BSc Thesis Applied Mathematics

Estimating graph properties with HyperLogLog-type algorithms

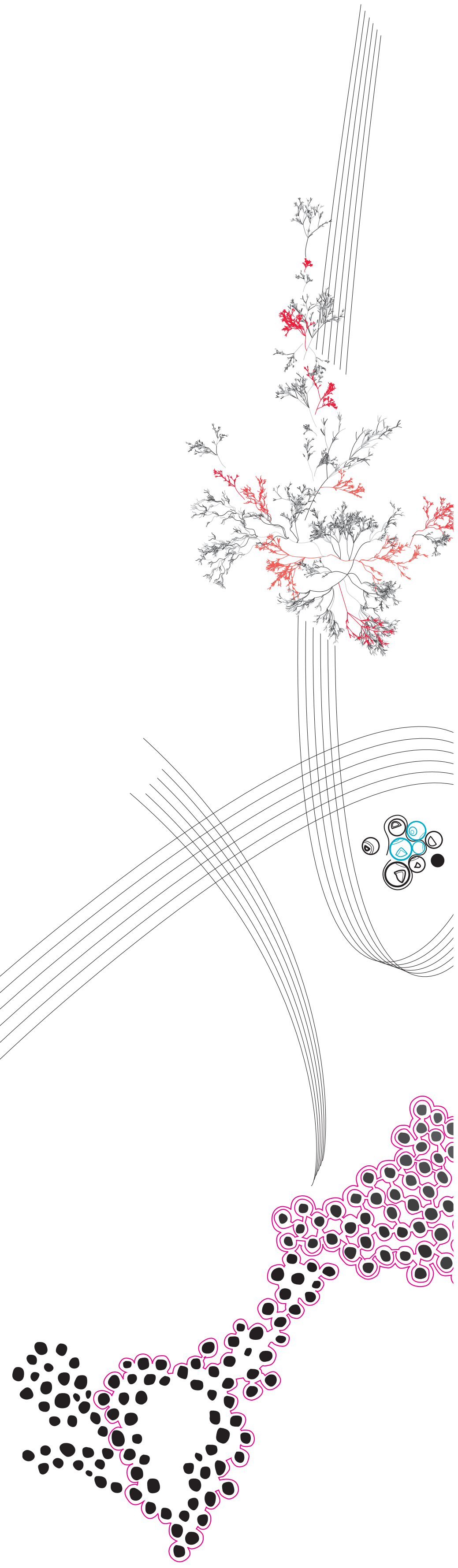
J.J. Huizinga

Supervisor: prof.dr. N.V. Litvak

June, 2019

Department of Applied Mathematics
Faculty of Electrical Engineering,
Mathematics and Computer Science

UNIVERSITY OF TWENTE.



Estimating graph properties with HyperLogLog-type algorithms

J.J. Huizinga*

June, 2019

Abstract

When one is dealing with algorithms in very large networks, the main memory often becomes too small. Therefore, other techniques are necessary to analyse these graphs. *Boldi and Vigna* showed that vertex ball cardinality can be estimated well using *HyperBall* [4], which is based on *HyperLogLog* [11] probabilistic counters. In this paper *HyperEdgeball*, similar to *HyperBall*, is created to estimate edgeball cardinality. Both algorithms are used in *HyperSurplus* to estimate the number of surplus edges locally, which gives good first practical results. Theory is deduced to count triangles and 4-cycles in a similar fashion, this yielded not yet in a properly performing algorithm.

Keywords: count, estimation, HyperBall, HyperEdgeball, HyperLogLog, HyperSurplus, graph theory, large networks, probabilistic counting, surplus edge, triangle, 4-cycle

1 Introduction

Knowledge of graph properties is used in all kinds of practical applications. For instance, *Bechetti et al.* [2] use the local triangle count to detect web spam. Algorithms that determine the number of triangles in a set exactly exist. However, when dealing with very large networks, the main memory of a computer becomes too small to run these kinds of algorithms. When one still wants a triangle count, one has to settle for an estimate. But then, when dealing with billions of vertices, one still has to severely limit the amount of memory used per vertex, that is in the order of several bytes per vertex. Also, in practice it is often impossible to read data at will. Therefore we rather assume that the graph can only be accessed in a semi-streaming fashion: adjacency lists of each vertex are scanned one after another.

Similar requirements occurred while counting distinct values in very large data sets, probabilistic techniques have been used to estimate the cardinality of such sets. An extremely efficient algorithm that performs this task of estimation is the *HyperLogLog* algorithm [11], a probabilistic counter with goods results in practice. This algorithm is later extended by *Boldi, Rosa and Vigna* [3] to the *HyperANF* algorithm, which approximates the neighbourhood function. Subsequently *Boldi and Vigna* extended this to *HyperBall* [4], a framework for computations that depend on at most or exactly distance t from a vertex.

In this paper we will first cover the theory basics on *HyperLogLog* counters and the *HyperBall* algorithm. Thereafter we will introduce an algorithm similar to *HyperBall*:

*E-mail: j.j.huizinga-1@student.utwente.nl

HyperEdgeball. *HyperEdgeball* estimates the cardinality of edgeballs, which are introduced in this paper. Edgeballs turn out to be of great help when estimating graph properties locally.

We introduce these algorithms to lay a foundation in probabilistic counting algorithms for graph usage. After this we study the estimation of several graph properties using *Hyper-LogLog*-type algorithms: surplus edges locally, triangles and 4-cycles. Exact estimation of these properties might be useful in several very large network applications. First practical results also have been performed and are discussed.

1.1 Definitions and notation

Let V be a set of n vertices and E be a set of m edges, which together form the graph $G = (V, E)$. Each vertex is associated with a fixed unique identifier, a number between 1 and n. We use vw to indicate that there is an edge from vertex v to vertex w. In undirected graphs $vw \in E$ implicates $wv \in E$.

The *distance*, $d(x, y)$, between vertices x and y is the length of the shortest path between those vertices. If no path exists, then $d(x, y) = \infty$. A vertex y is *reachable* from vertex x if there exists a path from x to y, i.e. $d(x, y) < \infty$.

The *ball of radius* r around a vertex x is the set $\mathcal{B}_r(x) = \{y | d(x, y) \leq r\}$, we will also refer to this as the r-ball of x. The r-neighbourhood of a vertex x is the set $\mathcal{N}_r(x) = \mathcal{B}_r(x) - \mathcal{B}_{r-1}(x)$, r being a positive integer. It is the set of vertices at exactly distance r from vertex x. Notice that $\mathcal{N}_1(x)$ contains the neighbourhood of x, therefore the notation $\mathcal{N}(x)$ is also used to indicate this set.

The following definition is not commonly used, but turns out to be of great importance from section 3 onward. We define $E_r(x)$, the r-edgeball of x, to be the set of edges that appear in at least one path of length r starting at $x \in V$. Logically, all vertices in $\mathcal{B}_r(x)$ are reachable via a path created with edges from $E_r(x)$.

2 Theory and relevant concepts

2.1 HyperLogLog counters

HyperLogLog [11] is a probabilistic counting technique to estimate the *number of distinct elements*, also know as *cardinality*, of very large dataset $\mathcal{Q} \subseteq \mathcal{D}$, where $\mathcal{D}$ is a domain. Normally the cardinality of a set can be determined exactly, but when dealing with large data, the dataset is often too large to fit in the main memory [1, 11]. Therefore it is in practice impossible to determine the cardinality exactly, and we have to settle for an estimation of the cardinality. For the same reason, the data often appears as a *stream*: a read-once sequence with arbitrary order.

A *HyperLogLog* counter relies on a *hash function*, $h : \mathcal{D} \to \{0,1\}^\infty$, that processes an element of the dataset $\mathcal{Q} \subseteq \mathcal{D}$ to a *hashed value*, a binary string of infinite length. Important is that the hash function is designed in such a way that the hashed values closely resemble a uniform model of randomness, i.e. bits are independent of each other and the probability of 0 or 1 is $\frac{1}{2}$. Knowing this we can use the principle of *bit pattern observables*, the observation of certain bit patterns at the beginning of the hashed value. Specifically, the *HyperLogLog* counter looks at the number of leading zeroes, while going through the data it saves the maximum number of the position of the first 1. Let the position of the leftmost 1 in a binary string x be given by $\rho(x)$. Let us now look at a binary string x that starts with the pattern 0^k1, so $\rho(x) = k+1$. The probability that x starts with this pattern is $(\frac{1}{2})^k \cdot \frac{1}{2} = (\frac{1}{2})^{k+1} = (\frac{1}{2})^{\rho(x)}$. This is an indication that the cardinality of the set is approximately $2^{\rho(x)}$.

It might happen that the *HyperLogLog* counter comes across a hashed value that has way too many leading zeros in comparison to the cardinality of the set, in this case the cardinality is severely overestimated. To prevent this *Flajolet and Martin* [12] introduced the idea of *stochastic averaging*, dividing the stream to get multiple estimators, in an earlier version of the *HyperLogLog* algorithm. This idea is also used in another predecessor by *Durand and Flajolet* [9]. The stream is divided into $p = 2^b$ *substreams.* For each substream a *register* in the counter is instated. Then the first b bits determine in which register the

Algorithm 1 The *HyperLogLog* counter [11], this is the algorithm as described in [4], approximates the number of distinct values in a data stream.

```
Require: $h : \mathcal{D} \to \{0,1\}^\infty$, a hash function from the domain of items
 1: $M[-]$, the counter, an array of $p = 2^b$ registers (indexed from 0) and set to $-\infty$
 2: function ADD($M$: counter, $x$: item)
 3:     $i \leftarrow h_b(x)$
 4:     $M[i] \leftarrow \max\{M[i], \rho(h^b(x))\}$
 5: end function
 6: function SIZE($M$: counter)
 7:     $Z \leftarrow \left(\sum_{j=0}^{p-1} 2^{-M[j]}\right)^{-1}$
 8:     $E \leftarrow \alpha_p p^2 Z$
 9:     return E
10: end function
11: for each item $x$ seen in the stream do
12:     ADD($M, x$)
13: end for
14: print SIZE(M)
```

hashed value is placed, the other part of the hashed value is used to, as before, determine the position of the leftmost 1. For future notation we use $h_b(x)$ to indicate the first b bits of the hashed value of an element x, and $h^b(x)$ to indicate the other part of the hashed value. By increasing the number of registers better estimations are obtained, but also more memory is needed. Both should be considered when determining the number of registers used.

In predecessors of the algorithm the geometric mean was used. It was noticed that the distribution of the values in the register was skewed to the right. For that reason the *HyperLogLog* algorithm uses the harmonic mean, it has a tendency to prevent this and to reduce the variance. The idea of using the harmonic mean instead originated from *Chassaing and Gérin*[6].

To summarize the *HyperLogLog* counter, Algorithm 1: the input is a data set $\mathcal{Q}$, the output is an estimation of the cardinality of $\mathcal{Q}$, i.e. $|\hat{\mathcal{Q}}|$. The elements of this set arrive as a stream, a read-once sequence and in arbitrary order. A predetermined number of p registers is created, where $p = 2^b$. The registers are indexed from 0 to $p-1$, the value of register i is $M[i]$. The register to place the value in is determined using the fist b bits of the hashed value of an element x. The position of the leftmost 1 of the remaining of the hashed value is determined and saved in the register if it is larger then the number in the register. When all elements passed, the algorithm computes the indicator

$$Z := \big(\sum_{j=0}^{p-1} 2^{-M[j]}\big)^{-1} .$$

After that, Algorithm 1 computes a normalized version of the harmonic mean of $2^{M[j]}$

$$E := \alpha_p p^2 Z, \text{ where } \alpha_p := \Big(m \int_0^\infty \Big(\log_2 \Big(\frac{2+u}{1+u}\Big)\Big)^m du\Big)^{-1} ,$$

which is the estimator $|\hat{\mathcal{Q}}|$ of the cardinality of the data set $\mathcal{Q}$.

A powerful feature of the *HyperLogLog* counter is that the cardinality of the union of several sets is easily determined. To clarify this, we first look at one substream. The *HyperLogLog* counter fed with the whole union would have a certain value in the register linked to this substream, this is the highest position of the leftmost 1 that was encountered. One or multiple elements in the substream have the leftmost 1 at this position. These elements would also appear in the same substream of at least one of the sets that make up the union, each counted with an individual *HyperLogLog* counter. Maximizing over the registers linked to this substream would therefore result in the same register value for this register. So to take the union of multiple sets, each of them counted with an individual *HyperLogLog* counter with the same number of registers, we combine the registers by register number and maximize over each register. Intersections cannot be computed in such a nice way, they have to be computed using the inclusion-exclusion principle [7].

In this paper we touched the main ideas of the *HyperLogLog* counter. A more extensive explanation on some of these parts can be found in the original paper by *Flajolet et al.* [11]. The original paper also proved certain properties of the error bounds of the *HyperLogLog* counter. Furthermore, they introduced small and large range corrections, which improve the algorithm. Other improvements have been made, e.g. *Heule et al.* presented a series of improvements when using the *HyperLogLog* counter in practice [13]. These improvements are not further discussed, because they fall outside the scope of this paper.

2.2 HyperBall

Let $G = (V, E)$ be a graph, not necessarily undirected or even simple. The *HyperBall* algorithm [4] is a general algorithm that can be used when one wants an approximation of $|\mathcal{B}_r(x)|$, the cardinality of a ball around vertex x with radius r. We let $\hat{\mathcal{B}}_r(x)$ denote this estimation. *Boldi and Vigna* based it on their earlier HyperANF (approximated neighbourhood function) algorithm [3]. It makes uses of the following two properties of balls:

$$\begin{aligned}\mathcal{B}_0(v) &= \{v\}\\ \mathcal{B}_{r+1}(v) &= \bigcup_{v \to w} \mathcal{B}_r(w) \cup \{v\}\ .\end{aligned}$$

Each vertex is associated with one *HyperLogLog* counter. To determine the number of registers for these counters, again a consideration should be made between memory usage and accuracy. Using the properties above, each counter gets updated with each pass of the *HyperBall* algorithm, Algorithm 2. After the rth iteration of the algorithm $\hat{\mathcal{B}}_r(x)$ is known for each vertex $x \in V$. From these estimations, an estimation for the cardinality of the r-neighbourhood of x is easily deduced: $\mathcal{N}_r(x) = \hat{\mathcal{B}}_r(x) - \hat{\mathcal{B}}_{r-1}(x)$.

Important is to note that again *HyperLogLog* counters are used here to severely limit the main memory necessary to run the algorithm.

Algorithm 2 The *HyperBall* algorithm as described in [4], used to estimate ball cardinality for each vertex. Functions from Algorithm 1 are also used.

```
c[−], an array of n HyperLogLog counters

function UNION($M$: counter, $N$: counter)
    for each $i < p$ do
        $M[i] \leftarrow \max\{M[i], N[i]\}$
    end for
end function

for each $v \in V$ do
    ADD($c[v], v$)
end for

$t \leftarrow 0$
repeat
    for each $v \in V$ do
        $a \leftarrow c[v]$
        for each $w \in \mathcal{N}(v)$ do
            $a \leftarrow$ UNION($c[w], a$)
        end for
        write $\langle v, a \rangle$ to disk, which estimates $\mathcal{B}_{t+1}(v)$
    end for
    update the array $c[-]$ with the new $\langle v, a \rangle$ pairs
    $t \leftarrow t + 1$
until no counter changes its value
```

2.3 HyperEdgeball

The estimation of the cardinality of edgeballs turns out to be useful in algorithms which estimate the number of surplus edges and the number of triangles and 4-cycles, this is further discussed in later sections. Using the same idea, we can modify the *HyperBall* algorithm to an algorithm that estimates the number of edges in the k-edgeball. The proposed algorithm is Algorithm 3: *HyperEdgeball.* We make use of the fact that any path of length $r+1$, starting in $v \in V$, can be created by moving to a neighbour of v and then attaching a path of length r:

$$\begin{aligned} E_1(v) &= \{vw \in E | w \in \mathcal{N}(v)\} \\ E_{r+1}(v) &= \bigcup_{v \to w} E_r(w) \cup \{vw \in E | w \in \mathcal{N}(v)\} \ . \end{aligned}$$

HyperEdgeball gives an approximation of $|E_r(v)|$, we denote this approximation by $\hat{E}_r(v)$. Just as *HyperBall*, it can be used in directed graphs. It has a great many other similarities to the *HyperBall* algorithm. However, since we start with $E_1(v)$ for every $v \in V$, $\hat{E}_r(v)$ is known after iteration $r-1$.

Algorithm 3 The *HyperEdgeball* algorithm, used to estimate edgeball cardinality for each vertex. Functions from Algorithm 1 and Algorithm 2 are also used.

```
1: $c[-]$, an array of n HyperLogLog counters

2: for each $v \in V$ do
3:     for each $e \in \{vw \in E | w \in \mathcal{N}(v)\}$ do
4:         ADD$(c[v], e)$
5:     end for
6:     write $\langle v, c[v] \rangle$ to disk, which estimates $E_1(v)$
7: end for

8: $t \leftarrow 0$
9: repeat
10:     for each $v \in V$ do
11:         $a \leftarrow c[v]$
12:         for each $w \in \mathcal{N}(v)$ do
13:             $a \leftarrow$ UNION$(c[w], a)$
14:         end for
15:         write $\langle v, a \rangle$ to disk, which estimates $E_{t+2}(v)$
16:     end for
17:     update the array $c[-]$ with the new $\langle v, a \rangle$ pairs
18:     $t \leftarrow t + 1$
19: until no counter changes its value
```

3 Surplus edges within balls

We will limit ourselves to simple graphs from now on. We can now use $\hat{\mathcal{B}}_r(v)$ and $\hat{E}_r(v)$ to estimate other graph properties. Let $H = \big(\mathcal{B}_r(v), E_r(v)\big)$ be a subgraph of G for some positive integer r. In this section we will show that the number of surplus edges within H can be approximated well using *HyperBall* and *HyperEdgeball*.

Surplus edges are the edges that are not used in a spanning tree of a graph, it is some measurement of the deviation of the graph having a tree-like structure. When analyzing random graphs, the number of surplus edges in H is of interest. For instance, *Dhara et al.* [8] show that there is a connection between the number of surplus edges and component sizes. It is also used to see up to which k-neighbourhood the graph is locally tree-like.

3.1 Counting surplus edges

With the following theorem we first deduce a relationship between the number of surplus edges within a ball and ball and edgeball cardinality.

Theorem 3.1. *Let G be a simple graph. For some $v \in V$ we have $\mathcal{B}_r(v) \subseteq V$, $E_r(v) \subseteq E$ and $H = \big(\mathcal{B}_r(v), E_r(v)\big) \subseteq G$. We denote the number of surplus edges in H by $\mathcal{S}_r(v)$. We have that*

$$\mathcal{S}_r(v) = |E_r(v)| - |\mathcal{B}_r(v)| + 1 \ .$$

Proof. It is well known that for every tree, the number of edges is the number of nodes minus one [5]. Therefore the number of edges needed in any spanning tree of H is $|\mathcal{B}_r(v)| - 1$. By definition, the number of surplus edges is the number of edges that is not in a spanning tree:

$$\begin{aligned}\mathcal{S}_r(v) &= |E_r(v)| - (|\mathcal{B}_r(v)| - 1)\\ &= |E_r(v)| - |\mathcal{B}_r(v)| + 1 \ .\end{aligned}$$

□

3.2 Estimating the number of surplus edges

From the theorem above and sections 2.2 and 2.3 it follows that $\hat{\mathcal{S}}_r(v)$, an estimation of $\mathcal{S}_r(v)$, is

$$\hat{\mathcal{S}}_r(v) = \hat{E}_r(v) - \hat{\mathcal{B}}_r(v) + 1 \ .$$

The *HyperSurplus* algorithm, Algorithm 4, easily follows. This algorithm is of interest when one wants to estimate the number of surplus edges locally. When one wants to estimate the number of surplus edges in the whole graph an easier solution will suffice: *HyperBall* and *HyperEdgeball* can be omitted, the total number of vertices and edges can be estimated directly by *HyperLogLog*. When no list of vertices or edges is available, a *HyperBall*-like or *HyperEdgeball*-like algorithm for only one vertex might be used to accomplish this.

Algorithm 4 The *HyperSurplus* algorithm, used to estimate $\mathcal{S}_r(v)$ for each vertex $v \in V$, $r = 1, ..., k$. It makes use of *HyperBall*, Algorithm 2, and *HyperEdgeball*, Algorithm 3

```
 1: run $k$ iterations of HyperBall
 2: run $k-1$ iterations of HyperEdgeball

 3: $t \leftarrow 1$
 4: repeat
 5:     for each $v \in V$ do
 6:         $a \leftarrow \hat{E}_t(v) - \hat{\mathcal{B}}_t(v) + 1$
 7:         write $\langle v, a \rangle$ to disk, which estimates $\mathcal{S}_t(v)$
 8:     end for
 9:     $t \leftarrow t + 1$
10: until $t$ is equal to $k$
```

3.3 HyperSurplus in practice

We compared the *HyperSurplus* algorithm, with 512 registers for each counter, to an exact surplus edge counting algorithm. The graphs used are Erdös-Rényi random graphs [10]. Three different configurations of n and p have been used: $n = 1000, p = 0.01$; $n = 500, p = 0.02$; $n = 100, p = 0.1$. The graphs considered are fairly small, this is because the exact surplus edge counting algorithm, which is needed for the comparison, is computationally demanding.

For each configuration, Figure 1 shows how the number of surplus edges evolves within each k-neighbourhood, $k = 1, ..., 5$, according to the *HyperSurplus* algorithm and the exact surplus edge counting algorithm. It can be seen that *HyperSurplus* is overestimating the exact answer, but each estimation seems to be good.

To further investigate the error, we analysed 100 Erdös-Rényi random graphs for each configuration. For each of these graphs the relative error was averaged over the set of vertices for each k-neighbourhood, $k = 2, 3, 4, 5$. The relative error of the 1-neighbourhood is not included, it could not be determined since the exact answer is always zero for simple graphs. The results can be found in the histograms in Figure 2. For all these neighbourhoods the estimation is in the same order of magnitude as the exact number, just as expected. What stands out is that the *HyperSurplus* algorithm is overestimating most of the time, this might be due to the limited number of vertices used in each graph. All in all these first practical results look very promising for the estimation of surplus edges locally.

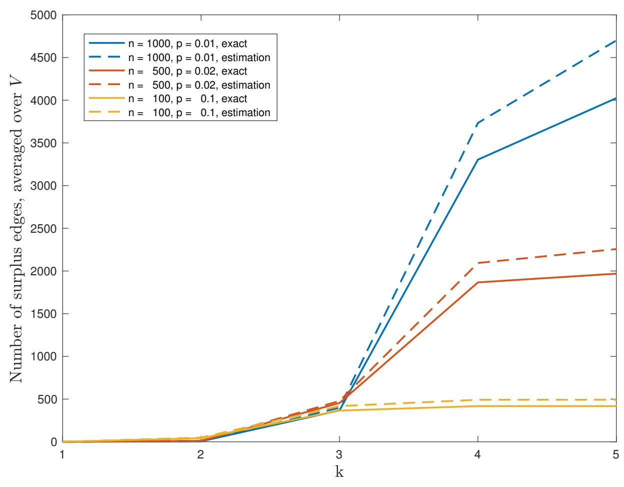


FIGURE 1: The number of surplus edges, averaged over the set of vertices, within each k-neighbourhood, $k = 1, ..., 5$, according to the *HyperSurplus* estimation algorithm and the exact surplus edge counting algorithm. Executed for three Erdös-Rényi random graphs: $n = 1000, p = 0.01$; $n = 500, p = 0.02$; $n = 100, p = 0.1$.

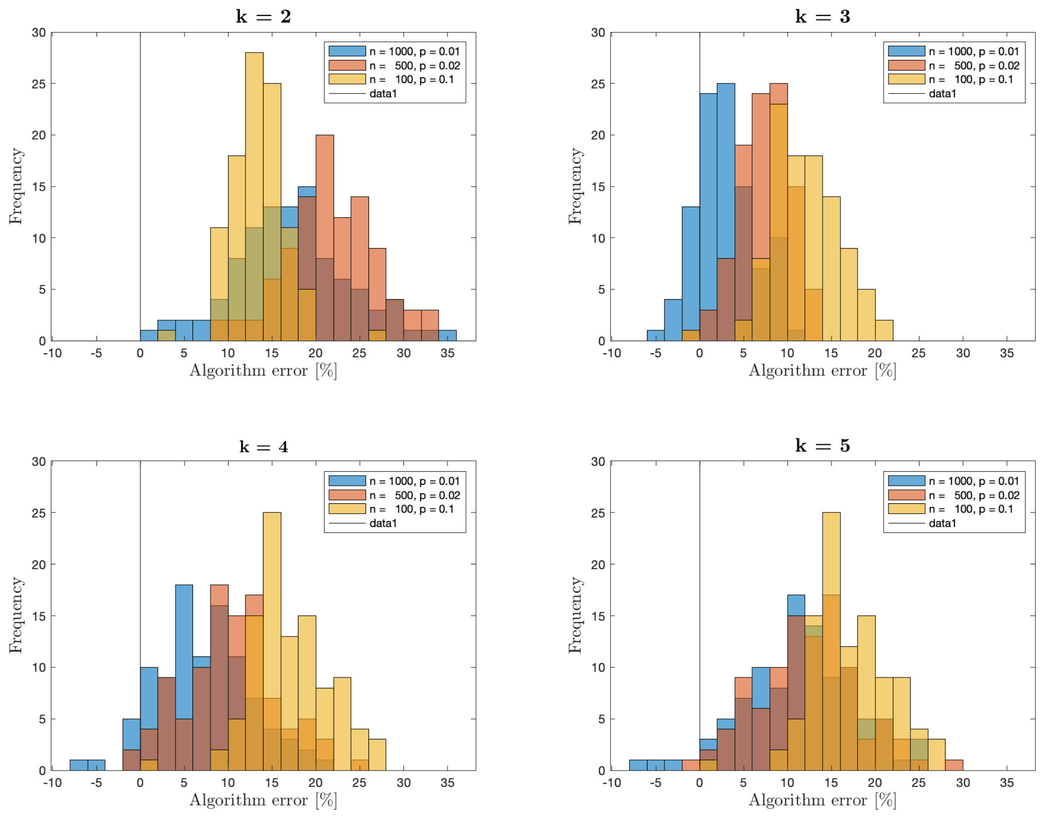


FIGURE 2: The relative error of *Hypersurplus*, with 512 registers for each *HyperLogLog* counter, to an exact surplus edge counting algorithm for each k-neighbourhood, $k = 2, 3, 4, 5$. Executed for three cases of each 100 Erdös-Rényi random graphs: $n = 1000, p = 0.01$; $n = 500, p = 0.02$; $n = 100, p = 0.1$.

4 Triangles and 4-cycles

Without surplus edges a graph is a tree and therefore has no cycles [5]. This also means that a graph without surplus edges has no triangles. So if a graph has a triangle, it should also have a surplus edge. So there is some link between the number of surplus edges and the number of triangles. However, a surplus edge might close more than one triangle. For instance, the complete graph K_4 has four triangles, but only three surplus edges. It might also be that a surplus edge does not close a triangle, but another cycle.

The counting of triangles has many real life applications. For instance, *Becchetti et al.* [2] use the local triangle count to detect web spam.

In this section, we study the estimation of the number of triangles and 4-cycles in a graph. We start with introducing theory on counting these numbers exactly using $\mathcal{B}_r(v)$ and $E_r(v)$, since those can be estimated well by *HyperBall* and *HyperEdgeball*.

4.1 Counting triangles

Lemma 4.1. *Let G be a simple graph. For every $v \in V$, the number of edges with one end in $\mathcal{N}(v)$ and the other end in $\mathcal{N}_2(v)$ is*

$$\sum_{y \in \mathcal{N}_2(v)} |\mathcal{N}(v) \cap \mathcal{N}(y)| \,.$$

Proof. Let y be a vertex from the 2-neighbourhood of v. $\mathcal{N}(v)$ is the set of vertices adjacent to v, $\mathcal{N}(y)$ is the set of vertices adjacent to y. The set $\mathcal{N}(v) \cap \mathcal{N}(y)$ is the set of vertices that are adjacent to v as well as y. Therefore, there are $|\mathcal{N}(v) \cap \mathcal{N}(y)|$ vertices that are adjacent to v as well as y. Since G is simple and since the vertices in $\mathcal{N}(v) \cap \mathcal{N}(y)$ are also in $\mathcal{N}(y)$, each of the vertices in $\mathcal{N}(v) \cap \mathcal{N}(y)$ is joined to y by exactly one edge. It follows that $|\mathcal{N}(v) \cap \mathcal{N}(y)|$ edges have y being one end and have the other end in $\mathcal{N}(v)$. Figure 3 might help to visualize the above part of the proof.

To get the total number of edges with one end in $\mathcal{N}(v)$ and the other end in $\mathcal{N}_2(v)$, we sum for all $y \in N_2(v)$. □

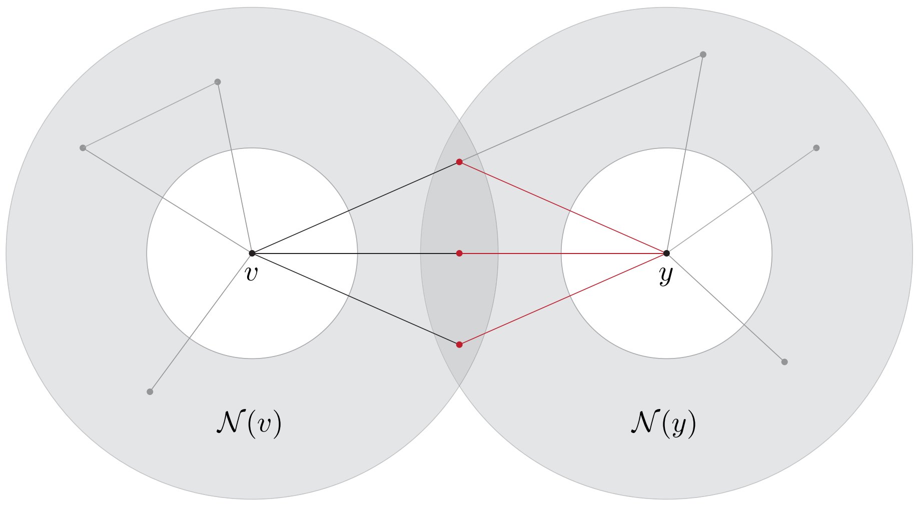


FIGURE 3: Vertices v and $y \in \mathcal{N}_2(v)$. The red vertices are in the set $\mathcal{N}(v) \cap \mathcal{N}(y)$. The red edges have y at one end and have the other end in $\mathcal{N}(v)$.

Theorem 4.1. *Let G be a simple graph. The number of distinct triangles including $v \in V$, $\triangle_v$, is*

$$\triangle_v = \frac{1}{2}\Big(\sum_{x\in\mathcal{N}(v)} |\mathcal{N}(x)| - |N(v)| - \sum_{y\in\mathcal{N}_2(v)} |\mathcal{N}(v) \cap \mathcal{N}(y)|\Big) .$$

Proof. Figure 4 might help to visualize the proof. Since G is simple, each neighbour of v is joined to v by exactly one edge. It follows that the total number of edges joining v to its neighbours is $|\mathcal{N}(v)|$. By lemma 4.1 the number of edges with one end in $\mathcal{N}(v)$ and the other end in $\mathcal{N}_2(v)$ is $\sum_{y\in\mathcal{N}_2(v)} |\mathcal{N}(v) \cap \mathcal{N}(y)|$.

Let x be a neighbour of v, it is incident with $|\mathcal{N}(x)|$ edges. The other ends of edges incident with x are incident with v, another neighbour of v or a 2-neighbour of v. We determine for each neighbour of v the number of edges it is incident with and sum these. If we thereafter subtract the number of edges that are incident with v or have one end in $\mathcal{N}_2(v)$, we are left with the number of edges joining neighbours of v. Each of these edges form, together with the edges going from the ends of this edge to v, a triangle including v. Since each of these edges is counted twice, once from each end, the number is divided by two to get the number of distinct triangles including v. ☐

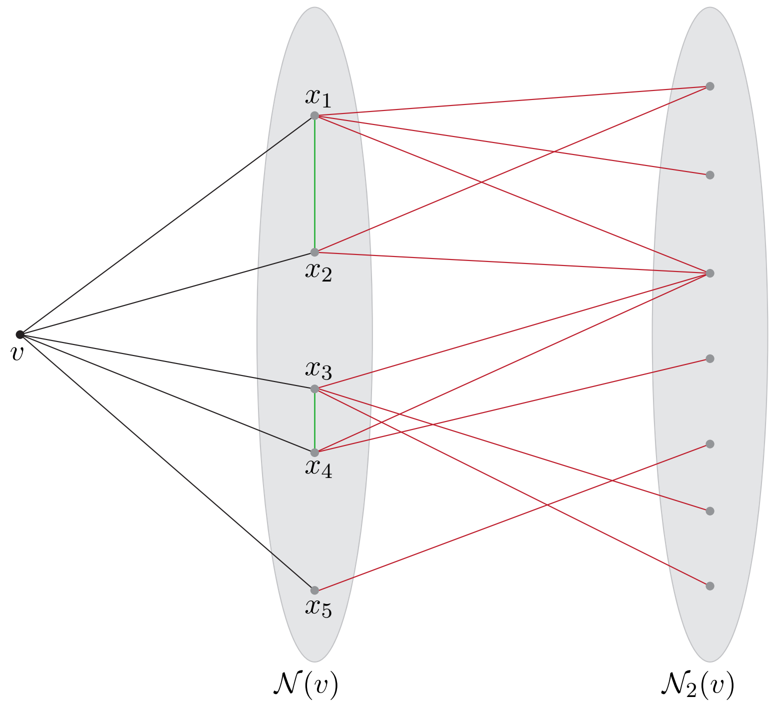


FIGURE 4: All edges depicted have at least one end in $\mathcal{N}(v)$. The black edges are edges that are joining v to its neighbours, there are $|\mathcal{N}(v)|$ of them. The red edges are the edges with one end in $\mathcal{N}(v)$ and the other end in $\mathcal{N}_2(v)$, by Lemma 4.1 there are $\sum_{y\in\mathcal{N}_2(v)} |\mathcal{N}(v) \cap \mathcal{N}(y)|$ of them. The green edges join two neighbours of v. A green edges forms together with two black edges a triangle including v. Notice that in $\sum_{x\in\mathcal{N}(v)} |\mathcal{N}(x)|$, each green edge is counted twice.

Corollary 4.1. *Let G be a simple graph. The total number of distinct triangles, $\triangle$, is*

$$\triangle = \frac{1}{6}\sum_{v\in V}\sum_{x\in\mathcal{N}(v)}|\mathcal{N}(x)| - \sum_{v\in V}|N(v)| - \sum_{v\in V}\sum_{y\in\mathcal{N}_2(v)}|\mathcal{N}(v)\cap\mathcal{N}(y)| \,.$$

Proof. If we sum up the equation of Theorem 4.1 for all vertices, each triangle is counted once from each vertex included in the triangle. Therefore the total number of triangles will be counted three times, the result directly follows:

$$\begin{aligned}\triangle &= \frac{1}{3}\sum_{v\in V}\triangle_v\\ &= \frac{1}{6}\sum_{v\in V}\Big(\sum_{x\in\mathcal{N}(v)}|\mathcal{N}(x)| - |N(v)| - \sum_{y\in\mathcal{N}_2(v)}|\mathcal{N}(v)\cap\mathcal{N}(y)|\Big)\\ &= \frac{1}{6}\sum_{v\in V}\sum_{x\in\mathcal{N}(v)}|\mathcal{N}(x)| - \sum_{v\in V}|N(v)| - \sum_{v\in V}\sum_{y\in\mathcal{N}_2(v)}|\mathcal{N}(v)\cap\mathcal{N}(y)| \,.\end{aligned}$$

□

As stated in section 2, the $|\mathcal{N}(x)|$ and $|\mathcal{N}(v)|$ terms can be estimated nicely by *HyperBall*. However, *HyperLogLog*, and therefore *HyperBall*, cannot give a good approximation of the $|\mathcal{N}(v)\cap\mathcal{N}(y)|$ term. We can therefore not give a good approximation of the distinct number of triangles using Corollary 4.1. We continue our search into the estimation of graph properties by looking at both triangles and 4-cycles.

4.2 Counting triangles and 4-cycles

We define the following numbers for each $v \in V$:

$$\begin{aligned}\mathcal{E}_{/}(v) &:= |\mathcal{N}(v)|\\ \mathcal{E}_{\triangle}(v) &:= \frac{1}{2}\Big(\sum_{x\in\mathcal{N}(v)}|\mathcal{N}(x)| - |N(v)| - \sum_{y\in\mathcal{N}_2(v)}|\mathcal{N}(v)\cap\mathcal{N}(y)|\Big)\\ \mathcal{E}_{\Diamond}(v) &:= \sum_{y\in\mathcal{N}_2(v)}|\mathcal{N}(v)\cap\mathcal{N}(y)| \,.\end{aligned}$$

As seen in the proof of Theorem 4.1: the number $\mathcal{E}_{/}(v)$ is the number of edges joining v to its neighbours; the number $\mathcal{E}_{\triangle}(v)$ is the number of edges with both ends in $\mathcal{N}(v)$. As seen in Lemma 4.1: the number $\mathcal{E}_{\Diamond}(v)$ is the number of edges with one end in $\mathcal{N}(v)$ and the other end in $\mathcal{N}_2(v)$. A graphic representation of these edges can be seen in Figure 4. Introducing $\mathcal{E}_{/}(v)$, $\mathcal{E}_{\triangle}(v)$ and $\mathcal{E}_{\Diamond}(v)$ results in more clarity in the upcoming theorems.

Theorem 4.2. *Let G be a simple graph. The number of distinct triangles including $v \in V$, $\triangle_v$, is*

$$\triangle_v = \mathcal{E}_{\triangle}(v) \,.$$

Proof. The result follows directly from the definition of $\mathcal{E}_{\triangle}$ and Theorem 4.1. □

Theorem 4.3. *Let G be a simple graph. The following inequalities hold for $\Diamond_v$, the number of distinct 4-cycles including v:*

$$\mathcal{E}_\Diamond(v) - |\mathcal{N}_2(v)| \leq \Diamond_v \leq \frac{\mathcal{E}^2_\Diamond(v) - \mathcal{E}_\Diamond(v)}{2} .$$

Proof. Figure 4 might aid the reader in visualizing parts of the proof.

Let (v, x, y, z, v), $v, x, y, z \in V$, be an arbitrary 4-cycle. We first notice that for every 4-cycle including v, one of the other vertices of the 4-cycle, y in this case, should by definition be in $\mathcal{N}_2(v)$. Furthermore the 4-cycle consists of two different paths between this vertex and v, (v, x, y) and (v, z, y) in this case. From this we can conclude that all 4-cycles that include v can be constructed by taking a vertex from $\mathcal{N}_2(v)$ and taking two different paths of length two between this vertex and v.

We first look at the trivial case of $|\mathcal{N}_2(v)| = 0$. Since there are no vertices in $\mathcal{N}_2(v)$, $\Diamond_v = 0$ by the previous argument. We also have that $\mathcal{E}_\Diamond(v) = 0$, and therefore that $\mathcal{E}_\Diamond(v) - |\mathcal{N}_2(v)| = \frac{\mathcal{E}^2_\Diamond(v) - \mathcal{E}_\Diamond(v)}{2} = 0$. Clearly the bounds hold for this case.

We now look at the nontrivial case $|\mathcal{N}_2(v)| \geq 1$. At the beginning of this proof we have shown that, to count the number of 4-cycles including v, the number of different paths of length two between $\mathcal{N}_2(v)$ and v is important. We therefore first have a look at how many of these paths there are. Let us look at some edge yq, with $y \in \mathcal{N}_2(v), q \in \mathcal{N}(v)$. Since G is simple, there is only one edge qv. These edges together form the path (y, q, v) of length two between $\mathcal{N}_2(v)$ and v. Thus for each edge with one end in $\mathcal{N}_2(v)$ and the other end in $\mathcal{N}(v)$, there is exactly one unique path between $\mathcal{N}_2(v)$ and v. We have already seen that there are $\mathcal{E}_\Diamond(v)$ of these edges, it follows that the number of paths of length two between $\mathcal{N}_2(v)$ and v is therefore also exactly $\mathcal{E}_\Diamond(v)$.

We first prove the lower bound of $\Diamond_v$. Each vertex in $\mathcal{N}_2(v)$ is connected to v by at least one path of length two. The remaining $\mathcal{E}_\Diamond(v) - |\mathcal{N}_2(v)|$ paths of length two between v and $\mathcal{N}_2(v)$ are therefore part of at least one 4-cycle, which gives us the lower bound: $\mathcal{E}_\Diamond(v) - |\mathcal{N}_2(v)| \leq \Diamond_v$.

We now have a look at the upper bound. Suppose that $|N_2(v)| = 1$, then $N_2(v) = \{y\}$. Vertex y is connected to v by $\mathcal{E}_\Diamond(v)$ paths of length two. It follows that there are $\binom{\mathcal{E}_\Diamond(v)}{2} = \frac{\mathcal{E}_\Diamond(v)(\mathcal{E}_\Diamond(v)-1)}{2} = \frac{\mathcal{E}^2_\Diamond(v)-\mathcal{E}_\Diamond(v)}{2}$ unique pairs of paths of length two, each of them forms a 4-cycle. We show that this is indeed an upper bound by looking at $|N_2(v)| = k > 1$. We have that $N_2(v) = \{y_1, ..., y_k\}$, with $|\mathcal{N}(v) \cap \mathcal{N}(y_i)| = a_i \geq 1$. We have by definition that

$$\mathcal{E}_\Diamond(v) = \sum_{y_i \in \mathcal{N}_2(v)} |\mathcal{N}(v) \cap \mathcal{N}(y_i)| = a_1 + ... + a_k .$$

Now,

$$\begin{aligned}
\Diamond_v &= \binom{a_1}{2} + ... + \binom{a_k}{2} \\
&= \frac{a_1^2 - a_1}{2} + ... + \frac{a_k^2 - a_k}{2} \\
&\leq \frac{a_1^2 - a_1}{2} + ... + \frac{a_k^2 - a_k}{2} + \frac{a_1(a_2 + ... + a_k)}{2} + ... + \frac{a_k(a_1 + ... + a_{k-1})}{2} \\
&= \frac{a_1^2 - a_1}{2} + \frac{a_1(a_2 + ... + a_k)}{2} + ... + \frac{a_k^2 - a_k}{2} + \frac{a_k(a_1 + ... + a_{k-1})}{2} \\
&= \frac{a_1(a_1 + ... + a_k - 1)}{2} + ... + \frac{a_k(a_1 + ... + a_k - 1)}{2} \\
&= \frac{(a_1 + ... + a_k)(a_1 + ... + a_k - 1)}{2} \\
&= \frac{\mathcal{E}_\Diamond(v)(\mathcal{E}_\Diamond(v) - 1)}{2} \\
&= \frac{\mathcal{E}_\Diamond(v)^2 - \mathcal{E}_\Diamond(v)}{2} \; .
\end{aligned}$$

□

It might be clear to the reader that tighter bounds of Theorem 4.3 can be found when the properties of the graph are constrained more. In the following corollary a special case will be shown to illustrate this.

Corollary 4.2. *Let G be a simple graph with at most two distinct paths of length two between each pair of vertices, i.e. $|\mathcal{N}(a) \cap \mathcal{N}(b)| \leq 2$ for all $a, b \in V$. The number of distinct 4-cycles including $v \in V$ is:*

$$\Diamond_v = \mathcal{E}_\Diamond(v) - |\mathcal{N}_2(v)| \; .$$

Proof. For each $y \in \mathcal{N}_2(v)$ there is by definition at least one path of length two. Clearly no 4-cycle which includes v and y exists if there is exactly one path of length two connecting v and y. By hypothesis, there are at most two paths of length two between v and y. If there are exactly two paths of length two, those two paths create together exactly one 4-cycle including v. It follows from these two cases, that the number 4-cycles including v and y is $|\mathcal{N}(v) \cap \mathcal{N}(y)| - 1$. Since for each 4-cycle including v at least one of the other vertices in the 4-cycle is contained in the set $\mathcal{N}_2(v)$, it follows that

$$\begin{aligned}
\Diamond_v &= \sum_{y \in \mathcal{N}_2(v)} \big(|\mathcal{N}(v) \cap \mathcal{N}(y)| - 1\big) \\
&= \sum_{y \in \mathcal{N}_2(v)} |\mathcal{N}(v) \cap \mathcal{N}(y)| - |\mathcal{N}_2(v)| \\
&= \mathcal{E}_\Diamond(v) - |\mathcal{N}_2(v)| \; ,
\end{aligned}$$

which is exactly the lower bound of Theorem 4.3. □

Theorem 4.4. *Let G be a simple graph with at most two distinct paths of length two between each pair of vertices, i.e. $|\mathcal{N}(a) \cap \mathcal{N}(b)| \leq 2$ for all $a, b \in V$. We have that*

$$\triangle_v + \Diamond_v = |E_2(v)| - |\mathcal{B}_2(v)| + 1 \, .$$

Proof. The edge set $E_2(v)$ can be partitioned into three subsets: the edges joining v to its neighbours, the edges with both ends in $\mathcal{N}(v)$ and the edges with one end in $\mathcal{N}(v)$ and one end in $\mathcal{N}_2(v)$. Figure **??** might give some more insight into the previous statement. The cardinality of these subsets are $\mathcal{E}_/(v)$, $\mathcal{E}_\triangle(v)$ and $\mathcal{E}_\Diamond(v)$, as seen at the beginning of section 4.2. Since those tree subsets are disjoint, we have

$$\begin{aligned} |E_2(v)| &= \mathcal{E}_/(v) + \mathcal{E}_\triangle(v) + \mathcal{E}_\Diamond(v) \\ |E_2(v)| - \mathcal{E}_/(v) &= \mathcal{E}_\triangle(v) + \mathcal{E}_\Diamond(v) \\ |E_2(v)| - |\mathcal{N}(v)| &= \mathcal{E}_\triangle(v) + \mathcal{E}_\Diamond(v) \, . \end{aligned} \tag{1}$$

Since G is a simple graph and each pair of vertices is connected by at most two distinct paths of length two, we have by Theorem 4.2 and Corollary 4.2 that

$$\triangle_v + \Diamond_v = \mathcal{E}_\triangle(v) + \mathcal{E}_\Diamond(v) - |\mathcal{N}_2(v)| \, .$$

By equation 1, this is the same as

$$\triangle_v + \Diamond_v = |E_2(v)| - |\mathcal{N}(v)| - |\mathcal{N}_2(v)| \, . \tag{2}$$

By the definition of $\mathcal{N}_r(v)$, as seen in section 2.2, and the fact that $\mathcal{N}_r(v)$ and $\mathcal{B}_{r-1}(v)$ are disjoint sets we have

$$\begin{aligned} |\mathcal{N}_r(v)| + |\mathcal{B}_{r-1}(v)| &= |\mathcal{B}_r(v)| \\ |\mathcal{N}_r(v)| &= |\mathcal{B}_r(v)| - |\mathcal{B}_{r-1}(v)| \end{aligned}$$

By substituting this into equation 2, we conclude the proof

$$\begin{aligned} \triangle_v + \Diamond_v &= |E_2(v)| - \big(|\mathcal{B}_1(v)| - |\mathcal{B}_0(v)|\big) - \big(|\mathcal{B}_2(v)| - |\mathcal{B}_1(v)|\big) \\ &= |E_2(v)| - \big(|\mathcal{B}_1(v)| - 1\big) - \big(|\mathcal{B}_2(v)| - |\mathcal{B}_1(v)|\big) \\ &= |E_2(v)| - |\mathcal{B}_2(v)| + 1 \, . \end{aligned}$$

□

We see that the terms which include an intersection cancel nicely. As stated in section 2, these terms were hard to estimate by *HyperLogLog* and therefore by *HyperBall*. Also, in the same section, we have seen that the terms $|\mathcal{B}_2(v)|$ and $|E_2(v)|$ can be estimated nicely by the *HyperBall* and the *HyperEdgeball* algorithms.

Corollary 4.3. *Let G be a simple graph with at most two distinct paths of length two between each pair of vertices, i.e. $|\mathcal{N}(a) \cap \mathcal{N}(b)| \leq 2$ for all $a, b \in V$. As before $\triangle$ is the total number of distinct triangles in G, furthermore $\Diamond$ is the total number of distinct 4-cycles in G. We have that*

$$3\,\triangle + 4\,\Diamond = \sum |E_2(v)| - \sum |\mathcal{B}_2(v)| + n\ .$$

Proof. If we sum up the equation of Theorem 4.4 for all vertices, each triangle is counted once from each vertex included in the triangle. Therefore the total number of triangles will be counted three times. Each 4-cycle is counted once from each vertex included in the 4-cycle. Therefore the total number of 4-cycles will be counted four times. The result directly follows:

$$\begin{aligned} 3\,\triangle + 4\,\Diamond &= \sum_{v\in V} \big(|E_2(v)| - |\mathcal{B}_2(v)| + 1\big) \\ &= \sum_{v\in V} |E_2(v)| - \sum_{v\in V} |\mathcal{B}_2(v)| + \sum_{v\in V} 1 \\ &= \sum_{v\in V} |E_2(v)| - \sum_{v\in V} |\mathcal{B}_2(v)| + n\ . \end{aligned}$$

□

4.3 Triangle and 4-cycle estimation in practice

As we have seen in a special case, according to Corollary 4.3, we can use *HyperBall* and *HyperEdgeball* to estimate three times the distinct number of triangles plus four times the distinct number of 4-cycles. Even if we have some graph that is alike a graph that meets the requirements of Corollary 4.3, it might still give a good estimation. However, we still cannot extend it to a more general case, mainly due to Theorem 4.3.

It is nevertheless interesting to see how an algorithm based on Corollary 4.3 performs in practice. This algorithm calculates $\sum_{v\in V} \hat{E}_2(v) - \sum_{v\in V} \hat{\mathcal{B}}_2(v) + n$, with 512 registers for each *HyperLogLog* counter, to estimate $3\,\triangle + 4\,\Diamond$. Algorithm 5 and Algorithm 6, included in appendix A, are used to count $\triangle$ and $\Diamond$ exactly. After that, both are compared. For the first test it is executed on 100 Erdös-Rényi random graphs with $n = 1000$ and $p = 0.01$. Results of the first test can be found in Figure 5. When looking at only this case, it looks like it is performing similarly to *HyperSurplus*: most of the time estimating a bit above the real value. However, a comparison with the second and third test in the same figure show that this is coincidental. The second and third test are performed with the following configuration: 100 Erdös-Rényi random graphs for $n = 1000, p = 0.025$ and $n = 1000, p = 0.05$. Here the algorithm is constantly underestimating the real values. This is as expected since the probability on 4-cycles is higher and since an algorithm based on Corollary 4.3 estimates the lower bound of the number of 4-cycles as seen in Theorem 4.3 and Corollary 4.2. We can conclude that, just as expected, an algorithm based on Corollary 4.3 does not perform well.

For future studies, it might be interesting to look for indicators that easily tell if an algorithm based on Corollary 4.3 can give a good estimation.

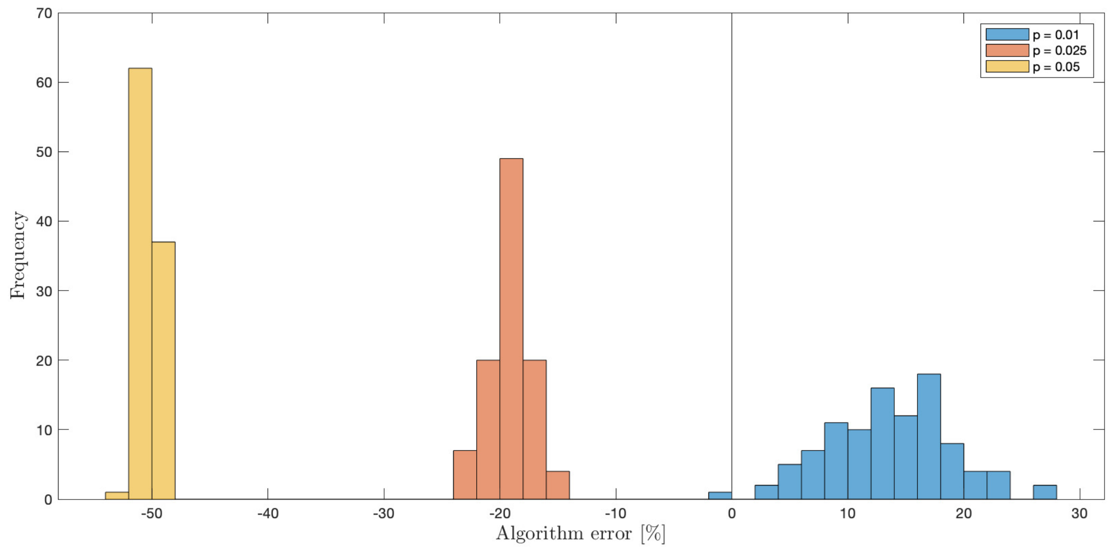


FIGURE 5: The relative error of an algorithm based on Corollary 4.3, with 512 registers for each *HyperLogLog* counter, to and exact counting algorithm for 3 $\triangle$ + 4 $\Diamond$. Three cases were calculated: $p = 0.01$; $p = 0.025$; $p = 0.05$. For each of the cases a 100 Erdös-Rényi random graphs with $n = 1000$ were used.

5 Conclusion and recommendations

The goal was to study and create HyperLogLog-type algorithms to estimate graph properties. To aid this we first introduced edgeballs. We then introduced the *HyperEdgeball* algorithm, an algorithm very similar to *HyperBall*, which can estimate the cardinality of edgeballs well. In future studies, one can look at the theoretical error bounds of *HyperEdgeball*, which already has been done for *HyperLogLog* [11] and *HyperBall* [4].

The introduction of edgeballs was necessary to, in combination with balls, count surplus edges locally. Theory to do this was introduced in section 3. The first practical results of the *HyperSurplus* algorithm that followed turned out well. We recommend that theoretical error bounds on this algorithm should be examined. Furthermore, statistical research on this algorithm should be performed, especially with larger graphs than used in this study. Furthermore it might be interesting to compare results of the algorithm to cases where theoretical results are known. In this way a comparison algorithm, which is computationally heavy, can be omitted.

Using balls and edgeballs for triangle counting turned out to be hard since $E_2(v)$ cannot distinguish between an edge that goes from a neighbour to another neighbour and an edge that goes from a neighbour to the 2-neighbourhood. In Corollary 4.1 we tried to circumvent this, but a term using set intersections appeared, which is hard for *HyperLogLog*-type algorithms to deal with. This term neatly disappeared in the special case of Theorem 4.4. It might be interesting to look at indicators that can tell if a graph is closely like this special case, and thus that some estimator of three times the number of triangles plus four times the number of 4-cycles can be calculated.

In a general case however, the edges counted with $|E_2(v)| - |\mathcal{B}_2(v)| + 1$ close either one triangle or one or more 4-cycles instead of one triangle or one 4-cycle, as in the special case. This turns out to be a major difficulty when one wants to extrapolate an estimator of the number of triangles and 4-cycles from balls and edgeballs. As we can see in this research, this represents a fundamental obstacle when applying *HyperBall*-type algorithms for estimating graph patterns. Possibly, this can be resolved in further research, for example, by allowing to use some more memory in nodes or edges.

References

[1] Mohammad Al Hasan and Vachik S. Dave. Triangle counting in large networks: a review. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 8(2), 2018.

[2] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. page 16, 2008.

[3] Paolo Boldi, Marco Rosa, and Sebastiano Vigna. HyperANF: Approximating the Neighbourhood Function of Very Large Graphs on a Budget. page 625, 2011.

[4] Paolo Boldi and Sebastiano Vigna. In-core computation of geometric centralities with HyperBall: A hundred billion nodes and beyond. In *Proceedings - IEEE 13th International Conference on Data Mining Workshops, ICDMW 2013*, 2013.

[5] John Adrian Bondy, Uppaluri Siva Ramachandra Murty, et al. *Graph theory with applications*. Citeseer, 1976.

[6] Philippe Chassaing and Lucas Gerin. Efficient estimation of the cardinality of large data sets. *Discrete Mathematics and Theoretical Computer Science, DMTCS Proceedings*, 2006.

[7] Matt Curcio. Sketch of the Day : HyperLogLog - Cornerstone of a Big Data Infrastructure, 2012.

[8] Souvik Dhara, Remco van der Hofstad, Johan S.H. Van Leeuwaarden, and Sanchayan Sen. Critical window for the configuration model: Finite third moment degrees. *Electronic Journal of Probability*, 22:1–34, 2017.

[9] Marianne Durand and Philippe Flajolet. Loglog Counting of Large Cardinalities. pages 605–617, 2010.

[10] Paul Erdös and Alfréd Rényi. On random graphs, i. *Publicationes Mathematicae (Debrecen)*, 6:290–297, 1959.

[11] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. *Conference on Analysis of Algorithms AH*, pages 127–146, 2007.

[12] Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

[13] Stefan Heule, Marc Nunkesser, and Alexander Hall. HyperLogLog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. *Proceedings of the 16th International Conference on Extending Database Technology - EDBT '13*, page 683, 2013.

A Exact counting algorithms

A useful insight into the counting algorithms below is the following. Let A be the adjacency matrix of a simple graph G. The number A^2_{ij} is exactly the number of different paths of length two from vertex i to vertex j. The number A^3_{ij} is exactly the number of different paths of length three from vertex i to vertex j.

Algorithm 5 The algorithm used in section 4.3 to count $\triangle$ in a simple graph G.

```
Require: $A$, the adjacency matrix $G$
1: $\triangle \leftarrow \frac{1}{6}\sum_{i=1}^{n} A^3_{ii}$

2: return $\triangle$
```

The number $\sum_{i=1}^{n} A^3_{ii}$ counts each triangle six times: two times, once in each direction, from each vertex included in the triangle. Therefore this number is divided by six to get the number of triangles.

Algorithm 6 The algorithm used in section 4.3 to count $\Diamond$ in a simple graph G.

```
Require: $A$, the adjacency matrix of a graph $G$
1: $\Diamond \leftarrow 0$

2: for $i = 1 : n$ do
3:     for $j = 1 : n$ do
4:         $\Diamond \leftarrow \Diamond + \frac{A^2_{ij}(A^2_{ij}-1)}{2}$
5:     end for
6: end for

7: $\Diamond \leftarrow \frac{1}{4}\Diamond$

8: return $\Diamond$
```

Between each pair of vertices, $i, j \in V$, there are $\frac{A^2_{ij}(A^2_{ij}-1)}{2}$ different pairs of paths of length two. Each of these pairs forms exactly one 4-cycle. We have that each 4-cycle is counted from each vertex included, therefore the answer is divided by four to get the number of 4-cycles.