universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„Algorithm Engineering for Fully Dynamic Subgraph Counting“

verfasst von / submitted by

Leonhard Paul Sidl, BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien, 2022 / Vienna, 2022

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	A 066 910
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Computational Science
Betreut von / Supervisor:	Univ.-Prof. Dr. Monika Henzinger
Mitbetreut von / Co-Supervisor:	Dr. Kathrin Hanauer, B.Sc. M.SC.

Contents

Abstract

In this thesis, we explore means to maintain the number of size four subgraphs in a dynamic graph over time. To achieve this, we start with an empty graph and change the subgraph counts whenever we insert or delete an edge. The number by which we update the count is the number of subgraphs that contain that edge. This principle is used by two algorithms for which the theoretical foundations have already been laid by Hanauer et al. [11, 12] and Eppstein et al. [8].

We implement and test both those algorithms using C++ and contrast the results with the theoretical values. The algorithm by Hanauer et al. has a run time of $\mathcal{O}(m^{\frac{2}{3}})$ for paws, four-cycles and diamonds compared to $\mathcal{O}(h^2)$ for the algorithm by Eppstein et al. The run times also differ for three-paths and triangels, with $\mathcal{O}(m^{\frac{1}{2}})$ and $\mathcal{O}(h)$ respectively. To provide a reference, we compare both algorithms to a static algorithm by Ortmann and Brandes [19], using a diverse set of real-word graphs that include hyperlink networks of Wikipedia sites and protein-protein interactions in yeast.

With this analysis, we show that both dynamic algorithms outperform the static algorithm in settings where the subgraph counts need to be computed regularly over the lifetime of the dynamic graph. The main disadvantage of the dynamic algorithms lies in the irregularity of their run time, with subsequent changes in the graph requiring vastly different computation times. We determined, that the algorithm by Hanauer et al. is more versatile and compact than the algorithm by Eppstein et al., but has a slightly worse practical run time. We also discuss changes to the algorithm by Hanauer et al. that make it possible to improve the run time even further.

Zusammenfassung

In dieser Arbeit vergleichen wir mehrere Methoden, um die Anzahl an Subgraphen in einem dynamischen Graphen im Laufe der Zeit zu zählen. Die verwendeten Algorithmen starten mit einem leeren Graphen und verändern die Zähler der einzelnen Subgraphen, sobald eine Kante eingefügt oder gelöscht wird. Dabei wird jeder Zähler immer um die Anzahl an Subgraphen verändert, die die betreffende Kante enthalten. Wir beschäftigen uns mit zwei Algorithmen, deren theoretischer Hintergrund bereits erforscht wurde und die das oben erwähnte Prinzip anwenden [11, 12, 8].

Wir implementieren diese Algorithmen in C++ und vergleichen deren praktische Laufzeiten mit den theoretischen Werten. Der erste Algorithmus von Hanauer et al. hat eine Laufzeit von $\mathcal{O}(m^{\frac{2}{3}})$, wenn die Strukturen paw, four-cycle und diamond gezählt werden. Im Vergleich dazu benötigt der zweite Algorithmus von Eppstein et al. ein Laufzeit von $\mathcal{O}(h^2)$. Einen Unterschied zwischen den Algorithmen gibt es auch bei den Strukturen three-path und triangle, die Laufzeiten betragen $\mathcal{O}(m^{\frac{1}{2}})$ beziehungsweise $\mathcal{O}(h)$. Um die praktischen Ergebnisse gegenüberstellen zu können, vergleichen wir die dynamischen Algorithmen mit einem statischen von Ortmann und Brandes [19]. Die verwendeten Graphen kommen aus verschiedensten Anwendungsbereichen wie Molekularbiologie, Informatik oder Linguistik.

Wir zeigen mit diesen Experimenten, dass beide dynamischen Algorithmen dem statischen bezüglich der Laufzeit überlegen sind, wenn die Subgraphen in regelmäßigen Abständen gezählt werden. Der größte Nachteil der dynamischen Algorithmen in dieser Situation ist, dass ihre Laufzeit sehr stark von der momentanen Veränderung im Graphen abhängt. Wir konnten zeigen, dass der Algorithmus von Hanauer et al. sowohl vielfacher einsetzbar, als auch leichter zu beschreiben ist. Allerdings hat der Algorithmus von Eppstein et al. in der Praxis eine kürzere Laufzeit. Zusätzlich stellen wir Strategien vor, um die praktische Laufzeit des Algorithmus von Hanauer et al. noch zu verbessern.

1 Introduction

Modern hardware enables the storage and analysis of ever more real-world data. A big portion, such as social- or transportation-networks, can be conveniently stored as graphs. Such representations allow for efficient calculations of various statistics, such as finding connected regions or shortest paths [25].

Another operation that is frequently required is counting the occurrences of small subgraphs like triangles or small cliques. Use cases for such statistics range from analysing communication patterns in large social networks like Facebook or Twitter [3], to applications in cheminformatics. Those typically use smaller but much more dense graphs, often containing additional information such as vertex and edge labels, to detect important functional groups [14].

The more general task of finding an arbitrary subgraph of size k requires $\mathcal{O}(n^k)$ time, when n is the number of vertices in the larger graph, when using exhaustive search. Much more optimized algorithms already exist for more limited queries, such as counting all subgraphs of size three or four [19]. But most modern algorithms are only designed for static graphs, that can't easily take the time dependency of many data sets into account [25]. However, around 65% of practically used graphs are dynamic in nature, with scalability and speed in the top five challenges to overcome. This is especially the case for data created in the ever-changing domain of the internet, such as the connectivity of websites via hyperlinks [1] or relationships between social media accounts [21]. Both the problems of scalability and speed can be overcome by using algorithms specifically designed for dynamic graphs, like the ones implemented in this thesis. The central idea of such algorithms is, that a table of the required statistics (and auxiliary information) is maintained for the whole run time of the algorithm and updated accordingly whenever a change in the graph is observed. This has the benefit of providing the results at each time step without having to restart the calculations every step, as would be necessary for static algorithms. This also allows the program to spread the necessary computation time over a larger time period and avoid the large bursts of calculations that would be needed when using a static algorithm.

The goal of this thesis is to implement and compare two such algorithms ([12] and [8]) with each other and their theoretical run times. Because static algorithms are still regularly used for dynamic graphs, a comparison to the static algorithm by Ortmann and Brandes [19], for both correctness and run time, will also be made.

2 Preliminaries

2.1 Basic Definitions

Let a static graph be defined as a set of vertices V and a set of edges E, where each edge $e \in E$ is defined as a set of two vertices $e = \{u, v\}$ where both u and v are an element of V and $u \neq v$. Let $n = |V|$ and $m = |E|$. Only simple graphs will be considered from here on, meaning each unordered pair of vertices is connected by at most one edge. Two vertices are said to be *adjacent*, if they are connected by an edge. Vertices are denoted with lower-case letters. The *degree* of a vertex $deg(v)$ is the number of edges it is part of. The *density* of a graph is the fraction between the number of edges that exist in the graph and the maximal number of edges that could exist. The *h-index* of a graph, which can be used to categorize vertices as low or high degree, is defined as the maximum number such that the graph contains h vertices of degree at least h [8]. A *path* is defined as a sequence of distinct edges where each neighbouring pair shares a unique vertex. A *cycle* is a path where the first and last vertices are identical.

The main task in this thesis is to count *subgraphs* of a graph $G = (V, E)$. Those are defined as graphs $G' = (V', E')$ where $V' \subseteq V$, $E' \subseteq E$ and all edges in E' connect vertices that are only contained in V'. The subgraph count $c(G, G')$ is then defined as the number of subgraphs that are isomorphic to

G' divided by the count of automorphisms of G'. The *s-count* of a vertex $c(v, G, G')$ is defined as the count $c(G, G')$, where each subgraph contains the vertex v. The subgraphs considered in this thesis are as follows: A *triangle* is defined as a cycle containing three vertices. A *length-three path* is defined as a path containing three edges. A *claw* consists of a central vertex which is adjacent to three other vertices. A *paw* is defined as a triangle where one of the three vertices has an additional edge. A *four-cycle* is a cycle containing four vertices. A *diamond* is a four-cycle with an additional edge connecting two opposite vertices. Finally, a *four-clique* is a set of four vertices, where each pair is connected by an edge. The subgraphs can be seen in Figure 1.

A dynamic graph allows the storage of temporal changes of a graph. Those can be broadly categorized into attributional and topological changes. The former occurs, if the attributes of edges or vertices are changed, but the topology is retained. This will not be considered further in this thesis. Topological changes occur, if edges or vertices get deleted or inserted. Here, the operation of removing a vertex can be split into first removing all edges attached to the respective vertex, followed by the trivial operation of removing the isolated vertex. There are two main methods of storing dynamic graphs. They can either be stored as a sequence of static graphs called *snapshots*, leading to a discrete representation, or as one large graph where each topological change is kept track of with the according time-point. The second method facilitates algorithms that need a stream of topological changes without discontinuities, resulting in a continuous graph [25].

The bounds for run times of static algorithms are always given as worst case run times for the analysis of a single graph. The described dynamic algorithms on the other hand use an *amortized* run time for each operation done to the graph. This considers, that not all steps require the same time and distributes the additional time occasionally required for restructuring or recomputing over a number of previously done steps.

2.2 Related Work

2.2.1 Static Algorithms

Because every dynamic graph can be analysed with static algorithms using snapshots, a comparison with static algorithms solving the subgraph problem is possible. Besides approximative algorithms, which will not be discussed here, the tools for static graphs can be divided into enumerating and analytic algorithms. The first representatives of the former approach work by first searching for all subgraphs of a certain size k and then counting the frequency of the ones required as output. This approach is very inefficient for this task because most of the time not all subgraphs of a certain size are of interest. Additionally, the position of the individual subgraphs if often not needed. A naive algorithm of this type would take $\mathcal{O}(n^k)$ time where k is the size of the subgraph. More optimized algorithms like the one by Itzhack et al. [15] only require $\mathcal{O}(nc^{k-1}log^{k-2}(c))$ time where c is the average degree of the vertices. But those run times were improved even further with algorithms like [6] by Demeyer et al. by optimizing for structures without (anti-)parallel edges and with symmetries. This was done by using clever data structures and abusing their symmetric properties [20].

The analytical approaches are generally newer and skip the step of locating all subgraphs. An early example of such a method is ORCA [13] with a run time of $\mathcal{O}(\Delta(G)^2\, m\, log\Delta(G))$ where $\Delta(G)$ is the maximal degree of any vertex in the graph G. The main idea of algorithms like ORCA is to find relations between the frequency of the desired subgraph and frequencies of equally large or smaller subgraphs and create a system of linear equations using those relations. When counting subgraphs of size k, only a single subgraph of this size needs to be explicitly counted. This can then be used to determine all other counts via the system of linear equations. This allows the usage of linear algebra, which is probably the

most researched topic in high performance computing with highly optimized and parallel algorithms. A newer algorithm with a similar approach to count subgraphs to size four that managed to improve run time to $\mathcal{O}(\alpha\, m)$, where α is the arboricity of the graph, was created by Ortmann and Brandes [19]. If the four-clique also needs to be counted, the run time increases to $\mathcal{O}(\alpha^2\, m)$ [19]. This algorithm will be used in this thesis to check correctness of the newly implemented algorithm and compare computation time. Despite an increase in research and interest in dynamic algorithms, static methods play a very important role and are continuously developed [20].

2.2.2 Dynamic Algorithms

Counting subgraphs in the dynamic setting is a newer field of studies with fewer algorithms available. One possible approach is to create data structures for dynamic graphs that are optimized for the usage of modified static algorithms. One example would be the algorithm by Lin et al. which uses a system of linear equations to calculate size four s-counts in $\mathcal{O}(n + \alpha m)$ amortized time, where α is the arboricity of the graph, once all triangles are found [16]. Restricting the possible input graphs is also possible, as was done by Dvorak and Tuma [7] by not allowing graphs to be nowhere-dense or having an unbounded expansion to reduce the amortized update time for edge insertion to $\mathcal{O}(log^{\binom{k}{2}-1}(n))$ where k is the size of the subgraph to search for. Algorithms for individual subgraphs like triangles already exist, with examples being the algorithm by Kara et al. [16] which has an amortized run time of $\mathcal{O}(\sqrt{m})$ or an algorithm by Eppstein et al. [9] which has an amortized run time of $\mathcal{O}(h)$ where h is the h-index of the graph. Eppstein also later described a new algorithm together with Spiro [8] that expends the idea of counting subgraphs using the h-index to all subgraphs up to size four with an amortized time complexity of $\mathcal{O}(h^2)$ where h is again the h-index. A similar approach, but using a different method to characterize vertices into high and low degree, was described by Hanauer et al. with an amortized update time of $\mathcal{O}(m^{\frac{1}{2}})$ for length-three paths and triangles as well as $\mathcal{O}(m^{\frac{2}{3}})$ for any four-vertex subgraph except cliques [12]. The last two algorithms will be described further and compared in the following sections.

3 Algorithm Theory

This thesis seeks to implement and optimize algorithms to detect three- and four-vertex subgraphs in a dynamic graph. Both algorithms are designed to work with a continuous graph, meaning one operation needs to take place for each topological change in the graph. To use the storage method of a sequence of static graphs, one would therefore need to create edge insertions and removals in arbitrary order between each snapshot that represent the changes in topology. A single static graph could also be analysed by inserting all existing edges one by one.

This type of algorithm is optimal for large graphs that are created and must be analysed in real time. Because the graph changes constantly over time, a classic static algorithm would have to redo its whole computations after each operation or for each provided snapshot. A dynamic algorithm, on the other hand, only has to update its result after each step, therefore providing the interim results for each time step. This can be especially useful for very large graphs where the total run time of the dynamic algorithm might still be higher than for a single run of the static algorithm, but the dynamic algorithm provides the results for all interim time steps in addition to having a better distributed computation time.

3.1 Principles of Algorithm I

This section will describe the algorithm designed by Hanauer et al. to count size four total- and s-subgraphs in dynamic graphs. The amortized run time required for each update using this algorithm

equates to $\mathcal{O}(1)$ for claws, $\mathcal{O}(m^{\frac{1}{2}})$ for three-paths, and triangles, and $\mathcal{O}(m^{\frac{2}{3}})$ for four-cycles, paws, and diamonds [12]. While it is currently not possible to improve the theoretical run time for maintaining four-cliques below $\mathcal{O}(m)$, the run time of the naive approach without auxiliary counts, a reduction in practical run time could be feasible and should be explored in future work. [12] For completeness, an older algorithm by Hanauer et al., that uses auxiliary counts to maintain four-cliques, was implemented. It is therefore only supplementary to the primary algorithm and not part of the main work. The algorithm can be roughly divided into three subparts:

- An epsilon partition that divides all vertices into low degree and high degree.
- A set of auxiliary counts that maintain the number of auxiliary substructures (see Figure 1) for each vertex.
- A set of the final subgraph (s-)counts.

The first part is an epsilon partition. This means each vertex is labelled as either high or low degree by the algorithm. This table is maintained with each change in the graph. In regular intervals, the partition will be cleared, all vertices newly partitioned, and the auxiliary counts recalculated. Whenever such a recalculation takes place, the cut-off point is determined by the parameter $\theta = (2m_0)^\epsilon$. Here, ϵ is a parameter of the algorithm and m_0 is the number of edges at the last recomputation. In between recomputations, a vertex is only shifted to low degree, if its degree gets below $\frac{1}{2}\theta$ and is shifted to high degree, if its degree becomes larger than $\frac{3}{2}\theta$. Recomputation, where the whole partition is cleared and recalculated, happens whenever the number of current edges has changed by a factor of 2 or $\frac{1}{2}$ from the last recomputation. This means for all vertices v labelled as low degree, $deg(v) \in \mathcal{O}(m^\epsilon)$, and for all vertices v that change partition when an arc is inserted or deleted, $deg(v) \in \Theta(m^\epsilon)$. This information can be used to avoid looping through all neighbours of vertices with a high degree, which would lead to an increased amortized run time. The size of such a partition is as follows: $|H| \in \mathcal{O}(m^{1-\epsilon})$ [12]

In order to be able to avoid looping through neighbours of high degree vertices, some auxiliary counts need to be maintained. Those differ from the final counts in that some vertices they contain are required to be in a certain epsilon partition and that the counts are not updated for each vertex they contain but only for anchor vertices. Those are a set of prediefined vertices for each auxiliary structure. Whenever a structure is found, the respective count for the set of the anchor vertices gets updated. All auxiliary counts can be seen in Figure 1. As an additional benefit, the counts also represent the possibility to cache structure-counts that are frequently used to determine the final number of subgraphs, therefore avoiding repeated calculations. The count of those structures together with the information from the epsilon partition are then used to determine the total counts for the desired subgraphs as well as the s-counts for vertices of interest [11, 12].

To make this procedure more understandable, the following paragraph gives a short outline of the calculations required to maintain the total count for triangles:
If an edge $e = (u, v)$ is inserted, all new triangles that contain that edge must be counted. This means all vertices that are connected to both u and v need to be found. The naive approach would be to loop over all neighbours of u or v. Without additional information about the degree of those vertices, this would result in a run time of $\mathcal{O}(n)$. But the epsilon partition allows us to split this task into two subtasks: To find all connected vertices that have a high degree, one only needs to check all vertices with high degree, which results in a run time of $\mathcal{O}(|H|) = \mathcal{O}(m^{1-\epsilon})$. The remaining triangles whose third vertex has low degree can be determined by querying the auxiliary count uLv[u,v], which takes $\mathcal{O}(1)$, if a hash-set is used. Finally, the auxiliary count uLv also needs to be maintained at each update. Because each edge of the structure uLv contains at least one low-degree vertex, one can design the update in a way, that only the neighbours of said low degree vertex need to be checked, leading to a run time of $\mathcal{O}(m^\epsilon)$. Both those

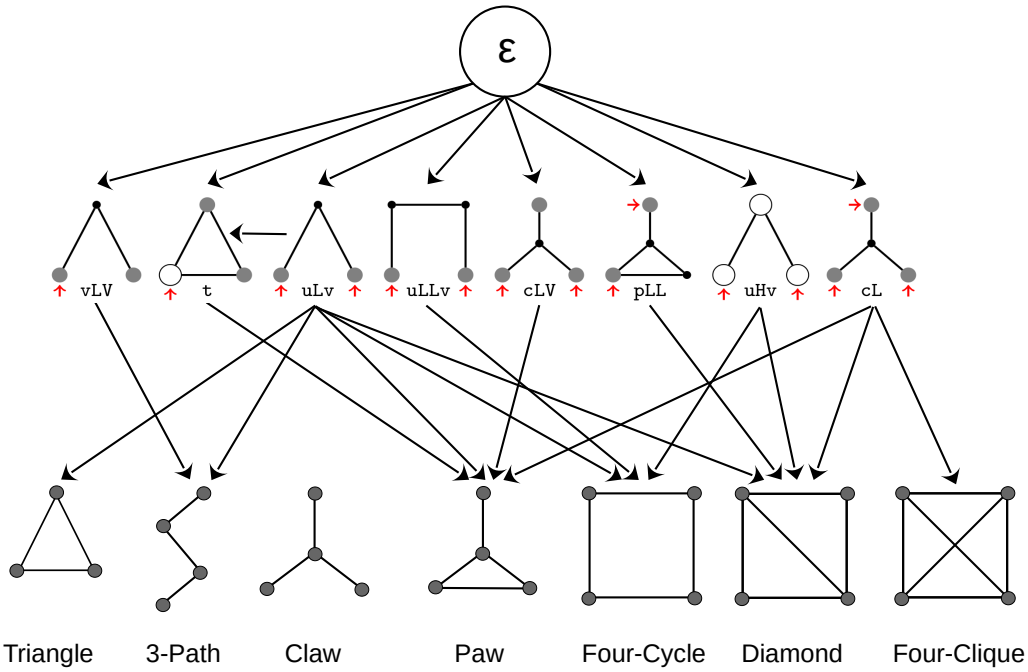


Figure 1: Overview of the dependencies between the different counts maintained by the algorithm by Hanauer et al.. The first row represents the epsilon partition. The second row shows the different substructures, where small dots represent low degree vertices, large and empty dots represent high degree vertices, and large filled dots represent any vertex. The anchor vertices are marked with a red arrow. (Pictograms taken from [12]) The last row depicts the subgraphs whose counts are the final results. Dependencies between elements are drawn as arrows.

subtasks amount to a total update time of $\mathcal{O}(m^{max(1-\epsilon,\epsilon)})$, which equates to an amortized run time of $\mathcal{O}(\sqrt{m})$ if $\epsilon = \frac{1}{2}$ is used [11, 12].

This results in a modular algorithm. Only the epsilon partition always needs to be maintained. If a subset of the subgraphs is of interest, only some substructure counts need to be maintained. Those dependencies can be seen in Figure 1.

The actions of edge removal and edge insertion are symmetric, with the only difference being the order the counts are updated in and that the counts get reduced and not increased. This means, if an edge were to be inserted and then immediately deleted, the counts would be updated in the following order: $eps.\ partition \rightarrow aux.\ counts \rightarrow subgraphs \xrightarrow{\text{deletion}} subgraphs \rightarrow aux.\ counts \rightarrow eps.\ partition$

3.2 Principles of Algorithm II

This section will describe the algorithm designed by Eppstein et al. to count the total number of subgraphs up to size four [8]. Many parallels can be drawn to the algorithm described in section 3.1. Although the general structure is identical, also using the three distinct subparts, the two algorithms differ significantly in detail. The amortized run time for updates is $\mathcal{O}(1)$ for claws, $\mathcal{O}(h)$ for three-paths, and triangles, and $\mathcal{O}(h^2)$ for four-cycles, paws, diamonds, and four-cliques [8].

This algorithm also divides all vertices into high and low degree. But instead of using a constant parameter ϵ to determine a cut-off, all vertices whose degree is larger than h are classified as high degree. Here, h is the h-index. The h-index avoids having to set a parameter like ϵ in the first algorithm, but thereby also gives up an opportunity for tuning in a practical setting. This also avoids the necessity for recomputing the table and auxiliary counts, which tended to be a significant time-sink for the first

algorithm. The exact value of h, and by extension also the theoretical run time of the algorithm, is very dependent on the structure of the graph itself. While additional vertices with low degrees have no impact on h and a few vertices with a very high degree also only have a limited influence, the number of vertices with an intermediate degree is very relevant. The value for h lies between $\frac{m}{n}$ and $\sqrt{2m}$, which implies an amortized run time of $\mathcal{O}(m)$ when counting paws, four-cycles, and diamonds compared to the run time of $\mathcal{O}(m^{\frac{2}{3}})$ when using the first algorithm [8].

Auxiliary counts are also used by this algorithm, though they differ in the exact structures that are saved. A more general difference is that only vertices of high degree are used as anchors. This reduces the number of structures that are found. This is a reasonable choice to reduce the time to count those structures because this algorithm only computes total subgraph counts. On the other hand, the first algorithm can also calculate s-counts. This makes it more efficient to spend more time calculating the auxiliary counts because one structure could play a role in calculating s-counts for several neighbouring vertices, leading to a reduction in redundant calculations. The counts for the actual subgraphs are then determined using the auxiliary counts and the information from the h-index.

The procedure to update the count of triangles is naturally very similar to the one described for the first algorithm:
We again need to find all vertices that are connected to both u and v if the inserted edge is $e = (u, v)$. If u, v, or both are low degree, one can loop through all their neighbours and find all triangles in $\mathcal{O}(deg(u)) = \mathcal{O}(h)$ time. If both have high degree, all triangles with a third high degree vertex can be determined by looking at all vertices in H in $\mathcal{O}(|H|) = \mathcal{O}(h)$ time. The remaining case, where u and v are high degree and a third vertex is low degree, is then covered by an auxiliary count of exactly that structure. The lookup of that information is possible in $\mathcal{O}(1)$ with a hash-map. The maintenance of said auxiliary count is again possible in $\mathcal{O}(h)$, because only neighbours of the central low degree vertex need to be considered. This procedure totals an amortized run time of $\mathcal{O}(h) = \mathcal{O}(m^{\frac{1}{2}})$ [8].

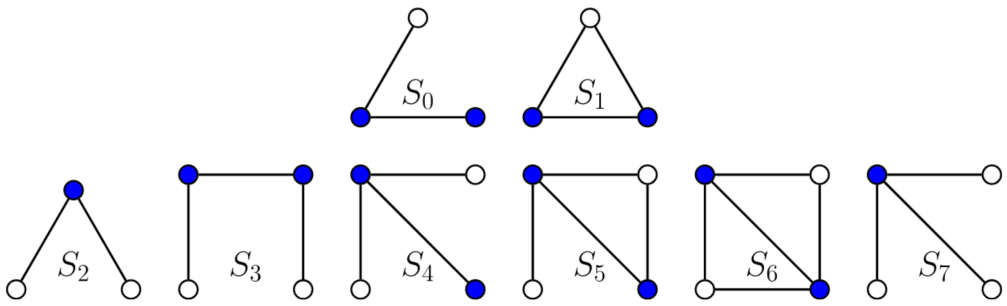


Figure 2: Overview of the auxiliary counts used by the algorithm by Eppstein et al. White circles represent anchor vertices that are high degree. Blue circles are vertices that are low degree. Picture taken from [8]

4 Software Design

The algorithms as described in [11, 12] and [8] were implemented in C++ using the Algora-Framework [10]. Both algorithms could be divided into three separate parts that build on each other: The epsilon partition, the auxiliary structure counts, and the subgraph counts. To be able to use them independently or swap with other implementations, each of the parts described above was implemented as a separate class in C++. The main calculations are started whenever an operation in the given graph occurs. This was facilitated by the option in the Algora-Framework to queue functions as observers to the graph that are then called whenever an operation takes place. As already mentioned above, one big advantage of these algorithms is that they are modular. For example, if only certain subgraphs need to be counted, not all auxiliary structures need to be maintained. This is the case without any additional optimizations, although certain combinations of subgraphs are more efficient than others. Most static algorithms do not allow for such flexibility. For example, the system of linear equations used in ORCA and other similar methods cannot be easily modified to include only certain subgraphs without losing much of their efficiency.

All counts and the currently present edges are stored in hash maps. This means that some of the most frequently called functions are the insertion and deletion of elements from those hash maps, meaning those operations need to be as fast as possible. First, implementations of hash maps/sets of the C++ Standard Library and Boost Library were used. This was later replaced with a version by Malte Skarupke [23]. It was equally important to find an adequate hash function. Most keys were a pair of vertex-IDs (integers). The boost library provides a hash function for pairs that works by combining each value with a seed set at the beginning of the program. While this results in a very balanced hash function, the computational cost is significant. The opposite approach is to just concatenate the bits of both 32-bit integers and interpret them as the new 64-bit hash value. The middle way, that proved to be most effective, was to first concatenate the two numbers and then apply a standard hash-function for 64-bit integers to it. The only caveat with this method is, that it is only efficient as long as the number of edges is below the largest 32-bit integer. This is, however, not a problem in the given setting.

5 Implementation and Optimization

The implementations of the algorithms by Hanauer et al.[1] and Eppstein et al.[2] can be found on GitLab.

5.1 Epsilon Partition

Whenever an edge is inserted, the degree of the vertices it connects are checked and their labels are changed if necessary. No operation is needed when vertices are removed or inserted, because this can per definition only happen to isolated vertices (degree of zero) which are always labelled as low degree. The IDs of all vertices with a high degree are stored in a hash-map to minimize lookup time.

5.2 Substructures/Subgraphs

The counts for all needed substructure and subgraph counts need to be updated, whenever an edge is inserted or deleted. All dependencies that need to be considered for those updates can be seen in Figure 1. If a substructure or subgraph contains a vertex, the ID of the vertex is stored in a hash-map together with the count of the respective substructure or subgraph. If the Epsilon-Partition needs to be recalculated, all hash tables containing the auxiliary counts and epsilon partitions get cleared and the labels of the

[1] https://gitlab.com/leonhards98/subgraph-counting
[2] https://gitlab.com/leonhards98/subgraph-counting_eppstein

vertices are determined from scratch. This is followed by calling the functions that update the auxiliary structures for all already inserted edges in an arbitrary order. The maintenance of the subgraphs is not affected by that.

5.3 Optimizations

The primarily used data structures were hash-maps to achieve an amortized lookup of $\mathcal{O}(1)$. Vertices with a count of zero were removed from the hash-maps to reduce memory requirements.

One of the most influential factors for performance is the method the graphs are stored with. The two main options are adjacency list or adjacency matrix. Both have advantages and disadvantages, as listen below.

Adjacency List	Adjacency Matrix
(+) More compact storage	(-) Inefficient storage of sparse graphs
(+) Fast to loop through all neighbours of a vertex in $\mathcal{O}(deg(v))$	(-) Slow to loop through all neighbours of a vertex in $\mathcal{O}(n)$
(-) Slow to remove edges from the graph in $\mathcal{O}(\Delta(G))$	(+) Fast to remove and add edges to the graph in $\mathcal{O}(1)$
(-) Slow to check for edges between given vertices in $\mathcal{O}(\Delta(G))$	(+) Fast to check for edges between given vertices in $\mathcal{O}(1)$

[22]

As already stated in the introduction, the algorithm relies on looping through the neighbours of vertices designated as low degree by the epsilon partition. As can be seen above, the adjacency list has a big advantage is this regard. Another reason to use an adjacency list is to take advantage of the sparse representation. It is infeasible to use a standard adjacency matrix for large graphs, the domain the algorithm should excel in due to the step wise update-scheme. Even though sparse matrix representations exist, they are typically designed to optimize access time for matrix-matrix or matrix-vector multiplications and not for simple lookup operations. This leaves one problem: One big disadvantage of adjacency lists is that in order to determine, if two vertices are connected by an edge, the adjacency lists of those vertices, which have a length of n in the worst case, need to be searched. Those operations are however also very prevalent in the algorithm and therefore increase the run time significantly.

One possibility to avoid the space requirements of the adjacency matrix, and not lose the advantage of a fast edge-search, is to additionally save all present edges in a hash table. This can be maintained relatively cheap in addition to the adjacency list and provides a faster, though in practice not as fast as an actual adjacency matrix, edge lookup. This optimization was chosen in the implementation.

5.4 Problems and Challenges

Besides the aforementioned optimizations, additional challenges with the algorithms themselves had to be overcome. It was necessary to make slight changes for both algorithms in order to catch some corner cases that were not fully considered in the theoretical papers. However, those were, although tedious to find, not relevant to the concept of the algorithms. Neither did they affect the theoretical run time in any way. For the algorithm by Eppstein et al. it was also necessary to fill in missing details, whenever the algorithmic instructions were not fully complete. Because those details were not critical for the concepts or performance of the algorithm and could be derived with a thorough understanding of the algorithms, they were not discussed with the original authors.

6 Experiments

6.1 Preprocessing

Preprocessing was necessary for certain graph instances to fulfil the algorithmic requirements. If a graph was directed, the heads and tails of all edges were sorted by vertex-ID and all multiple edges were removed. All operations were sorted according to the given timestamps, and the vertices were inserted as isolated points beforehand. Those operations were not counted towards the run time of the algorithm.

6.2 Setup

Both algorithms described above were implemented in C++ and compiled using g++ version 7.5.0 with the optimization flag -O3. All experiments were run on a machine with AMD Opteron 6174 processors with a max clock rate of 2.2GHz and 256Gb RAM under Ubuntu Linux 18.04.6 with kernel 4.15. Each experiment was assigned exclusively to one CPU. All time measurements were set in relation to the implementation of the static subgraph algorithm by Ortmann and Brandes [19] given with the respective paper. This was done to have a used and proven static algorithm as a fixed reference point to compare performance to.

The static algorithm used as reference doesn't count the subgraphs directly but the node automorphism classes, also called orbits, of each vertex. For example, the claw contains two node orbits, where one is the center node and the other the outer nodes. This means if a vertex has a count of one for both those orbits, it is part of two claws, as center vertex for one and outer vertex for the other. Those can then be used to calculate the s-counts with simple additions. On the other hand, the dynamic algorithms can maintain total subgraph counts independent of the counts for each vertex. This gives the dynamic algorithm a large advantage if only the total counts are needed. To represent a more diverse set of potential use cases, time measurements were additionally taken for maintaining the s-counts of all subgraphs for all vertices and the s-counts of all subgraphs for ten randomly chosen vertices.

The set of input graphs was created to cover as many types and sizes of graphs as possible. Several real-world graphs with different sizes were chosen. They represent two of the most important potential application. The Wikipedia graphs represent networks in the domain of the internet, while the graph about protein interactions is from the field of computational biology. Additionally, another graph that maps associations between words was chosen. This was created by presenting words to individual persons and asking which word they associate it with. Words which are often associated were then connected by an edge. Besides those real-world graphs, randomly generated ones with varying densities were used to provide a controlled view of the resulting run times. The details and sources for both the newly generated and real-world graphs that were used to test the algorithms, as can be seen in Table 1. For the generated graphs, all edges were first inserted and then removed in a random order. For the graphs about protein interactions in yeast and word associations, all edges were inserted one after another. The instances from Wikipedia were dynamic in nature, contain a mix of insertions and deletions throughout the whole timeline.

Each of the graphs was analysed using both the static [19] and dynamic algorithms. For the generated graphs, five graphs with identical statistics were created and analysed. The average was then plotted below. All graphs were analysed three times and the median was taken. To improve the readability of the plots, each data point represents the average over the last 1000 steps. For the algorithm by Hanauer et al. the theoretically optimal values for epsilon were used. The counts for claws were not calculated because they are characterized by a constant run time and don't provide any additional insight. Furthermore, both dynamic algorithms use the same procedure to maintain them. Instead, the maintenance of the h-index without any other counts was measured for the algorithm by Eppstein et al.

Table 1: Graph-Instances used to test the algorithms. The first two graphs were generated with a homogeneous edge distribution. The real-world graphs from Wikipedia have a larger range of density. Only the first two million Operations were used in the experiments

Name	# Unique Vertices	# Unique Edges	# Operations	Source
Dense	10 000	100 000	200 000	[18]
Sparse	100 000	100 000	200 000	[18]
Wikipedia de	2 166 669	31 105 755	82 023 142	[1]
Wikipedia en simple	100 312	746 086	1 627 086	[2]
Protein Interactions	2 361	6 646	6 646	[4]
Word Association	23 219	325 624	325 624	[17]

6.3 Results

6.3.1 Generated Graphs

Two graphs of this type were tested, as can be seen in Table 1. Both had a uniform distribution of vertex degrees, with the only difference being the density. In the first half of the operations, all edges were inserted. All edges were removed in the second half. Because all operations with a graph of similar size take approximately the same computation time, this allows for a good visualization of how the algorithms scale with graph size. It can also be seen that edge insertion and deletion is, at least except for the very beginning and end, symmetric for both algorithms. All experiments were repeated five times on different graphs with the same statistics. The average of those runs was taken and plotted.

The measurements using the dense graph can be seen in Figure 3. The difference in run time of the various counts can be best observed in Figure 3c where only the total counts were calculated. For that, only the baseline of the curves should be considered. The time for both triangles and three-paths, whose theoretical run time is $\mathcal{O}(m^{\frac{1}{2}})$, behaves as expected with very similar values. On the next step w.r.t. run time complexity should be paws, diamonds, and four-cycles with $\mathcal{O}(m^{\frac{2}{3}})$. The strong difference between triangles/three-paths and those counts is clearly visible, with all of those counts exhibiting a very similar behaviour with an increase in m. The only count that doesn't behave as expected are four-cliques. Only a very slight increase in slope can be observed when compared to the other counts.

The run time of the second algorithm does not show clear separation between the counts with different theoretical run time. It can also be seen that the run time for partitioning the vertices is computationally expensive due to the numerous hash-map and vector operations needed. It appears that, at least for these types of graphs, four-cycles are most costly to count, while four-cliques are in practice less so. This order also remains for the second set of generated graphs, suggesting this to be a property of graphs with evenly distributed edges when using those algorithms.

The behaviour when maintaining s-counts of random vertices is very similar. When all s-counts need to be calculated, additional factors come into play. The naive approach would be to update the s-counts for each vertex after each operation. But this is not always necessary, as the insertion or deletion of an edge can only influence the s-counts of vertices in its immediate proximity. Now consider the insertion of an edge $e = (u, v)$. As an example, when counting triangles, only the immediate neighbours of either u or v need to be considered. For three-paths, on the other hand, all vertices up to three edges away from u or v need to be considered. This explains the big difference in run time between the two counts, although the run time of the individual updates are the same. The counting of the four-clique also greatly benefits from this, because we again only need to consider all neighbours of either u or v. Paws are the most time-consuming count to maintain, because we again need to consider all vertices up to three edges away from u or v.

The very high and thin peaks that occur in regular intervals for all counts using the algorithm by

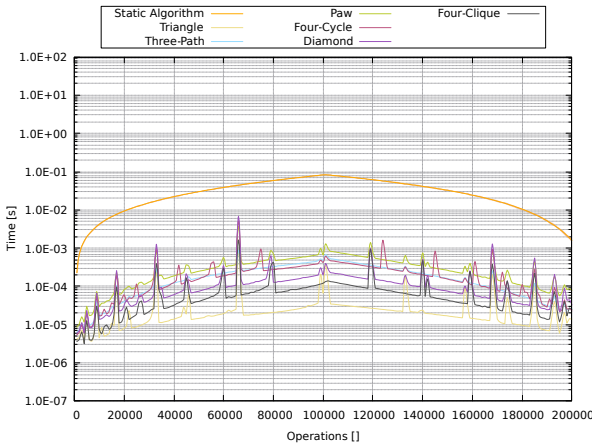
Hanauer et al. represent the recalculations of the epsilon partitions and auxiliary counts. Broader bumps that are not visible in all counts could have different reasons. Firstly, not all operations are equal. If a new edge is inserted between two vertices that happen to have a higher-than-average degree, the resulting calculations are more complex and time-consuming. The second explanation for those peaks can be reasoned with their shape. For those peaks, a strong fronting in combination with a lowering of the baseline can be observed. As both of these characteristics are typical for the memory-restructuring, that needs to take place for hash-maps when many new elements are inserted, this can be seen as the most likely explanation. This behaviour already shows one disadvantage of the dynamic algorithms: When compared to the static algorithm, the run time is significantly lower but also varies to a much greater extent due to different edge insertions having a varying complexity. The static algorithm is not influenced by such effects because it always considers the whole graph and has no concept of which edge was inserted when.

This does not contradict the theoretical run time. The run time of the static algorithm is stated as worst-case run time, meaning that there is a guaranteed upper bound to the run time of each step. The dynamic algorithms on the other hand uses amortized run times. This means there is no guarantee for the individual steps, and individual outliers are possible and expected as long as the surrounding steps compensate for them.

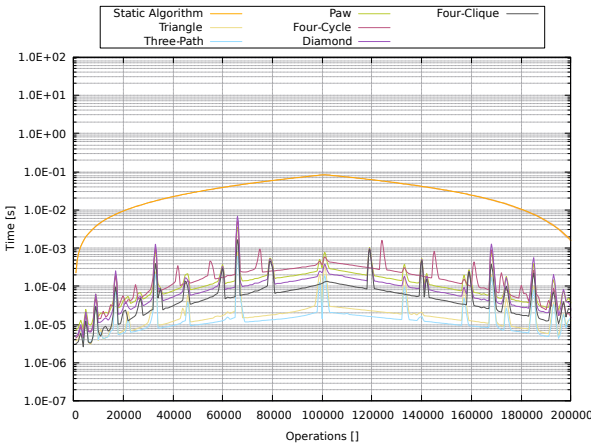
All measurements using the sparse graph can be seen in Figure 4. The algorithms behave very similarly, when compared to the results with the dense sparse graph. The only noticeable difference is, that the dynamic algorithms are significantly faster, both in absolute numbers and when comparing with the static algorithm. This is due to the statistics of the graph. The vertices affected by each operation in the sparse graph have on average a much lower degree compared to the dense graph. This results in a lower run time for maintaining the counts. The static algorithm benefits much less from this decrease in complexity. For a comparison between the two dynamic algorithms, see section 6.3.3

6.3.2 Real-World Graphs

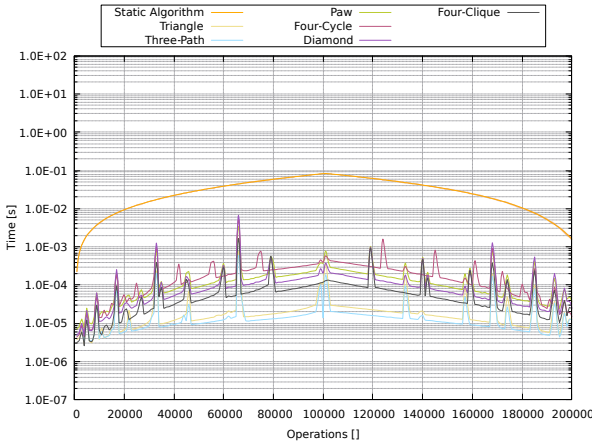
Due to time constraints, only the first two million operations of the graph representing the hyperlink network in German Wikipedia (see Table 1) were used. The runs were aborted if not finished after 180 hours. The results can be seen in Figure 5. The trends here are very similar to those observed in the generated graphs. The less uniform distribution of the edges in this graph is however immediately obvious. The computation time for the static graph is still a very smooth curve, because this algorithm always uses the whole graph. But the dynamic algorithm now has the disadvantage, that the time needed to update all counts is strongly dependent on the currently inserted edge. That is however only the less important of two reasons those peaks exit. In Figure 9 the change in the various subgraphs counts can be observed. This is only shown for the graph representing protein interactions, but the trend holds true for all used graphs. One can see that the change in counts after the first few operations is relatively smooth, meaning this effect can only partially explain the peaks. Even for the segments where the number of subgraphs rises very sharply like around operation 40000 for paws, no notable effects can be seen in the run time. Most of the larger peaks most likely occur for another reason that can be seen best in the total counts for diamonds. For the algorithm by Hanauer et al. this is an expensive count with large spikes. Their occurrence is again linked to the used auxiliary counts. The diamond requires the maintenance of cL and uLv among others. As can be seen in Figure 12, those counts are very populated, resulting in a large number of elements in the respective hash maps. This leads to frequent and extensive memory-restructuring. That a large portion of those peaks can be attributed to the restructuring of the hash maps can be seen in Figure 10. Here, the graph about word associations was analysed a second time. The only difference between the two runs was the used hash map. The hash map that was used for all other



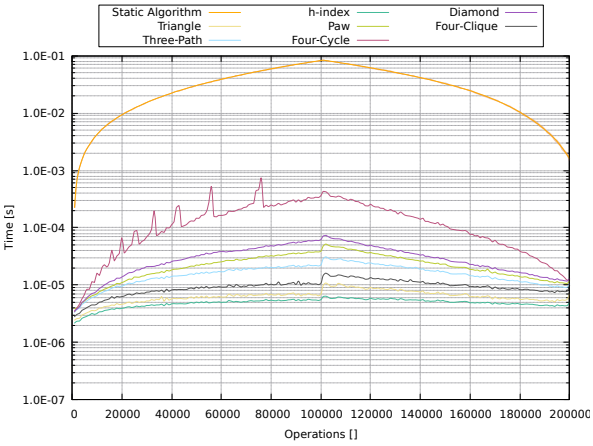
(a) Computing the s-counts of all vertices using the algorithm by Hanauer et al.



(b) Computing the s-counts for ten random vertices using the algorithm by Hanauer et al.



(c) Computing total subgraphs counts using the algorithm by Hanauer et al.



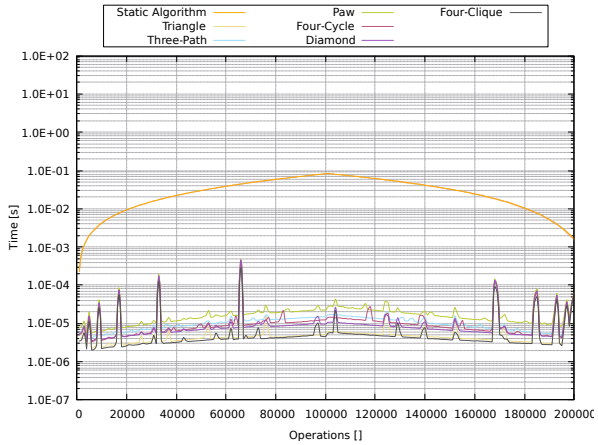
(d) Computing total subgraphs counts using the algorithm by Eppstein et al.

Figure 3: Computation times using both algorithms for the individual steps when inserting and removing edges from the dense graph, as given in Table 1. Comparison between the static algorithm and various setting of the dynamic algorithm.

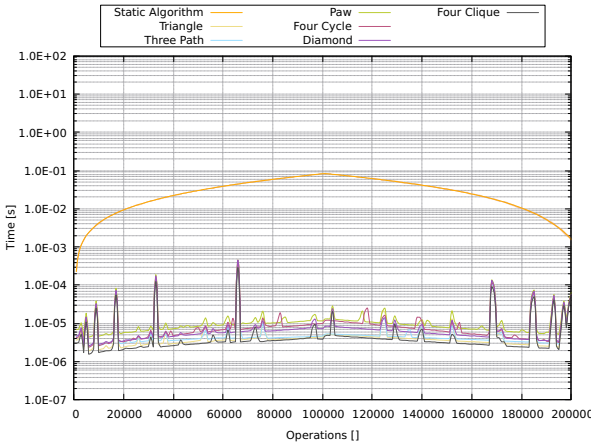
experiments was swapped with the implementation given by the boost library. When comparing the two resulting plots, it is evident, that most larger peaks, except the ones corresponding to recomputations which can be distinguished by their thin shape and presence in all subgraph calculations, are dependent on the used hash map. It can also be observed that the implementation by boost is less efficient but needs less restructuring, as evident by the decreased frequency in large peaks.

The results for the graph representing the hyperlink network in the simple English Wikipedia graph can be seen in Figure 6. The plots here are very similar to those already described above.

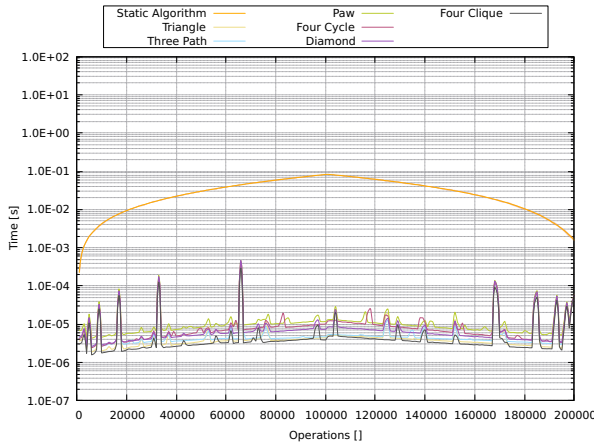
The third and fourth graphs in this category describe the interactions between proteins found in yeast and word associations. Because the original graphs were static, each edge was inserted one after another. This graph instances were also used by Eppstein et al. to test the maintaining of the triangle-count with their algorithm. The results can be seen Figure 7 and Figure 8. These results are very similar to those using the hyperlink network of the German Wikipedia, which are described in the previous paragraph.



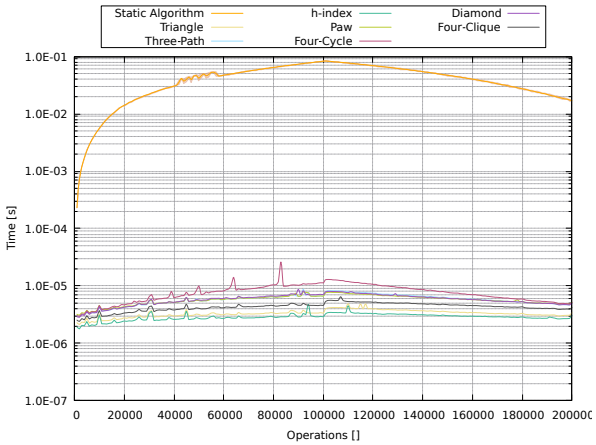
(a) Computing the s-counts of all vertices using the algorithm by Hanauer et al.



(b) Computing the s-counts for ten random vertices using the algorithm by Hanauer et al.

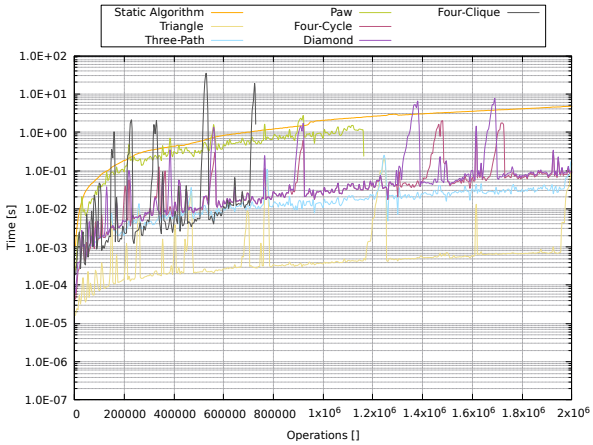


(c) Computing the total subgraph counts using the algorithm by Hanauer et al.

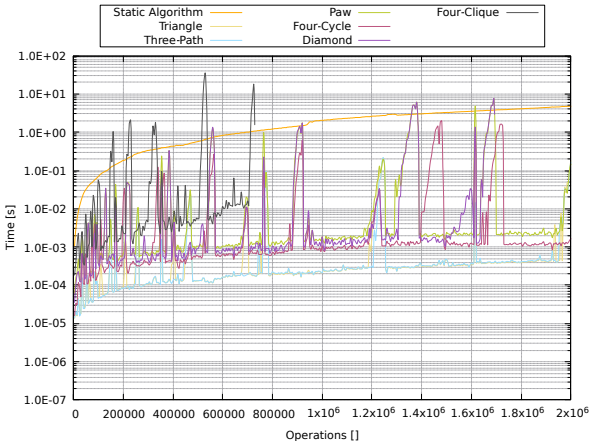


(d) Faction on time spent on various counts using the algorithm by Eppstein et al.

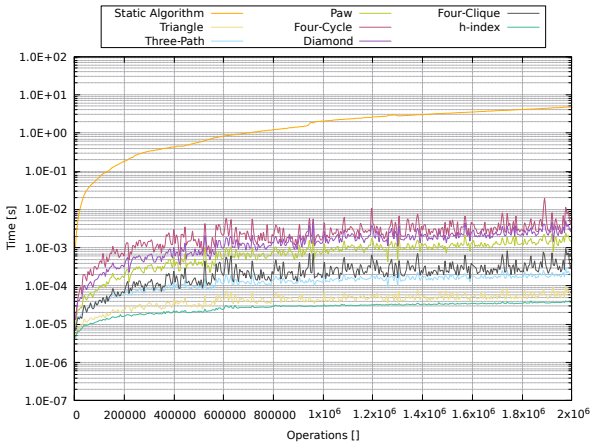
Figure 4: Computation times for the individual steps when inserting and removing edges from the sparse graph, as given in Table 1. Comparison between the static algorithm and various settings of the dynamic algorithm by Hanauer et al.



(a) Computing the s-counts of all vertices using the algorithm by Hanauer et al.

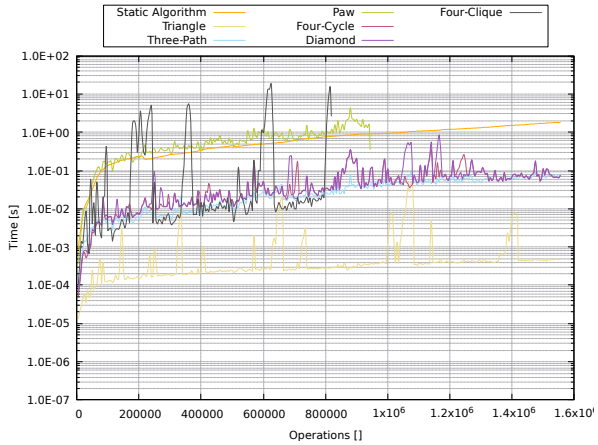


(b) Computing the total subgraph counts using the algorithm by Hanauer et al.

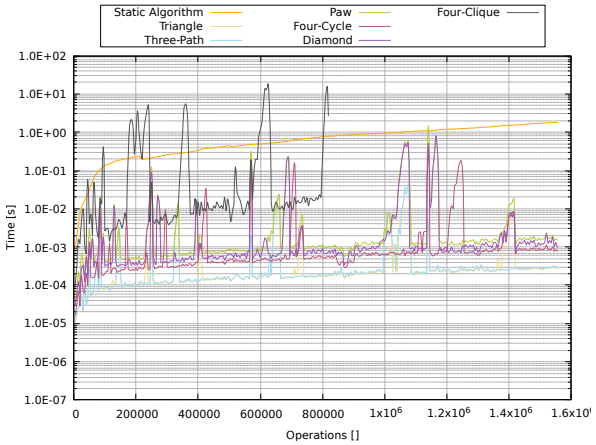


(c) Faction on time spent on various counts using the algorithm by Eppstein et al.

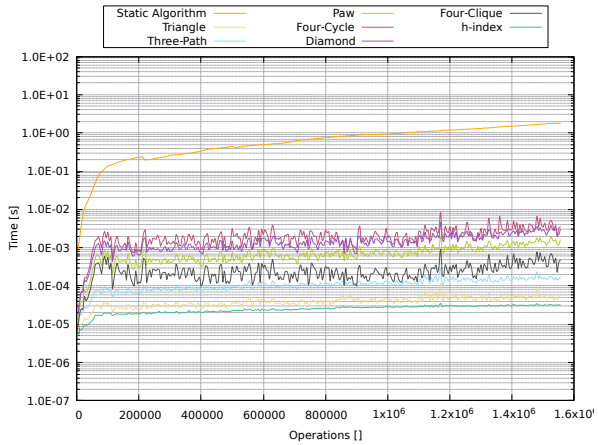
Figure 5: Computation times for the individual steps from the German Wikipedia graph, as given in Table 1. Comparison between the static algorithm and various settings of the dynamic algorithm by Hanauer et al. All computations were aborted if not finished after 180 hours.



(a) Computing the s-counts of all vertices using the algorithm by Hanauer et al.

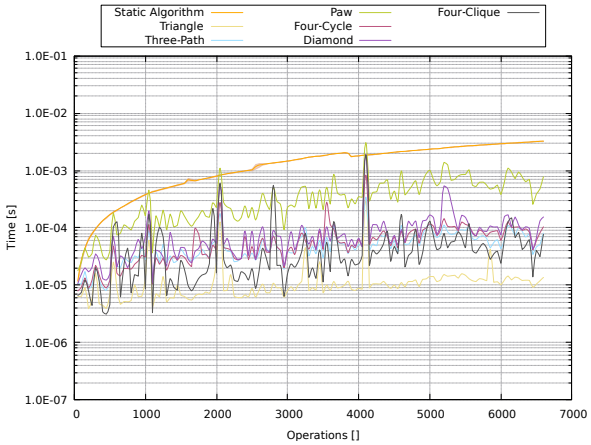


(b) Computing the total subgraph counts using the algorithm by Hanauer et al.

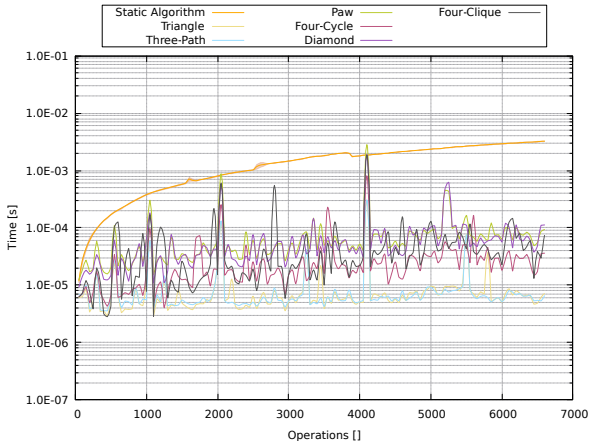


(c) Faction on time spent on various counts using the algorithm by Eppstein et al.

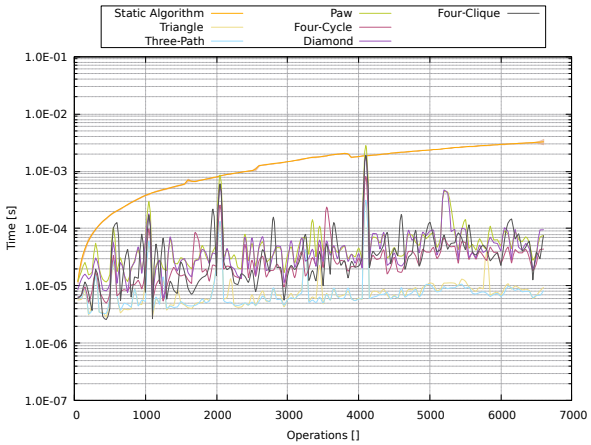
Figure 6: Computation times for the individual steps from the Simple English Wikipedia graph, as given in Table 1. Comparison between the static algorithm and various settings of the dynamic algorithm by Hanauer et al. All computations were aborted if not finished after 180 hours.



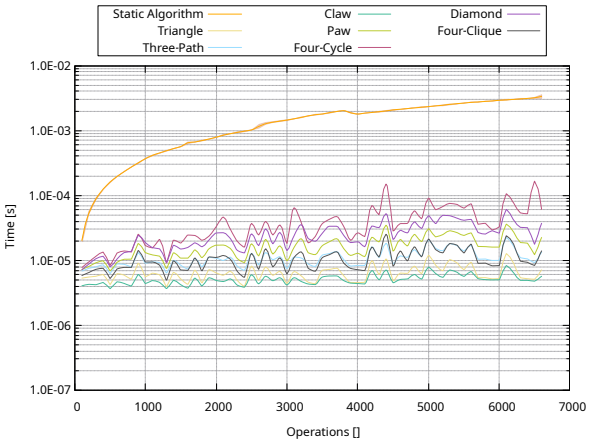
(a) Computing the s-counts of all vertices using the algorithm by Hanauer et al.



(b) Computing the s-counts for ten random vertices using the algorithm by Hanauer et al.

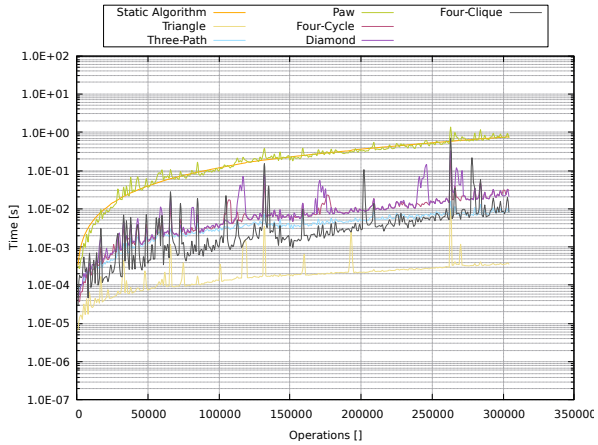


(c) Computing the total subgraph counts using the algorithm by Hanauer et al.

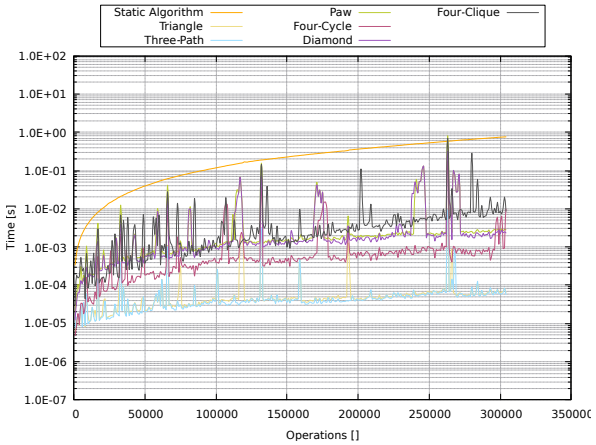


(d) Faction on time spent on various counts using the algorithm by Eppstein et al.

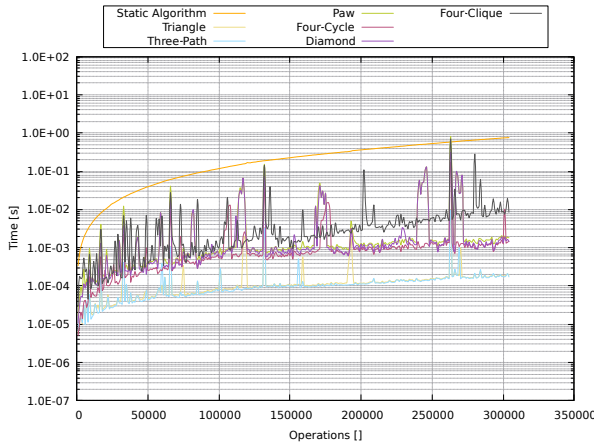
Figure 7: Computation times for the individual steps when inserting and removing edges from the graph containing protein interactions, as given in Table 1. Comparison between the static algorithm and various settings of the dynamic algorithm.



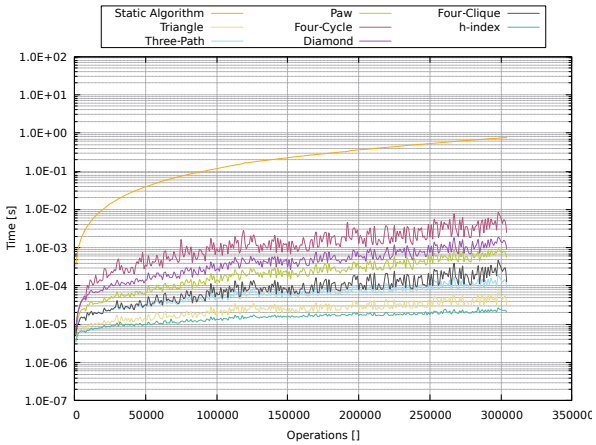
(a) Computing the s-counts of all vertices using the algorithm by Hanauer et al.



(b) Computing the s-counts for ten random vertices using the algorithm by Hanauer et al.



(c) Computing the total subgraph counts using the algorithm by Hanauer et al.



(d) Faction on time spent on various counts using the algorithm by Eppstein et al.

Figure 8: Computation times for the individual steps when inserting and removing edges from the graph containing word associations, as given in Table 1. Comparison between the static algorithm and various settings of the dynamic algorithm.

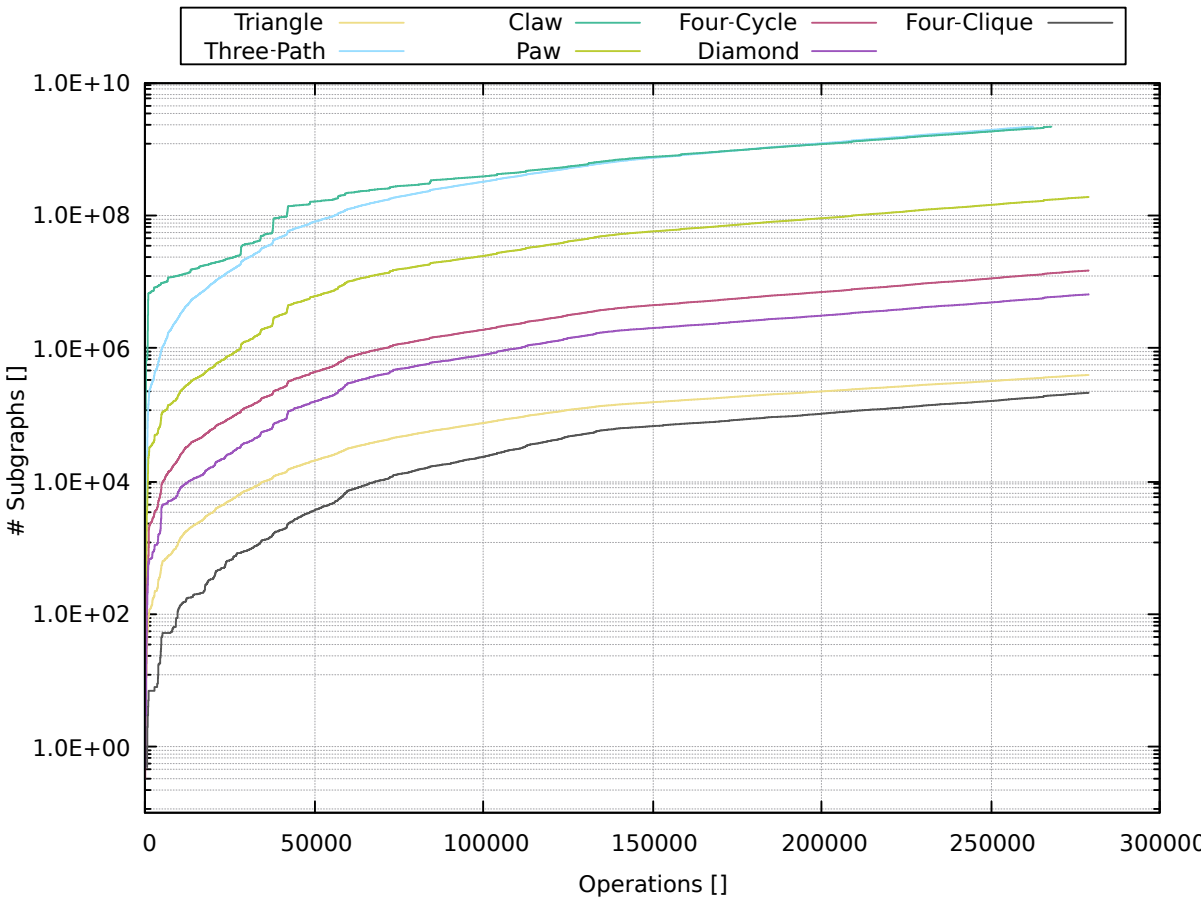


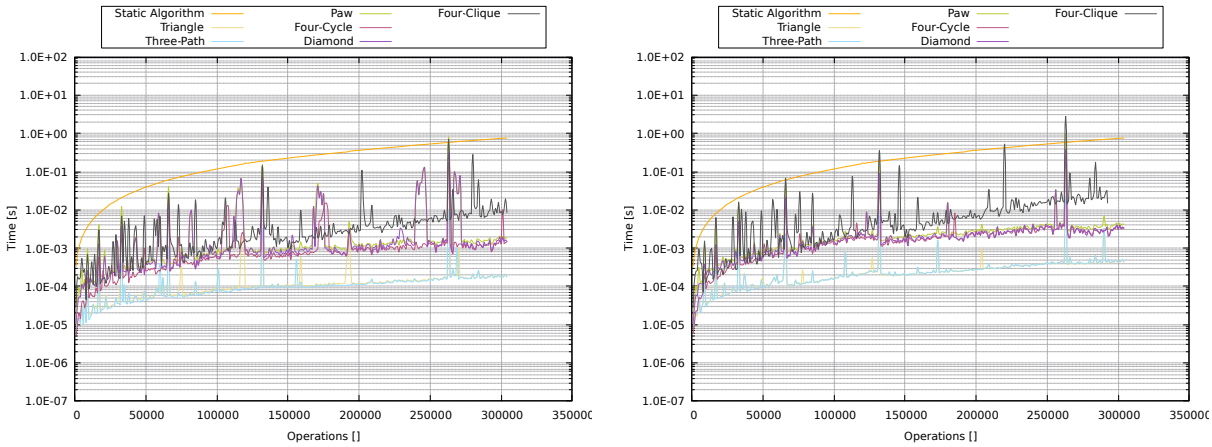
Figure 9: Number of found subgraphs over the lifetime of the word association graph.

6.3.3 Comparison between Algorithms

One main task of this thesis is to compare the performance of the algorithms by Hanauer et al. and Eppstein et al. Firstly, it should be mentioned, that the two algorithms don't have the same capability. The calculation of s-counts cannot be compared, since the second algorithm does not count them. The first algorithm also has a significantly simpler description, with less edge cases that need to be considered.

The second algorithm clearly has a higher base-complexity. Firstly, the run time for all counts is significantly higher for the first few operations. Secondly, the run time for partitioning the vertices, which is constant for the algorithm by Hanauer et al. has a noticeable increase with the number of edges m. While putting a vertex into one of the two partitions is a simple task for the first algorithm, only requiring one lookup for its degree, the second algorithm needs to maintain the h-index each step. This requires a significant number of hash-map and vector manipulations. While these operations have constant run time in theory, this seems to be not the case in practice. The order of the other counts is very similar.

The general trend, which can be seen ever more clearly in the real-world graphs, seems to be that the second algorithm is in practice faster when calculating total counts. The most important factor when analysing the differences in practical run times for total counts is the choice of auxiliary structures. While the algorithm by Hanauer et al. uses auxiliary structures, where many vertices can be either high or low degree, the second algorithm requires a specific label for each vertex, noticeable all anchor vertices need to be high degree. The approach by Hanauer et al. results in a much larger number of found structures. This is a reasonable approach when maintaining s-counts, because updating the s-counts



(a) Run time when using the standard hash function by Malte Skarupke [23]

(b) Run time when using the hash map provided with the boost library

Figure 10: Comparison of the run time when using different implementations of the hash map. The difference in the observed peaks suggests that those are created by restructuring operations of the hash maps.

of several neighbouring vertices often requires the lookup of the same auxiliary counts. This means redundant recalculations can be avoided by choosing broader definitions for the auxiliary counts. It also allows for a shorter description of the algorithm. What is a benefit with s-counts however turned out to be a disadvantage for total counts. Here, each count can at most be used once in each update step. This means the advantage of caching is removed, while the problem of counting structures for vertices that are never queued is exacerbated. Additionally, the high number of counts that need to be stored also slow down all operation of the used hash maps and increase the need for restructuring the hash maps, resulting in the peaks described in the section above. The effect the size of a hash map can have on the lookup times can be seen in Figure 11 which is taken from a blog of the creator of the used hash map. It should be noted that those numbers were measured using single integers as keys, which is more efficient than the keys necessary in the implemented algorithm. Those peaks also don't occur as regularly or have the exact same shape as in the plots for the real-world graphs because not every operation changes the hash maps equally.

To make the difference in auxiliary counts more visible, the counts of the structures used in the two algorithms that only differ in the required vertex labels, were plotted in Figure 12. These measurements were taken for the first 100 000 operations of the German Wikipedia graph with an epsilon of 0.33. The results are consistent with the previously mentioned notion, that the algorithm by Hanauer et al. uses much more frequent auxiliary counts. For nearly all comparable counts, the differences are very pronounced, and up to two orders of magnitude large. The general shape is however quite similar, which was expected.

6.3.4 Effects of the Parameter Epsilon

As already mentioned above, the parameter epsilon can be used to tune the algorithm by Hanauer et al. in practice. In Figure 13, the speedup for maintaining total counts when using different epsilons compared to the theoretical optimum can be seen for the first 100 000 steps of the German Wikipedia graph and the graph describing protein interactions. For the German Wikipedia graphs, the trend is the same for all subgraphs: The best epsilon in practice is smaller than the theoretical one. This is especially the case for the counts that have a theoretical optimum of $\frac{1}{2}$. The reason for the large discrepancy in those cases comes again down to the resulting number of auxiliary counts. Choosing a smaller epsilon

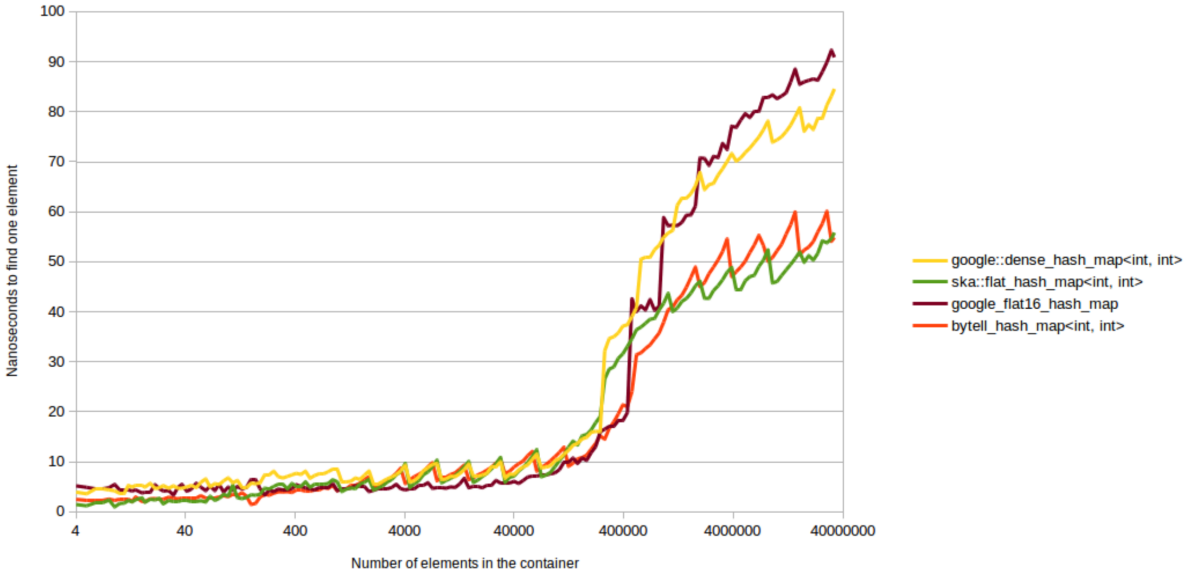


Figure 11: Speed of successful lookups for various hash maps using integer-keys. The hash map used for the implementation of both algorithms corresponds to the red line [24].

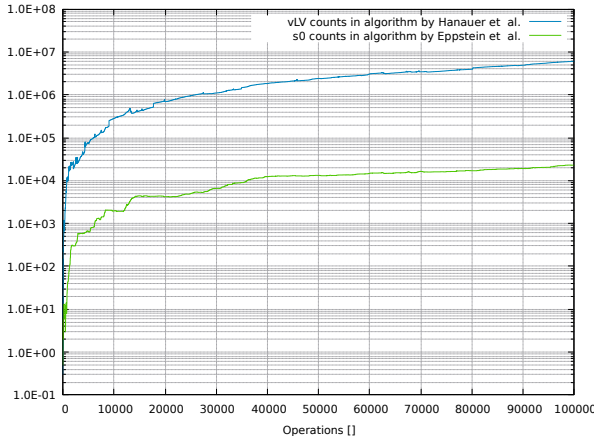
leads to fewer auxiliary structures. This leads to faster computation and better performance of the used hash maps. Faster counting of the final subgraphs seems to be unable to offset this phenomenon. Those effects can be observed for triangles and four-cliques. The effect, that a change in epsilon can have on auxiliary counts can be seen in Figure 16 for cL but was similar for other auxiliary counts. This increase in size of cL was also evident while running the algorithm as counting four-cliques required approximately ten times more RAM then other subgraphs.

The same principle, albeit to a lesser degree, also applies to the remaining counts that have an optimal value of $\epsilon = \frac{1}{3}$ in theory. Here, the optimal value is shifted only slightly from $0.\dot{3}$ to 0.3.

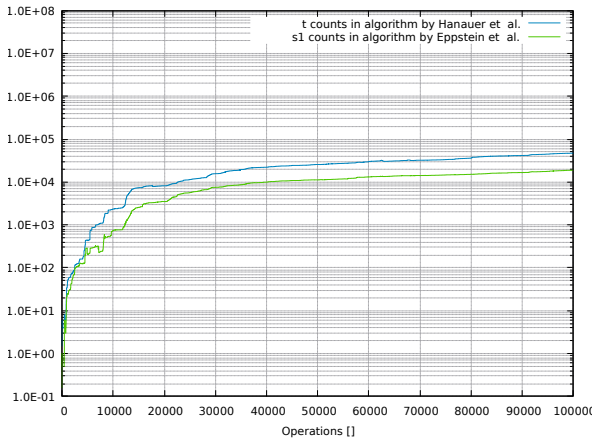
When comparing those findings to the curves for the yeast graph, the big problem with optimizing the epsilon becomes obvious: While the general trend, that the practical best epsilon is slightly lower than the theoretical value, is still valid for most subgraphs, further similarities to the German Wikipedia graph can not be observed. This means, that the optimal epsilon is very dependent on the actual graph used. It can be seen that the speedup possible for the Yeast graph is significantly lower on average. This can be explained with the size of the graphs. The Yeast graph is smaller, meaning the all used hash maps contain fewer elements, reducing negative effects as noticeable in Figure 11.

To complicate the matter even further, the best epsilon in practice is not constant within a graph, when comparing total- and s-counts. This is explored in Figure 14, where the possible speedup, when maintaining s-counts is plotted against different values for epsilon. As already stated above, an increased epsilon leads to a larger number of found structures, which can then be used multiple times per step, when maintaining the s-counts of neighbouring vertices. This explains the shift of the optimal epsilon towards larger numbers. As already noted with the normal counts, the possible speedups are generally lower for the Yeast graph, because of its smaller size.

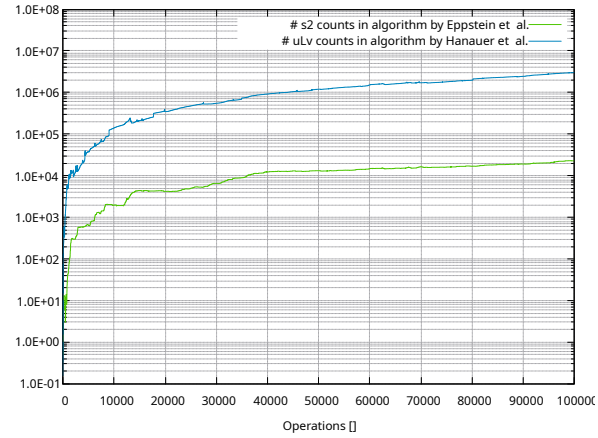
It should be noted, that the mentioned values only apply to the exact intervals of the graphs they were measured in. Those numbers cannot be generalized to all possible inputs. Different graphs will result in different optimal epsilons. And even within a single graph, the optimal epsilon can vary depending on the current time step. For large graphs, where it is known that their basic composition and structure does not change over time, it might be feasible to optimize the epsilon with the first few thousand steps, but this would most likely not be worthwhile for graphs without those known properties. However, it



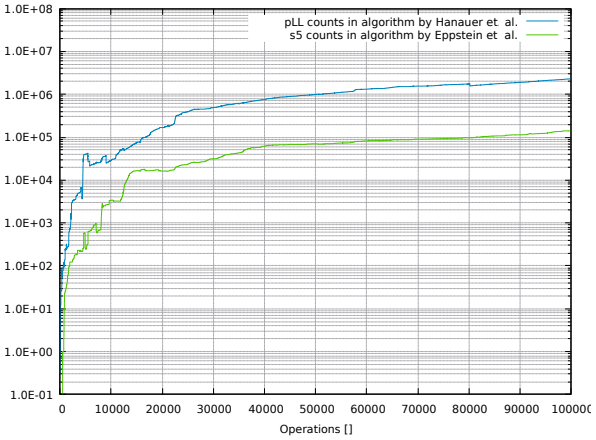
(a) Comparison between vLV and s0



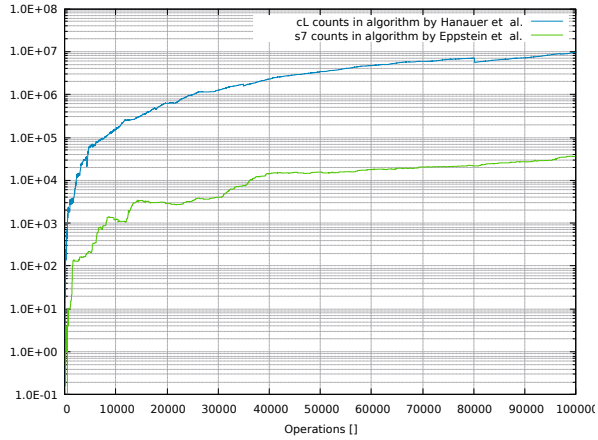
(b) Comparison between t and s1



(c) Comparison between uLv and s2



(d) Comparison between pLL and s5



(e) Comparison between cL and s7

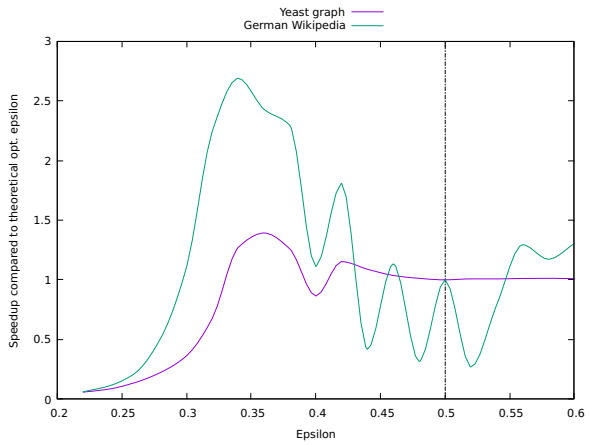
Figure 12: Comparisons of the auxiliary counts used in the two algorithms that only differ in the required labels for the vertices after the first 100 000 operations of the German Wikipedia graph with an epsilon of 0.33. The exact structures can be found in [12] and [8]

can be said, that the optimal epsilon for most graphs most likely lies between 0.3 and 0.35. Even though that may seem to be a small interval, the narrowness of the peaks means, that already a slight variance from the optimal epsilon can lead to a much worse performance.

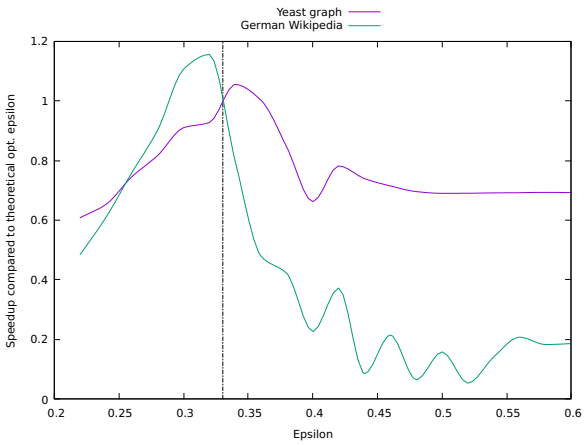
A completely different behaviour was observed for the generated graphs. Only a single plot is shown here in Figure 15, because they looked identical for all counts. Already at a small value for epsilon, a very stable plateau was reached. This is most likely, because from this point on all vertices get classified as low degree, making any further increases irrelevant. Before that, all vertices were classified high degree. This is possible, because the properties of the graph result in all vertices having approximately the same degree. One hypothesis, why the state of all vertices being low is more beneficial is, that six out of the eight auxiliary counts contain at least one low degree vertex, while only two contain a high degree vertex. If all vertices are high degree, only those two auxiliary counts can be used, resulting in a decrease in efficiency.

So far, each subgraph count was observed independently of the others. However, it would be likely, that several or all subgraph counts are determined at once. While it is definitely not beneficial to maintain the auxiliary counts for each subgraph with its optimal epsilon, two different epsilon partitions and auxiliary counts for the theoretical optima of $\epsilon = \frac{1}{2}$ and $\epsilon = \frac{1}{3}$ might be a good compromise between additional calculations and more efficient counting. The speedup when using a single epsilon compared to two partitions with the before mentioned values can be seen in Figure 17. This experiment was done using the German Wikipedia and Yeast Graph. At least for the used graphs, choosing a single epsilon which is a compromise between the optimal values was the better choice. The trend, that can be seen here suggests that the optimal epsilon for computing all total counts at once is between 0.35 and 0.37. The run time of the algorithm by Hanauer et al. when calculating all total counts while using only a single epsilon partition with an epsilon of 0.35 can be seen in Figure 18. Here, all the previously mentioned advantages and disadvantages can be seen. While the dynamic algorithm is significantly faster if the subgraph counts at each step need to be calculated, the variance in run time is much higher. The advantage the dynamic algorithm has, is not as large as to make it efficient to use it, when the subgraph counts are only required at a few steps in the dynamic graph. This is the configuration we would recommend, if all total subgraphs are required.

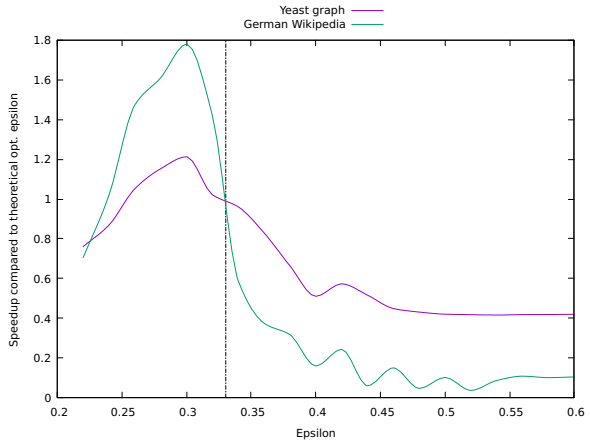
Another big difference between the two algorithms is that the one by Hanauer et al. requires recomputations of both the epsilon partition and auxiliary counts. While the amortized run time takes those into account, it still results in spikes in the run time. One option to reduce those effects is to space the recomputations further apart. The comparison in Figure 19 shows the effect this can have. It can be seen, that doubling the interval removes the spikes in run time, but has no other significant influence. Nevertheless, removing the recomputations completely would result in a decreased performance for larger graphs in the course of the calculations. It would therefore be advisable to reduce the number of recomputations by half or more. For smaller graphs, limiting recomputations even more would most likely be a good choice.



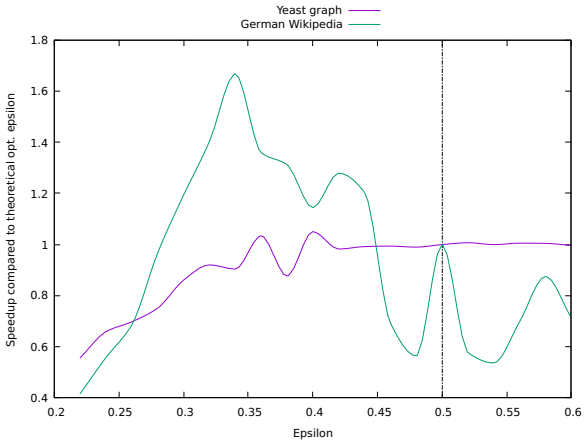
(a) Four-Cliques, 2.67, 1.39;
using cL



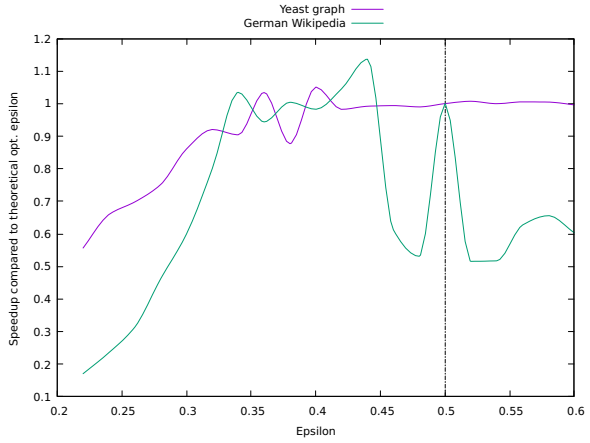
(b) Paws, 1.16, 1.05;
using t, uLv, cLV, cL



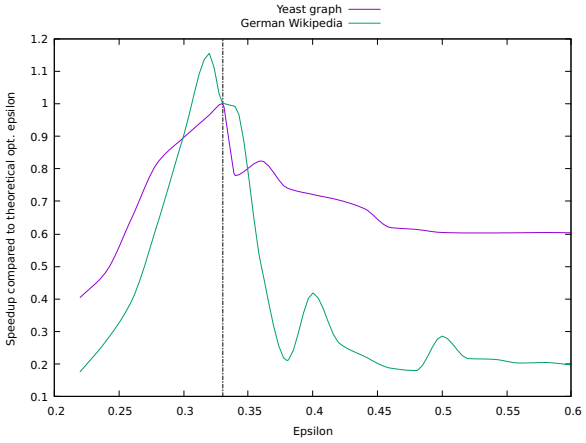
(c) Diamonds, 1.78, 1.21;
using uLv, pLL, uHv, cL



(d) Triangles, 1.68, 1.05;
using uLv

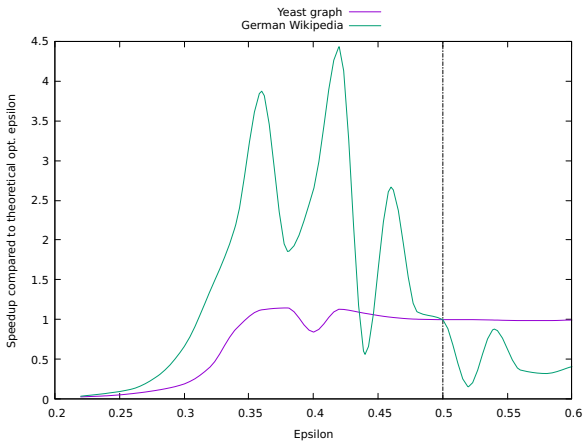


(e) Three-Paths, 1.13, 1.047;
using vLV, uLv

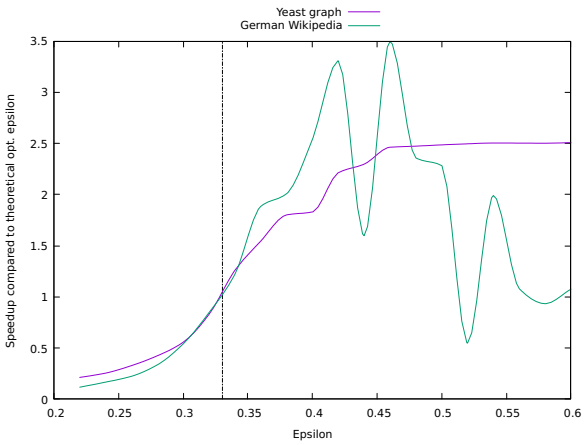


(f) Four-Cycles, 1.15, 1.00;
using uLv, uLLv, uHv

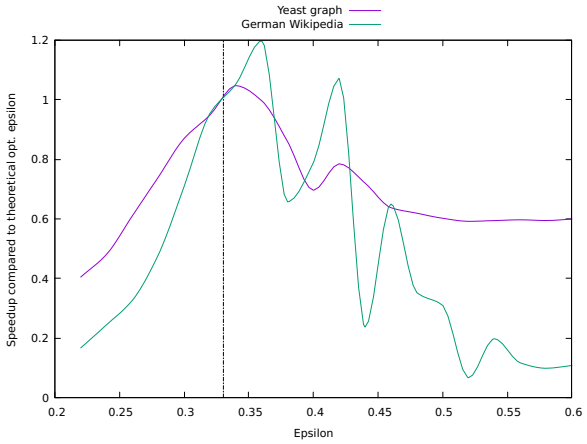
Figure 13: Possible speedup for maintaining total counts when using different values for epsilon, compared to the theoretically optimal value, which is visualized as a dotted line. The total run time for the first 100 000 steps of the German Wikipedia graph was used. The subcaptions contain the used subgraph, followed by the maximal speedup possible for the German Wikipedia graph and the maximal speedup possible for the yeast graph. Additionally, the auxiliary counts required for the calculations are listed.



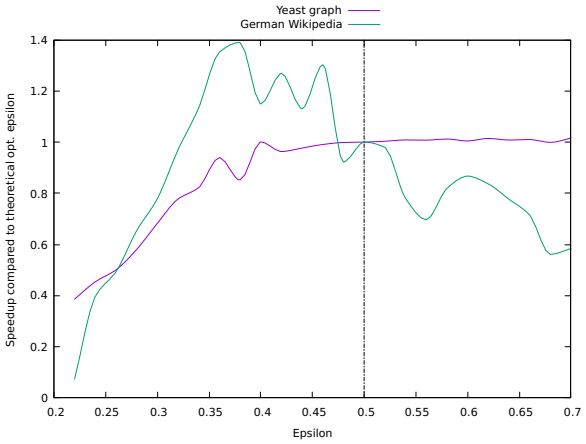
(a) Four-Cliques, 4.48, 1.15;
using cL



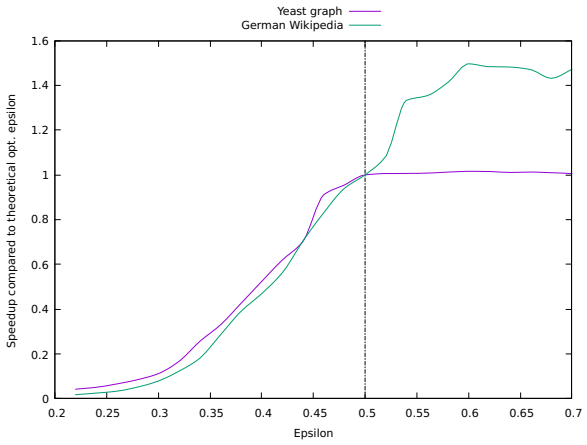
(b) Paws, 3.50, 2.50;
using t, uLv, cLV, cL



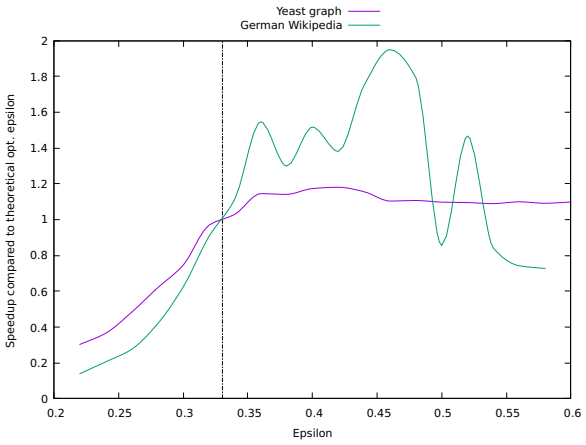
(c) Diamonds, 1.20, 1.05;
using uLv, pLL, uHv, cL



(d) Triangles, 1.39, 1.01;
using uLv



(e) Three-Paths, 1.47, 1.00;
using vLV, uLv



(f) Four-Cycles, 1.95, 1.18;
using uLv, uLLv, uHv

Figure 14: Possible speedup for maintaining all s-counts when using different values for epsilon, compared to the theoretically optimal value, which is visualized as a dotted line. The total run time for the first 100 000 steps of the German Wikipedia graph was used. The subcaptions contain the used subgraph, followed by the maximal speedup possible for the German Wikipedia graph and the maximal speedup possible for the yeast graph. Additionally, the auxiliary counts required for the calculations are listed.

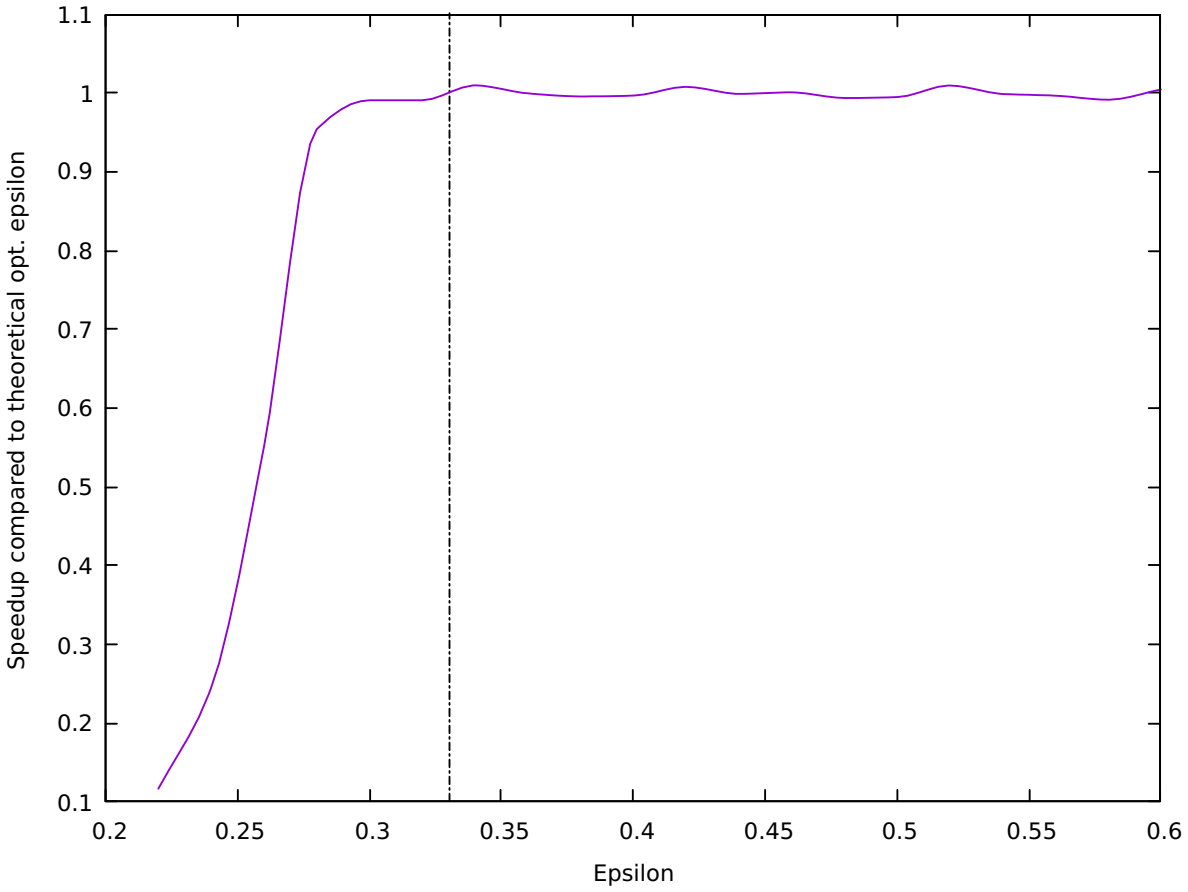


Figure 15: Possible speedup for maintaining the total number of paws when using different values for epsilon, compared to the theoretically optimal value, which is visualized as a dotted line. The dense graphs as specified in Table 1 were used. The maximal speedup possible was 1.02.

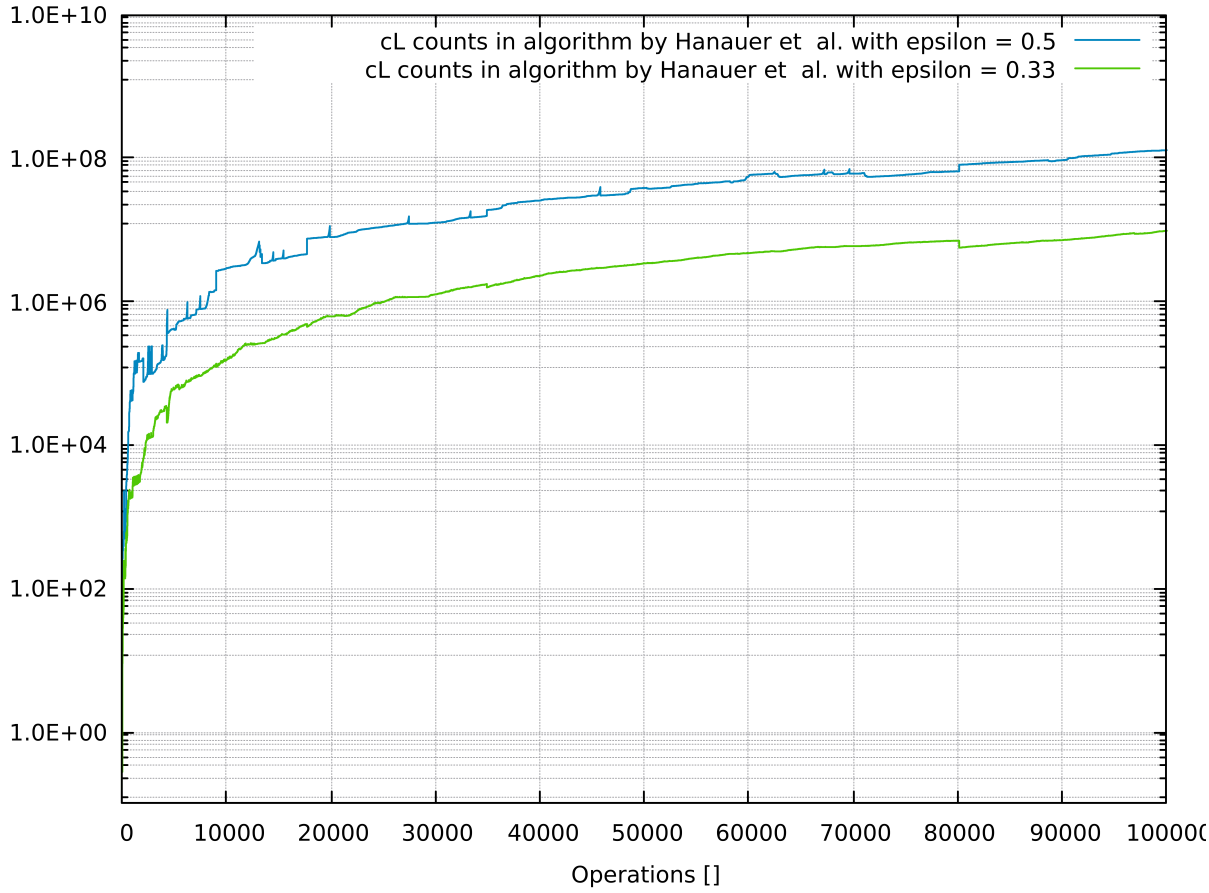


Figure 16: Comparison of the number of auxiliary structures of type cL with different values for epsilon. Measurements taken for the first 100 000 Steps of the German Wikipedia graph.

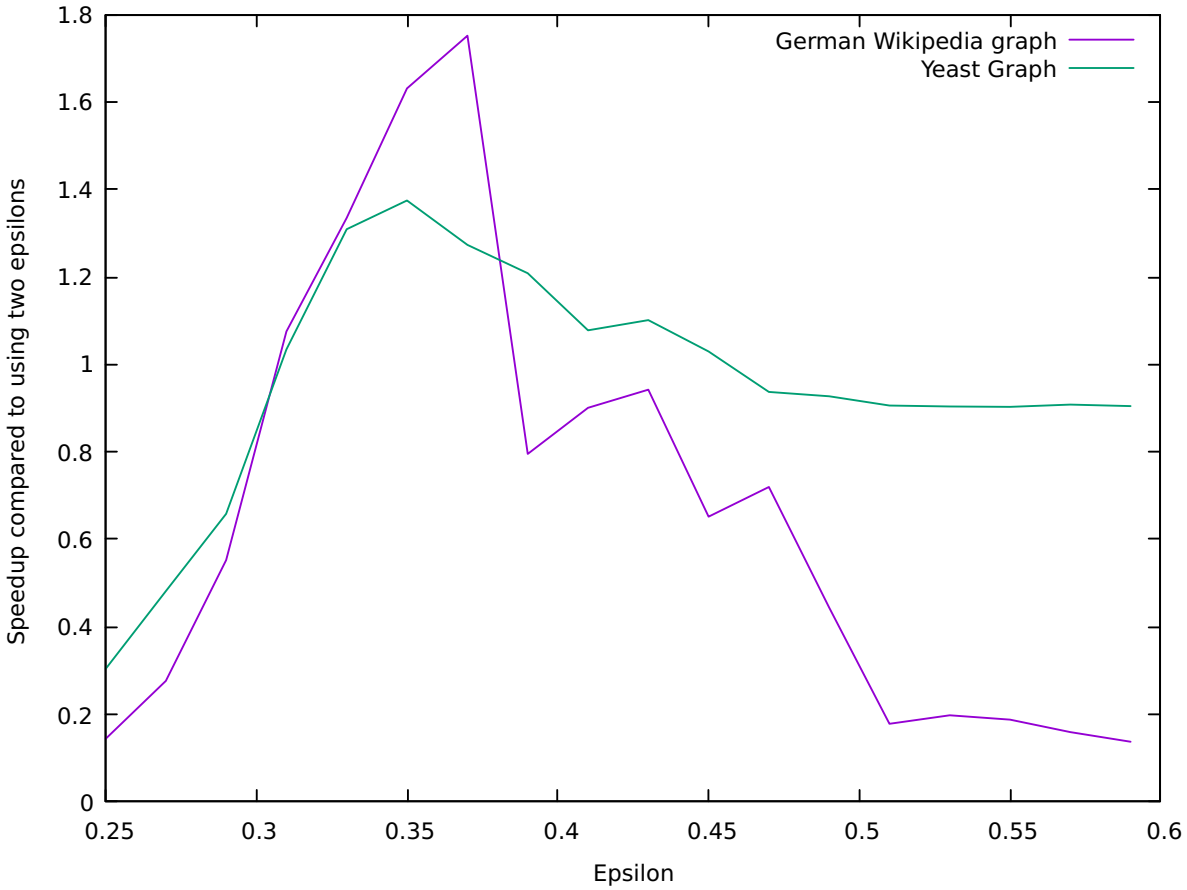


Figure 17: Speedup achieved for counting all subgraphs when using a single epsilon partition, compared to two epsilon partitions with the theoretically optimal epsilon values. Measurements taken for the first 100 000 operations in the German Wikipedia graph and the whole Yeast graph. The maximal possible speedup is 1.75 and 1.32

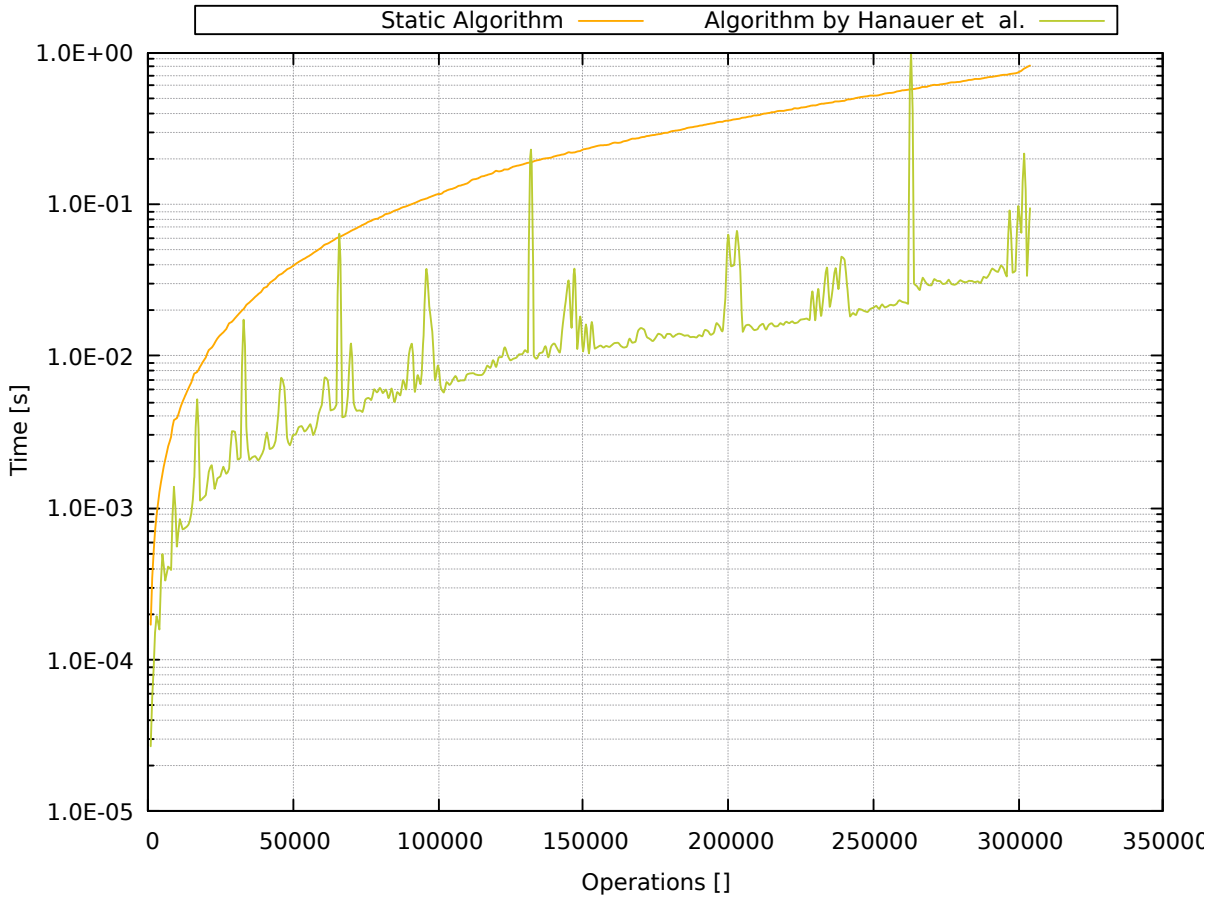
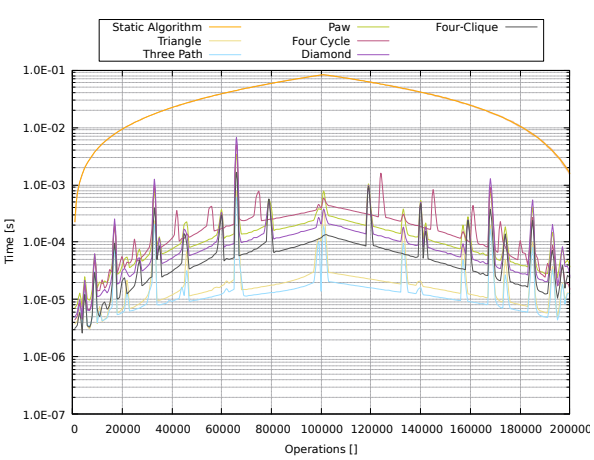
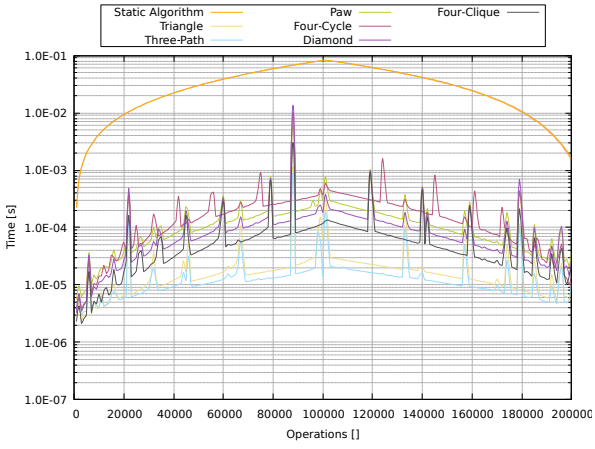


Figure 18: Run time for maintaining all total subgraph counts using the algorithm by Hanauer et al., compared to the run time of the static algorithm. For the dynamic algorithm, a single epsilon partition with $\epsilon = 0.35$, which was found to be the practical optimum, was used.



(a) Run time when recomputing as specified in [12]



(b) Run time when recomputing only half as often as specified in [12]

Figure 19: Run time for the individual steps when inserting and removing edges from the dense graphs given in Table 1 using the algorithm by Hanauer et al. Firstly, with recomputation as specified in [12], such that recomputing happens whenever the current number of edges is more than double or less than a quarter of the value it had at the last recomputation. Secondly, with recomputation half as often. When the current number of edges is more than quadruple or less than a sixteenth of the value it had at the last recomputation. The count for the claws was not included, since they don't require the vertex partition or any auxiliary counts.

7 Future Work

The modularity of both dynamic algorithms allows for many possible variations in further work. One question, that is still unanswered, is if counting four-cliqes also benefits from using auxiliary counts. Even though the theoretical run time is $\mathcal{O}(m)$ both for the naive approach and for the method used here, an advantage in practice is a possibility [12]. The most obvious modification is to extend the calculations to subgraphs of size five or even larger. This would require additional auxiliary counts. Due to the exponential increase in possible graphs with the number of vertices, it is a better approach to consider only those relevant in the desired application. The algorithms can also be extended to utilize additional attributes of the graph like edge or vertex labels, that also need to match for a subgraph to be counted. This would however require additional auxiliary counts, that also consider labels. If a very large number of subgraphs need to be counted, it might be faster to first enumerate all subgraphs with the desired structure, followed by counting those where the labels match. This strategy also allows changing the labels of the desired subgraphs more easily. One possible application of such an algorithm is to count important functional groups in chemical reaction networks. In this case, each vertex represents one atom with the label describing its type and properties, while an edge between two vertices implies a bond between them with the bond type specified in the label. A change in the graph then represents chemical reactions, that form or break bonds. For large biological networks, such an algorithm could maintain the fingerprints [5] of the involved molecules, which are used to predict reactivity and other properties of the molecules.

Another problem that can be tackled using the principles of the algorithms discussed here, is to add a dynamic aspect to the subgraphs. This means, a subgraph is no longer just a static object that can exist at one time of the dynamic graph, but also changes over time. This would make it possible to count transitions between different structures. One can not only use such an algorithm in cheminformatics to observe specific reaction, which can be defined as changes in edges, but it would also make it possible to more closely analyse the communication patterns of social network graphs.

8 Conclusion

In this thesis, we compared different methods to maintain the count of size four subgraphs in a dynamic graph by implementing two new algorithms [11, 12] [8] and comparing run times with the implementation of a similar static algorithm [19]. Both algorithms we tested, managed to reduce the run time per step significantly when compared to the static algorithm, while providing a more modular approach, where any combination of subgraph counts can be calculated without losing efficiency. The main disadvantage was a much more unpredictable run time with some operations, depending on the operation and current state of the algorithm, having a time requirement up to two orders of magnitude larger than surrounding operations. To optimize performance, we determined, that a data structure that allows for both fast checks if there is an edge between two vertices and fast looping through all neighbours, is necessary. We found, that those requirements are best fulfilled by using the standard incidence list in combination with a hash set containing all pairs of connected vertices. Because of this choice, together with our use of hash maps for the auxiliary counts, even a small improvement in the hash map or hash function will result in a relevant improvement of run time.

The different intentions between the two dynamic algorithms can be seen when comparing their run time for maintaining total subgraph counts. Even though the algorithm by Hanauer et al. has a better theoretical run time for total counts, it uses much more frequent auxiliary counts, which increases the practical run time above the one for the algorithm by Eppstein et al. This is because the algorithm does not need all of those counts every step, but the increased number of entries in the hash maps decrease

efficiency for access and insertion operations. This is however a reasonable compromise when considering s-counts, since a single auxiliary count can often be used for the s-counts of multiple vertices in close proximity, reducing the total run time. The algorithm by Eppstein et al. doesn't need such considerations, as it is not designed to maintain s-counts. This makes it easy for us to recommend the algorithm by Hanauer et al. for s-counts, since Eppstein does not provide an alternative. When the total counts are required, we consider the algorithm by Eppstein et al. to have a more stable and generally slightly lower run time. One noticeable exception occurs, if many very similar graphs need to be analysed. If we optimize the parameter epsilon of the algorithm by Hanauer et al., it can result in it having the lower run time. The parameter epsilon turns out to be a very sensitive modifier, where changes by a few percent in some cases half the run time of the algorithm. We can not determine an optimal value for epsilon for the individual subgraphs, because there are too many hidden influences. As a general trend, we determined the optimal value for total counts to be between 0.3 and 0.35 depending on the used graph. The optimal value when determining s-counts is on average a bit higher than for total counts. When maintaining all total counts at once, the optimal value for epsilon was more similar when using different graphs. For this case, we advise an epsilon between 0.35 and 0.37. For calculating claws, both algorithms use the same procedure.

We believe that there is still a large potential to modify the algorithm by Hanauer et al. Firstly, the used auxiliary counts can be modified such that the anchor vertices need to be high degree, therefore reducing the number of found structures. To reduce the impact that would have on the s-counts, where the algorithm uses one auxiliary count to calculate the s-counts of several neighbouring vertices, we would propose those counts to be calculated once per step and then stored. This would still increase the needed calculations, since the required counts have to be determined from scratch, but at least redundant calculations within one step can be avoided. Such modifications will require the description of the algorithm to be more complicated, because more corner cases need to be considered. Secondly, our results show, that the interval of recomputations can be increased by at least a factor four and probably even more for graphs, that only contain a few tens of thousands of operations.

List of Figures

List of Tables

References

[1] Wikipedia dynamic (de). http://konect.cc/networks/link-dynamic-dewiki/.

[2] Wikipedia dynamic (simple). http://konect.cc/networks/link-dynamic-simplewiki/.

[3] Akhtar, N., Javed, H., and Sengar, G. Analysis of facebook social network. In *2013 5th International Conference and Computational Intelligence and Communication Networks* (2013), pp. 451–454.

[4] Bu, D., Zhao, Y., Cai, L., Xue, H., Zhu, X., Lu, H., Zhang, J., Sun, S., Ling, L., Zhang, N., Li, G.-J., and Chen, R. Topological structure analysis of the protein-protein interaction network in budding yeast. *Nucleic acids research 31* (06 2003), 2443–50.

[5] Cereto-Massagué, A., Ojeda, M. J., Valls, C., Mulero, M., Garcia-Vallvé, S., and Pujadas, G. Molecular fingerprint similarity search in virtual screening. *Methods 71* (2015), 58–63. Virtual Screening.

[6] Demeyer, S., Michoel, T., Fostier, J., Audenaert, P., Pickavet, M., and Demeester, P. The index-based subgraph matching algorithm (isma): Fast subgraph enumeration in large networks using optimized search trees. *PLOS ONE 8*, 4 (04 2013), 1–15.

[7] Dvořák, Z., and Tůma, V. A dynamic data structure for counting subgraphs in sparse graphs. In *Algorithms and Data Structures* (Berlin, Heidelberg, 2013), F. Dehne, R. Solis-Oba, and J.-R. Sack, Eds., Springer Berlin Heidelberg, pp. 304–315.

[8] Eppstein, D., Goodrich, M. T., Strash, D., and Trott, L. Extended dynamic subgraph statistics using h-index parameterized data structures. In *Combinatorial Optimization and Applications* (Berlin, Heidelberg, 2010), W. Wu and O. Daescu, Eds., Springer Berlin Heidelberg, pp. 128–141.

[9] Eppstein, D., and Spiro, E. S. The h-index of a graph and its application to dynamic subgraph statistics. *Journal of Graph Algorithms and Applications 16*, 2 (2012), 543–567.

[10] Hanauer, K. Algora core (version 1.1.), 2020.

[11] Hanauer, K., Henzinger, M., and Hua, Q. C. Fully dynamic four-vertex subgraph counting. *arXiv preprint arXiv:2106.15524* (2021).

[12] Hanauer, K., Henzinger, M., and Hua, Q. C. Fully dynamic four-vertex subgraph counting. In *1st Symposium on Algorithmic Foundations of Dynamic Networks, SAND 2022, March 28-30, 2022, Virtual Conference* (2022), J. Aspnes and O. Michail, Eds., vol. 221 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 18:1–18:17.

[13] Hočevar, T., and Demšar, J. A combinatorial approach to graphlet counting. *Bioinformatics 30*, 4 (12 2014), 559–565.

[14] Huan, J., Wang, W., and Prins, J. Efficient mining of frequent subgraphs in the presence of isomorphism. In *Third IEEE International Conference on Data Mining* (2003), pp. 549–552.

[15] Itzhack, R., Mogilevski, Y., and Louzoun, Y. An optimal algorithm for counting network motifs. *Physica A: Statistical Mechanics and its Applications 381* (2007), 482–490.

[16] Kara, A., Ngo, H. Q., Nikolic, M., Olteanu, D., and Zhang, H. Counting triangles under updates in worst-case optimal time, 2018.

[17] Kiss, G. R., Armstrong, C., Milroy, R., and Piper, J. An associative thesaurus of english and its computer analysis. *The computer and literary studies* (1973), 153–165.

[18] Mishra, S. K. Test-case-generator. https://github.com/snehm/Test-Case-Generator, 2017.

[19] Ortmann, M., and Brandes, U. Efficient orbit-aware triad and quad census in directed and undirected graphs. *Applied network science 2*, 1 (2017), 1–17.

[20] Ribeiro, P., Paredes, P., Silva, M. E. P., Aparicio, D., and Silva, F. A survey on subgraph counting: Concepts, algorithms, and applications to network motifs and graphlets. *ACM Comput. Surv. 54*, 2 (mar 2021).

[21] Sahu, S., Mhedhbi, A., Salihoglu, S., Lin, J., and Özsu, M. T. The ubiquity of large graphs and surprising challenges of graph processing. *Proc. VLDB Endow. 11*, 4 (dec 2017), 420–431.

[22] Singh, H., and Sharma, R. Role of adjacency matrix & adjacency list in graph theory. *International Journal of Computers & Technology 3*, 1 (2012), 179–183.

[23] Skarupke, M. Flat hash map. https://github.com/skarupke/flat_hash_map, 2017.

[24] Skarupke, M. A new fast hash table in response to google's new fast hash table. Probably Dance, https://probablydance.com/2018/05/28/a-new-fast-hash-table-in-response-to-googles-new-fast-hash-table/, May 2018.

[25] Zaki, A., Attia, M., Hegazy, D., and Amin, S. Comprehensive survey on dynamic graph models. *International Journal of Advanced Computer Science and Applications 7* (02 2016).