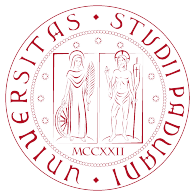

Università degli Studi di Padova

Department of Information Engineering
University of Padova

*Master's Degree in Computer Engineering*

# Algorithms with Predictions for Triangle Counting in Temporal Networks


Candidate:

**Giorgio Venturin**

Supervisor:

Prof. **Fabio Vandin**



September 7, 2023
Department of Information Engineering
University of Padova
Padova, Italy
Academic Year 2022/2023

# Abstract

In recent years, there have been major efforts in refining existing algorithms by considering problem instances that are likely to appear, while maintaining a formal and rigorous methodology as in time and space complexity analysis. This approach exploits Machine Learning to extract additional information from the data received in input and predict useful properties of the considered instances that can be used to improve the algorithm performance. Such an approach is referred to as "Algorithms with Predictions". This work aims to apply this framework to the Motif Counting Problem in large Temporal Networks, focusing on improving standard sampling algorithms techniques and providing guarantees on the results obtained. In particular, we design and analyze the first algorithm for counting triangles in temporal networks using machine learning predictions. We also perform an in-depth experimental evaluation of the algorithm presented, in order to gather evidence on its practical effectiveness and efficiency.

# Sommario

Negli ultimi anni, ci sono stati grandi sforzi nel perfezionare algoritmi preesistenti considerando istanze di problemi che abbiano una alta probabilità di presentarsi, mantenendo al contempo una metodologia formale e rigorosa come nell'analisi della complessità temporale e spaziale. Questo approccio sfrutta il Machine Learning per estrarre informazioni aggiuntive dai dati ricevuti in input e prevedere proprietà utili delle istanze considerate che possano essere utilizzate per migliorare le prestazioni dell'algoritmo. Tale approccio è definito "Algorithms with Predictions". Questo lavoro mira ad applicare il suddetto approccio al problema del conteggio di motivi in grandi reti temporali, concentrandosi sull'estensione e miglioramento degli algoritmi con campionamento e fornendo garanzie sui risultati ottenuti. In particolare, si è progettato ed analizzato il primo algoritmo per il conteggio di triangoli in reti temporali usando predizioni basate su machine learning. Si è inoltre provveduto a realizzare una dettagliata valutazione sperimentale dell'algoritmo proposto, in modo da dimostrarne la reale efficienza ed efficacia.

# Contents

# Chapter 1

# Introduction

In the last few decades, because of the so-called Big Data phenomenon, the advances in the Information Technology field and the breakthrough results obtained with statistical analysis, several scientific domains have been trying to develop automatic tools to effectively analyze huge amount of data. As a result, due to the increasing complexity of the information to be represented, specific kinds of representation structures are witnessing a growing interest. One of such structures are networks. It is important to note that the formal model to represent networks is the graph structure, which is defined by means of nodes (or vertices) and edges. However, by using the word "network", we can better emphasize the importance that the relations among entities have with respect to the entities themselves. As mentioned above, networks are becoming essential in multiple scientific disciplines [2]. In biology and in chemistry, networks are used to represent molecules, proteins or even networks of genes [17]. In social networks analysis, they are exploited to model relationships between users and to find communities [20]. Recently, networks have also been applied to urban traffic prediction [5], blockchains analysis [9], recommendation systems [24], mapping services [4] and many other areas.

A crucial step to understand and extract new useful information from network-

encoded data consists in finding recurrent and statistically significant patterns, which may reveal domain-specific structures that were not expected to appear. Such patterns, in the context of networks, are (small) sub-graphs named *motifs*. Generally speaking, motif mining is considered one of the most important primitives in network analysis and has proven to be effective in capturing subnetworks' functions and structures [11, 12, 21]. As a consequence, several problems related to motifs exist, such as the *Motif Counting Problem*, for which multiple algorithms have already been developed [1, 23]. The majority of such techniques focuses on static networks, which, as the name suggests, are not subject to changes. In other words, the number of nodes and edges of the graph considered is immutable. However, with static networks, we may lose rich information when modeling specific scenarios, as for example time information. To this end, *temporal networks* have been introduced. The major difference with respect to static networks is that each edge has an associated timestamp at which the edge appeared. Because of the additional information introduced, motif-related problems become much more complex. Indeed, the Motif Counting Problem in temporal networks is NP-Hard when considering its general formulation. Still, it is possible to tackle this problem, and multiple solutions have been proposed. In [15], Paranjape et al. present efficient algorithms to count exactly the occurrences of temporal motifs in a temporal network. Of particular interest is the counting of temporal triangles, since they represent one of the simplest form of clustering and are also building blocks of more complex patterns. Ad-hoc algorithms have been designed for counting temporal triangles, such as in [16].

Unfortunately, two important issues arise when trying to scale these approaches to really large networks. First of all, the entire network may be too large to fit in a computer memory, and therefore we cannot assume to have access to all the nodes and edges at the same time. A second issue is that an exact computation for a large network may become intractable with standard algorithms. This is why several studies have focused on solving the Motif Counting Problem for large networks by providing an approximation of the motif counts instead of the exact counts. There are examples of such approaches for both static [8] and temporal networks [18, 19, 22]. The main common technique used within these approaches is *sampling*: each element of the network is stored or considered according to a given probability. This reduces the computational burden and the memory requirements

at the cost of obtaining only an estimate of the true count.

Nevertheless, one of the biggest drawback of classical algorithms, including sampling algorithms, is that they cannot capture any additional detail on the data given in input. Roughly speaking, algorithms are blind with respect to the data they are provided with. A recent framework, known as *"Algorithms with Predictions"* (AwP), aims at extracting useful information from the input instances using Machine Learning, so to improve the effectiveness of standard algorithms. The AwP framework was first introduced by Mitzenmacher and Vassilvitskii in [13]. The original idea was to enhance standard algorithms such that the worst case analysis of a given algorithm remains unchanged while, on average, the majority of the problem instances (i.e., likely instances) could be solved more efficiently exploiting Machine Learning predictions. Basically, a specific Machine Learning model is coupled with a standard algorithm to gather some additional knowledge that would be otherwise lost. Several variants to this first formulation have been proposed, but the core idea remains the same: extract statistical information from the data to enhance the performance of classical algorithms.

In [3], Chen et al. developed sampling based algorithms for triangle and four cycle counting in large static networks exploiting the AwP framework, with notable results. To the best of our knowledge, similar approaches have not yet been applied to temporal networks. The aim of this work is to design an algorithm for triangle counting in large temporal networks within the AwP framework, provide guarantees on the results obtained and perform experimental evaluations on the devised approach.

## 1.1 Related Work

In this section, we review the main works related to this thesis. In subsection 1.1.1, we analyze the problem addressed, namely, the Motif Counting Problem (MCP) in Temporal Networks, and present some representative results. Approximate solutions to the MCP are then presented in subsection 1.1.2. Finally, in subsection 1.1.3, we describe the Algorithm with Predictions framework together with its application to a specific instance of the MCP we are interested in, that is, the *Triangle Counting Problem* (TCP).

### 1.1.1 Motif Counting Problem in Temporal Networks

When studying networks, motifs are a key aspect to consider, since they allow to characterize a given network by means of its simplest components. A crucial problem, in this sense, is to quantify how many of such motifs occur within a network. The count is usually done for each specific type of motif, in order to obtain meaningful results. Such problem is known as the Motif Counting Problem, which has been deeply explored in the context of static networks. However, in the case of temporal networks, several additional challenges emerge. As an example, in undirected static networks, a motif is uniquely identified given the number of nodes, the number of edges and the topology of the motif, that is, how the nodes are connected one to each other. In a directed static network, we would also need to pay attention to the direction of each edge involved. In the case of temporal motifs we have a further additional information to account for, which is time. As a consequence, the number of possible motifs to look for in a temporal network increases rapidly. Indeed, a static and undirected triangular motif translates in eight different configurations of triangles in a temporal network. It is clear that, when considering larger motifs, the problem becomes even harder. Another additional variable to consider is about the total duration of a motif, that represent the total time elapsed between the first and the last temporal edge of the motif. In this sense, a motif that has a duration of 10 seconds may be treated different from the same motif with a duration of 3 days. Often, in practice, an upper bound to the total duration is set, meaning that all the motifs that have a duration lower than the upper bound are considered. Because of these major issues, several new notions had to be introduced and many novel algorithms developed.

One of the first studies that proposed a solution to the Motif Counting Problem in temporal networks is presented in [15], which proposes several algorithms leveraging subgraph enumeration for exactly count all the motifs present in a temporal network, with a major focus in 2- and 3-node, 3-edge temporal motifs counting. In the same study, a first formal definition of *temporal motifs* is provided, in particular by introducing the notion of $\delta$-instance of a motif. The study also shows that the analysis of temporal motifs can be used to reveal important structural patterns characterizing a temporal network. A more recent work that explored the problem of temporal motif counting was presented by Pashanasangi et al. [16], in which they exploit the concept of degeneracy ordering to exactly count temporal triangles

in a temporal network. The technique devised brought consistent improvements in terms of efficiency and hardware requirements.

Unfortunately, an exact computation may become infeasible for really large temporal networks. Because of this main limitation, new extremely efficient and fast solutions to the Motif Counting Problem have been proposed in the literature, at the cost of providing only an approximate solution.

### 1.1.2 Approximating Motif Counts in Large Networks

As aforementioned, when dealing with large temporal networks, exact algorithms may fail in provide a solution to the Motif Counting Problem in a reasonable amount of time and resources, even for really small motifs. To address this issue, approximate solutions to the problem have been explored. In particular, most of them makes use of a simple yet effective technique, named *sampling*. The main idea is the following: retain an element of the network considered (for instance, an edge) according to a given probability. Because of the bias introduced with the sampling, additional corrections to the values computed must be added. Such correction factors can be derived directly from the designed algorithm. Indeed, these approaches often provide guarantees on the results obtained, by means of analysis on the expected value and eventually on the variance of the quantities considered.

Two representative works involving sampling techniques applied to the Motif Counting Problem in temporal networks are presented in [19] and in [18]. In the first one, Sarpe and Vandin present an efficient and scalable sampling algorithm to obtain approximate solutions to the motif counting problem in temporal networks, by also providing guarantees on the obtained estimates. The algorithm has been designed to work for all temporal motifs, without any constraint. In the second study, the major focus was put in solving the temporal motif counting problem for multiple motifs simultaneously, without requiring multiple and time-consuming algorithm executions. The work also suggests important considerations regarding the probability distribution that is used in the sampling process, stating that a data-driven distribution may improve the algorithm's performance. This aspect is of crucial importance within this work, since it is one of the core idea of the "Algorithm with Predictions" framework, which will be better discussed in the following section.

In [22], Wang et al. exploited sampling to solve the Motif and Triangle Counting Problems (i.e., count the number of triangles appearing in a temporal network) in a temporal network, by leveraging also the concept of wedge, a commonly adopted strategy when dealing with triangles. In particular, they devised two different algorithms: one that can deal with any temporal motif and one specifically designed for counting temporal motifs with 3 nodes and 3 edges. The approach was also extended to temporal networks received in a streaming fashion, which is a very common scenario when considering large networks. An interesting idea that was introduced in this work is about the extension of the sampling technique to the network's wedges. Instead of retaining all the possible wedges built from a given set of edges, only a subset of them is maintained, by means of sampling.

One of the main drawbacks of these kind of algorithms is that the probability distribution used to perform the sampling is not guaranteed to be optimal and it may exists a different distribution with which we would get a better estimate. In other words, the sampling probability is independent of the input data the algorithm is provided. To this end, a possible improvement can be achieved by extracting some additional information from the data in order to perform an informed sampling and retain the most important element of the network for our specific problem. A possible way to extract such information is to learn some peculiar characteristics of the input instances, using Machine Learning techniques. This methodology, which couples an algorithm with a machine learning predictor, is known as "Algorithms with Predictions", which is presented right below.

### 1.1.3 Algorithms with Predictions

An important limitation of classical algorithms is that they do not have any knowledge on the problem instances that they solve. This is due to the fact that any instance is treated blindly and independently from the other ones. However, there may exist some information embedded in the data that can be extracted to better solve the problem considered. In [13], a framework named "Algorithms with Predictions" (AwP) is presented: it is intended to address this limitation by exploiting Machine Learning models so to capture statistical information of the input instances provided to the classical algorithm and improve its performance. The goal is to keep the complexities of the worst case scenario unchanged, while improving the effectiveness and efficiency on the most likely instances. In order to

achieve this objective, the algorithm should also be robust against wrong predictions. In this first work, Mitzenmacher and Vassilvitskii propose several extension to standard algorithms. In the first example presented, they show how a simple binary search can be improved by incorporating specific predictions provided by a machine learning model. The enhanced version of the binary search is then also theoretically analyzed in terms of time complexity, by providing a formal proof of the improvement obtained. A key concept that emerges from this first example is that the worst case scenario for the standard binary search is preserved, even in the case of really bad predictors. As stated above, this highlights the fact that an algorithm developed in the AwP framework should be robust against erroneous predictions. A second interesting example is about the problem of counting sketches for data streams, which we will discuss soon. Further applications are then explored, such as learning bloom filters, improve caching with predictions and perform scheduling using predicted job times. Because of the huge amount of possible applications of this framework, a lot of researches have started to explore it. Several variants of the original approach exists, but we will try to stick to the original formulation.

In [7], Hsu et al. proposed a method for solving the frequency estimation problem in data streams by leveraging learning-based algorithms, with consistent improvements on previous works using standard algorithms. The work recalls the concept of bloom filters, previously cited. The main idea is to learn an Oracle that predicts a specific property of an input instance or of its components, which, in this case, are the elements of the data stream. In such scenarios, a widely used property is the *"heaviness"* of an element, which allows to understand which are the most important items in the data stream. In the context of frequency estimation, the heaviness property is strictly related to the frequency with which an element appears in the stream. This property is extremely useful, since it allows, when using "hashing-based" algorithms, to reduce the estimation errors derived from collisions. In order to be able to classify a given item as heavy or not, a specifically designed predictor is learned. In this study, the model was based on Recurrent Neural Networks, a *Deep Learning* architecture that allows to capture causal information from the data distribution given in input. The results show significant improvements over classical approaches, providing additional evidence of the effectiveness of the AwP framework. A notable aspect that is mentioned in this work is that the Oracle model allows to analyze the potential improvements that a predictor can bring

to a standard algorithm without actually requiring a machine learning predictor. Because the word Oracle may be perceived as ambiguous, it is important to state that it is only a common way to model a general predictor. A deeper explanation of the Oracle will be given in section 2.3.

The AwP framework was also applied to Clustering problems (specifically, in the context of K-Means and K-Medians clustering) with interesting results [6]. The major difference with respect to other approaches is that the clustering is performed starting from an a priori, per-element labeling, which are provided by a predictor. Several different models have been tested, as for example Nearest Neighbors and Artificial Neural Networks. An important aspect that is reported is about the *learnability* of the predictor for the considered task. Ergun et al. give formal learning bounds for learning a good predictor exploiting the *Probably Approximately Correct* (PAC) learning framework. It is indeed a crucial detail to be considered. A subsequent study devised some variants of the algorithms proposed in [6], which are more robust against wrong predictions of the labels [14].

One last important example is provided in the work done by Chen et al. [3], where a learning-based sampling algorithm is used to solve the triangle and four cycle Counting Problem in static networks. Similarly to the studies cited above, the goal of this work is to improve a standard sampling algorithm for motif counting by exploiting an Oracle that predicts useful properties of the input instances. The algorithms presented were differentiated in order to deal with both adjacency list and data streaming models, with the main goal of improving over classical algorithms for what concerns space and time complexities. All the algorithms designed in this study are 1-pass algorithms, meaning that they need to receive in input the entire network only once. Several models for the Oracle have been proposed, including noisy Oracle models, that were used to capture possible behaviours of real predictors without actually requiring to design or train them. Another key point to account for is the property to be used for enhancing the algorithms' performance. Since the input network is received edge by edge, a useful information that can be captured from the data is the amount of triangles (or four cycles) an edge is involved in, which is reformulated as the "heaviness" property of an edge, similarly to what was done in the case of the frequency estimation problem in [7]. Multiple Machine Learning models were tested in order to build a real predictor, as for

example *Linear Regression* and *Graph Neural Networks*. Basic approaches such as building a look-up table with the top 10% heaviest edges were also used and were found to be effective as well. As previously mentioned, this kind of models also allows to gather some insights on the potential improvements that an algorithm can receive when coupled with a predictor. From the experimental evaluations presented, we can see that this approach allows to collect estimates that are more accurate or comparable with respect to the state of the art sampling algorithms, while maintaining a high level of efficiency.

Because of the improvements brought by the AwP framework, we decided to apply it to the Motif Counting Problem in temporal networks, which, as far as we can tell, is an interesting and challenging scenario that has not yet been explored.

## 1.2 Contributions

The main contributions of this work are the following:

- We provided an algorithm inspired by the work from Chen et. al. [3] in the context of large temporal networks, considering a streaming setting.
- We developed our algorithm incrementally, by providing three different algorithms for solving the Triangle Counting Problem leveraging the concept of temporal wedge. The first algorithm computes the counts exactly, the second one exploit standard sampling techniques, while the last one makes use of predictions by means of an Oracle.
- We formally proved the unbiasness of the estimates for both the standard sampling algorithm and for the algorithm with predictions presented.
- We performed an in-depth experimental evaluation of the algorithms proposed and provided empirical comparisons between them, proving the effectiveness of our approach.

## 1.3 Roadmap

This thesis is organized as follows. In Chapter 2, we provide the necessary preliminaries for the problem considered, including temporal networks, motif counting,

and algorithm with predictions. Chapter 3 presents three algorithms to solve the Triangle Counting Problem in temporal networks, with additional details on the subroutines used. In Chapter 4 we derive proofs of unbiasness for the standard sampling algorithm and for the algorithm with predictions presented. Finally, in Chapter 5, we present experimental evaluations and comparisons on the algorithms proposed, focusing on the algorithm developed using the AwP framework.

# Chapter 2

# Preliminaries

This chapter introduces the main concepts underlying this work. Section 2.1 describes the basic notions related to Temporal Networks. In section 2.2 we present the Motif Counting Problem and the Triangle Counting Problem in the case of Temporal Networks. We then revise the Algorithm with Predictions framework and its instantiation for our problem in section 2.3. Section 2.4 provides a list of the main notation used in this work. Additional symbols not hereby included will be contextually introduced.

## 2.1 Temporal Networks

In this section we will define the notion of temporal graph and temporal wedge, mainly following what is presented in [22], together with some additional concepts that will be used in the next sections.

**Definition 1.** *A temporal graph $T = (V_T, E_T)$ is defined by a set of vertices $V_T$ and a set of edges $E_T$. A temporal edge $e = (u, v, t) \in E_T$, where $u, v \in V_T$ and $t \in \mathbb{R}^+$, is a directed edge from $u$ to $v$ with associated timestamp $t$.*

We will assume that each temporal edge $e \in E_T$ has a unique timestamp $t$, that is: $\nexists\ (e_1 = (u_1, v_1, t_1), e_2 = (u_2, v_2, t_2)) \in E_T \times E_T : t_1 = t_2$. This is not a strict

limitation, and, in practice, is a reasonable assumption.

We now define the concept of temporal wedge.

**Definition 2.** *A temporal wedge $w$ is defined by an ordered pair of edges $\langle e_1 = (u_1, v_1, t_1),\ e_2 = (u_2, v_2, t_2)\rangle$, $e_1, e_2 \in E_T$, with the additional constraint that a couple of nodes belonging to different edges must represent the same node in the temporal graph $T$, that is, one of the following conditions is true:*

1. $u_1 = u_2$ *and* $v_1 \neq v_2$;
2. $u_1 = v_2$ *and* $u_2 \neq v_1$;
3. $u_2 = v_1$ *and* $u_1 \neq v_2$;
4. $v_1 = v_2$ *and* $u_1 \neq u_2$.

Note that the definition of wedge considers edges that belongs to a temporal graph $T$. An alternative (and possibly more general) definition may consider a wedge as a temporal motif $M = \langle (u_1, v_1), (u_2, v_2)\rangle$, but we will stick to the above definition.

In this work, we will focus on solving the Temporal Motif Counting problem in a *streaming* setting, where each temporal edge $e = (u, v, t)$ is received only if the current time-step $i$ is greater or equal to the timestamp of the edge, i.e. $i \geq t$ (the edge cannot be received ahead of time). We also assume that the stream of edges is temporarily ordered: if $e_1 = (u_1, v_1, t_1)$ comes before $e_2 = (u_2, v_2, t_2)$ in the stream, it must hold that $t_1 < t_2$ (where the strict inequality holds since we assumed that each temporal edge as an unique timestamp). All the algorithms that we will provide are 1-pass algorithms, meaning that the temporal network is received in input only once and no other additional passes over the data are allowed. We will denote the data stream of a temporal graph $T$ with the symbol $\tau$, from which we receive a temporally ordered sequence of edges.

## 2.2 Motif and Triangle Counting Problems

In this section we will define the notions of temporal motif, motif instances and temporal motif counting problem.

**Definition 3.** *A temporal motif $M = (V_M, E_M, \sigma)$ is a weakly connected graph defined by a set of $k$ vertices $V_M$, a set of $\ell$ edges $E_M$ and an ordering $\sigma$ on the edges in $E_M$.*

Note that a temporal motif $M$ can be represented as an ordered sequence of $\ell$ couples $(x, y)$ with $x, y \in V_M$: $\langle (x_1, y_1), (x_2, y_2), ..., (x_\ell, y_\ell) \rangle$.

**Definition 4.** *A sequence of $\ell$ temporal edges $S = \langle (u_1, v_1, t_1), (u_2, v_2, t_2), ..., (u_\ell, v_\ell, t_\ell) \rangle$ with $t_1 < t_2 < ... < t_\ell$ from a temporal graph $T$ is a $\delta$-instance of a temporal motif $M = \langle (x_1, y_1), (x_2, y_2), ..., (x_\ell, y_\ell) \rangle$ if:*

*1) $\exists$ bijection $f : f(u_i) = x_i, f(v_i) = y_i$ for $i \in 1, ..., \ell$;*

*2) The total duration is at most $\delta : t_\ell - t_1 < \delta$.*

**Definition 5.** *Temporal Motif Counting problem: given a temporal graph $T$, a temporal motif $M$ and a time span $\delta$, output the number of $\delta$-instances $C_M$ of $M$ that appear in $T$.*

We will first focus our work on a specific type of temporal motif, which will refer to as temporal 3-cycle.

**Definition 6.** *A temporal 3-cycle is a temporal motif $M = (V_M, E_M, \sigma)$ of the form $M = \langle (x_1, x_2), (x_2, x_3), (x_3, x_1) \rangle$ where $(x_1, x_2), (x_2, x_3), (x_3, x_1) \in E_M$.*

We will also denote with $\mathcal{C}$ the set of all temporal 3-cycles appearing in a given temporal graph $T$, and with $N_C = |\mathcal{C}|$ its cardinality.

A temporal 3-cycle is, in turn, a particular kind of temporal triangle, that we will hereby define.

**Definition 7.** *A temporal triangle is a temporal motif $M = (V_M, E_M, \sigma)$ of the form $M = \langle (u_1, v_1), (u_2, v_2), (u_3, v_3) \rangle$ with $(u_1, v_1), (u_2, v_2), (u_3, v_3) \in E_M$ such that one of the following conditions is true:*

1. $u_1 = u_3$, $v_1 = v_2$, $u_2 = v_3$;
2. $u_1 = v_3$, $v_1 = v_2$, $u_2 = u_3$;
3. $u_1 = u_3$, $v_1 = u_2$, $v_2 = v_3$;
4. $v_1 = u_2$, $v_2 = u_3$, $v_3 = u_1$;

5. $u_1 = u_2$, $v_1 = v_3$, $v_2 = u_3$;

6. $u_1 = v_2$, $v_1 = v_3$, $u_2 = u_3$;

7. $u_1 = u_2$, $v_1 = u_3$, $v_2 = v_3$;

8. $u_1 = v_2$, $v_1 = u_3$, $v_3 = u_2$.

Following the definition of temporal triangle, we can derive the temporal 3-cycle motif defined above by setting $u_1 = v_3, v_1 = u_2$ and $v_2 = u_3$, which corresponds to the fourth condition expressed above. Additionally, we can see that there are a total of eight different temporal triangles. It is important to state that we are hereby considering temporal triangles only: if we let three nodes correspond to the same unique node, we would also take into account different motifs, commonly referred to as temporal stars.

## 2.3 Algorithm with Predictions

As presented in section 1.1.3., the "Algorithms with Predictions" framework makes use of a predictor coupled with a standard algorithm to improve its performance. In our case, we are interested in extending a sampling algorithm with a predictor. Our goal is to perform an informed sampling exploiting an Oracle which is predicting the *heaviness* property of an edge. We first consider the case of temporal 3-cycles only, but we will then present an extension to temporal triangles in section 3.5.

We will define an edge $e$ as *heavy* if, for a given threshold $\rho \geq 0$, the number $\eta(e)$ of temporal 3-cycles in which $e$ is involved is greater or equal than the given threshold $\rho$, that is, $\eta(e) \geq \rho$. We will also define the set of all edges $e \in E_T$ that are heavy, given a threshold $\rho$, as:

$$C_H = \{e \mid \eta(e) \geq \rho\}.$$

We will instead denote as $C_L$ the set of edges that do not satisfy the heaviness property for the same threshold $\rho$:

$$C_L = \{e \mid \eta(e) < \rho\}.$$

We will refer to edges in $C_L$ also as "light" edges. Note that $C_H$ and $C_L$ are complementary sets with respect to $E_T$. Indeed, we have $E_T = C_H \cup C_L$. Let us now define the concept of perfect oracle in the case of temporal 3-cycle motif.

**Definition 8.** *A perfect oracle $O_\rho$ is a function $O_\rho : E_T \to \{0, 1\}$ that, given an edge $e \in E_T$ and a threshold $\rho \geq 0$, perfectly predicts whether the edge is involved in at least $\rho$ temporal 3-cycles. Denoting $\eta(e)$ as the number of temporal 3-cycles in which an edge $e$ is involved, we can define the oracle as follows:*

$$O_\rho(e) = \begin{cases} 1 & \text{if } \eta(e) \geq \rho \\ 0 & \text{if } \eta(e) < \rho \end{cases}$$

An extension of this Oracle for temporal triangles is provided in section 3.5.

Note that a perfect Oracle can be derived practically exploiting an exact algorithm for temporal 3-cycles (or triangles) counting, by computing the number of 3-cycles in which each edge is involved. Further details on this process are presented in section 5.3.

It is also important to state that a perfect Oracle always predicts correctly the heaviness property of an edge. On the contrary, a real predictor is far from perfect, since its predictions can be wrong. We will make use of a simple technique to simulate a real predictor which consists in *corrupting* a perfect Oracle and modify its knowledge about edges' heaviness. In this way, we can evaluate several goodness levels of an arbitrary Machine Learning predictor without actually requiring an instance of such a predictor. This technique is also useful to gather some insights about upper and lower bounds on the algorithm performance. Specifically, the perfect Oracle represents the best possible scenario, while a heavily corrupted Oracle can be used to understand how the algorithm behaves in the worst cases. More details on the corruption techniques used will be presented in section 5.6.2.

## 2.4 Notation

We now provide a list of symbols with their corresponding meaning used in the algorithms that follow. Additional symbols not included in this list will be introduced and explained contextually when needed.

| Symbol | Description |
|---|---|
| $\tau$ | Temporal graph stream associated to temporal graph $T$ |
| $E[t-\delta, t]$ | Set of edges with timestamps within the interval $[t-\delta, t]$ |
| $W$ | Set of wedges (without any distinction) |
| $S$ | Set of sampled edges |
| $H$ | Set of heavy edges with timestamps within the interval $[t-\delta, t]$ currently acquired from the stream |
| $S_L$ | Set of sampled light edges with timestamps within the interval $[t-\delta, t]$ |
| $p$ | Light edge sampling probability |
| $W_{H,H}$ | Set of wedges where each wedge is formed by two heavy edges |
| $W_{H,S_L}$ | Set of wedges where each wedge is formed by an heavy edge and a light edge |
| $W_{S_L,S_L}$ | Set of wedges where each wedge is formed by two light edges |
| $\ell_1$ | Counter of 3-cycles when considering the current edge in the stream and a wedge from $W_{S_L,S_L}$ |
| $\ell_2$ | Counter of 3-cycles when considering the current edge in the stream and a wedge from $W_{H,S_L}$ |
| $\ell_3$ | Counter of 3-cycles when considering the current edge in the stream and a wedge from $W_{H,H}$ |

Table 2.1: Notation symbols.

# Chapter 3

# Algorithms for Triangle Counting in Temporal Networks

In this chapter, we present the three algorithms developed to solve the Triangle Counting Problem, first considering only temporal 3-cycles and then extending all of them to the general case of temporal triangles.

## 3.1 Exact Algorithm

The first algorithm presented computes the exact count of temporal 3-cycles appearing in an given temporal network, named COUNT_CYCLES_EXACT algorithm. It receives in input a temporal graph stream and a time span $\delta$ representing the total duration of the 3-cycles we are interested in. It then outputs the number $\tau_\ell$ of $\delta$ instances of temporal 3-cycle found in the graph stream. Note that, because this algorithm is exact, it requires to store all the edges seen in the stream in the time interval $[t - \delta, t]$. For an edge $e$ received in input from the graph stream at time $t$ (i.e., the edge has timestamp $t$), we first collect all possible wedges that appeared within the time interval $[t - \delta, t]$ using the COLLECT_WEDGES algorithm (see section 3.4 for further details). Then, for each wedge $w$, we check if, together with

$e$, it forms a 3-cycle using the CHECK_CYCLE algorithm (again, see section 3.4).

We can hereby see that the algorithm time complexity mainly depends on the COLLECT_WEDGES procedure and, subsequently, on the cardinality of the set $E[t-\delta, t]$. If we let $n$ be the maximum cardinality of the set $E[t-\delta, t]$, the COLLECT_WEDGES algorithm has an upper bound for what concerns time complexity of $O(n^2)$. Assuming that the CHECK_CYCLE algorithm has time complexity $O(1)$ and because the maximum cardinality over time $t$ of the set $W$ is less or equal than $n^2$, we end up with an overall upper-bound for time complexity for the COUNT_CYCLES_EXACT algorithm of $O(|E_T| \cdot n^2)$, which is linear with respect to the number of temporal edges of the network. Thus, the dominant factor of the time complexity of this algorithm is related to the cardinality of the set $E[t-\delta, t]$. We therefore expect that, for specific values of $\delta$, larger networks may be tackled faster than smaller networks.

**Algorithm 1** COUNT_CYCLES_EXACT($\tau$, $\delta$)

**Input**: Temporal graph stream $\tau$, time span $\delta$.

**Output**: Number $\ell$ of $\delta$ instances of temporal 3-cycle in $\tau$.

```
1: ℓ ← 0;
2: for each e = (u, v, t) ∈ τ do
3:     W ← COLLECT_WEDGES(E[t − δ, t], E[t − δ, t]);
4:     for each w = ⟨e1, e2⟩ ∈ W do
5:         if CHECK_CYCLE(w, e) = True then
6:             ℓ ← ℓ + 1;
7: return ℓ;
```

## 3.2 Sampling Algorithm

We now provide a sampling algorithm to perform an estimation of the number of temporal cycles appearing in a temporal graph stream. It makes use of a simple sampling probability $p$. The COUNT_CYCLES_SAMPLING algorithm performs similar operations with respect to the exact algorithm (COUNT_CYCLES_EXACT). The main difference relies in the set of edges from which we are collecting the wedges. Indeed, in this case we are maintaining only a subset $S$ of $E[t-\delta, t]$, since an edge is stored only with probability $p$. The final result is divided by $p^2$ in order to obtain an unbiased estimate. See Chapter 4 for additional details.

**Algorithm 2** COUNT_CYCLES_SAMPLING($\tau$, $\delta$, $p$)

**Input**: Temporal graph stream $\tau$, time span $\delta$, edge sampling probability $p$.

**Output**: Estimate $\ell$ of the number of $\delta$ instances of temporal 3-cycle in $\tau$.

```
1: ℓ ← 0;
2: S ← ∅;
3: for each e = (u, v, t) ∈ τ do
4:     Remove from S the edges with timestamp smaller than t − δ;
5:     Toss a biased coin with success probability p;
6:     if success then
7:         S ← S ∪ {e};
8:     W ← COLLECT_WEDGES(S, S);
9:     for each w ∈ W do
10:        if CHECK_CYCLE(w, e) = True then
11:            ℓ ← ℓ + 1;
12: return ℓ/p²;
```

The COUNT_CYCLES_SAMPLING algorithm performs similar operations with respect to the exact algorithm (COUNT_CYCLES_EXACT). The main difference relies in the set of edges from which we are collecting the wedges. Indeed, in this case we are maintaining only a subset $S$ of $E[t-\delta, t]$, since an edge is stored only with probability $p$. The final result is divided by $p^2$ in order to obtain an unbiased estimate. See Chapter 4 for additional details.

As before, the time complexity of the algorithm is strictly dependent on the COLLECT_WEDGES subprocedure. However, this time, the computation is done over the set $S$, which comprises less edges with respect to the set $E[t-\delta, t]$. In particular, if we let again $n = \max_t |E[t-\delta, t]|$ be the maximum cardinality of the set of considered edges over time $t$, the corresponding maximum cardinality of the set $S$ would be, in expectation, equal to $|S| = p \cdot n$ where $p$ is the (constant) sampling probability. The COLLECT_WEDGES procedure would then have an upper bound of $O(n^2 \cdot p^2)$. For similar reasons explained in the case of the exact algorithm, we neglect the CHECK_CYCLE contributions to the time complexity. Overall, the upper bound for time complexity of the COUNT_CYCLES_SAMPLING algorithm is $O(|E_T| \cdot p^2 \cdot n^2)$, which, for $p < 1$, is lower than the one obtained for the exact algorithm, as we would expect.

## 3.3 Algorithm with Predictions

Finally, we present an extension of the standard sampling algorithm previously discussed within the AwP framework.

**Algorithm 3** COUNT_CYCLES_ORACLE($\tau$, $\delta$, $O_\rho$, $\rho$, $p$)

**Input**: Temporal graph stream $\tau$, time span $\delta$, oracle $O_\rho$ with associated threshold $\rho$, light edge sampling probability $p$.

**Output**: Estimate $\ell$ of the number of $\delta$ instances of temporal 3-cycle in $\tau$.

$H \leftarrow \varnothing$; $S_L \leftarrow \varnothing$;
$\ell_1 \leftarrow 0$; $\ell_2 \leftarrow 0$; $\ell_3 \leftarrow 0$;
**for** each $e = (u, v, t) \in \tau$ **do**
    Remove from $H$ and $S_L$ the edges with timestamp smaller than $t - \delta$;
    **if** $O_\rho(e) = 1$ **then**
        $H \leftarrow H \cup \{e\}$;
    **else**
        Toss a biased coin with success probability $p$;
        **if** success **then**
            $S_L \leftarrow S_L \cup \{e\}$;
    $W_{H,H} \leftarrow$ COLLECT_WEDGES$(H, H)$;
    $W_{H,S_L} \leftarrow$ COLLECT_WEDGES$(H, S_L)$;
    $W_{S_L,S_L} \leftarrow$ COLLECT_WEDGES$(S_L, S_L)$;
    **for** each $w = (e1, e2) \in W_{H,H}$ **do**
        **if** CHECK_CYCLE$(w, e)$ = True **then**
            $\ell_3 \leftarrow \ell_3 + 1$;
    **for** each $w = (e1, e2) \in W_{H,S_L}$ **do**
        **if** CHECK_CYCLE$(w, e)$ = True **then**
            $\ell_2 \leftarrow \ell_2 + 1$;
    **for** each $w = (e1, e2) \in W_{S_L,S_L}$ **do**
        **if** CHECK_CYCLE$(w, e)$ = True **then**
            $\ell_1 \leftarrow \ell_1 + 1$;
**return** $\ell = \frac{\ell_1}{p^2} + \frac{\ell_2}{p} + \ell_3$;

The COUNT_CYCLES_ORACLE algorithm makes use of an Oracle in order to achieve a better estimate of the number of temporal cycles. The algorithm exploits

two different sets, one for storing the edges considered heavy ($H$) and one for storing edges that are, on the contrary, considered light ($S_L$). It is important to note that we add a new edge to the set $S_L$ according to a sampling probability $p$. The algorithm works as follows: for each new edge received in input, we first classify if the edge is heavy or not, according to the oracle $O_\rho$. If the edge is heavy, we add it to the heavy set $H$, otherwise we sample it with a probability $p$ and eventually add it to the set $S_L$. As already mentioned, light edges are kept according to a given probability, so we will need to take this into account when collecting the final estimate. As a next step, we collect three different sets of wedges. One is built from edge pairs both belonging to the heavy set $H$, the second one is built from edge pairs both from the set of sampled light edges $S_L$ and the third is considering pairs of edges in which one of the two is light and the other is heavy. For each of these sets of wedges, we perform (similarly to before) a verification to understand if a given wedge forms a cycle when coupled with the current edge in the stream. Because we are considering three different sets of wedges, we must define three different counters (each associated with the corresponding set) in order to get an unbiased estimate. The counter associated with the set $W_{H,H}$ is already unbiased, while the counters associated to the sets $W_{H,S_L}$ and $W_{S_L,S_L}$ must be multiplied by a factor $\frac{1}{p}$ and $\frac{1}{p^2}$ respectively. A formal proof is provided in Chapter 4.

For what concerns the time complexity of the algorithm, we first analyze the three calls to the COLLECT_WEDGES procedure separately. Recall that the set of heavy edges of a temporal graph has been denoted with $C_H$, while set of light edges with $C_L$. The first call of the COLLECT_WEDGES procedure is done over the set $H$, which comprises the set of heavy edges (with timestamps within the interval $[t-\delta, t]$) currently acquired from the graph stream. Basically, we are performing the computation over the set $C_H[t-\delta, t]$, since we are using a perfect Oracle. If we let $m = \max_t |C_H[t-\delta, t]|$ be the maximum cardinality of the set of heavy edges over time $t$, the time complexity upper bound of the first COLLECT_WEDGES computation is then $O(m^2)$. We now denote with $k = \max_t |C_L[t-\delta, t]|$ the maximum cardinality of the set of light edges over time $t$. The second call to the COLLECT_WEDGES procedure will then be bounded, in expectation, by $O(p \cdot k \cdot m)$, since we are using a set of *sampled* light edges, with probability $p$. Analogously, the third call will have an upper bound (again, in expectation) of $O(p^2 \cdot k^2)$. In the end, the COUNT_CYCLES_ORACLE will

have an upper bound of $O(|E_T| \cdot (m^2 + p^2 \cdot k^2 + mpk))$. Since we can assume that $(m + p \cdot k) < n$ (with $n = \max_t |E[t - \delta, t]|$), we have an improvement in terms of asymptotic performance over the exact algorithm.

## 3.4 Subroutines

In this section, we provide the subroutines used in the three algorithms, namely, COLLECT_WEDGES and CHECK_CYCLE subroutines. We will first start by introducing the COLLECT_WEDGES procedure.

**Algorithm 4** COLLECT_WEDGES($S_1, S_2$)

**Input**: Sets $S_1, S_2$ of temporal edges.
**Output**: Set $W_{S_1,S_2}$ of wedges where $w \in W_{S_1,S_2}$ is a wedge of the form $w = \langle e_1, e_2 \rangle$.

```
1:  W_{S1,S2} ← ∅;
2:  for each e_i ∈ S_1 do
3:      for each e_j ∈ S_2 do
4:          Let e_i = (u_i, v_i, t_i);
5:          Let e_j = (u_j, v_j, t_j);
6:          if ((v_i = v_j) ∧ (u_i ≠ u_j)) ∨ ((v_i = u_j) ∧ (u_i ≠ v_j)) ∨
7:             ((u_i = u_j) ∧ (v_i ≠ v_j)) ∨ ((u_i = v_j) ∧ (v_i ≠ u_j)) then
8:              w = ⟨e_i, e_j⟩;
9:          if t_i > t_j then
10:             w = ⟨e_j, e_i⟩;
11:         W_{S1,S2} ← W_{S1,S2} ∪ {w};
12: return W_{S1,S2};
```

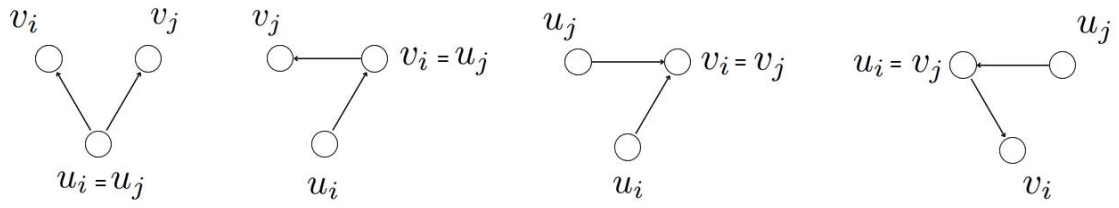


Figure 3.1: The four different types of wedges when neglecting time information.

The COLLECT_WEDGES algorithm, given in input two sets $S_1, S_2$ of temporal edges, returns the set $W$ of all wedges that can be part of a temporal triangle, where each element $w \in W$ is a wedge built from two edges $e_1,\ e_2$. Note that we do not require any ordering of the input sets. The two input sets are also considered different in general. As a first step, we check if an edge $e_i \in S_1$ has **one and only**

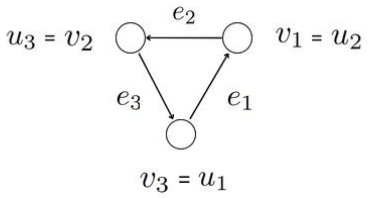


Figure 3.2: A temporal cycle involving three nodes.

**one** vertex in common with an edge $e_j \in S_2$ (otherwise we would also consider temporal stars motifs). We have a total of four different cases to verify (if we neglect the timestamp information, see Figure 3.1). If one of these cases is satisfied, we then check the temporal timestamps $t_i$ and $t_j$ of $e_i$ and $e_j$ respectively: if we have $t_i < t_j$ we will build $w$ as $w = \langle e_i, e_j \rangle$, otherwise we will have $w = \langle e_j, e_i \rangle$. However, storing a wedge as a temporally ordered pair of edges still requires an additional verification to understand the type of the wedge considered. A possible improvement in this sense consists in differentiating between the types of wedges, and return four different sets instead of a single set, so that we know in advance which type of wedge we are addressing. Note also that this algorithm collects all possible wedges that can potentially be part of a temporal triangle. Therefore, this algorithm can be left unchanged when extending the previous algorithms to deal with all the possible temporal triangular motifs.

**Algorithm 5** CHECK_CYCLE$(w, e)$

**Input**: A wedge $w = \langle e_1, e_2 \rangle$ and a temporal edge $e$.
**Output**: True if the triplet $\langle e_1, e_2, e \rangle$ forms a 3-cycle, False otherwise.

```
1: Let e_1 = (u_1, v_1, t_1), e_2 = (u_2, v_2, t_2);
2: Let e = e_3 = (u_3, v_3, t_3);
3: if (v_1 = u_2) ∧ (v_2 = u_3) ∧ (v_3 = u_1) then
4:     return True;
5: else
6:     return False;
```

We now present and discuss the CHECK_CYCLE subroutine, which returns whether or not a given wedge $w$ forms a temporal cycle with an edge $e$. Recall that a wedge is an ordered pair of edges $\langle e_1, e_2 \rangle$, so we have $t_1 < t_2$. Here we assume that the edge $e$ is received after the edges that form the wedge $w$. This implies that $t_1 < t_2 < t$, where $t$ is the timestamp of $e$. We also know that $t - t_1 < \delta$, since all the wedges are collected within the time interval $[t - \delta, t]$ and so there is no

need to check time consistency. We need instead to verify that the wedge we are considering is of the correct type. Given that $e_1 = (u_1, v_1, t_1),\ e_2 = (u_2, v_2, t_2)$, we need to check if $v_1 = u_2$ in order to understand if the wedge we are looking at is correct. Once we verified the correctness of the wedge, we simply check if the edge $e$ "closes" the loop.

## 3.5 Extension to Temporal Triangles

The algorithms discussed before can be adapted to deal with temporal triangle motifs that are not temporal cycles. Indeed, by modifying the CHECK_CYCLE algorithm, we can consider different kinds of triangular motifs. This approach also allows to distinguish between the eight different types of triangular motifs, which we report in Figure 3.3. We show in here a simple extension to a triangular motif of the form $T_7 = \langle (x_1, x_2), (x_1, x_3), (x_2, x_3) \rangle$. First, we recall that the COLLECT_WEDGES algorithm provides all the wedges we are interested in between two given sets of temporal edges, so there is no need to introduce any modification. For what concerns the CHECK_CYCLE algorithm, we now introduce a modified version of the algorithm, named CHECK_$T_7$.

**Algorithm 6** CHECK_$T_7(w, e)$

**Input**: A wedge $w = \langle e_1, e_2 \rangle$ and a temporal edge $e$.
**Output**: True if the triplet $\langle e_1, e_2, e \rangle$ forms a $T_7$ triangle , False otherwise.

```
1: Let e_1 = (u_1, v_1, t_1), e_2 = (u_2, v_2, t_2);
2: Let e = e_3 = (u_3, v_3, t_3);
3: if (u_1 = u_2) ∧ (v_1 = u_3) ∧ (v_3 = v_2) then
4:     return True;
5: else
6:     return False;
```

The major change with respect to the CHECK_CYCLE algorithms is the condition in Line 3: we first verify that the wedge we are choosing is correct for generating an instance of $T_7$ and we subsequently verify if the third edge $e = e_3$ allows to build an instance of $T_7$. Recall also that we already know that $e_1, e_2, e_3$ are temporally ordered and that $e_1, e_2$ are collected in such a way that $t - t_1 < \delta$, with $t$ timestamp of $e = e_3$ and $t_1$ timestamp of $e_1$.

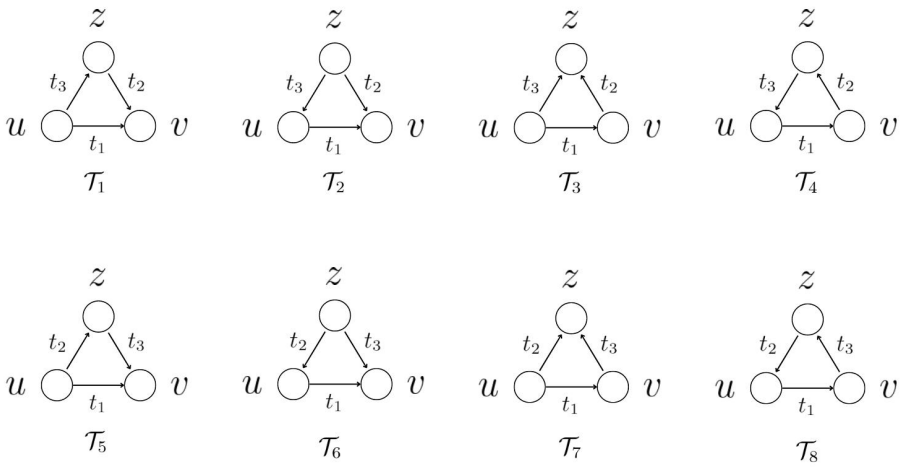


Figure 3.3: All eight temporal triangles.

In total, we need eight different versions for the CHECK_CYCLE algorithm in order to cover all triangular motifs, which is still a reasonable amount. We can also introduce a unified procedure that checks all the possible eight conditions we have expressed in the definition of temporal triangle (Definition 7 of section 2.2). Nevertheless, this may become a strong limitation when dealing with larger motifs (motifs $M = (V_M, E_M, \sigma)$ with $|V_M|, |E_M| > 3$).

An important discussion must be done instead for what concerns the definition of Oracle and the heaviness property. We can exploit two different approaches. The first one consists in considering an edge as *heavy* if the number of temporal triangles $\eta_T(e)$ in which $e$ is involved is greater or equal than the given threshold $\rho$, that is, $\eta_T(e) \geq \rho$. A second possibility is to define an Oracle for each different triangle, ending up with a total of eight oracles. This may improve the accuracy of the estimates, but becomes rapidly infeasible when considering larger motifs. For this, reason, we will stick with the first approach when practically developing our algorithms.

# Chapter 4

# Algorithms Analysis

We will hereby prove that both the COUNT_CYCLES_SAMPLING and the COUNT_CYCLES_ORACLE algorithms return and unbiased estimate of the number of $\delta$-instances of the 3-cycle motif $N_C$ in $T$, where $\delta$ is the provided duration of the motif and $T$ is the temporal graph considered. It is important to note that the following proofs can be slightly modified in order to consider all possible temporal triangles instead of temporal cycles only.

## 4.1 Sampling Algorithm Unbiasness

**Theorem 1.** *Given a temporal graph $T = (V_T, E_T)$ and a total duration $\delta$, the COUNT_CYCLES_SAMPLING algorithm returns an unbiased estimate $\frac{\ell}{p^2}$ of the number of $\delta$-instances of the cycle motif $N_C$ appearing in $T$, that is:*

$$\mathbb{E}\left[\frac{\ell}{p^2}\right] = N_C. \tag{4.1}$$

*Proof.* Let us first define, for each cycle $C = \langle e_1, e_2, e_3 \rangle \in T$ a random variable $x_C$:

$$x_C = \begin{cases} 1 & \text{if } e_1, e_2 \in S \\ 0 & \text{otherwise} \end{cases}$$

We have that:

$$P(x_C = 1) = p^2 \tag{4.2}$$

Note that $P(x_C = 1)$ is equal to the probability of sampling the first two edges $e_1, e_2$ of the cycle $C$, that is $p^2$, since the edges are sampled independently one from the other. Let us now recall that $\ell$ can be elicited as:

$$\ell = \sum_{C \in \mathcal{C}} x_C \tag{4.3}$$

Therefore:

$$\mathbb{E}[\ell] = \sum_{C \in \mathcal{C}} \mathbb{E}[x_C] = \sum_{C \in \mathcal{C}} P(x_C = 1) = \sum_{C \in \mathcal{C}} p^2 = p^2 N_C. \tag{4.4}$$

Finally from (4.4), and since $p$ is a constant value, we obtain:

$$\mathbb{E}\left[\frac{\ell}{p^2}\right] = N_C \tag{4.5}$$

which concludes the proof.

□

## 4.2 Algorithm with Predictions Unbiasness

**Theorem 2.** *Given a temporal graph $T = (V_T, E_T)$, a duration $\delta$, and a perfect oracle $O_\rho$ (with associated threshold $\rho$), the COUNT_CYCLES_ORACLE algorithm returns an unbiased estimate $\ell$ of the number of $\delta$-instances of the cycle motif $N_C$ appearing in $T$, that is:*

$$\mathbb{E}[\ell] = N_C. \tag{4.6}$$

*Proof.* Recall that, given a threshold $\rho \geq 0$ and a temporal graph $T = (V_T, E_T)$, we define a temporal edge $e$ as heavy if it is involved in a number of cycles $\eta(e) \geq \rho$. The set of all heavy edges for the given graph $T$ is denoted as $C_H = \{e \in E_T |\ \eta(e) \geq \rho\}$. The complementary set of light edges is instead denoted as $C_L = \{e \in E_T |\ \eta(e) < \rho\}$. Note also that $C_H \cup C_L = E_T$ and that $C_H \cap C_L = \varnothing$.

Let us now denote with $\mathcal{C}_{H,H}$ the set of all cycles $C = \langle e_1, e_2, e_3 \rangle \in T$ where the first two edges $(e_1, e_2) \in C_H \times C_H$ are heavy. Shortly:

$$\mathcal{C}_{H,H} = \{C \in T \mid C = \langle e_1, e_2, e_3 \rangle, (e_1, e_2) \in C_H \times C_H\}. \tag{4.7}$$

We define similarly the sets $\mathcal{C}_{L,H}$ and $\mathcal{C}_{L,L}$ as:

$$\mathcal{C}_{L,H} = \{C \in T \mid C = \langle e_1, e_2, e_3 \rangle, e_1 \in C_L, e_2 \in C_H\} \tag{4.8}$$

$$\mathcal{C}_{L,L} = \{C \in T \mid C = \langle e_1, e_2, e_3 \rangle, (e_1, e_2) \in C_L \times C_L\}. \tag{4.9}$$

where we considered 3-cycles with a light edge and a heavy edge $e_1 \in C_L, e_2 \in C_H$ and 3-cycles with both light edges $(e_1, e_2) \in C_L \times C_L$, respectively. Let us also denote with $N_{H,H} = |\mathcal{C}_{H,H}|$, $N_{L,H} = |\mathcal{C}_{L,H}|$ and $N_{L,L} = |\mathcal{C}_{L,L}|$, the cardinality of each set described above.

We first underline the following property:

$$N_C = N_{H,H} + N_{L,H} + N_{L,L} \tag{4.10}$$

which follows from the fact that the set of all 3-cycles $\mathcal{C}$ in $T$ (with $|\mathcal{C}| = N_C$) can be partitioned in the three sets $\mathcal{C}_{H,H}, \mathcal{C}_{L,H}$ and $\mathcal{C}_{L,L}$.

Let us now define, for a 3-cycle $C = \langle e_1, e_2, e_3 \rangle \in \mathcal{C}_{L,H}$, a random variable $x_C$:

$$x_C = \begin{cases} 1 & \text{if } e_1 \in S_L \\ 0 & \text{otherwise} \end{cases}$$

where $S_L$ is the set of sampled light edges. Similarly, we define $y_C$ for a 3-cycle $C = \langle e_1, e_2, e_3 \rangle \in \mathcal{C}_{L,L}$ as:

$$y_C = \begin{cases} 1 & \text{if } e_1, e_2 \in S_L \\ 0 & \text{otherwise} \end{cases}$$

Note that, similarly to what we have seen in the proof of Theorem 1, we have:

$$P(x_C = 1) = p, P(y_C = 1) = p^2. \tag{4.11}$$

Recalling that the estimate returned by the COUNT_CYCLES_ORACLE algorithm is:

$$\ell = \frac{\ell_1}{p^2} + \frac{\ell_2}{p} + \ell_3 \tag{4.12}$$

we can analyze each term of the sum separately by exploiting the linearity of the expectation:

$$\mathbb{E}[\ell] = \mathbb{E}\left[\frac{\ell_1}{p^2}\right] + \mathbb{E}\left[\frac{\ell_2}{p}\right] + \mathbb{E}[\ell_3] \tag{4.13}$$

We will first focus on the $\ell_3$ term. We have:

$$\ell_3 = \sum_{\substack{C=\langle e_1,e_2,e_3\rangle \in \mathcal{C} \\ e_1,e_2 \in H}} 1 = \sum_{C \in \mathcal{C}_{H,H}} 1 = N_{H,H} \tag{4.14}$$

which follows from the fact that we are exploiting a perfect oracle $O_\rho$. Indeed, we have that the set of heavy edges $H$ built from the COUNT_CYCLES_ORACLE algorithm is exactly equal to the true set of heavy edges $C_H$. Therefore, we are counting all the 3-cycles in which the first two edges are heavy, which is the definition of the cardinality of $\mathcal{C}_{H,H}$. Obviously, we will also have that:

$$\mathbb{E}[\ell_3] = N_{H,H}. \tag{4.15}$$

Let us now analyze the term $\ell_2$:

$$\ell_2 = \sum_{\substack{C=\langle e_1,e_2,e_3\rangle \in \mathcal{C} \\ e_1 \in S_L, e_2 \in H}} 1 = \sum_{C \in \mathcal{C}_{L,H}} x_C \tag{4.16}$$

which is accounting for 3-cycles in which the first edge is light and the second is heavy. In this case, the set of light edges $C_L$ is not equal to $S_L$, since we are performing a uniform sampling with probability $p$ over the set $C_L$. When taking the expectation with respect to $\ell_2$ we obtain:

$$\mathbb{E}[\ell_2] = \sum_{C \in \mathcal{C}_{L,H}} \mathbb{E}[x_C] = \sum_{C \in \mathcal{C}_{L,H}} p = p \cdot N_{L,H} \tag{4.17}$$

.

Analogously, we now provide a similar derivation for the term $\ell_1$:

$$\ell_1 = \sum_{\substack{C=\langle e_1,e_2,e_3\rangle \in \mathcal{C} \\ e_1,e_2 \in S_L}} 1 = \sum_{C \in \mathcal{C}_{L,L}} y_C \tag{4.18}$$

By taking the expectation we get:

$$\mathbb{E}[\ell_1] = \sum_{C \in \mathcal{C}_{L,L}} \mathbb{E}[y_C] = \sum_{C \in \mathcal{C}_{L,L}} p^2 = p^2 \cdot N_{L,L}. \tag{4.19}$$

Substituting each term in equation (4.12) gives:

$$\mathbb{E}[\ell] = \mathbb{E}\left[\frac{p^2 \cdot N_{L,L}}{p^2}\right] + \mathbb{E}\left[\frac{p \cdot N_{L,H}}{p}\right] + \mathbb{E}[N_{H,H}] = N_{L,L} + N_{L,H} + N_{H,H} = N_C \tag{4.20}$$

which concludes the proof.

□

## 4.3 Generalization to Temporal Triangles

In this section, we provide some generalization details in order to extend the proofs to the case of temporal triangles.

As a first step, we need to revise some concepts. We denote with $\mathcal{T}$ the set of all temporal triangles, which also includes the set of all temporal cycles $\mathcal{C}$. $N_T$ denotes instead the cardinality of the set $\mathcal{T}$. If we consider the new set $\mathcal{T}$ instead of $\mathcal{C}$, we can extend the proof regarding the sampling algorithm's unbiasness without any further modification. Basically, the summation in (4.3) will be done over all the triangles belonging to $\mathcal{T}$, giving:

$$\mathbb{E}\left[\frac{\ell}{p^2}\right] = N_T. \tag{4.21}$$

For what concerns the algorithm with predictions' proof, we need to account for some additional changes. In particular, the sets $\mathcal{C}_{H,H}$, $\mathcal{C}_{L,H}$, $\mathcal{C}_{L,L}$ defined in (4.7), (4.8) and (4.9), should be modified in order to consider temporal triangles, which we will denote with $\mathcal{T}_{H,H}$, $\mathcal{T}_{L,H}$, $\mathcal{T}_{L,L}$. These sets are, again, a partition of $\mathcal{T}$, so nothing has changed in this sense. Note also that the sets of heavy and light edges $C_H, C_L$ remains the same, except for the fact that the count of cycles $\eta(e)$ becomes the count of all temporal triangles $\eta_T(e)$. The property reported in (4.10) is preserved also for the case of temporal triangles. Indeed, if we denote with $\mathcal{N}_{H,H}, \mathcal{N}_{L,H}$ and $\mathcal{N}_{L,L}$ the cardinalities of the sets $\mathcal{T}_{H,H}$, $\mathcal{T}_{L,H}$ and $\mathcal{T}_{L,L}$ respectively, we end up with:

$$N_T = \mathcal{N}_{H,H} + \mathcal{N}_{L,H} + \mathcal{N}_{L,L}. \tag{4.22}$$

By accounting for these modifications from equation (4.11) to equation (4.19) we end up with:

$$\mathbb{E}[\ell] = N_T. \tag{4.23}$$

We have therefore proved that both the sampling and the Oracle-based algorithms' outputs are unbiased estimates of the number of temporal triangles appearing in a temporal network.

# Chapter 5

# Experiments

In this chapter, we will present an experimental evaluation of the algorithms introduced in this work. We will first have a look at the datasets on which we have tested our algorithms and provide some insights of our implementation. Then a brief section describes the results obtained with the exact algorithm. The results obtained with the standard sampling algorithm and with the Oracle-based algorithm are then presented, with a dedicated section on a corrupted version of the Oracle. Finally, we will perform some comparisons between the sampling algorithm and the Oracle-based algorithm, in order to access the effectiveness of our approach. We will switch the temporal triangles considered during our experiments, in order to show the results on different triangle types while reducing the level of verbosity.

## 5.1 Datasets

For our experiments we have used publicly available datasets provided by the Stanford Network Analysis Project (SNAP) repository [10]. It provides several different typologies of networks: social networks, networks with ground-truth communities, communication networks, citation networks, temporal networks, road networks, collaboration networks, web graphs, autonomous systems graphs, signed networks

and many others.

In particular, we have included in our experiments the following temporal networks:

- **CollegeMsg**: it comprises thousands of messages sent on an online social network at the University of California. A temporal edge $e = (u, v, t)$ specify that a user $u$ sent a message to user $v$ at time $t$.
- **email-Eu-core**: anonymized e-mail data gathered from an European reserach institution. A temporal edge $e = (u, v, t)$ specify that a user $u$ sent an e-mail to a user $v$ at time $t$.
- **sx-askubuntu**: it is a temporal network describing interactions (questions, answers and comments) between users of the stack exchange web site Ask Ubuntu[1]. A temporal edge $e = (u, v, t)$ specify that a user $u$ interacted with a user $v$ at time $t$.
- **sx-mathoverflow**: temporal network of interactions on the stack exchange web site Math Overflow[2]. Temporal edges are defined similarly to the sx-askubuntu dataset.
- **sx-superuser**: represent interactions between users of the stack exchange web site Super User[3]. Edges are defined following sx-mathoverflow and sx-askubuntu datasets.

We provide some general statistics of the datasets considered in Table 5.1.

| **Dataset** | **Nodes** | **Temporal edges** | **Static edges** | **Time span** |
|---|---|---|---|---|
| CollegeMsg | 1899 | 20296 | 59835 | 193 days |
| email-Eu-core | 986 | 332334 | 24929 | 803 days |
| sx-askubuntu | 159316 | 964437 | 596933 | 2613 days |
| sx-mathoverflow | 24818 | 506550 | 239978 | 2350 days |
| sx-superuser | 194085 | 1443339 | 924886 | 2773 days |

Table 5.1: General statistics of the datasets included in our experiments.

Because we are analyzing temporal networks, an important parameter to set it the time duration $\delta$ of the temporal motifs we are looking for. In our case, we

[1] Ask Ubuntu site: https://askubuntu.com/
[2] Math Overflow site: https://mathoverflow.net/
[3] Super User site: https://superuser.com/

decided to set the value of $\delta$ to be three times the average time elapsed between two consecutive temporal edges received, which we denote with $t_{avg}(E_T)$. This choice is related to the fact that we are trying to consider an average duration $\delta$ of the temporal triangles appearing in the network. We report in Table 5.2 the values used in our experiment, rounded to the nearest integer.

| **Dataset** | $t_{avg}(E_T)$ **(s)** | $\delta$ **(s)** |
|---|---|---|
| CollegeMsg | 280 | 840 |
| email-Eu-core | 137 | 411 |
| sx-askubuntu | 243 | 729 |
| sx-mathoverflow | 520 | 1560 |
| sx-superuser | 189 | 567 |

Table 5.2: Duration $\delta$ considered for each dataset, computed as three times the average time elapsed between two consecutive temporal edges.

## 5.2 Evaluation Criteria

In order to experimentally evaluate our sampling and Oracle-based algorithms, we have considered the following criteria:

- Relative error: it is computed as the ratio between the absolute difference of the true count and the estimated count and the true count. If we denote with $C$ the true count of a given temporal triangle and with $\tilde{C}$ the estimated count provided by the algorithm, the relative error is computed as $\frac{|C-\tilde{C}|}{C}$. Because we have performed multiple trials, the relative error reported is the *average* of all the relative errors.

- Root Mean Squared Error (RMSE): denoting with $N$ the total number of trials, the RMSE is computed as

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(C - \tilde{C})^2}$$

- Execution time: for both algorithms, we will provide the average execution time over the trials, in order to make additional comparisons.

- Memory occupancy: we decided to obtain some insights also on the average amount of memory needed by the two algorithms. We restrict ourselves only

to the memory occupied by stored edges. Each stored edge contributes to one unit of memory. The memory occupancy is computed in a cumulative way over the data stream.

## 5.3 Implementation Details

In this section, we provide some elements regarding the implementation of the algorithms described in this work. The code was developed using the Python language in order to be able to eventually extend the code with Machine Learning predictors, thanks to the availability of several powerful libraries, such as pytorch-geometric.

First of all, we present a code snippet used to pre-process all the datasets considered. This code was used to accomplish three main tasks:

- Verify that the edges of each temporal network were received in a temporally-ordered way.
- Collect the average time between consecutive edges in the stream, which was used as a starting point for setting the values for the $\delta$ parameter.
- Check the presence of self-loops and eventually remove them.

```python
def load_graph(file, delimiter=' '):
    temporal_edges = []
    first_line = True
    scaling, prev_t, avg_delta = 0, 0, 0
    with open(file) as infile:
        for line in infile:
            chunks = line.split(delimiter)
            u = int(chunks[0])
            v = int(chunks[1])
            if u == v:
                continue
            if first_line:
                t = 0
                scaling = int(chunks[2])
                first_line = False
            else:
                t = int(chunks[2]) - scaling
                avg_delta += (t - prev_t)
                prev_t = t
            temporal_edge = (u, v, t)
            temporal_edges.append(temporal_edge)
    avg_delta = np.round(avg_delta / len(temporal_edges))
    min_t = min(x[2] for x in temporal_edges)
    if min_t < 0:
        temporal_edges = [x[:2] + (x[2] + min_t,) for x in temporal_edges]
    temporal_edges = sorted(temporal_edges, key=lambda x: x[2])
    return temporal_edges, avg_delta
```

Figure 5.1: Code snippet for dataset preprocessing.

Because of the increasing amount of temporal edges when considering large networks (such as in sx-superuser), we have devised an alternative formulation of the COLLECT_WEDGES algorithm in the case in which the two input sets where identical. Basically, we exploited this property in order to reduce the number of different pairs of edges to consider, thus obtaining a slightly better time complexity. Indeed, when we have two different sets $S_1 \neq S_2$, we perform a total of $|S_1| \times |S_2|$ iterations over the two sets, while in the second case we have that $S_1 = S_2$, $|S_1| = |S_2| = n$, and therefore this gives us a total amount of iterations of $\frac{n}{2}(n+1)$.

```
def collect_wedges(s1, s2):
    wedges = []
    if s1 != s2:
        for e_i in s1:
            for e_j in s2:
                u_i, v_i, t_i = e_i
                u_j, v_j, t_j = e_j
                w = []
                if ((v_i == v_j) and (u_i != u_j)) or ((v_i == u_j) and (u_i != v_j)) or \
                   ((u_i == u_j) and (v_i != v_j)) or ((u_i == v_j) and (v_i != u_j)):
                    w = [e_i, e_j]
                if t_i > t_j:
                    w = [e_j, e_i]
                if len(w) != 0 and w not in wedges:
                    wedges.append(w)
    else:
        for i in range(len(s1)):
            for j in range(i, len(s2)):
                e_i, e_j = s1[i], s2[j]
                u_i, v_i, t_i = e_i
                u_j, v_j, t_j = e_j
                w = []
                ...
```

Figure 5.2: Code snippet for the COLLECT_WEDGES subroutine.

Another important element to consider is the perfect Oracle implementation. Our approach consists in constructing a dictionary where, at each edge (i.e., the key), we associate the total amount of temporal triangles in which it is involved (i.e, the value). The perfect Oracle is then retrieved by filtering this dictionary and retaining only the edges that are heavy according to the given threshold value $\rho$. Note that a triangle-specific Oracle may work better, but requires a total of eight oracles. As already mentioned, for larger motifs this may become infeasible, so we maintained a unique Oracle for all the temporal triangles. In order to keep the Oracle behavior as close as possible to a real predictor, we decided to build the set of heavy edges as a look-up table, from which we can recover all the heavy edges in constant time, exploiting the Python Set implementation with hash tables.

```
def get_oracle(edges_dictionary, threshold, percentage_retain):
    heavy_edges = []
    dic = {k: v for k, v in sorted(edges_dictionary.items(),
                                   key=lambda item: item[1],
                                   reverse=True)}
    for key, value in dic.items():
        if value > threshold:
            heavy_edges.append(eval(key))
    n = int(len(heavy_edges) * percentage_retain)
    return heavy_edges[:n]
```

Figure 5.3: Code snippet for constructing the Oracle.

## 5.4 Exact Algorithm Results

We now present briefly the results obtained with the implementation of our exact algorithm. We compared the results with the algorithm provided in [15], in order to check the correctness and consistency of our approach. The experiments were done by fixing the value of $\delta$ to 200 seconds and computing the count for each triangle type. The results are depicted in Table 5.3.

| **Dataset** | **T1** | **T2** | **T3** | **T4** | **T5** | **T6** | **T7** | **T8** | **Time (s)** $(*)$ |
|---|---|---|---|---|---|---|---|---|---|
| CollegeMsg | 38 | 34 | 18 | 14 | 22 | 37 | 33 | 32 | 4.16 |
| email-Eu-core | 39 | 509 | 20 | 2 | 477 | 567 | 499 | 8 | 100.40 |
| sx-askubuntu | 199 | 316 | 68 | 9 | 50 | 143 | 83 | 16 | 1.27 |
| sx-mathoverflow | 40 | 72 | 15 | 1 | 8 | 17 | 14 | 2 | 0.48 |
| sx-superuser | 393 | 683 | 122 | 21 | 103 | 233 | 123 | 34 | 2.23 |

Table 5.3: Exact temporal triangles counts with a total duration $\delta$ of 200 seconds. $(*)$ The times reported are to be considered as an indicative measure only.

It is possible to see from the results obtained that our simple exact algorithm requires a considerable amount of time for solving the problem. This is mainly due to the COLLECT_WEDGES algorithm, which requires comparisons among several pairs of edges. Note that the time complexity of the algorithm does not depend directly of the number of edges of the network considered. Indeed, we are considering only a subset of the temporal edges that are within the interval $[t-\delta, t]$.

This is the reason why we may obtain a lower execution time for networks that are larger if considered as a whole, as we expected by the time complexity analysis done in section 3.1. As an example, we may look at the datasets CollegeMsg and sx-mathoverflow. The first one has 20296 temporal edges, while the second has 506550 edges; however, the time required for the exact algorithm for CollegeMsg is roughly 8 times higher than the one required for sx-mathoverflow. This means that, within the same time interval, the CollegeMsg dataset comprises more edges with respect to the sx-mathoverflow dataset. If we increase the value of $\delta$, we may obtain completely different results, since we will take into account more and more edges. Ultimately, for a really large value of delta, the entire set of edges would be considered, and therefore we would obtain higher execution times for larger networks. We will keep this execution times as a reference when comparing them with the ones obtained by the sampling and Oracle-based algorithms. Of course, we will provide the time execution of the exact algorithm according to the value of $\delta$ considered when performing time comparisons.

## 5.5 Sampling Algorithm Results

We hereby show the counts obtained using the standard sampling algorithm for a fixed value of the sampling probability $p$, which is of 0.1. Additional values for the probability $p$ have been tested and are presented in section 5.7. Remind that, as we increase the value of $p$, we are requiring a greater amount of memory in order to store the edges of the given temporal graph. In particular, if we assume that a stored edge occupies one unit of memory, the memory occupancy can be formulated (in expectation) as $p \cdot |E_T|$, where we recall that $|E_T|$ is the total number of edges of the input temporal network $T$. The occupancy may slightly differ in practice, since we are not completely sure to be able to sample with a probability that is exactly equal to $p$. We present some memory usage analysis in section 5.7. All the experiments shown were repeated for a total of 50 trials.

In Table 5.4, we report, for each dataset and for the corresponding duration $\delta$ considered, the relative error and the root mean squared error (RMSE) for each triangle type, averaging over the trials. Table 5.5 shows instead the average time required for a single trial of the sampling algorithm and the speed-up factor obtained

with respect to the exact algorithm, computed as $t_{ex}/t_s$ where $t_{ex}$ and $t_s$ are the computation times required by the exact and the sampling algorithm respectively.

| Dataset | Error | T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 |
|---|---|---|---|---|---|---|---|---|---|
| CollegeMsg | Relative | 0.73 | 0.53 | 0.82 | 0.78 | 0.67 | 0.61 | 0.64 | 0.67 |
| | RMSE | 497.74 | 390.03 | 492.12 | 435.85 | 496.30 | 446.56 | 464.83 | 395.34 |
| email-Eu-core | Relative | 0.81 | 0.25 | 0.71 | 1.26 | 0.31 | 0.29 | 0.26 | 1.38 |
| | RMSE | 137.14 | 387.639 | 99.70 | 61.12 | 397.31 | 472.04 | 405.11 | 73.05 |
| sx-askubuntu | Relative | 0.25 | 0.22 | 0.38 | 0.75 | 0.44 | 0.26 | 0.33 | 0.70 |
| | RMSE | 409.70 | 438.511 | 266.23 | 113.86 | 229.556 | 326.66 | 294.76 | 187.79 |
| sx-mathoverflow | Relative | 0.28 | 0.26 | 0.43 | 1.02 | 0.53 | 0.38 | 0.44 | 0.69 |
| | RMSE | 314.53 | 383.98 | 168.88 | 83.62 | 202.09 | 410.66 | 237.83 | 151.60 |
| sx-superuser | Relative | 0.21 | 0.20 | 0.36 | 0.74 | 0.42 | 0.22 | 0.28 | 0.52 |
| | RMSE | 494.73 | 546.50 | 305.69 | 113.17 | 248.04 | 372.80 | 267.92 | 184.133 |

Table 5.4: Relative error and RMSE error for each triangle type obtained with sampling algorithm.

| Dataset | Average time (s) | Exact algorithm time (s) | Speed-up factor |
|---|---|---|---|
| CollegeMsg | 0.14 | 18.94 | 135.28 |
| email-Eu-core | 0.56 | 288.39 | 515.98 |
| sx-askubuntu | 0.49 | 4.19 | 8.55 |
| sx-mathoverflow | 0.26 | 2.21 | 8.50 |
| sx-superuser | 0.73 | 5.95 | 8.15 |

Table 5.5: Average time and speed-up factor over the exact algorithm obtained with the standard sampling algorithm.

From the results obtained we can see that the standard sampling algorithm obtains quite large errors, especially when considering the least frequent temporal triangles. Indeed, for the triangle $T_4$ (temporal 3-cycles) we get the highest relative errors. This is mainly due to the fact that this type of triangle is rarely counted, since it appears only few times. Conversely, the lowest relative error is achieved when considering the most frequent type of triangle in the datasets, which is the triangle $T_2$. The *average relative error* over all triangles and over all datasets is of **0.53**, while for the triangles $T_4$ and $T_2$ is of 0.91 and of 0.29, respectively (averaging over the five datasets). It is also interesting to note that the best estimates are achieved when considering larger networks. As an example, in the sx-superuser dataset, the relative errors achieved are undoubtedly the smallest ones. Again, this may be due mainly to the fact that there is a larger amount of triangles, thus increasing the probability to sample a higher amount of such triangles. In other words, the sampling algorithm works better when making an estimate on a larger amount of motifs rather than estimating counts of very rare motifs.

For what concerns the execution time of the algorithm, it is clear that the sampling algorithm is much faster than the exact one, since it operates on a small subset of the original set of edges of the network considered. As expected, the time required by the sampling algorithm increases when considering datasets that have an higher average cardinality of the set $E[t-\delta, t]$, as discussed before. For such datasets, the speed-up factor is really large, demonstrating the efficiency brought by the sampling technique. Note that these values are meaningful within our implementation of the exact algorithm and as long as we are using the same implementation of the subroutines.

## 5.6 Oracle-based Algorithm Results

We will now look at the counts obtained using the Oracle-based algorithm. We keep fixed the sampling probability $p$ to 0.1 to get a fair comparison with respect to the sampling algorithm, which we will discuss in section 5.7. We present the results obtained with a threshold value for the Oracle of 2. As before, additional values for the probability $p$ and for the threshold $\rho$ have been tested and are presented in section 5.7. Again, all the experiments shown were repeated for a total of 50 trials. We first consider the case of a perfect Oracle, and then we will switch to its corrupted version, which is used to better model a real Machine Learning predictor.

### 5.6.1 Perfect Oracle Scenario

In this first scenario, we are considering an Oracle that knows exactly the edges that are heavy, according to a threshold of 2. This means that an edge is considered heavy if it is involved in at least two temporal triangles. In Table 5.6, we report the relative error and the root mean squared error (RMSE) for each triangle type and for each dataset considered, averaged over the 50 trials. Table 5.7 shows the average time required for a single trial of the Oracle-based algorithm and the speed-up factor obtained with respect to the exact algorithm, computed as $t_{ex}/t_o$ where $t_{ex}$ and $t_o$ are the computation times required by the exact and the Oracle-based algorithm, respectively.

| Dataset | Error | T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 |
|---|---|---|---|---|---|---|---|---|---|
| CollegeMsg | Relative | 0.01 | 0.03 | 0.02 | 0.01 | 0.02 | 0.01 | 0.02 | 0.01 |
| | RMSE | 7.71 | 25.49 | 9.48 | 4.29 | 16.25 | 11.82 | 12.78 | 4.34 |
| email- | Relative | 0.20 | 0.12 | 0.15 | 0.20 | 0.09 | 0.10 | 0.09 | 0.12 |
| Eu-core | RMSE | 45.46 | 161.43 | 21.24 | 14.75 | 121.75 | 146.34 | 137.17 | 6.40 |
| sx- | Relative | 0.13 | 0.14 | 0.09 | 0.19 | 0.18 | 0.16 | 0.08 | 0.16 |
| askubuntu | RMSE | 224.13 | 266.11 | 55.50 | 36.05 | 93.96 | 198.34 | 72.64 | 41.94 |
| sx- | Relative | 0.14 | 0.15 | 0.15 | 0.20 | 0.23 | 0.15 | 0.16 | 0.15 |
| mathoverflow | RMSE | 157.26 | 215.4 | 57.663 | 23.55 | 76.98 | 157.86 | 79.72 | 38.64 |
| sx- | Relative | 0.12 | 0.12 | 0.14 | 0.26 | 0.18 | 0.13 | 0.12 | 0.22 |
| superuser | RMSE | 278.14 | 328.57 | 122.12 | 50.79 | 118.09 | 221.13 | 106.59 | 68.39 |

Table 5.6: Relative error and RMSE error for each triangle type obtained with Oracle-based algorithm.

| Dataset | Average time (s) | Exact algorithm time (s) | Speed-up factor |
|---|---|---|---|
| CollegeMsg | 0.27 | 18.94 | 70.15 |
| email-Eu-core | 1.59 | 288.39 | 181.38 |
| sx-askubuntu | 0.94 | 4.19 | 4.46 |
| sx-mathoverflow | 0.51 | 2.21 | 4.33 |
| sx-superuser | 1.39 | 5.95 | 4.28 |

Table 5.7: Average time and speed-up factor over the exact algorithm obtained with the Oracle-based sampling algorithm.

The results obtained shows that, with an Oracle-based algorithm, we can improve the estimates on the temporal triangles counts by large margins on all the datasets. As opposed to the case of the sampling algorithm, the best counts are obtained on networks in which there are less temporal triangles. This is due to the fact that, thanks to the perfect Oracle, we are deterministically storing the edges that contribute the most to the temporal triangles' counts. In this way, we are increasing the probability to count triangles that would be otherwise neglected. Overall, the Oracle-based algorithm obtains an *average relative error* over all datasets and over all triangles of **0.12**, which is roughly four times smaller than the one obtained with the standard sampling algorithm.

We will now focus our analysis on the execution time of the algorithm. From the values reported in Table 5.7, we can notice that the Oracle-based algorithm maintains a considerable speed-up factor over the exact algorithm, even in the case of large networks. The highest speed-up are however achieved on the CollegeMsg and email-Eu-core temporal datasets, which are the smallest ones. Even if it seems

counterintuitive at first, recall that the algorithm time complexity is not directly correlated to the size of the network, but to the amount of edges whose timestamps are within the interval $[t - \delta, t]$.

### 5.6.2 Corrupting the Oracle

In this section, we consider the case in which the Oracle is not perfect, and may produce wrong predictions. We have performed two different kinds of corruption on the perfect Oracle for the purpose of our experiments:

- Instead of the entire set of heavy edges, the Oracle's knowledge is constrained such that only a (small) percentage of heavy edges is known. We decided to maintain the heaviest edges and progressively remove the least heavy ones. This approach is based on the idea that, for a real predictor, it is probably easier to distinguish between really heavy edges and light edges. Intuitively, edges for which their corresponding counts $\eta_T(e)$ are near the threshold $\rho$ may be harder to distinguish. The corruption is controlled by a single parameter, which we will denote with $P_r$, that represents the percentage of heavy edges that is withheld.
- A second type of corruption is achieved by switching a subset of the heavy edges with light ones. This time, we tried to corrupt the Oracle in the opposite way: we left unchanged the least heavy edges and changed the heaviest ones with light edges. This scenario is considering a much worse predictor with respect to the first case, in order to test the robustness of the algorithm to bad Oracles. The corruption is controlled by a single parameter, which we will refer to as corruption level $L_c$, that represents the percentage of heavy edges that are discarded and replaced by light ones.

We will first present the experiments done with the first type of corruption. We performed, as before, 50 trials for each configuration considered. The sampling probability and the threshold value are again of 0.1 and 2, respectively. In Tables 5.8 and 5.9, we present the results obtained for multiple values of the percentage of retained heavy edges $P_r$. For visualization and synthesis purposes, we hereby present the results associated to the temporal triangles $T_4$ and $T_2$. Similar results also holds for the other triangles' types. The results with $P_r = 1.0$ are the one obtained with the perfect Oracle. Follows Table 5.10, which reports the speed-up factors obtained.

| **Dataset** | **Error** | $P_r = 0.1$ | $P_r = 0.5$ | $P_r = 0.8$ | $P_r = 1.0$ |
|---|---|---|---|---|---|
| CollegeMsg | Relative | 0.38 | 0.12 | 0.04 | 0.03 |
| | RMSE | 265.82 | 92.97 | 34.42 | 25.49 |
| email-Eu-core | Relative | 0.24 | 0.14 | 0.10 | 0.12 |
| | RMSE | 352.80 | 208.84 | 153.84 | 161.43 |
| sx-askubuntu | Relative | 0.22 | 0.15 | 0.16 | 0.14 |
| | RMSE | 424.68 | 297.79 | 320.21 | 266.11 |
| sx-mathoverflow | Relative | 0.27 | 0.16 | 0.15 | 0.15 |
| | RMSE | 376.02 | 244.44 | 212.48 | 215.4 |
| sx-superuser | Relative | 0.17 | 0.14 | 0.12 | 0.12 |
| | RMSE | 491.03 | 373.55 | 343.65 | 328.57 |

Table 5.8: Relative error and RMSE error for triangle $T_2$ obtained with a corrupted Oracle.

| **Dataset** | **Error** | $P_r = 0.1$ | $P_r = 0.5$ | $P_r = 0.8$ | $P_r = 1.0$ |
|---|---|---|---|---|---|
| CollegeMsg | Relative | 0.29 | 0.02 | 0.02 | 0.01 |
| | RMSE | 158.74 | 10.73 | 8.31 | 4.29 |
| email-Eu-core | Relative | 0.77 | 0.35 | 0.28 | 0.20 |
| | RMSE | 41.18 | 26.192 | 20.11 | 14.75 |
| sx-askubuntu | Relative | 0.93 | 0.33 | 0.26 | 0.20 |
| | RMSE | 158.25 | 46.11 | 40.73 | 36.05 |
| sx-mathoverflow | Relative | 0.73 | 0.46 | 0.31 | 0.20 |
| | RMSE | 62.84 | 41.86 | 36.20 | 23.55 |
| sx-superuser | Relative | 0.62 | 0.42 | 0.35 | 0.26 |
| | RMSE | 104.77 | 63.61 | 58.20 | 50.79 |

Table 5.9: Relative error and RMSE error for triangle $T_4$ obtained with a corrupted Oracle.

| Dataset | $P_r$ | Average time (s) | Speed-up factor |
|---|---|---|---|
| CollegeMsg | 0.1 | 0.20 | 94.70 |
| | 0.5 | 0.24 | 78.92 |
| | 0.8 | 0.26 | 72.85 |
| | 1.0 | 0.27 | 70.15 |
| email-Eu-core | 0.1 | 0.80 | 360.49 |
| | 0.5 | 1.43 | 201.67 |
| | 0.8 | 1.48 | 194.86 |
| | 1.0 | 1.59 | 181.38 |
| sx-askubuntu | 0.1 | 0.89 | 4.71 |
| | 0.5 | 0.90 | 4.60 |
| | 0.8 | 0.92 | 4.55 |
| | 1.0 | 0.94 | 4.46 |
| sx-mathoverflow | 0.1 | 0.48 | 4.60 |
| | 0.5 | 0.49 | 4.51 |
| | 0.8 | 0.50 | 4.42 |
| | 1.0 | 0.51 | 4.33 |
| sx-superuser | 0.1 | 1.35 | 4.40 |
| | 0.5 | 1.35 | 4.40 |
| | 0.8 | 1.37 | 4.34 |
| | 1.0 | 1.39 | 4.28 |

Table 5.10: Average times and speed-up factors obtained, considering different levels of corruption.

From the results illustrated by Tables 5.8 and 5.9 we can clearly see that, as we reduce the percentage of retained heavy edges, the performance declines. However, the errors do not increase at the same rate at which the $P_r$ value is decreased. Indeed, for $P_r = 0.5$ we can see that the errors are still quite low, but we are retaining only half of the total heavy edges. This means that a predictor that can classify correctly half of the heavy edges of a temporal network (including the heaviest ones) is already effective in improving the standard sampling algorithm.

From the execution times reported in Table 5.10 it is possible to understand that the size of the list of retained heavy edges matters within our implementation. In practice, the average time would not depend on the Oracle, since we can assume that a real Machine Learning model can perform a prediction in constant time for a given temporal edge. It is also possible to see that, for smaller datasets, the speed-up factors change significantly when considering different values for $P_r$, while for bigger networks the speed-up remains almost the same.

We now consider the case in which the corruption is done over the heaviest edges, which are substituted with light edges taken at random from the temporal network. We devised two different experiments to test the robustness of our algorithm. First, we corrupted a perfect Oracle ($P_r = 1.0$) by changing 20% of its heaviest edges. Then, we applied the same corruption level to an Oracle with $P_r = 0.1$. We will compare the results presented above by showing the effect of the corruption, focusing on the relative errors.

| **Dataset** | $L_c$ | $P_r = 0.1$ | $P_r = 1.0$ |
|---|---|---|---|
| CollegeMsg | 0.0 | 0.38 | 0.03 |
| | 0.2 | 0.50 | 0.54 |
| email-Eu-core | 0.0 | 0.24 | 0.12 |
| | 0.2 | 0.29 | 0.13 |
| sx-askubuntu | 0.0 | 0.22 | 0.14 |
| | 0.2 | 0.23 | 0.23 |
| sx-mathoverflow | 0.0 | 0.27 | 0.15 |
| | 0.2 | 0.23 | 0.14 |
| sx-superuser | 0.0 | 0.17 | 0.12 |
| | 0.2 | 0.16 | 0.15 |

Table 5.11: Relative error for triangle $T_2$ obtained with a corrupted Oracle.

| **Dataset** | $L_c$ | $P_r = 0.1$ | $P_r = 1.0$ |
|---|---|---|---|
| CollegeMsg | 0.0 | 0.29 | 0.01 |
| | 0.2 | 0.46 | 0.64 |
| email- | 0.0 | 0.77 | 0.20 |
| Eu-core | 0.2 | 0.95 | 0.93 |
| sx- | 0.0 | 0.93 | 0.20 |
| askubuntu | 0.2 | 0.73 | 0.99 |
| sx- | 0.0 | 0.73 | 0.20 |
| mathoverflow | 0.2 | 0.70 | 0.60 |
| sx- | 0.0 | 0.62 | 0.26 |
| superuser | 0.2 | 0.68 | 0.39 |

Table 5.12: Relative error for triangle $T_4$ obtained with a corrupted Oracle.

The results obtained show that the Oracle-based algorithm is generally robust against Oracle corruption when considering frequent temporal triangles. Indeed, when looking at the relative errors for the triangle $T_2$ with $P_r = 1.0$, the difference between the errors obtained with and without corruption is quite small. If instead we look at rare motifs, we may obtain very bad estimates, as in the case of the triangle $T_4$. We can also notice that, in general, the gap between relative errors for the same level of corruption $L_c$ and for the same dataset are not really large, meaning that there may be few heavy edges that are making the majority of the temporal triangles, which are lost with the corruption, independently from the value of $P_r$ (since we remove the heaviest edges).

## 5.7 Algorithms Comparisons

In this section we compare the sampling algorithm and the Oracle-based algorithm by looking at the errors and execution times they achieve, the memory occupancy and other statistics.

First of all, we will look at the best counts obtained by both algorithms among the 50 trials. We consider a count as the best one if its associated relative error is the smallest one among all the counts obtained by the algorithm in all the trials. We present the results obtained with different values for the threshold $\rho$, using the perfect Oracle.

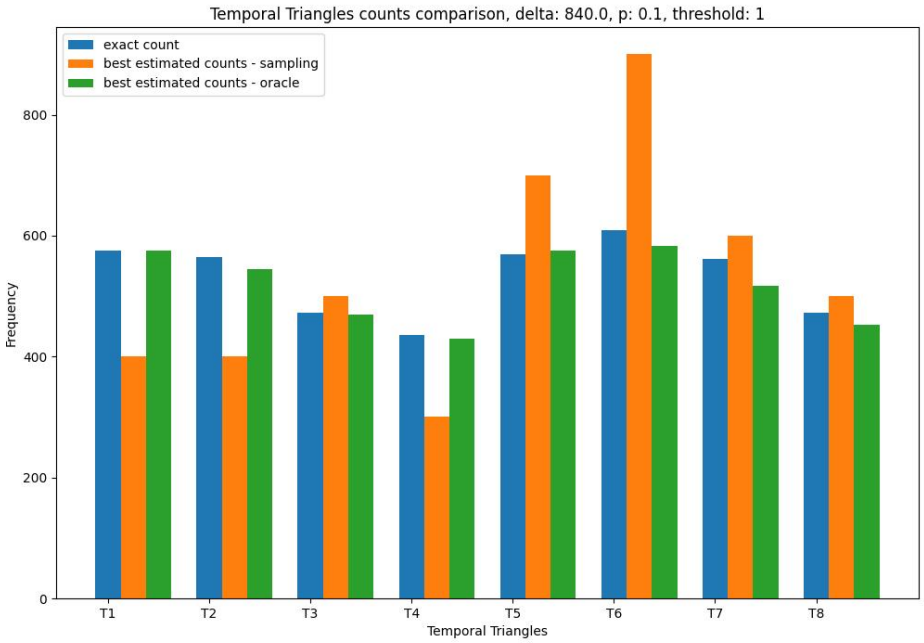


Figure 5.4: Counts obtained by the three algorithms on the CollegeMsg dataset. The threshold value $\rho$ is set to 2.

It is possible to see from these charts that the threshold value does not impact significantly the results obtained. We must consider however that we are using similar values. Higher differences in the choice of $\rho$ may change the results consistently. Overall, it is clear that the Oracle-based algorithm achieves better estimates, as already mentioned before. We now show the results collected for the other datasets. In Figure 5.7, we can notice that rare motifs are captured by the Oracle-based algorithm, while they are neglect by the sampling algorithm.

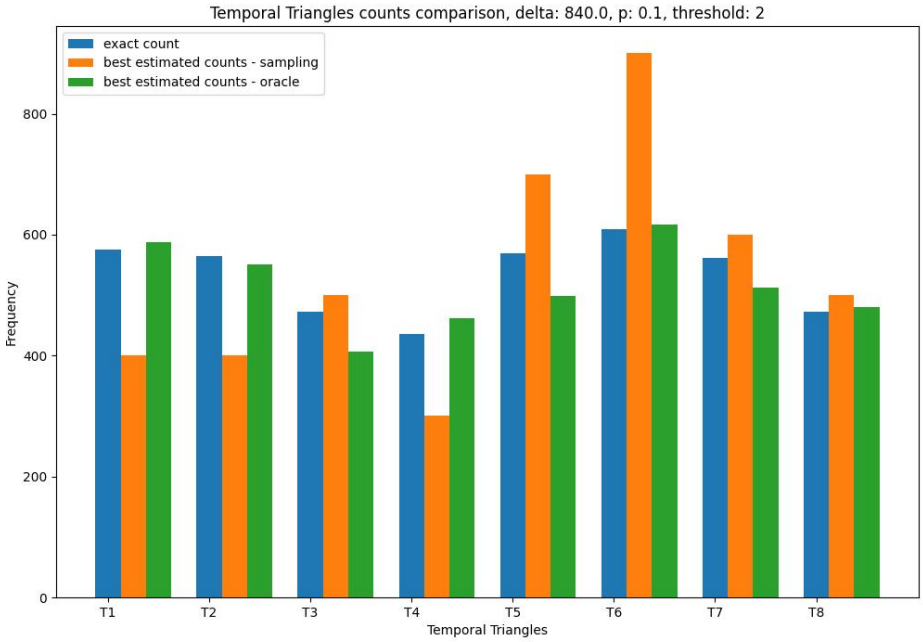


Figure 5.5: Counts obtained by the three algorithms on CollegeMsg with $\rho$ equal to 3.

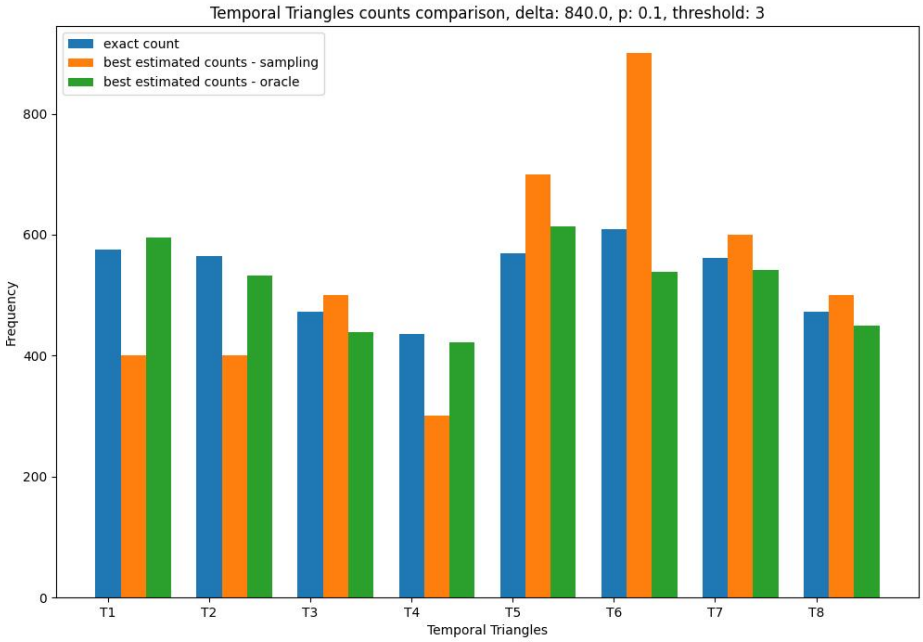


Figure 5.6: Counts obtained by the three algorithms on CollegeMsg with $\rho = 4$.

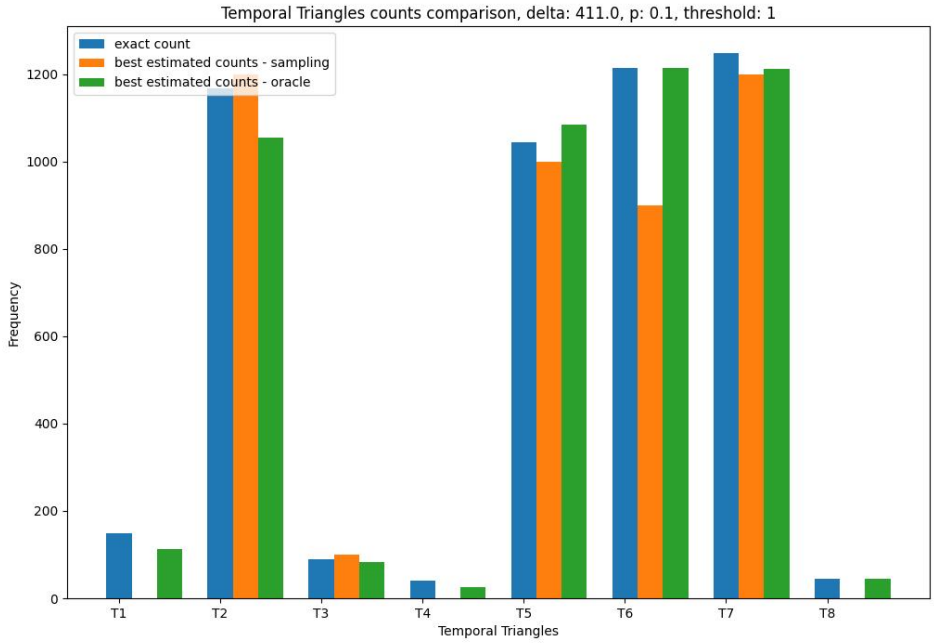


Figure 5.7: Histogram depicting the counts obtained by the three algorithms on the email-Eu-core dataset. The threshold value $\rho$ is set to 2.

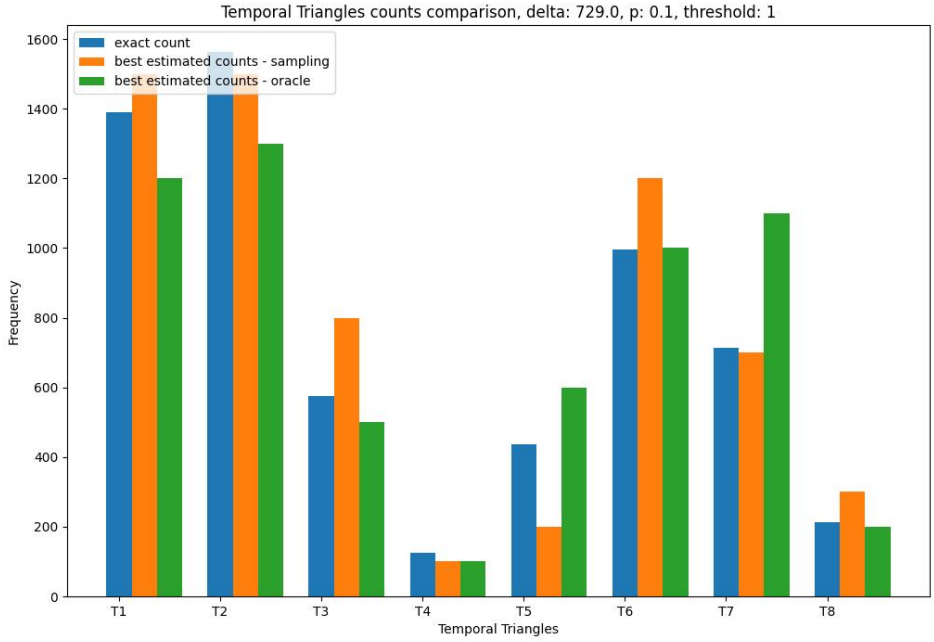


Figure 5.8: Histogram depicting the counts obtained by the three algorithms on the sx-askubuntu dataset. The threshold value $\rho$ is set to 2.

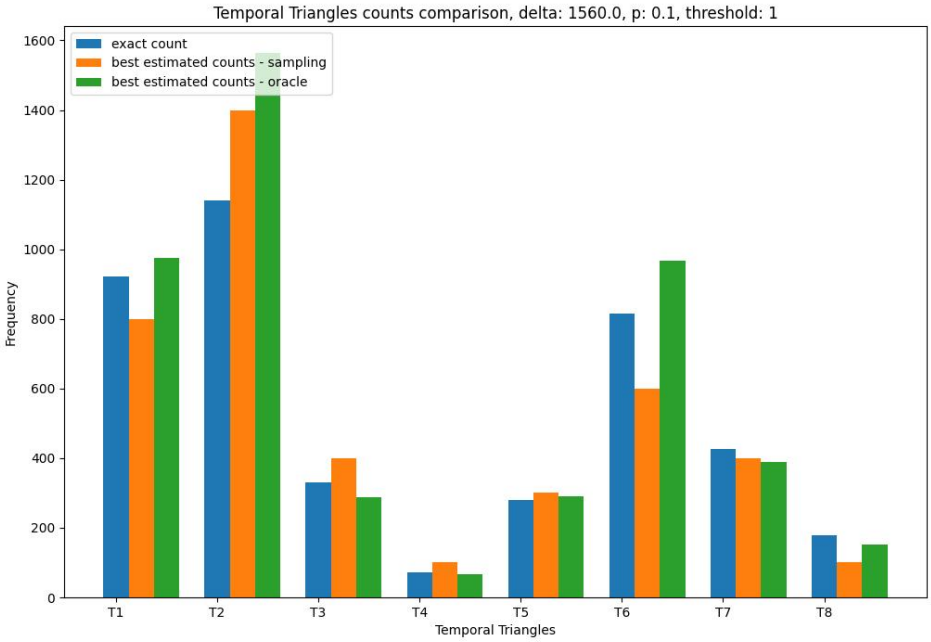


Figure 5.9: Histogram depicting the counts obtained by the three algorithms on the sx-mathoverflow dataset, with a threshold value $\rho$ of 2.

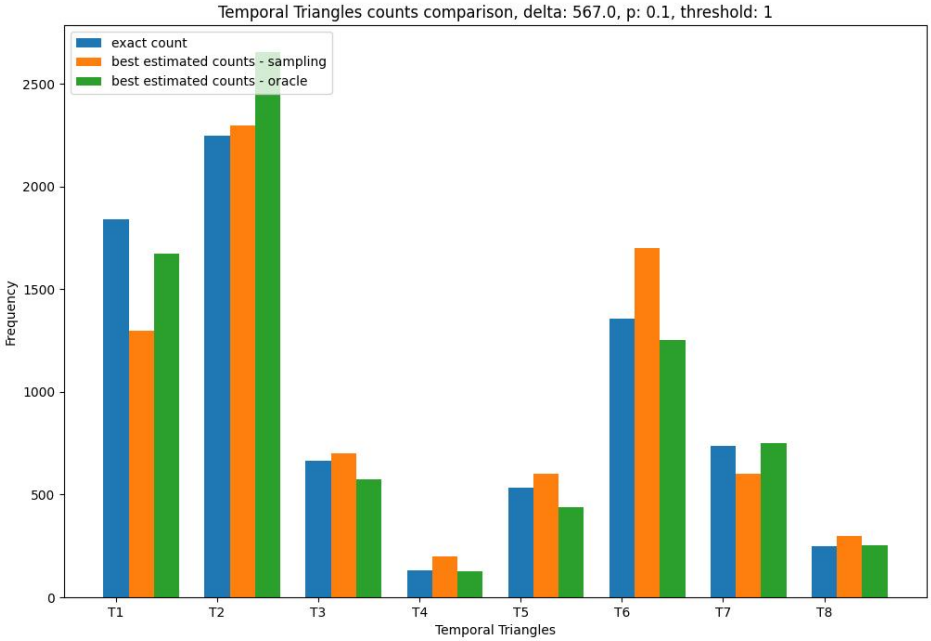


Figure 5.10: Histogram depicting the counts obtained by the three algorithms on the sx-superuser dataset, with a threshold value $\rho$ of 2.

Another important aspect to analyze is the performance of the algorithms on a single trial. To do this, we collected, for a specific triangle type (i.e. $T_6$), the counts obtained by both algorithms on each trial and compared them. We first depict all the counts in a unique histogram, where we inserted an horizontal red line indicating the true count for the specific triangle. We then derived the associated distribution of the counts, for each of the two algorithms. In the end, after computing the mean and the standard deviation of the counts, we depicted the associated gaussian distribution, which allows to have a better intuition on the distributions computed. We have also scaled the two distributions in order to easily compare them. In Figure 5.11, we report an example for the CollegeMsg dataset. The most important thing that we can notice is that the Oracle-based algorithm collects counts that are more often close to the true count. In other words, as the distributions shows, the Oracle-based algorithm has a lower variance than the sampling algorithm. In Figures 5.12 - 5.15, we show the results obtained in the other datasets.

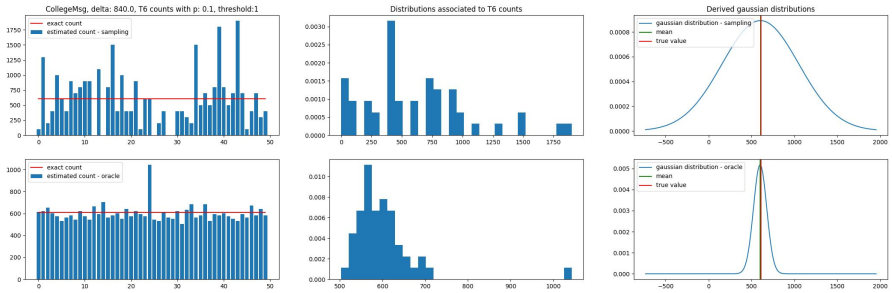


Figure 5.11: Dataset: CollegeMsg. Left: counts obtained with the sampling algorithm (top) and Oracle-based algorithm (bottom) for triangle $T_6$. Center: distribution of the counts derived. Right: visualization of a gaussian distribution with previously computed mean and standard deviation.

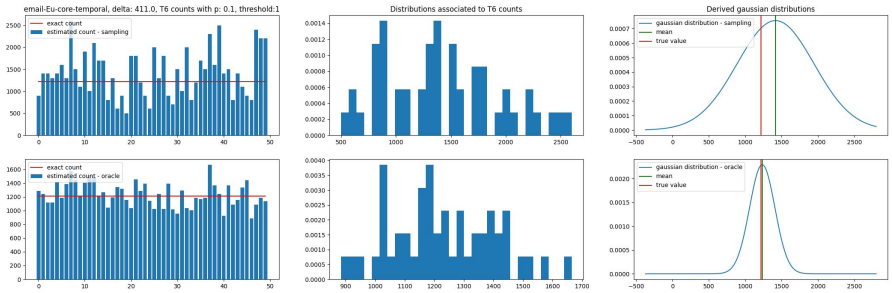


Figure 5.12: Dataset: email-Eu-core.

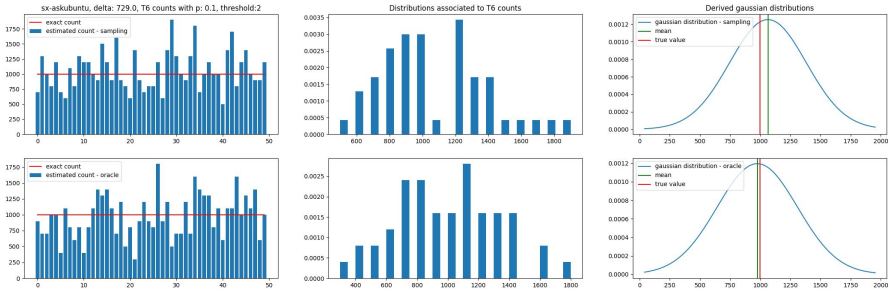


Figure 5.13: Dataset: sx-askubuntu.

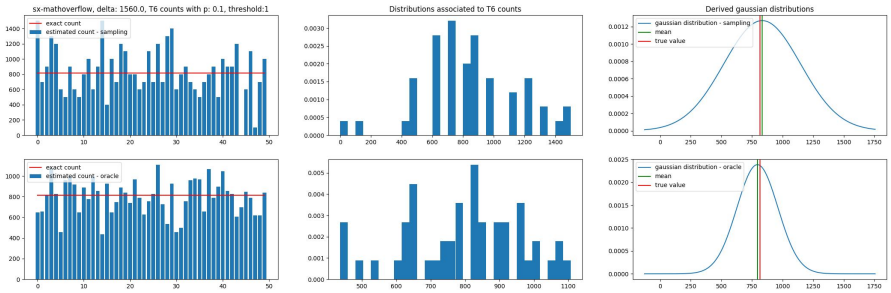


Figure 5.14: Dataset: sx-mathoverflow.

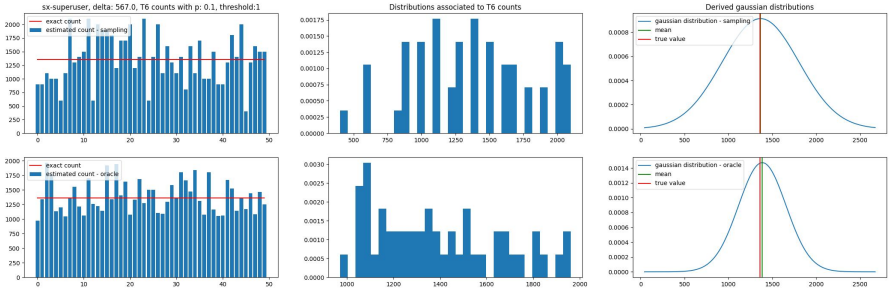


Figure 5.15: Dataset: sx-superuser.

The charts provided prove the effectiveness of the Oracle-based algorithm over the sampling algorithm, since it allows to collect better estimates and with a lower variance.

However, although the Oracle-based algorithm outperforms the sampling algorithm in terms of accuracy, we need to consider also some drawbacks of the approach.

In particular, the execution time required by the Oracle algorithm is substantially larger with respect to the one required by the sampling algorithm. However, it is possible to obtain a trade-off between the accuracy and the time required by constraining the Oracle knowledge. We recall once again that this statements are valid as long as we are considering an implementation that makes use of a list of known heavy edges. In a real scenario, we can assume that a predictor can classify an edge in time $O(1)$.

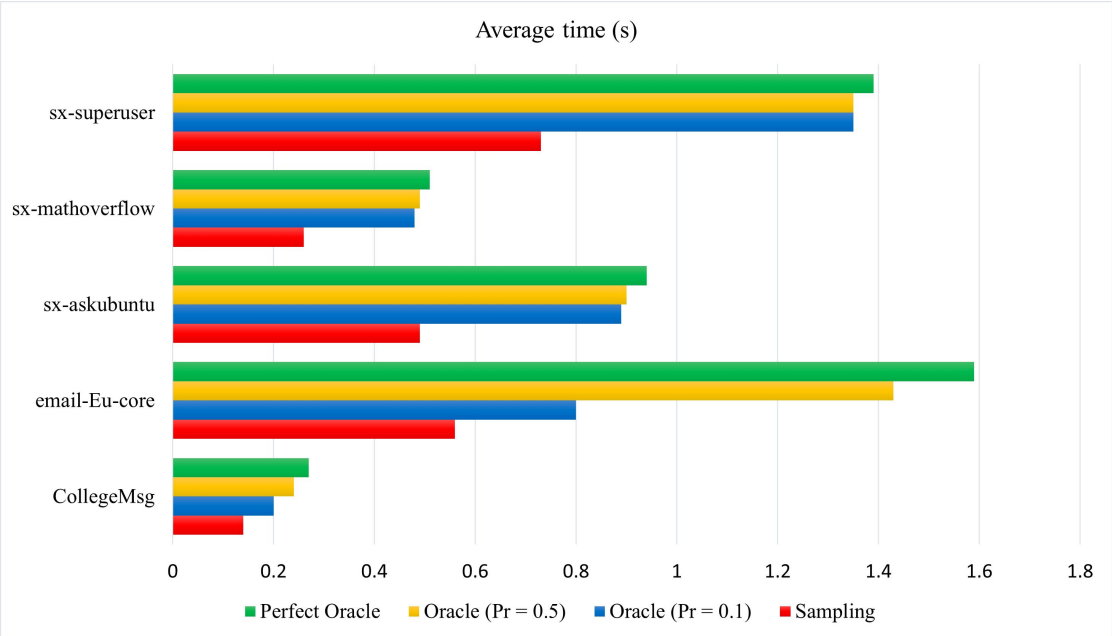


Figure 5.16: Average times obtained by the sampling and Oracle-based algorithms on the five datasets.

We will now consider the memory resources required by both algorithms in order to work correctly. We first show the results obtained for a fixed value of the sampling probability $p = 0.1$. We considered only the memory required to store the edges, during the entire stream. From Figure 5.17, we can notice that the additional memory required for storing the heavy edges is almost negligible. This also shows that the amount of heavy edges is not really high (for the considered threshold, which is of 2), and it is more likely that there are few but really heavy edges.

A further parameter to consider is the value of the sampling probability $p$. As it can be easily deduced, an higher value of $p$ will lead us to better results, since we are sampling more edges. On the contrary, the space and time complexities will increase. We report some representative results in Figures 5.18 - 5.19 obtained

on the CollegeMsg dataset. As already mentioned, the threshold values we have chosen does not seem to have a strong impact on the overall performance of the Oracle-based algorithm. In general, lower thresholds work better since we are storing a larger amount of edges. This of course results in an increase in the execution time (see Figure 5.19).

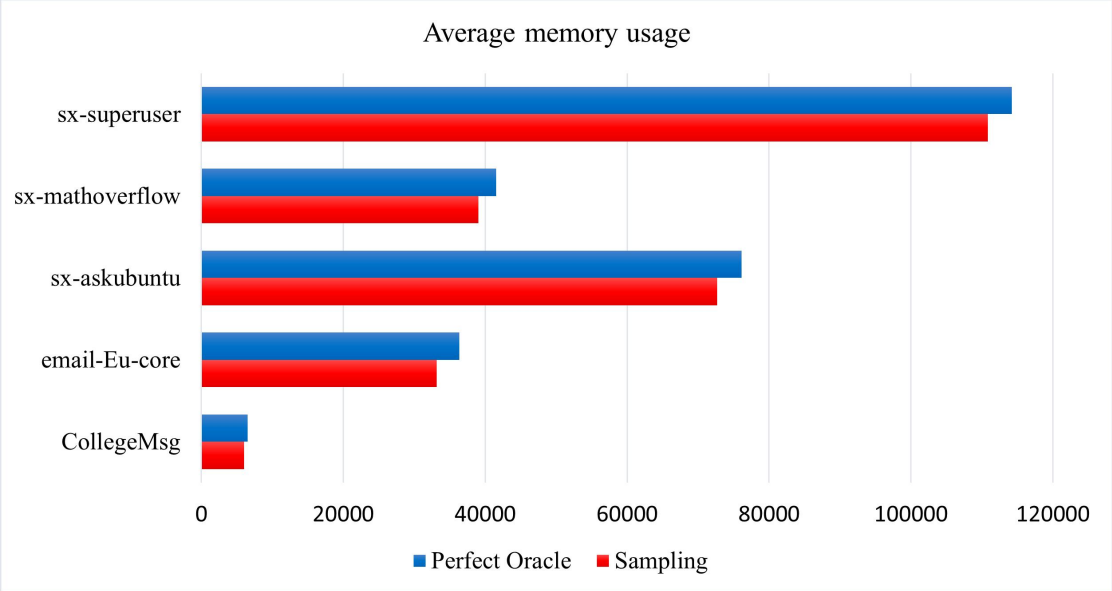


Figure 5.17: Memory usage for the sampling and Oracle-based algorithms on the five datasets. The Oracle threshold $\rho$ is of 2.

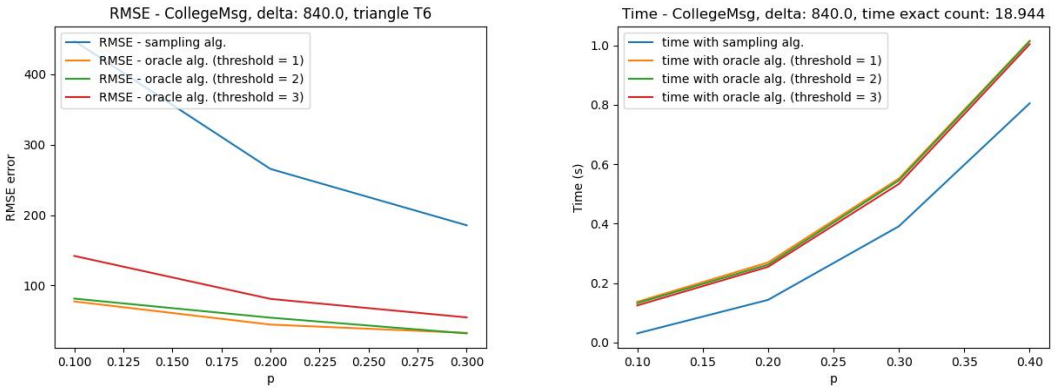


Figure 5.18: Left: RMSE obtained with the sampling and Oracle-based algorithms on CollegeMsg, with increasing value of $p$. Right: average times obtained by the sampling and Oracle-based algorithms on CollegeMsg.

Lastly, we compare the corrupted version of the Oracle with the sampling algorithm, in order to see if the algorithm is robust to prediction errors. In Figures 5.19 and

5.20, we show the relative errors obtained by the sampling algorithm and the corrupted versions of the Oracle, considering the triangles $T_2$ and $T_4$. In particular, we included four different Oracles:

- $O_1$ with $P_r = 0.1$
- $O_2$ with $P_r = 0.5$
- $O_3$ with $P_r = 0.8$
- $O_4$ with $P_r = 0.1, L_c = 0.2$
- $O_5$ with $P_r = 1.0, L_c = 0.2$

We can see that the (corrupted) Oracles $O_2$ and $O_3$ maintain a constant advantage over the sampling algorithm. The Oracle $O_1$ is instead roughly comparable with the sampling algorithm. In general, the second type of corruption seems to have a larger impact on the overall performance of the algorithm. However, the errors obtained are always comparable, even in the case of the Oracles $O_4$ and $O_5$. It is also interesting to highlight that, on the biggest datasets we have considered, the Oracle-based algorithm has almost always an advantage in terms of accuracy. This testify the fact that our approach is both effective and robust to prediction errors when operating on large networks.

In conclusion, we have presented several experiments over the sampling and Oracle-based algorithms which provides evidences of the effectiveness of the Algorithm with Predictions approach. The Oracle-based algorithm obtains lower errors, and it is also robust against corruption of the perfect Oracle. This suggests that the algorithm can work well even with a real Machine Learning predictor. Overall, the approach seems promising for improving the approximation of temporal triangles' counts in large temporal networks exploiting additional statistical information provided by an ad-hoc predictor.

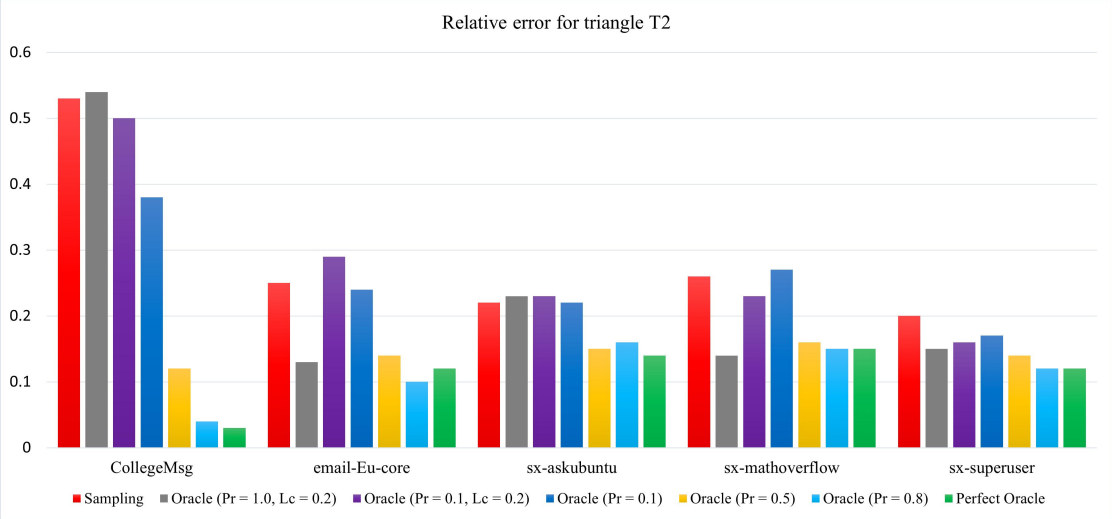


Figure 5.19: Relative error for triangle $T_2$ obtained by the sampling and Oracle-based algorithms (with corrupted variants) on the five datasets.

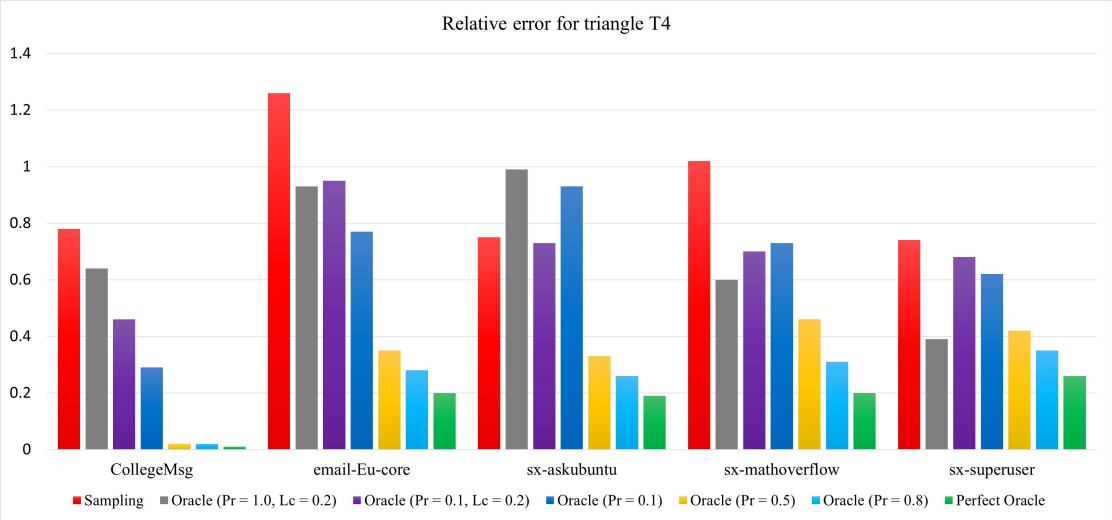


Figure 5.20: Relative error for triangle $T_4$ obtained by the sampling and Oracle-based algorithms (with corrupted variants) on the five datasets.

# Chapter 6

# Conclusions

In this work, we provided a new approach to approximate temporal triangles' counts in large temporal networks, considering a streaming settings. In particular, we devised an algorithm within the "Algorithms with Predictions" framework to improve the performance of a standard sampling algorithm, by providing an ad-hoc predictor with which we can perform an informed sampling. The algorithms were also analyzed in order to guarantee the unbiasness of the results obtained and ensure the correctness of the approach. The experimental evaluation done gave consistent evidence of the effectiveness of the Oracle-based algorithm, suggesting also that it can be extended to work with a real Machine Learning predictor. However, several additional evaluations and research must be done in order to ensure the practical applicability of this approach, including:

- Perform additional experiments on the execution time, by considering additional values of $\delta$ and by improving the original implementation.
- Derive a Machine Learning model that can classify temporal edges according to the heaviness property.
- Compare the approach with other State of the Art techniques, in order to understand further limitations and strengths.

We will leave these as the starting points for future works on the algorithms presented in this thesis.

# Bibliography


[1] Nesreen K Ahmed, Jennifer Neville, Ryan A Rossi, and Nick Duffield. Efficient graphlet counting for large networks. In *2015 IEEE international conference on data mining*, pages 1–10. IEEE, 2015.

[2] Albert-László Barabási. Network science. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 371(1987):20120375, 2013.

[3] Justin Y Chen, Talya Eden, Piotr Indyk, Honghao Lin, Shyam Narayanan, Ronitt Rubinfeld, Sandeep Silwal, Tal Wagner, David P Woodruff, and Michael Zhang. Triangle and four cycle counting with predictions in graph streams. *arXiv preprint arXiv:2203.09572*, 2022.

[4] Matthew T Dickerson and Michael T Goodrich. Two-site voronoi diagrams in geographic networks. In *Proceedings of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems*, pages 1–4, 2008.

[5] Rui Ding, Norsidah Ujang, Hussain Bin Hamid, Mohd Shahrudin Abd Manan, Rong Li, Safwan Subhi Mousa Albadareen, Ashkan Nochian, and Jianjun Wu. Application of complex networks theory in urban traffic network researches. *Networks and Spatial Economics*, 19:1281–1317, 2019.

[6] Jon C Ergun, Zhili Feng, Sandeep Silwal, David P Woodruff, and Samson Zhou. Learning-augmented $k$-means clustering. *arXiv preprint arXiv:2110.14094*, 2021.

[7] Chen-Yu Hsu, Piotr Indyk, Dina Katabi, and Ali Vakilian. Learning-based frequency estimation algorithms. In *International Conference on Learning Representations*, 2019.

[8] Nadav Kashtan, Shalev Itzkovitz, Ron Milo, and Uri Alon. Efficient sampling algorithm for estimating subgraph concentrations and detecting network motifs. *Bioinformatics*, 20(11):1746–1758, 2004.

[9] Arijit Khan and Cuneyt Gurcan Akcora. Graph-based management and mining of blockchain data. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, pages 5140–5143, 2022.

[10] Jure Leskovec and Rok Sosic. Snap: A general purpose network analysis and graph mining library in c++, 2014.

[11] Ron Milo, Shalev Itzkovitz, Nadav Kashtan, Reuven Levitt, Shai Shen-Orr, Inbal Ayzenshtat, Michal Sheffer, and Uri Alon. Superfamilies of evolved and designed networks. *Science*, 303(5663):1538–1542, 2004.

[12] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.

[13] Michael Mitzenmacher and Sergei Vassilvitskii. Algorithms with predictions. *Communications of the ACM*, 65(7):33–35, 2022.

[14] Thy Nguyen, Anamay Chaturvedi, and Huy Lê Nguyen. Improved learning-augmented algorithms for k-means and k-medians clustering. *arXiv preprint arXiv:2210.17028*, 2022.

[15] Ashwin Paranjape, Austin R Benson, and Jure Leskovec. Motifs in temporal networks. In *Proceedings of the tenth ACM international conference on web search and data mining*, pages 601–610, 2017.

[16] Noujan Pashanasangi and C Seshadhri. Faster and generalized temporal triangle counting, via degeneracy ordering. In *Proceedings of the 27th ACM*

*SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 1319–1328, 2021.

[17] Neeru Redhu and Zoozeal Thakur. Chapter 23 - network biology and applications. In Dev Bukhsh Singh and Rajesh Kumar Pathak, editors, *Bioinformatics*, pages 381–407. Academic Press, 2022.

[18] Ilie Sarpe and Fabio Vandin. Oden: simultaneous approximation of multiple motif counts in large temporal networks. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, pages 1568–1577, 2021.

[19] Ilie Sarpe and Fabio Vandin. Presto: Simple and scalable sampling techniques for the rigorous approximation of temporal motif counts. In *Proceedings of the 2021 SIAM International Conference on Data Mining (SDM)*, pages 145–153. SIAM, 2021.

[20] John Scott. Social network analysis: developments, advances, and prospects. *Social network analysis and mining*, 1:21–26, 2011.

[21] Shai S Shen-Orr, Ron Milo, Shmoolik Mangan, and Uri Alon. Network motifs in the transcriptional regulation network of escherichia coli. *Nature genetics*, 31(1):64–68, 2002.

[22] Jingjing Wang, Yanhao Wang, Wenjun Jiang, Yuchen Li, and Kian-Lee Tan. Efficient sampling algorithms for approximate motif counting in temporal graph streams. *arXiv preprint arXiv:2211.12101*, 2022.

[23] Sebastian Wernicke. Efficient detection of network motifs. *IEEE/ACM transactions on computational biology and bioinformatics*, 3(4):347–359, 2006.

[24] Kaige Yang and Laura Toni. Graph-based recommendation system. In *2018 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pages 798–802. IEEE, 2018.

# Acknowledgments

I would like to personally thank my supervisor, Professor Fabio Vandin, for his guidance and for supporting my ideas during the development of this work.