}\sum_i (A^3)_{ii}(d_i - 2)$. If we apply the final formula and remove these two homomorphisms from the total number of closed paths of length 5, we will have the number C_5 in a graph.

In terms of implementation, we can get theses two homomorphisms either by using the matrix multiplication or by using NodeIterator (Alg:4). Basically we can either compute A^5 and A^3 by computing A^4 and A^2 and so doing 4 matrix multiplications, or we can avoid computing A^3 and instead using NodeIterator and only do a total of 3 matrix multiplications to get A^5 and count the number of C_5. The choice is already done, using NodeIterator is faster than doing one extra matrix multiplication since NodeIterator is already faster than the algorithm to count triangles by matrix squaring (Alg:3) as we saw in this thesis. For example, to count the number of C_5 for *ego-Facebook*, it takes around `5800 ms` with 4 matrix multiplication and only around `4800 ms` with 3 matrix multiplication + NodeIterator, on our machine using BLAS SSYMM function.

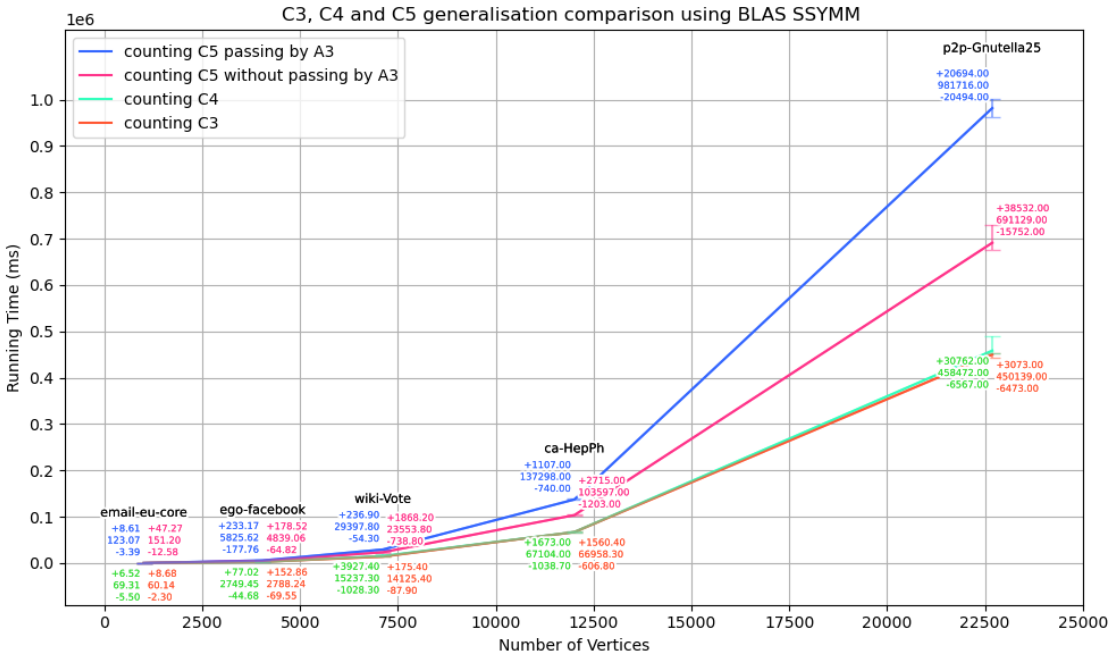


Figure 34: Comparison of counting C_3, C_4 and C_5 using BLAS SSYMM function based on the generalisation

By looking at the plot, we can clearly see what we expected. The complexity is indeed related to the computation of the matrix multiplications and thus to $O(n^\omega)$.

For C_3 and C_4, since we are doing 2 matrix multiplications, in both cases the results are very close. However, C_4 seems to be slightly slower most of the times, which makes sens since we need to remove the two homomorphism of C_4 which are non-simple.

Comparatively to C_5, this one is two time slower than C_3 and C_4. This is logical because we do 4 matrix multiplication instead of 2. What is more interesting, is about counting C_5 using NodeIterator for counting the $C3$ homomorphims of C_5, which avoid the usage of A^3. Clearly this decrease the number of multiplication to 3 and improve the running time, the only exception is for the very small graphs like *email-eu-core*, indeed her the usage of NodeIterator seems to be slower than the extra multiplication normally required for counting C_5.

4.6 Summary Of Our Experimentations

Knowing the fact that the presentation and implementation of algorithms for counting triangles and 4-cycles is our main objective in this thesis. We were able to experiment with this in several stages, starting with *Programming Language Choice* and *Experimental Setup And Methodology*. In which we presented our choice for the C language, the different technologies and libraries and the approach we took.

Initially, we focused on algorithms for counting *Triangles (C_3/K_3)*, starting with *Matrix Multiplication Based* algorithms, which are clearly the least efficient, although huge improvements can be made using BLAS. Then we compared two algorithms, *NodeIterator Vs NodeIterator++*, with different data structures and different graphs. After a bit of research, we realised the impact that the clustering coefficient and its distribution could have on performance. The same applies to *Algorithms Designed For Listing*, whose influence can varies from one case to another.

Then we did more or less the same thing for the *4-Cycles (C_4)*. We looked to see if there was an *Matrix Multiplication Based* algorithm. Then we compared 3 *Counting Algorithms* and once again realised the importance of the clustering coefficient and its distribution. Particularly with a *Listing Algorithm*, where the task could be much slower and in particular much more greedy in memory.

Finally we explained a *Generalization For k-Cycles (C_k)*. We did this by going through various explanations on paths and homomorphic graphs. In the end we had the *Final Formula And Its Complexity* based mainly on the adjacency matrix and its multiplication via the exponent ω. Then to really finish, we did a *Comparisons Between C_3, C_4 and C_5* counting and realised that we could do some optimisation on the homomorphic subgraph for C_5 counting.

5 Improvement With Parallel Programming

In this section, we're going to check whether there are any improvements that can be made using basic parallelisation. To do this, we'll first review Reason Why Parallelisation Exist (Moore's Law), then look at Theoretical Limits (Amdahl's Law). Then we'll analyse the Matrix Multiplication Based Algorithms Speedups and the NodeIterators Speedups to really see acceleration possible with parallelisation.

5.1 Reason Why Parallelisation Exist (Moore's Law)

We might say that it's perfectly logical to have parallelisation on our machines. After all, our computers have been multi-tasking for decades and are capable of running several programs at the same time. But even before the 2000s, with the most basic computers, the OS, the programmes, the management of the user's keyboard input and the display were already multi-tasking. In reality, multi-tasking arrived long before parallelisation (via multi-processor/multi-core), thanks to schedulers, meaning the ability of a processor to switch between several tasks very quickly. In reality, multi-processor and multi-core architectures did not appear until much later, in the early 2000s, and this had nothing to do with multi-tasking.

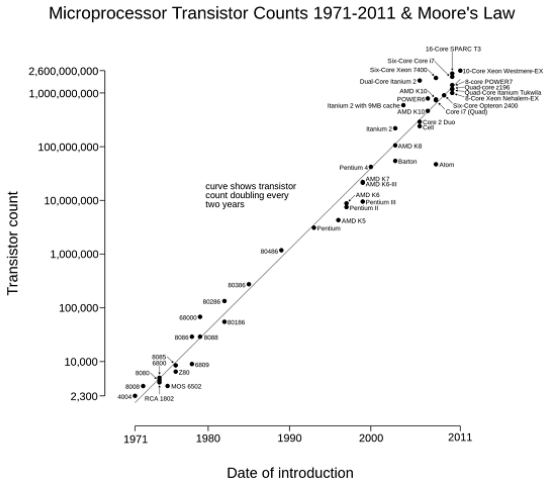


Figure 35: Evolution of transistor counts in chips

It's basically a solution to the desire to respect Moore's law. This law, expressed by Gordon Moore [32], co-founder of Intel, said that the number of transistors in integrated circuits would double every year, and then this law evolved to every two years. In concrete terms, this means that the power of our machines will increase exponentially. Unfortunately, the reality is that because of the difficulties of miniaturisation and cost, it has become difficult, if not impossible, to follow Moore's

Law on a single chip. As a result, manufacturers have come up with a solution of multi-core and multi-processor architectures.

Indeed, on the machine on which we are conducting our experiments, which is 7 years old, the Intel Core I7-7700HQ processor has 4 cores and is made using the 14nm process. Added to this is Intel's Hyper-Threading technology, which doubles the number of cores visible to the OS, giving us a total of 8 logical cores. That sounds like a lot, but in reality the very latest processors, such as the Core I7-14700K, have 20 cores, 8 high-performance and 12 low-performance, for a total of 28 logical cores, but are still made using the 10nm process. This clearly shows that the trend is towards multi-core architectures because of the difficulties of miniaturisation.

5.2 Theoretical Limits (Amdahl's Law)

We might think that all we have to do is increase the number of cores to make our programs proportionally faster, but in reality that's not true. The programs also need to take advantage of parallelisation, otherwise it would be pointless.

This is why we have Amdahl's law [20, 21], theorised by Gene Amdahl in 1967, which is a formula for calculating the speedup of a program according to its ability to be parallelised. For an acceleration denoted $Speedup(N)$ where N corresponds to the number of computing units and P corresponds to the parallelisable portion of a program, Amdahl's law can be formulated as follows:

$$Speedup(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

As a result, acceleration will always be limited by the non-parallelisable part of a program, and once a certain number of computing units have been reached, the gain in performance will be limited. This limit can be expressed as follows:

$$Speedup(N) \leq \frac{1}{(1-P)} \quad \text{or} \quad \lim_{N \to \infty} Speedup(N) = \frac{1}{(1-P)}$$

In practice, N should not correspond to the number of calculation units, but rather to the speed-up of the parallelised part. Indeed, a machine with 2 computing units will not always be able to do a perfectly shared task twice as fast. There are various reasons for this, such as competition for memory access (critical sections), scheduler interruptions, etc. But this gives us an idea of the limits of the parallelisation approach.

5.3 Matrix Multiplication Based Algorithms Speedups

In order to parallelize the algorithms based on matrix multiplication, we decided to rely on the parallelization of the multiplication it self. The parallelisation of the BLAS function is done by using the *`openblas_set_num_threads()`* to set the number of threads. For the naive function, we implemented a basic parallelisation using the *`<thread>`* header. To measure the possible speed-up, firstly, we are going to understand the impact of multiplication functions and see the *BLAS and Naive Speed-ups*. Then we'll do *Comparisons Between counting C_3, C_4 and C_5 Speed-ups*. This to better understand the impact of the non-parallelised part of the algorithms on the final speed-up.

5.3.1 BLAS and Naive Speed-ups

To compare the BLAS and naive functions, we decided to use the triangle counting algorithm (via the matrix cube) as a base line. We used the *email-eu-core* and *ego-facebook* graphs, which are of different sizes, also in order to see whether the size (and therefore the duration) of the operation also has an impact on the speed-up.

The first question we might ask is whether there is a difference in speed-up between the SGEMM and SSYMM functions when the number of threads is increased. If we look at the following plot, we can clearly see that they are almost identical, which is logical since we saw earlier that the performance was similar between the two functions (see 4.3.1).

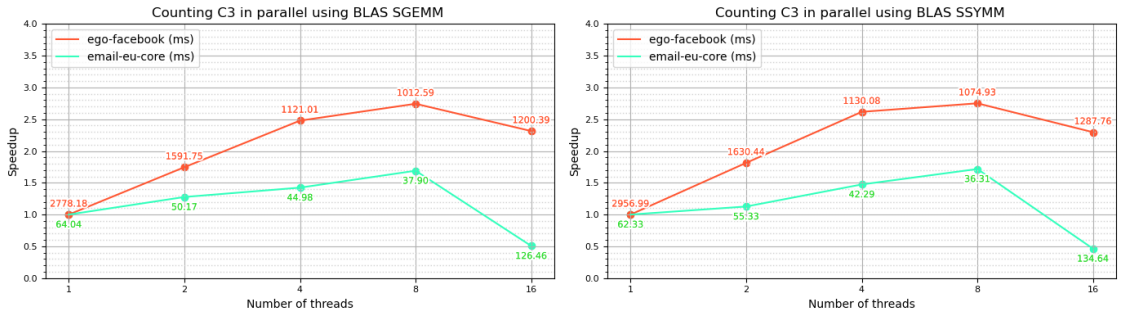


Figure 36: Comparison between the general and symmetrical BLAS function to count C_3 by cubing the adjacency matrix

However, we can see two other things. The first is that from 1 to 8 threads we get an acceleration and then a decrease. So we follow Amdahl's law (linearly) but at around 4-8 threads there is a slowdown due to the overhead of using more threads.

Since our machine has 8 logical cores, we can't run more than 8 threads simultaneously at a time T, so we'll simply be subject to additional thread management. The second is that with *email-eu-core* which is smaller and whose running time is faster, the acceleration is small. This is because the matrix multiplication task is smaller, so the non-parallelisable part has a greater impact on the speedup.

The second question is, what happens with slower matrix multiplication functions? Namely the BLAS DSYMM function (in double precision) and our naive function. To do this, we carried out the same experiment, the results of which are shown in the following plots.

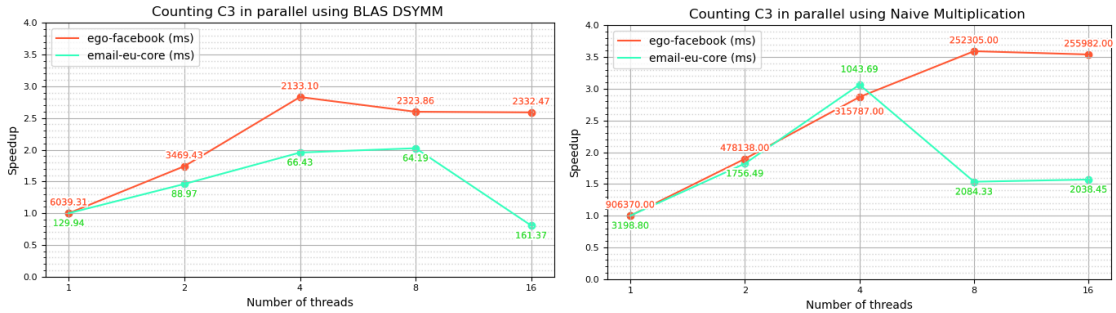


Figure 37: Speed-ups for BLAS DSYMM and Naive functions

On the BLAS DSYMM side, we can see the same maximum speed-up for *ego-facebook* of $\times 2.7$. For the second graph, *email-eu-core*, since the parallelised part takes longer and is therefore larger, the speed-up increases slightly from $\times 1.5$ to $\times 2.0$. But in both cases, the overhead seems to have been fixed at 4 threads. This is quite logical, given Amdahl's law and the fact that BLAS functions use SIMD instructions [37] for which only 1 logical core per processor is dedicated.

On the naive function side, the speed-up is much better, being $\times 3.5$ and $\times 3$ respectively. We still notice the overhead arriving earlier at 4 threads for smaller instances and more abruptly but can also continue very linearly up to 8 threads in bigger instances. In terms of pure speed-up factor for parallelisation, the naive function does slightly better. But in terms of pure performance, the BLAS functions have an indisputable advantage.

5.3.2 Comparisons Between counting C_3, C_4 and C_5 Speed-ups

So if we want to compare the different functions based on generalisation, meaning the counting algorithms for C_3, C_4 and C_5. We are going use the same two graphs as

above but also add two others *wiki-Vote* and *ca-HepPh* to see if there is a difference between the algorithms with instances of various sizes. We therefore started by comparing the counts of C_3 and C_4, both of which does 2 matrix multiplication have the same performances.

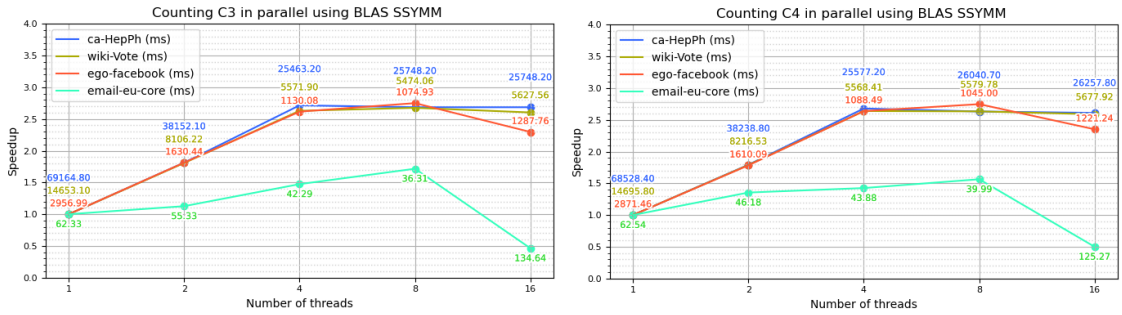


Figure 38: Speed-ups for the C_3 and C_4 counting algorithms

What we can see is that there is no great difference between the two algorithms in terms of speed-up. The smallest instance, the graph with the shortest running time, has a limited speed-up in both cases. When it comes to the larger graphs, it's interesting to note that even if we use larger instances, the speed-up remains capped at $\times 2.7$ and with an overhead still at 4 threads.

The final comparison is between the two variants of the C_5 counting. As a reminder, the first variant uses 4 matrix multiplication to calculate A^5 and A^3, while the second uses NodeIterator (in single thread) and avoids calculating A^3.

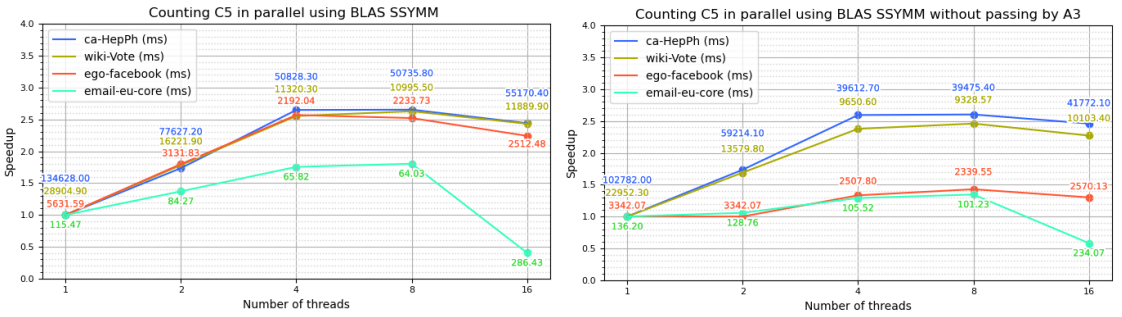


Figure 39: Speed-ups for the C_5 and C_5 (without A^3) counting algorithms

What we can see is that since the non-parallelised part is smaller for the algorithm on the right, due to the usage of NodeIterator in single core. The speed-up of the two graphs, *email-eu-core* and *ego-facebook* are extremely reduced, under the bar of $\times 1.5$. The other two graphs fare better, still following Amdahl's law up to 4 threads, even if *wiki-Vote* has a slight drop on the right.

5.4 NodeIterators Speedups

NodeIterator and NodeIterator++ are the only two algorithms not based on matrix multiplication that can be easily parallelised. Indeed, these two algorithms have nearly no critical sections, so we simply need to parallelise the main loop and recover the counters of all the threads at the end. There are other ways of parallelising, particularly for distributed systems, as explained in this article [2], but we won't cover that.

Meanwhile for the other algorithms, whether for listing C_3 or for counting/listing C_4, there are too much critical sections because of the various lists used in them which will certainly destroy the performances if we try to handle theses sections without huge modifications of the algorithms.

That's why we're simply going to check the *Impact Of Clustering Coefficient Distribution And Running Time* on the speed-ups with NodeIterator and NodeIterator++.

5.4.1 Impact Of Clustering Coefficient Distribution And Running Time

We try to see the impact of the clustering coefficient and the running time on the possible speed-up because, as we saw earlier (see 4.3.2), by analysing the number of paths processed by NodeIterators algorithms, that graphs with a high clustering coefficient and distribution were much faster with NodeIterator++. It would therefore be interesting to know whether, for the same number of triangles, two distinct graphs can have a different speedup.

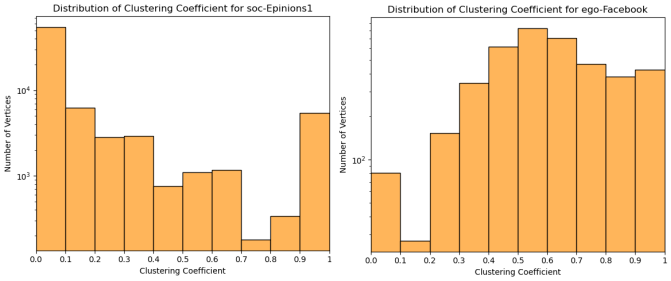


Figure 40: soc-Epinions1 and ego-facebook clustering coefficient distribution

To do this, we'll use the graphs *soc-Epinions1* ($n = 75879|m = 508837$) and *ego-facebook* ($n = 4039|m = 88234$), both of which have around 1.6 million triangles. Due to the fact that they have a very different number of vertices and edges, they respectively have a cc_{avg} of 0.1378 and 0.6055 and therefore the distribution above.

So we compared NodeIterator and NodeIterator++ in their parallelised versions.

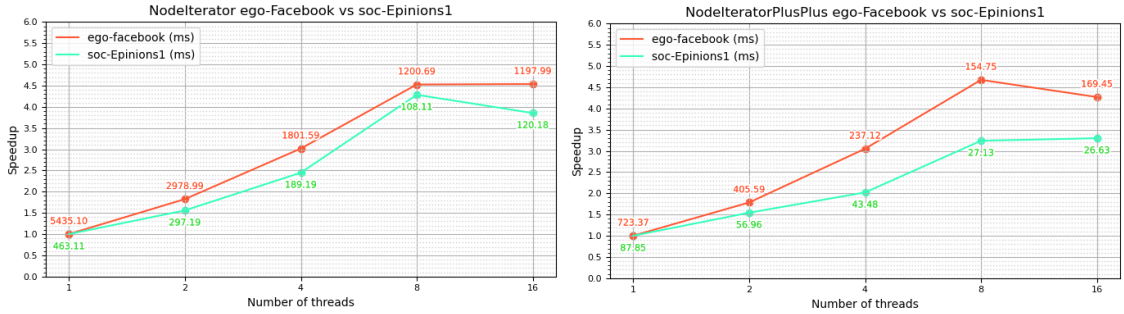


Figure 41: soc-Epinions1 and ego-facebook speedups with NodeIterators algorithms

The first thing we noticed compared with algorithms based on matrix multiplication is that the speed-up is much higher for *ego-facebook*, here it goes up to $\times 4.5$ whereas before we were limited to $\times 2.7$. Which makes sense because here the entire algorithm is parallelised.

Furthermore, we are certain that a parallelization of 8 threads running on 8 logic cores is possible. Previously, with the BLAS functions, it was possible that a part could only run with 4 logical cores on our machine because of the use of SIMD instructions, limited to one per processor. And this is what we see on the plots, Amdahl's law seems to be respected, linearly up to 8 threads without overhead.

Finally, *soc-Epinions1*, because of its already very low running time, seems to have a lower speed-up growth with NodeIterator++ than with NodeIterator. It therefore does not retain the same acceleration provided by NodeIterator++ with a higher number of threads, unlike *ego-facebook*. We are at a point where the running time is so low that even the simple fact of creating threads has an impact on the overall speed-up.

6 Conclusion

In the context of this thesis, we have seen that the use of graphs has a predominant place in our society. We are constantly connected via social networks on an unprecedented scale. So much so that there is an enormous amount of information to be exploited around groups of people and optimisation to be carried out for the study of these communities. As we have seen, this can be useful in concrete cases. This is the main motivation behind our objective to study algorithms for counting triangles and 4-cycles in graphs.

First of all, we started with a few theoretical points linked to the graph concept. Starting with the implication of graphs in social networks, and pointing out that there are different types of network, as well as the usages to which they can be applied. We then covered a few important points, such as the notations used, the concepts of graph homomorphism and density and, most importantly, the concept of clustering coefficients. These were the theoretical building blocks for the algorithms we implemented.

Next, we covered the various matrix multiplication techniques, so as to be able to choose the approach best suited to real experimentation for the implementation and study of algorithms. We looked at the naive divide-and-conquer method and the more theoretical laser method, before finally focusing on BLAS, which is the most suitable and optimized method for real-world use.

Then comes the main part, which meets our objective. This involves implementing and studying the triangle and 4-cycle counting algorithms. To do this, we have chosen a programming language, in this case C++, and explained the reasons for this choice, as well as the configuration and methodology we have chosen. Then, for both triangles and 4-cycles, we presented and implemented different algorithms, those based on matrix multiplication, those optimized for counting and likewise for listing. After much analysis, we realized the importance of several parameters. Firstly, the data structure used, both for the algorithm and for the graph, must be as well adapted as possible to get the best performance from a given choice. Secondly, the size of the graph. This is somewhat related to the data structure, as it can influence both performance and feasibility: some instances simply can't be run on a conventional computer. Thirdly, the clustering coefficient and its distribution of a graphs. In concrete terms, some algorithms do better than others, depending on the given graph. Then we explored a generalization for k-cycles.

Before moving to the conclusion, we also tried to experiment with parallelisation to see if it was possible to make considerable gains. First, we covered Moore's and Amdahl's laws to understand the context and possible limits. Then we tried some simple parallelisations on algorithms based on matrix multiplication and naive triangle counting (NodeIterators algorithms). We realized that the general speed-up was not very high for algorithms using BLAS multiplication, because BLAS multiplication was already very efficient. We also saw the impact of running time with larger or smaller instances on the speed-up and on the respect of Amdahl's law depending on the size of the parallelised part.

In conclusion, this work has enabled us to do several things. From a topic-related point of view, we've gained a much better understanding of the difficulties that can lie behind algorithms dealing with graphs. The importance of certain technical choices and the desired objective in order to have the most suitable implementation. From a scientific point of view, it made us curious and taught us how to carry out research work. Obviously there's still a lot to cover, other algorithms, other ways of implementing, we've only briefly covered parallelisation but it was already a nice challenge.

7 Bibliography

Main Articles

[1] Paul Burkhardt and David G. Harris. "Simple and efficient four-cycle counting on sparse graphs". In: (2023). arXiv: 2303.06090 [cs.DS]. URL: https://arxiv.org/abs/2303.06090.

[2] Siddharth Suri and Sergei Vassilvitskii. "Counting Triangles and the Curse of the Last Reducer". In: *Proceedings of the 20th International Conference on World Wide Web*. WWW '11. Hyderabad, India: Association for Computing Machinery, 2011, pp. 607–614. ISBN: 9781450306324. DOI: 10.1145/1963405.1963491. URL: https://doi.org/10.1145/1963405.1963491.

[3] Thomas Schank. "Algorithmic Aspects of Triangle-Based Network Analysis". In: (2007). DOI: 10.5445/IR/1000007159. URL: https://publikationen.bibliothek.kit.edu/1000007159.

[4] N. Alon, R. Yuster, and U. Zwick. "Finding and counting given length cycles". In: *Algorithmica* 17.3 (1997), pp. 209–223. ISSN: 1432-0541. DOI: 10.1007/BF02523189. URL: https://doi.org/10.1007/BF02523189.

[5] Alon Itai and Michael Rodeh. "Finding a Minimum Circuit in a Graph". In: *SIAM Journal on Computing* 7.4 (1978), pp. 413–423. DOI: 10.1137/0207033. eprint: https://doi.org/10.1137/0207033. URL: https://doi.org/10.1137/0207033.

[6] Norishige Chiba and Takao Nishizeki. "Arboricity and Subgraph Listing Algorithms". In: *SIAM Journal on Computing* 14.1 (1985), pp. 210–223. DOI: 10.1137/0214017. eprint: https://doi.org/10.1137/0214017. URL: https://doi.org/10.1137/0214017.

Articles

[7] Esteban Ortiz-Ospina. "The rise of social media". In: *Our World in Data* (2019). URL: https://ourworldindata.org/rise-of-social-media.

[8] Michele Ianni, Elio Masciari, and Giancarlo Sperlí. "A survey of Big Data dimensions vs Social Networks analysis". In: *Journal of Intelligent Information Systems* 57.1 (Aug. 1, 2021), pp. 73–100. ISSN: 1573-7675. DOI: 10.1007/s10844-020-00629-2. URL: https://doi.org/10.1007/s10844-020-00629-2.

[9] Ruth Haas. "Characterizations of Arboricity of Graphs". In: *Ars Comb.* 63 (Apr. 2002). URL: http://www.science.smith.edu/~rhaas/papers/Trees.pdf.

[10] Don R. Lick and Arthur T. White. "k-Degenerate Graphs". In: *Canadian Journal of Mathematics* 22.5 (1970), pp. 1082–1096. DOI: 10.4153/CJM-1970-125-1. URL: https://www.cambridge.org/core/journals/canadian-journal-of-mathematics/article/kdegenerate-graphs/45829E7755FA1F4D72FD84530F492E8A.

[11] Paul Burkhardt, Vance Faber, and David Harris. "Bounds and algorithms for graph trusses". In: *Journal of Graph Algorithms and Applications* 24 (Jan. 2020), pp. 191–214. DOI: 10.7155/jgaa.00527. URL: https://www.researchgate.net/publication/340520725_Bounds_and_algorithms_for_graph_trusses.

[12] D J Watts and S H Strogatz. "Collective dynamics of 'small-world' networks". en. In: *Nature* 393.6684 (June 1998), pp. 440–442. URL: https://www.nature.com/articles/30918.

[13] Chee Yap. "A Real Elementary Approach to the Master Recurrence and Generalizations". In: vol. 6648. May 2011, pp. 14–26. ISBN: 978-3-642-20876-8. DOI: 10.1007/978-3-642-20877-5_3.

[14] Volker Strassen. "Gaussian elimination is not optimal". In: *Numerische Mathematik* 13.4 (1969), pp. 354–356. ISSN: 0945-3245. DOI: 10.1007/BF02165411. URL: https://doi.org/10.1007/BF02165411.

[15] Josh Alman and Virginia Vassilevska Williams. "A Refined Laser Method and Faster Matrix Multiplication". In: (2020). DOI: arXiv:2010.05846. arXiv: 2010.05846 [cs.DS]. URL: https://doi.org/10.48550/arXiv.2010.05846.

[16] Don Coppersmith and Shmuel Winograd. "Matrix multiplication via arithmetic progressions". In: *Journal of Symbolic Computation* 9.3 (1990). Computational algebraic complexity editorial, pp. 251–280. ISSN: 0747-7171. DOI: https://doi.org/10.1016/S0747-7171(08)80013-2. URL: https://www.sciencedirect.com/science/article/pii/S0747717108800132.

[17] Rui Pereira et al. "Ranking programming languages by energy efficiency". In: *Science of Computer Programming* 205 (2021), p. 102609. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2021.102609. URL: https://www.sciencedirect.com/science/article/pii/S0167642321000022.

[18] Ran Duan, Hongxun Wu, and Renfei Zhou. "Faster Matrix Multiplication via Asymmetric Hashing". In: (2023). DOI: 10.48550/arXiv.2210.10173. arXiv: 2210.10173 [cs.DS]. URL: https://doi.org/10.48550/arXiv.2210.10173.

[19] Kazushige Goto and Robert Van De Geijn. "High-performance implementation of the level-3 BLAS". In: *ACM Trans. Math. Softw.* 35.1 (Aug. 2008). ISSN: 0098-3500. DOI: 10.1145/1377603.1377607. URL: https://doi.org/10.1145/1377603.1377607.

[20] Gene M. Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: AFIPS '67 (Spring) (1967), pp. 483–485. DOI: 10.1145/1465482.1465560. URL: https://doi.org/10.1145/1465482.1465560.

[21] David P. Rodgers. "Improvements in multiprocessor system design". In: *SIGARCH Comput. Archit. News* 13.3 (June 1985), pp. 225–231. ISSN: 0163-5964. DOI: 10.1145/327070.327215. URL: https://doi.org/10.1145/327070.327215.

Books

[22] Reinhard Diestel. *Graph theory*. eng. 4th ed. Graduate texts in mathematics ; 173. Heidelberg ; Springer, 2010. ISBN: 9783642142789.

[23] Thomas H. Cormen. *Introduction to algorithms*. eng. 3rd ed. Cambridge, Mass: MIT Press, 2009. ISBN: 9780262533058.

Websites

[24] Sajil C. K. *Matrix Multiplication in Real-Life*. Intuitive Tutorials. URL: https://intuitivetutorial.com/2023/05/17/matrix-multiplication-in-real-life/#:~:text=A%20matrix%20can%20represent%20a,%2C%20economics%2C%20and%20other%20fields. (visited on 07/28/2023).

[25] Kevin Hartnett. *Matrix Multiplication Inches Closer to Mythic Goal*. Quanta Magazine. URL: https://www.quantamagazine.org/mathematicians-inch-closer-to-matrix-multiplication-goal-20210323/ (visited on 08/08/2023).

[26] Duane Q. Nykamp. *Multiplying matrices and vectors*. Math Insight. URL: https://mathinsight.org/matrix_vector_multiplication (visited on 07/16/2023).

[27] Josh Alman. *Josh Alman. Algorithms and Barriers for Fast Matrix Multiplication*. Youtube. URL: https://youtu.be/DruwS2_cVys (visited on 07/28/2023).

[28] Isaac Gouy. *The Computer Language Benchmarks Game*. CLBG. URL: https://benchmarksgame-team.pages.debian.net/benchmarksgame/ (visited on 03/29/2024).

[29] TIOBE Software BV. *TIOBE Index*. TIOBE. URL: https://www.tiobe.com/tiobe-index/ (visited on 04/01/2024).

[30] CS Academy. *CS Academy Graph Editor*. CS Academy. URL: https://csacademy.com/app/graph_editor/ (visited on 04/15/2024).

[31] Ryan O'Donnell. *Triangle Counting and Matrix Multiplication*. Camegie Mellon University. URL: https://www.cs.cmu.edu/~15750/notes/lec1.pdf (visited on 07/16/2023).

[32] Intel. *Moore's Law*. Intel. URL: https://www.intel.com/content/www/us/en/newsroom/resources/moores-law.html#:~:text=Follow%20Intel%20Newsroom%20on%20social%3A&text=Moore's%20Law%20is%20the%20observation,original%20paper%20published%20in%201965. (visited on 04/09/2024).

Frameworks And Tools

[33] National Science Foundation. *BLAS (Basic Linear Algebra Subprograms)*. URL: https://www.netlib.org/blas/ (visited on 03/12/2024).

[34] Cppreference.com. *C++ Standard Library*. URL: https://en.cppreference.com/w/cpp/standard_library (visited on 03/24/2024).

[35] Martin Kroeker Zhang Xianyi. *OpenBLAS*. URL: https://www.openblas.net/ (visited on 03/13/2024).

[36] The University of Utah. *LAPACK (Linear Algebra Package)*. URL: https://www.math.utah.edu/software/lapack/ (visited on 03/13/2024).

[37] Intel. *Intel Instruction Set Extensions Technology*. URL: https://www.intel.com/content/www/us/en/support/articles/000005779/processors.html (visited on 08/06/2023).

[38] SNAP group Stanford University. *Stanford Network Project Platform (SNAP)*. URL: https://snap.stanford.edu/index.html (visited on 03/24/2024).

Images/Figures

[39] Alhussein Fawzi et al. *FIGURE: Tensor Example (Discovering faster matrix multiplication algorithms with reinforcement learning)*. 2022. DOI: 10.1038/s41586-022-05172-4. URL: https://doi.org/10.1038/s41586-022-05172-4.

[40] Rui Pereira et al. *FIGURE: Binary-trees comparison (Ranking programming languages by energy efficiency)*. 2021. DOI: https://doi.org/10.1016/j.scico.2021.102609. URL: https://www.sciencedirect.com/science/article/pii/S0167642321000022.

[41] Andrew Davies. *FIGURE: Evolution of transistor counts in chips*. Australian Strategic Policy Institute. 2013. URL: https://www.aspistrategist.org.au/graph-of-the-week-moores-law/ (visited on 04/24/2024).

8 Appendix

8.1 Tables

No.	Name	Nodes	Edges	cc_{avg}	Triangles	4-Cycles
Social Networks						
1	ego-Facebook	4039	88234	0.6055	1612010	144023053
2	ego-Twitter	81306	1768149	0.5653	13082506	1023837230
3	soc-Epinions1	75879	508837	0.1378	1624481	166635817
4	soc-Slashdot0811	77360	905468	0.0555	551724	49965153
5	soc-Slashdot0922	82168	948464	0.0603	602592	58678630
6	wiki-Vote	7115	103689	0.1409	608389	57654491
Ground-truth Communities						
7	com-Youtube	1134890	2987624	0.0808	3056386	468774021
8	com-DBLP	317080	1049866	0.6324	2224385	55107655
9	com-Amazon	334863	925872	0.3967	667129	3125323
10	email-Eu-core	1005	25571	0.3994	105461	4647873
Communication Networks						
11	email-EuAll	265214	420045	0.0671	267313	18421946
12	email-Enron	36692	183831	0.4970	727044	36262229
13	wiki-Talk	2394385	5021410	0.0526	9203519	21520131141
Citation Networks						
14	cit-HepPh	34546	421578	0.2848	1276868	39015537
15	cit-HepTh	27770	352807	0.3120	1478735	63698507
Collaboration Networks						
16	ca-AstroPh	18772	198110	0.6306	1351441	44916549
17	ca-CondMat	23133	93497	0.6334	173361	1505383
18	ca-GrQc	5242	14496	0.5296	48260	1054723
19	ca-HepPh	12008	118521	0.6115	3358499	486866960
20	ca-HepTh	9877	25998	0.4714	28339	239081
Web Graphs						
21	web-BerkStan	685230	7600595	0.5967	64690980	127118333411
22	web-Google	875713	5105039	0.5143	13391903	539575204
23	web-NotreDame	325729	1497134	0.2346	8910005	884960527
24	web-Stanford	281903	2312497	0.5976	11329473	13316840570
Product Co-purchasing Networks						
25	amazon0302	262111	1234877	0.4198	717719	2477087
26	amazon0312	400727	3200440	0.4022	3686467	32229022
27	amazon0505	410236	3356824	0.4064	3951063	35492004
28	amazon0601	403394	3387388	0.4177	3986507	35661461
Internet Peer-to-Peer Networks						
29	p2p-Gnutella04	10876	39994	0.0062	934	28497
30	p2p-Gnutella05	8846	31839	0.0072	1112	64998
31	p2p-Gnutella06	8717	31525	0.0067	1142	90764
32	p2p-Gnutella08	6301	20777	0.0109	2383	87885
33	p2p-Gnutella09	8114	26013	0.0095	2354	96902
34	p2p-Gnutella24	26518	65369	0.0055	986	8794
35	p2p-Gnutella25	22687	54705	0.0053	806	13517
36	p2p-Gnutella30	36682	88328	0.0063	1590	46363
37	p2p-Gnutella31	62586	147892	0.0055	2024	42466
Road Networks						
38	roadNet-CA	1965206	2766607	0.0464	120676	262339
39	roadNet-PA	1088092	1541898	0.0465	67150	157802
40	roadNet-TX	1379917	1921660	0.0470	82869	183252

Table 10: Graphs used for the NodeIterator++ acceleration results on Figure 19

8.2 Repository

The experiments and analysis of this thesis were possible using the code and resources I codded and built on Github. The repository has all the scripts, datasets and plots I used to write this thesis. The only requirement are a C++ compiler and an appropriate OpenBLAS[35] installation.

If you want to check the source code or do the experiments again, you can find the whole repository on GitHub at this link:
https://github.com/oruckaplanulb/MEMO-F524—Master-Thesis—Counting-Rapidly-Triangles-And-Others-Subgraphs.