Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master Thesis

Graph sparsification techniques for triangle counting

Matthias Hermann

Course of Study:	Informatik
Examiner:	Prof. Dr. Kurt Rothermel
Supervisor:	Michael Schramm, M.Sc.
Commenced:	October 15, 2019
Completed:	September 2, 2020

Abstract

The triangle count of a graph is a key metric in graph analysis. Especially for social networks, the triangle count is important to assess connectedness of vertices in the graph. However, these social networks in particular can produce large graphs with trillions of edges. In fact, the size of graphs appears to grow faster than the computational resources to analyze and process these graphs. Confronted with similar problems in the past, the solutions for developers of algorithms were oftentimes approximating algorithms. Research in approximate triangle counting algorithms has lead to a multitude of various approximating algorithms. Comparing and understanding the differences in the mechanisms they use to provide faster and more accurate results has therefore become complicated.

This work presents an analysis on existing triangle counting algorithms to improve understanding of which mechanisms work best for fast triangle count approximations. In order to further this understanding even more, an analysis on graph structures and their influence on triangle counts is presented, as well. Results of this analysis include a method for decentralized coordination and reducing communication in distributed computations as well as a method for estimating a triangle count of a graph by using a small sample of vertices and their degree values.

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

BSP Bulk Synchronous Parallel. 21

DAE degree-assisted estimation. 67

PRNG pseudorandom number generator. 36

RNG random number generator. 36

1 Introduction

The evolution of sizes of social networks caused a need for processing huge graphs with billions of vertices and trillions of edges [CEK+15]. Han et al. [HZZ+15] analyzed the user network of Weibo, a Chinese social network, to detect communities, explore structural properties, and compare them with Twitter's network. Retailers, such as Walmart and eBay, or websites like Twitter use graph processing in order to give better recommendations to customers and users [GJL+18; Neo15; Neo19]. In the finance industry, graph analysis is used to perform fraud detection in financial transactions [Neo17]. However, not only large and well known companies process huge graphs. Even small companies may have the need to process graphs with more than one billion edges, as a survey of Sahu et al. [SMS+17] found out.

Many graph processing tasks depend on graph properties like *clustering coefficient* or *transitivity ratio* to calculate their results [TKMF09]. Triangle counting is used in order to calculate these properties and perform graph analysis with them [New03]. As a result, triangle counting is used in several domains. Aggarwal et al. [AS14] analyze networks and their evolution over time, which requires the triangle count as a metric. Becchetti et al. [BBCG08] could detect sources of spam or judge content quality in networks by counting triangles. Eckmann and Moses showed that triangle counting can also be used for analyzing the internet and detecting topics in its structure [EM02].

Processing large graphs results in a need of large amounts of computing power. However, the amounts of data to be processed grow faster than the computational resources available to process this data, according to an industry study sponsored by EMC (now Dell EMC) [GR11; Mit16]. Efficiency of algorithms in terms of runtime and computational resources will therefore only increase in importance. Even nowadays the size of web scale graphs, such as the Yahoo web graph [Res19], causes issues due to the super-linear runtime complexity of some algorithms used for analysis of those graphs [SWW+12]. Due to these issues the following challenges result. Firstly, keeping the runtime of algorithms in the order of minutes instead of hours or even days. Analysis on graphs is oftentimes required to be performed in a timely manner, as information extracted from a graph may be needed in real-time [GJL+18]. Secondly, the size of graphs will certainly grow even further and therefore measures need be taken in order to be able to manage even larger graphs [Sch19].

Historically, similar challenges have been solved in several ways. The rise of distributed systems originated from the need to distribute load off of a single processor and enabled computing resources to be clustered together in order to build powerful clusters capable of more complex computations than single processor machines [Pea72]. Many graph processing frameworks nowadays such as Gunrock [WRO+17], Apache Giraph [Apa] or GraphLab [LBG+12] already use distributed processing to perform analysis on large graphs, which do not fit into the memory of a single machine or would take too long to process otherwise. Code optimizations do help in making algorithms more efficient, however, there is the limitation of runtime complexity restricting how much an algorithm can be optimized. In algorithm research, approximations have therefore been a research topic for a long time. Approximations allow a trade-off between runtime of an algorithm and accuracy of its

results. Such a trade-off allows to define a margin of error, which is tolerable, in order to reduce consumption of computational resources. Depending on the problem, using an approximation may give sufficiently good results while saving large amounts of computational resources [Joh73].

Approximating algorithms are commonly used in combinatorics on commonly known NP-complete problems such as the set cover problem or the boolean satisfiability problem [Joh73]. In numerical analysis Runge-Kutta methods have been in use for a long time to calculate approximate solutions of differential equations with computers [Mar58]. First steps in using approximation techniques in graph processing were taken by Iyer et al. [IPV+18]. Their GAP system uses sparsification of the input graph to reduce the amount of data to be processed. Sparsification of a graph is usually performed by sampling certain vertices or edges of a graph and ignoring all others. Schramm [Sch19] reinterpreted graph sparsification as dropping of messages, which are exchanged between vertices during the execution of an algorithm. The goal of approximation techniques is to reduce overall execution time of an algorithm, as well as its resource consumption. This effect may be achieved by reducing the time and memory requirements for materializing the graph structure in memory or by reducing the overall amount of calculations to be performed. In distributed graph processing, the overhead caused by communication between computers participating in a calculation may be reduced by limiting the amount or size of messages exchanged in order to speed up processing and limiting network usage.

In order to further improve upon approximating algorithms for triangle counting, the central parts of this work are an analysis of structures in graphs, which may correlate with triangle counts, and an analysis of preexisting triangle counting algorithms. Both aim at generating insights into how triangle counting can be made more efficient by understanding which structures in graphs and features of algorithms can be used to improve estimates and runtimes. As a result of this analysis, a new method for estimating triangle counts in graphs using a small sample of local triangle counts of vertices and structural properties is presented. Furthermore, a method for decentralized coordination and reducing communication in distributed computations using cryptographic hash functions is presented. While the latter does not improve upon estimates, it may reduce runtime by reducing resource usage caused by communication between processes in a distributed computation. All algorithms covered in this work were implemented using the graph processing framework Apache Giraph [Apa] such that comparisons are done using a common baseline. The graphs used for analysis and algorithm comparisons are chosen from various domains to reflect the diversity of real-world graphs.

1.1 Thesis organization

This work is structured as follows. Chapter 2 introduces all the background knowledge required for comprehension of this work. This includes the problem of triangle counting, estimating triangle counts in sparse graphs, the computational model used by Apache Giraph – the graph processing framework used in this work –, and statistical measures used in the analysis.

Chapter 3 presents related work in the context of triangle counting – especially algorithms –, approximate computing in general, and the use of hash functions for coordination tasks. It closes with the contributions of this work.

Chapter 4 presents a method of decentralized coordination and reducing runtime for distributed computations, which helped with the implementation of a triangle counting algorithm used in the algorithm comparisons.

Chapter 5 contains the analysis of graph structures and their correlation with triangle counts as well as performance comparisons of existing triangle counting algorithms.

Chapter 6 presents a new method of estimating a triangle count in a graph with the assistance of structural properties of the graph.

Chapter 7, the last chapter, concludes this work with a summary of its results and contributions as well as possible starting points for future work.

2 Background

It is necessary to first present the definition of triangle counting used in this work before algorithms on triangle counting and other related works are presented. In addition to that definition, the general approach of computing a triangle count estimate using a sparse graphs is presented. This is followed by an overview of the graph processing framework used in this work alongside its underlying computational model. At the end of the chapter, a selection of correlation coefficients are presented, which are used for statistical evaluations later on.

2.1 Triangle Count

The triangle count is a property of a graph $G(V, E)$ with $V = \{v_1, ..., v_n\}$ being the set of vertices and $E \subseteq \{(u, v) \mid u, v \in V\}$ being the set of edges. A triangle in a graph $G(V, E)$ is a set of three vertices $u, v, w \in V$, which are connected to each other by one edge each, i. e.

$$\begin{aligned} &\{u, v, w\} \text{ is a triangle in } G(V, E) \\ \Leftrightarrow &\{u, v, w\} \subseteq V \wedge \{(u, v), (u, w), (v, w)\} \subseteq E. \end{aligned}$$

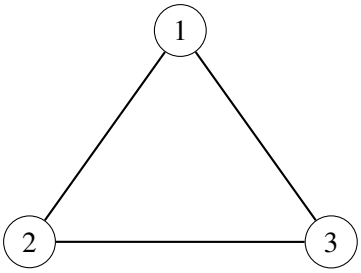
Let $N(v) = \{\, u \in V \mid (v, u) \in E \,\}$ be the neighborhood of a vertex v, i. e. the set of vertices which v is connected to by an edge and $d(v)$ be the degree of a vertex v, i. e. the size of its neighborhood $|N(v)|$ or the amount of out-going edges incident to that vertex v. In order for a vertex v to be part of a triangle it must hold that $d(v) \geq 2$. Furthermore, only vertices $u, w \in N(v)$ can form a triangle $\{u, v, w\}$ with v.

Triangles are to be distinguished from open triples, which are missing any one of the three edges required to form a triangle as depicted in figure 2.1. An open triple is said to be centered around a vertex v, if it is the vertex in the open triple connected to the other two vertices, i. e.

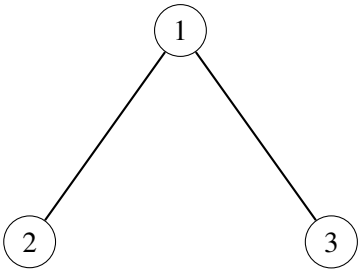
$$\begin{aligned} &\{u, v, w\} \text{ is an open triple in } G(V, E) \text{ centered around vertex v} \\ \Leftrightarrow &\{u, v, w\} \subseteq V \wedge \{(u, v), (v, w)\} \subseteq E \wedge \{(u, w),\ (w, u)\} \nsubseteq E. \end{aligned}$$

The definition of "triangle" in this work extends to both, directed and undirected graphs. Edges in undirected graphs are interpreted as two separate, directed edges between the same two vertices with opposite directions, i. e. the undirected graph is interpreted as directed graph. Edges in a directed graph are duplicated with opposite direction, such that for every directed edge (u, v) another edge (v, u) with opposite direction is added but without introducing duplicate edges.

Triangle counting is the task of counting how many distinct triangles are contained in a graph $G(V, E)$. This is not to be confused with obtaining the local triangle count of a vertex, which represents the number of distinct triangles that vertex is a part of. Unless specified otherwise, triangle counting in this work refers to obtaining a count of triangles in the complete graph. Besides



(a) A simple triangle consisting of three vertices 1, 2, and 3.



(b) An open triple centered around vertex 1.

Figure 2.1: Difference between a triangle and an open triple.

merely counting all triangles in a graph, i. e. producing a single value indicating the amount of triangles, one can also enumerate all triangles, i. e. produce a set of vertex triples, which correspond to the triangles found in the graph. However, since enumerating triangles in a sparse graph could lead to confusing interpretation of results when comparing with results of the complete graph, this work focuses on merely counting triangles.

When counting triangles, most of the time only distinct triangles are considered. Borrowing the notation of Hasan and Dave [HD17], let $T(G)$ be the set of all triangles contained in a graph $G(V, E)$ and $vert(t)$ and $edg(t)$ are the sets of vertices and edges forming a triangle t, respectively. Then, the set of distinct triangles Λ is defined as:

$$\forall t_1, t_2 \in T(G) : t_1 \in \Lambda \wedge t_2 \in \Lambda \implies vert(t_1) \neq vert(t_2) \vee edg(t_1) \neq edg(t_2)$$

The result of enumerating all triangles in a graph is the set of distinct triangles Λ, whereas the result of merely counting all triangles in a graph is the size of the set of distinct triangles $|\Lambda|$. This implies that an algorithm able to enumerate all triangles in a graph can also produce a count of all triangles, but not the other way round.

2.2 Estimating triangle counts on sparse graphs

Evaluating the triangle count Λ of a graph can be achieved by simply running any exact triangle counting algorithm on the graph. In the case of sparse graphs, which are produced by removing parts of a graph, the triangle count Λ_s produced by running a triangle counting algorithm on the sparse graph has to be corrected for the amount of triangles missed due to sparsification of the graph. A triangle count estimate Λ_e for the original graph is therefore produced from the triangle count Λ_s of the sparse graph by applying a correction factor c_Δ to it as follows:

$$\Lambda \approx \Lambda_e = c_\Delta * \Lambda_s \tag{2.1}$$

The value of this correction factor c_Δ always depends on the method of sparsification of the original graph.

Many sparsification techniques discussed later on in section 3.1.2 are Bernoulli processes, i. e. they are built based on a series of random and independent decisions regarding which edges, vertices or triples to sample or to drop [FEHP10]. Some of them focus, for example, on triangles as a set of

three edges from which edges are kept with probability p and removed with probability $1 - p$. In order to produce a correct estimate of the total amount of triangles present in the original graph, the amount of triangles not counted due to not sampling the corresponding edges has to be accounted for. If an algorithm only samples and counts a third of all triangles in the graph, the resulting count would have to be multiplied by a correction factor of three to receive a correct estimate of the triangle count in the original graph. In the case of Bernoulli processes, however, determining the correction factor is more complicated than multiplying by a constant factor.

When sampling edges of a triangle independently of one another, every edge of a triangle has the same probability p of being sampled. Sampling of edges in a triangle can therefore be seen as a Bernoulli process of length $n = 3$ and with success probability p. Let X be the random variable modeling the amount of edges sampled from a triangle. The probability of sampling k edges from the set of three edges forming a triangle is given by the probability mass function of the binomial distribution with $n = 3$ and p being the sampling probability:

$$P(X = k) = \binom{3}{k} * p^k * (1 - p)^{3-k} \tag{2.2}$$

The correction factor for such sampling procedures extrapolates the amount of triangles found in the sample to an estimate of triangles in the complete graph. It is given by the multiplicative inverse of the probability for sampling all vertices or edges required to count a triangle in the sparse graph. Assuming an algorithm requires all three edges of a triangle to be sampled in order for it to be counted, the correction factor would be $c_\Delta = P(X = 3)^{-1} = p^{-3}$. In the case of an algorithm requiring at least two out of three edges of a triangle to be sampled, the correction has to account for both cases, $k = 2$ and $k = 3$. The resulting correction factor is given by:

$$c_\Delta = \frac{1}{P(X \geq 2)} \tag{2.3}$$

with:

$$\begin{aligned} P(X \geq 2) &= \binom{3}{3} * p^3 * (1 - p)^0 + \binom{3}{2} * p^2 * (1 - p)^1 \\ &= -2p^3 + 3p^2 \end{aligned}$$

These formulas are all fit for use in combination with sampling procedures which are based on Bernoulli processes. Note that not all algorithms have to use a Bernoulli process to sample a graph. If that is the case, however, the parameters n and k of the binomial distribution as well as threshold value for the random variable X have to be chosen with care, such that these parameters correctly model the sampling process.

2.3 Bulk Synchronous Parallel

Bulk Synchronous Parallel (BSP) is a computing model describing the execution of a computation in the context of multiple separate components working together in parallel and communicating using messages. It was developed to model parallel and distributed computations, which previous models such as the *von Neumann* architecture alone were unable to [Val90]. This section first describes the BSP model, followed by an outline of two implementations of this model in the context of graph processing, namely Pregel and Apache Giraph.

2.3.1 The model

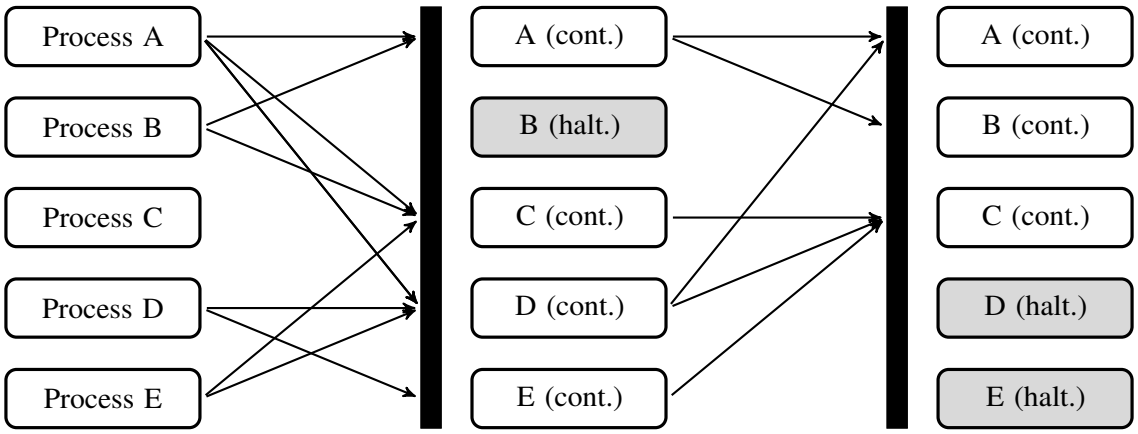


Figure 2.2: Exemplary execution of a computation in the Bulk Synchronous Parallel model

The way in which sequential computers with a single CPU and main memory work is described by the *von Neumann* architecture [Neu93]. It is an abstract bridging model describing the way in which a computation, i. e. the execution of a program, would run on computers implementing that model. Similarly, the BSP model has been proposed by Leslie Valiant [Val90] for modeling computations on parallel computers. Such a model allows abstractions of a concrete hardware platform to be made when designing algorithms with available capabilities in mind.

In the BSP model, as it is proposed by Valiant [Val90], there exist multiple *components* responsible for performing computations and acting as memory storage. Communication is enabled by a *router* delivering messages between components. Additionally, components are synchronized completely or partially at regular intervals. The segments between two synchronization points are called *supersteps*. During such a superstep each component receives messages, performs local computations, and sends messages to other components. However, messages sent in one superstep are available at the receiver for processing earliest in the following superstep. Note that this model does not use shared memory. Instead, shared memory would need to be simulated using messaging.

Figure 2.2 visualizes a potential execution of a computation in the BSP model. The processes A to E each perform computations, exchange messages, and are synchronized at regular intervals, which is indicated by the black barriers in the figure. At the beginning of the computation, all processes are active and can autonomously decide to become inactive. Once they receive a message, an inactive process becomes active again. The computation is terminated once all processes become inactive and no message is in transit.

2.3.2 Pregel & Apache Giraph

In the field of graph processing, Pregel [MAB+10] is a framework for large-scale computations on graph data and is based on the BSP model. It follows the "Think-like-a-vertex" approach in which the computation is formulated from a vertex's point of view, i. e. a graph's vertex corresponds to a component in the BSP model.

Pregel assumes directed graphs, in which edges are associated with their source vertex. Every vertex has an identifier used for messaging and a user-modifiable value, which acts as output value of that vertex. Within each superstep of a BSP computation, every vertex executes the same user-defined `compute()`-function. In that function, a vertex may perform local computations, change its value or the value of outgoing edges, receive messages sent by other vertices in the previous superstep, send messages to other vertices of which their identifier is known, and even change the structure of the graph. A computation proceeds as long as at least one vertex is still active. All vertices are active at the start of a computation. They can become inactive by voting to halt and become active again when they receive a message. The computation is considered terminated when all vertices are inactive and no message is in transit.

Apache Giraph [Apa] is the open source implementation of this Pregel model of graph processing in the Hadoop ecosystem. Apache Hadoop is a framework for performing distributed computations on large datasets. Giraph was built using the Hadoop framework with parallelization and fault tolerance in mind to enable computations on large graphs using commodity hardware. A Giraph computation can be run on a regular Hadoop cluster. Reading inputs and writing outputs is done using the Hadoop File System (HDFS), which is a distributed file system designed for storage of large amounts of data for processing by Hadoop clusters. Martella et al. [MSL15] further explain the details of processing graphs using Apache Giraph on a Hadoop cluster.

Apache Giraph in version 1.2.0, which is the latest stable release at the time of writing, is the graph processing framework used in this work for analyzing triangle counting algorithms. The most important reasons for this choice are the BSP programming model and its free availability. The programming model is easy to understand and to program for but also allows for easy analysis of how algorithms operate. Additionally, an analysis performed by Varbanescu et al. [VVL+15] showed that performance of graph processing is strongly influenced by the executed algorithm and the input graph. Guo et al. [GBV+14] could find no clear winner among the frameworks they have tested. Therefore, the overall performance of the graph processing framework had less weight than an easy implementation of algorithms and easy analysis of algorithms' behavior when choosing the framework to be used.

2.4 Statistical measures

Detecting a provable correlation, i. e. a statistical relationship, between two variables in a bivariate dataset can indicate that predictions of one variable using the other variable may be possible [ZTS03]. Measures to define the strength of such a correlation are called correlation coefficients. They take into account all datapoints of the available bivariate data and produce a number, which specifies how closely the two variables of the data follow a certain type of statistical relationship. There are three correlation coefficients used in this work: Pearson's correlation coefficient, which is the most commonly-used one and can detect linear relationships in bivariate data, Spearman's correlation coefficient, which is slightly more robust and can detect monotonic, non-linear relationships, and the so called distance correlation coefficient, which is able to detect more general relationships in data.

The simplest kind of such a relationship is a linear one, in which the data can ideally be described by a linear equation or a straight line. Pearson's correlation coefficient r detects a linear relationship between two variables X and Y in the data and measures its strength. It is defined by the covariance

of the two variables X and Y divided by the product of their respective standard deviations [DE12; MBL+12]:

$$r = \frac{Cov(X,\ Y)}{\sigma_X * \sigma_Y} = \frac{\sum_{t=1}^{n}(X_t - \overline{X}) * (Y_t - \overline{Y})}{\sqrt{\sum_{t=1}^{n}(X_t - \overline{X})^2} * \sqrt{\sum_{t=1}^{n}(Y_t - \overline{Y})^2}} \tag{2.4}$$

Values around 0 indicate no relationship between the two variables, while values near ±1 indicate a perfect linear relationship [MBL+12].

A disadvantage of Pearson's r is its restriction to linear relationships. For non-linear relationships which can be described by a monotonic function, Spearman's rank correlation coefficient can be used. It does not use the actual values of the variables X and Y, but the rank of those values, i. e. the smallest value of a variable receives a rank of one, the next biggest value of that variable – no matter the difference between those two values – receives a rank of two etc. Using the ranks of each datapoint instead of the datapoints themselves, the correlation coefficient is calculated using the same equation (2.4) as for Pearson's r. By ignoring the actual differences between values and just examining their rank, Spearman's r can detect non-linear relationships of the form of e. g. a monotonic and cubic polynomial or sigmoid functions. [DE12]

The last correlation coefficient used in this work is the so called distance correlation coefficient. The name is derived from the distance of the joint characteristic function $f_{X,Y}$ to the product of the marginal characteristic functions f_X and f_Y of two random vectors X and Y, i. e. the measures for which the correlation coefficient is calculated. Detailed explanations on how this measure is calculated can be found in the work of Szekely et al. [SRB07]. This correlation coefficient is even more general and is defined for arbitrary random vectors with finite first moments, i. e. finite expectation. As a result, any correlation not detected by Pearson's or Spearman's correlation coefficients should be detected by the distance correlation coefficient.

3 Related work

In this chapter, results of prior research are discussed. First and foremost this includes a selection of existing triangle counting algorithms, both exact and approximate. This is followed by research on approximate computing in general and use cases of hash functions for coordination tasks. This chapter closes with the contributions this work provides.

3.1 Triangle Counting algorithms

There are several types of algorithms for counting triangles, which can be categorized by assumed access restrictions on the input data. In the following, the focus lies on algorithms assuming random access on the input data, i. e. the input graph is available as a whole. These algorithms can in turn be grouped into exact algorithms, which consider every single triangle present in a graph, and approximate algorithms, which only consider parts of a graph and calculate an estimate of the total amount of triangles present in the graph.

3.1.1 Exact triangle counting algorithms

Schank and Wagner [SW05] compare in their work the most common exact triangle counting algorithms found in literature. The simplest algorithm for enumerating triangles is the `NodeIterator` algorithm (see algorithm 3.1), which iterates over all vertices of the graph. For every vertex v the algorithm checks for all distinct pairs of neighbors of v, whether both neighbors are connected by an edge. If such a pair of neighbors is connected, a triangle is found consisting of the two neighbor vertices and the vertex v itself. The runtime complexity of this algorithm is $\Theta(|V| * d_{max}^2)$ with d_{max} being the maximum vertex degree in the input graph $G(V, E)$. The `EdgeIterator` algorithm (see algorithm 3.2) is very similar, but iterates over edges instead of vertices. Triangles are formed by intersecting the adjacency lists, i. e. the lists of neighbors, of both vertices incident to the current edge. Every common neighbor of both vertices forms a triangle with them. Its runtime complexity is commonly stated as $\Theta(m^{\frac{3}{2}})$, which is equivalent to the runtime complexity of the `NodeIterator` algorithm and the proven optimum for enumeration of triangles [SW05].

Another enumerating algorithm, is the so called `Forward` algorithm (see algorithm 3.3). It was developed by Schank and Wagner [SW05] and improves upon the `EdgeIterator` algorithm. At the beginning, the set of edges is sorted by the degree of the source vertex in increasing order. All edges $(u, v) \in E$ for which the degree of the source vertex u is larger than the degree of the destination vertex v are ignored. The data structure $A(v)$ of a vertex $v \in V$ is initially empty, but will be filled during execution with all neighboring vertices with a lower or equal degree. For the remaining edges $(u, v) \in E$, the common neighbors $w \in A(u) \cap A(v)$ of the vertices u and v form triangles $\{u, v, w\}$ with the two vertices. These triangles are then output as the result of the

Algorithm 3.1 `NodeIterator`

Input: $G(V, E)$
Output: Λ
  $T \leftarrow \emptyset$
  for all $v \in V$ **do**
    for all $(u, w) \in \mathrm{N}(v) \times \mathrm{N}(v)$ **do**
      if $(u, w) \in E \wedge u \neq w$ **then**
        $T \leftarrow T \cup \{\{u, v, w\}\}$
      end if
    end for
  end for

Algorithm 3.2 `EdgeIterator`

Input: $G(V, E)$
Output: Λ
  $T \leftarrow \emptyset$
  for all $(u, v) \in E$ **do**
    for all $w \in \mathrm{N}(u) \cap \mathrm{N}(v)$ **do**
      $T \leftarrow T \cup \{\{u, v, w\}\}$
    end for
  end for

algorithm. Alternatively, the same triangle enumeration can be achieved using preprocessing by first removing all edges starting at vertices with a higher degree and ending in vertices with a lower degree and then using the regular `EdgeIterator` algorithm. The runtime complexity is $\Theta(m^{\frac{3}{2}})$, as well; however, it is still significantly faster due to not counting triangles redundantly.

Algorithm 3.3 `Forward`

Input: $G(V, E)$
Output: Λ
  $T \leftarrow \emptyset$
  $E_s \leftarrow \textsc{sort}(E)$
  for all $v \in V$ **do**
    $A(v) \leftarrow \emptyset$
  end for
  for all $(u, v) \in E_s$ **do**
    if $\textsc{deg}(u) \leq \textsc{deg}(v)$ **then**
      for all $w \in A(u) \cap A(v)$ **do**
        $T \leftarrow T \cup \{\{u, v, w\}\}$
      end for
      $A(v) \leftarrow A(v) \cup \{u\}$
    end if
  end for

Another exact triangle algorithm commonly found in literature is the AYZ algorithm developed by Alon, Yuster and Zwick [AYZ97]. Kolountzakis et al. note memory requirements to be an issue even for medium-sized graphs, due to the algorithm being based on matrix multiplications. Because of this practical obstacle, the algorithm will not be used in analysis and comparisons in this work. However, the conceptual idea behind the algorithm is still worth mentioning. In the AYZ algorithm, the set of vertices are split into two sets of high-degree and low-degree vertices according to a threshold Δ. The low-degree vertices execute the algorithm NodeIterator. Low-degree vertices do not have to check many pairs of neighbors to find all triangles and since NodeIterator scales quadratically in the vertices' degrees it is best to avoid high-degree vertices executing NodeIterator. For the remaining high-degree vertices, a fast matrix multiplication is used instead. It is therefore an early example of improving runtime of triangle counting algorithms by taking vertices' degrees into account.

3.1.2 Approximate triangle counting algorithms

Exact triangle counting algorithms, which count all triangles in a graph, cannot be more efficient in terms of runtime complexity than those described in the previous section [SW05]. However, obtaining a count of triangles can be improved upon by using sparsification techniques and approximating the result. In order to speed up a computation, small errors in the results are assumed to be tolerated. Results are not exact anymore when using sparsification techniques, but estimates of a graph's triangle count using these methods can still be fairly accurate. In general, when applying sparsification techniques, parts of the graph or messages exchanged during the computation are removed in a probabilistic manner.

Tsourakakis et al. [TKMF09] proposed such an approximating triangle counting algorithm called DOULION. In the first step of DOULION, edges of the input graph $G(V, E)$ are sampled with probability p and removed with probability $1 - p$ to form a sparse graph G_s. The second step of DOULION consists of executing an exact triangle counting algorithm on this sparse graph G_s. In order to estimate the triangle count in G, it is necessary to know how many triangles are lost in the process of sparsification. Finding a triangle in G_s requires for all three edges of the triangle to be sampled, which happens with probability p each. In total, there is a probability of p^3 for a triangle in G to remain in G_s and therefore the triangle count of G can be obtained by multiplying the triangle count in G_s with a correction factor of p^{-3}.

Etemadi et al. [ELT16] improved upon DOULION in order to increase the chance of counting a triangle of G in G_s. For lack of a better name, that algorithm is referred to in this work as ELT. Similarly to DOULION, algorithm ELT samples edges in G with probability p and removes them with probability $1 - p$. However, when an open triple is encountered in the sparse graph G_s, ELT checks whether the missing edge to form a triangle is present in the original graph G and still counts the triangle, if that edge does exist. As a result, only two edges of a triangle have to remain in G_s in order for it to be counted. The correction factor in this case is $(-2p^3 + 3p^2)^{-1}$ due to triangles with two and three sampled edges being counted. A detailed explanation on how to arrive at this correction factor can be found in section 2.2.

Pagh and Tsourakakis [PT12] propose a different method for sampling a graph called "colorful triangle sampling". Every vertex in the graph is assigned one of N colors, with $\frac{1}{p} = N \in \mathbb{N}$ and p being the sampling probability of an edge. An edge is sampled, if both incident vertices have

the same color, i. e. if the edge is *monochromatic*. As a consequence, if a triangle has already two monochrome edges connecting the vertices, then all three vertices must have the same color and therefore the third edge must be monochrome, as well. Similar to `ELT`, two out of three edges of a triangle being sampled by this coloring scheme are therefore enough to ensure a triangle is counted. However, the correction factor in this case is p^{-2}. The reason for this is that if a triangle has two monochrome edges, which happens with probability p^2, it is directly implied that it is the case for the third edge, as well. While sampling a similar portion of the graph, the advantage of `ColorfulTriangleSampling` over `ELT` is that the former does not require the original graph G to be accessed again.

Instead of sampling triangles, another class of triangle counting approximations samples triples and estimates the graph's transitivity ratio γ, i. e. the ratio of closed triples to the total number of triples. Important to these schemes is uniform sampling of triples, which may not be trivial depending on the degree distribution of the vertices, in order to obtain an unbiased estimate of the transitivity ratio [HD17]. Al Hasan [Has16] describes a methodology to allow for correction of biased triple sampling to allow for simple sampling techniques to be used. Such a simple technique would be to sample a vertex v uniformly at random and then select two of its neighbors uniformly at random, as well. Triple sampling as described in the work of Hasan and Dave [HD17] with that sampling correction is used in this work for further comparisons. It is referred to as `TripleSampling` and the count of triangles produced by it has to be multiplied with a correction factor of $\frac{1}{p}$ to arrive at the final estimate. Similarly, probabilistic versions of `NodeIterator` and `EdgeIterator` have been proposed, which sample a fraction of vertices or edges, respectively [HD17; RH13]. The probabilistic version of `NodeIterator`, which randomly samples vertices from a graph, is referred to as `VertexSampling` in the following. It uses a correction factor of $\frac{1}{p}$, as well. A probabilistic version of `EdgeIterator` is not considered any further, as an implementation in Apache Giraph would be nearly identical to an implementation of the algorithm `DOULION`.

In scenarios with restricted access, algorithms based on random walks can be used to sample the graph. However, these algorithms can only estimate the transitivity ratio, i. e. the number of triples in the graph must be known in order to estimate the triangle count. Hasan and David [HD17] discuss two types of random walk: walking over vertices and walking over triples. Random walks over vertices work similar to sampling algorithms described in the previous paragraph. For each vertex, a pair of neighboring vertices is sampled and the walk continues on to a neighboring vertex according to some stationary probability distribution. Similarly, the restrictions for estimating an unbiased transitivity ratio also apply with random walks over vertices, however, the same correction method described by Al Hasan [Has16] can be used to correct for any bias. Random walks over triples differ from a random walk over vertices only in that they consider triples as a whole, not single vertices. A corresponding neighborhood definition for triples must be determined in order to find the next step in the walk. One such definition could be: two triples are neighbors, if they have two out of three vertices in common. Sampling triples uniformly still has to be accounted for, details of which can be found in the work of Rahman and Al Hasan [RH14].

In the context of limited main memory, streaming algorithms allow processing graphs, which do not fit into main memory. Instead of materializing the graph structure in memory, the edges of the graph are streamed in arbitrary order and processed immediately. Multiple passes over the data, however, are possible. Much of research on triangle counting algorithms in the last years focused on streaming algorithms, which is likely due to flexible memory consumption. Buriol et al. [BFL+06] describe a simple streaming algorithm, which requires three passes over the list of edges. In the

first pass, given a graph $G(V, E)$, the number of edges $|E|$ is counted. In the second pass, an edge $e = (a, b) \in E$ and a vertex $v \in V \{a, b\}$ are sampled uniformly from the graph. In the third pass, the algorithm checks, whether v connects to a and b to form a triangle and returns $\beta = 1$ if that is the case or $\beta = 0$ otherwise. Buriol et al. [BFL+06] also describe a version of their algorithm requiring only one pass over the edge list of the input graph. Jha et al. [JSP13] also developed a single pass algorithm combining ideas of triple sampling with reservoir sampling in streaming scenarios, which is able to outperform the algorithm of Buriol et al. [BFL+06].

The research on triangle counting algorithms produced a wide range of algorithms. However, not all of them can be covered, compared and analyzed in this work. The approximating algorithms used for further comparisons and experiments are therefore chosen to be the ones operating without any access restrictions, i. e. `DOULION`, `ELT`, `ColorfulTriangleSampling`, `TripleSampling`, and `VertexSampling`, in order to cover the most general case.

3.2 Approximate Computing & Approximation Algorithms

Approximating results of computations is not only relevant to triangle counting or graph processing in general. Sparsh Mittal [Mit16] summarizes in his survey paper reasons, applications, and methods for approximate computing. According to him, efforts made in recent years in the field of approximate computing are motivated primarily by the requirements for computational power, which grow faster than available hardware is able to supply in the long run. Therefore, if an exact solution to a problem is not required, a trade-off between quality of results and usage of computational resources is a possibility. In these cases, allowing for small errors may already significantly reduce resource usage for many problems.

Mittal [Mit16] also notes certain pitfalls of analyzing the feasibility of approximating a computation. For example, approximating all parts of a computation equally (uniform approximation) does not usually work. Careful choice of approximable portions of data or parts of a computation is required instead. These choices are application-specific and may not prove useful for other computations. Additionally, finding the right balance between introduced error and efficiency gain requires a quality metric for assessing results. Such a quality metric can furthermore allow for configuration of an approximate algorithm to change quality levels based on current needs.

Some of the concrete methods used in approximate computing according to the survey [Mit16] include the following:

- precision scaling: reducing precision/bit-width of variables
- memoization: saving method results for later executions with same inputs
- load value approximation: prediction of what value a memory read would have returned
- loop perforation: skipping some iterations of a loop
- omitting portions of input data

Some of these techniques are used for example in designing inexact electronic circuits [KGE11] or avoiding performance degradation caused by branch diversion in Single-Instruction-Multiple-Data (SIMD) architectures [SK13].

Similarly, Du et al. [DKH12] describe several strategies to use when developing approximation algorithms in the field of optimization problems. The first one is restriction of the problem by introducing additional constraints, e. g. solving the Steiner Tree Problem, which is NP-hard, by restricting the solution to be a minimum spanning tree, which can efficiently be calculated [DKH12]. Another strategy is using a divide-and-conquer approach by partitioning inputs and building a solution of the complete input by combining solutions of the partitions, which may cause inaccurate results depending on the problem. Relaxation is a strategy contrary to restriction and attempts to make a computation more efficient by removing constraints. When using relaxation, the solution of the relaxed problem may not be a feasible solution to the original problem and must be transformed first. An example for this strategy is relaxing an integer linear program to a regular linear program, which is not restricted to integer values and can be solved more efficiently. The solution of the linear program must then be transformed to only use integer values such that it can be interpreted as a valid but not necessarily optimal solution to the original integer linear program.

3.3 Hash functions for coordination

Computations, which run on several distinct computing nodes, always require some resources to be used for coordination. In the case of graph processing, for example, a partitioning step is always required to split the input graph into several parts, which are distributed to the nodes. Therefore, efforts have been made to make coordination of tasks less resource intensive using hash functions.

Aarag and Jennings [AJ05] propose using network address translation based on a hash function for evenly distributing incoming connections across servers in a network. Connections in their system are forwarded to another server based on a “mark”, which is calculated by applying a hash function to a value derived from the source IP address and TCP port. That mark is then used to identify the server in the network to which the connection should be forwarded to such that load is distributed evenly across the cluster.

Klots et al. [KGB01] patented a method for mapping objects in a distributed system to nodes by applying a hash function to the name of an object. The mapping of such an object to a node is solely determined by the name of the object and therefore does not require further communication to coordinate where that object is saved. Changing the mapping due to changes in the system is done by changing the hash function. Consequently, creating and updating the mapping can be achieved with reduced network usage.

Similarly, Joshi et al. [JTN+14] patented the concept of a distributed object-to-node mapping in distributed systems. In their system, every node maintains a local copy of the system’s state. This state holds information regarding which nodes are online, as well as other parameters influencing the mapping. Nodes are resolved based on applying a hash function to that local state information. In order to achieve consistency, this local state information is exchanged with neighbors from time to time such that every node remains updated and can therefore produce a mapping, which is consistent with its neighbors. The advantage of this “soft-state” approach is that a node’s failure does not cause the whole cluster to be locked until it becomes synchronized through message-based coordination again, which could cause minutes of downtime for large clusters.

All of these methods have in common that some object, e. g. a file to be stored or a packet to be routed, is assigned to some destination, e. g. a certain node. They also rely on the mapping to be random, such that objects are distributed evenly across the available resources. However, these approaches do not consider decisions on whether to accept or drop a given object in the first place, which would be of interest in regards to graph sparsification.

3.4 Datasets

Name	$\|V\|$	$\|E\|$	$\|\Delta\|$	$\|\Delta\|/\|V\|$	$\|\Delta\|/\|E\|$
roadNet-CA [BMSW12]	1,957,027	2,760,388	120,492	0.06	0.04
road-usa [BMSW12]	23,947,347	28,854,312	438,804	0.02	0.02
enron-email-dynamic [Coh]	86,978	297,456	1,180,387	13.57	3.97
mouse-retina-1 [ALB+13]	1,076	90,811	3,289,057	3,056.74	36.22
youtube [MMG+07]	495,957	1,936,748	2,443,886	4.93	1.26
copresence-InVS15 [Soc]	219	16,725	7,130,020	32,557.17	426.31
NotreDame [AJB99]	325,729	1,090,108	8,910,005	27.35	8.17
WormNet-v3 [CSH+14]	16,347	762,822	15,279,134	934.68	20.03
psmigr1 [Sla83]	3,140	410,781	25,298,991	8,057.00	61.59
livejournal [MMG+07]	4,033,137	27,933,062	83,552,703	20.71	2.99
HepTh [LKF05]	22,908	2,444,798	191,358,360	8,353.34	78.27
coauthors-dblp [BMSW12]	540,486	15,245,729	444,095,058	821.66	29.13
uk-2005 [BRSV11]	129,632	11,744,049	837,885,720	6,463.57	71.35
clique	108,716	11,143,165	247,668,082	2,278.12	22.23

Table 3.1: Graphs used in the experiments

The results of the experiments performed in this work should be applicable to as many real-world graphs as possible in order to allow conclusions for general use of triangle count approximations. The graphs used to run those experiments on are therefore selected in such a way that they represent various fields and properties. All the graphs are taken from Network Repository [RA15]. They are preprocessed to filter out duplicate edges, remove self-loops and ensure that for every edge there exists an edge connecting the same two vertices in the opposite direction. As a consequence, vertex, edge and triangle counts may vary from the original source. This preprocessing is done to improve comparability of results with other works in the field.

Table 3.1 contains the list of graphs used for the experiments. The table also contains the amount of vertices, edges, and distinct triangles in the graph after preprocessing. Graphs included in the list are road networks (`roadNet-CA`, `road-usa`), social networks (`youtube`, `livejournal`), web graphs (`NotreDame`, `uk-2005`), biological networks (`mouse-retina-1`, `WormNet-v3`), interaction networks (`enron-email-dynamic`, `HepTh`, `coauthors-dblp`), an infection network (`copresence-InVS15`) and a network of inter-county migration in the US (`psmigr1`).

The last graph in the list, `clique`, is an artificially generated graph consisting only of cliques of sizes one to 200, such that there are at least 500 vertices of each degree. It is intended solely for examining runtime behavior of the various algorithms depending on vertex degrees in a later chapter.

Using cliques of various sizes allows to observe with a fine granularity how the algorithms scale with increasing vertex degree. A lower limit of the number of vertices with a certain degree ensures that enough runtime measurements are available to filter out outliers caused by the operating system or memory management.

Using this diverse set of graphs to analyze runtime behavior of algorithms should allow for transferring insights resulting from the experiments to other graph datasets. The variety of fields the graphs are taken from should result in a balanced representation of real-world graphs.

3.5 Contributions of this work

Various algorithms have been developed for triangle counting. Both exact and approximate algorithms have received attention; efforts were made towards developing fast and accurate algorithms. However, while direct comparisons in terms of runtime and accuracy exist, in-depth analysis on why some algorithms perform better than others are hard to find. Knowing which methods and strategies work better is nevertheless essential, when trying to derive new and potentially better algorithms.

Analysis of algorithm behavior and issues in implementing one of the presented algorithms lead to the first contribution, a concept of decentralized coordination and for reducing communication in distributed computations, such as triangle counting, by using cryptographic hash functions. This concept reduces the need for message-based communication between processes in a distributed computation by allowing pseudorandom decisions, e. g. sampling edges, to be deterministically calculated by every process. As a result, the results of these decisions would not have to be communicated explicitly.

Influence on triangle counting performance, however, is not limited to coordination and communication overhead. The input graph significantly affects runtime and accuracy of approximated results. Since it is hard to gain insights into the way the input graph's structure influences algorithm results, the second contribution of this work is the examination of how a graph's structure correlates with triangle counting results on various graphs. The focus lies on analyzing relationships between structural properties of a graph and its vertices' local triangle counts, which could allow for improving triangle count estimates.

In an effort to better understand the effects of algorithm design choices on the algorithm's runtime behavior, the third contribution of this work is a qualitative analysis of the exact and approximate triangle counting algorithms discussed in this chapter, except for those algorithms based on streaming or random walks. This exception is due to assuming no access restrictions being the more general case and the computational model of Apache Giraph being a better fit for these algorithms. The analysis examines design choices of algorithms and the effects they have on algorithm execution and results. One part examines the runtime behavior of algorithms on the level of subroutines the algorithms consist of. The other part examines the effects an algorithm's design choices have on overall runtime and results of the algorithm.

Based on that analysis, a new approach for computing triangle count estimates is presented. Enabled by insights into the structural properties of graphs, the third contribution is a method of using structural properties of a graph to calculate triangle count estimates from a sample of vertices of a graph.

These contributions are made using Apache Giraph as graph processing framework and results may therefore vary in other scenarios using other frameworks, although many results are generally applicable. Lastly, the contributions of this work are also evaluated, discussed and put into context at the end of this work. Before that, however, the following chapter covers a method for reducing communication in distributed computations.

4 Hash-based Coordination

Communication between single computing nodes in distributed computations is required for exchanging intermediate results, synchronization, joint termination or coordination in general. Communication is, however, always bound to require some overhead and cause latency. Reducing communication could therefore speed up computations the same way an algorithmic optimization could. This chapter investigates the possibilities of using cryptographic hash functions as so called "Random Oracles" to reduce communication in a distributed triangle counting computation.

However, the aim of this chapter is not only to present a method of reducing communication but also show how this method can be used for implementing graph sparsification. The motivation for this stems from the algorithm ELT's requirement to access the original graph when checking whether triples in the sparse graph form triangles in the original graph. As a consequence, edges cannot be actually removed when running ELT but have to be marked instead. The issue with this is that, in contrast to removing edges, Apache Giraph does not allow modifying edges in the input phase of a computation. A different method for distinguishing sampled and dropped edges is required because of that. Therefore, this chapter also describes the use of random oracles with the intention of enabling random decisions of whether an edge should be marked as "dropped", which are consistent across all processes executing the computation, in order to be able to implement the algorithm ELT efficiently using Apache Giraph.

4.1 Random Oracle Model & cryptographic hash functions

In cryptography, the Random Oracle Model (ROM) is a theoretical model describing computations with access to a certain type of oracle, the so called *random oracle*. Given a query x, a random oracle returns an element from its set of output values uniformly at random. The random oracle remembers this output value such that everytime it is presented with the same query x it will return the same output value. Bellare and Rogaway [BR93] formalize a random oracle R to be a mapping from the set of finite binary strings to the set of infinite binary strings, i. e.

$$R : \{0,1\}^* \to \{0,1\}^\infty.$$

The value of $R(x)$ is chosen uniformly at random for every input x. Due to output values being random, two similar input values produce independent and completely different output values.

In reality, such true random oracles do not exist. Instead, cryptographic hash functions are commonly used in implementations as a substitute for a random oracle. A cryptographic hash function $h_n(\cdot)$ is a deterministic function producing pseudorandom, fixed-length output values of length n for any given input bit string. It can be formally defined as:

$$h_n : \{0,1\}^* \to \{0,1\}^n.$$

Similar to a random oracle, it aims to always produce unpredictable output values, which are always the same given the same input.

The key difference between a cryptographic hash function and a true random oracle is that cryptographic hash functions are not truly random but merely pseudorandom due to being deterministic. There has been some controversy in the past regarding security weaknesses in cryptographic signature and encryption schemes caused by substituting random oracles with a cryptographic hash functions, as Koblitz and Menezes [KM15] outline. However, the scope of this work does not include protecting the scheme against maliciously crafted input graphs, which would cause triangle count estimates to be wrong by exploiting weaknesses in cryptographic hash functions. Instead, the focus is on reducing communication overhead by utilizing deterministically computed and pseudorandom hash values without considering malicious manipulation of inputs.

4.2 Cryptographic hash functions & graph sparsification

The triangle counting algorithms discussed in this work use random sampling of parts of graphs as sparsification technique. They rely on unbiased sampling in order to provide accurate estimates. Sampling more or less edges than anticipated would therefore directly cause estimates to be less accurate. Moreover, having edges sampled in a non-uniform manner could cause estimates to be wrong in hardly predictable ways such that correcting estimates becomes complicated. That is why sampling graphs with a good source of randomness is vital for producing good estimates of triangle counts. However, in order to achieve our goal of reducing communication using random oracles we furthermore need the ability to reproduce randomly made decisions on different processes or machines such that decisions are made consistently across those processes and machines. The way in which cryptographic hash functions solve these issues sufficiently and the reasons why other methods do not work as well are covered in this section.

Computers are deterministic machines and can therefore not generate truly random values. There exists a class of dedicated hardware devices, which purpose it is to observe certain physical processes and interpret these observations as a stream of random numbers. For example, Thomas Tkacik [Tka03] developed such a hardware-based random number generator (RNG), which uses the randomly varying periods of two independent ring oscillators to generate random numbers. Some CPU manufacturers include similar hardware-based RNGs into their CPU designs nowadays [Adv17; MI18]. However, while providing good randomness, these solutions cannot be used as a random oracle across multiple machines, since every machine would independently generate different random numbers using such devices.

Using the same sequence of random numbers in multiple processes or machines, would require to transfer those random numbers, causing communication overhead, or deterministically generate them where they are needed. The latter approach can be realized by using a pseudorandom number generator (PRNG). A PRNG starts with a random seed value given as input and deterministically generates a sequence of numbers, which appears to be random, although only its seed value may be truly random. An example of such a PRNG is based on the stream cipher ChaCha20 orginally developed by Daniel J. Bernstein [Ber08] and is used in the Linux-Kernel since version 4.8 [Mue18]. Using the same randomly generated seed value in all processes and machines allows for the same sequence of random numbers to be generated in all of them. That alone is impractical, however, as it would be necessary to coordinate the sampling decisions to be made with the generated random

numbers, i. e. when using a random number to decide whether to drop an edge, the same edge must be processed using the same random number in every process. It would be much less coordination effort, if the randomness used to decide on whether to drop an edge would depend on properties of that edge in addition to a seed value. If that was the case, the decision to drop an edge could be made consistently, be repeated, and reproduced later on.

That is the conceptual advantage of a random oracle compared to a random number generator and the reason why a cryptographic hash function, as a substitute for a random oracle, needs to be used over PRNGs for consistent random decisions across processes without the coordination overhead that would be necessary when using mere PRNGs. A random oracle accepts a query, i. e. the input value, which determines the oracle's random output value. When sampling edges, this random oracle query can be constructed from the description of an edge e by using the vertex IDs of both vertices adjacent to that edge e. In the implementation, the input for the cryptographic hash function substituting the random oracle would additionally include a randomly chosen seed value. This seed allows the calculated hash values to be different for subsequent computations on the same graph but is otherwise just a regular part of the input. It can be distributed to all processes participating in a computation alongside all other inputs to that computation. For the decision on whether to drop an edge or not, a pseudorandom hash value can be calculated for every edge $e = \{u,\ v\} \in E$ as follows:

$$h(\mathsf{seed} \mid\mid u \mid\mid v). \tag{4.1}$$

Here, || denotes the concatenation of the byte representations of the values in this equation. Based on this pseudorandom hash value a decision to drop an edge can be made consistently across process and machine boundaries without additional coordination, since hash functions are computed deterministically.

The only aspect left to examine is the pseudorandomness of cryptographic hash functions, as sampling edges of a graph should happen as randomly as possible. Fortunately, the pseudorandom outputs of cryptographic hash functions are random enough for them to be actually used to construct PRNGs. Implementations of "SHA1PRNG", which is a pseudorandom number generator used in the Java programming language and is based on the SHA1 cryptographic hash function, produce pseudorandom numbers by repeatedly calculating SHA1 hash values using a truly random seed value and an incrementing counter as input [Ora20]. Tests performed by Wang and Nicol [WN15] show that newer cryptographic hash functions of the SHA2 and Keccak (SHA3) families show similar randomness properties for their output values. As a result, cryptographic hash functions can generate pseudorandom values with the same randomness as the alternative, a PRNG, would in many cases provide.

Having established the reasoning behind the choice of cryptographic hash functions for use in the sparsification of graphs, the next step is to actually describe how to use them for this purpose in more detail, which is done in the next section.

4.3 Reducing communication with random oracles

Components in the Bulk Synchronous Parallel (BSP) model communicate using messages. In distributed computations, sending messages via network can cause a significant overhead due to message construction, protocol overhead for reliably sending messages and wait times caused by

Algorithm 4.1 `NodeIterator (BSP)`

Input: $G(V, E)$
Output: Λ_v

```
1:  function COMPUTE(v)
2:      if superstep == 1 then
3:          for all u ∈ N(v) do
4:              for all w ∈ N(v), w ≠ u do
5:                  SENDMESSAGE(u, w)
6:              end for
7:          end for
8:      else if superstep == 2 then
9:          for all m ∈ RECEIVEDMESSAGES do
10:             if DOESEDGEEXIST(m.neighbor) == True then
11:                 T ← T + 1
12:             end if
13:         end for
14:     else
15:         OUTPUT(T)
16:     end if
17: end function
```

network latency. Even when performing a computation on a single machine there is still some overhead caused by message construction and inter process communication, if multiple processes are used for harnessing parallelism on multi-core machines. Reducing communication in such computations can therefore be seen as a way of improving algorithm runtimes in addition to existing optimizations and approximation techniques. The previous section already presented a hint on how this could be achieved using cryptographic hash functions. This section will elaborate on this idea and describe in detail how cryptographic hash functions can be used to reduce communication in triangle counting computations.

The algorithm `DOULION` [TKMF09] will be used as an example to illustrate the idea. `DOULION` samples a subset of edges E' uniformly at random from the original graph $G(V,\ E)$ and then executes an exact triangle counting algorithm, for example `NodeIterator`, on the sampled graph $G(V,\ E')$. Instead of using a local PRNG on a machine to sample the edges, the pseudorandom output of a cryptographic hash function can be used to decide whether an edge should be sampled or not. An encoding of an edge is used as input to the cryptographic hash function in order to base that decision on each edge individually.

Using the algorithm `NodeIterator` as basis for `DOULION`, the first consideration is its implementation. It can be implemented in the BSP model using the "Think-like-a-vertex" approach of Apache Giraph as follows. In the first superstep, every vertex v sends messages to all its neighbors u_i asking whether they share an edge with another vertex u_j neighboring v, which would let those three vertices form a triangle. In the second superstep, every vertex processes those messages by checking whether the requested edge exists and increments its local triangle count by one if it does exist. The pseudocode of this algorithm `NodeIteratorBSP` can be seen in algorithm 4.1.

Having established an implementation reference, sparsification of a graph using cryptographic hash functions works as follows. The sampling of edges of a graph $G(V,\ E)$ is not done using a PRNG but for every edge $e = (u, v) \in E$ a pseudorandom hash value with a length of n bit is calculated using such a cryptographic hash function $h_n(\cdot)$ and an encoding function $c(\cdot)$. The encoding function transforms an edge into a byte sequence, which can then be used as input for $h_n(\cdot)$. Let $c(\cdot)$ be defined as follows:

$$\forall u,\ v \in V : c(u, v) = min(u,\ v)|_b \ ||\ max(u,\ v)|_b, \tag{4.2}$$

where $||$ is the concatenation operator for sequences of bytes, $u|_b$ is a representation of vertex u in bytes, and vertices are ordered by their IDs. The two vertices are concatenated in order of their IDs to guarantee that a pair of directed edges between two vertices u and v is always consistently sampled or not. Otherwise, it could happen that an edge (u, v) would be sampled while the corresponding reverse-edge (v, u) would not be sampled, which would conflict with the input interpretations mentioned in section 2.1.

An edge e is sampled with a sample rate of $p \in (0, 1]$, if the following condition is satisfied:

$$h_n(c(e)) \leq p * (2^n - 1) \tag{4.3}$$

The term $p * (2^n - 1)$ splits the range of the hash function in two parts. Assuming the output values of the cryptographic hash function $h_n(\cdot)$ are uniformly distributed on the interval $[0, 2^n - 1]$, which would be the case for an ideal cryptographic hash function, the value of $h_n(c(e))$ will be in the interval $[0, p * (2^n - 1)]$ with probability p. The condition will therefore sample an edge with probability p, if the pseudorandom hash values are distributed uniformly across the range of the cryptographic hash function. In reality, the hash values of cryptographic hash functions can be distinguished from the uniform distribution [WN15] but, as explained earlier, the regular PRNG implementation of Java – SHA1PRNG – would not be better.

Using the construction described above every vertex can check the existence of the edge in question before sending a request to its neighbor by calculating $h_n(c(e))$. The hash function $h_n(\cdot)$ is deterministic, i. e. given the same input it will always produce the same output. If an edge has not been sampled based on its hash value, every vertex can independently confirm this without the need of communication by calculating the hash value by itself. In that case, the vertex already knows the edge in question does not exist and does not need to send the corresponding request. However, if the hash value determines that an edge between two vertices u and v would have been sampled, the corresponding request has to be sent. This is due to the hash value being unable to determine whether an edge (u, v) actually exists. It can determine only whether an edge between u and v would be sampled or not. In case of the algorithm ELT, this approach allows the algorithm to check whether edges are sampled without having to mark edges in the complete graph on all machines taking part in a computation.

So far, every run of a computation using the same input would always produce the same sampling. When using regular PRNGs, this problem is avoided by using varying seed values. A seed is an additional input to a PRNG from which a PRNG generates its sequence of random numbers. Differing seed values therefore produce completely different sequences of random numbers. Seed values can be used in a similar way with cryptographic hash functions, as even small changes in inputs to a cryptographic hash function generally produce completely different output values. Changing equation (4.3) to accept a seed value, as in equation (4.4), allows a user to add a randomly

chosen seed value as an additional input to a computation, which is to be used in all hash function evaluations.

$$h_n(\texttt{seed}|_b \mid\mid c(e)) \leq p * (2^n - 1) \tag{4.4}$$

The result of this change are completely different hash values and therefore different pseudorandom decisions, e. g. sampling of edges, made in each execution of an algorithm, if differing seed values are used.

Constructions similar to the one described in this section can be employed in other situations, in which pseudorandomness is needed, as well. Using deterministic hash functions allows to coordinate decisions across processes and machines, as long as they are based on inputs, which are available to every process. In the case of triangle counting this is the case for vertex IDs, as every vertex knows all their neighbors' IDs and only direct neighbors can participate in a triangle.

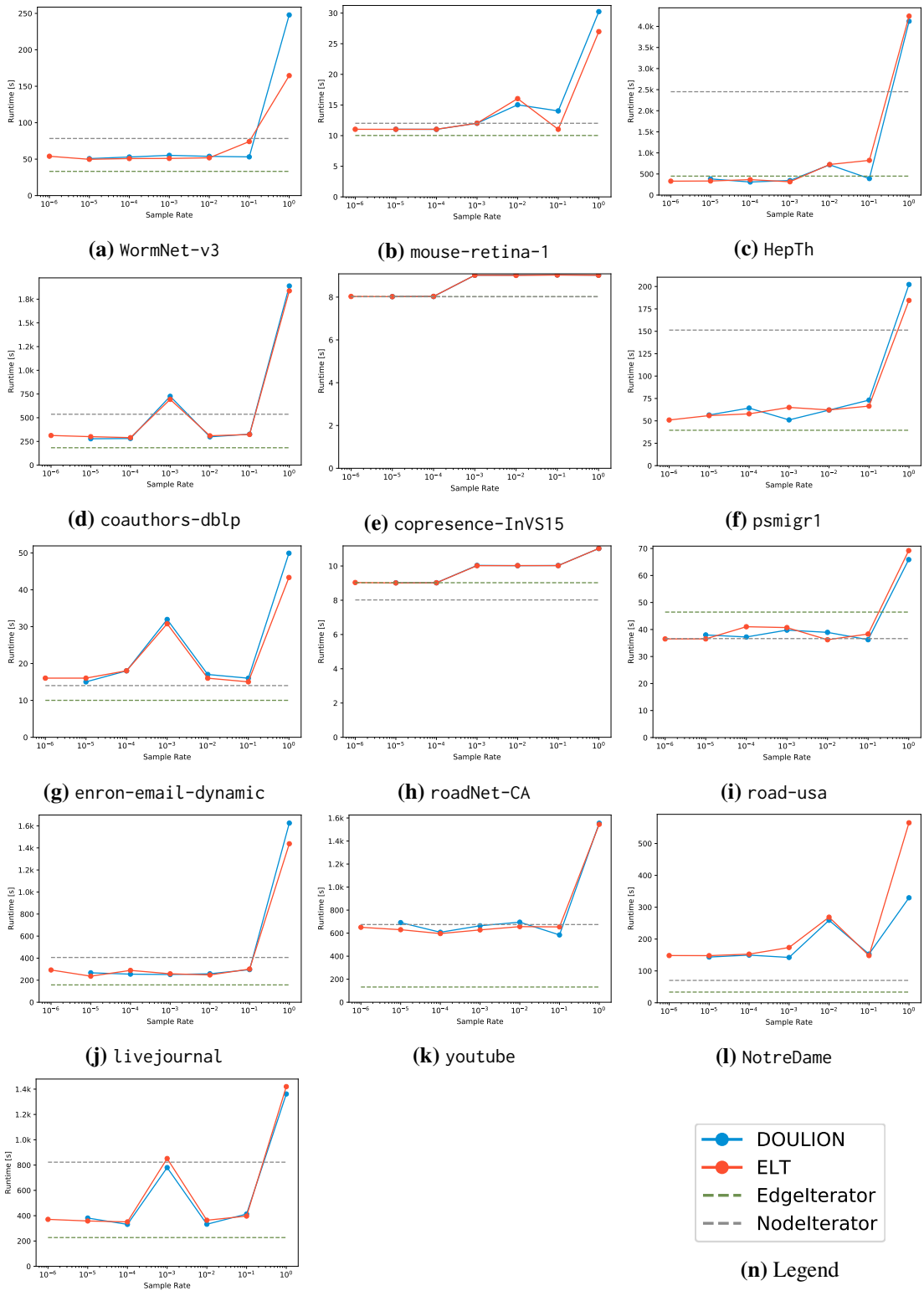
4.4 Effect on performance

In order for this construction to be viable for use in implementations of algorithms, the runtime speedup gained from reducing communication must outweigh the computational overhead caused by having to evaluate a cryptographic hash function. The following section describes a comparison between the regularly implemented algorithms `NodeIterator` and `EdgeIterator` as well as implementations of the algorithms `ELT` and `DOULION` using random oracles based on cryptographic hash functions for sampling edges.

As described in the beginning of this chapter, the algorithm `ELT` is not trivial to implement using Apache Giraph, due to its need to access the original graph after sparsification. Therefore, actual removal of edges is not a possibility under these circumstances. Instead, the process of sampling edges of the algorithm `ELT` is implemented using equation (4.4) with a randomly chosen seed value and `SHA-256` as the chosen cryptographic hash function. If an edge should not be sampled according to this sampling procedure, the algorithm does not remove it, but ignore it in all situations, in which it does not intend to access the complete graph.

The algorithm `DOULION` does not need to access the original graph after sparsification and could therefore be implemented using an input filter, which can simply skip and ignore edges randomly during the input phase of a computation. However, in order to measure the effects of avoiding computations and reducing communication and not the effect of loading a smaller portion of the graph into memory, for the purpose of this experiment, the complete graph is loaded into memory and instead of actually dropping edges they are merely ignored during the computation. Similarly to `ELT`, the sampling process of the algorithm `DOULION` is implemented using the `SHA-256` hash function as source of randomness. This modified version of `DOULION` will be referred to as `RO-DOULION` to avoid confusion with the regular implementation of `DOULION`.

These two implementations of `ELT` and `RO-DOULION` are compared to the standard triangle counting algorithms `NodeIterator` and `EdgeIterator`, which constitute points of reference for the comparison. For the purpose of performance analysis, these four algorithms are executed on the real-world graphs described in section 3.4. The sample rates used for the approximating algorithms `ELT` and `RO-DOULION` range from 10^{-6} to 10^{0} in increments of one order of magnitude. Figure 4.1 shows the resulting runtimes of these algorithms on the 13 real-world graphs.



(a) WormNet-v3 **(b)** mouse-retina-1 **(c)** HepTh

(d) coauthors-dblp **(e)** copresence-InVS15 **(f)** psmigr1

(g) enron-email-dynamic **(h)** roadNet-CA **(i)** road-usa

(j) livejournal **(k)** youtube **(l)** NotreDame

(m) uk-2005 **(n)** Legend

Figure 4.1: Runtimes comparison of triangle counting algorithms implemented using random oracles for graph sparsification.

The results show that both ELT and RO-DOULION perform very similar in terms of runtime. Both algorithms show a significant overhead for sample rates of 1, i. e. when all edges are sampled and the algorithms basically mimic an exact triangle counting algorithm while still performing all computations required for the sampling process. This leads to the assumption that the algorithmic differences of ELT and DOULION are of rather minor importance for this comparison, as both algorithms already work fairly similar. The significant difference between ELT and DOULION is ELT's additional check for the existence of non-sampled, triangle-closing edges in the original graph, which does not appear to significantly increase runtimes of ELT. Furthermore, reducing the sample rate below 0.1 appears to have hardly any effect on runtimes, which may be due to the computational overhead caused by the graph processing framework dominating the overall runtime for small sample rates. The only variations in runtime for small sample rates are sudden spikes, which can be seen for graphs coauthors-dblp, enron-email-dynamic, NotreDame, and uk-2005. These spikes are likely caused by unfortunate sampling of high degree vertices, as they regularly appear in algorithm runs for varying sample rates.

When comparing these random-oracle based implementations of ELT and RO-DOULION with the exact algorithms NodeIterator and EdgeIterator, it can be seen that the former can actually reduce runtimes by avoiding computations based on random oracle decisions. Both ELT and RO-DOULION can outperform NodeIterator, the algorithm on which basis they were implemented due to ease of implementation, on most graphs for sample rates below 1. However, they are both unable to outperform EdgeIterator on almost all graphs, which is likely due to EdgeIterator already being significantly faster in this setting than NodeIterator, such that the runtime saved by sampling edges cannot compensate for using NodeIterator as basis for implementation. Nonetheless, this shows that the method of using a cryptographic hash function for sampling edges can in principle actually speed up computations.

While skipping parts of computations works to make them faster, triangle counting is not the best example to showcase the potential of this method. When a vertex checks whether an edge exists and evaluates the cryptographic hash function to check whether it is sampled, the amount of computations that can be skipped, if it is not sampled, is fairly small. The potential savings in terms of computations for every evaluation of a cryptographic hash function is limited to not sending a message via the non-sampled edge and the vertex, which would have received that message, not having to check whether a requested vertex is its neighbor. The benefits of this method are therefore expected to be more pronounced in computations, in which each decision to skip parts of the computation can cause a larger portion of the whole computation to be skipped than is the case for triangle counting, as this would reduce the relative overhead of evaluating a cryptographic hash function. Tests performed on the server cluster used for all experiments, in which 100 000 000 SHA-256 hash values of integers were calculated and 100 000 000 random numbers were generated, show that evaluating the SHA-256 hash function takes on average approximately 100 times longer than generating a random number. The overhead of evaluating a hash function compared to generating a random number should therefore be considered, especially if the potential savings in terms of computations for every hash function evaluation are small.

In conclusion, cryptographic hash functions can be a valuable and effective tool for implementing pseudorandom decisions which need to be made deterministically based on their input. This method can be used for making triangle counting algorithms faster by using it for graph sparsification, as can be seen in the implementation of RO-DOULION. However, it is advised to prefer using regular pseudorandom number generators over cryptographic hash functions where possible due to the

significant overhead of evaluating such a cryptographic hash function. Because of this method it was possible to implement the algorithm ELT in Apache Giraph without additional messaging overhead for deciding which edges to sample. Having thus a viable implementation of the algorithm ELT fit for comparison with the other algorithms discussed in chapter 3 on related work, the next chapter continues with an analysis of structural properties relevant for triangle counting and further comparisons of triangle counting algorithms.

5 Analysis

The first prerequisite for improving upon the state-of-the-art algorithms is understanding which aspects of triangle counting influence runtime and results of said algorithms. This chapter aims at generating a deeper understanding on how the presented algorithms operate and how structural properties of the input graphs can influence results. In order to achieve that, the experiments of this chapter are designed to measure runtimes of algorithms and their subroutines as well as statistical relationships between structures of the input graph and local triangle counts.

This chapter is divided into two parts. The first part describes the experiment setup with which the behavior and results of algorithms is analyzed. The second part reports on findings and results gained from these experiments.

5.1 Context and Goal

In regard to the experiment setup, this section will describe the existing constraints, aims, and the configuration used to produce the results described in the next section as well as cover the reasoning behind choices.

Starting with the constraints of triangle counting, the sole requirement for a triangle to exist is to have three distinct vertices being fully connected to each other with edges. Therefore, only vertices in the direct neighborhood $N(v)$ of a vertex v can influence the local triangle count of that vertex v. A vertex $u \notin N(v)$ can therefore be ignored for the purpose of finding triangles in which the vertex v is part of. As a result, only properties of a vertex v itself and its direct neighbors can influence the local triangle count of v. As long as no further application-specific knowledge about the structure of the graph is assumed, e. g. values associated with vertices or edges, such properties can only be concerned with the existence of edges. This is due to the existence of a triangle being determined purely by structural properties of the graph it is contained in, i. e. which vertices are connected by edges. The only remaining measure able to possibly indicate the amount of triangles at a vertex is therefore its degree, i. e. the amount of adjacent edges.

Including a vertex's degree in the input graph is implicitly done, if an input format representing an adjacency list is used, in which each vertex is directly followed by a list of its neighbors. A vertex's degree could then be derived from the number of its neighbors. However, the majority of graph data available on the Stanford Network Analysis Project (SNAP) [LK14] and Network Repository [RA15] is provided in formats based on edge lists. Therefore, the input format for the following experiments is assumed to be a list of edges and information about vertices' degrees is only available after the graph is materialized in memory. In order to minimize assumptions, a simple input procedure, which can only construct the graph structure in memory and sample edges individually based on the IDs of the adjacent vertices, is assumed to be used. A vertex at the beginning of a an Apache Giraph computation would therefore only know its own degree and its

neighbor's IDs but would have no further knowledge about the neighbors themselves. If accessing properties of a neighboring vertex, such as their degree, would be required at the beginning of a computation, it could cause a significant overhead in generating the graph structure, as degree values of all vertices would have to be communicated to their neighbors via messages.

Most triangle counting algorithms presented in section 3.1 actually rely on exploiting solely structural properties of a graph for performance improvements. The algorithm `Forward` (3.3), for example, reduces the degree of high-degree vertices in order to lessen the disproportional runtime cost of vertices with high degree. In order for the algorithm `Forward` to be more efficient than `EdgeIterator`, however, the vertices' degrees need to differ from the average vertex degree in the graph to some extent [Sch07]. Another example for the described constraints being considered are the probabilistic estimations of the algorithm `DOULION` [TKMF09]. They acknowledge that only direct neighbors of a vertex v can form a triangle with it by considering no structures larger than a single triple of vertices.

The aim of the following experiments is to improve understanding of the problem of triangle counting. In order to achieve this, the topic of statistical relationships between vertices' degrees and the average amount of triangles vertices are a part of is examined. Furthermore, the way in which algorithms utilize different design choices in order to improve performance is examined, as well. Resulting from the considerations above, the specific questions at hand are therefore:

- How does the degree distribution of a graph correlate with local triangle count distribution?
- How do degrees in a pair of vertices influence the share of neighbors both vertices have in common?
- How is runtime usage divided up among the algorithms' subroutines?
- What are the key elements for performance improvements in state-of-the-art algorithms?

These questions will be addressed by the following experimental setups.

5.2 Experimental setup

Having established the questions to be researched, this section will explain the experimental setup used to answer these questions. The algorithms used for these experiments are described in section 3.1. All algorithms were implemented using Apache Giraph as graph processing framework. Reimplementing those algorithm in the same framework is expected to enable better comparisons between algorithms by means of having a common baseline for all algorithms. Execution of all experiments was performed on a cluster of five servers with two AMD Epyc 7401 CPUs each and 1.1 TB of main memory in total.

5.2.1 Correlation of a vertex's degree & triangle count

The first step of identifying structural properties of a graph which can be used to improve existing triangle counting algorithms is to understand how differences in input affect the output. As explained in section 5.1, only structural properties of a graph, i. e. which pairs of vertices are connected by edges, can have an effect on the triangle count. A vertex's degree is defined as the number of edges

incident to that vertex and thus the number of vertices it is connected to. It is the simplest structural measure in this context, cheap to compute and easily available in Apache Giraph's computational model. This first experiment therefore measures the statistical relationship between vertices' degrees and the local triangle counts of those vertices. If no such relationship would exist, simple heuristics considering only a single vertices' degrees could not be used to improve results of triangle counting algorithms, as that measure could not indicate anything in regard to distributions and amount of triangles.

The existence of a relationship between the degrees of vertices and their local triangle counts is examined on a variety of real-world graphs described in section 3.4. The relationships are evaluated using the three different correlation measures described in section 2.4. They are used to detect different kinds of statistical relationships: Pearson's correlation coefficient [MBL+12] for linear relationships, Spearman's rank correlation coefficient [DE12] for monotonic, (non-)linear relationships and distance correlation [SRB07] used for linear and non-linear relationships.

5.2.2 Correlation of a vertex pair's degrees & triangle count

The next step of finding properties of a graph which can be used to improve existing triangle counting algorithms is to examine the degrees of two neighboring vertices. Having the degrees of two neighboring vertices, estimating the number of common neighbors to form triangles with based on those degree values would allow for a cheap approximation of triangle counts. Therefore, the aim of this experiment is to search for a statistical relationship between the degrees of two vertices and the share of their neighbors they have in common. In order to analyze such a correlation, the two degree values of a pair of vertices must be transformed into a single value first.

For this purpose, the *degree product* is defined with the following considerations in mind. The degree product is intended to model the notion of a mutual degree of two neighboring vertices. Due to the interest in the share of common neighbors of two vertices, the individual vertex's degrees are multiplied, as this reflects the multiplicative increase in potential common neighbors with increasing degrees of the individual vertices. Dividing each degree value by the amount of vertices of the graph is done to normalize the value for comparisons across graphs. The resulting equation for the degree product $dp(u, v)$ is the following:

$$\forall u, v \in V : dp(u, v) = \frac{d(u)}{|V|} * \frac{d(v)}{|V|} \tag{5.1}$$

A strong statistical relationship between the degree product of two vertices and the amount of neighbors those two vertices have in common would imply that degree products could possibly be used to estimate triangle counts.

The degree product measure could, of course, be generalized to combine the degree values of three vertices, as well. However, communicating those additional degree values and estimating triangle counts based on them would most likely be not much faster than directly counting the triangles themselves. Because of this, this experiment only considers pairs of vertices but not triples.

5.2.3 Comparison of algorithms' runtime breakdowns

An important element of understanding algorithm behavior is understanding how much different parts of the algorithm contribute to the overall runtime of the algorithm and how that changes with differing inputs. All vertices in Apache Giraph's "Think-like-a-vertex" model perform the same computations and significant differences should therefore be caused primarily by the degrees of vertices. In order to receive a more detailed overview over which parts of a triangle counting algorithm need the most time to compute or which portions of a graph require the most computations, a fine-grained approach is required.

This is achieved by implementing the algorithms described in section 3.1 in a modular way encapsulating different tasks during the computation, e. g. message creation or counting triangles from messages, into separate subroutines. The vertex-centered "Think-like-a-vertex" model allows to measure execution times of subroutines on a per vertex basis and therefore allows measuring runtime as a function of a vertex's degree. In order to measure these runtime for various degree values, this experiment runs on an artificial graph consisting of cliques of sizes one to 200. Reliability of those measurements is improved by having at least 500 vertices of each degree value such that outliers, which are caused by system processes or memory management, can be filtered out. These limits were chosen such that the server cluster used for these experiments would not distort results by spending too much time on memory management, in case the remaining free memory would be too little for efficient memory allocation. Measuring these subroutine runtimes on such an artificial graph provides us with insights on which parts of algorithms are the most expensive computation-wise and how that is influenced by vertices' degrees.

5.2.4 Effect of algorithm's design choices on overall runtime & results

When using triangle counting algorithms, probably the most important characteristics are the total runtime and the accuracy of the results. The different design choices of existing algorithms can provide further insights into the problem of counting triangles by revealing which approaches work best for reducing runtime or increasing accuracy. In this experiment, the performance of different algorithms and their optimizations or approximations is evaluated and compared.

It is analyzed how much overhead operations used for approximations impose and how those approximations reduce overall runtime. An example of such an overhead would be the generation of random numbers in order to randomly drop an edge. These differences are compared to exact triangle counting algorithms, upon which these approximations try to improve. This qualitative analysis of existing triangle counting algorithms is aimed at revealing insights into design choices of algorithms and the way in which structural properties of a graph can be used for improvement of algorithms.

5.2.5 Notes on algorithm implementations

The implementation of an algorithm strongly influences its performance. In order to allow for fair comparisons between the algorithms, an efficient implementation is very important. However, the combination of algorithm's design choices and abilities of tools and frameworks do cause some restrictions, which prevent ideal implementations. Due to this, there also have to be some remarks regarding the way the algorithms were implemented for these experiments.

Apache Giraph allows to define filter classes, which decide for each edge or vertex when loading the input graph whether they should be ignored or kept for the computation. Sparsification methods, which do not require structural information, such as a vertex's degree, can be implemented by implementing such a filter class and use any exact triangle counting algorithm, e. g. `NodeIterator` or `EdgeIterator`. This way of implementation using a filter class is used for `DOULION` and `ColorfulTriangleSampling`. The input filter used for implementing `DOULION` samples edges randomly given a sample rate. Since filtering for `ColorfulTriangleSampling` is aimed at filtering edges but coloring is done on vertices, the input filter has to ensure that vertices can be colored consistently when only given edges. The resulting implementation uses the vertex's ID modulo N, the number of colors to be used, as vertex color. The vertex IDs have to be permutated randomly beforehand in order for this to work properly. However, this permutation step is not considered further in the experiments, as it was considered not to be an issue of the algorithm rather than an issue of the context of implementation.

An approach using input filters is not possible for the algorithm `Forward`, as it requires the degrees of both adjacent vertices of an edge to decide whether to drop or sample that edge. `TripleSampling` also cannot be implemented on the basis of dropping single vertices or edges, i. e. the whole graph needs to be present in memory. Instead, every vertex v in the `TripleSampling` Giraph computation samples only a subset of its pairs of neighbors, with which said vertex v forms a triple, to count its adjacent triangles. `VertexSampling` cannot be implemented using filters, because it requires all vertices in the graph to be able to respond to messages and thus be present. The sampling step in this algorithm is realized by a portion of vertices executing the regular `NodeIterator` algorithm and the rest of the vertices going inactive immediately, only reacting to incoming messages.

Similarly as already mention in chapter 4, the algorithm `ELT` requires the whole graph to be present in memory, as well, in order to check whether open triples in the sparse graph are triangles in the original graph. The same implementation of `ELT`, which is based on the use of a cryptographic hash function for sampling edges as described in chapter 4, is used for these experiments. However, the use of a cryptographic hash function for sampling edges adds a significant overhead. For this reason, comparisons with `ELT` in terms of runtime use the algorithm `RO-DOULION` from chapter 4 as reference, which is an implementation of `DOULION` similarly using cryptographic hash functions for sampling edges.

Graph	Pearson's r	Spearman's r	Distance correlation
copresence-InVS15	0.59	0.76	0.69
mouse-retina-1	0.69	0.71	0.66
enron-email-dynamic	−0.14	−0.41	0.40
psmigr1	0.80	0.80	0.78
roadNet-CA	−0.38	−0.41	0.51
road-usa	−0.34	−0.35	0.51
WormNet-v3	0.28	0.42	0.39
youtube	0.40	−0.43	0.42
coauthors-dblp	−0.64	−0.82	0.81
NotreDame	−0.01	−0.33	0.17
livejournal	−0.73	−0.92	0.86
HepTh	0.57	0.44	0.44
uk-2005	0.08	0.46	0.30

Table 5.1: Correlation between vertex degree and local triangle count at that vertex.

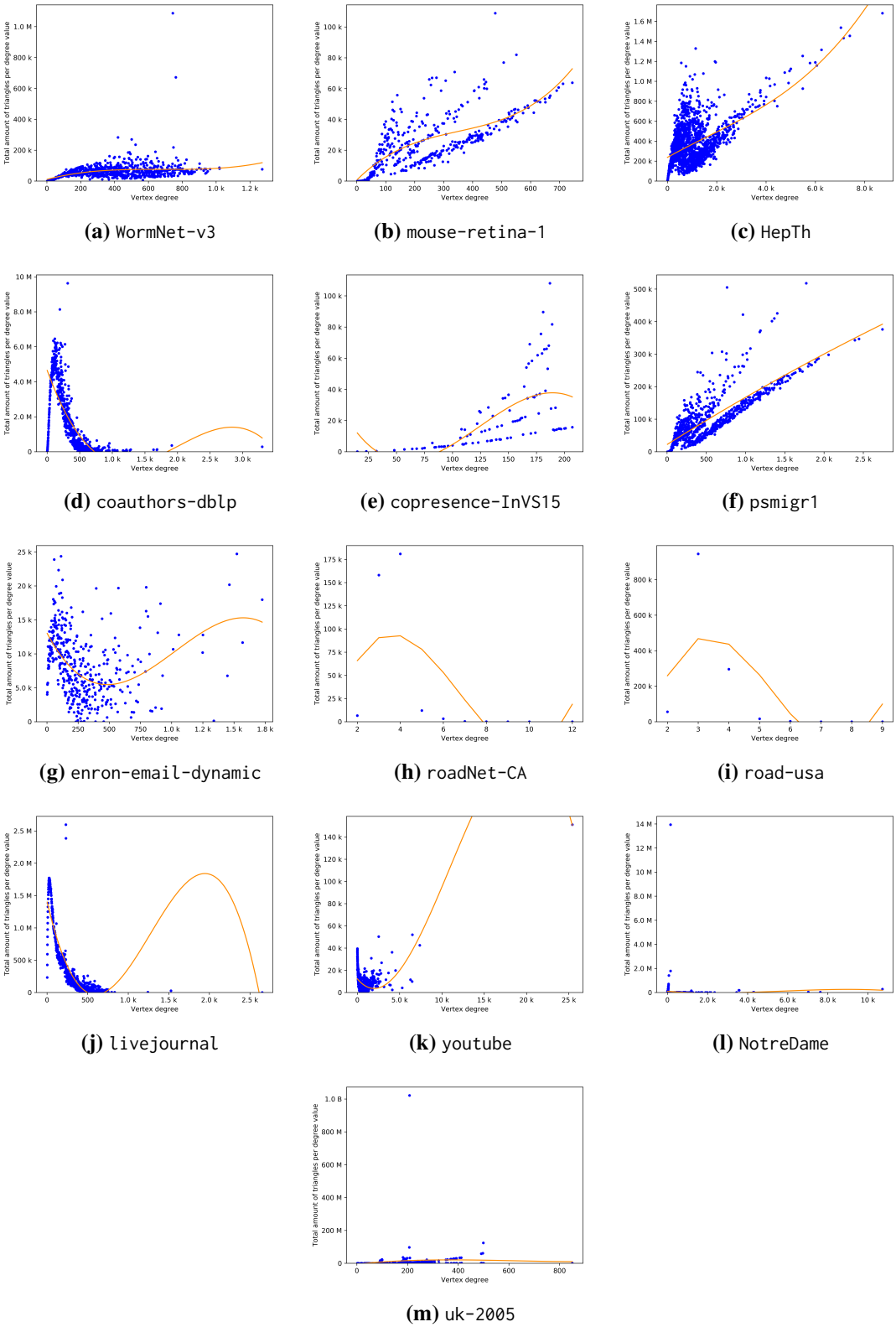
5.3 Findings

Having presented the approaches of the analysis as well as implementation details of the algorithms, which are examined, the following sections present the results of this analysis. Each section focuses on one of the presented experiments. At the end of the chapter, a conclusion is drawn from the analysis' results, based on which a new approach for calculating triangle count estimates is derived in the following chapter.

5.3.1 Correlation of a vertex' degree & triangle count

Beginning with the simplest of the four examinations, the results for correlations between a vertex's degree value and the total amount of triangles containing at least one vertex having that degree vary quite a bit. Table 5.1 contains all correlation measures for those correlations for all examined graphs. Although there are a few graphs for which a linear relationship between degree and triangle count exist – indicated by Pearson's $|r| > 0.6$ – there are also graphs with weak correlations. A wide range of correlation values from almost 0 to 0.8 can be observed, which suggests that other properties of a graph heavily influence the distribution of triangles.

The reason for this variance in strength of the correlations and some limitations of mere correlation values may be seen in Figure 5.1. This figure contains plots of vertex degrees on the x-axis and the total amount of triangles in a graph containing at least one vertex with that degree value on the y-axis. Also included in the figures are regression polynomials (orange) to illustrate how well the data could be modeled by a simple polynomial of third degree. However, many of the measured values deviate from the regression polynomial quite a bit such that most of the regression polynomials do not appear to be fitting well. Such plots were made for all real-world graphs described in section 3.4. It is obvious that every graph has a different distribution of triangles across vertex degrees. While a correlation value can only indicate the strength of some type of relationship, these graphs give



(a) WormNet-v3 **(b)** mouse-retina-1 **(c)** HepTh

(d) coauthors-dblp **(e)** copresence-InVS15 **(f)** psmigr1

(g) enron-email-dynamic **(h)** roadNet-CA **(i)** road-usa

(j) livejournal **(k)** youtube **(l)** NotreDame

(m) uk-2005

Figure 5.1: Plots of summed triangle counts (y-axis) of vertices with given degree (x-axis).

Graph	Pearson's r	Spearman's r	Distance correlation
copresence-InVS15	0.98	1.00	0.99
mouse-retina-1	0.99	1.00	0.99
enron-email-dynamic	0.78	0.79	0.76
psmigr1	0.99	1.00	0.99
roadNet-CA	0.94	0.97	0.94
road-usa	0.87	1	0.92
WormNet-v3	0.92	0.98	0.94
youtube	0.88	0.86	0.79
coauthors-dblp	0.90	0.94	0.92
NotreDame	0.74	0.41	0.67
livejournal	0.44	0.64	0.59
HepTh	0.98	0.99	0.97
uk-2005	0.83	0.94	0.94

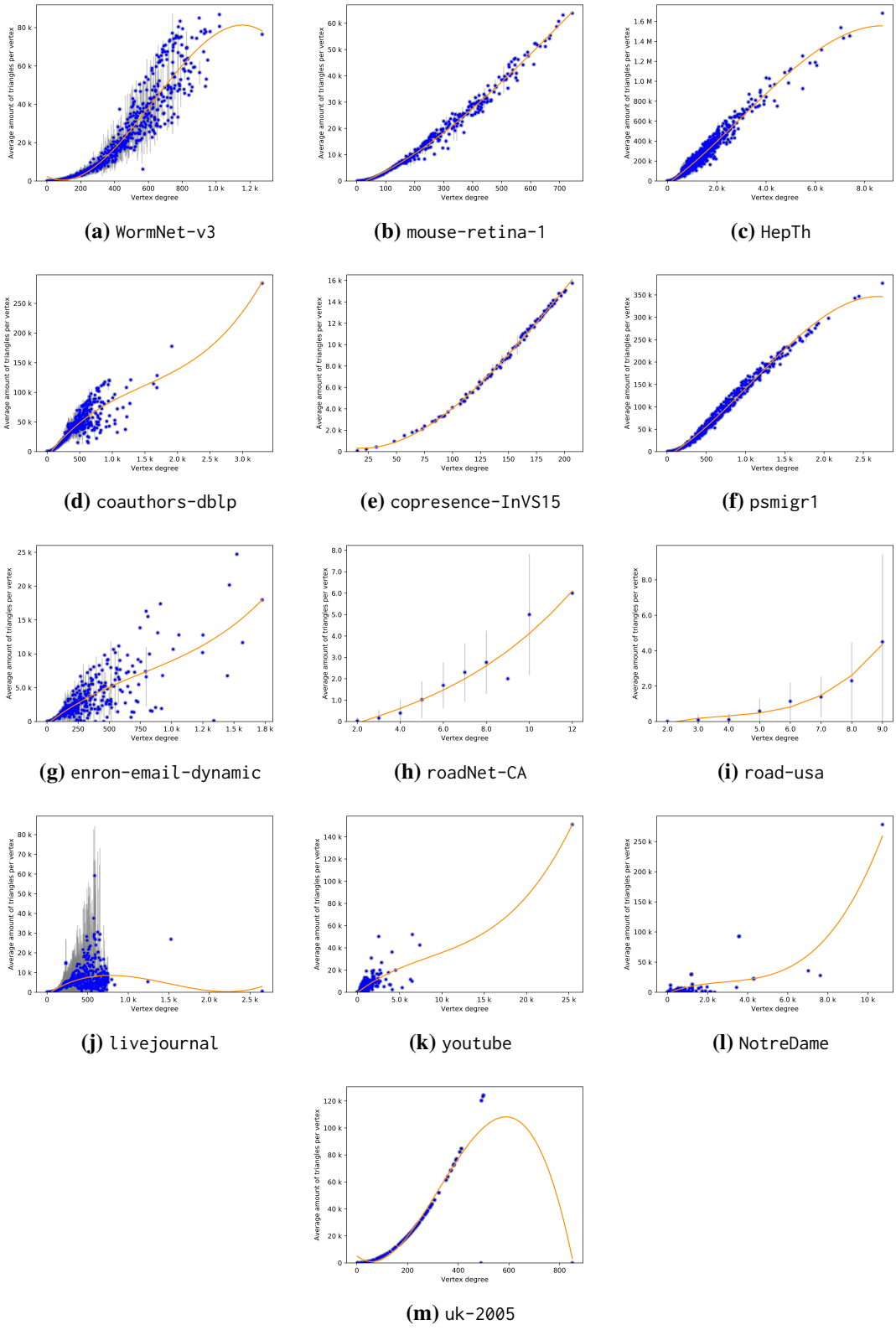
Table 5.2: Correlation between vertex degree and average local triangle count at that vertex.

a more detailed insight into the shapes of those distributions. For example, the graphs `psmigr1` and `livejournal` have comparatively strong correlations, but significantly different distributions of triangles in the graph.

In this set of graphs, two shapes of distributions stand out especially. The first one, a "bump"-like shape, has a steep increase and peak at low degree values and descends quickly after that peak. This shape suggests that vertices with comparatively low degrees are well connected with each other, while vertices with high degree have many neighbors which are not well connected with each other. Graphs with this distribution include `coauthors-dblp`, `livejournal`, `enron-email-dynamic`, and to some degree `youtube`. The former two graphs appear to to descend up to a degree value of ≈ 500 with no vertices with high degree and high local triangle count. The latter two graphs descend only up to a degree value of ≈ 250 and contain some vertices with high degree value as well as high local triangle count.

The other shape of triangle count distribution consists of multiple elongated and narrow linear clusters. Graphs with this shape include `mouse-retina-1`, `HepTh`, `copresence-InVS15`, and `psmigr1`. The individual clusters do not necessarily have the same density along their length, but they do appear to follow a straight line. Due to multiple vertices having the same degree on the one hand and local triangle counts of vertices with the same degree being summed in this plot on the other hand, these clusters may be caused by groups of vertices having similar average local triangle counts, but consisting of different amounts of vertices with a certain degree. This would imply that not the relationship between vertex degree and absolute local triangle count but between vertex degree and average local triangle count of vertices with a certain degree could be a single linear cluster.

In fact, that is actually the case for these graphs with multiple linear clusters, as can be seen in Figure 5.2. These graphs show a fairly linear relationship between vertex degree and average local triangle count for each degree value. For low degree vertices, however, this relationship appears to be rather polynomial, which is the reason that cubic regression polynomials were chosen to be used in figures 5.1 and 5.2. These regression polynomials fit the data of average triangle counts closely



(a) WormNet-v3 **(b)** mouse-retina-1 **(c)** HepTh

(d) coauthors-dblp **(e)** copresence-InVS15 **(f)** psmigr1

(g) enron-email-dynamic **(h)** roadNet-CA **(i)** road-usa

(j) livejournal **(k)** youtube **(l)** NotreDame

(m) uk-2005

Figure 5.2: Plots of average local triangle counts (y-axis) of vertices with given degree (x-axis).

for these graphs. The standard deviations for the average triangle counts per vertex degree value – depicted as gray error bars for each data point – are small as well and indicate that the observed relationship is followed rather closely. In contrast to this, graphs with a "bump"-like distribution of summed up local triangle counts per vertex degree value, as shown in figure 5.1, show a "cone"-like distribution for average triangle counts per vertex degree value in figure 5.2. This "cone"-like shape also causes the standard deviations of the average triangle counts to be much larger.

For those graphs, which do show a strong correlation between vertex degree and average triangle count at a vertex, it should be possible to use this relationship for producing triangle count estimates for a complete graph. One could try to produce such a triangle count estimate by sampling a few vertices and extrapolating the average local triangle count distribution of that sample for the complete graph. A naïve estimate could then be produced by multiplying the frequency of each degree value with the estimated average number of triangles for vertices with that degree.

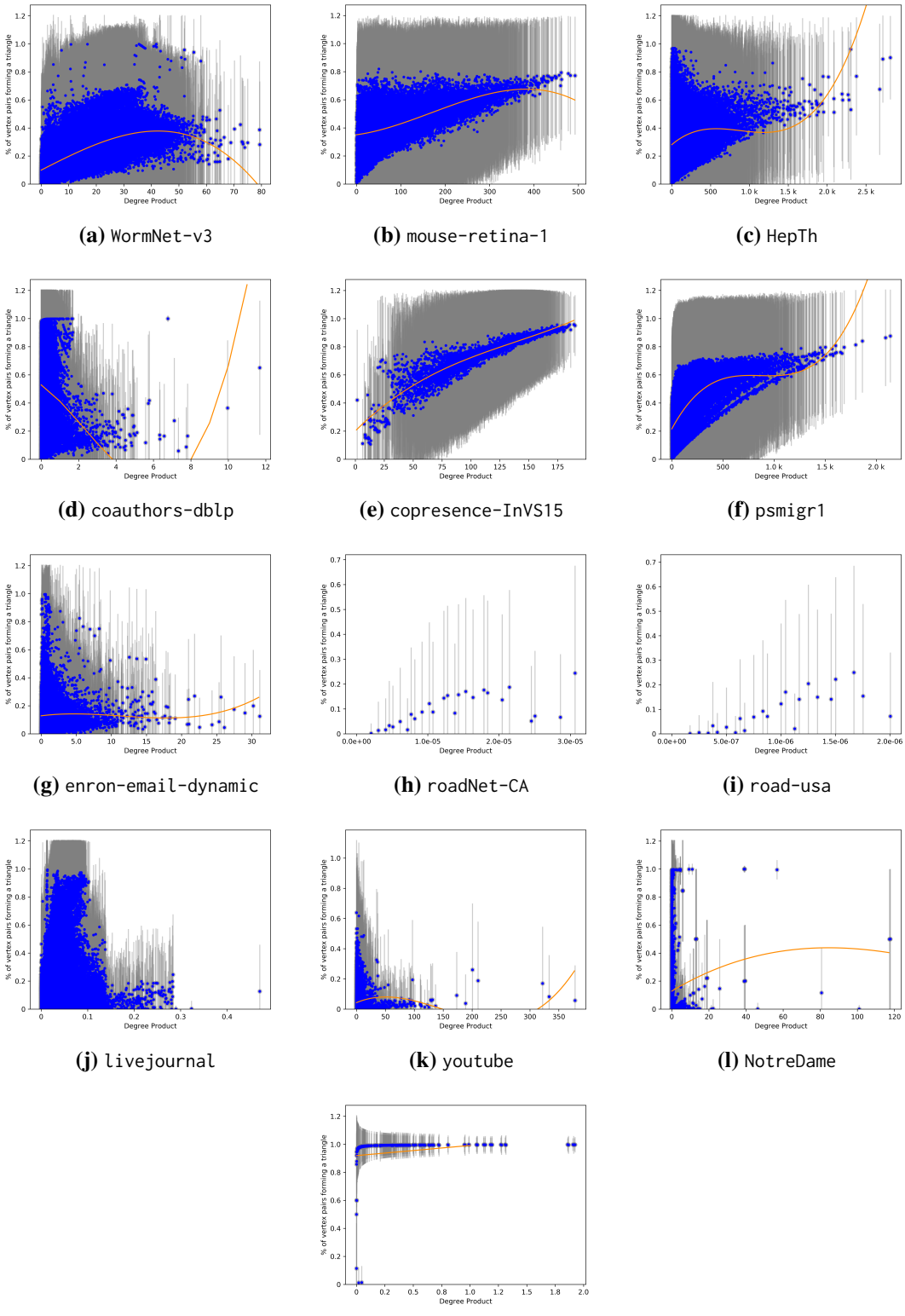
However, this approach has some uncertain aspects. Firstly, the graphs observed in this work are only a small sample of all the real-world graphs used in practice. Graphs, in general, do not necessarily need to follow this pattern. Therefore, in practice it would need to be checked first, whether a graph has this property of having a strong correlation between vertex degree and average local triangle count. Depending on the application, this distribution could theoretically be arbitrary and would be only limited by every vertex with degree d being able to have at most $\binom{d}{2}$ adjacent triangles. Due to this uncertainty regarding the average triangle count distribution, the second issue with this approach would be the theoretical basis for such an estimation in order to provide error boundaries. As a first step of understanding this method of estimating a triangle count, chapter 6 investigates the applicability and feasibility of this method for producing triangle count estimates by utilizing the correlation between vertices' degrees and their average local triangle count on graphs, in which this correlation is very strong.

In conclusion, observing the distributions of total and average local triangle counts of vertices in regards to their degree lead to the discovery of a number of emerging patterns. While regularities between groups of graphs exist, it is yet unclear to which extent they can be used for providing more accurate triangle count estimates for all graphs. Sampling vertices to model a distribution of triangles and generating a triangle count estimate from this triangle distribution may provide accurate results for all graphs, but this approach has to be examined further first.

5.3.2 Correlation of a vertex pair's degrees & triangle count

Considering the degree of two vertices is the next step in analyzing the influence of structural properties on the triangle count of a graph. While the degrees of a single vertex appear to be able to indicate correlations with average local triangle counts for certain groups of graphs, the product of two vertices' degrees could provide better results by utilizing additional information. The underlying assumption being that, similar to the case of observing only a single vertex's degree, additionally taking into account the degree of a neighboring vertex could lead to more accurate predictions of the connectedness of vertices among themselves, which influences the local triangle count of a vertex.

The actual results, however, point into another direction. Figure 5.3 shows for all pairs of neighboring vertices in a graph the share of neighbors they have in common across degree product values. The increased amount of data points compared to figures of the previous experiment stems from



(a) WormNet-v3 **(b)** mouse-retina-1 **(c)** HepTh

(d) coauthors-dblp **(e)** copresence-InVS15 **(f)** psmigr1

(g) enron-email-dynamic **(h)** roadNet-CA **(i)** road-usa

(j) livejournal **(k)** youtube **(l)** NotreDame

(m) uk-2005

Figure 5.3: Percentage of vertex pairs forming a triangle (y-axis) by degree product (x-axis).

there being many more vertex-neighbor pairs than single vertices in a graph. Additionally, the resulting combinations of vertex degrees allow many distinct degree product values. These plots show widely different shapes with far less similarities between graphs compared to plots of the previous experiment.

Ideal for predictions would be a perfectly straight line, which would indicate a linear relationship. However, most plots show a wide corridor of values, especially for small degree products, and accordingly large standard deviations. Almost the complete range of values is covered for these small degree products. As a result, predicting the amount of triangles formed by two neighboring vertices solely based on their degrees is infeasible due to there hardly being a relationship between the degree product and the share of common neighbors for most graphs. For example, when a degree product of a pair of vertices in the graph `livejournal` of 0.0 to 0.1 is encountered – which is the case for most vertex pairs in that graph – the share of common neighbors for a pair of neighboring vertices ranges from 0% to over 90%.

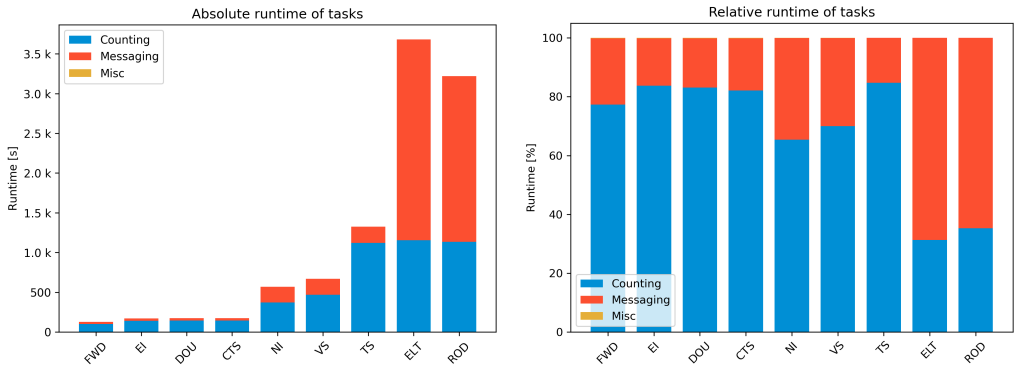
The analysis of the degree product of two neighboring vertices with the intend of finding a better structural measurement than a single vertex's degree for estimating triangle counts has proven not successful. The degree product of two neighboring vertices could not give a clear indication on how many triangles such a pair of vertices would form together with their common neighbors. While the degree of a single vertex could to some extent predict the average local triangle count in certain graphs, this appears to be not possible by utilizing degree products, as the the variances in results for similar degree products are too big.

5.3.3 Comparison of algorithms' runtime breakdowns

The focus of the results in this section are differences between algorithms on a rather detailed level. Observations are made to describe to what extend messaging and counting of triangles influence the overall runtime of algorithms as well as the amount of overhead introduced by sparsification techniques. This finer grained examination allows to better analyze how the design choices of the various algorithms influence their performance.

Sample rates for all algorithm runs were kept at 100% in order to be able to analyze the overhead introduced by algorithms compared to the baseline algorithms of `NodeIterator` or `EdgeIterator`. Such overhead is caused by computations required for decisions of whether to consider an edge or vertex in the computation. Lower sample rates would offset these overheads by omitting calculations through graph sparsification so that the overhead could not be compared.

The first observation made from runtime measurements of algorithm runs on the `clique` graph is that runtimes of all computations performed for a vertex scale quadratically with the degree of that vertex. This is not surprising considering that for finding triangles every vertex checks all distinct combinations of two of its neighbors, the count of which increases quadratically with the number of neighbors of a vertex. Both messaging and counting tasks in these algorithms scale quadratically, as well. However, it is not possible to make statements on whether the shares of algorithm runtime of these two tasks change for certain ranges of degree values or stay constant. Deviations in the measurement results do not allow reliable statements in this regard.



(a) Absolute runtimes

(b) Percentages of runtime

Figure 5.4: Shares of algorithms' runtime being used for messaging, counting and other tasks (IO)

Comparing the baseline algorithms `NodeIterator` and `EdgeIterator` reveals, however, that the latter is already three times faster. This difference comes mostly from vertices in `EdgeIterator` being able to broadcast their lists of neighbors while vertices in `NodeIterator` create messages regarding pairs of neighbors forming a triangle individually. The former approach allows reusing messages, while the latter requires creating individual messages for each pair of neighbors, which in turn causes more overhead for message creation and especially processing.

Improving upon `EdgeIterator`, the algorithm `Forward` further reduces messaging overhead by filtering edges, which are redundant to counting a triangle, and reducing vertex degrees, especially for vertices with a high degree [Sch07]. Although the algorithm `Forward` has the additional overhead of filtering out edges so that messages are only sent along "forward" edges, overall runtime of the algorithm `Forward` is lower than the runtime of the algorithm `EdgeIterator`. It was measured that this filtering accounted for roughly 30% of the algorithm's runtime. Note that in contrast to other sparsification algorithms examined in this work, the sparsification done in the algorithm `Forward` is not an approximation and does not reduce accuracy of results.

Figure 5.4 shows runtime measurements of the algorithms `Forward` (FWD), `EdgeIterator` (EI), `DOULION` (DOU), `ColorfulTriangleSampling` (CTS), `NodeIterator` (NI), `VertexSampling` (VS), `TripleSampling` (TS), `ELT`, and `RO-DOULION` (ROD) being run on the previously described `clique` graph containing only cliques of sizes one to 200. Figure 5.4a shows the total time it took each algorithm to process the graph and Figure 5.4b shows the percentage of runtime it took each algorithm to perform the necessary messaging and counting of triangles.

It can be seen from Figure 5.4a that algorithms `Forward`, `DOULION`, `ColorfulTriangleSampling`, which were implemented using the algorithm `EdgeIterator` as baseline, and `EdgeIterator` itself are the fastest algorithms overall. The filter-based implementation of `DOULION` and `ColorfulTriangleSampling` causes only little overhead. They also spend a comparatively low percentage of their runtime on messaging, only around 20%, as can be seen in Figure 5.4b.

The remaining algorithms, which were implemented using the algorithm `NodeIterator` as baseline, take significantly more time for both counting triangles and sending messages. The algorithm `VertexSampling` has the lowest overhead, as it only has to randomly decide for each vertex, whether it should start the computation active, i. e. actively requesting whether an edge exists between pairs

of neighbors, or inactive, i. e. only reacting to incoming requests. The overhead for the algorithm TripleSampling is significantly higher, which is possibly due to the synchronization overhead of having to collect the amount of open triples and closed triples at a single server to calculate the final triangle count estimate. While the algorithm ELT shows the largest overall runtime in this experiment, this is mainly due to the way in which it had to be implemented and may differ when using other frameworks. As the comparison with RO-DOULION shows, both algorithms have a similar amount of their runtime dedicated to counting triangles and therefore an efficient implementation of ELT could possibly compete with a regular implementation of DOULION. The main reason for the high runtime measurements of ELT is the use of cryptographic hash functions for deciding which edges to sample and therefore which neighboring vertices to consider for sending messages to. This can be seen in Figure 5.4b, as the proportion of runtime spent on messaging routines, in which the decision to consider or ignore an edge is made, is especially large for ELT with more than 60%.

Common to all algorithms in this experiment is that miscellaneous tasks, which in the case of these algorithms are mostly IO tasks for producing the output of each vertex, are negligible. Most of the runtime of the algorithms – except for ELT and RO-DOULION – is spent on counting triangles, not on messaging. For most algorithms only 20% to 35% of the runtime are used for messaging-related routines. The dominating part is used for actually counting triangles, which mostly consists of checking the existence of edges, which are required for forming a triangle.

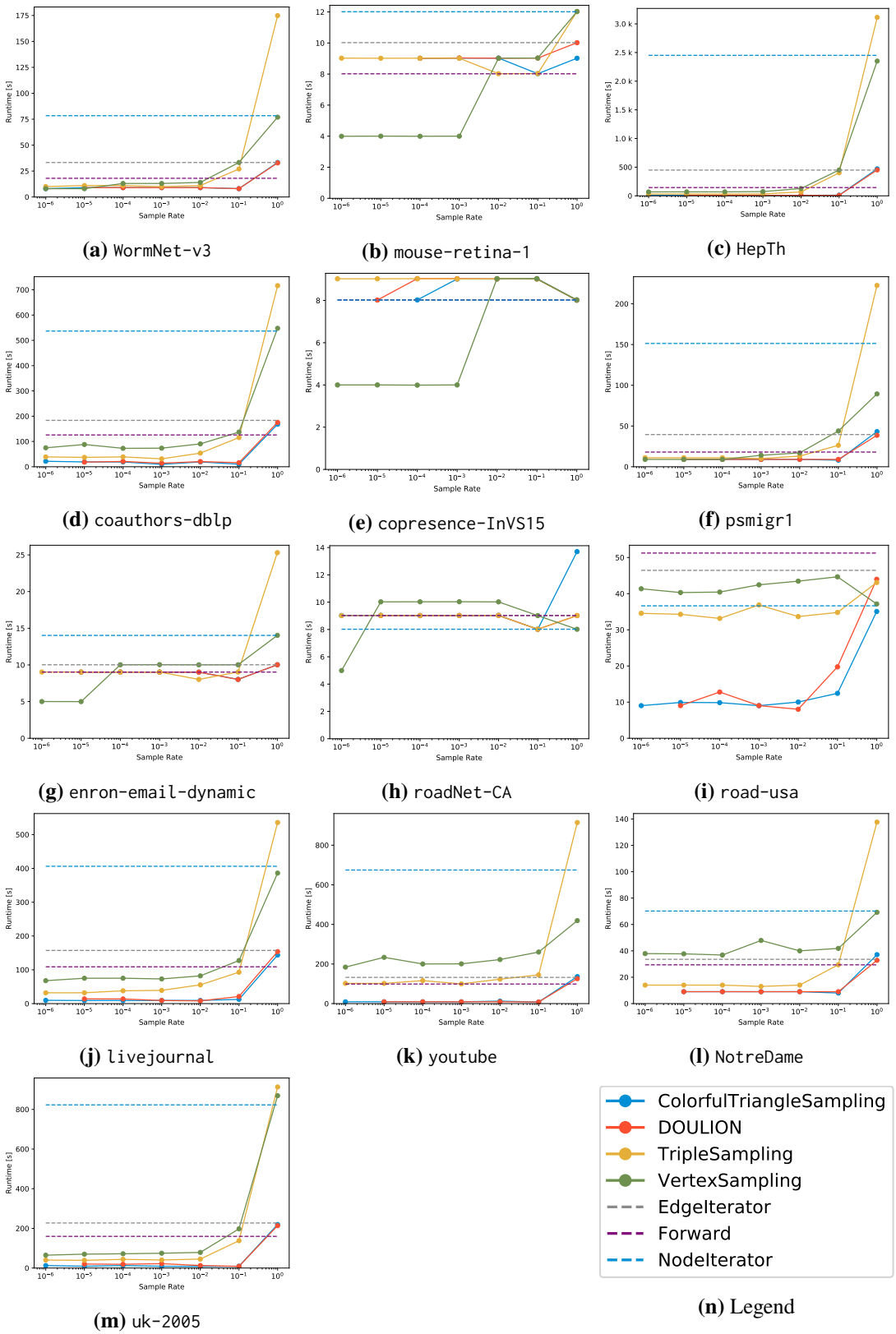
The results of this experiment show that simple approximation schemes can provide approximations with hardly any overhead and show what the causes for the overhead in these algorithms are. However, the measured overhead must not necessarily be inherent to the algorithm itself, but can also be caused indirectly by limitations of software tools and frameworks, as is the case for ELT and RO-DOULION. Therefore, an algorithm also has to be compatible with and be able to harness the concepts supported by the framework, with which the algorithm is to be implemented. Additionally, analyzing the ratio of runtime being used for counting versus messaging shows that counting is by far the more time consuming task in most examined triangle counting algorithms.

5.3.4 Effect of algorithm's design choices on total runtime & results

The overall most important aspects in regards to triangle counting algorithms certainly are their runtime and the quality of their results. This section covers comparisons of the runtime of the presented triangle counting algorithms as well as the quality of their results using different sample rates. Additionally, possible explanations for varying performance are given, where explanations can reasonably be made.

Runtimes

Beginning with algorithm runtimes, figure 5.5 contains plots of algorithm runtimes for sample rates ranging from 10^{-6} to 10^{0} taken at increments of one order of magnitude for all real-world graphs in the dataset described in section 3.4. Exact triangle counting algorithms are plotted using a dashed line, while approximating algorithms are plotted using solid lines. Missing values are caused by either by sample rates being too low for an algorithm, such that the resulting correction factor would be too large to produce triangle count estimates, which would be stored in a 64-bit variable, or there were no edges sampled at all.



(a) WormNet-v3 **(b)** mouse-retina-1 **(c)** HepTh

(d) coauthors-dblp **(e)** copresence-InVS15 **(f)** psmigr1

(g) enron-email-dynamic **(h)** roadNet-CA **(i)** road-usa

(j) livejournal **(k)** youtube **(l)** NotreDame

(m) uk-2005 **(n)** Legend

Figure 5.5: Runtimes of algorithms for different sample rates.

The exact algorithms of EdgeIterator and Forward can outperform the algorithm NodeIterator on almost all graphs apart from road networks. These road networks have vertices with fairly low degrees. In these cases, checking whether a specific vertex is a vertex's neighbor individually, as it is done in the algorithm NodeIterator, may be faster than calculating the intersection of two neighbor lists to count triangles, as it is done in the algorithms EdgeIterator and Forward. For other graphs with higher vertex degrees, however, the method of sending and processing single messages for each pair of neighbors, as is it done for the implementation of NodeIterator, is apparently significantly slower than broadcasting a vertex's neighbor list and intersecting those neighbor lists for counting, which is done by algorithms EdgeIterator and Forward. On these graphs with high vertex degrees, the algorithm Forward can also consistently outperform the algorithm EdgeIterator despite the overhead of filtering "forward edges".

The algorithm ELT is left out in this section, as its runtime results are already discussed in chapter 4. Runtimes of ELT using the implementation, which utilizes cryptographic hash functions for edge sampling, are consistently above runtimes of all other approximate triangle counting algorithms and would therefore only make the diagrams harder to read. Note that this is due to this special implementation, which is necessary due to the need of ELT to keep the complete graph accessible and the limitations of Apache Giraph, as is discussed in chapter 4. As a result, it would not be a fair comparison between ELT and the other algorithms.

The simple approximation algorithms, which can be implemented by an input filter sampling edges during the input phase of the algorithm, show also the best runtime results. This includes the algorithms ColorfulTriangleSampling and DOULION. Both consistently outperform the other algorithms, as they are the only ones in this comparison, which do not have to materialize the complete graph in memory. Having fewer vertices in memory implies fewer vertices having to execute a computation and fewer targets for sending messages to. In the algorithms VertexSampling and TripleSampling vertices have to at least start the computation, even when they are not sampled, because the complete graph has to be materialized in memory and Apache Giraph executes a computation for all vertices that are present.

VertexSampling and TripleSampling can outperform the exact triangle counting algorithms NodeIterator and EdgeIterator on almost all graphs for sample rates below 1, and can even outperform Forward on some graphs with sample rates below 0.1. However, they cannot outperform ColorfulTriangleSampling and DOULION. The few exceptions in which algorithm VertexSampling is faster than all other algorithms on graphs mouse-retina-1 and copresence-InVS15 are likely caused by no vertex being sampled and all vertices immediately becoming inactive. VertexSampling, despite its simplicity, is slower than TripleSampling on most graphs for sample rates below 1. The results suggest that making non-sampled vertices become passive from the beginning of the computation reduces the runtime by a smaller amount than vertices sampling their pairs of neighbors, with which they form triples. Another effect of sampling triples instead of vertices is that the former reduces the amount of messages to be sent for all vertices, which depends on the sampled amount of distinct neighbor pairs and therefore triples, proportional to the sample rate. In contrast, VertexSampling can only make vertices go inactive, which cannot avoid high-degree vertices having a disproportional impact on runtimes.

Analyzing the runtimes of these algorithms shows that the rather simple approaches of ColorfulTriangleSampling and DOULION, which can be implemented using input filters and therefore reduce the amount of vertices and edges present in memory, are the fastest. However, all of

the approximate algorithms discussed in this section can outperform the exact triangle counting algorithms, if the sample rates are chosen small enough. Whether the resulting estimates are still accurate enough to be useful using such small sample rates is discussed in the following.

Estimates

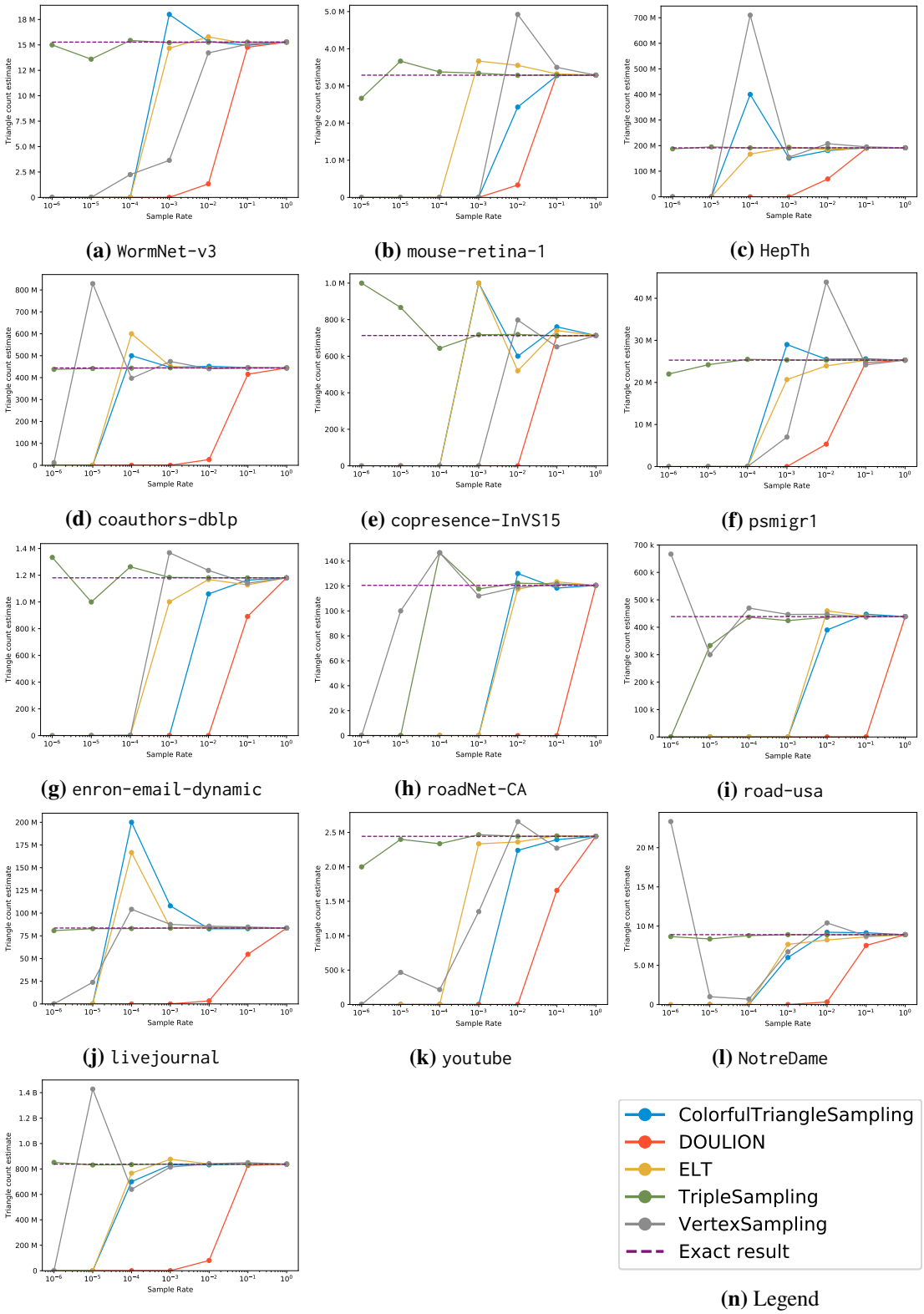
The accuracy of triangle count estimates defines their usefulness. Using very low sample rates is of no use, if no triangles are found as a result. However, it also depends on the graph which sample rates are sufficient for producing useful results, as sampling large graphs with small sample rates can still leave enough edges for forming triangles, whereas a small graph would have hardly any edges left.

The triangle count estimates of the various approximate triangle counting algorithms are depicted in figure 5.6. The dashed line marks the actual, exact triangle count and the solid lines depict the triangle count estimates of the approximate algorithms for varying sample rates. These estimates are more distinct compared to the runtimes of those algorithms. A general, expected trend is that higher sample rates provide estimates, which are closer to the true estimate, and the same is true for algorithms, which sample a bigger portion of the graph by design.

For example, `DOULION`'s results become unstable and drop to 0 with edge sample rates one to two orders of magnitude higher compared to the results of `ELT` and `ColorfulTriangleSampling`. This is due to `DOULION` requiring all three edges of a triangle being sampled in order to count it, i. e. using a sample rate of $p \in (0, 1]$ only a fraction of p^3 of all triangles are sampled. For `ELT` it suffices to sample at least two out of three edges of a triangle in order to count it and consequently a larger fraction of all triangles is sampled. `ColorfulTriangleSampling` can similarly count a triangle, if two out of three edges of a triangle are monochrome, i. e. their adjacent vertices are colored with the same color. As only such monochrome edges are sampled, the third edge is implicitly monochrome and sampled, if the other two edges are monochrome, resulting in a fraction of p^2 of all triangles being sampled.

`VertexSampling` provides varying stability of results depending on the graph. It does provide better estimates than `DOULION` overall, however, it can provide better estimates than `ELT` only on some graphs. This variance in quality of results may be explained by the average amount of triangles per vertex in a graph. `VertexSampling` performs comparatively well on graphs with comparatively few triangles per vertex, e. g. `road-usa`, `roadNet-CA`, `enron-email-dynamic`, and `coauthors-dblp` but performs worse on graphs with a high average of triangles per vertex, e. g. `WormNet-v3`, `mouse-retina-1`, `psmigr1`, `HepTh`, and `copresence-InVS15`. It may therefore be viable to prefer `VertexSampling` for graphs with a high number of vertices, especially if the ratio of triangles per vertex is low, as is the case for the graphs `roadNet-CA` and `road-usa`.

In regards to the quality of estimates, `TripleSampling` appears to be performing best when comparing estimates by sample rate, but this result may not be directly transferable to the case of sampling edges or vertices. The difference lies in the fact that triples of vertices may overlap, i. e. share vertices. Sampling a small portion of triples in the graph may result in large portion of the complete graph being sampled in terms of vertices and edges, as the remaining portion of triples could consist of the same vertices and edges. However, since all triangles in the graph are triples, they are sampled with the same probability p as triples. Because of this, `TripleSampling` samples even more triangles than `ColorfulTriangleSampling` and `ELT`. When comparing the different portions of triangles being



(a) WormNet-v3 **(b)** mouse-retina-1 **(c)** HepTh

(d) coauthors-dblp **(e)** copresence-InVS15 **(f)** psmigr1

(g) enron-email-dynamic **(h)** roadNet-CA **(i)** road-usa

(j) livejournal **(k)** youtube **(l)** NotreDame

(m) uk-2005 **(n)** Legend

Figure 5.6: Triangle count estimates of algorithms for different sample rates.

sampled, one might therefore expect TripleSampling estimates to similarly drop to zero for sample rates one to two orders of magnitude smaller than ColorfulTriangleSampling does, as is the case for the comparison of ColorfulTriangleSampling and DOULION. Contrary to this expectation, estimates of TripleSampling drop either for even smaller sample rates than expected or stay more or less accurate without dropping to zero. This observation leads to the assumption that while TripleSampling overall does sample a larger portion of the graph compared to ColorfulTriangleSampling and ELT, the disproportionately high improvement in accuracy of estimates may be due to ignoring especially those edges, which cannot be part of a triangle anyway. For example, edges are never sampled in the algorithm TripleSampling, if they are not part of an open triple. The combination of sampling a larger portion of the graph and sampling especially those edges, which are relevant for forming a triangle, are likely reasons for the quality of estimates of the algorithm TripleSampling, which produces by far the most accurate estimates in this comparison.

The quality of results of approximate triangle counting algorithms seems to be fundamentally reliant on the method of sampling. The more of a graph is sampled, the better are the results, but that results in longer runtimes. However, sampling in smart ways can allow producing accurate results while still sampling comparatively small portions of the input graph. Dropping preferably edges which cannot be part of a triangle, as TripleSampling does, can improve an algorithm without introducing another compromise between accuracy of estimates and resulting runtime.

5.4 Conclusion

This chapter has taken a look at statistical relationships between graph structures and triangle counts as well as compare runtime behavior and results of several triangle counting algorithms. The comparison of the triangle counting algorithms lead to multiple insights.

Algorithms using a simple mechanism to randomly drop edges, such as ColorfulTriangleSampling and DOULION, showed the best overall runtimes. Other approaches, which required the original graph structure to be present in order to sample triples (TripleSampling) or for non-sampled vertices to respond to messages (VertexSampling), seem to cause significantly more overhead in this setting simply by having more vertices, for which the computation has to be executed. The implementation of the algorithm ELT used in these experiments additionally has a large portion of its runtime spent on messaging, which is caused by the use of cryptographic hash functions for edge sampling in this implementation, whereas the other algorithms generally use most of their runtime for actually determining the triangle count based on received messages.

It has to be noted, however, that much of this described overhead may be caused by the graph processing framework not natively supporting the primitives required by the algorithms. For example, the algorithm ELT would benefit greatly, if edges could not only be dropped but also manipulated during the input process. Likewise, TripleSampling could be faster, if sampling triples directly would be possible. Implementing the various algorithms using Apache Giraph showed that the capabilities of the framework can greatly influence such a comparison of algorithms, as the resulting implementations might have to take inefficient compromises in order to actually implement a certain algorithm. Therefore, the results of the runtime analysis have to be used with care.

Results of the comparison of the estimates produced by the triangle counting algorithms do not suffer from this issue. In general, the accuracy of estimates was primarily determined by the size of the portion of the graph being sampled, since the more parts of the graph are used for estimation, the more accurate the results are. While this does impact algorithm runtimes, sampling the graph in a smart way can allow for better compromises between sample size and algorithm runtime, as long as the computational overhead introduced by the sampling procedure is not too large. This can be achieved, for example, by systematically ignoring edges or vertices which cannot form a triangle.

Analysis of statistical relationships between graph structures and triangle counts revealed a strong correlation between a vertex's degree and the average local triangle count of vertices with that degree. This may constitute starting point for a different method of calculating triangle count estimates, which take into account the degree of vertices. The next chapter continues at that point by presenting a method for calculating triangle count estimates by utilizing vertices' degrees and evaluates the viability of this approach.

6 Degree-assisted Estimation

In the previous chapter it was discovered that the average local triangle count strongly correlates with the vertex degree in many graphs, which were considered. The goal of this chapter is to exploit this regularity of the average local triangle count to improve global triangle count estimates by extrapolating average local triangle count measures to vertices, which are not sampled. In the first part of the chapter, the method of using sampled vertices to extrapolate their average local triangle count to non-sampled vertices is explained. The second part of the chapter compares results of the regular `VertexSampling` algorithm with the results gained by additionally considering the vertices' degrees.

6.1 Local triangle count extrapolation

Strong correlations indicate an existing statistical relationship, which in this case could allow predictions regarding non-sampled vertices. By using additional information in the form of a vertex's degree when extrapolating, one would assume the estimates should become more accurate than without using that additional information, as long as this relationship holds sufficiently and is not overshadowed by other factors. The proposed method consists of four steps:

1. Sample local triangle counts of a limited number of vertices
2. Calculate averages of local triangle counts per vertex degree value
3. Calculate a regression polynomial r modeling the relationship between vertex degree and average triangle count from samples
4. Estimate local triangle counts of all vertices in the graph based on their degree and the regression polynomial r

In order for the last step to be performed more easily it is assumed that the graph is available as an adjacency list, from which a vertex's degree can be read more easily by counting the vertex's neighbors. If an edge list is used as graph representation format, the occurrences of a vertex in the source of edges can be counted, such that the degree of all vertices can be obtained this way. These two variants provide the degree of all vertices in time $O(|V|)$ or $O(|E|)$, respectively.

Because of the measurement results of section 5.3.1 it is expected for many graphs that all of their vertices exhibit a similar – ideally linear – relationship between their degree value and their local triangle count. Therefore, a small sample of vertices should suffice in these cases to generate a regression polynomial r of the average triangle counts by vertex degree using regression analysis. Applying this regression polynomial r to the degree values of the complete set of vertices should then produce an estimate of those vertices' local triangle counts, as those vertices are expected to follow the relationship described above. The details of this process are as follows.

In the first step, the algorithm `VertexSampling` is executed without any modifications. The results are local triangle counts of a subset of vertices in the graph. Additionally, let the output of each vertex contain its own degree. This figure can easily be both accessed and added to the result in Apache Giraph. The local triangle counts obtained this way are then averaged for all vertices with the same degree such that the same measure as in Figure 5.2 is calculated, which depicts the observed relationship between vertex degrees and average local triangle counts. At that point, the average triangle counts need to show a similarly strong correlation with the vertex degrees for the further steps to succeed.

If the average triangle counts do show this correlation, the next step is to perform a regression analysis on them. Although the measured correlations indicate a linear relationship, i. e. a linear regression should also produce usable results, most graphs in Figure 5.2 show a slight curvature on closer observation. A polynomial regression should therefore be better able to model this relationship and, furthermore, could also describe a linear relationship, as well. Performing the regression analysis is done using a weighted least squares fit method [Bra14] implemented in the Python library *NumPy*. Each average local triangle count of vertices with degree d in the sample is weighted by $\sqrt{n}$, with n being the number of vertices with degree d in the sample, to account for the standard error of the sample. The result of this regression analysis, a regression function r, is expected to model how many triangles on average a vertex of the same graph with a certain degree value is expected to be a part of.

In the last step, this polynomial r is used to produce a global triangle count estimate from the degree value of each vertex in the complete graph. In order to achieve this, the amount of vertices having a certain degree is needed for all possible degree values d of the graph. Let that number of vertices with degree d of a graph $G(V,\ E)$ be denoted as

$$n_d(V) = |\{v \in V \mid d(v) = d\}|. \tag{6.1}$$

As this information is usually not collected for all vertices of the graph during a run of the algorithm `VertexSampling`, it has to be obtained from the original graph input file. Then, the expected amount of triangles adjacent to a vertex v is calculated for all vertices in the graph. This is done by multiplying the count of vertices with a certain degree value d, i. e. $n_d(V)$, with the expected average number of triangles for that vertex degree value d, which is obtained from the regression polynomial by evaluating $r(d)$. The global triangle count estimate is then calculated by summing up these values and applying the correction factor $c_\Delta = p^{-1}$ of the algorithm `VertexSampling`. Let d_{max} be the maximum degree of all vertices in $G(V,\ E)$. The global triangle count estimate Λ of a graph $G(V,\ E)$ can then be calculated using the following formula:

$$\Lambda(G(V,E)) = \sum_{d=2}^{d_{max}} n_d(V) * r(d) * \frac{1}{p} \tag{6.2}$$

Using this method of sampling, regression, and prediction, the estimates of the algorithm `VertexSampling` can be processed to use more of the available properties of the graph in order to hopefully deliver more accurate estimates of triangle counts. The amount to which this method can actually improve results is examined in the next section.

Graph	Sample Rate	DAE estimate	Relative deviation (%)
roadNet-CA	1	120 492	2.22e−16
	1e−1	120 463.333 333 333	4.44e−16
	1e−2	121 666.666 666 667	2.22e−16
	1e−3	126 333.333 333 333	4.44e−16
	1e−4	126 666.666 666 667	−1.11e−16
	1e−5	66 666.666 666 667	−2.22e−16
	1e−6	333 333.333 333 333	−3.33e−16
WormNet-v3	1	15 279 134	−1.11e−16
	1e−1	14 972 123.333 333 400	1.33e−15
	1e−2	18 523 833.333 333 300	−1.78e−15
	1e−3	4 970 666.666 666 660	−3.33e−16
	1e−4	0	0
	1e−5	0	0
	1e−6	0	0
NotreDame	1	8 910 005.000 000 010	1.11e−15
	1e−1	8 655 793.333 333 330	2.22e−16
	1e−2	8 813 466.666 666 680	8.88e−16
	1e−3	14 095 666.666 666 700	1.33e−15
	1e−4	6 503 333.333 333 330	0
	1e−5	1 233 333.333 333 330	−3.33e−16
	1e−6	0	0

Table 6.1: Excerpt of degree-assisted triangle count estimates with relative deviation from regular `VertexSampling` estimates.

6.2 Quality of estimates

In order to be an effective improvement, the estimates produced with the method of degree-assisted estimation (DAE) described above need to be closer to the actual count of triangles in the graph than the estimate produced by the algorithm `VertexSampling`. The following describes they way in which `VertexSampling` estimates and DAE estimates are compared and discusses the results produced by this comparison.

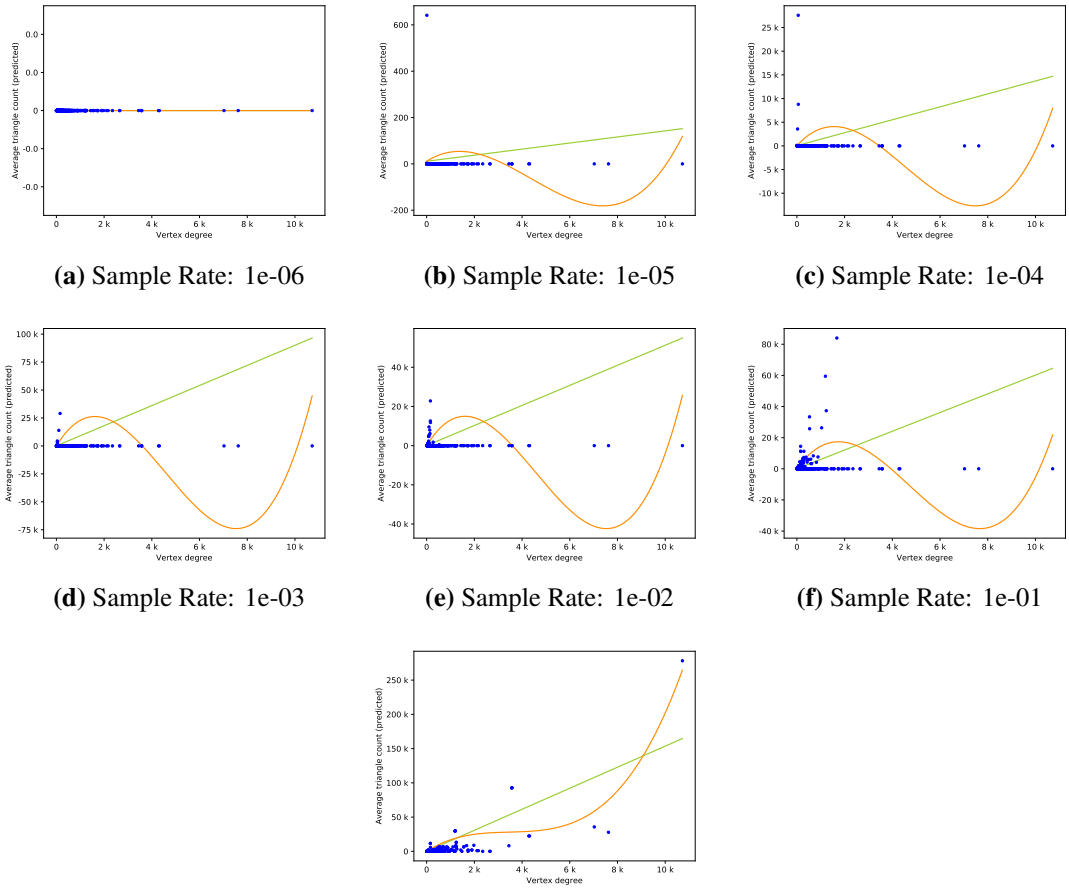
Both approaches are compared solely based on the estimated global triangle count estimates they produce. For the algorithm `VertexSampling` this is the regular triangle count estimate it produces. For the DAE method, the sample of vertices and their corresponding local triangle counts are taken from the algorithm `VertexSampling`. Based on this sample, a regression function describing the average triangle count for vertex degree values is calculated. For this comparison, both a linear and a cubic regression are used. The reason for this is that both Pearson's and Spearman's correlation coefficients indicate strong relationships in the graph dataset used in this work. The estimate used for this comparison is the triangle count produced as described in equation (6.2) using the aforementioned regression functions and the frequencies of vertex degree values obtained from the original graph.

Table 6.1 displays an excerpt of estimates produced by the DAE method using linear regression alongside the corresponding sample rates and the relative deviation of the DAE estimate from the estimate produced by the algorithm `VertexSampling` in percent. The rest of the results is nearly identical in terms of the relative deviation of results and is therefore omitted in the table. The estimates of the DAE method themselves are very close to the ones produced by the algorithm `VertexSampling`. DAE estimates deviate by percentages in the range of 1.11e−16% to 4.88e−15%, i. e. 0.1 to 1 quadrillionth of a percent for all sample rates and all real-world graphs in the complete dataset of section 3.4. The results can therefore be considered to be essentially equal to the estimates produced by the algorithm `VertexSampling`. This indicates that the average local triangle count distribution calculated from a sample of vertices is not a sufficiently accurate approximation of the distribution in the complete graph. The DAE method could therefore not provide an additional benefit to improve estimates at all. The results of the DAE method using a cubic polynomial are very similar. DAE estimates produced using a cubic regression polynomial deviate from the `VertexSampling` estimates by 1.11e−16% to 3.23e−14%, i. e. up to one order of magnitude more compared to the case using linear regression. However, no graph on Network Repository [RA15] was found for which this would have made a difference of at least one triangle in the global triangle count.

It is no surprise that the local triangle counts of the sample of vertices taken from `VertexSampling` can model the average triangle count distribution equally well as the regression function, as the latter is created from the former. However, one might expect that including the remaining vertices of the graph would then improve estimates by applying the modeled average triangle count distribution to all vertices in the graph, including those previously not considered by `VertexSampling`. In reality, the regression function appears to be unable to model the average local triangle count distribution in the original graph accurately enough. This can be seen in figure 6.1, which shows both the linear (green) and cubic (orange) regression functions calculated from the graph `NotreDame` for different sample rates. The graph `NotreDame` was chosen because it shows the largest differences for varying sample rates and therefore allows to see those differences more clearly. The average local triangle count samples (blue) used to generate the regression functions are depicted in the figure, as well.

Figures 6.1b to 6.1f, which show the regression functions for sample rates from 1e−5 to 1e−1, depict those regression functions varying substantially from the case of a sample rate of 1.0. The most obvious difference is the varying shape of the cubic regression polynomial, which even assumes negative values in these figures. Additionally, the range of values for both regression functions significantly varies for different sample rates, too. Due to this disparity in distributions, the triangle count in the original graph is therefore calculated assuming an unsuitable triangle count distribution in these cases of small sample rates. As a result, the additional information used by considering the frequencies of all degree values in the complete graph cannot better estimates. Merely utilizing the assumption that a strong statistical relationship between a vertex degree and the average local triangle count at vertices with that degree value exists does not help correcting that disparity. The quality of the sample therefore appears to be defining the quality of results in such a way that extrapolating the average triangle count distribution to the complete graph does not have a significant effect.

Having investigated the potential use of the statistical relationship between vertex degree values and the average count of triangles incident to vertices with those degree values, it has to be concluded that it appears to not be useful for improving triangle count estimates produced by the algorithm `VertexSampling`. The quality of the sample used for extrapolating local triangle counts for the



(a) Sample Rate: 1e-06 **(b)** Sample Rate: 1e-05 **(c)** Sample Rate: 1e-04

(d) Sample Rate: 1e-03 **(e)** Sample Rate: 1e-02 **(f)** Sample Rate: 1e-01

(g) Sample Rate: 1.0

Figure 6.1: Average local triangle counts for varying vertex degrees and sample rates (blue) and the corresponding linear (green) and cubic (orange) regression functions of the graph `NotreDame`.

complete graph is far more important for the accuracy of the global triangle count estimate. What remains, however, is the knowledge that the aforementioned statistical relationship exists, which may prove useful in another context.

7 Conclusion and future work

The problem of counting triangles in a graph has been examined in this work with a focus on approximations. A number of approximate triangle counting algorithms were implemented using the graph processing framework Apache Giraph in order to enable a comparison between common sparsification techniques and algorithms. The results of this analysis include, on the one hand, insights into the structural properties of graphs and statistical relationships of them with the local triangle counts of vertices in those graphs. A significant relationship between the degree of a vertex and the average local triangle count of vertices in the graph with that degree was found with a Pearson's r between 0.8 to 0.99 for most examined graphs. This result may especially be of use in graph analytics.

On the other hand, the analysis produced insights into the aspects, which make approximate triangle counting algorithms fast and accurate. The most simple approximating algorithms proved to be the fastest, since they make the least assumptions, which in turn allows implementations without inefficient workarounds due to framework or language limitations. Reimplementing the algorithms examined in this work showed that the compatibility of an algorithm's primitives with the supported operations of a framework is an important factor for the overall runtime of an implementation. In terms of accuracy, those algorithms performed best, which sampled the largest portion of the graph. While this is to be expected, the best compromise of runtime and accuracy is achieved by algorithms sampling the graph in smart ways. Following that advice, an algorithm should avoid wasting computational resources on redundant computations or on parts of a graph, which cannot contain triangles anyway.

However, results of performing this analysis on the problem of triangle counting and on algorithms for counting triangles are not restricted to these insights. A new method of implementing decentralized but consistent sampling decisions based on the random oracle model has been developed to enable the implementation of a triangle counting algorithm despite limitations of the graph processing framework. Using cryptographic hash functions as an implementation substitute for a random oracle, the sampling of edges was implemented utilizing their pseudorandom output to make the pseudorandom sampling decision. A detailed description of this method and its implementation is provided along with performance comparisons on real-world graphs. This technique of decentralized sampling decisions is not restricted to graph sparsification but may very well be used in other fields, as well. Using cryptographic hash functions for making decisions is possible in every situation, in which a pseudorandom decision has to be made consistently and deterministically based on an input available to the processes having to make such a decision. This approach allows to reduce communication, as sampling decisions do not have to be communicated, if every process can simply reproduce that decision. It can therefore be used for all kinds of algorithms, in which pseudorandom decisions are used to e. g. omit computations.

Based on the statistical relationship found between vertices' degrees and their average local triangle count, a new approach for producing a triangle count estimate based on vertices' degrees was presented. By extrapolating the distribution of average local triangle counts across vertex degree values from a sample of vertices to the complete graph, it was possible to match triangle count estimates calculated using an existing triangle counting algorithm based on vertex sampling. While an improvement of estimates was not possible using this method, the results show that a regression polynomial modeling the distribution of average local triangle counts of vertices by degree is equally suitable for producing a triangle count estimate. Future work may therefore examine methods for incrementally building a sample of vertices and their local triangle counts to incrementally approximate the average local triangle count distribution of the graph. Such an approach could enable approximating triangle counting algorithms, which produce more accurate results the longer they can run but can be terminated at any time, if available computation time runs out, without the intermediate results being of no use.

Finally, some general remarks on triangle counting algorithms as presented in this work are made. The limiting factor for executing large graphs in the experiments of this work, at least with Apache Giraph, was the memory required by a computation. The memory usage is primarily caused by the size of the graph to be processed and the amount and size of messages exchanged during the computation. CPU usage in contrast was fairly low, as the long-lasting processing of high-degree vertices on one core lead to most other cores idling, since the computation cannot continue with the next superstep until all vertices finished the current superstep. Parallelizing computations of a single vertex may therefore significantly improve runtimes for algorithms using the Bulk Synchronous Parallel computational model and having vertices requiring different amounts of CPU time for processing.

Bibliography

[Adv17] Advanced Micro Devices, Inc. *AMD Random Number Generator*. https://www.amd.com/system/files/TechDocs/amd-random-number-generator.pdf. June 2017 (cit. on p. 36).

[AJ05] H. Aarag, J. Jennings. "HashNAT: A Distributed Packet Rewriting System Using Adaptive Hash Functions". In: *Sixth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing and First ACIS International Workshop on Self-Assembling Wireless Networks (SNPD/SAWN'05)*. IEEE, 2005, pp. 282–287. DOI: 10.1109/snpd-sawn.2005.42 (cit. on p. 30).

[AJB99] R. Albert, H. Jeong, A.-L. Barabási. "Diameter of the World-Wide Web". In: *Nature* 401.6749 (Sept. 1999), pp. 130–131. DOI: 10.1038/43601 (cit. on p. 31).

[ALB+13] K. Amunts, C. Lepage, L. Borgeat, H. Mohlberg, T. Dickscheid, M.-É. Rousseau, S. Bludau, P.-L. Bazin, L. B. Lewis, A.-M. Oros-Peusquens, N. J. Shah, T. Lippert, K. Zilles, A. C. Evans. "BigBrain: An Ultrahigh-Resolution 3D Human Brain Model". In: *Science* 340.6139 (2013), pp. 1472–1475 (cit. on p. 31).

[Apa] Apache Software Foundation. *Apache Giraph*. online. URL: http://giraph.apache.org (cit. on pp. 15, 16, 23).

[AS14] C. Aggarwal, K. Subbian. "Evolutionary Network Analysis". In: *ACM Computing Surveys* 47.1 (May 2014), pp. 1–36. DOI: 10.1145/2601412 (cit. on p. 15).

[AYZ97] N. Alon, R. Yuster, U. Zwick. "Finding and counting given length cycles". In: *Algorithmica* 17.3 (Mar. 1997), pp. 209–223. DOI: 10.1007/bf02523189 (cit. on p. 27).

[BBCG08] L. Becchetti, P. Boldi, C. Castillo, A. Gionis. "Efficient semi-streaming algorithms for local triangle counting in massive graphs". In: *Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD 08*. ACM Press, 2008. DOI: 10.1145/1401890.1401898 (cit. on p. 15).

[Ber08] D. J. Bernstein. "ChaCha, a variant of Salsa20". In: *Workshop Record of SASC*. Vol. 8. DocumentID: 4027b5256e17b9796842e6d0f68b0b5e. Jan. 2008. URL: https://cr.yp.to/chacha.html#chacha-paper (cit. on p. 36).

[BFL+06] L. S. Buriol, G. Frahling, S. Leonardi, A. Marchetti-Spaccamela, C. Sohler. "Counting triangles in data streams". In: *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems - PODS '06*. ACM Press, 2006. DOI: 10.1145/1142351.1142388 (cit. on pp. 28, 29).

[BMSW12] D. A. Bader, H. Meyerhenke, P. Sanders, D. Wagner. "Graph Partitioning and Graph Clustering". In: *10th DIMACS Implementation Challenge Workshop*. 2012 (cit. on p. 31).

[BR93] M. Bellare, P. Rogaway. “Random oracles are practical”. In: *Proceedings of the 1st ACM conference on Computer and communications security - CCS ’93*. ACM Press, 1993. DOI: 10.1145/168588.168596 (cit. on p. 35).

[Bra14] S. Brandt. “Linear and Polynomial Regression”. In: *Data Analysis*. Springer International Publishing, 2014, pp. 321–329. DOI: 10.1007/978-3-319-03762-2_12 (cit. on p. 66).

[BRSV11] P. Boldi, M. Rosa, M. Santini, S. Vigna. “Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks”. In: *WWW*. 2011, pp. 587–596 (cit. on p. 31).

[CEK+15] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, S. Muthukrishnan. “One trillion edges”. In: *Proceedings of the VLDB Endowment* 8.12 (Aug. 2015), pp. 1804–1815. DOI: 10.14778/2824032.2824077 (cit. on p. 15).

[Coh] W. Cohen. *Enron email dataset*. http://www.cs.cmu.edu/~enron/. Accessed in 2009. (cit. on p. 31).

[CSH+14] A. Cho, J. Shin, S. Hwang, C. Kim, H. Shim, H. Kim, H. Kim, I. Lee. “WormNet v3: a network-assisted hypothesis-generating server for Caenorhabditis elegans”. In: *Nucleic acids research* 42.W1 (2014), W76–W82 (cit. on p. 31).

[DE12] S. van Dongen, A. J. Enright. “Metric distances derived from cosine similarity and Pearson and Spearman correlations”. In: (Aug. 14, 2012). arXiv: http://arxiv.org/abs/1208.3145v1 [stat.ME] (cit. on pp. 24, 47).

[DKH12] D.-Z. Du, K.-I. Ko, X. Hu. *Design and Analysis of Approximation Algorithms*. Springer New York, 2012. DOI: 10.1007/978-1-4614-1701-9 (cit. on p. 30).

[ELT16] R. Etemadi, J. Lu, Y. H. Tsin. “Efficient Estimation of Triangles in Very Large Graphs”. In: *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management - CIKM ’16*. ACM Press, 2016. DOI: 10.1145/2983323.2983849 (cit. on p. 27).

[EM02] J.-P. Eckmann, E. Moses. “Curvature of co-links uncovers hidden thematic layers in the World Wide Web”. In: *Proceedings of the National Academy of Sciences* 99.9 (Apr. 2002), pp. 5825–5829. DOI: 10.1073/pnas.032093399 (cit. on p. 15).

[FEHP10] C. Forbes, M. Evans, N. Hastings, B. Peacock. “Bernoulli Distribution”. In: *Statistical Distributions*. John Wiley & Sons, Inc., Dec. 2010, pp. 53–54. DOI: 10.1002/9780470627242.ch7 (cit. on p. 20).

[GBV+14] Y. Guo, M. Biczak, A. L. Varbanescu, A. Iosup, C. Martella, T. L. Willke. “How Well Do Graph-Processing Platforms Perform? An Empirical Performance Evaluation and Analysis”. In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, May 2014. DOI: 10.1109/ipdps.2014.49 (cit. on p. 23).

[GJL+18] A. Grewal, J. Jiang, G. Lam, T. Jung, L. Vuddemarri, Q. Li, A. Landge, J. Lin. “RecService: Distributed Real-Time Graph Processing at Twitter”. In: *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*. Boston, MA: USENIX Association, July 2018. URL: https://www.usenix.org/conference/hotcloud18/presentation/grewal (cit. on p. 15).

[GR11] J. Gantz, D. Reinsel. *Extracting Value from Chaos*. https://web.archive.org/web/20160313205027/https://www.emc.com/collateral/analyst-reports/idc-extracting-value-from-chaos-ar.pdf. Retrieved on 5th December 2019. June 2011 (cit. on p. 15).

[Has16] M. A. Hasan. "Methods and Applications of Network Sampling". In: *Optimization Challenges in Complex, Networked and Risky Systems*. INFORMS, Oct. 2016, pp. 115–139. DOI: 10.1287/educ.2016.0147 (cit. on p. 28).

[HD17] M. A. Hasan, V. S. Dave. "Triangle counting in large networks: a review". In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 8.2 (Oct. 2017), e1226. DOI: 10.1002/widm.1226 (cit. on pp. 20, 28).

[HZZ+15] W. Han, X. Zhu, Z. Zhu, W. Chen, W. Zheng, J. Lu. "Weibo, and a Tale of Two Worlds". In: *Proceedings of the 2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2015 - ASONAM '15*. ACM Press, 2015. DOI: 10.1145/2808797.2809333 (cit. on p. 15).

[IPV+18] A. P. Iyer, A. Panda, S. Venkataraman, M. Chowdhury, A. Akella, S. Shenker, I. Stoica. "Bridging the GAP". In: *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) - GRADES-NDA '18*. ACM Press, 2018. DOI: 10.1145/3210259.3210269 (cit. on p. 16).

[Joh73] D. S. Johnson. "Approximation algorithms for combinatorial problems". In: *Proceedings of the fifth annual ACM symposium on Theory of computing - STOC '73*. ACM Press, 1973. DOI: 10.1145/800125.804034 (cit. on p. 16).

[JSP13] M. Jha, C. Seshadhri, A. Pinar. "A space efficient streaming algorithm for triangle counting using the birthday paradox". In: *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '13*. ACM Press, 2013. DOI: 10.1145/2487575.2487678 (cit. on p. 29).

[JTN+14] V. Joshi, A. Tsukerman, A. Nithrakashvap, J. Shi, T. Bosman. "Maintaining item-to-node mapping information in a distributed system". English. U.S. pat. 8671151. Mar. 2014. URL: https://patents.google.com/patent/US8671151B2/en (cit. on p. 30).

[KGB01] B. Klots, A. Ganesh, R. J. Bamford. "Methodology for hosting distributed objects at a predetermined node in a distributed system". English. U.S. pat. 6173313. Jan. 2001. URL: https://patents.google.com/patent/US6173313B1/en (cit. on p. 30).

[KGE11] P. Kulkarni, P. Gupta, M. Ercegovac. "Trading Accuracy for Power with an Underdesigned Multiplier Architecture". In: *2011 24th Internatioal Conference on VLSI Design*. IEEE, Jan. 2011. DOI: 10.1109/vlsid.2011.51 (cit. on p. 29).

[KM15] N. Koblitz, A. J. Menezes. "The random oracle model: a twenty-year retrospective". In: *Designs, Codes and Cryptography* 77.2-3 (May 2015), pp. 587–610. DOI: 10.1007/s10623-015-0094-2 (cit. on p. 36).

[LBG+12] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, J. M. Hellerstein. "Distributed GraphLab". In: *Proceedings of the VLDB Endowment* 5.8 (Apr. 2012), pp. 716–727. DOI: 10.14778/2212351.2212354 (cit. on p. 15).

[LK14] J. Leskovec, A. Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data. June 2014 (cit. on p. 45).

[LKF05] J. Leskovec, J. Kleinberg, C. Faloutsos. “Graphs over time: densification laws, shrinking diameters and possible explanations”. In: *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. ACM. 2005, pp. 177–187 (cit. on p. 31).

[MAB+10] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, G. Czajkowski. “Pregel”. In: *Proceedings of the 2010 international conference on Management of data - SIGMOD ’10*. ACM Press, 2010. DOI: 10.1145/1807167.1807184 (cit. on p. 22).

[Mar58] D. W. Martin. “Runge-Kutta Methods for Integrating Differential Equations on High Speed Digital Computers”. In: *The Computer Journal* 1.3 (Mar. 1958), pp. 118–123. DOI: 10.1093/comjnl/1.3.118 (cit. on p. 16).

[MBL+12] I. Monedero, F. Biscarri, C. León, J. I. Guerrero, J. Biscarri, R. Millán. “Detection of frauds and other non-technical losses in a power utility using Pearson coefficient, Bayesian networks and decision trees”. In: *International Journal of Electrical Power & Energy Systems* 34.1 (Jan. 2012), pp. 90–98. DOI: 10.1016/j.ijepes.2011.09.009 (cit. on pp. 24, 47).

[MI18] J. P. Mechalas, Intel Corporation. *Intel® Digital Random Number Generator (DRNG) Software Implementation Guide*. https://software.intel.com/content/www/us/en/develop/articles/intel-digital-random-number-generator-drng-software-implementation-guide.html. Oct. 2018 (cit. on p. 36).

[Mit16] S. Mittal. “A Survey of Techniques for Approximate Computing”. In: *ACM Computing Surveys* 48.4 (Mar. 2016), pp. 1–33. DOI: 10.1145/2893356 (cit. on pp. 15, 29).

[MMG+07] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, B. Bhattacharjee. “Measurement and Analysis of Online Social Networks”. In: *Proceedings of the 5th ACM/Usenix Internet Measurement Conference (IMC’07)*. San Diego, CA, Oct. 2007 (cit. on p. 31).

[MSL15] C. Martella, R. Shaposhnik, D. Logothetis. *Practical Graph Analytics with Apache Giraph*. Apress, 2015. DOI: 10.1007/978-1-4842-1251-6 (cit. on p. 23).

[Mue18] S. Mueller. *Analysis of the Linux Random Number Generator*. Tech. rep. BSI – Federal Office for Information Security, Mar. 2018. URL: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Studies/LinuxRNG/LinuxRNG_EN.pdf (cit. on p. 36).

[Neo15] Neo4j. *Walmart Optimizes Customer Experience with Real-time Recommendations*. https://web.archive.org/web/20200411103517/http://neo4j.com:80/case-studies/walmart/. 2015 (cit. on p. 15).

[Neo17] Neo4j. *Real-time Graph Analysis of Financial Data Creates Potential for Millions in Fraud Detection Savings*. https://neo4j.com/case-studies/fortune-500-financial-services/?ref=blog. 2017 (cit. on p. 15).

[Neo19] Neo4j. *Next-Gen Master Data Management Comes to Marketing at Pitney Bowes*. https://neo4j.com/case-studies/ebay/?ref=blog. 2019 (cit. on p. 15).

[Neu93] J. von Neumann. “First draft of a report on the EDVAC”. In: *IEEE Annals of the History of Computing* 15.4 (1993), pp. 27–75. DOI: 10.1109/85.238389 (cit. on p. 22).

[New03] M. E. J. Newman. “The Structure and Function of Complex Networks”. In: *SIAM Review* 45.2 (Jan. 2003), pp. 167–256. DOI: 10.1137/s003614450342480 (cit. on p. 15).

[Ora20] Oracle. *Java Security Standard Algorithm Names*. https://docs.oracle.com/en/java/javase/14/docs/specs/security/standard-names.html. 2020 (cit. on p. 37).

[Pea72] T. Pearcey. "Distributed Computer Systems". In: *Australian Computer Journal* 4.1 (Feb. 1972), pp. 3–11 (cit. on p. 15).

[PT12] R. Pagh, C. E. Tsourakakis. "Colorful triangle counting and a MapReduce implementation". In: *Information Processing Letters* 112.7 (Mar. 2012), pp. 277–281. DOI: 10.1016/j.ipl.2011.12.007 (cit. on p. 27).

[RA15] R. A. Rossi, N. K. Ahmed. "The Network Data Repository with Interactive Graph Analytics and Visualization". In: *AAAI*. 2015. URL: http://networkrepository.com (cit. on pp. 31, 45, 68).

[Res19] Y. Research. *Yahoo! altavista web page hyperlink connectivity graph, circa 2002*. online. Dec. 2019. URL: http://webscope.sandbox.yahoo.com/ (cit. on p. 15).

[RH13] M. Rahman, M. A. Hasan. "Approximate triangle counting algorithms on multi-cores". In: *2013 IEEE International Conference on Big Data*. IEEE, Oct. 2013. DOI: 10.1109/bigdata.2013.6691744 (cit. on p. 28).

[RH14] M. Rahman, M. A. Hasan. "Sampling Triples from Restricted Networks using MCMC Strategy". In: *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management - CIKM '14*. ACM Press, 2014. DOI: 10.1145/2661829.2662075 (cit. on p. 28).

[Sch07] T. Schank. "Algorithmic Aspects of Triangle-Based Network Analysis". en. In: (2007). DOI: 10.5445/IR/1000007159 (cit. on pp. 46, 57).

[Sch19] M. Schramm. "Approximating distributed graph algorithms". en. MA thesis. University of Stuttgart, 2019. DOI: 10.18419/OPUS-10383 (cit. on pp. 15, 16).

[SK13] J. Sartori, R. Kumar. "Branch and Data Herding: Reducing Control and Memory Divergence for Error-Tolerant GPU Applications". In: *IEEE Transactions on Multimedia* 15.2 (Feb. 2013), pp. 279–290. DOI: 10.1109/tmm.2012.2232647 (cit. on p. 29).

[Sla83] P. B. Slater. *Migration regions of the United States: two county-level 1965-70 analyses*. Community and Organization Research Institute, University of California, 1983 (cit. on p. 31).

[SMS+17] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, M. T. Özsu. "The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing: Extended Survey". In: *The VLDB Journal, 2019* (Sept. 10, 2017). DOI: 10.1007/s00778-019-00548-x. arXiv: http://arxiv.org/abs/1709.03188v3 [cs.DB] (cit. on p. 15).

[Soc] SocioPatterns. *Infectious contact networks*. URL: http://www.sociopatterns.org/datasets/ (cit. on p. 31).

[SRB07] G. J. Székely, M. L. Rizzo, N. K. Bakirov. "Measuring and testing dependence by correlation of distances". In: *The Annals of Statistics* 35.6 (Dec. 2007), pp. 2769–2794. DOI: 10.1214/009053607000000505 (cit. on pp. 24, 47).

[SW05] T. Schank, D. Wagner. “Finding, Counting and Listing All Triangles in Large Graphs, an Experimental Study”. In: *Experimental and Efficient Algorithms*. Extended version available at https://i11www.iti.kit.edu/extra/publications/sw-fclt-05_t.pdf. Springer Berlin Heidelberg, 2005, pp. 606–609. DOI: 10.1007/11427186_54 (cit. on pp. 25, 27).

[SWW+12] Z. Sun, H. Wang, H. Wang, B. Shao, J. Li. “Efficient subgraph matching on billion node graphs”. In: *Proceedings of the VLDB Endowment* 5.9 (May 2012), pp. 788–799. DOI: 10.14778/2311906.2311907 (cit. on p. 15).

[Tka03] T. E. Tkacik. “A Hardware Random Number Generator”. In: *Cryptographic Hardware and Embedded Systems - CHES 2002*. Springer Berlin Heidelberg, 2003, pp. 450–453. DOI: 10.1007/3-540-36400-5_32 (cit. on p. 36).

[TKMF09] C. E. Tsourakakis, U. Kang, G. L. Miller, C. Faloutsos. “DOULION”. In: *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD ’09*. ACM Press, 2009. DOI: 10.1145/1557019.1557111 (cit. on pp. 15, 27, 38, 46).

[Val90] L. G. Valiant. “A bridging model for parallel computation”. In: *Communications of the ACM* 33.8 (Aug. 1990), pp. 103–111. DOI: 10.1145/79173.79181 (cit. on pp. 21, 22).

[VVL+15] A. L. Varbanescu, M. Verstraaten, C. de Laat, A. Penders, A. Iosup, H. Sips. “Can Portability Improve Performance?” In: *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering - ICPE ’15*. ACM Press, 2015. DOI: 10.1145/2668930.2688042 (cit. on p. 23).

[WN15] Y. Wang, T. Nicol. “On statistical distance based testing of pseudo random sequences and experiments with PHP and Debian OpenSSL”. In: *Computers & Security* 53 (Sept. 2015), pp. 44–64. DOI: 10.1016/j.cose.2015.05.005 (cit. on pp. 37, 39).

[WRO+17] Y. Wang, A. T. Riffel, J. D. Owens, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu. “Gunrock”. In: *ACM Transactions on Parallel Computing* 4.1 (Aug. 2017), pp. 1–49. DOI: 10.1145/3108140 (cit. on p. 15).

[ZTS03] K. H. Zou, K. Tuncali, S. G. Silverman. “Correlation and Simple Linear Regression”. In: *Radiology* 227.3 (June 2003), pp. 617–628. DOI: 10.1148/radiol.2273011499 (cit. on p. 23).

All links were last followed on August 25, 2020.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature