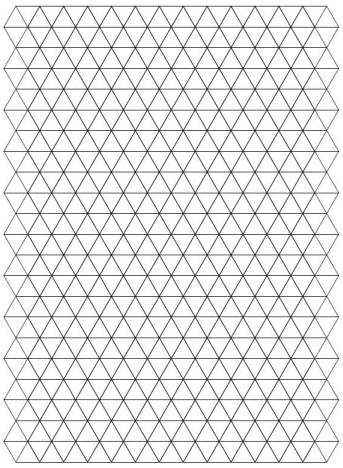
ULB UNIVERSITÉ LIBRE DE BRUXELLES

Faculty of Sciences

Departement of Computer Science



MEMO-F403 - Preparatory work for the master thesis

Counting rapidly triangles and 4-cycle subgraphs

Author :

Oruç Kaplan

000540662

Promoter :

Prof. Gwenaël JORET

August 11, 2023

M-INFOS

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, even over longer distances, but this still came at a considerable cost, which, from a purely physical and economic point of view, 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, having fun, working and communicating, much more widely, more easily and at a reduced cost. This has had the impact of creating larger communities doing things together.

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 [1] , as shown by the following graph showing the number of active users on the respective social networks between 2004 and 2018.

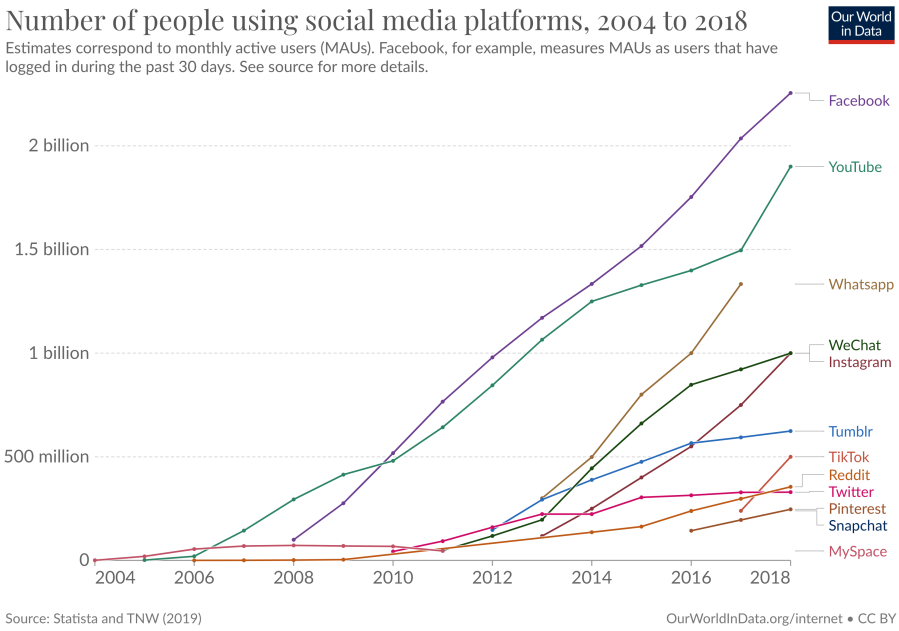


Figure 1: Evolution of the number of monthly active users over the years on every social media platform

Before getting into why these groups interest us and how we can analyze them, we first need to model them. To do this, the simplest approach is to represent the people by dots and the fact that they communicate with each other by lines. In other words, we're going to use a mathematical object widely used in computer science, namely *Graphs* made up of interconnected vertices linked

by edges. The notation details being developed in this section (see 2.1). Many studies like [2] have already shown that there are different structure 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 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 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.

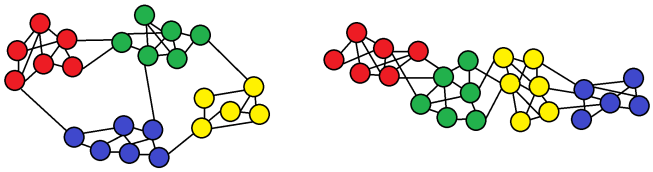


Figure 2: Example of assortative groups on the left and ordered on the right

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.

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 pre-thesis, which will be followed by the complete thesis, I'm going to study algorithms for counting the number of triangles, 4-cycles and, why not, other types of subgraphs 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 I discuss in the dedicated section (see 2.3) and which is an indicator of clustering.

So this pre-thesis is divided into three parts. Firstly, *Theoretical Aspects*, which covers all the purely theoretical aspects that will help you understand what follows. Then the *Matrix Multiplication* section, which is a key mathematical operation in many graph-manipulating algorithms. And finally, *Counting Algorithms*, which contains the various counting algorithms that I'll be able to implement and experiment with in my thesis. Obviously this won't be the only thing I'll be doing during my thesis, which is why there's a final *Objectives* section where I describe what I intend to do in order to complete my final thesis.

2 Theoretical Aspects

In this first section, we will explain all the theoretical aspects that will be used in the algorithms that will be developed in this thesis. First, we'll look at *Graph Notations*, 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. Finally, we'll explain the concept of the *MapReduce* programming model, which enables parallelization in distributed systems.

2.1 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 a set of vertices in which each 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 [3].

2.2 Graph Density

In this section, we present two 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, but the ones we're most interested in are arboricity and average degeneracy.

2.2.1 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 [4] 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 3 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. Thus, we can write $a(K_6) = 3$.

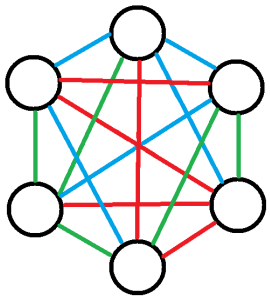


Figure 3: K_6 Arboricity example

2.2.2 Average Degeneracy

Not to be confused with graph degeneracy, which is the smallest k such that the graph is k-degenerate [5] of a graph, which means that for a graph G and all its subgraphs, there exists a vertex of degree k at most. The average degeneracy [6, 7] 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,

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

2.3 Clustering Coefficient

2.3.1 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.

2.3.2 Definition

Before going further, there is a distinction to make between what we call a local clustering coefficient and a global one. The local one is focused on a specific vertex meanwhile the global is an average of all the clustering coefficient of a graph.

The (local) clustering coefficient, as mentioned in theses articles [8, 9], is defined as follows :
Let $G = (V, E)$ be an unweighted, undirected simple graph and $v \in V$. The (local) clustering coefficient for a node $v \in V$ in an undirected graph is defined by this formula,

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

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 4.1).

As shown in the following Figure 4, the possible values of the clustering coefficient of a given vertex are between 0 and 1.

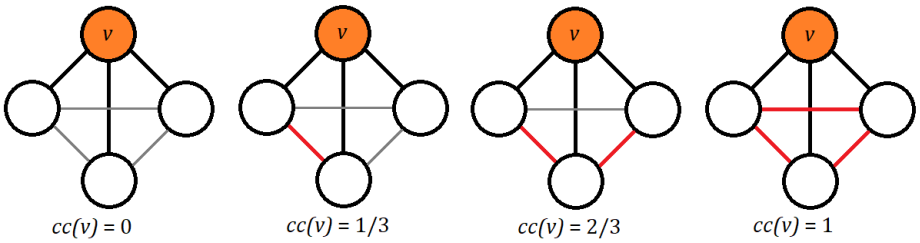


Figure 4: Clustering coefficient example

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.

2.4 MapReduce

With the development of large-scale applications such as social networking, scientific research and web analysis, a programming model and framework introduced by Google [10] has emerged. Called MapReduce, it enables the processing of large-scale data in distributed systems. MapReduce makes it possible to process large quantities of data through clusters of machines, something that is difficult to achieve with conventional sequential programming.

The main idea is to divide the task into two parts. The first is map, where the data is distributed across several machines for parallel processing. Each machine will transform the data in its respective set to form a $\langle key; value \rangle$ pair. These are then sorted and reorganized in an intermediate phase called shuffle, before moving on to the reduce phase. The second part, reduce, consists of receiving these $\langle key; value \rangle$ pairs and reducing them using a user-defined function, in order to replace the key with a set of values, preferably into one key with a single value.

Proceeding in this way in distributed systems makes the process fault-tolerant. In fact, by dividing up the process, in the event of a machine failure, all you have to do is re-execute the map, shuffle or reduce part that failed.

2.4.1 Architecture

In order for this to work in a distributed system like Apache Hadoop [11], a huge framework for setting up distributed systems to store and process huge amounts of data and which is based on MapReduce, two essential roles must be fulfilled. The first is that data must be efficiently available and accessible. The second is the assignment of different tasks to machines/nodes.

The first task is performed by the distributed system itself. The data will be made available in copies across several nodes in the same cluster. If a node fails, the data can be recovered by the copied replicas. The nature of the system also enables parallelism with all nodes running in the same cluster, so data is easily accessible with fault-tolerance capability.

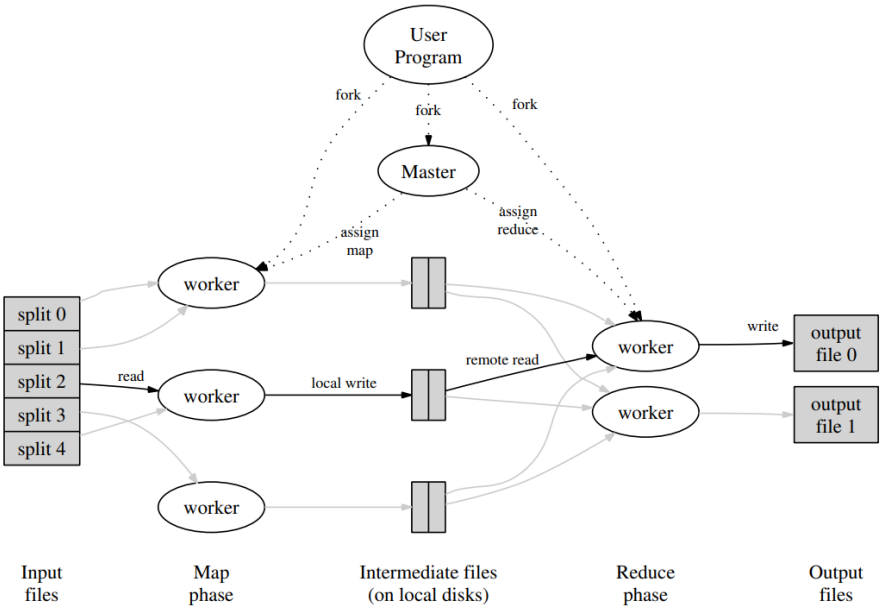


Figure 5: MapReduce processing

The second task is performed by the MapReduce framework used by the distributed system. In concrete terms, a node master will be in charge of coordinating the nodes, dividing up the data to be shared between them and assigning the map and reduce tasks to the different machines.
This is done by first performing the initial split, then assigning the map tasks to the nodes, which corresponds to the Map phase whose intermediate result will pass through the Shuffle phase performed by the node master, and finally the nodes having received the reduce task will process this, the Reduce phase, which will generate the final result.

2.4.2 Example

To understand how MapReduce works, the best way is to illustrate it with the word counting problem. This is one of the many problems presented in the original Google paper [10], such as Distributed Grep, URL Access Frequency, Inverted Index, Distributed Sort and so on.

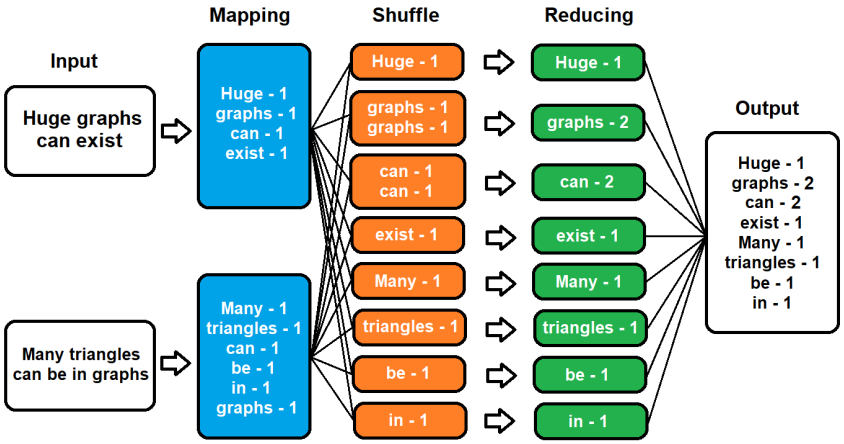


Figure 6: Words counting example with MapReduce

The idea is simple: we want to count the occurrence of words present in a certain number of files. To do this, each node processes its file(s) and associates the keys, that is, the word, with a value corresponding to their occurrence, via its map function. During the Shuffle phase, the master then sorts all the data it receives to group the keys together. Finally, other nodes take over by adding up the different occurrences of a particular word using the reduce function. Finally, the output file will receive the final result.

3 Matrix Multiplication

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 [12], 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 work mostly with adjacency matrices. 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 and also see the history of the improvements done from the basic method to the last discoveries.

In the following Figure 7, you will find the curve of the evolution [13] 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.

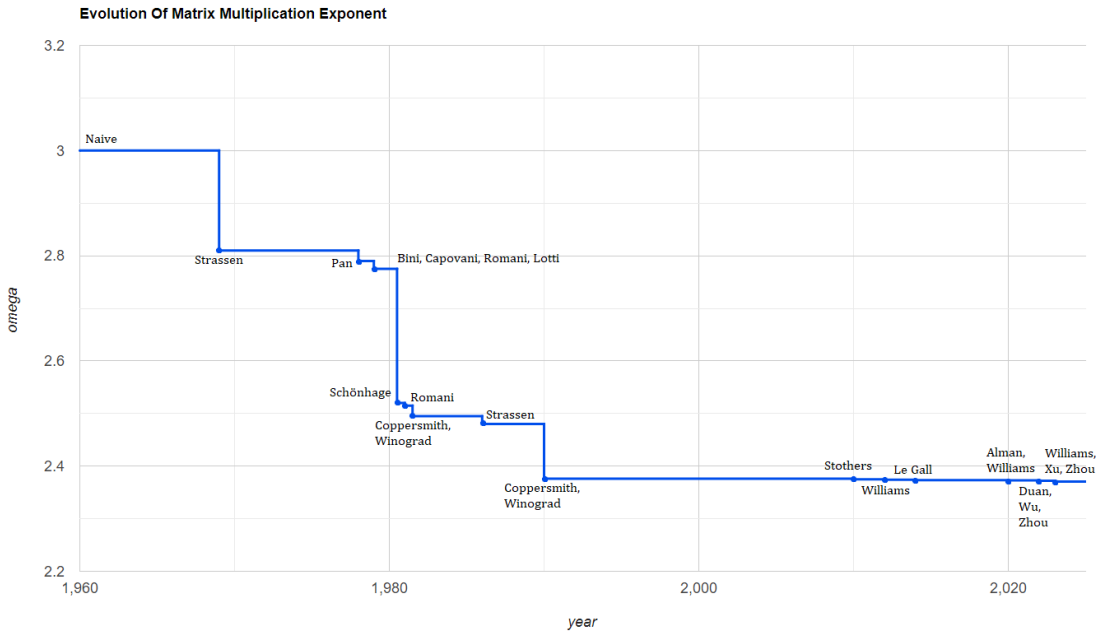


Figure 7: Evolution of the matrix multiplication algorithms time complexity given by the value of ω for $O(n^\omega)$

3.1 Naive Iterative Way

To calculate a matrix multiplication [14], 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 [15, 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 [16]. 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 [17]('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 $A \times B$
1: **if** `len`$(A) <= 2$ **then**
2: **return** $A \times B$
3: Split A and B into four, (a,b,c,d) and (e,f,g,h), each of size $n/2 \times n/2$
4: **Compute the following seven products:**
5: $P_1 \leftarrow$ `Strassen`$(a + d, e + h)$
6: $P_2 \leftarrow$ `Strassen`$(d, g - e)$
7: $P_3 \leftarrow$ `Strassen`$(a + b, h)$
8: $P_4 \leftarrow$ `Strassen`$(b - d, g + h)$
9: $P_5 \leftarrow$ `Strassen`$(a, f - h)$
10: $P_6 \leftarrow$ `Strassen`$(c + d, e)$
11: $P_7 \leftarrow$ `Strassen`$(a - c, e + f)$
12: **Compute the four blocks of the resulting matrix C:**
13: $C_{11} \leftarrow P_1 + P_2 - P_3 + P_4$
14: $C_{12} \leftarrow P_5 + P_3$
15: $C_{21} \leftarrow P_6 + P_2$
16: $C_{22} \leftarrow P_5 + P_1 - P_6 - P_7$
17: **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 [16] we end up with the following time complexity, which 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 [18, 19]. 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 8, the entries $\langle a_1, b_2, c_2\rangle$ and $\langle a_2, b_4, c_2\rangle$ are set to 1.

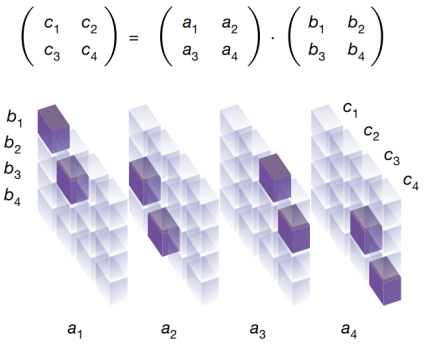


Figure 8: Tensor representing the computation of a 2×2 matrix

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, $x_iy_jz_k$ 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,

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

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 [17]('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 [18].

The most commonly used approach is to use the Tensor introduced by Coppersmith-Winograd [20]('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 $q = 32$.

4 Counting Algorithms

4.1 Triangles

In this section, we'll look at algorithms for counting the number of triangles in a graph. Counting triangles is very interesting indeed, as it enables us, for example, to calculate what we call the clustering coefficient (see 2.3). 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 this article [8]. From the two basic naive methods to those two presented by the author using MapReduce technique (see 2.4). There is also another algorithm [21] based on matrix multiplication that can be used to count triangles. And another one [22] which is based on the lowest possible complexity exponent ω of a matrix multiplication.

4.1.1 NodeIterator

The first way of counting triangles in a graph is a rather naive method. The algorithm is called NodeIterator, see algorithm (Alg:2), 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 2 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 do be quadratic ($O(n^2)$) in running time. Thus, the complexity can be summarized as follows: $O(\sum_{v \in V} d_v^2)$.

4.1.2 Adjacency Matrix Squared Based

Before going any further, there's already an algorithm [21] for improving complexity from $O(n^3)$ to something between $O(n^3)$ and $O(n^2)$ in the "best case". This algorithm is based on matrix multiplication, the principle of this algorithm is that $\sum_{i,j} A_{ij} \cdot (A^2)_{ij}$ gives us the number of triangles in the graph. Where A represents the adjacency matrix of the graph and n the size of it and thus also the number of vertices.

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** $T/3$

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 $\omega < 2.371866$ [23].

4.1.3 NodeIterator++

This third algorithm (Alg:4) 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 $d_v \geq d_v$. Thus in the following algorithm, v will be responsible to count if its neighbors (in the corresponding triangle) have a higher degree.

Algorithm 4 NodeIterator++(V,E)

1: $T \leftarrow 0$
2: **for** $v \in V$ **do**
3: **for** $u \in \Gamma(v)$ and $u \succ v$ **do**
4: **for** $w \in \Gamma(v)$ and $w \succ u$ **do**
5: **if** $(u, w) \in E$ **then**
6: $T \leftarrow T + 1$
7: **return** T

The running time of this algorithm is $O(m^{3/2})$ [7]. To illustrate this upper bound, the author of the article [8] 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.

4.1.4 MR-NodeIterator++

The problem with the previous algorithms (Alg:2,3,4) is that they assume that the whole graph can be stored in the memory of a single machine. In practice, for very large graphs such as those modeling social networks, this is impossible. This is why the author of [8] presents an adaptation of the NodeIterator++ algorithm to MapReduce (see 2.4), which will allow the task to be divided.

The idea is to divide the algorithm (Alg:5) into two parts by running the MapReduce framework twice. The first will generate all paths of length two for each vertex. The second will check for each of them whether the path can be closed to form a triangle and count it.

Algorithm 5 MR-NodeIterator++(V, E)

```
1:  Map 1(Input: ⟨(u, v); ∅⟩)
2:      if v ≻ u then
3:          emit ⟨u; v⟩
4:  Reduce 1(Input: ⟨v; S ⊆ Γ(v)⟩)
5:      for (u, w) : u, w ∈ S do
6:          emit ⟨v; (u, w)⟩
7:  Map 2
8:      if Input of type ⟨v; (u, w)⟩ then
9:          emit ⟨(u, w); v⟩
10:     if Input of type ⟨(u, v); ∅⟩ then
11:         emit ⟨(u, v); $⟩
12: Reduce 2(Input: ⟨(u, w); S ⊆ V ∪ {$}⟩)
13:     if $ ∈ S then
14:         for v ∈ S ∩ V do
15:             emit ⟨v; 1⟩
```

As for time complexity, the author indicates that for the first part, the algorithm generates $O(n^2)$ paths, and that for the second part, where the number of triangles is counted, it is $O(m^{3/2})$. In other words, not much has changed in this respect between MR-NodeIterator++ and NodeIterator++. What is interesting, however, is the reduction in space complexity. Each machine during Reduce 1 receives $O(\sqrt{m})$ edges. The proof is also detailed here [8].

4.1.5 MR-GraphPartition++

Where the previous algorithm (Alg:5) tried to generate all paths of length two and then check whether they were triangles by trying to close them, by dividing the task. This second algorithm (Alg:6) proposed by the author [8] aims instead to divide the graph into subsets of equal size. The idea is to have p groups of vertices $(V_1, V_2, ..., V_p)$ where p is greater than 0 and $Vi \cap Vj = \emptyset$, meaning that each vertex is assigned to exactly one group. The algorithm will create all the combinations of induced subgraphs in the form G_{ijk} that exist in G, which means the combination of 3 different partitions (V_i, V_j, V_k), this is done in the first Map part. The Reduce part is responsible for counting the triangles. But to avoid counting triangles incorrectly or several times, as each triangle may appear in several subgraphs, the algorithm will count them using weights. This algorithm is designed to use any sequential counting method, it can be directly incorporated in it to count the triangles in each G_{ijk}.

Algorithm 6 MR-GraphPartition(V, E, ρ)

```
1:  MAP 1(Input: ⟨(u, v); 1⟩)
2:      Let h(·) be a universal hash function into [0, ρ − 1]
3:      i ← h(u)
4:      j ← h(v)
5:      for a ∈ [0, ρ − 1] do
6:          for b ∈ [a + 1, ρ − 1] do
7:              for c ∈ [b + 1, ρ − 1] do
8:                  if {i, j} ⊆ {a, b, c} then
9:                      emit ⟨(a, b, c); (u, v)⟩
10: REDUCE 1(Input ⟨(i, j, k); E_ijk ⊆ E⟩)
11:     Count triangles on G_ijk
12:     for every triangle (u, v, w) do
13:         z ← 1
14:         if h(u) = h(v) = h(w) then
15:             z ← C(h(u), 2) + h(u)(ρ − h(u) − 1) + C(ρ − h(u) − 1, 2)
16:         else if h(u) = h(v) | h(v) = h(w) | h(u) = h(w) then
17:             z ← ρ − 2
18:         Scale triangle (u, v, w) weight by 1/z
```

The algorithm has been shown [8] to be highly efficient in terms of space and running time. The author demonstrates that the total space used at the end of the first map part is $O(pm)$ and that the expected input size of Reduce is $O(m/p^2)$. He also proves that the weight system works, that each triangle is counted only once and that for a number of partitions less than $\sqrt{m}$, the total amount of work by all machines is $O(m^{3/2})$.

4.1.6 AYZ Algorithm

The algorithm described in the following articles [22, 24] finally gets it down under the $O(m^{3/2})$ mark. The idea is to divide the vertices into two sets, those with a low degree and those with a high degree, and process them independently. The limit is set by $\beta = m^{(\omega-1)/(\omega+1)}$ where V_{low} vertices are $\{v \in V : d_v \leq \beta\}$ and V_{high} receives the other vertices. ω being the matrix multiplication exponent, in other words the exponent in the time complexity of a matrix multiplication that can be defined as $O(n^\omega)$. The principle is a mix between node-iterator algorithms and matrix multiplication, where V_{low} are processed as node-iterator and V_{high} are processed by matrix multiplication. At the end the algorithm will count the local counting of triangles for each vertex v given by $\delta(v)$.

Algorithm 7 Triangle Counter "ayz"

Require: Graph $G = (V, E)$; matrix multiplication parameter ω
Ensure: Number of triangles $\delta(v)$ for each vertex v

1: $\beta \leftarrow m^{(\omega-1)/(\omega+1)}$
2: **for** $v \in V$ **do**
3: $\quad \delta(v) \leftarrow 0$
4: $\quad$ **if** $d_v \leq \beta$ **then**
5: $\quad\quad V_{low} \leftarrow V_{low} \cup \{v\}$
6: $\quad$ **else**
7: $\quad\quad V_{high} \leftarrow V_{high} \cup \{v\}$
8: **for** $v \in V_{low}$ **do**
9: $\quad$ **for all** pairs of Neighbors $\{u, w\}$ of v **do**
10: $\quad\quad$ **if** edge between u and w exists **then**
11: $\quad\quad\quad$ **if** $u, w \in V_{low}$ **then**
12: $\quad\quad\quad\quad$ **for** $v \in \{v, u, w\}$ **do**
13: $\quad\quad\quad\quad\quad \delta(v) \leftarrow \delta(v) + \frac{1}{3}$
14: $\quad\quad\quad$ **else if** $u, w \in V_{high}$ **then**
15: $\quad\quad\quad\quad$ **for** $v \in \{v, u, w\}$ **do**
16: $\quad\quad\quad\quad\quad \delta(v) \leftarrow \delta(v) + 1$
17: $\quad\quad\quad$ **else**
18: $\quad\quad\quad\quad$ **for** $v \in \{v, u, w\}$ **do**
19: $\quad\quad\quad\quad\quad \delta(v) \leftarrow \delta(v) + \frac{1}{2}$
20: $A \leftarrow$ adjacency matrix of node-induced subgraph of V_{high}
21: $M \leftarrow A^3$
22: **for** $v \in V_{high}$ **do**
23: $\quad \delta(v) \leftarrow \delta(v) + \frac{M(i,i)}{2}$ where i is index of v in V_{high}

The algorithm distinguishes between three, two, one or zero vertices belonging to V_{low} for a given triangle. If all vertices are of low degree, then each contributes $1/3$ to the local count for the current vertex. If there are two, it will be $1/2$ and if there is only one, i.e. v, it will be 1. In the case where v is of high degree, a simple count of the diagonal of the adjacency matrix is performed. This matrix is obtained by only putting the adjacency matrix of the subgraphs with V_{high} vertices to the cube, while the loop at line 8 processes only the low-degree vertices. For this reason, this algorithm has a time complexity of $O(m^{2\omega/(\omega+1)})$, while the rest of the algorithm runs in constant time.

Therefore, as for the algorithm (Alg:3), if we take into consideration that ω is the matrix multiplication exponent, the value was set to $\omega < 2.376$ when the article [22] was published, they based their ω value on Don Coppersmith and Shmuel Winograd [20]. But the value is currently set to $\omega < 2.371866$ [23]. We arrive at a complexity of $O(m^{1.4069})$ instead $O(m^{1.4076})$. Since the theoretical limit to compute a matrix multiplication is $O(n^2)$, with this algorithm, it's impossible to get below $O(m^{1.3334})$.

4.2 4-Cycles

In the previous section, several triangle-counting algorithms are presented. It turns out that other types of subgraphs can also be counted. In particular, the author [8] mentions that the algorithm (Alg:6) can be adapted to count subgraphs such as $K_{2,3}$ or any other type. In this section, we'll look at algorithms for counting 4-cylces (C_4), which in a bipartite graph correspond to $K_{2,2}$, known as a butterfly. Many have already studied butterfly counting in bipartite graphs in articles such as [25, 26]. But the article [7] 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, and which are deterministic unlike many algorithms using hash tables.

The complexity measures proposed here are based on graph density (see 2.2) because it is known that $\overline{\delta}(G) \leq 2a(G) \leq O(\sqrt{m})$ [7] and an efficient algorithm for C_4 counting should $O(\overline{\delta}(G)) \leq O(ma(G))$ time. In addition, the author forbids sorting the graph, as sorting the adjacency matrix would take $\Omega(m \log n)$ time with standard sorting techniques. In the case of a graph G where $\overline{\delta}(G) \ll \log n$, an algorithm using sorting will have worse time complexity.

4.2.1 Base Counting And Enumeration Algorithms

The idea behind this algorithm (Alg:8) is similar to that used for triangles (Alg:4), 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. The number of 4-cycles in the graph G being stored in $\diamond(G)$.

Algorithm 8 4-cycle counting

Require: zero-initialized array L 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 ◇(G) := ◇(G) + L(y)              ▷ Count of 4-cycles
5:             set L(y) := L(y) + 1
6:     for u ∈ Γ(v) : u ≺ v do
7:         for y ∈ Γ(u) : y ≺ v do
8:             set L(y) := 0                        ▷ Reset L(y) for the next iteration
```

A very detailed explanation is already provided by the author [7]. 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 1: Execution Table for Algorithm 8 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 $O(m\bar{\delta}(G))$, the same as the algorithm for the triangle, since $O(m\bar{\delta}(G)) = O(m^{3/2})$ and space $O(n)$, since this is the size of our list L. But instead of just counting the number of 4-cyles, if you want to enumerate them, (Alg:9) is a modified version of it. The time complexity increases to $O(m\bar{\delta}(G) + \#C_4) = O(m^{3/2} + \#C_4)$ and the space complexity remains at $O(n)$. This is due to the use of a linked 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 9 4-cycle enumeration

Require: linked-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(v)
7:     for u ∈ Γ(v) : u ≺ v do
8:         for y ∈ Γ(u) : y ≺ v do
9:             set L(y) := ∅
```

4.2.2 Local Counting Algorithms

Sometimes, we need to count not only the total number of 4-cycles in a graph, but also 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, as for example, with the MapReduce (see 2.4) technique explained previously, of course some adjustment might be required like in triangle algorithms (see 4.1).

Algorithm 10 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) + 1
7:              set L(y) := L'(y)
8:      for u ∈ Γ(v) : u ≺ v do
9:          for y ∈ Γ(u) : y ≺ v do
10:             set ◇(u) := ◇(u) + L(y) − 1
11:             set L'(y) := 0
```

In any case, the author of the article [7] presents an algorithm (Alg:10) still based on the first one (Alg:8), 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 $O(m\bar{\delta}(G))$ 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.

As mentioned at the beginning of this section, it's also possible to do the same kind of counting, but on the edges. To do this, the algorithm remains the same, except that we store the count in a hash table H where each entry represents an edge $\{x, y\}$ and which is sorted on each access.

Algorithm 11 Hash-based Edge-local 4-cycle counting

Require: zero-initialized arrays L', L of size n indexed by V
Require: zero-initialized hash table H of size m

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 $L'(y) := L'(y) + 1$
5: set $L(y) := L'(y)$
6: **for** $u \in \Gamma(v) : u \prec v$ **do**
7: **for** $y \in \Gamma(u) : y \prec v$ **do**
8: set $H(\{v, u\}) := H(\{v, u\}) + L(y) - 1$
9: set $H(\{u, y\}) := H(\{u, y\}) + L(y) - 1$
10: set $L'(y) := 0$
11: **for** $e = xy \in E$ **do**
12: Output $\diamond(xy) = H(\{x, y\})$

In this algorithm, direct counting is removed from the first nested loop, and only the updating of the L and L' lists is retained. It's in the second nested loop (line 6-10) that the H hash table is updated. Counting for the $\{v, u\}$ and $\{u, y\}$ edges is done at the same time, somewhat as before. At the end of the algorithm, the number of 4-cycles per edge is returned. Concerning the time complexity, we could say that it is $O(m\overline{\delta}(G))$ and the space complexity $O(n + m)$. But since we're using a hash table that must be sorted regularly, the complexity can also be $\Omega(n \log n)$ if $\overline{\delta}(G) \ll \log n$.

Since this algorithm doesn't have the same time complexity as all the previous algorithms, another variant exists which doesn't use a hash table H but simply an array M in its place and a second array T. In concrete terms, the array M will act as an edge list, and in order to know where all edges belonging to a vertex x in the direction $\{x, y\}$ 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,

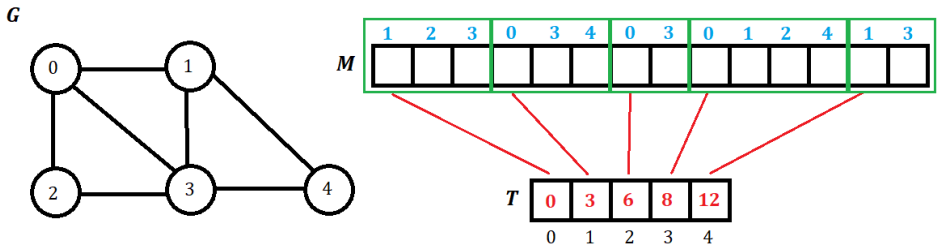


Figure 9: Example of how the array M will be for the graph G

Algorithm 12 Edge-local 4-cycle counting

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

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 $L(y) := L(y) + 1$
5: **for** $i = 0$ to $d_v - 1$ **do**
6: Set u as the ith neighbor of v
7: **for** $j = 0$ to $d_u - 1$ **do**
8: Set y as the jth neighbor of u
9: **if** $u \prec v$ and $y \prec v$ **then**
10: Set $M(T(v) + i) := M(T(v) + i) + L(y) - 1$
11: Set $M(T(u) + j) := M(T(u) + j) + L(y) - 1$
12: Set $L(y) := 0$
13: **for** $x \in V$ **do**
14: **for** $i = 0$ to $d_x - 1$ **do**
15: Set y as the ith neighbor of x
16: **if** $y \prec x$ **then**
17: Find the index j of x in $\Gamma(y)$
18: Set $\diamond(xy) := M(T(x) + i) + M(T(y) + j)$

The way the algorithm works is still the same, except that because of the use of the M and T arrays, the loops on lines 5,7,14 have been adapted. And to respect the total order, an if condition has been added each time. Access to set the counters is quite intuitive, as shown in Figure 9. At the end of the algorithm, to obtain the value of $\diamond(xy)$, the counters of the respective vertices are summed. Since this is only done when $y \prec x$ and there is no longer any sorting, the complexity remains now at $O(m\bar{\delta}(G))$ and for the space complexity is still $O(n + m)$.

5 Objectives

The main theme of my thesis being triangle counting and other types of subgraphs, and if we look at everything that has been presented, we can very easily set the objectives in the following 3 categories. The first is to try to implement the algorithms presented above, in order to analyze their complexity, in particular by empirically testing them on different types of graphs, probably in a low-level language. The second is to explore the different levels of parallelism: we've already discussed MapReduce, but we can obviously talk about parallelization at the level of a single machine. Once again, an implementation and performance evaluation can be carried out. Finally, continue research beyond triangles and 4-cycles by counting other types of subgraphs, developing other points and the implication of these algorithms in real life.

5.1 Algorithmic Implementation And Analysis

When it comes to implementation, the first task to do is of course to choose the programming language. Technically, you could easily implement using any language, as long as you pay attention to the complexity of the various built-in functions associated with the different existing data structures. For example, does it take $O(1)$ to access an element in an array? Or does the "sort" function execute in $\Omega(n \log n)$? Otherwise, you'll have to implement your own functions and data structures.

But where the choice of programming language matters is at what level it belongs. In reality, there aren't really any precise levels: we can speak of so-called low-level languages and so-called high-level languages, but each language is more or less low-level or more or less high-level, depending on the level of abstraction it offers in relation to the hardware on which it runs. A low-level language will allow better memory management and more efficient code execution, but on the other hand will be difficult to program on, since the syntax is basically designed to work at low level. On the other hand, a high-level language won't offer all these advantages, but on the other hand, it will make programming simpler and more multi-platform. For the time being, I haven't yet decided which language I'm going to use, but it's a safe bet that my choice will be C/C++. The reason is simple: it's a language whose syntax is close to that of most other languages, while allowing for more advanced memory manipulation. Finally, it's very popular and already possesses some very well optimized libraries and frameworks in particular for parallelization (see 5.2).

Once the language has been chosen and implemented in various ways. The second task is to analyze what has been done and see how the algorithms behave in the real world.

In fact, one of the first sub-tasks will be to empirically measure the complexity of the implemented algorithms to see if they correspond to the theoretical complexity announced. To do this, we'll need to test with larger and larger instances of the same graph type, i.e. with more and more vertices, and also with more and more edges. After a certain size, it will probably be inconceivable to use a single machine or processor to run the algorithm, so we'll have to parallelize. The second sub-task will be to use different types of graphs, but also with different densities. We'll need to measure the extent to which graph sparsity, in other words arboricity and degeneracy, affect algorithm efficiency. Perhaps a very dense graph will require a longer running time. Another thing to check for the same density is what happens if the graph is generally dense, or if it has several separate dense zones.

All these questions can't be perfectly answered without experimentation, which is why this is the first objective of my thesis.

5.2 Parallel Computing Exploration

Another goal would be to explore parallel programming a little further. This would be both a theoretical and practical (implementation) objective. Indeed, there are different levels of parallelism: Instruction-Level, Thread-Level, Data-Level and Task-Level parallelism. We could therefore adapt certain algorithms or use those designed for parallelism as part of my implementation and measure the impact of parallelism on the efficiency of the algorithms.

Of course, exploring Instruction-Level Parallelism [27] is far too low a level, since it involves using the way instructions are processed by a processor, with the four steps of the instruction pipeline (fetch, decode, execute, write), in order to execute instructions almost simultaneously. These are things already implemented by modern processors, and they don't add anything to the algorithmic aspect of my thesis. However, the other three levels of parallelism are interesting to explore, since they involve the design of algorithms, i.e. algorithms will probably have to be adapted in order to implement parallelism.

In addition to what I briefly covered during this pre-thesis, namely MapReduce, which enables different tasks to be spread out in a distributed system and is therefore part of Task-Level parallelism. If I were to use the C/C++ languages, there are many techniques, libraries and frameworks available for parallel programming, such as OpenMPI [28] or Intel's Instruction Set Extensions [29]. Or if I want to port my implementation to GPUs, Nvidia's CUDA [30] environment can easily accommodate a program written in C/C++ with certain adaptations, since it's based on this language.

All these are interesting paths to explore, both from a theoretical point of view and for my future implementations that I intend to perform during my thesis.

5.3 Generalization And More

A final objective is to extend the scope of my research beyond triangles and 4-cylces to explore other types of subgraphs, such as cliques or stars. This will enable me to analyze other algorithms and implement new ones.

This could lead to a number of things, such as finding the various possible applications for these different subgraphs, the idea being to highlight them. Perhaps some types of subgraph have specific characteristics that make them require algorithms of a certain complexity, and we can't do otherwise, so we could highlight the things they have in common. Finally, we could go even further and look for a generalization on cycles based on the size n, whether from a theoretical or practical point of view, the algorithms grow proportionally with the space or time complexity to that.

All this will enable me to show the impact this can have in the real world, in the context of all applications based on graph-based networks.

6 Conclusion

In the context of this pre-thesis, we have seen that the use of graphs is preponderant in the society that surrounds us. Indeed, as we are constantly connected via social networks on a massive scale, and as the amount of information and people to be managed continues to grow, extracting information from this immense network is becoming increasingly complicated. We've seen that among this information, the way in which groups are formed and the degree to which people are connected is becoming more and more complicated to guess. This is why the use and study of graphs for their exploration has become crucial.

We've started by looking at the fundamentals of what a graph is and the concepts that surround it in the context of group formation around social networks. To do this, we defined a few basic notations. Then we saw that one property of graphs was their density, which could be defined in several different ways, the first being arboricity and the second being average degeneracy. These can be useful for measuring algorithmic complexity. Finally, the last theoretical point purely associated with graphs and social networks was the clustering coefficient, a measure of the degree of interconnectivity between vertices and therefore people, requiring the counting of triangles in graphs. In addition, a very practical tool we saw was the use of the MapReduce programming technique in distributed systems to process large quantities of data and, why not, graphs. All these were the basic keys to building algorithms for understanding cluster formation in social networks.

Then, one of the fundamental operations, enabling cycles and triangle counting was matrix multiplication. A technique that can be used in many counting algorithms by applying it to the adjacency matrix of graphs. We briefly reviewed some of the major techniques that have been discovered for performing this multiplication. Starting with a naive solution, then using the divide-and-conquer methodology, which has led to a degree of complexity improvement over the years. Finally, a solution that has become a field of research in its own right is the use of Tensors through the Laser Method to demonstrate that it is possible to perform multiplication faster by pushing ω the matrix multiplication exponent lower and lower.

We then looked at various algorithms for counting the number of triangles and 4-cylces in a graph. A whole range of different algorithms, some more complex than others, with progressively reduced complexity. Some dedicated to local counting, others to more general counting and finally some designed to be parallelized. Of course, we haven't finished going through them all yet, since there must be thousands of them, not to mention the other subgraph types we haven't explored yet. One thing's for sure: there's still a lot to do for the final thesis.

Exactly, and since this is a pre-thesis and there's still a lot to do, I've set myself the following 3 objectives: Namely, to implement the algorithms I've presented in order to study them, to explore parallel programming in order to gain in performance, and finally to try to go further with new algorithms and new theoretical research in order to draw further conclusions.

In conclusion, this pre-thesis allowed me to do two things, the first being to build the foundations of what will become my thesis. The second, and more important in my opinion, is that it taught me how to carry out a research project, how to find and process information that is useful to me. It awakened my curiosity to go further and further. Of course, there's still a long way to go, but I'm convinced that I'll get there.

7 Bibliography

Main Articles

[7] 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.

[8] 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.

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

[24] 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.

Articles

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

[2] 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.

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

[5] 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.

[6] 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.

[9] 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.

[10] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 2004, pp. 137–150. URL: https://static.googleusercontent.com/media/research.google.com/fr//archive/mapreduce-osdi04.pdf.

[16] 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.

[17] 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.

[18] 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.

[20] 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.

[23] 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.

[25] Kai Wang et al. “Accelerated butterfly counting with vertex priority on bipartite graphs”. In: *The VLDB Journal* 32.2 (2023), pp. 257–281. ISSN: 0949-877X. DOI: 10.1007/s00778-022-00746-0. URL: https://doi.org/10.1007/s00778-022-00746-0.

[26] Alexander Zhou, Yue Wang, and Lei Chen. “Butterfly Counting on Uncertain Bipartite Graphs”. In: *Proc. VLDB Endow.* 15.2 (Oct. 2021), pp. 211–223. ISSN: 2150-8097. DOI: 10.14778/3489496.3489502. URL: https://doi.org/10.14778/3489496.3489502.

Books

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

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

Websites

[12] 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).

[13] 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).

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

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

[21] 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).

[27] Sergey Slotin. *Instruction-Level Parallelism*. Algorithmica. URL: https://en.algorithmica.org/hpc/pipelining/ (visited on 08/06/2023).

Frameworks And Tools

[11] Apache Software Foundation. *Apache Hadoop*. URL: https://hadoop.apache.org/ (visited on 08/06/2023).

[28] The Open MPI Development Team. *OpenMPI*. URL: https://www.open-mpi.org/ (visited on 08/06/2023).

[29] 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).

[30] Nvidia. *Nvidia CUDA Toolkit*. URL: https://developer.nvidia.com/cuda-toolkit (visited on 08/06/2023).

Images

[31] Alhussein Fawzi et al. *FIGURE 8: 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.