UNIVERSITY OF CALIFORNIA
SANTA CRUZ

THE ROLE OF DEGENERACY IN REAL-WORLD SUBGRAPH COUNTING

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Noujan Pashanasangi

December 2021

The Dissertation of Noujan Pashanasangi
is approved:

Professor C. Seshadhri, Chair

Professor Yang Liu

Professor Daniele Venturi

Peter Biehl
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

The Role of Degeneracy in Real-World Subgraph Counting

by

Noujan Pashanasangi

Many real-world phenomena are modeled by large graphs. Subgraph counting, the problem of counting occurrences of small target pattern graphs in large input graphs is a fundamental algorithmic task in network analysis. Subraph counting has been extensively studied in both theory and practice and has found applications in areas such as network analysis, social sciences, and bioinformatics.

Graph orientation techniques for subgraph counting based on vertex orderings such as degeneracy ordering is a classical idea. These techniques have inspired many recent practical subraph counting algorithms. In this thesis we analyze the role of graph orientation and degeneracy in subgraph counting, both in theory and practice. Based on these techniques, we present efficient algorithms for getting local subgraph counts (orbits counts) of all 5-vertex patterns, and counting triangles in temporal networks.

In modern applications, input graphs are large and one desires (near) linear time algorithms. We focus on the case where the input graph is in the class of bounded degeneracy graphs. This is a rich class of sparse graphs that is practically relevant as real-world graphs such as social networks have been shown to have low degeneracy. We consider the problem of counting all connected subgraphs with k vertices, and determine for what values of k this problem is solvable in linear time, assuming a standard conjecture in fine-grained complexity. We also give a clean characterizations of all subgraph patterns whose homomorphisms could be counted in near linear time in bounded degeneracy graphs.

To my family

Acknowledgments

I would first like to sincerely thank my advisor, Prof. C. Seshadhri, for his continued advice, guidance, and encouragement. This thesis would have been impossible without his mentorship and support. I would also like to thank my other reading committee members, Prof. Yang Liu and Prof. Daniele Venturi for their valuable feedback and comments. I am grateful to Prof. David Helmbold for insightful discussions and his useful comments.

I would like to express my gratitude to my collaborators, especially my co-author Suman Bera for the many discussions that we had. I was fortunate to work with him and have learnt a lot from him. I also would like to thank Akul Goyal for helpful discussions and his collaboration. I like to thank Shweta Jain and Andrew stolman for being supportive colleagues. I am grateful for all my friends at UCSC who have helped me get to this point.

Finally, I wholeheartedly thank my wife for her love and patience. I am grateful to her for believing in me and encouraging me every day. I am grateful to my sister and my brother-in-law who have supported me from the first day of my Ph.D. journey. Last but definitely not least, I would like to express my deepest gratitude to my parents for their unconditional love and support throughout my entire life.

Chapter 1

Introduction

Many real-world phenomena could be modeled as a set of discrete objects and a set of connections between certain pairs of them. Graphs are the result of mathematical abstraction of such representations. For example, in a social network, nodes represent users and an edge represent the connection between two users. Examples of real-world graphs include citation networks, transaction networks, protein-protein interaction networks, and web graphs. Modern real-world networks are large (tens of millions of edges), but they have structural properties that help us analyze them.

Consider a social network such as Facebook. One interesting structural property of such a network is how often two friends of a user are friends with each other. This parameter is called the *clustering coefficient* [189]. To compute the clustering coefficient of a network we need the counts of triangles, set of three pairwise connected vertices. Indeed, social networks as well as many other types of real-world graphs have a significantly higher number of triangles (and other motifs) than random graphs [118]. Thus, the counts of triangles captures a structural property of these large graphs. Triangle counting is a fundamental tool in network analysis, and there has been a rich line of work on counting triangles in graphs [10, 12, 79, 100, 161]. The triangle counts appear in form of different parameters such as clustering coefficient [189] and *transitivity ratios* [188]. Triangle counting has applications in social networks analysis [135], indexing graph databases [88], community discovery [127], and spam detection [16].

A more general problem is the problem of subgraph counting that ask for the counts of occurrences of small target pattern graphs in a large input host graph. This problem is also referred to as pattern counting, motif analysis, and graphlet analysis. Subgraph counting is a fundamental algorithmic problem in network analysis with a rich line of work in both theory [12, 38, 43, 54, 79, 121, 123, 181] and practice [17, 36, 41, 76, 77, 88, 138, 139, 140, 153, 154, 173, 179]. Subgraph counting has found applications in areas such as bioinformatics and biological networks [77, 139, 140], social networks [62, 124, 165, 171, 179], social sciences [36, 41, 76, 138], community and dense subgraph detection [17, 154, 173, 175], and many other applications [7, 15, 60, 67, 190]. (Refer to the tutorial [159] for more details on applications.).

Subgraph counting can appear in various forms based on the pattern graphs we are interested in counting. Let H be the pattern graph and G be the input host graph. We denote the problem of counting copies of H in G by $\textsc{sub-cnt}_H$. Counting each type of pattern has its own challenges. Counting special patterns such as cliques and cycles have received a lot of attention in the history of subgraph counting. A common version of subgraph counting is to count the frequency of all connected subgraphs with k vertices [8, 55, 57, 75, 83, 110, 125, 137, 141, 187]. We will denote this problem as $\textsc{sub-cnt}_k$. Recently there has been many exact and approximate algorithms for $\textsc{sub-cnt}_k$.

The type of the occurrence of the target pattern is also an important factor of the subraph counting problem. A non-induced subgraph is obtained by taking a subset of edges of a graph. An induced subgraph of a graph is a subset of its vertices and all edges among them. The mapping of interest from the target pattern graph H to the "occurrence" in the host graph G is also of importance. An injective edge preserving map $f : V(H) \to V(G)$ corresponds to the common notion of the subgraph. We call such a mapping an *embedding*, and the occurrence a *match* of the pattern graph. If we lift the constraint of being an injection for the edge preserving map $f : V(H) \to V(G)$, then we get a *homomorphism* of the pattern graph H (an H-homomorphism). We use $\text{Hom}_H(G)$ to denote the count of the distinct H-homomorphisms in G. $\textsc{hom-cnt}_H$ denotes the problem of obtaining the counts of H-homomorphism.

The problem of computing $\text{Hom}_H(G)$ for various choices of H is a deep subfield of study in graph algorithms [12, 32, 34, 35, 43, 45, 47, 50, 63, 79, 106, 107, 152]. Homomorphism counting has numerous applications in logic, properties of graph products, partition functions in statistical physics, database theory, and network science [32, 35, 37, 50, 46, 129, 137]. Many practical and theoretical algorithms for subgraph counting are based on homomorphism counting. Subgraph counts can be expressed as a linear combination of homomorphism counts [43]. It is known that the problems $\textsc{sub-cnt}_H$ and $\textsc{hom-cnt}_H$ is $\#W[1]$-hard when parameterized by k (even when H is a k-clique), so we do not expect $n^{o(k)}$ algorithms for general H [45]. Yet the n^k barrier can be beaten when H has structure. Notably, Curticapean-Dell-Marx proved that if H has treewidth at most 2, then $\text{Hom}_H(G)$ can be computed in $\text{poly}(k) \cdot n^\omega$, where ω is the matrix multiplication constant [43].

In its typical description, subgraph counting asks for the total counts of a pattern graph in the host graph. But, in many applications we need *local* counts of a pattern, also referred to as graphlet distributions, orbit counts, or k-profiles. Local counts is a much finer grained description of the graph, and can be used to

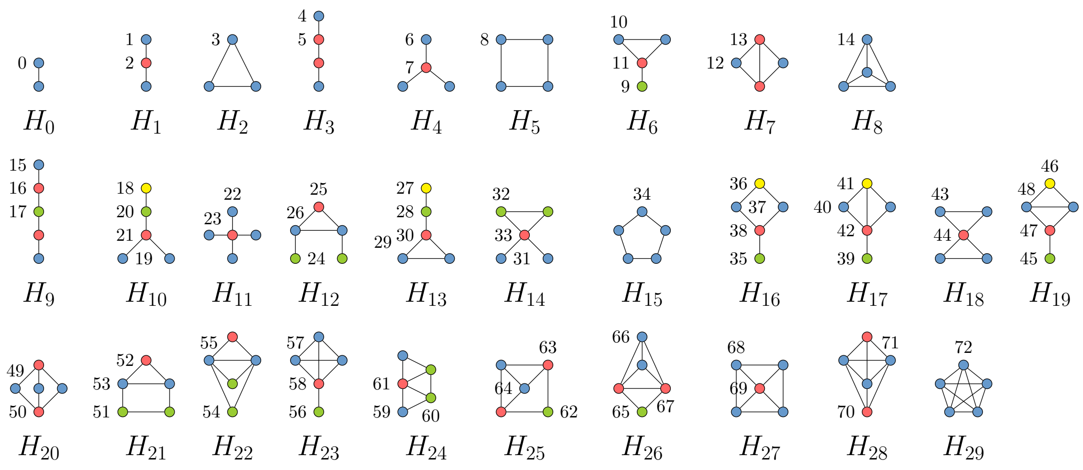


Figure 1.1: All vertex orbits for 5-vertex patterns. Within any pattern, vertices of the same color form an orbit.

generate features for vertices. A compelling application of these local counts are the *graphlet kernel*, where local counts are used to construct vector representations of vertices for machine learning [162]. Similar to the description of SUB-CNT$_k$, many applications require local counts for all pattern subgraphs with at most k vertices. Orbit counts are the more informative version of local counts. Fig. 1.1 shows all connected subgraphs with at most 5 vertices and within each pattern, vertices are present in different "roles" or *orbits*. In some patterns like the 5-cycle (H_{15}) and 5-clique (H_{29}), there is just one orbit. In contrast, H_{10} has four different orbits, indicated by the different colors. Thus, a vertex of G can participate in a copy of H_{10} in four different ways, and we wish to determine all of these four different counts.

Another factor that could also introduce new varieties of subgraph counting is the type of the input graph. Much of the rich history of subgraph counting algorithms focus on counting patterns in static graphs. But many real-world graph are essentially *temporal*, meaning that every edge has an associated timestamp [61, 64, 96]. The example of these networks include but are not limited to communication networks, email networks, transaction networks, and social interaction networks. To model these networks, we can use directed *temporal networks*, where each edge has a timestamp, instead of static networks. For example, a directed edge in a directed temporal network modeling an email network could represent an email from a sender to the receiver where the timestamp of the edge represents the time of the email.

Recently, there has been significant interest in temporal triangle and motif counting algorithms [30, 33, 104, 108, 134, 168, 176, 177, 185]. Temporal triangle counts provide a far richer set of counts than standard counts. These counts take into account the temporal ordering of edges in a triangle, and potentially impose constraints on the timestamp difference among edges. Temporal triangle and motif counting has applications in graph representation learning [177], expressivity of graph neural networks (GNNs) [33], network classification [176], temporal text network analysis [184], computer networks [180], and brain networks [49].

Subgraph counting is extremely challenging due to *combinatorial explosion*. For example, the counts of 5-vertex patterns in graphs with a few million edges can be in the order of billions to trillions [8, 84, 137]. In all applications that use any variety of subgraph counting, it is essential to have efficient algorithms.

There is a rich line of theoretical work on getting $n^{\mu k}$ time algorithms, for $\mu < 1$, using matrix multiplication and tree decomposition methods [12, 28, 29, 43, 44, 79, 95, 97, 123, 182]. From an application standpoint, these algorithms are typically not practical, and do not provide algorithmic guidance. $\textsc{sub-cnt}_H$ when parametrized by $|V(H)| = k$ is #W[1]-hard, so it is not beleived that $f(k) \cdot n^{o(k)}$ algorithms exist. Real-world graphs are large and one typically desires (near) linear-time algorithms in most modern applications. An approach around this is to search for efficient algorithms for restricted graph classes that correspond to real-world graphs. We focus on the class of bounded degeneracy graphs, a rich class of sparse graphs including all minor-closed families, preferential attachment graphs, and bounded expansion graphs. *Degeneracy* of a graph G is the smallest integer k such that every subgraph of G has a vertex of degree at most k and is a standard sparsity measure. Graph degeneracy appears heavily in network science, and it has been empirically shown that real-world graphs have low degeneracy [19, 22, 66, 81, 163].

Graph orientation based on vertex orderings such as degree ordering and degeneracy ordering is a classic and central idea in many subgraph counting algorithms, pioneered by Chiba-Nizhizeki [38]. In a seminal result, Chiba-Nishizeki proved that k-cliques could be counted in $O(m\kappa^{k-2})$ where m is the number of edges of G and κ is its degeneracy. They also gave a $O(m\kappa)$ algorithm for 4-cycle counting [38]. There have been many practical subgraph counting algorithms based on the techniques of Chiba-Nishizeki [81, 83, 125, 137].

In this thesis, we make progress on understanding the role of degenracy in subgraph counting, both in practice and theory. We present efficient and scalable subgraph counting algorithms based on graph orientation and degeneracy ordering. We also analyze the theoretical complexity of subgraph counting in the class of bounded degeneracy graphs and whether or not linear time algorithms are possible.

1.1 Main Questions and Challenges

The main questions we address in this thesis are asked towards a better theoretical understanding of subgraph counting and the role of graph orientations and degeneracy ordering and to improve the state of the art of practical subgraph counting algorithms.

1.1.1 Subgraph Counting in Bounded Degeneracy Graphs

The problems of $\textsc{sub-cnt}_k$ for $k \leqslant 5$ have been successfully tackled in practice using approaches pioneered by Chiba-Nishizeki [8, 125, 137]. These algorithms are often tailored for k (using, for example, specific tricks to count individual 4-vertex subgraphs) and it is not clear how far they will extend for larger k.

Towards a better theoretical understanding, we pose the following question.

For what values k, does the $\textsc{sub-cnt}_k$ problem admit a linear time algorithm in bounded degeneracy graphs?

Next, we focus on the problem of homomorphism counting. In chapter 3, we prove that a near-linear time algorithm is possible for subgraph and homomorphism counting when $|V(H)| \leqslant 5$. In a significant generalization, Bressan [34] defines an intricate notion of *DAG treewidth*, and shows (among other things) that a near-linear time algorithm exists when the DAG treewidth of H is one. These results lead us to the following question.

Can we characterize the pattern graphs H for which $\mathrm{Hom}_H(G)$ is computable in near-linear time (when G has bounded degeneracy)?

1.1.2 Vertex Orbit Counting

There are efficient algorithms for getting subgraph counts for all patterns with up to 5 vertices [137]. These algorithms use clever counting methods to avoid enumerations, so in spite of the combinatorial explosion, they are efficient. But these techniques are tailored for getting global counts in the input graph G. There has been recent work on randomized methods for local counting, but these require large parallel hardware even for graphs with tens of millions of edges [58]. To the best of our knowledge, there is no algorithm that (even approximately) computes all orbit counts for all 5-vertex paterns, for all vertices of G, even for graphs with tens of millions of edges. For simplicity, we refer to these counts as 5-vertex orbit counts (5-VOCs). Results on global counting are much faster, but it is not clear how to implement these ideas for orbit counting [8, 137]. The ORCA package [75] is the only algorithm that actually computes all 5-VOCs, but it does not terminate after days for graphs with tens of millions of edges. The total number of orbit counts is easily in the order of trillions, and a fast algorithm should ideally avoid touching each 5-vertex subgraph in G. On the other hand, orbit counts are an extremely fine-grained statistic, so purely global methods do not work. We pose the following question.

Can we give an algorithm for getting all 5-VOCs that avoid expensive enumeration and scales to large input graphs?

1.1.3 Temporal Triangle Counting

Counting temporal triangles in (directed) temporal networks introduces new challenges to that of triangle counting in static graphs. The first challenge is actually defining types of temporal triangles (or motifs). In essence, all definitions specify constraints on the time difference between edges of a triangle. For example, Kovanen et al. [96] restricted temporal triangles to cases where the gap between two consecutive edges in the temporal ordering is at most Δ time units, and the two edges incident to each node are consecutive event of that node. Paranjape-Benson-Leskovec (henceforth PBL) introduced *δ-temporal triangles*, where all edges of the triangle/motif have to occur within δ timesteps [128]. These varying definitions necessitate different algorithms. Our first motivating question is whether one can design algorithms for a more general notion of temporal triangles.

Secondly, there is a significant gap between the best static triangle counting algorithms and temporal triangle counters. Specifically, the classic and immensely practical triangle counting algorithm of Chiba-Nishizeki runs in time $O(m\kappa)$. The current state-of-the-art temporal triangle counting algorithm of PBL runs in $O(m\sqrt{\tau})$ time, where τ is the total triangle count (of the underlying static graph). There is a large gap between κ (which is typically in the hundreds and thought of as a constant) and τ (which is superlinear in m).

These twin issues motivate our study on temporal triangle counting.

Can we define a more general notion of temporal triangles, and give an algorithm whose asymptotic running time is closer to that of static triangle counting?

1.2 Results and Contributions

In this thesis, we theoretically characterize patterns countable in near-linear time in bounded degeneracy graphs. We also present practical tools for getting 5-VOCs and temporal triangle counting. The main results of Chapter 3, Chapter 4, Chapter 5, and Chapter 6 were published in ITCS 2020, SODA 2021, WSDM 2020, and KDD 2021, respectively.

1.2.1 Linear Time Subgraph Counting and The Chasm at Size Six

We prove that for $k < 6$, the problem of $\textsc{sub-cnt}_k$ in bounded degeneracy graphs can be solved in linear time. More than the counting algorithm for k-vertex patterns where $k < 6$, our main contribution is the structural decompositions of the subgraphs that lead to these results. This decomposition also sheds light on why certain k-vertex subgraphs, for $k > 6$, do not seem to have any efficient algorithms in bounded degeneracy graphs. Assuming TRIANGLE DETECTION CONJECTURE, a standard conjecture in fine-grained complexity, we prove that for all $k \geqslant 6$, $\textsc{sub-cnt}_k$ cannot be solved even in near-linear time. This paper [20] was published in ITCS 2020.

1.2.2 The Barrier of Long Induced Cycles

There is a rich history of *complexity dichotomies* for homomorphism detection and counting problems [45, 50, 68, 72, 152]. In this work, we discover such a dichotomy for near-linear time algorithms for homomorphism counting in bounded degeneracy graphs.

We give a surprisingly clean characterization for pattern graphs H for which $\mathrm{Hom}_H(G)$ is computable in near-linear time assuming G has bounded degeneracy. Let m denote the number of edges in G. We prove the following: if the largest induced cycle in H has length at most 5, then there is an $O(m \log m)$ algorithm for counting H-homomorphisms in bounded degeneracy graphs. If the largest induced cycle in H has length at least 6, then assuming TRIANGLE DETECTION CONJECTURE there is a constant $\gamma > 0$, such that there is no $o(m^{1+\gamma})$ time algorithm for counting H-homomorphisms. This paper [21] was published in SODA 2021.

1.2.3 Counting Vertex Orbits of All 5-vertex Subgraphs

We present `EVOKE`, a scalable algorithm that can determine vertex orbits counts for all 5-vertex pattern subgraphs. `EVOKE` can process graphs with tens of millions of edges within an hour on a commodity machine, is typically hundreds of times faster than previous state-of-the-art algorithms, and gets results on datasets beyond the reach of previous methods.

Theoretically, we generalize a recent "graph cutting" framework [137] to get vertex orbit counts. This framework generate a collection of polynomial equations relating vertex orbit counts of larger subgraphs to those of smaller subgraphs. EVOKE carefully exploits the structure among these equations to rapidly count. We prove and empirically validate that EVOKE only has a small constant factor overhead over the best (total) 5-vertex subgraph counter. This paper [129] was published in WSDM 2020.

1.2.4 Generalized Temporal Triangle Counting

In this work, we define $(\delta_{1,3}, \delta_{1,2}, \delta_{2,3})$-temporal triangles, a general temporal triangle notion, that allows for separate time constraints for all pairs of edges of the triangle. $\delta_{1,3}$ specifies the maximum gap allowed between the first and third edges in the temporal ordering of the edges of the triangle. $\delta_{1,2}$ and $\delta_{2,3}$ specify the maximum time difference between the timestamps of the first and second edge in the temporal ordering and the second and third edge, respectively. Our main result is a new algorithm, DOTTT (Degeneracy Oriented Temporal Triangle Totaler), that exactly counts all directed variants of $(\delta_{1,3}, \delta_{1,2}, \delta_{2,3})$-temporal triangles. Using the classic idea of degeneracy ordering with careful combinatorial arguments, we can prove that DOTTT runs in $O(m\kappa \log m)$ time, where m is the number of (temporal) edges of the input graph and κ is the graph degeneracy of the underlying static graph of the input graph. Up to log factors, this matches the running time of the best static triangle counters. DOTTT has excellent practical behavior and runs twice as fast as existing state-of-the-art temporal triangle counters (and is also more general). For example, DOTTT computes all types of temporal queries in a Bitcoin temporal network with half a billion edges in less than an hour on a commodity machine. This paper [130] was published in KDD 2021.

Chapter 2

Preliminaries

In this chapter we present definitions and notations common to subgraph counting algorithms and theoretical hardness results presented in this thesis. Generally, the input graph is denoted by $G = (V(G), E(G))$ and the target pattern graph is denoted by $H = (V(H), E(H))$. Both G and H are considered to be simple, undirected, and connected graphs, unless stated otherwise.

In both our theoretical results and practical subgraph counting tools, we heavily use directed graphs and acyclic orientations of graphs. In a directed graph, the out-neighborhood and in-neghborhood of a vertex u is denoted by $N_G^+(u)$ and $N_G^-(u)$, respectively. We use $d_G^+(u)$ and $d_G^-(u)$ to denote the out-degree and in-degree of u. We denote an acyclic orientation of a simple graph H by $H^{\rightarrow}$.

2.1 Degeneracy and Vertex Ordering

A graph G is *k-degenerate* if each of its non-empty subgraphs has minimum vertex degree of at most k. The *degeneracy* of a graph G is the smallest integer k such that G is k-degenerate. We denote the degeneracy of G by $\kappa(G)$. Another graph sparsity measure that is closely related to degeneracy is *arboricity*. The *arboricity* of a graph G, denoted as $\alpha(G)$, is the smallest integer k such that the edge set $E(G)$ can be partitioned into k forests. When the graph G is clear from the context, we simply write κ, and α, instead of $\kappa(G)$ and $\alpha(G)$. A classic theorem of Nash-Williams shows that the degeneracy and arboricity are closely related. All our results can be stated in terms of either of the parameters.

Theorem 2.1.1. *(Nash-Williams [122]) In every graph G, $\alpha(G) \leqslant \kappa(G) \leqslant 2\alpha(G) - 1$.* □

Vertex ordering is central to many subgraph counting algorithms [20, 38, 81, 125, 129, 137, 178]. In this thesis, we mostly work with the *degeneracy ordering* of G, which is defined as follows.

Definition 2.1.2. *Degeneracy ordering* of a graph G, denoted by $\lhd$, is obtained by repeatedly removing the vertex with minimum degree. The ordering is defined by the removal time.

For example, if $u \lhd v$, then u is removed before v according to the above process. *Degeneracy ordering* can be found in linear time [112].

Using any vertex ordering $\prec$ of an undirected graph G, we construct a directed graph $G^{\rightarrow}_{\prec}$ as follows: for each edge $\{u, v\} \in E(G)$, direct the edge from u to v iff $u \prec v$. We denote this directed edge as (u, v). Observe that $G^{\rightarrow}_{\prec}$ is necessarily acyclic. Prominent examples of vertex orderings used by subgraph counting algorithms are degree ordering and degeneracy ordering. We denote the directed graph obtained from *degeneracy ordering* $\lhd$ as $G^{\rightarrow}_{\lhd}$. Whenever we use '$\rightarrow$' in denoting a graph such as in $G^{\rightarrow}_{\lhd}$ and $H^{\rightarrow}$, the directed graph is a DAG. The following two are folklore results about vertex ordering and degeneracy and can be derived from Prop. 5.2.2 of [48].

Lemma 2.1.3. *For each vertex $v \in G^{\rightarrow}_{\lhd}$, $d^+(v) \leqslant \kappa$.* □

Lemma 2.1.4. *If there exists a vertex ordering $\prec$ of G such that in the corresponding directed graph $G^{\rightarrow}_{\prec}$, $d^{+}(v) \leqslant k$ for each vertex v, then $\kappa(G) \leqslant k$.* □

The technique that connects vertex ordering, graph orientation, and degeneracy to subgraph counting is to count occurrences of H in G, by counting matches of all possible acyclic orientations $H^{\rightarrow}$ of H in $G^{\rightarrow}_{\lhd}$ instead. This classic idea enables the bounded out-degree in $G^{\rightarrow}_{\lhd}$ to play an important role in subgraph counting algorithms.

2.2 Subgraph Counting

In this section, we formally define homomorphism, embedding, and match (copy) of a target subgraph H in the input graph G.

Definition 2.2.1. A homomorphism from H to G is a mapping $\pi : V(H) \rightarrow V(G)$ such that, $\{\pi(u), \pi(v)\} \in E(G)$ for all $\{u, v\} \in E(H)$. If H and G are both directed, then π should preserve the directions of the edges. If π is injective, then it is called an embedding of H in G.

Next, we define a match (also called copy) of H in G.

Definition 2.2.2. A match of H in G is a subgraph of G that is isomorphic to H. If a match of H is an induced subgraph of G, then it is an induced match of H in G.

Observe that each embedding of H in G corresponds to a match of G. More precisely, for each match of H in G there are $|Aut(H)|$ (number of automorphisms of H) many embeddings of H in G that map H to that specific match of H.

Our lower bound results for both the SUB-CNT$_k$ problem and HOM-CNT$_H$ assume the TRIANGLE DETECTION CONJECTURE. Abboud and Williams introduced the TRIANGLE DETECTION CONJECTURE on the complexity of determining whether a graph has a triangle [6]. Assuming there is no $O(m^{1+\gamma})$ time triangle detection algorithm, they proved lower bounds for many classic graph algorithm problems. It is believed that the constant γ could be arbitrarily close to 1/3 [6].

Conjecture 2.2.3 (TRIANGLE DETECTION CONJECTURE [6])**.** *There exists a constant $\gamma > 0$ such that in the word RAM model of $O(\log n)$ bits, any algorithm to detect whether an input graph on m edges has a triangle requires $\Omega(m^{1+\gamma})$ time in expectation.*

Chapter 3

Linear Time Subgraph Counting and the chasm at Size Six

In this chapter we theoretically analyze the complexity of the $\textsc{sub-cnt}_k$ problem in bounded degeneracy graphs. More specifically, we address the following question.

For what values k, does the $\textsc{sub-cnt}_k$ *problem admit a linear time algorithm in bounded degeneracy graphs?*

The question above has a surprisingly clean resolution, assuming conjectures from fine-grained complexity. For simplicity, we assume that the input graph G is connected. We assume Las Vegas randomized algorithms, so we talk of expected running times.

Our main theorem asserts linear time algorithms for counting (up to) 5-vertex subgraphs in bounded degeneracy graphs. For counting 6-vertex subgraphs and beyond, it is unlikely that even near-linear time algorithms exists.

Theorem 3.0.1 (The chasm at size 6)**.** *For $k \leqslant 5$, there is an expected $O(m\kappa^{k-2})$ time algorithm for* $\textsc{sub-cnt}_k$*.*

Assume the Triangle Detection Conjecture *(Conj. 2.2.3). There exists an absolute constant $\gamma > 0$ such that the following holds. For any $k \geqslant 6$ and any function $f : \mathbb{N} \to \mathbb{N}$, there is no (expected) $o(m^{1+\gamma} f(\kappa))$ algorithm for* SUB-CNT$_k$.

3.1 Main Ideas

Conditional Lower Bounds It is instructive to look at the conditional lower bounds. The reduction of triangle detection to subgraph counting in bounded degeneracy graphs is actually quite simple. Suppose we want to detect (or even count) triangles in an input graph G. Get graph G' by subdividing each edge into two, so a triangle in G becomes a $\mathcal{C}_6$ (6-cycle) in G'. But the degeneracy of G' is just 2! (In any induced subgraph of G', the minimum degree is at most 2, proving the bound.) Thus, if there exists $o(f(\kappa)m^{1+\gamma})$ time algorithms for counting 6-cycles, that would violate the Triangle Detection Conjecture.

It is fairly straightforward to generalize this idea for larger cycles, by replacing edges in G by short paths. Assuming Triangle Detection Conjecture, for all $k \geqslant 6$ and $k \neq 8$, we can rule out linear time algorithms for counting $\mathcal{C}_k$ in bounded degeneracy graphs. Our reduction does not work for $\mathcal{C}_8$; instead we consider a different subgraph for the case of $k = 8$ ($\mathcal{C}_7$ with a tail). We give the details in Section 3.5.

This reduction fails for counting 5-cycles and in general, it does not work for counting any 5-vertex subgraph. For good reason, as we discovered an efficient algorithm for this problem. This is the more technical part of our paper.

Algorithmic Framework We present an algorithmic framework for solving the SUB-CNT$_k$ problem, that generalizes the core idea of Chiba and Nishizeki [38]. It is known from past work that their ideas basically provide an $O(m\kappa^{k-2})$ algorithm for SUB-CNT$_k$, for $k = 3, 4$. The main challenge is to get such an algorithm for $k = 5$, thereby nailing down the chasm of Theorem 3.0.1. This leads to new results for counting various 5-vertex subgraphs. Perhaps more than these individual results, our main contribution lies in identifying structural decompositions of the pattern subgraphs that allows for efficient algorithms. This decomposition also sheds light on why certain k-vertex subgraphs, for $k \geqslant 6$, does not seem to have any efficient algorithms in bounded arboricity graphs. We give an outline of our framework next, and present it formally in Section 3.4.

The key idea that comes from Chiba-Nishizeki is to perform subgraph counting on $G^{\rightarrow}$, an acyclic orientation of G where the out degree of each vertex is bounded by $O(\kappa)$[1]. The classic clique and 4-cycle counting algorithms enumerate directed stars and directed paths of length 2 to count subgraphs. We note that the algorithm does *not* enumerate 4-cycles, since there can be $\Omega(n^2)$ 4-cycles. It requires clever indexing to solve this problem, which we generalize in our algorithm.

The crucial generalization of this idea is to enumerate directed rooted trees. Specifically, we count occurrences of a connected pattern H by counting occurrences of all possible acyclic orientations (up to isomorphism) $H^{\rightarrow}$ of H in $G^{\rightarrow}$. The main idea is to find the largest directed rooted tree in $H^{\rightarrow}$, with edges directed away from the root. Call this tree T. Since outdegrees in $G^{\rightarrow}$ are bounded, we can efficiently enumerate all copies of T. Any copy of $H^{\rightarrow}$ in $G^{\rightarrow}$ is formed by extending a copy of T, but $H^{\rightarrow}$ may contain vertices that are not in T. Thus, the extensions could be expensive to compute. But when H has at most 5 vertices, we can prove that $H^{\rightarrow} \setminus T$ is itself either a collection of rooted stars or paths. We can create hash tables that store information about the occurrences of the latter. The final count of $H^{\rightarrow}$ is obtained by enumerating T and carefully combining counts

[1]Technically, this is *not* the idea of Chiba-Nishizeki, who use the degree orientation. But it was somewhat of a folklore result that it is easy to get the same result using the degeneracy orientation. Arguably the first such reference is Schank-Wagener [155].

from the hash tables.

3.2 Related Work

Subgraph counting problems has a long and rich history. More than three decades ago, Itai and Rodeh [79] gave the first non-trivial algorithm for the triangle detection and counting problems with $O(m^{3/2})$ runtime. Subsequently, Chiba and Nishizeki [38] gave an elegant algorithm based on the degree based vertex ordering that solves triangle counting, 4-cycle counting and ℓ-clique counting with running times of $O(m\kappa)$, $O(m\kappa)$, and $O(m\kappa^{\ell-2})$ respectively (κ denotes the degeneracy). In comparison, our algorithm exploits the *degeneracy ordering* of the vertices (see Section 3.3 for a formal definition); this enables us to create a uniform framework for any k-vertex subgraph for $k \in \{4, 5\}$. In dense graphs, the best bounds for the clique counting problem are achieved by fast matrix multiplications based algorithms [54, 123]; Vassilevska [181] gave combinatorial algorithm with significantly reduced space requirement. For general subgraphs, there is a rich line of research based on matrix multiplication, tree decomposition and vertex cover methods [12, 28, 29, 43, 44, 79, 95, 97, 123, 182] — these works focus on getting $n^{\mu k}$ time algorithmis, for $\mu < 1$.

Subgraph counting problems, specifically triangle counting, clique counting and cycle counting problems, has also been studied extensively in various Big Data models such as property testing model [13, 51, 52], MapReduce settings [40, 90, 169], and streaming model [9, 14, 18, 82, 86, 87, 109, 113, 133]. Most of these work focuses on an approximate count, rather than an exact count. In the applied world, there are many efficient algorithms that are based on clever sampling techniques [25, 26, 77, 83, 141, 187, 191, 192, 199]. Exact counting has also been studied extensively in the applied world [8, 27, 55, 57, 67, 73, 74, 75, 110, 117, 125, 137, 166]. In particular, Ahmed et al. [8] presented an algorithmic framework for solving the SUB-CNT$_4$ problem, called PGD (Parametrized Graphlet Decomposition), which scales to graphs with tens of millions of edges. Pinar et al. [137] studied the SUB-CNT$_5$ problem, and gave the current state of the art ESCAPE library based on degree ordering techniques. However, the provable runtime of their algorithm for certain 5-vertex subgraphs is quadratic, $O(n^2)$. For a deeper exploration of related applied work, refer to the tutorial on subgraph counting by Seshadhri and Tirthapura [159], and the subgraph counting survey at [144].

The subgraph detection problem, which asks whether an input graph has a copy of the subgraph, is a well-studied problem [11, 12, 79, 89, 97, 121, 194]. For the triangle detection problem, the best known algorithm is based on fast matrix multiplication and it runs in time $O(\min\{n^\omega, m^{2\omega/(\omega+1)}\})$ [12]. If $\omega = 2$, this would give us $O(\min\{n^2, m^{4/3}\})$ algorithm for the triangle detection problem. Hence, to falsify the TRIANGLE DETECTION CONJECTURE, it would require a major breakthrough result in the algorithmic graph theory world. For a more detailed discussion on the TRIANGLE DETECTION CONJECTURE and its implications, refer to the paper by Abboud and Williams [6].

In the subgraph enumeration problem, the goal is to output each occurrences of the target subgraph. Chiba and Nishizeki [38] showed that it is possible to enumerate all the triangles in a graph along with counting the total number of triangles in $O(m\kappa)$ time. For enumerating all the triangles, $O(m\kappa)$ time is effectively optimal assuming the 3SUM CONJECTURE [94, 131]. Eppstein [59] studied the bipartite subgraph enumaration problem in bounded arboricity graphs.

3.3 Preliminaries

In this paper, we study the SUB-CNT$_k$ problem. We consider k to be a constant.For a fixed subgraph H, we use SUB-CNT$_H$ to denote the problem of counting all occurrences of H in the input graph G. When H is the triangle subgraph, we denote the corresponding counting problem as TRI-CNT. In this chapter we abuse notation and overload the word "match" and define it as follows.

Definition 3.3.1. A match of H in G is a bijection $\pi : S \to V(H)$ where $S \subseteq V(G)$ and for any two vertices u and v in S, $\{u, v\} \in E(G)$ if $\{\pi(u), \pi(v)\} \in E(H)$.

Definition 3.3.2. A match of H' in G' is a bijection $\pi : S \to V(H')$ where $S \subseteq V(G')$ and for any ordered pair of vertices (u, v) where u and v are in S, $(u, v) \in E(G')$ if $(\pi(u), \pi(v)) \in E(H')$.

We denote the number of matches of H in G by $\mathrm{M}(G, H)$. An incomplete match of H in G is an injection $\pi : S \to V(H)$ (so $|S| < |V(H)|$), that has the same properties of a match except being surjective. Consider two incomplete matches (injections) of H, $\pi_1 : S_1 \to V(H)$, and $\pi_2 : S_2 \to V(H)$. Let $V_{\pi_1} = \{\pi_1(u) \mid u \in S_1\}$ and $V_{\pi_2} = \{\pi_2(u) \mid u \in S_2\}$. We say that π_2 completes π_1 to be a match of H, when $V(H) = V_{\pi_1} \cup V_{\pi_2}$ (surjective), $V_{\pi_1} \cap V_{\pi_2} = \emptyset$ (injective), and for any two vertices $u \in S_1$ and $v \in S_2$, $\{u, v\} \in E(G)$ if $\{\pi_1(u), \pi_2(v)\} \in E(H)$. In case of directed graphs, it should hold that $(u, v) \in E(G')$ if $(\pi_1(u), \pi_2(v)) \in E(H')$ and $(v, u) \in E(G')$ if $(\pi_2(v), \pi_1(u)) \in E(H')$.

Two matches are distinct if they are not authomorphims of a match. In other words, two matches π_1 and π_2 of H are equivalent, if they map two automorphisms of the exact same subgraph of G to H. We denote the number of distinct matches of H in G by $\mathrm{DM}(G, H)$. In the SUB-CNT$_k$ problem, we are interested in $\mathrm{DM}(G, H)$ for all k-vertex subgraphs H.

3.4 Subgraph Counting Through Graph Orientation and Directed Trees

In this section, we discuss our algorithmic framework for solving the $\textsc{sub-cnt}_k$ problem. Instead of directly counting the number of occurrences of a k-vertex subgraph H in the input graph G, we count the occurrences of all possible DAG $H^{\rightarrow}$ (up to isomorphism) of H in the graph $G^{\rightarrow}_{\triangleleft}$. To achieve this, our main idea is to find the largest directed tree of $H^{\rightarrow}$, enumerate all matches of this tree, and then count matches of the remaining vertices using structures we save in a hash table. In Section 3.4.1, we show that our framework solves the $\textsc{sub-cnt}_5$ problem in expected $O(m\kappa^3)$ time. In Section 3.4.2, we demonstrate the limitation of our framework as it fails to solve the $\textsc{sub-cnt}_{\mathcal{C}_6}$ problem *efficiently*.

Algorithm 1 Counting distinct matches of all 5-vertex subgraphs in G ($\textsc{sub-cnt}_5$)

```
1: procedure Count-All-5($G$)
2:     Derive $G^{\rightarrow}_{\triangleleft}$ by orienting $E(G)$ with respect to degeneracy ordering.
3:     for all connected 5-vertex subgraphs $H$ except 4-star do
4:         Run Count-Match($G^{\rightarrow}_{\triangleleft}, H$) and save the result for $H$.
5:     Save $\sum_{u \in V(G)} \binom{d(u)}{4}$ for 4-star.
```

3.4.1 5-vertex Subgraph Counting

Our main algorithmic result is given in the following theorem.

Theorem 3.4.1. *There is an algorithm that solves the* $\textsc{sub-cnt}_5$ *problem in* $O(m\kappa^3)$ *time.*

Our strategy is to count matches of all possible DAGs (up to isomorphism) $H^{\rightarrow}$ of H in $G^{\rightarrow}_{\triangleleft}$, to obtain the number of distinct matches of H in G. Alg. 2

Algorithm 2 Counting distinct matches of H in G (SUB-CNT$_H$)

```
1:  procedure COUNT-MATCH($G^{\rightarrow}_{\triangleleft}, H$)
2:      DM$(G, H) \leftarrow 0$
3:      for all possible DAGs (up to isomorphism) $H^{\rightarrow}$ of $H$ do
4:          M$(G^{\rightarrow}_{\triangleleft}, H^{\rightarrow}) \leftarrow 0$
5:          Find one of the largest DRTSs in $H^{\rightarrow}$, and call it $T_{\max}$.
6:          for all match $\pi$ of $T_{\max}$ in $G^{\rightarrow}_{\triangleleft}$ do
7:              if $\pi$ is a match of $H^{\rightarrow}$ then      ▷ $V(T_{\max}) = V(H^{\rightarrow})$. Lemma 3.4.6
8:                  M$(G^{\rightarrow}_{\triangleleft}, H^{\rightarrow}) \leftarrow$ M$(G^{\rightarrow}_{\triangleleft}, H^{\rightarrow}) + 1$
9:              else if $\pi$ is an incomplete match of $H^{\rightarrow}$ then      ▷ Lemma 3.4.6
10:                 $k \leftarrow$ number of ways to complete $\pi$ to a match of $H^{\rightarrow}$.      ▷ Lemma 3.4.8
11:                 M$(G^{\rightarrow}_{\triangleleft}, H^{\rightarrow}) \leftarrow$ M$(G^{\rightarrow}_{\triangleleft}, H^{\rightarrow}) + k$
            DM$(G, H) \leftarrow$ M$(G^{\rightarrow}_{\triangleleft}, H^{\rightarrow}) / |Aut(H^{\rightarrow})|$
12:     return DM$(G, H)$
```

demonstrates this subroutine of our algorithm for SUB-CNT$_5$, which is shown in Alg. 1. First, we find one of the largest directed rooted tree subgraphs (*DRTS*), which we define as follows, in $H^{\rightarrow}$.

Definition 3.4.2. Given any directed graph D, a directed rooted tree subgraph (DRTS) of D, is a subgraph T of D, where the underlying undirected graph of T is a rooted tree, and edges are oriented away from the root in T.

The following lemma shows that we can find all matches of any DRTS in $H^{\rightarrow}$ in the desired time.

Lemma 3.4.3. *Let T be a directed tree with k vertices. All matches of T in $G^{\rightarrow}_{\triangleleft}$ can be enumerated in $O(m\kappa^{k-2})$.*

Proof. Let $t_1, \ldots, t_k$ be a BFS ordering of T starting at the root t_1. Fix an edge $(u, v) \in E(G^{\rightarrow}_{\triangleleft})$ and map u to t_1 and v to t_2. There are m possible matches for (t_1, t_2), which we can find by enumerating the edges of $G^{\rightarrow}_{\triangleleft}$. Now, we will choose vertices to map to $t_3, \ldots, t_k$, one by one, in this order. Since the out-degree of each vertex in $G^{\rightarrow}_{\triangleleft}$ is at most κ, if we have already mapped vertices to

$t_1, \ldots, t_i$, there are at most κ vertices that could be mapped to t_{i+1}. Therefor $\mathrm{M}(G^{\rightarrow}_{\lhd}, T) = O(m\kappa^{k-2})$, and we can enumerate all of them by first choosing (u, v) to map to (t_1, t_2) and then choosing vertices to map to $t_3, \ldots, t_k$, in this order and one by one. □

Observation 3.4.4. *Call a vertex v of a directed graph a* source *vertex, if $d^-(v) = 0$. Consider T to be one of the largest DRTSs of a DAG D. T has to have a source vertex of D as the root, otherwise the root has an in-neighbor v, which is not in T as it would create a cycle. Adding v to T creates a new DRTS which has one more vertex than T. This contradicts the fact that T is one of the largest DRTSs of D. Hence, the root of T has to be a source vertex of D.*

Given a 5-vertex DAG $H^{\rightarrow}$, we can find a DRTS that has the most number of vertices among all DRTSs of $H^{\rightarrow}$ in constant time. First, find all source vertices, and then apply a Breath First Search (BFS) starting from each of these vertices and pick a BFS tree with the most number of vertices among all. The following lemma shows that the largest DRTS has at least 3 vertices for a 5-vertex connected subgraphs, except 4-star. Notice that, the largest DRTS of a 4-star with all the edges oriented towards the center has two vertices.

Lemma 3.4.5. *Let H be a connected undirected 5-vertex graph that is not a 4-star. Each largest DRTS of any DAG $H^{\rightarrow}$, which is an acyclic orientation of H, has at least three vertices.*

Proof. We prove this lemma by contradiction. Assume that any DRTS of $H^{\rightarrow}$ has at most two vertices. A directed 2-path, or any vertex with at least two outgoing edges result in a DRTS with three vertices. Therefore,

(a) $H^{\rightarrow}$ does not have a 2-path,

(b) each vertex in $H^{\rightarrow}$ has at most one outgoing edges.

Notice that, since $H^{\rightarrow}$ is a DAG, it has at least one source vertex. Consider a source vertex u. Since H is connected, u has at least one neighbor, and by (b) it should have exactly one neighbor. Let $N^{+}(u) = \{v\}$, then $N^{+}(v) = \emptyset$, by (a). So, v should have at least one incoming neighbor w. By (a), w has no incoming edges, and it has no outgoing edges by (b). Call the other two vertices x and y. As H is connected, there should be a connection between $\{u, v, w\}$ and $\{x, y\}$. u and w cannot have any neighbor other than v, so x and y could only be connected to v. Since H is not a star, there should be an edge between x and y. Without loss of generality, let (x, y) be that edge. By (a), $(y, v) \notin E(H^{\rightarrow})$ and by (b) $(x, v) \notin E(H^{\rightarrow})$. So, $\{u, v, w\}$ is not connected to $\{x, y\}$, and H is disconnected, which is a contradiction. Thus, the assumption that any DRTS of $H^{\rightarrow}$ has at most two vertices is wrong, and each largest DRTS of $H^{\rightarrow}$ has at least three vertices. $\square$

So far, we know that we can find one of the largest DRTSs of H, which has at least 3 vertices. We use $T_{\max}$ to denote this DRTS. By Lemma 3.4.3, we can enumerate all matches of $T_{\max}$ in $G^{\rightarrow}_{\triangleleft}$ in $O(m\kappa^3)$ time. For each such match, we need to validate whether it is a (incomplete) match of $H^{\rightarrow}$ or not. If it is not, then it could not be completed to a match of $H^{\rightarrow}$. The following lemma shows that we can perform this validation efficiently. In the remaining part of this section, "constanct expected time", refers to constant amortized time access to hash maps that we use.

Lemma 3.4.6. *Let T be a DRTS of a DAG $H^{\rightarrow}$ of a connected k-vertex graph H. Assume edges of $G^{\rightarrow}_{\triangleleft}$ are saved in a hash table. For each match π of T in $G^{\rightarrow}_{\triangleleft}$, it takes $O(|E(H^{\rightarrow})|)$ expected time to validate whether π is a (incomplete) match of $H^{\rightarrow}$ or not.*

Proof. Since π is a bijection, it has an inverse which we denote by π^{-1}. Let

$H^{\rightarrow}[V(T)]$ denote the subgraph of $H^{\rightarrow}$ induced on $V(T)$. Observe that, there could be edges in $H^{\rightarrow}[V(T)]$ not present in T. For π to be a match (if $V(T) = V(H^{\rightarrow})$) or incomplete match of $H^{\rightarrow}$, these edges have to be present between corresponding vertices in $G^{\rightarrow}_{\lhd}$ mapped to T by π. Formally, consider all ordered pairs of vertices $(a,b) \in V(T) \times V(T)$ such that $(a,b) \in E(H^{\rightarrow})$ and $(a,b) \notin E(T)$, π is a match or incomplete match of $H^{\rightarrow}$ iff $(\pi^{-1}(a), \pi^{-1}(b)) \in E(G^{\rightarrow}_{\lhd})$ for all such pairs of vertices. To validate this, we enumerate all edges (a,b) of $H^{\rightarrow}[V(T)]$ which are not present in T, and search for $(\pi^{-1}(a), \pi^{-1}(b))$ in hashed edges of $G^{\rightarrow}_{\lhd}$ in expected constant time. So this only requires $O(|E(H^{\rightarrow})|)$ expected time. $\square$

If $V(T_{\max}) = V(H^{\rightarrow})$, then a match of $T_{\max}$ could be a match of $H^{\rightarrow}$ too, which could be verified as explained. If there is a vertex in $H^{\rightarrow}$ which is not present in $T_{\max}$, then after validating that a match of $T_{\max}$ is an incomplete match of $H^{\rightarrow}$, we need to find the number of ways to complete it to a match of $H^{\rightarrow}$. For this we need to count matches of each possible structures that $T_{\max}$ does not cover in $H^{\rightarrow}$. We save the count of these structures in $G^{\rightarrow}_{\lhd}$, in hash tables. The following lemma shows that this can be done efficiently.

Lemma 3.4.7. *In $O(m\kappa^3)$ time and space, we can save all the following key and value pairs in hash maps $\mathcal{HM}_1$, $\mathcal{HM}_2$, and $\mathcal{HM}_3$.*

1. $\mathcal{HM}_1 : ((u,v),1)$ *where* $(u,v) \in E(G^{\rightarrow}_{\lhd})$
2. $\mathcal{HM}_2 : (S,k)\ \forall S \subseteq V(G^{\rightarrow}_{\lhd})$ *where* $1 \leqslant |S| \leqslant 4$, *and k is the number of vertices u such that* $S \subseteq N^{+}(u)$
3. $\mathcal{HM}_3 : ((S_1,S_2),\ell)\ \forall S_1, S_2 \subseteq V(G^{\rightarrow}_{\lhd})$, *where* $1 \leqslant |S_1 \cup S_2| \leqslant 3$, *and ℓ is the number of edges* $e = (u,v) \in E(G^{\rightarrow}_{\lhd})$ *such that* $S_1 \subseteq N^{+}(u)$ *and* $S_2 \subseteq N^{+}(v)$.

Proof. We show how to enumerate and save all these structures in $\mathcal{HM}_1$, $\mathcal{HM}_2$, and $\mathcal{HM}_3$.

1. $\mathcal{HM}_1$: We can easily do this in $O(m)$ by enumerating the out-neighbors of each vertex

2. $\mathcal{HM}_2$: For each edge $e = (u, v)$, we can enumerate all subsets T of the set $\{w \in N^+(u) \mid v \lhd w\}$, where $|T| \leqslant 3$, in $O(\kappa^3)$ time, and increment the value for the key $T \cup \{v\}$ in the hash map by one.

3. $\mathcal{HM}_3$: For each edge $e = (u, v)$ $(v \in N^+(u))$, we enumerate all possible subset $S_1 \subseteq N^+(u) \setminus \{v\}$ where $|S_1| \leqslant 3$. And, for each S_1 we enumerate all possible $S_2 \setminus S_1$ in subsets of $N^+(v)$, such that $1 \leqslant |S_1 \cup S_2| \leqslant 3$. This takes $O(\kappa^3)$ as the out-degree of each vertex is at most κ, and we choose up to three vertices. All possible $S_1 \cap S_2$ can be determined by checking the connection between v and each vertex in S_1 using the hashed edges of $G^{\rightarrow}$ in $\mathcal{HM}_1$. □

The following lemma shows that we can count the number of ways to complete a match of $T_{\max}$, which is also an incomplete match of $H^{\rightarrow}$, to a match of $H^{\rightarrow}$ efficiently.

Lemma 3.4.8. *Let H be a 5-vertex connected graph, $H^{\rightarrow}$ be a DAG of H, and $T_{\max}$ be one of the largest DRTSs in $H^{\rightarrow}$. Assume $\mathcal{HM}_1$, $\mathcal{HM}_2$, and $\mathcal{HM}_3$ are given. For each match π of $T_{\max}$ in $G^{\rightarrow}_{\lhd}$ which is an incomplete match of $H^{\rightarrow}$, we can count the number of ways to complete π to a match of $H^{\rightarrow}$ in expected constant time.*

Proof. By Lemma 3.4.5, $T_{\max}$ has at least 3 vertices, and since π is an incomplete match (not a match) of $H^{\rightarrow}$, we can assume that $|V(T_{\max})| < 5$. Observe that, $T_{\max}$ is a maximal DRTS. Any vertex in $H^{\rightarrow}$ which is not in $T_{\max}$ can only be connected to vertices of $T_{\max}$ by outgoing edges, otherwise they could be added to

$T_{\max}$ to create a larger DRTS of $H^{\rightarrow}$, which contradicts the maximality of $T_{\max}$. We consider two cases where $T_{\max}$ has three or four vertices.

Let $|V(T_{\max})| = 4$, and i be the only vertex in $H^{\rightarrow}$ that is not in $T_{\max}$. To complete π to a match of $H^{\rightarrow}$, we need to choose a vertex in $G^{\rightarrow}_{\triangleleft}$, that is connected by outgoing edges to vertices mapped to the out-neighborhood of i in $H^{\rightarrow}$. Let $S_i = \{\pi^{-1}(t) \mid t \in N^{+}_{H^{\rightarrow}}(i)\}$. $\mathcal{HM}_2(S_i)$ is the number of vertices that could be mapped to i, but some of them may be already mapped to a vertex in $T_{\max}$, by π. Let r_i denote the number of vertices $v \in \{\pi^{-1}(t) \mid t \in V(T_{\max})\}$, where $S_i \subseteq N^{+}_{G^{\rightarrow}_{\triangleleft}}(v)$. We can obtain r_i in expected constant time, by enumerating vertices mapped to $V(T_{\max})$, and counting vertices that are connected to all vertices in S_i. For any vertex, we can check the connection to each vertex of S_i using $\mathcal{HM}_1$ in expected constant time. The number of ways to complete π to a match of $H^{\rightarrow}$ in this case is $\mathcal{HM}_2(S_i) - r_i$.

Now we consider the case where $|V(T_{\max})| = 3$. Let $V(H^{\rightarrow}) \setminus V(T_{\max}) = \{i, j\}$. To complete π to a match of $H^{\rightarrow}$, we only need to choose two vertices of $G^{\rightarrow}_{\triangleleft}$ to map to i and j. Let $S_i = \{\pi^{-1}(t) \mid t \in V(T_{\max}) \cap N^{+}_{H^{\rightarrow}}(i)\}$ and $S_j = \{\pi^{-1}(t) \mid t \in V(T_{\max}) \cap N^{+}_{H^{\rightarrow}}(j)\}$. We consider two cases, where i and j are connected or not. If they are connected, without loss of generality, assume $(i, j) \in E(H^{\rightarrow})$. If $(i, j) \in E(H^{\rightarrow})$, then we can use $\mathcal{HM}_3$ in Lemma 3.4.7, to find the number of edges (u, v) where u and v could be mapped to i and j, respectively. Let $r_{(i,j)}$ be the number of edges $e = (w, x) \in E(G^{\rightarrow}_{\triangleleft})$, where w and x are mapped to vertices in $T_{\max}$ by π, such that, $S_i \subseteq N^{+}_{G^{\rightarrow}_{\triangleleft}}(w)$, and $S_j \subseteq N^{+}_{G^{\rightarrow}_{\triangleleft}}(x)$. We can obtain $r_{(i,j)}$ in expected constant time using $\mathcal{HM}_1$. Then the number of edges (u, v) that could be mapped to (i, j) is $\mathcal{HM}_3((S_i, S_j)) - r_{(i,j)}$. Next case is when $(i, j) \notin E(H^{\rightarrow})$. In this case, we use $\mathcal{HM}_2$ to find the number of pair of vertices of $G^{\rightarrow}_{\triangleleft}$ which could be mapped to i and j. Let r_i (r_j resp.)

denote the number of vertices $v \in V(G^{\rightarrow}_{\triangleleft})$ where v is mapped to a vertex in $T_{\max}$ and $S_i \subseteq N^{+}_{G^{\rightarrow}_{\triangleleft}}(v)$ ($S_j \subseteq N^{+}_{G^{\rightarrow}_{\triangleleft}}(v)$ resp.). Also, we use $r_{i,j}$ to denote the number of vertices $v \in V(G^{\rightarrow}_{\triangleleft})$ that are counted in both r_i and r_j, meaning $S_i \cup S_j \subseteq N^{+}_{G^{\rightarrow}_{\triangleleft}}(v)$. We can obtain r_i, r_j, and $r_{i,j}$ easily in expected constant time using $\mathcal{HM}_1$. The number of pairs of vertices which could be mapped to i and j is equal to $(\mathcal{HM}_2(S_i) - r_i) \cdot (\mathcal{HM}_2(S_j) - r_j) - (\mathcal{HM}_2(S_i \cup S_j) - r_{i,j})$. □

Now, we have all the tools to efficiently count distinct matches of a DAG of $H^{\rightarrow}$ in $G^{\rightarrow}_{\triangleleft}$. The following lemma shows that we can do this in $O(m\kappa^3)$ expected time.

Lemma 3.4.9. *There is an algorithm which counts distinct matches for each possible DAG (up to isomorphism) $H^{\rightarrow}$ of a 5-vertex connected subgraphs H, in $O(m\kappa^3)$ expected time.*

Proof. Fix a DAG $H^{\rightarrow}$ of H. If H is a 4-star and $H^{\rightarrow}$ has ℓ incoming neighbors, then the number of distinct matches of $H^{\rightarrow}$ is $\sum_{u \in V(G^{\rightarrow}_{\triangleleft})} \binom{d^{-}(u)}{\ell}\binom{d^{+}(u)}{4-\ell}$. Assume that H is not a 4-star. Find a DRTS of $H^{\rightarrow}$ with the most number of vertices among all its DRTSs, and call it $T_{\max}$. This can be done in constant time for $H^{\rightarrow}$. By Lemma 3.4.5, $T_{\max}$ has at least three vertices. We will now enumerate all matches of $T_{\max}$ in $G^{\rightarrow}_{\triangleleft}$. By Lemma 3.4.3, this step requires $O(m\kappa^3)$ expected time. For each match π of $T_{\max}$ in $G^{\rightarrow}_{\triangleleft}$, we can verify whether π is a match (if ($|V(T_{\max})| = 5$) or incomplete match of $H^{\rightarrow}$ in expected constant time, by Lemma 3.4.6. If $|V(T_{\max})| = 5$, while enumerating all matches of $T_{\max}$, we only count them if they are a match of $H^{\rightarrow}$. So in this case we can count $\mathrm{M}(G^{\rightarrow}_{\triangleleft}, H^{\rightarrow})$ in $O(m\kappa^3)$ expected time.

Otherwise, $T_{\max}$ has 3 or 4 vertices. In this case, for each match π of $T_{\max}$, we first verify that it is also an incomplete match of $H^{\rightarrow}$. Then, we count the number of ways to complete π to a match of $H^{\rightarrow}$, which we can do in expected constant time, by Lemma 3.4.8. To obtain $\mathrm{M}(G^{\rightarrow}_{\lhd}, H^{\rightarrow})$, we simply sum the ways to complete each incomplete match we have found, to a match of $H^{\rightarrow}$.

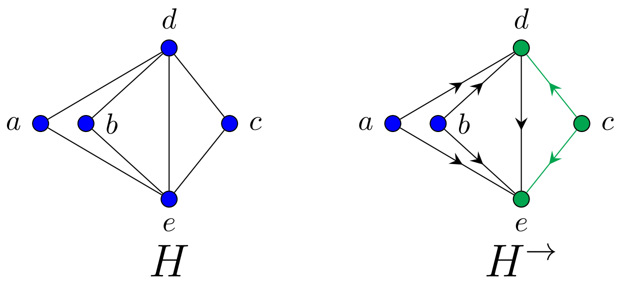
This approach gives us the number of all (not necessarily distinct) matches of $H^{\rightarrow}$ in $G^{\rightarrow}_{\lhd}$. Let $H^{\rightarrow}_{\pi}$ be a subgraph of $G^{\rightarrow}_{\lhd}$ that π maps to $H^{\rightarrow}$. Each automorphism of $H^{\rightarrow}$, gives a new match π' which is not distinct from π, as it is still mapping H_{π} (the same copy of H) to $H^{\rightarrow}$ (example in Fig. 3.1b). As each match of $H^{\rightarrow}$, also maps vertices to $T_{\max}$, resulting in a match of $T_{\max}$ and an (incomplete) match of $H^{\rightarrow}$, we will find all distinct matches of $H^{\rightarrow}$ and count each one exactly $|Aut(H^{\rightarrow})|$ times. We want the number of distinct matches, which we can obtain by dividing the count of all matches by $|Aut(H^{\rightarrow})|$.

Thus, it requires $O(m\kappa^3)$ expected time to create $\mathcal{HM}_1$, $\mathcal{HM}_2$, and $\mathcal{HM}_3$ by Lemma 3.4.7, $O(m\kappa^3)$ time for enumerating matches of $T_{\max}$, expected constant time to validate these matches, and expected constant time for counting ways to complete each such match, that is verified to be an incomplete match of $H^{\rightarrow}$, to a match of $H^{\rightarrow}$. So overall, we can find $\mathrm{DM}(G^{\rightarrow}_{\lhd}, H^{\rightarrow})$ in $O(m\kappa^3)$ expected time.

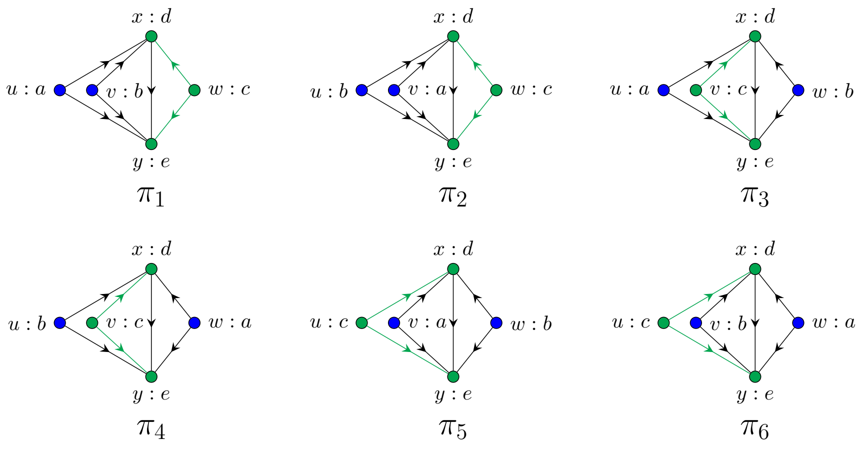
This completes the proof of this lemma. □

Lastly, we can prove Theorem 3.4.1 as follows.

Proof of Theorem 3.4.1. Given a 5-vertex connected subgraph H, we can count all distinct matches of each possible DAG $H^{\rightarrow}$ of H, in $G^{\rightarrow}_{\lhd}$ in $O(m\kappa^3)$ expected time, by Lemma 3.4.9. To count all distinct matches of H in G, we just need to sum the number of distinct matches of all possible DAGs (up to isomorphism) of H. The number of such DAGs is constant for H. There are 21 different connected 5-vertex subgraphs (illustrated in [137]), and we perform this process on all of them. This completes the proof of the theorem. □



(a) H is 5-vertex connected subgraph and $H^{\rightarrow}$ is one possible acyclic orientation of it. $T_{\max}$ (largest DRTS of $H^{\rightarrow}$) is shown in green and contains three vertices.



(b) All six figures show exactly the same subgraph in $G^{\rightarrow}_{\triangleleft}$. $\pi_1, \ldots, \pi_6$ are six equivalent matches of $H^{\rightarrow}$ in $G^{\rightarrow}_{\triangleleft}$, one for each automorphism of $H^{\rightarrow}$. Notice (u, v, w) being mapped to all permutations of (a, b, c).

Figure 3.1: Application of Alg. 2 on a DAG $H^{\rightarrow}$ of an example 5-vertex connected subgraph H.

3.4.2 Limitations of Our Framework for a Six Vertex Subgraph

Consider $\mathcal{C}_6$, shown as H in Fig. 3.2. Then $H^{\rightarrow}$, shown in the right side of Fig. 3.2, is a possible DAG of H. In $H^{\rightarrow}$, s_1, s_2, and s_3 are the source vertices, and t_1, t_2, and t_3 are the sink vertices. Any DRTS of $H^{\rightarrow}$ has at most three vertices, and there are three such DRTS, T_1, T_2, and T_3 rooted at s_1, s_2 and s_3, respectively. T_1 is shown by red in Fig. 3.2. For each of T_1, T_2, and T_3, the remaining vertices

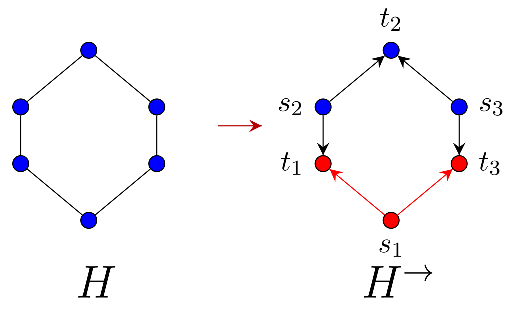


Figure 3.2: Let $H^{\rightarrow}$ be a DAG of H ($\mathcal{C}_6$). Considering any largest DRTS of $H^{\rightarrow}$, the remaining vertices include a vertex with two incoming edges (in-in wedge). Even graphs with bounded degeneracy can have $\Omega(n^2)$ in-in wedges. So hashing in Alg. 2 will not be bounded by m and κ for H.

include a vertex, with two incoming edges, which we call an in-in wedge. For example, t_2 is such a vertex for T_1. Even graphs with bounded degeneracy can have $\Omega(n^2)$ in-in wedges. We cannot hash the count of such structures in expected time bounded by m and κ. So, Alg. 2 fails to count occurrences of $\mathcal{C}_6$ in the desired time. In the next section, we discuss why such limitations are natural to any framework for the SUB-CNT$_k$ problem at and beyond $k = 6$.

3.5 A Chasm at Six

At the end of the previous section, we showed the limitations of our framework in counting certain 6-vertex subgraphs. In this section, we show that perhaps such limitations are fundamental to any subgraph counting algorithms. In particular, the landscape of SUB-CNT$_k$ problem in the bounded degeneracy graphs changes dramatically as we move beyond $k = 5$. We prove that for every integer $k \geqslant 6$, there exists a k-vertex subgraph H such that, the running time of any algorithm for the SUB-CNT$_H$ problem does not depend on the degeneracy of the input graph, assuming the TRIANGLE DETECTION CONJECTURE. In contrast, for $k \leqslant 5$, $O(m\kappa^{k-2})$ algorithms exists for SUB-CNT$_k$ (see Section 3.4). The following theorem captures the main result of this section.

Theorem 3.5.1. *Assume the* Triangle Detection Conjecture *(Conjecture 2.2.3). There exists an absolute constant $\gamma > 0$ such that the following holds. For any $k \geqslant 6$ and any function $f : \mathbb{N} \to \mathbb{N}$, there exists a k-vertex subgraph H such that there is no (expected) $o(m^{1+\gamma} f(\kappa))$ algorithm for* SUB-CNT$_H$.

Outline of the Proof For each $k \geqslant 6$ and $k \neq 8$, the subgraph of interest will be the k-cycle graph, $\mathcal{C}_k$. For $k = 8$, the subgraph of interest will be the $\mathcal{C}_7$ with a tail (see Figure 3.3). We first give a proof outline. Fix some $k \geqslant 6$ and let H_k denote the target subgraph of size k. Recall the TRI-CNT problem — count the number of triangles in a graph with m edges. Conjecture 2.2.3 asserts that for any algorithm $\mathcal{A}$ for the TRI-CNT problem, $T(\mathcal{A}) = \omega(m)$ where $T(\mathcal{A})$ denotes the worst case time complexity of the algorithm $\mathcal{A}$. Our strategy is to reduce from the TRI-CNT problem to the SUB-CNT$_{H_k}$ problem. To this end, we construct a new graph G_k from the input instance G of the TRI-CNT problem such that G_k has $O(m)$ edges, and has degeneracy at most 2. More importantly, the number of triangles in G is a simple linear function of the number of H_k in G_k. Hence, we can derive the number of triangles in G by counting the number of H_k in G_k. As $\kappa(G_k) \leqslant 2$, any $O(mf(\kappa))$ algorithm for the SUB-CNT$_{H_k}$ problem translates to a $O(m)$ algorithm for the TRI-CNT problem, contradicting the Triangle Detection Conjecture. We remark that, for $k = 8$, our proof strategy will be slightly different — instead of reducing from the TRI-CNT problem, we shall reduce from the triangle detection problem itself. However, the gadget construction will follow the same basic principle.

The construction of G_k from G is rather simple. The details of the construction depends on whether k is a multiple of 3 or not. We take two examples to describe the construction.

First, we take $k = 6$, and the target subgraph $H_6 = \mathcal{C}_6$. For each edge e in $E(G)$, we replace e with a length two path $\{e_1, e_2\}$ in $E(G_6)$. To accomplish this, we add a new vertex v_e for each edge: $V(G_6) = V(G) \cup \{v_e\}_{e \in E(G)}$. This is shown in Figure 3.4a. Each triangle in G creates a $\mathcal{C}_6$ in G_6. We formally prove in Lemma 3.5.3 that the number of triangles in G is same as the number of $\mathcal{C}_6$ in G_6. In Lemma 3.5.2, we bound the degeneracy of G_6 by 2. This construction can be generalized for any $k = 3\ell$ where $\ell \geqslant 2$, by replacing each edge in $E(G)$ with ℓ-length path.

Next consider the case $k = 7$. For each edge $e \in E(G)$, we first create two parallel copies of e, and then replace the first one with a length two path $\{e_{1,1}$, $e_{1,2}\}$, and the second one with a length three path $\{e_{2,1}$, $e_{2,2}$, $e_{2,3}\}$. So in $E(G_7)$, we have 5 edges for each edge in $E(G)$. We create 3 new vertices per edge to accomplish this, and denote them as v_e, u_{e_1}, u_{e_2}. See Figure 3.4b for a pictorial demonstration. In Lemma 3.5.3, we argue that the number of $\mathcal{C}_7$ is exactly 3 times the number of triangles in G. In Lemma 3.5.2, we bound the degeneracy of G_7 by 2. This construction generalizes to any $k = 3\ell + i$ where $\ell \geqslant 2$ and $i \in \{1, 2\}$ (except for the case when $k = 8$, that is $\ell = 2$ and $i = 2$) by splitting each edge into ℓ and $\ell + 1$ many parts respectively.

Finally, we consider the case of $k = 8$. Note that the target subgraph H_8 is the 7-cycle with a tail in this case (see Figure 3.3). It is natural to wonder why do we not simply take $H_8 = \mathcal{C}_8$? After all, for all other values of k, taking $H_k = \mathcal{C}_k$ suffices. At a first glance, it seems like if we consider the same graph G_7 as described above (and in Figure 3.4b) the number of $\mathcal{C}_8$ would be a simple linear function of the number of triangles in G — for each triangle in G, there will be exactly three $\mathcal{C}_8$ in G_7. However, each $\mathcal{C}_4$ in G would also lead to a $\mathcal{C}_8$ in G_8. Observe that for $k > 8$, we do not run into this problem. A more formal treatment of this issue appear in Section 3.5.

So instead, we take H_8 to be the subgraph $\mathcal{C}_7$ with a tail to prove our conditional lower bound for SUB-CNT$_8$. The construction of the graph G_8 remains exactly the same as that of G_7. We show in Lemma 3.5.4 that, there exists a $\mathcal{C}_7$ with a tail in G_8 if and only if there exist a triangle in G.

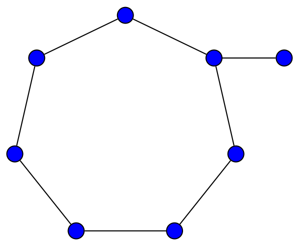
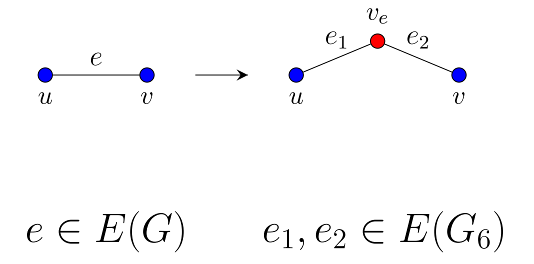


Figure 3.3: Target subgraph for proving conditional lower bounds for SUB-CNT$_8$: the $\mathcal{C}_7$ with a tail

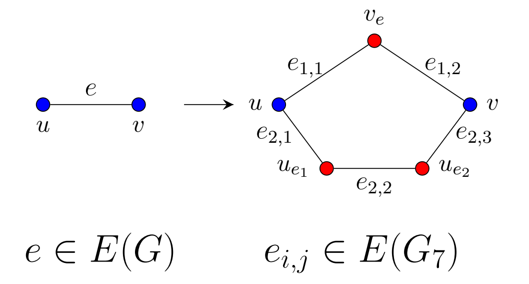
We now present the proof of Theorem 3.5.1 in full details.

Proof of Theorem 3.5.1. Fix some $k \geqslant 6$. Let the subgraph H_k denote the target subgraph of size k. For $k \neq 8$, H_k is $\mathcal{C}_k$, and for $k = 8$, H_k is $\mathcal{C}_7$ with a tail (see Figure 3.3). We reduce from the TRI-CNT problem to the SUB-CNT$_{H_k}$. Let $G = (V, E)$ be the input instance for the TRI-CNT problem with $|V| = n$ and $|E| = m$. We construct an input instance $G_k = (V_k, E_k)$ for the SUB-CNT$_{H_k}$ problem from G. The construction of G_k differs based on whether k is divisible by 3 or not. We next consider these two cases separately.

Details of the Reduction. First assume $k = 3\ell$ for some integer $\ell \geqslant 2$. We first define the vertex set V_k. For each vertex in V, we add a vertex in V_k. For each edge $e \in E$, we add a set of $\ell - 1$ many vertices, denoted as $V_e = \{v_{e,1}, v_{e,2}, \ldots, v_{e,\ell-1}\}$. We collect all these second type of vertices into the



(a) Construction of the edge set $E(G_6)$ from the edge set $E(G)$. The red colored nodes are only present in $V(G_6)$, and not in $V(G)$.



(b) Construction of the edge set $E(G_7)$ from the edge set $E(G)$. The red colored nodes are only present in $V(G_7)$, and not in $V(G)$.

Figure 3.4: Reduction from the TRI-CNT problem to the SUB-CNT$_{\mathcal{C}_k}$ problem for $k = 6$ (left) and $k = 7$ (right).

set V_E. Formally, we have

$$V_k = V \cup V_E \,,$$
$$\text{where } V_E = \bigcup_{e \in E} V_e \,,$$
$$\text{for } V_e = \{v_{e,1}, v_{e,2}, \ldots, v_{e,\ell-1}\} \,.$$

We now describe the edge set E_k. We treat each edge $e = \{u, v\} \in E$ as an ordered pair (u, v) where the ordering can be arbitrary of the vertices (for example, assume lexicographical ordering). Now for each edge $e = (u, v)$ construct an ℓ-length path between u and v in V_k by connecting the vertices in $\{u\} \cup V_e \cup \{v\}$ sequentially. More precisely, we define E_k as follows.

$$E_k = \bigcup_{e \in E} E_e \,,$$
$$\text{where } E_e = \{\{u, v_{e,1}\}, \{v_{e,1}, v_{e,2}\}, \ldots, \{v_{e,\ell-2}, v_{e,\ell-1}\}, \{v_{e,\ell-1}, v\}\} \text{ for } e = (u, v) \,.$$

This completes the construction of the graph $G_k = (V_k, E_k)$. We give an example in Figure 3.4a for $k = 6$.

Now assume $k = 3\ell + i$ for some some integer $\ell \geqslant 2$ and $i \in \{1, 2\}$. In the previous case, we added a set of $\ell-1$ many vertices for each edge in E. But now, for each edge $e \in E$, we add two sets of vertices, one with $\ell - 1$ many vertices and the other with ℓ many vertices. We denote the first set as $V_e = \{v_{e,1}, v_{e,2}, \ldots, v_{e,\ell-1}\}$, and the second set as $U_e = \{u_{e,1}, u_{e,2}, \ldots, u_{e,\ell}\}$. We also add the set of vertices in V to V_k. Formally, we have

$$\begin{aligned} V_k &= V \cup V_E\,, \\ \text{where } V_E &= \bigcup_{e \in E} V_e \cup U_e\,, \\ \text{for } V_e &= \{v_{e,1}, v_{e,2}, \ldots, v_{e,\ell-1}\}\,, \\ \text{and } U_e &= \{u_{e,1}, u_{e,2}, \ldots, u_{e,\ell}\}\,. \end{aligned}$$

To construct the edge set E_k, as before we treat each edge in $e = \{u, v\} \in E$ as an ordered pair (u, v) according to some arbitrary ordering of the vertices. Now, for each edge $e = (u, v)$, construct an $2\ell + 1$-length cycle between u and v in V_k by creating a ℓ-length path via the vertices in V_e and another $\ell + 1$-length path via the vertices in U_e. We denote the corresponding edge sets as $E_{V,e}$ and $E_{U,e}$ respectively. Formally, we define E_k as follows.

$$\begin{aligned} E_k &= \bigcup_{e \in E} (E_{V,e} \cup E_{U,e})\,, \\ \text{where } E_{V,e} &= \{\{u, v_{e,1}\}, \{v_{e,1}, v_{e,2}\}, \ldots, \{v_{e,\ell-2}, v_{e,\ell-1}\}, \{v_{e,\ell-1}, v\}\}\,, \\ \text{and } E_{U,e} &= \{\{u, u_{e,1}\}, \{u_{e,1}, u_{e,2}\}, \ldots, \{u_{e,\ell-1}, u_{e,\ell}\}, \{u_{e,\ell}, v\}\} \text{ for } e = (u, v)\,. \end{aligned}$$

This completes the construction of the graph $G_k = (V_k, E_k)$. Note that the construction is independent of the value of i. Hence, we produce the same graph G_k for $k = 3\ell + 1$ and $k = 3\ell + 2$. We give an example in Figure 3.4b for $k = 7$.

Note that although our target subgraph for the case $k = 8$ is a 7-cycle with a tail instead of 8-cycle, our construction is still the same.

Correctness of the Reduction In Lemma 3.5.2, we prove that G_k has degeneracy at most 2. In Lemma 3.5.3, we show that, for $k \neq 8$, the number of $\mathcal{C}_k$ in the graph G_k is a linear function of the number of triangles in G. In Lemma 3.5.4, we show that G_8 is H_8 free if and only if G is triangle free.

Lemma 3.5.2. *$\kappa(G_k) \leqslant 2$.*

Proof. To prove the lemma it is sufficient to exhibit a vertex ordering $\prec$ such that in the corresponding directed graph $G^{\rightarrow}_{\prec}$, $d^+(v) \leqslant 2$ for all $v \in V_k$ (application of Lemma 2.1.4). We use an ordering $\prec$ where $V_E \prec V$ and the ordering within each set is arbitrary. Observe that each vertex $v \in V_E$ has degree exactly 2 and no two vertices in V are connected to each other. Hence, $d^+(v) \leqslant 2$ for all $v \in V_k$. $\square$

Lemma 3.5.3. *Let $\ell \geqslant 2$ be some integer. For $k = 3\ell$, $\mathrm{DM}(G_k, \mathcal{C}_k) = \mathrm{DM}(G, \mathcal{C}_3)$. For $k = 3\ell + i$ with $i \in \{1, 2\}$ and $k \neq 8$, $\mathrm{DM}(G_k, \mathcal{C}_k) = 3 \cdot \mathrm{DM}(G, \mathcal{C}_3)$.*

Proof. Let $\mathcal{T}$ be the set of triangles in G and $\mathcal{C}$ be the set of $\mathcal{C}_k$ in G_k. Note that a triangle in $\mathcal{T}$ and a k-cycle in $\mathcal{C}$ can be uniquely identified by a set of three and k edges, respectively.

We first take up case of $k = 3\ell$ for some $\ell \geqslant 2$. Let g be the mapping between the sets $\mathcal{T}$ and $\mathcal{C}$, $g : \mathcal{T} \to \mathcal{C}$, defined as follows: $g(\{e_1, e_2, e_3\}) = E_{e_1} \cup E_{e_2} \cup E_{e_3}$. To prove the lemma, it is sufficient to exhibit that g is a bijection. To this end, note that if $g(\tau_1) = g(\tau_2)$, then $\tau_1 = \tau_2$. This follows immediately from the definition of g, since $E_{e_1} \cap E_{e_2} = \emptyset$ for all $e_1 \neq e_2$. We now show that every k-cycle in $\mathcal{C}$ has an inverse mapping in g. Let ξ be a k-cycle in $\mathcal{C}$. Fix some edge $e \in E$. By construction, either all the edges from the set E_e are present in ξ, or none of them are. Hence, ξ must be of the form $E_{e_1} \cup E_{e_2} \cup E_{e_3}$ for some three distinct edges e_1, e_2, and e_3. Clearly, $\{e_1, e_2, e_3\}$ forms a triangle in G.

Now assume $k = 3\ell + i$ for some $\ell \geqslant 2$ and $i \in \{1, 2\}$, and $k \neq 8$. It is not difficult to see that each triangle in $\mathcal{T}$ leads to exactly three k-cycles in $\mathcal{C}$. The non-trivial direction is to show that for each k-cycles in $\mathcal{C}$ there is an unique triangle in $\mathcal{T}$. Let ξ be a k-cycle in $\mathcal{C}$. Fix some edge $e \in E$. By construction, exactly one of the following must be true: (i) all the ℓ edges from the set $E_{V,e}$ are present in ξ, (ii) all the $\ell + 1$ edges from the set $E_{U,e}$ are present in ξ, (iii) none of the edges from the set $E_{V,e} \cup E_{U,e}$ are present in ξ. First assume $i = 1$. Since ξ has $3\ell + 1$ many edges, and $\ell \geqslant 2$, it must consist of one $E_{U,e}$ set of size $\ell + 1$, and two $E_{V,e}$ sets of size ℓ. When $i = 2$ and $\ell > 2$, ξ must consist of two $E_{U,e}$ set of size $\ell + 1$, and one $E_{V,e}$ sets of size ℓ. Clearly, the three edges corresponding to these sets form a unique triangle in G. (When $k = 8$, that is $\ell = 2$ and $i = 2$, taking four distinct sets $E_{V,e}$ creates a copy of $\mathcal{C}_8$, and hence the argument does not work.) □

Lemma 3.5.4. *The input graph G is triangle free if and only if G_8 does not have any $\mathcal{C}_7$ with a tail.*

Proof. Observe that, if there exists a triangle τ in G, then in G_8, there would be at least one $\mathcal{C}_7$ with a tail (in fact, the exact number would depend on the degree of the involved vertices). In the proof of Lemma 3.5.3, we argued that each 7-cycle in G_7 (which is isomorphic to G_8) corresponds to a triangle in G. Also, by our construction, if G_8 has a $\mathcal{C}_7$, then that 7-cycle necessarily has a tail. Therefore, existence of $\mathcal{C}_7$ with a tail in G_8 implies existence of a triangle in G. This completes the proof of the lemma. □

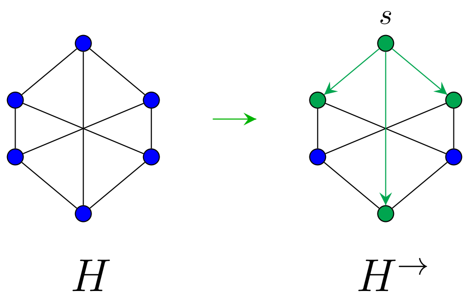


Figure 3.5: Alg. 2 succeeds to count the number of distinct matches of H in linear time for bounded (constant) degeneracy graphs. Each acyclic orientation of H has a source vertex s, which is connected to exactly three vertices, as in $H^{\rightarrow}$. So, the largest DRTS has at least four vertices (shown in green). Number of matches of the remaining vertices (shown in blue) could be counted using $\mathcal{HM}_2$

Lemmas 3.5.2 to 3.5.4 together prove the theorem: if there exists an algorithm $\mathcal{A}$ for the SUB-CNT$_{\mathcal{C}_k}$ problem with $T(\mathcal{A}) = O(mf(\kappa))$, then $\mathcal{A}$ is an algorithm for the TRI-CNT problem (or the triangle detection problem in the case of $k = 8$) with $T(\mathcal{A}) = O(m)$, where $T(\mathcal{A})$ denotes the worst case time complexity of the algorithm $\mathcal{A}$. $\square$

3.6 Future Directions

Although our algorithmic framework fails to produce a linear time algorithm for SUB-CNT$_{\mathcal{C}_6}$ in bounded degeneracy graphs, there are certain other 6-vertex subgraphs where it indeed succeeds. An easy example is SUB-CNT$_{\mathcal{K}_6}$. In fact, our framework gives a linear time algorithm for counting any constant size clique in bounded degeneracy graphs — for each acyclic orientation of a clique, the source vertex construct a DRTS covering all the remaining vertices. There exists other non-clique 6-vertex subgraphs as well, where Alg. 2 succeeds. Consider the subgraph H shown in Fig. 3.5. It is easy to see that, any acyclic orientation of H such as $H^{\rightarrow}$ has at least one source vertex s that is a root of a DRTS with four vertices. Thus, we can solve SUB-CNT$_H$ in $O(m\kappa^3)$ expected time.

Despite the chasm at six, there exist subgraphs H with 6-vertices (or more) such that SUB-CNT$_H$ admits a linear time algorithm in bounded degeneracy graph. We end this exposition with the following natural problem:

Characterize all subgraphs H such that SUB-CNT$_H$ *has a linear time algorithm in bounded degeneracy graphs.*

Chapter 4

The Barrier of Long Induced Cycles

The main result of this chapter is a surprisingly clean resolution of the following problem, assuming fine-grained complexity results.

Can we characterize the pattern graphs H for which $\mathrm{Hom}_H(G)$ is computable in near-linear time (when G has bounded degeneracy)?

Let $\mathrm{LICL}(H)$ be the length of the largest induced cycle in H.

Theorem 4.0.1. *Let G be an input graph with n vertices, m edges, and degeneracy κ. Let $f : \mathbb{N} \to \mathbb{N}$ denote some explicit function. Let $\gamma > 0$ denote the constant from the* Triangle Detection Conjecture*.*

If $\mathrm{LICL}(H) \leqslant 5$: there exists an algorithm that computes $\mathrm{Hom}_H(G)$ in time $f(\kappa) \cdot m \log n$.

If $\mathrm{LICL}(H) \geqslant 6$: assume the Triangle Detection Conjecture*. For any function $g : \mathbb{N} \to \mathbb{N}$, there is no algorithm with (expected) running time $g(\kappa)o(m^{1+\gamma})$ that computes $\mathrm{Hom}_H(G)$.*

(We note that the condition on H involves induced cycles, but we are interested in counting non-induced homomorphisms.)

4.1 Main Ideas

Background for the Upper Bound. We begin with some context on the main algorithmic ideas used for homomorphism/subgraph counting in bounded degeneracy graphs. Any graph G of bounded degeneracy has an acyclic orientation $G^{\rightarrow}$, where all outdegrees are bounded. Moreover, $G^{\rightarrow}$ can be found in linear time [112]. For any pattern graph H, we consider all possible acyclic orientations. For each such orientation $H^{\rightarrow}$, we compute the number of $H^{\rightarrow}$-homomorphisms (in $G^{\rightarrow}$). (Directed homomorphisms are maps that preserve the direction of edges.) Finally, we sum these counts over all acyclic orientations $H^{\rightarrow}$. This core idea was embedded in the seminal paper of Chiba-Nishizeki, and has been presented in such terms in many recent works [20, 34, 125, 137].

Since $G^{\rightarrow}$ has bounded outdegrees, for any bounded, rooted tree $T^{\rightarrow}$ (edges pointing towards leaves), *all* $T^{\rightarrow}$-homomorphisms can be explicitly enumerated in linear time. To construct a homomorphism of $H^{\rightarrow}$, consider the rooted trees of a DFS forest $T_1^{\rightarrow}, T_2^{\rightarrow}, \ldots$ generated by processing the sources first. We first enumerate all homomorphisms of $T_1^{\rightarrow}, T_2^{\rightarrow}, \ldots$ in linear time. We need to count how many tuples of these homomorphisms can be "assembled" into $H^{\rightarrow}$-homomorphisms. (We note that the number of $H^{\rightarrow}$-homomorphisms can be significantly super-linear.) The main idea is to index the rooted tree homomorphisms appropriately, so that $H^{\rightarrow}$-homomorphisms can be counted in linear time. This requires a careful understanding of the shared vertices among the rooted DFS forest $T_1^{\rightarrow}, T_2^{\rightarrow}, \ldots$.

The previous work of the authors showed how this efficient counting can be done when $|V(H)| \leqslant 5$, though the proof was ad hoc [20]. It did a somewhat tedious case analysis for various H, exploiting specific structure in the various small pattern graphs. Bressan gave a remarkably principled approach, introducing the notion of the *DAG treewidth* [34]. We will take some liberties with the original definition, for the sake of exposition. Bressan defined the DAG treewidth of $H^{\rightarrow}$, and showed that when this quantity is 1, $\mathrm{Hom}_{\vec{H}}(G^{\rightarrow})$ can be computed in near-linear time. The DAG treewidth is 1 when the following construct exists. For any source s of $H^{\rightarrow}$, let $R(s)$ be the set of vertices in $H^{\rightarrow}$ reachable from s. The sources of $H^{\rightarrow}$ need to be be arranged in a tree $\mathcal{T}$ such that the following holds. If s lies on the (unique) path between s_1 and s_2 (in $\mathcal{T}$), then $R(s_1) \cap R(s_2) \subseteq R(s)$. In some sense, this gives a divide-and-conquer framework to construct (and count) $H^{\rightarrow}$-homomorphisms. Any $H^{\rightarrow}$-homomorphism can be broken into "independent pieces" that are only connected by the restriction of the homomorphism to $R(s)$. By indexing all the tree homomorphisms appropriately, the total count of $H^{\rightarrow}$-homomorphisms can be determined in near-linear time by dynamic programming. Note that we need the DAG treewidth of *all* acyclic orientations of H to be 1, which is a challenging notion to describe succinctly.

From Induced Cycles to DAG tree decompositions. We observe an interesting contrast between the previous work of the authors and Bressan's work. The former provides a simple family of H for which $\mathrm{Hom}_H(G)$ can be computed in near-linear time in bounded degeneracy graphs, yet the proofs were ad hoc. The latter gave a principled algorithmic approach, but it does not succinctly describe what kinds of H allow for such near-linear algorithms. Can we get the best of both worlds?

Indeed, that is what we achieve. By a deeper understanding of *why* $|V(H)| \leqslant 5$ was critical in [20] and generalizing it through the language of DAG tree decompositions, we can prove: the DAG treewidth of H is one iff $\mathrm{LICL}(H) \leqslant 5$.

When $\mathrm{LICL}(H) \leqslant 5$, for any acyclic orientation $H^{\rightarrow}$, we provide a (rather complex) iterative procedure to construct the desired DAG tree decomposition $\mathcal{T}$.

The proof is intricate and involves many moving parts. The connection between induced cycles and DAG tree decompositions is provided by a construct called the *unique reachability graph.* For any set S of sources in $H^{\rightarrow}$, construct the following simple, undirected graph $UR(S)$. Add edge (s, s') if there exists a vertex that is in $R(s) \cap R(s')$, but not contained in any $R(s'')$, for $s'' \in S \setminus \{s, s'\}$. A key lemma states that if $UR(S)$ contains a cycle (for any subset S of sources), then H contains an induced cycle of at least twice the length. Any cycle in a simple graph has length at least 3. So if $UR(S)$ has a cycle, then H has an induced cycle of length at least 6. Thus, if $\text{LICL}(H) \leqslant 5$, for all S, the simple graph $UR(S)$ is a forest.

For any set S of sources, we will (inductively) construct a *partial* DAG tree decomposition that only involves S. Let us try to identify a “convenient” vertex $x \in S$ with the following property. We inductively take the partial DAG tree decomposition $\mathcal{T}'$ of $S \setminus \{x\}$, and try to attach x as a leaf in $\mathcal{T}'$ preserving the DAG tree decomposition conditions (that involve reachability). By carefully working out the definitions, we identify a specific intersection property of $R(x)$ with the reachable sets of the other sources in $S \setminus \{x\}$. When this property holds, we can attach x and extend the partial DAG tree decomposition, as described above. When the property fails, we prove that the degree of x in $UR(S)$ is at least 2. But $UR(S)$ is a forest, and thus contains a vertex of degree 1. Hence, we can always identify a convenient vertex x, and can iteratively build the entire DAG tree decomposition.

We also prove the converse. If $\text{LICL}(H) \geqslant 6$, then the DAG treewidth (of some orientation) is at least two. This proof is significantly less complex, but crucially uses the unique reachability graph.

The Lower Bound: Triangles Become Long Induced Cycles. We start with the simple construction of [20] that reduces triangle counting in arbitrary graphs to 6-cycle counting in bounded degeneracy graphs. Given a graph G where we wish to count triangles, we consider the graph G' where each edge of G is subdivided into a path of length 2. Clearly, triangles in G have a 1-1 correspondence with 6-cycles in G'. It is easy to verify that G' has bounded degeneracy.

Our main idea is to generalize this idea for any H where $\mathrm{LICL}(H) = 6$. The overall aim is to construct a graph G' where each H-homomorphism corresponds to a distinct induced 6-cycle in G', which comes from a triangle in G. We will actually fail to achieve this aim, but get "close enough" to prove the lower bound.

Let $\overline{H}$ denote the pattern obtained after removing the induced 6-cycle from H. Let us outline the construction of G'. We first take three copies of the vertices of G. For every edge (u, v) of G, connect copies of u and v that lie in *different* copies by a path of length two. Note that each triangle of G has been converted into six 6-cycles. We then add a single copy of $\overline{H}$, and connect $\overline{H}$ to the remaining vertices (these connections depend on the edges of H). This completes the description of G'. Exploiting the relation of degeneracy to vertex removal orderings, we can prove that G' has bounded degeneracy.

It is easy to see that every triangle in G leads to a distinct H-homomorphism. Yet the converse is potentially false. We may have "spurious" H-homomorphisms that do not involve the induced 6-cycles that came from triangles in G. By a careful analysis of G', we can show the following. Every spurious H-homomorphism *avoids* some vertex in the copy of $\overline{H}$ (in G').

These observations motivate the problem of *partitioned*-homomorphisms. Let $\mathbf{P}$ be a partition of the vertices of G' into k sets. A partitioned-homomorphism is an H-homomorphism where each vertex is mapped to a different set of the partition. We can choose $\mathbf{P}$ appropriately, so that the triangle count of G is the number of partitioned-homomorphisms scaled by a constant (that only depends on the automorphism group of H). Thus, we reduce triangle counting in arbitrary graphs to counting partitioned-homomorphisms in bounded degeneracy graphs.

Our next insight is to give up the hope of showing a many-one linear-time reduction from triangle counting to H-homomorphisms, and instead settle for a Turing reduction. This suffices for the lower bound of Theorem 4.0.1. Using inclusion-exclusion, we can reduce a single instance of partitioned-homomorphism counting to 2^k instances of vanilla H-homomorphism counting. The details are somewhat complex, but this description covers the basic ideas.

When $\text{LICL}(H) > 6$, we replace edges in G by longer paths, to give longer induced cycles. The partitions become more involved, but the essence of the proof remains the same.

4.2 Related Work

Counting homomorphisms has a rich history in the field of parameterized complexity theory. Díaz et al. [47] designed a dynamic programming based algorithm for the $\text{Hom}_H(G)$ problem with runtime $O(2^k n^{\text{tw}(H)+1})$ where $\text{tw}(H)$ is the treewidth of the target graph H. Dalmau and Jonsson [45] proved that $\text{Hom}_H(G)$ is polynomial time solvable if and only if H has bounded treewidth, otherwise it is $\#W[1]$-complete. More recently, Roth and Wellnitz [152] consider a *doubly restricted* version of $\text{Hom}_H(G)$, where both H and G are from *restricted* graph classes. They primarily focus on the parameterized dichotomy between poly-time solvable instances and $\#W[1]$-completeness.

We give a brief review of the graph parameters treewidth and degeneracy. The notion of tree decomposition and treewidth were introduced in a seminal work by Robertson and Seymour [146, 147, 148]; although it has been discovered before under different names [24, 71]. Over the years, tree decompositions have been used extensively to design fast divide-and-conquer algorithms for combinatorial problems. Degeneracy is a nuanced measure of sparsity and has been known since the early work of Szekeres-Wilf [170]. The family of bounded degeneracy graphs is quite rich: it involves all minor-closed families, bounded expansion families, and preferential attachment graphs. Most real-world graphs tend to have small degeneracy ([19, 22, 66, 81, 163], also Table 2 in [19]), underscoring the practical importance of this class. The degeneracy has been exploited for subgraph counting problems in many algorithmic results [8, 38, 59, 81, 83, 125, 129, 137].

Bressan [34] introduced the concept of DAG treewidth to design faster algorithms for homomorphism and subgraph counting problems in bounded degeneracy graphs. They prove the following dichotomy for the subgraph counting problem. For a pattern H with $|V(H)| = k$ and an input graph G with $|E(G)| = m$ and degeneracy κ, one can count $\mathrm{Hom}_H(G)$ in $f(\kappa, k)O(m^{\tau(H)} \log m)$ time, where $\tau(H)$ is the DAG treewidth of H (Theorem 4.3.2). On the other hand, assuming the exponential time hypothesis [78], the subgraph counting problem does not admit any $f(\kappa, k)m^{o(\tau(H)/\ln \tau(H))}$ algorithm, for any positive function $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$. Previous work of the authors shows that for every $k \geqslant 6$, there exists some pattern H with k vertices, such that $\mathrm{Hom}_H(G)$ cannot be counted in linear time, assuming fine-grained complexity conjectures [20]. We note that these results do not give a complete characterization like Theorem 4.0.1. They define classes of H that admit near-linear or specific polynomial time algorithms, and show that *some* H (but not all) outside this class does not have such efficient algorithms.

We remark here that in an independent and parallel work, Gishboliner, Levanzov, and Shapira [65] effectively prove the same characterization for linear time homomorphism counting.

The problem of approximately counting homomorphism and subgraphs have been studied extensively in various Big Data models such as the property testing model [13, 51, 52, 53], the streaming model [9, 14, 18, 22, 82, 87, 109, 113, 133], and the map reduce model [40, 90, 169]. These works often employ clever sampling based techniques and forego exact algorithms.

Almost half a century ago, Itai and Rodeh [79] gave the first non-trivial algorithm for the triangle detection and finding problem with $O(m^{3/2})$ runtime. Currently, the best known algorithm for the triangle detection problem uses fast matrix multiplication and runs in time $O(\min\{n^{\omega}, m^{2\omega/(\omega+1)}\})$ [12]. Improving on the exponent is a major open problem, and it is widely believed that $m^{4/3}$ (corresponding to $\omega = 2$) is a lower bound for the problem. Thus, disproving the TRIANGLE DETECTION CONJECTURE would require a significant breakthrough. See [6] for a detailed list of other classic graph problems whose hardness is derived using TRIANGLE DETECTION CONJECTURE.

4.3 Preliminaries

We use m and n to denote $|V(G)|$ and $|E(G)|$ respectively, for the input graph G. We denote $|V(H)|$ by k.

If a subset of vertices $V' \subseteq V(G)$ is deleted from G, we denote the remaining subgraph by $G - V'$. We use $G[V']$ to denote the subgraph of G induced by V'. The length of the largest induced cycle in H is denoted by $\text{LICL}(H)$.

DAG tree decompositions. Bressan [34] defined the notion of *DAG tree decompositions* for DAGs, analogous to the widely popular tree decompositions for undirected graphs. The crucial difference in this definition is that only the set of source vertices in the DAG are considered for creating the nodes in the tree. Let D be a DAG and $S \subseteq V$ be the set of source vertices in D. For a source vertex $s \in S$, let $\textsc{reachable}_D(s)$ denote the set of vertices in D that are reachable from s. For a subset of the sources $B \subseteq S$, let $\textsc{reachable}_D(B) = \bigcup_{s \in B} \textsc{reachable}_D(s)$. When the underlying DAG is clear from the context, we drop the subscript D.

Definition 4.3.1 (DAG tree decomposition [34])**.** Let D be a DAG with source vertices S. A DAG tree decomposition of D is a tree $\mathcal{T} = (\mathcal{B}, \mathcal{E})$ with the following three properties.

1. Each node $B \in \mathcal{B}$ (referred to as a "bag" of sources) is a subset of the source vertices S: $B \subseteq S$.
2. The union of the nodes in $\mathcal{T}$ is the entire set S: $\bigcup_{B \in \mathcal{B}} B = S$.
3. For all $B,\ B_1,\ B_2 \in \mathcal{B}$, if B lies on the unique path between the nodes B_1 and B_2 in $\mathcal{T}$, then $\textsc{reachable}(B_1) \cap \textsc{reachable}(B_2) \subseteq \textsc{reachable}(B)$.

The *DAG treewidth* of a DAG D is then defined as the minimum over all possible DAG tree decompositions of D, the size of the maximum *bag*. For a simple undirected graph H, the DAG treewidth is the maximum DAG treewidth over all possible acyclic orientations of H. We denote the DAG treewidth of D and H by $\tau(D)$ and $\tau(H)$, respectively.

Bressan [34] gave an algorithm for solving the $\textsc{hom-cnt}_H$ problem in bounded degeneracy graphs.

Theorem 4.3.2 (Theorem 16 in [34])**.** *Given an input graph G on m edges with degeneracy κ and a pattern graph H on k vertices, there is an $O(\kappa^k m^{\tau(H)} \log n)$ time algorithm for solving the $\textsc{hom-cnt}_H$ problem.*

4.4 LICL and Homomorphism Counting in Linear Time

We prove that the class of graphs with LICL $\leqslant 5$ is equivalent to the class of graphs with $\tau = 1$.

Theorem 4.4.1. *For a simple graph H, $\mathrm{LICL}(H) \leqslant 5$ if and only if $\tau(H) = 1$.*

By Theorem 4.3.2, this implies that $\mathrm{Hom}_H(G)$ can be determined in near-linear time when $\mathrm{LICL}(H) \leqslant 5$ and G has bounded degeneracy.

We first prove that, for a simple graph H, if $\mathrm{LICL}(H)$ is at most five, then $\tau(H) = 1$. This is discussed in Section 4.4.2. Then we prove the converse: if $\mathrm{LICL}(H)$ is at least six, then $\tau(H) \geqslant 2$. We take this up in Section 4.4.3.

Outline of the Proof Techniques. Before discussing the proofs in detail, we provide a high level description of the proof techniques.

Fix an arbitrary acyclic orientation $H^{\rightarrow}$ of H. We use S to denote the set of source vertices. We describe a recursive procedure to build a DAG tree decomposition of width one, starting from a single source in S.

Note that property (2) in the definition of DAG tree decomposition (Definition 4.3.1) requires the union of nodes in the tree to cover all the source vertices in S. So, we need to be careful, if we wish to use induction to construct the final DAG tree decomposition. To this end, we relax the property (1) and (2) of DAG tree decomposition and define a notion of *partial DAG tree decomposition*. In a partial DAG tree decomposition with respect to a subset $S_p \subseteq S$, the nodes in the tree are subsets of S_p and the union of the nodes cover the set S_p. The requirement of property (3) remains the same. The width of the tree is defined same as before. We formalize this in Definition 4.4.5. Now, consider a subset $S_{r+1} \subseteq S$ of size $r + 1$. We show how to build a partial DAG tree decomposition of width one for S_{r+1}, assuming there exists a partial DAG tree decomposition of width one for any subset of S of size r.

Let $x \in S_{r+1}$ and S_{-x} denote the set after removing the element x: $S_{-x} = S_{r+1} \setminus \{x\}$. Let $\mathcal{T}_{-x}$ be a partial DAG tree decomposition of width one for the set S_{-x} (such a tree exists by assumption). We identify a "good property" of the tree $\mathcal{T}_{-x}$ that enables construction of a width one partial DAG tree decomposition for the entire set S_{r+1}. The property is the following: there exists a leaf node ℓ in $\mathcal{T}_{-x}$ connected to the node $d \in \mathcal{T}_{-x}$ such that $\textsc{reachable}(x) \cap \textsc{reachable}(\ell) \subseteq \textsc{reachable}(d)$. We make this precise in Definition 4.4.7. Assume $\mathcal{T}_{-x}$ has this good property, and $\ell \in \mathcal{T}_{-x}$ be the leaf that enables $\mathcal{T}_{-x}$ to posses the good property. Then we first construct a width one partial DAG tree decomposition $\mathcal{T}_{-\ell}$ for the set $S_{-\ell} = S_{r+1} \setminus \{\ell\}$ and after that add ℓ as a leaf node to the node d in $\mathcal{T}_{-\ell}$. We prove that the resulting tree is indeed a valid width one partial DAG tree decomposition for S_{r+1}(we prove this in Claim 4.4.9). To complete the proof, it is now sufficient to show the existence of an element $x \in S_{r+1}$ such that a partial DAG tree decomposition for $\mathcal{T}_{-x}$ has the good property. This is the key technical element that distinguishes graphs with LICL at most 5 from those with LICL at least six.

We make a digression and discuss this key technical element further. We consider a graph that captures certain reachability aspects of the source vertices in $H^{\rightarrow}$. We define this as the unique rechability graph, UR_{S_p}, for a subset of the source vertices $S_p \subseteq S$. The vertex set of UR_{S_p} is simply the set S_p. Two vertices s_1 and s_2 in UR_{S_p} are joined by an edge if and only if there exists a vertex $v \in V(H^{\rightarrow})$ such that only s_1 and s_2 among the vertices in S_p, can reach v in $H^{\rightarrow}$. We prove that, if the underlying undirected graph H has $\mathrm{LICL}(H) \leqslant 5$, then the graph UR_{S_p}, for any subset $S_p \subseteq S$, is acyclic. This is given in Lemma 4.4.3

in Section 4.4.1.

Now, coming back to the proof of Lemma 4.4.4, we show that there must exist an element $x \in S_{r+1}$ such that a partial DAG tree decomposition for $\mathcal{T}_{-x}$ has the "good property", as otherwise the unique reachability graph $\mathrm{UR}_{S_{r+1}}$ over the set S_{r+1} has a cycle. However, this contradicts $\mathrm{LICL}(H) \leqslant 5$ (Lemma 4.4.3). This is established in Claim 4.4.10. This completes the proof of Lemma 4.4.4.

Now consider Lemma 4.4.11. Observe that, to prove this lemma, it is sufficient to exhibit a DAG $H^{\rightarrow}$ of H with $\tau(H^{\rightarrow}) \geqslant 2$. We first prove in Lemma 4.4.12 that if the the unique reachability graph $\mathrm{UR}_{S(H^{\rightarrow})}$ has a triangle, then $\tau(H^{\rightarrow}) \geqslant 2$. Then, for any graph H with $\mathrm{LICL}(H) \geqslant 6$, we construct a DAG $H^{\rightarrow}$ such that the unique reachability graph $\mathrm{UR}_{S(H^{\rightarrow})}$ has a triangle. It follows that $\tau(H) \geqslant 2$.

4.4.1 Main Technical Lemma

In this section, we describe our main technical lemma. We define a unique reachability graph for a DAG $H^{\rightarrow}$ over a subset of source vertices $S_p \subseteq S(H^{\rightarrow})$. This graph captures a certain reachability aspect of the source vertices in S_p in the graph $H^{\rightarrow}$. More specifically, two vertices s_1 and s_2 are joined by and edge in UR_{S_p} if and only if there exists a vertex v in $V(H^{\rightarrow})$ such that only s_1 and s_2 among the vertices in S_p, can reach v in $H^{\rightarrow}$.

Definition 4.4.2 (Unique reachability graph)**.** Let $H^{\rightarrow}$ be a DAG of H with source vertices S and $S_p \subseteq S$ be a subset of S. We define a unique reachability graph $\mathrm{UR}_{S_p}(S_p, E_{S_p})$ on the vertex set S_p, and the edge set E_{S_p} such that there exists an edge $e = \{s_1, s_2\} \in E_{S_p}$, for $s_1, s_2 \in S_p$, if and only if the set $(\textsc{reachable}_{H^{\rightarrow}}(s_1) \cap \textsc{reachable}_{H^{\rightarrow}}(s_2)) \setminus \textsc{reachable}_{H^{\rightarrow}}(S_p \setminus \{s_1, s_2\})$ is non-empty.

We are interested in the existence of a cycle in UR_{S_p}. We show that a cycle in

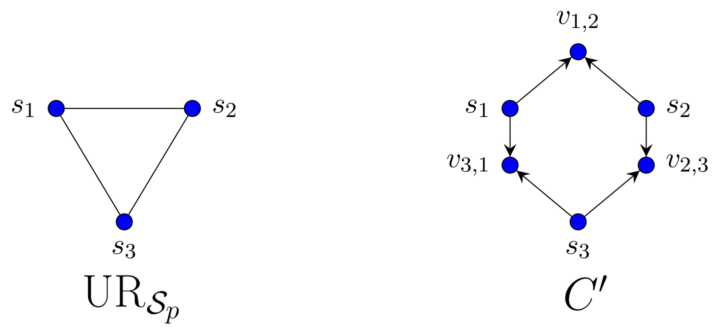


Figure 4.1: Let $S_p = \{s_1, s_2, s_3\}$. On the left, we give an example of a UR_{S_p} graph with a triangle. On the right, we give a possible example of the vertices $v_{1,2}$, $v_{2,3}$, and $v_{3,1}$ ($v_{i,j}$ is as defined in the proof of Lemma 4.4.3). C' forms an induced cycle of length six in H.

UR_{S_p} is closely related to an induced cycle in H, the underlying undirected graph of $H^{\rightarrow}$. More specifically, we prove that if LICL of H is at most five, then UR_{S_p} must be acyclic for each subset $S_p \subseteq S$.

Lemma 4.4.3. *Let $H^{\rightarrow}$ be a DAG of H with source vertices S and $S_p \subseteq S$ be an arbitrary subset of S. Let $\mathrm{UR}_{S_p}(S_p, E_{S_p})$ be the unique reachability graph for the subset S_p. If $\mathrm{LICL}(H) \leqslant 5$, then UR_{S_p} is acyclic.*

Proof. We in fact prove a stronger claim. We show that if UR_{S_p} has an ℓ-cycle, then $\mathrm{LICL}(H) \geqslant 2\ell$. Consider an edge $\{s_i, s_j\}$ in the edge set E_{S_p}. Let UNIQUE-REACHABLE(s_i, s_j) denote the set of vertices in $H^{\rightarrow}$ reachable from s_i and s_j both, but non-reachable from any other vertices in S_p. Let $\mathrm{dist}(s,t)$ denote the length of the shortest directed path from s to t in $H^{\rightarrow}$. We set $\mathrm{dist}(s,t) = \infty$, if t is not reachable from s. Now, let $v_{i,j}$ be the vertex in the UNIQUE-REACHABLE(s_i, s_j) set with the smallest total distance (directed) from s_i and s_j (breaking ties arbitrarily). Formally, $v_{i,j} = \arg\min_v \mathrm{dist}(s_i, v) + \mathrm{dist}(s_j, v)$, where $v \in$ UNIQUE-REACHABLE(s_i, s_j).

Let $C = s_1, s_2, \ldots, s_\ell, s_1 (= s_{\ell+1})$ be an ℓ-cycle in UR_{S_p}. Then using C and the vertices $v_{i,i+1}$, for $i \in [\ell]$ (abusing notation, we take $v_{\ell,\ell+1} = v_{\ell,1}$), we construct a cycle of length at least 2ℓ in H. Denote by $p_{s\to v}$ the directed path from a source vertex $s \in S_p$ to a vertex $v \in H^{\to}$. Intuitively, inserting the paths $p_{s_i\to v_{i,i+1}}$ and $p_{s_{i+1}\to v_{i,i+1}}$ between the source s_i and s_{i+1}, for each $i \in [\ell]$ (again, taking $s_{\ell+1} = s_1$), induces a cycle of length at least 2ℓ in H. See Figure 4.1 for a simple demonstration of this. We make this formal below.

Let $V(p)$ denote the set of vertices of a path p. Any edge between vertices in $V(p_{s_i\to v_{i,i+1}})$ other than the edges of $p_{s_i\to v_{i,i+1}}$, results in either a path between s_i and $v_{i,i+1}$ shorter than $\operatorname{dist}(s_i, v_{i,i+1})$ or a directed cycle in $H^{\to}$. Thus, the edges of $p_{s_i\to v_{i,i+1}}$ are the only edges between vertices in $V(p_{s_i\to v_{i,i+1}})$. It is easy to see that any edge between $V(p_{s_i\to v_{i,i+1}}) \setminus \{v_{i,i+1}\}$ and $V(p_{s_{i+1}\to v_{i,i+1}}) \setminus \{v_{i,i+1}\}$ result in a vertex $v'_{i,i+1}$ where $\operatorname{dist}(s_i, v'_{i,i+1}) + \operatorname{dist}(s_{i+1}, v'_{i,i+1}) < \operatorname{dist}(s_i, v_{i,i+1}) + \operatorname{dist}(s_{i+1}, v_{i,i+1})$, therefore no such edges exist. Also, any edge from a vertex in $\textsc{reachable}_{H^{\to}}(S_p\setminus\{s_i, s_{i+1}\})$ to a vertex in $V(p_{s_i\to v_{i,i+1}})$ or $V(p_{s_{i+1}\to v_{i,i+1}})$ implies that $v_{i,i+1} \in \textsc{reachable}_{H^{\to}}(S_p\setminus\{s_i, s_{i+1}\})$, which is not true by definition. Hence, there are no such edges either.

We use $E(p)$ to denote the set of edges of a path p. For convenience, we assume $\ell + 1$ in the index is to be treated as 1 instead. Let

$$V_{C'} = \bigcup_{p\in P} V(p), \qquad E_{C'} = \bigcup_{p\in P} E(p),$$
$$\text{where } P = \bigcup_{i\in[\ell]} \{p_{s_i\to v_{i,i+1}}, p_{s_{i+1}\to v_{i,i+1}}\}.$$

Now, it is easy to see that the graph induced by the set $V_{C'}$ is indeed an induced cycle C'. There are 2ℓ paths in P, and each path $p \in P$ has at least two vertices. As we showed above, these paths do not share edges. Thus the length of C' is at least 2ℓ, and $\operatorname{LICL}(H) \geqslant 2\ell$. Considering the contrapositive, we deduce that if $\operatorname{LICL}(H) \leqslant 5$, then UR_{S_p} is acyclic. □

4.4.2 DAG Treewidth for Graphs with LICL at most Five

In this section, we prove the following lemma.

Lemma 4.4.4. *For every simple graph H, if* $\mathrm{LICL}(H) \leqslant 5$ *then* $\tau(H) = 1$.

We introduce some notation. We start with defining the notion of partial DAG tree decomposition. In this definition, we consider a tree decomposition with respect to a subset of the source vertices of the original DAG.

Definition 4.4.5 (partial DAG tree decomposition)**.** Let $H^{\rightarrow}$ be a DAG with source vertices S. For a subset $S_p \subseteq S$, a partial DAG tree decomposition of $H^{\rightarrow}$ with respect to S_p is a tree $\mathcal{T} = (\mathcal{B}, \mathcal{E})$ with the following three properties.

1. Each node $B \in \mathcal{B}$ (referred to as a "bag") is a subset of the sources in S_p: $B \subseteq S_p$.
2. The union of the nodes in $\mathcal{T}$ is the entire set S_p: $\bigcup_{B \in \mathcal{B}} B = S_p$.
3. For all $B, B_1, B_2 \in \mathcal{B}$, if B is on the unique path between B_1 and B_2 in $\mathcal{T}$, then we have $\textsc{reachable}(B_1) \cap \textsc{reachable}(B_2) \subseteq \textsc{reachable}(B)$.

The partial DAG treewidth of $\mathcal{T}$ is $\max_{B \in \mathcal{B}} |B|$. Abusing notation, we use $\tau(\mathcal{T})$ to denote the partial DAG treewidth of $\mathcal{T}$.

Observe that, when $S_p = S$, we recover the original definition of DAG tree decomposition. Our proof strategy is to show by induction on the size of the subset S_p that there exists a partial DAG tree decomposition of width one for each $S_p \subseteq S$. In particular, when $S_p = S$, it follows that there exists a DAG tree decomposition for $H^{\rightarrow}$ of width one.

We next define INTERSECTION-COVER for a pair of vertices, based on the third property of the DAG tree decomposition. We generalize this notion to a subset of source vertices $S_p \subseteq S$ and define a notion of S_p-COVER. These notions will be useful in identifying a suitable source vertex in an existing partial DAG tree decomposition to attach a new node to it.

Definition 4.4.6 (INTERSECTION-COVER and S_p-COVER)**.** Let $H^{\rightarrow}$ be a DAG with sources S. Let s_1 and s_2 be a pair of sources in S. We call a source $s \in S$ an INTERSECTION-COVER of s_1 and s_2 if REACHABLE$(s_1) \cap$ REACHABLE$(s_2) \subseteq$ REACHABLE(s). Furthermore, assume $S_p \subseteq S$ is a subset of the sources S. We call a source $s \in S$, a S_p-COVER of $s_1 \in S$ if for each source $s_2 \in S_p$, s is an INTERSECTION-COVER for s_1 and s_2.

We now introduce one final piece of notation. Assume $S_p \subset S$ be a subset of the source vertices in the DAG $H^{\rightarrow}$. Let x be some source vertex in S that does not belong to S_p. Let $\mathcal{T}_{S_p}$ denote a partial DAG tree decomposition of width one for S_p. Now, consider a leaf node ℓ in $\mathcal{T}_{S_p}$. Let d denote the only node in $\mathcal{T}_{S_p}$ that is adjacent to ℓ. We claim that if d is an INTERSECTION-COVER for ℓ and x, then we can construct a partial DAG tree decomposition for $S_p \cup \{x\}$ of width one (we will make this more formal and precise in the following paragraph). We identify such source and partial DAG tree decomposition pair $(x, \mathcal{T}_{S_p})$ as a GOOD-PAIR.

Definition 4.4.7 (GOOD-PAIR)**.** Let $x \in S(H^{\rightarrow})$ be a source vertex and $\mathcal{T}_{S_p}$ be a partial DAG tree decomposition of width one for $S_p \subset S(H^{\rightarrow})$ where $x \notin S_p$. We call the pair $(x, \mathcal{T}_{S_p})$ a GOOD-PAIR if there exists a leaf node $\ell \in \mathcal{T}_{S_p}$ connected to the node $d \in \mathcal{T}_{S_p}$ such that d is an INTERSECTION-COVER for x and ℓ.

We prove a final technical lemma that provides insight into the process of adding a new source vertex to an existing partial DAG tree decomposition of width one.

Lemma 4.4.8. *Let $H^{\rightarrow}$ be a DAG of H with sources S and $S_p \subset S$ be a subset of S. Assume $\mathcal{T}$ is a partial DAG tree decomposition for S_p with $\tau(\mathcal{T}) = 1$. Consider a source $s \in S$ such that $s \notin S_p$. If $d \in S_p$ is a S_p-*COVER *of s, then connecting s to d in $\mathcal{T}$ as a leaf results in a tree $\mathcal{T}'$ that is a partial DAG tree decomposition for $S_p \cup \{s\}$. Furthermore, $\tau(\mathcal{T}') = 1$.*

Proof. We first prove that $\mathcal{T}'$ is a partial DAG tree decomposition for $S_p \cup \{s\}$. The properties (1) and (2) of partial DAG tree decomposition (see Definition 4.4.5) trivially hold for $\mathcal{T}'$. If $\mathcal{T}$ has one or two nodes, then by definition of S_p-COVER, $\mathcal{T}'$ satisfies property (3). So we assume $\mathcal{T}$ has at least 3 nodes.

Note that $\mathcal{T}$ and $\mathcal{T}'$ are identical barring the leaf node s. Hence, for any three nodes s_1, s_2, and s_3 in T' with $s \notin \{s_1, s_2, s_3\}$, property (3) of partial DAG tree decomposition (Definition 4.4.5) holds. Now, consider s with two other nodes s_1 and s_2 in $\mathcal{T}'$ where s_1 is on the unique path between s and s_2. If $s_1 = d$, then property (3) holds as d is a S_p-COVER of s. So assume $s_1 \neq d$. But then, s_1 is on the unique path between d and s_2 (by construction of $\mathcal{T}'$). Since property (3) holds for d, s_1, and s_2 in $\mathcal{T}$, we have $\textsc{reachable}(d) \cap \textsc{reachable}(s_2) \subseteq \textsc{reachable}(s_1)$. We also have $\textsc{reachable}(s) \cap \textsc{reachable}(s_2) \subseteq \textsc{reachable}(d)$ as d is a S_p-COVER of s. Hence, $\textsc{reachable}(s) \cap \textsc{reachable}(s_2) \subseteq \textsc{reachable}(s_1)$. Therefore, property (3) holds. Thus, $\mathcal{T}'$ is a partial DAG tree decomposition of $S_p \cup \{s\}$. As $\tau(\mathcal{T}) = 1$, it follows immediately from the construction that $\tau(\mathcal{T}') = 1$. □

We now have all the ingredients to prove Lemma 4.4.4. For the sake of completeness, we restate the lemma.

Lemma 4.4.4. *For every simple graph H, if* $\mathrm{LICL}(H) \leqslant 5$ *then $\tau(H) = 1$.*

Proof. The DAG treewidth of a simple graph H is defined as the maximum DAG treewidth of any DAG $H^{\rightarrow}$ obtained from H. So, we prove that $\tau(H^{\rightarrow}) = 1$ for each DAG $H^{\rightarrow}$ of H. In the rest of the proof, we fix a DAG of H, and call it $H^{\rightarrow}$. Let $S(H^{\rightarrow})$ denote the set of all source vertices in $H^{\rightarrow}$. When $H^{\rightarrow}$ is clear from the context, we simply use S.

Let $S_p \subseteq S$ denote a subset of S. We prove by induction on the size of the subset S_p that there exists a partial DAG tree decomposition (Definition 4.4.5) of width one for each $S_p \subseteq S$. In particular, when $S_p = S$, it follows that there exists a DAG tree decomposition for $H^{\rightarrow}$ of width one.

The base case of $|S_p| = 1$ is trivial: put the only source in S_p in a bag B as the only node in the partial DAG tree decomposition for $H^{\rightarrow}$. Similarly, for $|S_p| = 2$, put the two sources in two separate bags and connect them by an edge in the partial DAG tree decomposition for $H^{\rightarrow}$. The resulting tree is a partial DAG tree decomposition of width one. Now assume that, it is possible to build a partial DAG tree decomposition of width one for any subset $S_p \subset S$ where $|S_p| \leqslant r$, and $1 \leqslant r < |S|$. We show how to construct a partial DAG tree decomposition of width one for any subset $S_p \subseteq S$ where $|S_p| = r + 1$ for $r \geqslant 2$.

Fix a subset $S_{r+1} \subseteq S$ of size $r+1$. Consider an arbitrary source $x \in S_{r+1}$. By induction hypothesis, we can construct a partial DAG tree decomposition of width one for the set $S_{-x} = S_{r+1} \setminus \{x\}$. We call the tree $\mathcal{T}_{-x}$. Now recall that, we call the pair $(x, \mathcal{T}_{-x})$ a GOOD-PAIR if there exists a leaf node $\ell \in \mathcal{T}_{-x}$ connected to the node $d \in \mathcal{T}_{-x}$ such that d is an INTERSECTION-COVER for x and ℓ (see Definition 4.4.7). We argue the existence of a GOOD-PAIR $(x, \mathcal{T}_{-x})$ and give a constructive process to find a width one partial DAG tree decomposition of S_{r+1} from such a GOOD-PAIR $(x, \mathcal{T}_{-x})$.

A good-pair leads to a partial DAG tree decomposition of width one. We first show that if there exists a source $x \in S_{r+1}$ and a width one partial DAG tree decomposition $\mathcal{T}_{-x}$ for $S_{-x} = S_{r+1} \setminus \{x\}$ such that $(x, \mathcal{T}_{-x})$ is a GOOD-PAIR, then there exists a width one partial DAG tree decomposition for S_{r+1}. In fact, we give a simple constructive process to find such a partial DAG tree decomposition: construct a width one partial DAG tree decomposition $\mathcal{T}_{-\ell}$ for $S_{-\ell} = S_{r+1} \setminus \{\ell\}$, and then connect ℓ as a leaf to d in $\mathcal{T}_{-\ell}$.

Claim 4.4.9. *Let $x \in S_{r+1}$ be a source vertex and $\mathcal{T}_{-x}$ be a width one partial DAG tree decomposition for $S_{-x} = S_{r+1} \setminus \{x\}$ such that $(x, \mathcal{T}_{-x})$ is a* GOOD-PAIR*. Then, there exists a partial DAG tree decomposition $\mathcal{T}$ for S_{r+1} with $\tau(\mathcal{T}) = 1$.*

Proof. Since $(x, \mathcal{T}_{-x})$ is a GOOD-PAIR, there exists a leaf node $\ell \in \mathcal{T}_{-x}$ connected to the node $d \in \mathcal{T}_{-x}$ such that d is an INTERSECTION-COVER for x and ℓ. We build a partial DAG tree decomposition of width one for $S_{-\ell} = S_{r+1} \setminus \{\ell\}$ (such a tree exists by induction hypothesis), and then add ℓ as a leaf node to the node d. We prove that the resulting tree, denoted as $\mathcal{T}$, is partial DAG tree decomposition for S_{r+1} with $\tau(\mathcal{T}) = 1$.

Since ℓ is only connected to d in $\mathcal{T}_{-x}$, d is a S_{-x}-COVER of ℓ. Also, d is an INTERSECTION-COVER of ℓ and x, so d is a S_{r+1}-COVER of ℓ. Therefore, by applying Lemma 4.4.8, it follows that $\mathcal{T}$ is a partial DAG tree decomposition of S_{r+1} with $\tau(\mathcal{T}) = 1$. □

Existence of a good-pair. We have shown how to construct a partial DAG tree decomposition for the set S_{r+1} if there exists a GOOD-PAIR $(x, \mathcal{T}_{-x})$ where x is a source in S_{r+1} and $\mathcal{T}_{-x}$ is a width one partial DAG tree decomposition for S_{-x}. We now show that for any set S_{r+1}, there always exists a GOOD-PAIR $(x, \mathcal{T}_{-x})$.

Claim 4.4.10. *There exists a vertex $x \in S_{r+1}$ and a width one partial DAG tree decomposition $\mathcal{T}_{-x}$ for $S_{-x} = S_{r+1} \setminus \{x\}$, such that $(x, \mathcal{T}_{-x})$ is a* GOOD-PAIR*.*

Proof. Assume for contradiction, the claim is false. Consider the unique reachability graph on the vertex set S_{r+1}, denoted by $\mathrm{UR}_{S_{r+1}}(S_{r+1}, E_{S_{r+1}})$ (see Definition 4.4.2). Let $x \in S_{r+1}$ be an arbitrary source vertex. By assumption, $(x, \mathcal{T}_{-x})$ is not a GOOD-PAIR. So, for each leaf node $\ell \in \mathcal{T}_{-x}$ connected to the node $d \in \mathcal{T}_{-x}$, d is not an INTERSECTION-COVER for x and ℓ. Then, there exists a vertex v in $H^{\rightarrow}$, such that $v \in \text{REACHABLE}(x) \cap \text{REACHABLE}(\ell)$, but $v \notin \text{REACHABLE}(d)$. On the other hand, by construction, d is a S_{-x}-COVER for ℓ (d is the only node connected to ℓ in $\mathcal{T}_{-x}$). Hence, v is reachable from none of the source vertices in S_{-x}, other than ℓ. Therefore, the edge $\{x, \ell\} \in E_{S_{r+1}}$. Now, $\mathcal{T}_{-x}$ has at least two leaves, so the degree of the source vertex x in $\text{UR}_{S_{r+1}}$ is at least 2. The same argument holds for each $x \in S_{r+1}$. Hence, the degree of each vertex in $\text{UR}_{S_{r+1}}$ is at least two. Now $|S_{r+1}| \geqslant 3$ (recall $r \geqslant 2$), thus there exists a cycle $\mathcal{C}$ in $\text{UR}_{S_{r+1}}$ of length at least 3. By applying Lemma 4.4.3, we have $\text{LICL}(H) \geqslant 6$. But $\text{LICL}(H) \leqslant 5$, so this leads to a contradiction. Hence, the claim is true. □

We proved by induction that for any non-empty subset $S_p \subseteq S$, there exist a partial DAG tree decomposition for S_p with width one. In the case when $S_p = S$, the partial DAG tree decomposition is a DAG tree decomposition for $H^{\rightarrow}$. This completes the proof of Lemma 4.4.4. □

4.4.3 DAG Treewidth for Graphs with LICL at least Six

In this section, we prove the following lemma.

Lemma 4.4.11. *For every simple graph H, if* $\text{LICL}(H) \geqslant 6$ *then* $\tau(H) \geqslant 2$.

We first discuss the simple case of the 6-cycle. Note that, to prove that $\tau(H) \geqslant 2$, it suffices to show that there exists a DAG $H^{\rightarrow}$ of H such that $\tau(H^{\rightarrow}) \geqslant 2$. Let $H^{\rightarrow}$ be a DAG of H as shown in the middle figure in Fig. 4.2. Let $S = \{s_1, s_2, s_3\}$ be the set of sources in $H^{\rightarrow}$. Consider the unique reachability graph $\text{UR}_S(S, E_S)$, shown on the right in Fig. 4.2. The graph UR_S is a triangle: t_1 is not reachable

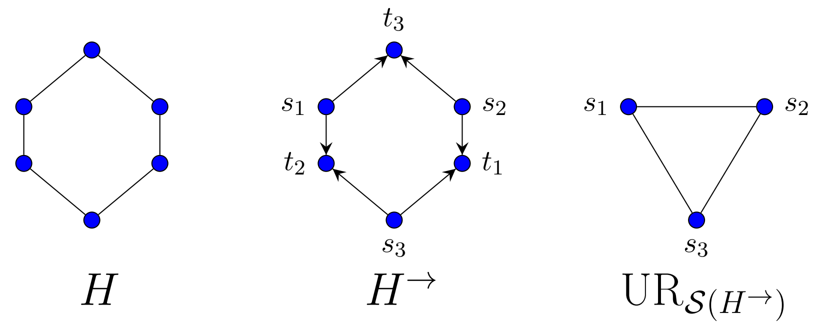


Figure 4.2: Let H be a six cycle. In the middle figure, we show the DAG $H^{\rightarrow}$ of H for which $\tau(H^{\rightarrow}) \geqslant 2$. On the right, we show the UR graph corresponding to $H^{\rightarrow}$. It is a triangle as t_1 is only reachable from $\{s_2, s_3\}$, t_2 is only reachable from $\{s_1, s_3\}$, and t_3 is only reachable from $\{s_1, s_2\}$.

from s_1, but reachable from s_2 and s_3, and so on. In any DAG tree decomposition $\mathcal{T}$ of $H^{\rightarrow}$ with width one, all source vertices are a vertex of $\mathcal{T}$ by themselves. So at least one of s_1, s_2, or s_3 (say s_1) would be on the unique path between the other two. But this would violate property (3) of DAG tree decomposition. It follows that $\tau(H) \geqslant 2$. In this case, it is not difficult to argue that $\tau(H) = 2$.

We formalize this intuition to prove in the following lemma: if the UR_S graph of a DAG $H^{\rightarrow}$ with source vertices S has a triangle in it, then it must the case that $\tau(H^{\rightarrow}) \geqslant 2$. Then, to prove $\tau(H) \geqslant 2$ for a graph H, it is sufficient to show the existence of a DAG $H^{\rightarrow}$ such that the corresponding $\mathrm{UR}_{S(H^{\rightarrow})}$ has a triangle.

Lemma 4.4.12. *Let $H^{\rightarrow}$ be a DAG of H with source vertices S and $\mathrm{UR}_S(S, E_S)$ be the* unique reachability *graph for the set S. If UR_S has a triangle, then $\tau(H) \geqslant 2$.*

Proof. Assume for contradiction, $\tau(H^{\rightarrow}) = 1$ and $\mathcal{T}$ be a DAG tree decomposition of width one. Let $\{s_1, s_2, s_3\}$ be a triangle in the graph UR_S. As $\mathcal{T}$ is a DAG tree decomposition with width one, all source vertices in S must be a node by themselves in $\mathcal{T}$. Observe that, there must exists a node $s \in \mathcal{T}$ that is between the unique path for a pair of nodes in $\{s_1, s_2, s_3\}$. As otherwise, these three nodes are all pairwise connected by an edge forming a triangle in $\mathcal{T}$. Wlog, assume s is on the unique path between s_1 and s_2 in $\mathcal{T}$. Since, $\{s_1, s_2\}$ is an edge in UR_S, by definition, there exists a vertex $t_{s_1,s_2} \in V(H^{\rightarrow})$, such that $t \in \text{REACHABLE}_{H^{\rightarrow}}(s_1) \cap \text{REACHABLE}_{H^{\rightarrow}}(s_2)$, but $t \notin \text{REACHABLE}_{H^{\rightarrow}}(s)$. But, this violates the property (3) of DAG tree decomposition in Definition 4.3.1. So such a tree $\mathcal{T}$ cannot exists and hence, $\tau(H^{\rightarrow}) \geqslant 2$. Therefore, $\tau(H) \geqslant 2$. □

We are now ready to prove the main lemma. We restate the lemma for completeness.

Lemma 4.4.11. *For every simple graph H, if* $\text{LICL}(H) \geqslant 6$ *then* $\tau(H) \geqslant 2$.

Proof. Let $\text{LICL}(H) = r$, where $r \geqslant 6$ and $r = 3\ell + q$, for some $\ell \geqslant 2$ where $q \in \{0, 1, 2\}$. Assume $C = v_1, v_2, \ldots, v_r, v_1$ is an induced cycle of length r in H. We construct a DAG $H^{\rightarrow}$ as follows.

Consider an edge $e = (u, v)$ in H. Assume only one of the end point does not belong to $V(C)$ — say $u \notin V(C)$. Then we orient the edge from v to u. Now consider the case when both $u, v \notin V(C)$. We orient such edges in an arbitrary manner ensuring the resulting orientation is acyclic. We now describe the orientation of the edges on the r-cycle C. We mark three vertices s_1, s_2 and s_3 in C as sources that are at least distance two apart from each other. Wlog, assume $s_1 = v_1$, $s_2 = v_{\ell+1}$, and $s_3 = v_{2\ell+1}$. Now we mark three vertices t_1, t_2, and t_3 as sinks such that t_1 is between s_1 and s_2, t_2 is between s_2 and s_3, and t_3 is between s_3 and s_1 in the cycle C. Again, wlong assume $t_1 = v_2$, $t_2 = v_{\ell+2}$, and $t_3 = v_{2\ell+2}$. Finally, orient the edges in C towards the sink vertices and away from the sources. This completes the description of $H^{\rightarrow}$.

Now let S denote the set of source vertices in $H^{\rightarrow}$. Consider the unique reachability graph $\text{UR}_S(S, E_S)$. We claim that UR_S includes a triangle. Indeed, we show that $\{s_1, s_2, s_3\}$ forms a triangle in UR_S. We first argue the existence of the edge $\{s_1, s_2\} \in E_S$. The vertex t_1 is reachable from s_1 and s_2, but not from

s_3. Since all the edges that are not part of the cycle C, are oriented outwards from the vertices in C, no other source vertices in S can reach t_1 in $H^{\rightarrow}$. Hence, $\{s_1, s_2\} \in E_S$. Similarly, we can argue the existence of the edges $\{s_2, s_3\}$ and $\{s_1, s_3\}$ in E_{S_p}. Applying Lemma 4.4.12, it follows that $\tau(H) \geqslant 2$. □

4.5 LICL and Homomorphism Counting Lower Bound

In this section, we prove our main lower bound result. We show that for a pattern graph H with $\text{LICL}(H) \geqslant 6$, the HOM-CNT$_H$ problem does not admit a linear time algorithm in bounded degeneracy graphs, assuming the TRIANGLE DETECTION CONJECTURE (Conjecture 2.2.3). We state our main theorem below.

Theorem 4.5.1. *Let H be a pattern graph on k vertices with* $\text{LICL}(H) \geqslant 6$*. Assuming the* TRIANGLE DETECTION CONJECTURE*, there exists an absolute constant $\gamma > 0$ such that for any function $f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$, there is no (expected) $f(\kappa, k)o(m^{1+\gamma})$ algorithm for the* HOM-CNT$_H$ *problem, where m and κ are the number of edges and the degeneracy of the input graph, respectively.*

Outline of the Proof. We first present an outline of our proof; the complete proof is discussed in Section 4.5.1. Let TRI-CNT denote the problem of counting the number of triangles in a graph. We prove the theorem by a linear time Turing reduction from the TRI-CNT problem to the HOM-CNT$_H$ problem. Assuming the TRIANGLE DETECTION CONJECTURE, any algorithm (possibly randomized) for the TRI-CNT problem requires $\omega(m)$ time, where m is the number of edges in the input graph. Given an input instance G of the TRI-CNT problem, we construct a graph G_H with bounded degeneracy and $O(|E(G)|)$ edges. We show how the number of triangles in G can be obtained by counting specific homomorphisms of H in G_H.

Let $\text{LICL}(H) = r$ and C be one of the largest induced cycles in H; let $V(C)$ denotes its vertices. We now describe the construction of the graph G_H. The graph G_H has two main parts: (1) the fixed component, denoted as G_{FIXED} (this part is independent of the input graph G and only depends on the pattern graph H) and (2) the core component, denoted as G_{CORE}. Additionally, there are edges that connect these two components, denoted by E_{BRIDGE}. Let $H_{C\text{-EXCLUDED}}$ denote the graph after we remove $V(C)$ from H. More formally, $H_{C\text{-EXCLUDED}} = H - V(C)$. The fixed component G_{FIXED} is a copy of $H_{C\text{-EXCLUDED}}$.

Next, we give an intuitive account of our construction. We discuss the role of G_{CORE} and how its connection to G_{FIXED} through E_{BRIDGE} ensures that the number of triangles in G can be obtained by counting homomorphisms of H. Then we give an overview of the construction.

Intuition behind the Construction. The main idea is to construct G_{CORE} and E_{BRIDGE} in such a way that each triangle in G transforms to an r-cycle in G_{CORE}, that then composes a match of H together with G_{FIXED} (recall that G_{FIXED} is a copy of $H_{C\text{-EXCLUDED}}$). To this end, we design G_{CORE} in r parts, ensuring the following properties hold for each r-cycle in G_{CORE} that contains exactly one vertex in each of these r parts: (1) It composes a match of H together with G_{FIXED} and (2) It corresponds to a triangle in G. Let $\mathcal{P}'$ denote the partition of $V(G_{\text{CORE}})$ into these r parts. Further, assume we construct G_H in a way that, each match of H that contains the vertices of G_{FIXED} and exactly one vertex in each set $V \in \mathcal{P}'$, corresponds to one of the r-cycles described above. It is now easy to see that if we can count these matches of H, we can then obtain the number of the described r-cycles in G_{CORE} and hence the number of triangles in G.

Consider a partition $\mathcal{P}$ of $V(G_H)$ where $|\mathcal{P}| = k$. Assume there is a linear time algorithm ALG for $\textsc{hom-cnt}_H$ in bounded degeneracy graphs. Then, Lemma 4.5.3 proves that there exists a linear time algorithm that, using ALG, counts the matches of H in G that include exactly one vertex in each set $V \in \mathcal{P}$.These matches are called $\mathcal{P}$-matches of H, as we define formally later in Definition 4.5.2. Also, each r-cycles in $G_{\textsc{core}}$ that contain exactly one vertex in each set $V \in \mathcal{P}'$ is a $\mathcal{P}'$-match of C. Now, we define the partition $\mathcal{P}$ of $V(G_H)$ as follows; $\mathcal{P}$ includes each set in $\mathcal{P}'$ and each of the $k - r$ vertices in $G_{\textsc{fixed}}$ as a set by itself.

Overall, by construction of G_H, we can get the number of triangles in G by the number of $\mathcal{P}'$-matches of C in $G_{\textsc{core}}$. Further, we can obtain the number of $\mathcal{P}'$-matches of C in $G_{\textsc{core}}$ by the number of $\mathcal{P}$-matches of H in G_H that we count using ALG. The following restates the desired properties of G_H we discussed, more formally.

(I) There is a bijection between the set of $\mathcal{P}$-matches of H in G_H and the set of $\mathcal{P}'$-matches of C in $G_{\textsc{core}}$.

(II) The number of triangles in G is a simple linear function of the number of $\mathcal{P}'$-matches of C in $G_{\textsc{core}}$.

Next, we give an overview of the construction of G_H for $r = 6$. We prove that properties (I) and (II) hold for our construction in the general case in Lemma 4.5.6 and Lemma 4.5.7, respectively.

Overview of the Construction. In what follows, we give an overview of $G_{\textsc{fixed}}$, $G_{\textsc{core}}$, and $E_{\textsc{bridge}}$. For the ease of presentation, we assume $r = 6$.

(1) $G_{\textsc{fixed}}$ is a copy of $H_{C\textsc{-excluded}}$. We denote the vertex set in $G_{\textsc{fixed}}$ as $V_{\textsc{fixed-set}}$. Observe that, $G_{\textsc{fixed}}$ does not depend on the input graph G.

(2) The core component $G_{\textsc{core}}$ consists of two set of vertices: $V_{\textsc{core-set}}$ and $V_{\textsc{auxiliary-set}}$. We first discuss the sets $V_{\textsc{core-set}}$ and $V_{\textsc{auxiliary-set}}$, and then introduce the edge set $E(G_{\textsc{core}})$.

   (a) $V_{\textsc{core-set}}$ consists of three set of vertices, $V_1 = \{w_1, \ldots, w_n\}$, $V_2 = \{x_1, \ldots, x_n\}$, and $V_3 = \{y_1, \ldots, y_n\}$ — each of size n (recall $|V(G)| = n$). The vertices in each of these sets correspond to the vertices in $V(G) = \{u_1, \ldots, u_n\}$.

   (b) The construction of $V_{\textsc{auxiliary-set}}$ depends on r. For $r = 6$, it consists of three sets, denoted as $V_{1,2}$, $V_{2,3}$, and $V_{1,3}$ — each of size $2m$ (recall $|E(G)| = m$). The vertices in each of these sets corresponds to the edges in $E(G)$. We index them using e, for each $e \in E(G)$: $V_{1,2} = \{v_e^{1,2}, v_e^{2,1}\}_{e \in E(G)}$, and so on. The role of these sets will become clear as we describe the edges of $G_{\textsc{core}}$.

   (c) Consider an edge $e = \{u_i, u_j\} \in E(G)$ and the pair V_1 and V_2. We connect the vertex $w_i \in V_1$ to the vertex $x_j \in V_2$ by a 2-path via the vertex $v_e^{1,2} \in V_{1,2}$. Similarly, we connect the vertex w_j to the vertex x_i by a 2-path via the vertex $v_e^{2,1}$. In particular, we add the edges $\{w_i, v_e^{1,2}\}$ and $\{v_e^{1,2}, x_j\}$, and the edges $\{w_j, v_e^{2,1}\}$ and $\{v_e^{2,1}, x_i\}$ to the set $E(G_{\textsc{core}})$. We repeat the process for the pairs (V_2, V_3) and (V_1, V_3) for each edge $e \in E(G)$.

(3) We now describe the edge set $E_{\textsc{bridge}}$ that serves as connections between $G_{\textsc{fixed}}$ and $G_{\textsc{core}}$. Let $\sigma_{\textsc{bridge}}$ be a bijective mapping between the sets $V(C)$ and $\{V_1, V_2, V_3, V_{1,2}, V_{2,3}, V_{1,3}\}$; $\sigma_{\textsc{bridge}} : V(C) \to \{V_1, V_{1,2}, V_2, V_{2,3}, V_3, V_{1,3}\}$. For each edge $e = \{u, v\} \in E(H)$ such that $u \in V(C)$ and $v \notin V(C)$, we do the following. Let $z_v \in V_{\textsc{fixed-set}}$ denote the vertex corresponding to the

vertex v (recall $G_{\text{FIXED-SET}}$ is a copy of $H_{C\text{-EXCLUDED}}$). We connect z_v to all the vertices in the set $\sigma_{\text{BRIDGE}}(u)$ and add these edges to E_{BRIDGE}.

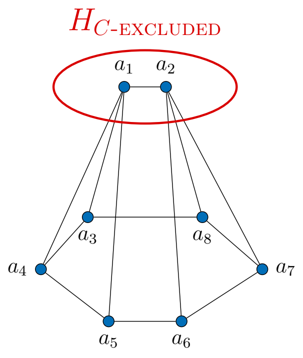
Note that, here $\mathcal{P}' = \{V_1, V_{1,2}, V_2, V_{2,3}, V_3, V_{1,3}\}$. Before diving into the details of deriving the triangle counts in G, we first take an example pattern graph H to visually depict the constructed graph G_H (see Figure 4.3) and discuss why properties (I) and (II) hold in our construction.

An Illustrative Example. Let H be the graph as shown in Figure 4.3a. In this example, $\text{LICL}(H) = 6$. Let $C = a_3, a_4, a_5, a_6, a_7, a_8, a_3$ be the induced 6-cycle in H. We demonstrate the constructed graph G_H in Figure 4.3b. We now discuss the various components of G_H.

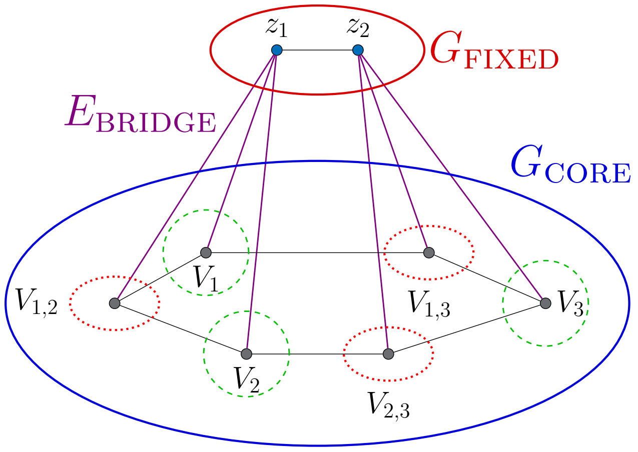
(1) The graph G_{FIXED} is shown by the red oval. The vertices z_1 and z_2 compose $V_{\text{FIXED-SET}}$, where z_1 corresponds to a_1 and z_2 corresponds to a_2.

(2) The graph G_{CORE} is shown by the blue oval. For each edge $e = \{u_i, u_j\} \in E(G)$, (for some input graph G, which is not shown in the figure), we add a total of six 2-paths: two between each pair of sets from $\{V_1, V_2, V_3\}$. For instance, between the set V_1 and V_2 these 2-paths are as follows: $\{w_i, v_e^{1,2}, x_j\}$ and $\{w_j, v_e^{2,1}, x_i\}$. The vertices w_i, w_j belong to V_1; x_i, x_j belong to V_2; and $v_e^{1,2}, v_e^{2,1}$ belong to $V_{1,2}$.

(3) Finally we describe the edge set E_{BRIDGE} (the edges in violet). We consider the following bijective mapping σ_{BRIDGE}: $\sigma_{\text{BRIDGE}}(a_3) = V_1$, $\sigma_{\text{BRIDGE}}(a_4) = V_{1,2}$, $\sigma_{\text{BRIDGE}}(a_5) = V_2$, $\sigma_{\text{BRIDGE}}(a_6) = V_{2,3}$, $\sigma_{\text{BRIDGE}}(a_7) = V_3$, $\sigma_{\text{BRIDGE}}(a_8) = V_{1,3}$. Now consider the edge $\{a_3, a_1\} \in E(G)$; $a_3 \in V(C)$ and $a_1 \notin V(C)$. So we connect z_1 (the vertex corresponding to a_1) to each vertex in the set $\sigma_{\text{BRIDGE}}(a_3) = V_1$. We repeat the same process for each edge $\{u, v\}$ in $E(H)$ where $u \in V(C)$ and $v \notin V(C)$.



(a) The pattern graph H



(b) The constructed graph G_H. In G_{CORE}, we only depict six edges corresponding to a triangle in G (there would be six more edges corresponding to the same triangle, that we do not show here). Also, we only depict the vertices relevant to the triangle.

Figure 4.3: G_H constructed for an example pattern graph H

Observe that in this example, $\mathcal{P} = \{\{z_1\}, \{z_2\}, V_1, V_{1,2}, V_2, V_{2,3}, V_3, V_{1,3}\}$. It is easy to see that E_{BRIDGE} connects G_{CORE} to G_{FIXED}, such that each 6-cycle in G_{CORE} compose a match of H together with G_{FIXED}. Each $\mathcal{P}$-match of H is actually an induced match as the only edges between its vertices in G_H are the edges that correspond to the match. Therefore, in this example, each $\mathcal{P}$-match of H in G_H include a 6-cycle in G_{CORE} that is actually a $\mathcal{P}'$-match of C. Thus, property (I) holds.

It is not difficult to see that a triangle in G introduces a total of six many 6-cycle in G_{CORE} that are $\mathcal{P}'$-matches of C in G_{CORE}. The converse follows as each $\mathcal{P}'$-match of C, which is a 6-cycle in G_{CORE}, must contain exactly one vertex from each of the three sets in each of $V_{\text{CORE-SET}}$ and $V_{\text{AUXILIARY-SET}}$. So, we could obtain the number of triangles in G by dividing the number of $\mathcal{P}'$-matches of C in G_{CORE} by six. Thus, property (II) holds.

Deriving The Triangle Counts in *G*. So far, we have shown that properties (I) and (II) hold in G_H for our construction. Therefore, the number of $\mathcal{P}$-matches of H in G_H reveals the number of triangles in G. However, we are interested in utilizing the homomorphism count of H to derive the triangle count in G. Indeed, we obtain the number of $\mathcal{P}$-matches of H in G_H by carefully looking at "restricted" homomorphisms from H to G. One crucial property of the graph G_H that we will require is bounded degeneracy. In fact, our construction of the graph G_H ensures that it has constant degeneracy irrespective of the degeneracy of G (we will formally prove this later in Lemma 4.5.5).

Let ALG be an algorithm for the $\textsc{hom-cnt}_H$ problem, that runs in $f(\kappa, k) \cdot O(m)$ time for some explicit function f, where m and κ are the number of edges and degeneracy of the input graph, respectively. Then, we can use ALG to count the homomorphisms from H to any subgraph of G_H in time $f(\kappa(G_H), k) \cdot O(m)$. Note that, here we use the fact that for any subgraph G'_H of G_H, $\kappa(G'_H) \leqslant \kappa(G_H)$.

We now solve the final missing piece of the puzzle: how to count the number of $\mathcal{P}$-matches of H in G_H using ALG? We present a two step solution to this question. First, we count the number of "$\mathcal{P}$ restricted" homomorphisms, denoted by $\mathcal{P}$-homomorphism and defined in Definition 4.5.2, from H to G_H by running ALG on carefully chosen subgraphs of G_H. Intuitively, a "$\mathcal{P}$ restricted" homomorphism is a homomorphism from H to G_H that involves at least one vertex in each part of $\mathcal{P}$. Second, we use the count from the first step to derive the number of $\mathcal{P}$-matches of H in G_H. We present this in Lemma 4.5.3.

We now formally define $\mathcal{P}$-match and $\mathcal{P}$-homomorphism.

Definition 4.5.2 ($\mathcal{P}$-match and $\mathcal{P}$-homomorphism)**.** Let $\mathcal{P} = \{V_1, \ldots, V_k\}$ be a partition of the vertex set $V(G)$ of the input graph G where $|V(H)| = k$ for the pattern graph H. Further assume $|V_i| \geqslant 1$ for each $i \in [k]$. Let $G_{\text{H-match}}$ be a subgraph of G such that $G_{\text{H-match}}$ is a match of H. We call $G_{\text{H-match}}$ a $\mathcal{P}$-match, if it includes exactly one vertex from each set V_i in $\mathcal{P}$: $|V(G_{\text{H-match}}) \cap V_i| = 1$ for each $i \in [k]$. Let $\pi : V(H) \to V(G)$ be a homomorphism from H to G. We call π a $\mathcal{P}$-homomorphism, if the image of π is non-empty in each set V_i: $|\{v : \pi(u) = v \text{ for } u \in V(H)\} \cap V_i| \geqslant 1$ for each $i \in [k]$.

In the following lemma, we prove that it is possible to count the number of $\mathcal{P}$-matches of H in G_H by running ALG on suitably chosen 2^k many subgraphs of G_H.

Lemma 4.5.3. *Assume that* ALG *is an algorithm for the* HOM-CNT$_H$ *problem that runs in time $O(mf(\kappa, k))$ for some function f, where $m = E(G)$ and $\kappa = \kappa(G)$ for the input graph G, and $k = V(H)$. Let $\mathcal{P} = \{V_1, \ldots, V_k\}$ be a partition of $V(G)$ with $|V_i| \geqslant 1$ for each $i \in [k]$. Then, there exists an algorithm that counts the number of $\mathcal{P}$-*match *of H in G with running time $O(2^k \cdot mf(\kappa, k))$.*

Proof. Let $\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_{2^k-1}$ be the non-empty subfamilies of the partition $\mathcal{P}$. Let $G_1, G_2, \ldots, G_{2^k-1}$ be the subgraphs of G where G_i is induced on the vertex set $V(G) \setminus (\bigcup_{S \in \mathcal{F}_i} S)$, for $i \in [2^k - 1]$. Note that a homomorphism from H to any subgraphs G_i, for $i \in [2^k - 1]$, is also a homomorphism from H to G. Since each G_i is a subgraph of G, the degeneracy $\kappa(G_i) \leqslant \kappa$. Then, ALG can count homomorphisms from H to any G_i in time $O(mf(\kappa, k))$. Using the inclusion-exclusion principle, we can obtain the number of homomorphisms from H to G that are also a homomorphism from H to at least one of the subgraphs in $\{G_1, G_2, \ldots, G_{2^k-1}\}$ in $O(2^k \cdot mf(\kappa, k))$. Hence, we can obtain the number of $\mathcal{P}$-homomorphisms from H to G as follows,

$$\text{Hom}_H(G) - \sum_{1 \leqslant i \leqslant 2^k - 1} (-1)^{|F_i| - 1} \text{Hom}_H(G_i) \,.$$

Note that if a homomorphism from H to G does not include any vertex in a set

V_i in $\mathcal{P}$, then it is also a homomorphism from H to at least one of the subgraphs in $\{G_1, G_2, G_{2^k-1}\}$. Thus, we do not count such a homomorphism from H to G. Since $k = |V(H)|$, $\mathcal{P}$-homomorphisms of H in G are actually embeddings of H in G that involve exactly one vertex in each part of P. Observe that each such embedding of H in G corresponds to a $\mathcal{P}$-match of H in G. For each match $G_{\text{H-match}}$ of H in G, there are $|Aut(H)|$ embeddings of H in G that map H to $G_{\text{H-match}}$. Thus, by dividing the number of $\mathcal{P}$-homomorphisms from H to G by $|Aut(H)|$, we obtain the number of $\mathcal{P}$-matches of H in G in $O(2^k \cdot mf(\kappa, k))$ time. $\square$

4.5.1 Proof of Main Theorem

We now present the details of the construction of G_H for the general case and prove Theorem 4.5.1.

Proof of Theorem 4.5.1. We present a linear time Turing reduction form the problem of TRI-CNT to the HOM-CNT$_H$ problem in bounded degeneracy graphs. Let G be the input instance of the TRI-CNT problem where $V(G) = \{u_1, \ldots, u_n\}$ and $|E(G)| = m$. First, we construct a graph G_H based on G and H such that G_H has bounded degeneracy and $O(m)$ edges.

Construction of G_H. Let LICL$(H) = r$ where $r \geqslant 6$; and let $V(H) = \{a_1, a_2, \ldots, a_k\}$, where $a_{k-r+1}, a_{k-r+2}, \ldots, a_k, a_{k-r+1}$ is an induced r-cycle C. Let $H_{C\text{-EXCLUDED}}$ denote $H - V(C)$. G_H has two main parts, G_{FIXED} and G_{CORE}. These two parts are connected by the edge set E_{BRIDGE}. G_{FIXED} is a copy of $H_{C\text{-EXCLUDED}}$ and has the vertex set $V_{\text{FIXED-SET}} = \{z_1, z_2, \ldots, z_{k-r}\}$. $z_i \in V_{\text{FIXED-SET}}$ corresponds to a_i in $H_{C\text{-EXCLUDED}}$ for $i \in [k-r]$. Thus, $z_i, z_j \in V_{\text{FIXED-SET}}$ are adjacent iff $\{a_i, a_j\}$ is an edge in $H_{C\text{-EXCLUDED}}$.

$G_{\textsc{core}}$ contains two sets of vertices, $V_{\textsc{core-set}}$, and $V_{\textsc{auxiliary-set}}$. Vertices in $V_{\textsc{core-set}}$ correspond to vertices in $V(G)$, and vertices in $V_{\textsc{auxiliary-set}}$ correspond to the edges in $E(G)$. $V_{\textsc{core-set}}$ consists of three copies of $V(G)$ without any edges inside them. More precisely, $V_{\textsc{core-set}}$ is composed of three sets of vertices $V_1 = \{w_1, \ldots, w_n\}$, $V_2 = \{x_1, \ldots, x_n\}$, and $V_3 = \{y_1, \ldots, y_n\}$. For $i \in [n]$, vertices $w_i \in V_1$, $x_i \in V_2$, and $y_i \in V_3$ correspond to $u_i \in V(G)$. There are no edges inside $V_{\textsc{core-set}}$. We describe $V_{\textsc{auxiliary-set}}$ next.

$V_{\textsc{auxiliary-set}}$ corresponds to the vertices of the paths of length $r/3$ that we add between V_1, V_2, and V_3. Let $r = 3\ell + q$, for some $\ell \geqslant 2$ and $q \in \{0, 1, 2\}$. The vertices in $V_{\textsc{auxiliary-set}}$ consists of the sets of vertices $V_{1,2}$, $V_{2,3}$, and $V_{1,3}$. For each edge $e \in E(G)$ and each pair in $\{V_1, V_2, V_3\}$, we add two sets of vertices to $V_{\textsc{auxiliary-set}}$. Next, we describe the vertices we add to $V_{1,2}$, $V_{2,3}$, and $V_{1,3}$ for an edge $e \in E(G)$. For the pair V_1 and V_2, we add

$$\begin{aligned} V_e^{1,2} &= \left\{v_{e,1}^{1,2}, \ldots, v_{e,\ell-1}^{1,2}\right\} \\ \text{and } V_e^{2,1} &= \left\{v_{e,1}^{2,1}, \ldots, v_{e,\ell-1}^{2,1}\right\} \end{aligned}$$

to $V_{1,2}$. For the pair V_2 and V_3, we add

$$\begin{aligned} V_e^{2,3} &= \left\{v_{e,1}^{2,3}, \ldots, v_{e,\ell-1+\lfloor q/2 \rfloor}^{2,3}\right\} \\ \text{and } V_e^{3,2} &= \left\{v_{e,1}^{3,2}, \ldots, v_{e,\ell-1+\lfloor q/2 \rfloor}^{3,2}\right\} \end{aligned}$$

to $V_{2,3}$. And finally, for the pair V_1 and V_3, we add

$$\begin{aligned} V_e^{1,3} &= \left\{v_{e,1}^{1,3}, \ldots, v_{e,\ell-1+\lfloor (q+1)/2 \rfloor}^{1,3}\right\} \\ \text{and } V_e^{3,1} &= \left\{v_{e,1}^{3,1}, \ldots, v_{e,\ell-1+\lfloor (q+1)/2 \rfloor}^{3,1}\right\} \end{aligned}$$

to $V_{1,3}$. The following defines $V_{1,2}$, $V_{2,3}$, and $V_{1,3}$ more formally. For $i, j \in \{1, 2, 3\}$ where $i < j$,

$$V_{i,j} = \bigcup_{e \in E(G)} V_e^{i,j} \cup V_e^{j,i}.$$

This completes the description of $V(G_{\text{CORE}})$. We describe $E(G_{\text{CORE}})$ next.

The edges inside G_{CORE} stitch vertices in $V_{\text{AUXILIARY-SET}}$ to form paths of length $r/3$ between each pair in $\{V_1, V_2, V_3\}$. $E(G_{\text{CORE-SET}})$ consists of three sets of edges, $E_{1,2}$, $E_{2,3}$, and $E_{1,3}$. For each edge in G and each pair in $\{V_1, V_2, V_3\}$, we add two sets of edges to G_{CORE}. We describe the edges we add to G_{CORE} for each edge $e = \{u_i, u_j\} \in E(G)$. For the pair V_1 and V_2, we add

$$\begin{aligned} E_e^{1,2} &= \Big\{(w_i, v_{e,1}^{1,2}), (v_{e,1}^{1,2}, v_{e,2}^{1,2}), \ldots, (v_{e,\ell-1}^{1,2}, x_j)\Big\} \\ \text{and } E_e^{2,1} &= \Big\{(w_j, v_{e,1}^{2,1}), (v_{e,1}^{2,1}, v_{e,2}^{2,1}), \ldots, (v_{e,\ell-1}^{2,1}, x_i)\Big\} \end{aligned}$$

to $E_{1,2}$. Edges in $E_{1,2}$ form ℓ-paths between V_1 and V_2 with $V_{1,2}$ as interior vertices. For the pair V_2 and V_3, we add

$$\begin{aligned} E_e^{2,3} &= \Big\{(x_i, v_{e,1}^{2,3}), (v_{e,1}^{2,3}, v_{e,2}^{2,3}), \ldots, (v_{e,\ell-1+\lfloor q/2 \rfloor}^{2,3}, y_j)\Big\} \\ \text{and } E_e^{3,2} &= \Big\{(x_j, v_{e,1}^{3,2}), (v_{e,1}^{3,2}, v_{e,2}^{3,2}), \ldots, (v_{e,\ell-1+\lfloor q/2 \rfloor}^{3,2}, y_i)\Big\} \end{aligned}$$

to $E_{2,3}$. Edges in $E_{2,3}$ compose $\ell + \lfloor q/2 \rfloor$-paths between V_2 and V_3, by joining the vertices in $V_{2,3}$. And, for the pair V_1 and V_3, we add

$$\begin{aligned} E_e^{1,3} &= \Big\{(w_i, v_{e,1}^{1,3}), (v_{e,1}^{1,3}, v_{e,2}^{1,3}), \ldots, (v_{e,\ell-1+\lfloor (q+1)/2 \rfloor}^{1,3}, y_j)\Big\} \\ \text{and } E_e^{3,1} &= \Big\{(w_j, v_{e,1}^{3,1}), (v_{e,1}^{3,1}, v_{e,2}^{3,1}), \ldots, (v_{e,\ell-1+\lfloor (q+1)/2 \rfloor}^{3,1}, y_i)\Big\} \end{aligned}$$

to $E_{1,3}$. The edge set $E_{1,3}$ joins vertices in $V_{1,3}$ to form $\ell + \lfloor (q+1)/2 \rfloor$-paths between V_1 and V_3. We can describe the three sets of edges that compose $E(G_{\text{CORE}})$ more formally as follows. For $i, j \in \{1, 2, 3\}$ where $i < j$,

$$E_{i,j} = \bigcup_{e \in E(G)} E_e^{i,j} \cup E_e^{j,i}.$$

Now, we describe the edge set E_{BRIDGE} that connects G_{FIXED} and G_{CORE}. First, we partition $V_{1,2}$, $V_{2,3}$, and $V_{1,3}$ based on distance to V_1, V_2, and V_3, respectively. For instance, we define $V_{1,2}^i$ to be all the vertices in $V_{1,2}$ with i as the length of the shortest path to a vertex in V_1. Recall that each vertex in $V_{1,2}$ serves as an internal vertex of a path between a vertex in V_1 and a vertex in V_2. Formally, we define

$$\begin{aligned} V_{1,2}^i &= \bigcup_{e \in E(G)} \{v_{e,i}^{1,2}, v_{e,i}^{2,1}\} \text{ for } i \in \{1, \ldots, \ell - 1\}, \\ V_{2,3}^i &= \bigcup_{e \in E(G)} \{v_{e,i}^{2,3}, v_{e,i}^{3,2}\} \text{ for } i \in \{1, \ldots, \ell - 1 + \lfloor q/2 \rfloor\}, \\ \text{and } V_{1,3}^i &= \bigcup_{e \in E(G)} \{v_{e,i}^{1,3}, v_{e,i}^{3,1}\} \text{ for } i \in \{1, \ldots, \ell - 1 + \lfloor (q+1)/2 \rfloor\}. \end{aligned}$$

Now that we have partitioned $V_{\text{AUXILIARY-SET}}$, we add the sets V_1, V_2, and V_3 to this partition of $V_{\text{AUXILIARY-SET}}$ to define a partition $\mathcal{P}'$ of $V(G_{\text{CORE}})$ as follows.

$$\begin{aligned} \mathcal{P}' = \Big\{ & V_1, V_2, V_3, \\ & V_{1,2}^1, \ldots, V_{1,2}^{\ell-1}, \\ & V_{2,3}^1, \ldots, V_{2,3}^{\ell-1+\lfloor q/2 \rfloor}, \\ & V_{1,3}^1, \ldots, V_{1,3}^{\ell-1+\lfloor (q+1)/2 \rfloor} \Big\}. \end{aligned}$$

Observe that $|\mathcal{P}'| = r$. Let $\sigma_{\text{BRIDGE}} : V(C) \to \mathcal{P}'$ be a bijective mapping. We first describe E_{BRIDGE} based on σ_{BRIDGE} and then specify σ_{BRIDGE}. The following describes the edges we add to E_{BRIDGE} for each edge $e = \{u, v\} \in E(H)$ where $u \in V(C)$ and $v \notin V(C)$. Let $z_v \in V_{\text{FIXED-SET}}$ be the vertex corresponding to v. We add an edge between z_v and each vertex in $\sigma_{\text{BRIDGE}}(u)$. We describe σ_{BRIDGE} next.

We set σ_{BRIDGE} to map $V(C)$ to an r-cycle in G_{CORE} that is a $\mathcal{P}'$-match of C (recall Definition 4.5.2). Recall that $C = a_{k-r+1}, a_{k-r+2}, \ldots, a_k, a_{k-r+1}$. We break this cycle into three parts of length ℓ, $\ell + \lfloor q/2 \rfloor$, and $\ell + \lfloor (q+1)/2 \rfloor$, respectively, starting from a_{k-r+1}. We set $\sigma_{\text{BRIDGE}}(a_{k-r+1})$ to V_1, $\sigma_{\text{BRIDGE}}(a_{k-r+1+\ell})$ to V_2, and $\sigma_{\text{BRIDGE}}(a_{k-r+1+2\ell+\lfloor q/2 \rfloor})$ to V_3. In order for σ_{BRIDGE} to map C to a $\mathcal{P}'$-match of C, we set σ_{BRIDGE} to map vertices of C between a_{k-r+1} and $a_{k-r+1+\ell}$ to vertices of the paths between V_1 and V_2, which are vertices in $V_{1,2}$. Similarly, σ_{BRIDGE} maps vertices of C between $a_{k-r+1+\ell}$ and $a_{k-r+1+2\ell+\lfloor q/2 \rfloor}$ to $V_{2,3}$, and vertices of C between $a_{k-r+1+2\ell+\lfloor q/2 \rfloor}$ and a_{k-r+1} to $V_{1,3}$. Formally,

$$\begin{aligned}
\sigma_{\text{BRIDGE}}(a_{k-r+1}) &= V_1, \\
\sigma_{\text{BRIDGE}}(a_{k-r+1+i}) &= V_{1,2}^i, \text{ for } i \in \{1, \ldots, \ell - 1\}, \\
\sigma_{\text{BRIDGE}}(a_{k-r+1+\ell}) &= V_2, \\
\sigma_{\text{BRIDGE}}(a_{k-r+1+\ell+i}) &= V_{2,3}^i, \text{ for } i \in \{1, \ldots, \ell - 1 + \lfloor q/2 \rfloor\}, \\
\sigma_{\text{BRIDGE}}(a_{k-r+1+2\ell+\lfloor q/2 \rfloor}) &= V_3, \\
\text{and } \sigma_{\text{BRIDGE}}(a_{k-r+1+2\ell+\lfloor q/2 \rfloor+i}) &= V_{2,3}^i, \text{ for } i \in \{1, \ldots, \ell - 1 + \lfloor (q+1)/2 \rfloor\}.
\end{aligned}$$

This completes the description of E_{BRIDGE} and hence G_H. Before presenting the details of the reduction, we first show that G_H has bounded degeneracy and $O(m)$ edges.

The following lemma shows that in order to prove a graph G is t-degenerate, we only need to exhibit an ordering $\prec$ of $V(G)$ such that each vertex of G has t

or fewer neighbors that come later in the ordering $\prec$. Given a graph G and an ordering $\prec$ of $V(G)$, the DAG $G^{\rightarrow}_{\prec}$ is obtained by orienting the edges of G with respect to $\prec$.

Lemma 4.5.4. *[Szekeres-Wilf [170]] Given a graph G, $\kappa(G) \leqslant t$ if there exists an ordering $\prec$ of $V(G)$ such that the out-degree of each vertex in $G^{\rightarrow}_{\prec}$ is at most t.*

Next, we show that G_H has bounded degeneracy using *Lemma* 4.5.4.

Lemma 4.5.5. *$\kappa(G_H) \leqslant k - r + 2$.*

Proof. We present a vertex ordering $\prec$ for G_H such that for each vertex $v \in V(G_H)$, the out-degree of v is at most $k - r + 2$ in $G_H{}^{\rightarrow}_{\prec}$. Let $\prec$ be an ordering of $V(G_H)$ such that $V_{\text{AUXILIARY-SET}} \prec V_{\text{CORE-SET}} \prec V_{\text{FIXED-SET}}$, and ordering within each set is arbitrary. Each vertex in $V_{\text{AUXILIARY-SET}}$ is connected to exactly two other vertices in $V(G_{\text{CORE}})$ and at most to all $k - r$ vertices in $V_{\text{FIXED-SET}}$. So the out-degree of each vertex in $V_{\text{AUXILIARY-SET}}$ in $G_H{}^{\rightarrow}_{\prec}$ is at most $k - r + 2$. Since there are no edges inside $V_{\text{CORE-SET}}$, the only out-edges from vertices inside $V_{\text{CORE-SET}}$ is to vertices in $V_{\text{FIXED-SET}}$. Further, the only out-edges from vertinces in $V_{\text{FIXED-SET}}$ are to other vertices in $V_{\text{FIXED-SET}}$. Thus the out-degree of each vertex $v \in V(G_H)$ in $G_H{}^{\rightarrow}_{\prec}$ is at most $k - r + 2$. $\square$

Observe that G_H has at most $\kappa(G_H) \cdot |V(G_H)|$ edges. By construction of G_H, $|V(G_H)| < 6m\ell + 3n + k$ and by Lemma 4.5.5, $\kappa(G_H) < k$. Thus, G_H has $O(m)$ edges.

Details of the Reduction. We define a partition of $V(G_H)$ by adding each vertex in $V_{\text{FIXED-SET}}$ as a set by itself to $\mathcal{P}'$. Formally,

$$\begin{aligned}\mathcal{P} = \Big\{&\{z_1\}, \{z_2\}, \ldots, \{z_{k-r}\},\\ &V_1, V_2, V_3,\\ &V_{1,2}^{1}, \ldots, V_{1,2}^{\ell-1},\\ &V_{2,3}^{1}, \ldots, V_{2,3}^{\ell-1+\lfloor q/2 \rfloor},\\ &V_{1,3}^{1}, \ldots, V_{1,3}^{\ell-1+\lfloor (q+1)/2 \rfloor}\Big\}.\end{aligned}$$

Observe that $|\mathcal{P}| = k$. Also, since G_H has bounded degeneracy, each subgraph of G_H has bounded degeneracy too. Thus, by Lemma 4.5.3 we can count $\mathcal{P}$-matches of H in G_H in linear time if there exists an algorithm ALG for HOM-CNT$_H$ problem that runs in time $O(mf(\kappa, k))$ for a positive function f. In Lemma 4.5.6, we prove that there is a bijection between $\mathcal{P}$-matches of H in G_H and $\mathcal{P}'$-matches of C in G_{CORE}. Further, in Lemma 4.5.7, we prove that the number of triangles in G is a simple linear function of the number of $\mathcal{P}'$-matches of C in G_{CORE}. So, by counting $\mathcal{P}$-matches of H in G_H, we can obtain the number of triangles in G.

Lemma 4.5.6. *There exists a bijection between the set of* $\mathcal{P}$-match*es of* H *in* G_H *and the set of* $\mathcal{P}'$-match*es of* C *in* G_{CORE}.

Proof. Let H' be a $\mathcal{P}$-match of H in G_H. Observe that by construction of G_H, the only edges of G_H inside $V(H')$ are edges of H'. Therefore, H' is actually an induced match of H in G_H. By construction of G_H, specifically E_{BRIDGE}, the number of edges between $V_{\text{FIXED-SET}}$ and $V(H') \setminus V_{\text{FIXED-SET}}$ is equal to the number of edges between $H_{C\text{-EXCLUDED}}$ and C. As G_{FIXED} is a copy of $H_{C\text{-EXCLUDED}}$, there are exactly $|E(H_{C\text{-EXCLUDED}})|$ edges inside the set of vertices $V_{\text{FIXED-SET}}$. Thus, H' has exactly $|E(C)| = r$ edges inside G_{CORE}. We describe these edges next.

Let w_i, x_j, and y_t be the vertices of H' in V_1, V_2, and V_3, respectively. Inside

G_{CORE}, w_i could only be connected to the two vertices of H' in $V^1_{1,2}$ and $V^1_{1,3}$. Furthermore, x_j could only be adjacent to the two vertices of H' in $V^{\ell-1}_{1,2}$ and $V^1_{2,3}$. And finally, y_t could only be neighbors of the two vertices of H' in ${V_{2,3}}^{\ell-1+\lfloor q/2 \rfloor}$ and ${V_{1,3}}^{\ell-1+\lfloor (q+1)/2 \rfloor}$. In addition, each vertex in $V_{\text{AUXILIARY-SET}}$ has at most two neighbors inside G_{CORE}, and the same holds in H'. Inside G_{CORE}, H' has exactly r edges, so each vertex is connected (only) to their two possible neighbors specified above. Hence, there exist an ℓ-path between w_i and x_j, an $\ell + \lfloor q/2 \rfloor$-path between x_j and y_t, and an $\ell + \lfloor (q+1)/2 \rfloor$-path between w_i and y_t. Thus, $H' - V_{\text{FIXED-SET}}$ is an r-cycle inside G_{CORE} that includes exactly one vertex in each part of P', and hence is a $\mathcal{P}'$-match of C. It is easy to see that this $\mathcal{P}'$-match of C is actually an induced match. Next, we show the other direction.

Let C' be a $\mathcal{P}'$-match of C in G_{CORE}. It is easy to see that by construction of G_{CORE}, C' is an induced match. By construction of G_H, $G_H[V(C') \cup V_{\text{FIXED-SET}}]$ is an induced match of H. Therefore, C' corresponds to exactly one $\mathcal{P}$-match of H in G_H. $\square$

Lemma 4.5.7. *Let $\mathcal{P}'$-match(C, G_{CORE}) denote the set of $\mathcal{P}'$-matches of C in G_{CORE}. The number of triangles in G is equal to $|\mathcal{P}'$-match$(C, G_{CORE})|/6$.*

Proof. Consider a cycle $C' \in \mathcal{P}'$-match(C, G_{CORE}). Let w_i, x_j, and y_t be the the only vertices of C' in V_1, V_2, and V_3, respectively. There should be a path between w_i and x_j in C' that does not include y_t, so other than w_i and x_j, it only includes vertices in $V_{\text{AUXILIARY-SET}}$. The only possible such path in G_{CORE} is an ℓ-path between w_i and x_j. Therefore, this ℓ-path exists, and as a result $(u_i, u_j) \in E(G)$. Similarly, C' includes a path between x_j and y_t that only contain vertices in V_{aux} other than x_j and y_t. Therefore, there exists an $\ell + \lfloor q/2 \rfloor$-path between x_j and y_t in G_{CORE}, and hence $(u_j, u_t) \in E(G)$. Finally, a path between w_i and y_t that other than its endpoints, only includes vertices in $V_{\text{AUXILIARY-SET}}$, should be a part of C'. So, there exists an $\ell + \lfloor (q+1)/2 \rfloor$-path between w_i and y_t in $G_{\textsc{core}}$. As a result, $(u_i, u_t) \in E(G)$. Thus, C' corresponds only to the triangle u_i, u_j, u_t in G. Observe that, C' could be specified by its vertices in V_1, V_2, and V_3. Next, we prove the other direction; exactly 6 $\mathcal{P}'$-matches of C in $G_{\textsc{core}}$ correspond to each triangle in G.

Consider a triangle T with the vertex set $\{u_i, u_j, u_t\}$ in G and a $\mathcal{P}'$-match C' of C in $G_{\textsc{core}}$ that corresponds to T. There are six different bijective mappings from $\{u_i, u_j, u_t\}$ to $\{V_1, V_2, V_3\}$. As we showed above, C' could be specified by its vertices in V_1, V_2, and V_3. So, given a bijective mapping $\sigma_{\textsc{triangle}} : \{u_i, u_j, u_t\} \rightarrow \{V_1, V_2, V_3\}$, the three vertices $\sigma_{\textsc{triangle}}(u_i)$, $\sigma_{\textsc{triangle}}(u_j)$, and $\sigma_{\textsc{triangle}}(u_t)$ specify C'. Thus, there are exactly 6 $\mathcal{P}'$-matches of C in $G_{\textsc{core}}$ that correspond to T. As a result, the number of triangles in G is $|\mathcal{P}'\text{-match}(C, G_{\textsc{core}})|/6$. □

Lemma 4.5.6 and Lemma 4.5.7 together show that we can obtain the number of triangles in G from the number of $\mathcal{P}$-matches of H in G_H, in constant time. In conclusion, we have proved that if there exists an algorithm ALG for the HOM-CNT$_H$ problem that runs in time $O(mf(\kappa, k))$ for a positive function f, then there exists an $O(m)$ algorithm for the TRI-CNT problem. Assuming the TRIANGLE DETECTION CONJECTURE, the problem of TRI-CNT has the worst case time complexity of $\omega(m)$ for an input graph with m edges. Thus, the $O(m)$ Turing reduction from the TRI-CNT problem to HOM-CNT$_H$ problem we presented proves Theorem 4.5.1. □

Observation 4.5.8. *In the proof of Theorem 4.5.1, we count* $\mathcal{P}$-homomorphisms *(defined in Definition 4.5.2) from H to G_H using the algorithm* ALG. *Since $|\mathcal{P}| = |V(H)|$, each* $\mathcal{P}$-homomorphism *from H to G_H is an embedding of H in G_H. Thus, we can apply the same argument of Lemma 4.5.3 assuming there exists an algorithm for counting subgraphs, that has the same running time of* ALG.

Therefore, using the same argument as that of the proof of Theorem 4.5.1, we can prove the exact same statement of Theorem 4.5.1 for the SUB-CNT$_H$ *problem.*

4.6 Conclusion

In this paper, we study the problem of counting homomorphisms of a fixed pattern H in a graph G with bounded degeneracy. We provided a clean characterization of the patterns H for which near-linear time algorithms are possible — if and only if the largest induced cycle in H has length at most 5 (assuming standard fine-grained complexity conjectures). We conclude this exposition with two natural research directions.

While we discover a clean dichotomy for the homomorphism counting problem, the landscape for the subgraph counting problem is not as clear. Our hardness result (Theorem 4.5.1) holds for the subgraph counting version — if a pattern H has LICL $\geqslant 6$, then there does not exists any near-linear time (randomized) algorithm for finding the subgraph count of H (see Observation observation 4.5.8). However, the "only if" direction does not follow. It would be interesting to find a tight characterization for the subgraph counting problem.

Both this work and the work in chapter 3 attempt at understanding what kind of patterns can be counted in near-linear time in sparse graphs. It would be interesting to explore beyond linear time algorithms. Specifically, we pose the following question: Can we characterize patterns that are countable in quadratic time?

Chapter 5

Counting Vertex Orbits of All 5-vertex subgraphs

In this chapter we present `EVOKE`, an algorithm for computing all 5-VOCs. To the best of our knowledge there is only one algorithm, the ORCA package [73], for computing 5-VOCs, but it does not terminate after days on graphs with tens of millions of edges. `EVOKE` counts 5-VOCs in these large datasets efficiently (within an hour) on a commodity machine and is hundreds of times faster than the-state-of-the-art.

5.1 Problem Description

The input $G = (V, E)$ is a simple, undirected graph. Our aim is to get local counts, for every vertex in G, for all the patterns given in Fig. 5.2. Fig. 5.2 shows all connected subgraphs with at most 5 vertices. We will refer to these as *patterns*. (We do not focus on disconnected patterns; results in [137] imply that these can be easily determined from connected subgraph counts.)

We delay the exact formalism of orbits to §5.4. But hopefully, Fig. 5.2 gives a clear pictorial representation of the 73 different orbits, numbered individually.

Our aim is to design an algorithm that: for every vertex v in V and every orbit θ, exactly outputs the number of times that v occurs in a copy of θ. Thus, the output is a set of $73|V|$ counts. (Technically, we ask for induced counts, but can also get non-induced counts. Details in §5.4.) For example, the count of orbit 17 is the number of times that v is the middle of a 4-path, while the count of orbit 15 is the number of 4-paths that start/end at v. Analogously, the count of orbit 34 is the number of 5-cycles that v participates in. For a fixed orbit, we refer to these numbers as the *vertex orbit counts* (VOCs). Collectively (over all orbits), we wish to determine *VOCs for all* 5*-vertex subgraphs*. For convenience, we refer to this as simply 5-VOCs. We refer to the total subgraph count as "global" counts, which is clearly a much easier problem.

As can be seen, the desired output is an immensely rich local description of the vertices of G. This output subsumes a number of recent subgraph counting problems in the data mining community [8, 56, 58, 137].

Main challenges: To the best of our knowledge, there is no algorithm that (even approximately) computes all 5-VOCs even for graphs with tens of millions of edges. Results on global counting are much faster, but it is not clear how to implement these ideas for VOCs [8, 137]. The ORCA package is the only algorithm that actually computes all 5-VOCs, but it does not terminate after days for graphs with tens of millions of edges. We give more details of previous work in §5.3.

From a mathematical standpoint, the challenge is to get all 5-VOCs without an expensive enumeration. The total number of orbit counts is easily in the order of trillions, and a fast algorithm should ideally avoid touching each 5-vertex subgraph in G. On the other hand, VOCs are an extremely fine-grained statistic, so purely global methods do not work.

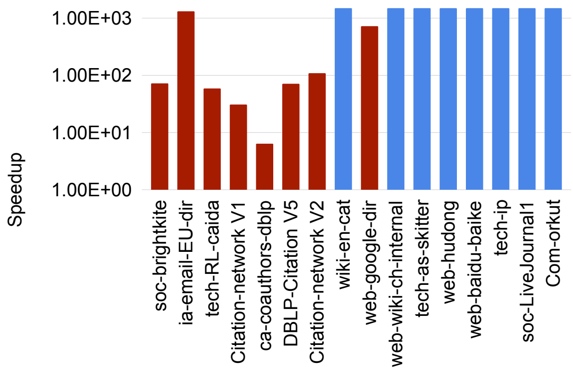


Figure 5.1: Runtime speedup for computing all 5-VOCs achieved by `EVOKE` over ORCA (computed as runtime of ORCA/runtime of `EVOKE`). Graphs are sorted by increasing number of edges from left to right. For the blue bars, ORCA ran out of memory or did not terminate after 1000 times the `EVOKE` running time. `EVOKE` is significantly faster than ORCA, and makes 5-VOC counting feasible for large graphs.

5.2 Main Contributions

Our primary result is the Efficient Vertex Orbit pacKagE (`EVOKE`), an algorithm to compute all 5-VOCs.

Practical local counting: `EVOKE` advances the state of the art of subgraph counting. It is the first algorithm that can feasibly obtain all 5-VOCs on graphs with tens of millions of edges. We do comprehensive tests on many public data sets. We observe that `EVOKE` gets counts on graphs with millions of edges in just minutes, and on graphs with tens of millions of edges within an hour. This is on a single commodity machine with 64GB memory, without any parallelization. In contrast, for the larger instances, the previous state of the art ORCA package takes more than two days or runs out of memory, on a more powerful machine (384GB RAM). Even on instances where ORCA terminates, `EVOKE` is about a hundred times faster. We show the speedup of `EVOKE` over ORCA in Fig. 5.1. `EVOKE` is also able to get 5-VOCs in a social network with 100M edges, in less

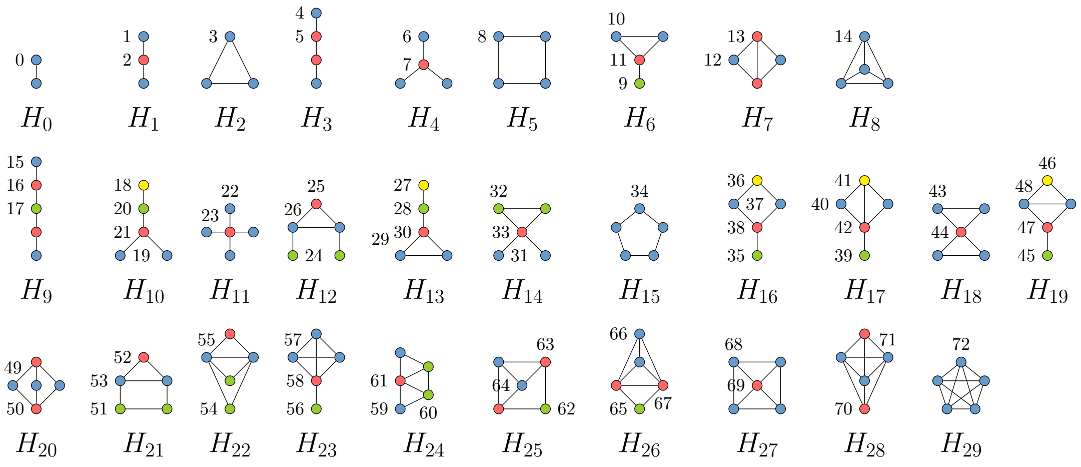


Figure 5.2: All vertex orbits for 5-vertex patterns. Within any pattern, vertices of the same color form an orbit.

than two days. (ORCA runs of out memory in such instances.) All the blue bars in Fig. 5.1 denote instances where ORCA runs of out of memory (in two days) or is a thousand times slower than `EVOKE`.

`EVOKE` has a large number of independent sub-algorithms. It is straightforward to run them in parallel, and we get about a factor two speedup. We do not consider this a significant novelty of `EVOKE`, but it does allow for an even faster running time.

Local counting without enumeration: Our work builds on the ESCAPE framework of Pinar-Seshadhri-Vishal [137]. One of their main insights is a combination of graph orientations and a "pattern cutting" technique. Larger patterns are carefully cut into smaller patterns. It is then shown that local counts of *smaller* patterns can be combined into *global* (total) counts of larger patterns. We formally prove that, for the orbits in Fig. 5.2, one can generalize their method to VOCs. This is mathematically quite technical and requires manipulations of various pattern automorphisms (which is not required for total counts). But the final result is a large collection of polynomial formulas to compute individual VOCs through some specialized local counts of smaller subgraphs. `EVOKE` exploits the structure among these formulas to count all VOCs efficiently.

Somewhat surprisingly, we mathematically prove that the running time is only a constant factor more than that of ESCAPE (which only computes total counts). This is borne out empirically where the running time of `EVOKE` is typically twice that of ESCAPE. Our result demonstrates the power of the cutting framework introduced in [137].

Fast computation of orbit frequency distributions: The distribution of VOCs is a useful tool in graph analysis, often called *graphlet degree distribution* in bioinformatics [139]. `EVOKE` makes it feasible to compute these distributions over real data. As a small demonstration of `EVOKE`, we observe interesting behavior in VOCs across graphs from different domains. Also, the VOCs of different orbits within the same pattern behave differently, showing the importance of getting such fine-grained information.

On 4-VOCs: We do not consider this as a new contribution, but a salient observation for those interested in subgraph counting. `EVOKE` determines all 4-VOCs as a preprocessing step, based on ideas in [137] and Ortmann-Brandes [125]. As stated in these results, the key insight is an implementation of an old algorithm of Chiba-Nishizeki for 4-cycle counting [39]. This method is incredibly fast, and computes 4-VOCs in minutes. (Even for the largest instance of more than 100M edges, it took less than an hour.) For example, for a LiveJournal social network with 42M edges, `EVOKE` took ten minutes on a commodity machine (we got the same time even on a laptop). Contrast this with previous results for counting 4-VOCs for the same graph, which used a MapReduce cluster [58]. (We note that `EVOKE`, and the other results, are technically computing edge orbit counts, a more general problem.)

5.3 Related Work

Subgraph counting is an immensely rich area of study, and we refer the reader to a tutorial for more details [159]. Here, we only document results relevant to our problem. For this reason, we do not discuss the extremely large body of work on triangle counting (the most basic subgraph counting problem).

Vertex orbit counts beyond triangles have found significant uses in network analysis and machine learning. Notably, Shervashidze et al. defined the *graphlet kernel*, that uses vertex orbits counts to get embeddings of vertices in a network [162]. Ugander-Backstrom-Kleinberg showed that 4-vertex orbit counts can be used for role discovery and distinguishing different types of graph neighborhoods [179]. In an exciting recent use of orbit counts, Rotabi-Kamath-Kleinberg-Sharma showed that four and five cycle counts can be used for weak tie discovery in the Twitter network [151]. Yin-Benson-Leskovec have defined higher-order clustering coefficients, which are ratios of specific orbit counts [196, 197]. There is a line of work on the surprising benefits of using cycle and clique counts as vertex or edge weights, to find denser and more relevant communities in networks [17, 23, 154, 173, 175].

We now discuss the literature on algorithms for subgraph counting. Ahmed-Neville-Rossi-Duffield gave the first algorithm that could count (total) 4-vertex subgraph counts for graphs with millions of edges [8]. Their PGD package was a significant improvement over past practical work for this problem [67]. Pinar-Seshadhri-Vishal designed the ESCAPE algorithm for practical (total) 5-vertex subgraph counting [137]. While these algorithms employed many clever combinatorial ideas, they did not focus on vertex orbit counting. There was concurrent development of sampling algorithms that are orders of magnitude faster, such as path-sampling [84] and the MOSS package [186].

Elenberg-Shanmugam-Borokhovich-Dimakis gave algorithms for 3, 4-vertex orbit counting [56, 58]. They employed a randomized algorithm, and proved convergence through polynomial concentration inequalities. The number of samples required for concentration was large, and they used Map-Reduce clusters to process graphs with tens of millions of edges. It was observed implicitly in the ESCAPE package and explicitly, by Ortmann-Brandes [125] that ideas from a classic result of Chiba-Nishizeki [39] gave a faster, exact algorithm for 4-vertex orbits.

The state of the art for local counting of 5-vertex orbits is the ORCA package of Hočevar-Demšar [75]. The algorithm is based on a method to build sets of linear equations relating various orbit counts. This saves computing all orbit counts independently. With some careful choices, ORCA tries to perform enumeration on the "easier" counts, and get the "harder" counts through the linear equations. There were also results on generating these linear equations auotmatically [116, 115]. We note that ORCA also has algorithms to generate 5-edge orbit counts, but this takes even longer than 5-VOCs. We leave the generalization of `EVOKE` to edge orbit counts as future work.

Rossi-Ahmed-Carranza-Arbour-Rao-Kim-Koh proposed a parallel algorithm for counting typed graphlets (subgraph patterns), which are a generalization of subgraph patterns to heterogeneous networks [150].

Most of the exact subgraph and orbit counting algorithms work on subgraphs of size 5 or less. Algorithms that attempt to count subgraphs beyond that size typically use randomization, which has inspired a rich literature [14, 26, 42, 80, 82, 84, 85, 91, 92, 99, 103, 114, 126, 132, 141, 157, 158, 164, 174, 186, 191, 198, 199].

5.4 Preliminaries

The input is an undirected simple graph $G = (V, E)$, with n vertices and m edges. The patterns of interest are all connected subgraphs with at most 5 vertices, denoted $H_0, \ldots, H_{29}$, as shown in Fig. 5.2. Previous results in [137] show that disconnected pattern counts can be determined by inclusion-exclusion from all connected pattern counts. Hence, we only focus on connected pattern subgraphs.

We now formally define orbits. The definitions below are taken from Bondy and Murty (Chapter 1, Section 2) [31].

Definition 5.4.1. Fix labeled graph $H = (V(H), E(H))$. An *automorphism* is a bijection $\sigma : V(H) \to V(H)$ such that $(u, v) \in E(H)$ iff $(\sigma(u), \sigma(v)) \in E(H)$.

Define an equivalence relation among $V(H)$ as follows. We say that $u \sim v$ $(u, v \in V(H))$ iff there exists an automorphism that maps u to v. The equivalence classes of the relation are called *orbits*.

Fig. 5.2 shows the 73 different orbits. Within any H_i, all vertices in an orbit are colored the same. For example, in H_{28}, there are two different orbits (blue and red). The blue (resp. red) vertices can be mapped to each other by automorphisms, and are therefore "equivalent".

Technically, we denote orbits as pairs (H, S), where H is a (labeled) pattern subgraph and S is the subset of vertices forming the orbit. Consider pattern H and orbit $\theta = (H, S)$. We denote:

- $\text{orb}(H)$: The set of orbits in the pattern H.
- $\text{sz}(\theta)$: $|S|$, the number of vertices in the orbit θ.

We can similarly define edge orbits as follows.

Definition 5.4.2. Fix labeled graph $H = (V(H), E(H))$. Define an equivalence relation among $E(H)$ as follows. Let $e = (u, v)$ and $e' = (w, x)$ be two edges in

H. We say that $e \sim e'$ iff there exists an automorphism that maps e to e' (i.e. it maps u to w and v to x or u to x and v to w). The equivalence classes of the relation are called *edge-orbits*.

Induced vs non-induced: A non-induced subgraph is obtained by taking a subset of edges. An induced subgraph is obtained by taking a subset of vertices and considering all edges and non-edges among them. (A clique contains all non-induced subgraphs of smaller sizes, but the only induced subgraphs it contains are smaller cliques.) A theorem in [137] proves that the vector of non-induced subgraph (up to a given size) counts can be converted to the corresponding induced counts, through a linear transformation. A directed generalization of the arguments holds for k-VOCs, in that non-induced orbit counts (for each vertex) can be converted to induced orbit counts by a linear transformation.

`EVOKE` computes both non-induced and induced counts. Algorithmically, it is easier to compute non-induced counts first; hence we shall only refer to them in the technical description.

We are ready to define VOCs.

Definition 5.4.3. Fix an orbit $\theta = (H, S)$ and a vertex $v \in V$ (in the input graph G). A *match* of θ involving v is a non-induced copy of H in G such that v is mapped to a vertex in S. Call two matches *equivalent*, if one can be obtained from the other by applying an automorphism. We define $\text{DM}(v, \theta)$ to be the number of distinct matches of θ involving v.

Our aim is to compute the entire list of numbers $\{UM(v, \theta)\}$, over all $v \in V$ and all θ in Fig. 5.2.

Conversion between Induced and Noninduced counts: Given $u \in V(G)$ and orbit θ, we used $\text{DM}_G(u, \theta)$ to denote the number of distinct non-induced matches of θ in G. Let $\mathrm{DIM}_G(u, \theta)$ denote the number of distinct induced orbit counts. Let $\theta_i = (H, S)$, where H has k vertices, where $k \leqslant 5$. It is easy to see that we can obtain $\mathrm{DM}_G(u, \theta_i)$ from the set $\{\mathrm{DIM}_G(u, \theta_i), \ldots, \mathrm{DIM}_G(u, \theta_j)\}$, where θ_j is the vertex orbit with the largest index in k-vertex patterns.

Consider orbit $\theta_j = (H', S')$, where $j \geqslant i$. Let v be a vertex in H', where $v \in S'$. If we use graph H' as our input graph (instead of G), $\mathrm{DM}_{H'}(v, \theta_i)$ is actually the number of non-induced matches of θ_j in G that an induced match of θ_j include as a subgraph.

If we think of the list of induced and non induced node orbit counts for any vertex u in any graph G as vectors $\mathrm{DIM}(u)$ and $\mathrm{DM}(u)$, there is a matrix $\boldsymbol{A}$ such that $\mathrm{DM}(u) = \boldsymbol{A}\ \mathrm{DIM}(u)$. The matrix for orbits which lie in 4-vertex patterns (orbits θ_4-θ_{14}) is given in Fig. 5.3. The matrix for orbits of 5-vertex patterns (orbits θ_{15}-θ_{72}) is too large to be included here, but we made it accessible at [2]. Note that $\boldsymbol{A}_{i,j}$ is the number of non-induced matches of θ_i that an induced match of orbit θ_j include, for any vertex u in any graph G.

Naturally, $\mathrm{DIM}(u) = \boldsymbol{A}^{-1}\mathrm{DM}(u)$, and that is how we get induced counts from non-induced counts. The inverse matrix for vertex orbits θ_4-θ_{14} is given in Fig. 5.4, the inverse matrix for orbits θ_{15}-θ_{72} is again too large to be included here, but could be found in [2].

Degree ordering: We will use the *degree orientation*, a fundamental tool for subgraph counting that was pioneered by Chiba-Nishizeki [39]. We will convert G into an DAG $G^{\rightarrow}$ as follows. Let $\prec$ denote the *degree ordering* of G. For vertices i, j, we say $i \prec j$, if either $d(i) < d(j)$ or $d(i) = d(j)$ and $i < j$ (comparing vertex id). The DAG $G^{\rightarrow}$ is obtained by orienting the edges with respect to $\prec$ ordering. In both the algorithm and analysis, all references to directed structures are with respect to $G^{\rightarrow}$.

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 2 & 2 & 1 & 0 & 4 & 2 & 6 \\ 0 & 1 & 0 & 0 & 2 & 0 & 1 & 2 & 2 & 2 & 6 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 2 & 1 & 3 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 3 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 2 & 0 & 3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 2 & 2 & 6 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 2 & 3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 5.3: Matrix transforming induced vertex orbit counts for orbits 0-14 to non-induced counts

Notation for subgraph counts: In our formulas for orbit counts, we will use the following notation. We use $d(v)$ for the degree of vertex v. We will use $W(G)$, $D(G)$, $DP(G^{\rightarrow})$, and $DBP(G^{\rightarrow})$ for the total count of wedges, diamonds, directed 3-paths, and directed bipyramids respectively. These subgraphs are shown in Fig. 5.5.

5.4.1 Main theorem

Theorem 5.4.4. *There is an algorithm for exactly counting all VOCs for orbits* 0-72*, whose running time is* $O(W(G) + D(G) + DP(G^{\rightarrow}) + DBP(G^{\rightarrow}) + m + n)$.

This theorem is analogous to that of ESCAPE ([137]) which gives the same asymptotic running time for just total counting of 5-vertex subgraphs. We consider it quite significant that one gets the same asymptotic running time, despite the output being much larger and far more fine-grained. We stress that the `EVOKE` algorithm is significantly different than ESCAPE, since the orbit counts behave differently from total subgraph counts. The final proof is long, and is based on a collection of more than 50 equations for counting different orbits.

$$A^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 & -2 & -2 & -1 & 0 & 4 & 2 & -6 \\ 0 & 1 & 0 & 0 & -2 & 0 & -1 & -2 & 2 & 6 & -12 \\ 0 & 0 & 1 & 0 & 0 & -1 & -1 & 0 & 2 & 1 & -3 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 1 & -1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & -1 & 3 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -2 & 0 & 3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -2 & -2 & 6 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -2 & 3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 5.4: Matrix transforming non-induced vertex orbit counts for orbits 0-14 to induced counts

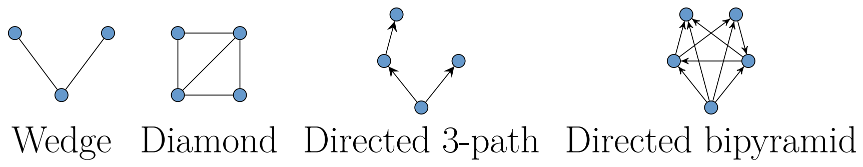


Figure 5.5: Fundamental patterns enumerated for orbit counting

5.5 Main ideas

`EVOKE` builds off the ideas in ESCAPE for total subgraph counts. First, we explain difficulties in directly applying previous techniques.

Pattern cutting: Intuitively, a 5-vertex pattern can be "cut" into smaller patterns that can be explicitly enumerated. An enumeration over these smaller patterns can then be used to get a subgraph count. As an example, consider the 4-path (H_9). By cutting at the center (green) vertex, one gets two wedges. Thus, we can basically square the number of wedges that end at a vertex, and then sum this to get the total number of 4-paths. (Not quite, there is some inclusion-exclusion required to "correct" this count, but it is fairly easy to work out.) But this fails for orbit counting. The 4-path has three distinct orbits, and the idea above only works for the green orbit.

This is even more problematic for patterns like H_{21}, H_{25}, H_{27}, H_{28}, where the removal of certain vertices does not "cut" the pattern into convenient smaller pieces. The main insight in ESCAPE was that all 5-vertex patterns have a convenient cutset of vertices, whose removal leads to fragments that can be easily enumerated. This is not true for orbits. We do have the freedom of choosing the convenient cutset.

From 4-edge orbit counts to 5-VOCs: Our main insight is that the suitable generalization of the pattern cutting approach connects 5-VOCs to 4-*edge* orbit counts. We essentially prove that many the orbit counts in Fig. 5.2 for a vertex v can be related (by non-trivial polynomial equations) to the *edge* orbits counts (of 4-vertex subgraphs) on edges incident to v. The edge orbits of 4-vertex subgraphs are given in Fig. 5.7. These edge orbits counts can be obtained by implementations of the Chiba-Nishizeki clique and 4-cycle counter [39], with extra inclusion-exclusion tricks to get all counts. `EVOKE` uses this as a preprocessing step. We will give more details in §5.7.

Careful indexing during enumeration: Even with the previous ideas, we still need an efficient implementation that can generate all the counts. We design a collection of vertex and edge indexed data structures, that are updated by an enumeration of the patterns shown in Fig. 5.5. Somewhat surprisingly, we show that as these patterns are enumerated, one can quickly update these data structures and generate all the orbit counts. This leads to Theorem 5.4.4.

5.6 The cutting framework for orbits

In this section, we describe the cutting framework for orbits. As mentioned earlier, this is a generalization of ideas in [137].

First, we formally define a match, which is a non-induced copy of H. For a set C where $C \subseteq V(H)$, we use $H|_C$ to denote the subgraph of H induced on C. We also denote the remaining graph after removing C from H, by $H \setminus C$.

Definition 5.6.1. A *match* of H in G is a bijection $\pi : T \to V(H)$ where $T \subset V$ and for any two vertices t_1 and t_2 in T, $(t_1, t_2) \in E$ if $(\pi(t_1), \pi(t_2)) \in E(H)$.

Definition 5.6.2. Fix an orbit $\theta = (H, S)$ and a vertex $v \in V$. We define $\mathcal{M}(v, \theta)$ to be the set of all (not necessarily distinct) matches $\pi : T \to V(H)$ of H, where $T \subset V$, such that $v \in T$ and $\pi(v) \in S$. We use $\mathrm{M}(v, \theta)$ to denote $|\mathcal{M}(v, \theta)|$.

Definition 5.6.3. For any orbit $\theta = (H, S)$ we define $\lambda = (H, i)$, where i is a vertex in S, as a representative of θ.
We use $r(\theta)$ to denote its representative (H, j), where j is the vertex with the smallest id in S.

Let $\lambda = (H, i)$ be a representative of an orbit θ. Abusing notation, for a vertex $v \in V$, we use $\mathcal{M}(v, \lambda)$ to denote the set of matches $\pi \in \mathcal{M}(v, \theta)$ where $\pi(v) = i$. Analogously, we use $\mathrm{M}(v, \lambda)$ to show $|\mathcal{M}(v, \lambda)|$. We can see that $\mathrm{M}(v, \theta) = \mathrm{sz}(\theta) \cdot \mathrm{M}(v, \lambda)$. Next, we define *fragments* in H, which are the result of cutting H using a cut set.

Definition 5.6.4. Let H be a subgraph pattern and consider a non-trivial cut set $C \subsetneq V(H)$. Let $S_1, S_2, \ldots$ be connected components of $H \setminus C$. The *fragments* of H obtained by removing C are the subgraphs of H induced by $C \cup S_1, C \cup S_2, \ldots$. We denote the set of these fragments by $\mathrm{Frag}_C(H)$.

A partial match $\pi : T \to V(H)$ is similar to a match, except that it is an injection, and is not surjective, thus $|T| < |V(H)|$.

Definition 5.6.5. A match $\pi : T \to V(H)$ *extends* a partial match $\sigma : T' \to V(H)$ if $T' \subset T$ and for any vertex t in T, $\pi(t) = \sigma(t)$. We denote the number of matches π of H that extend σ, by $\deg_H(\sigma)$.

Consider a match σ of $H|_C$. For σ to extend to a match of H, it is sufficient that it extends to disjoint matches of all fragments in $\text{Frag}_C(H)$. Merging these extensions leads to a match of H. If extension of σ to these fragments are not disjoint, merging them leads to a match of a different pattern H', which we call a *shrinkage.*

Definition 5.6.6. Let H, H' be subgraph patterns, $C \subsetneq V(H)$ be a cut set of H, and $\text{Frag}_C(H) = \{F_1, F_2, \ldots, F_{|\text{Frag}_C(H)|}\}$. Let $\tau : H|_C \to H'$ be a partial match of H'. For each $F_i \in \text{Frag}_C(H)$, let $\pi_i : F_i \to H'$ be a partial match of H' in H that extends τ. We call $\{\tau, \pi_1, \pi_2, \ldots, \pi_{|\text{Frag}_C(H)|}\}$ a C-shrinkage of H into H' if for each edge $(s,t) \in E(H')$, there exists an edge (a,b) in fragment $F_j \in \text{Frag}_C(H)$ such that $\pi_j(a) = s$ and $\pi_j(b) = t$.
We use $\text{Shrink}_C(H)$ to denote the set of patterns (up to isomorphism) H', to which there exist at least a C-shrinkage from H.

Definition 5.6.7. Consider graph H, $H' \in \text{Shrink}_C(H)$, $\lambda = (H,i)$, and $\lambda' = (H',j)$. We define $\text{numSh}_C(\lambda,\lambda')$ to be the number of distinct C-shrinkages of H into H' where $\tau(i) = j$.

Lemma 5.6.8. *Consider a pettern H, an orbit $\theta = (H,S)$, a representative $\lambda = (H,i)$ of θ, and a cut set C in H such that $i \in C$. Then,*

$$\begin{aligned} M(v,\lambda) = &\sum_{\sigma \in \mathcal{M}(v,(H|_C,i))} \prod_{F \in Frag_C(H)} \deg_F(\sigma) \\ &- \sum_{H' \in Shrink_C(H)} \sum_{\substack{\theta' \in orb(H'), \\ \lambda' = r(\theta')}} numSh_C(\lambda,\lambda') \cdot DM(v,\lambda') \end{aligned}$$

Proof. Consider any match σ of $H|_C$ in $\mathcal{M}(v,(H|_C,i))$, and all sets of maps $\{\pi_1, \ldots, \pi_{|\text{Frag}_C(H)|}\}$ where π_ℓ is a copy of $F_\ell \in \text{Frag}_C(H)$ that extends σ. The number of such sets is exactly:

$$\sum_{\sigma \in \mathcal{M}(v,(H|_C,i))} \prod_{F \in \mathrm{Frag}_C(H)} \deg_F(\sigma) \tag{5.1}$$

Consider one of these sets of maps $\{\pi_1, \ldots, \pi_{|\mathrm{Frag}_C(H)|}\}$, let $V(\pi_\ell)$ be the set of vertices that π_ℓ maps to F_ℓ. If all $V(\pi_\ell) \setminus V(C)$ are disjoint, we get a match in $\mathcal{M}(v, \lambda)$. Therefore, Each match of H in $\mathcal{M}(v, \lambda)$ is counted exactly one time in (5.1). But for each orbit $\theta' = (H', S')$ where $H' \in \mathrm{Shrink}_C(H)$, we have also counted some matches in $\mathcal{M}(v, \theta')$. The number of distinct matches of θ' involving v is $\mathrm{DM}(v, \theta')$. Let $\lambda' = (H', j)$ be $r(\theta')$. The number of distinct C-shrinkages of H into H', where $\tau(i) = j$, is $\mathrm{numSh}_C(\lambda, \lambda')$. Thus, per each orbit θ', we have counted $\mathrm{numSh}_C(\lambda, \lambda') \cdot \mathrm{DM}(v, \lambda')$ matches which should now be subtracted from (5.1).

The reason we considered only distinct matches of λ' involving v is that the shrinkage from H to H' gives us the labeling of H' and the set of maps $\{\pi_1, \ldots, \pi_{|\mathrm{Frag}_C(H)|}\}$, which resulted in counting this match, dictates the match. Also, notice that the shrinkage determines the vertex in H' that v is mapped to. That is why we consider number of shrinkages for a representative of θ'. □

Corollary 5.6.9. *As mentioned, $M(v, \theta) = sz(\theta) \cdot M(v, \lambda)$. Therefore, we can derive $DM(v, \theta)$, which is the number of distinct matches of θ, as follows: $DM(v, \theta) = sz(\theta) \cdot M(v, \lambda)/|Aut(H)|$.*

Application of Lemma 5.6.8 for vertex orbit 26: We will show how this lemma works applying it to H_{12} and computing VOCs for a vertex $v \in V$. Let $\theta_{26} = (H, S)$, where $S = \{2, 3\}$ and H is as shown in Fig. 5.6, denote orbit 26. Let the representative λ_{26} be $(H, 2)$.

Let triangle $\{1, 2, 3\}$ be the cut set C. So, $\mathrm{Frag}_C(H) = \{F_1, F_2\}$ as we can see in Fig. 5.6. Let $\hat{\lambda} = (H|_C, 2)$ be a representative of orbit 3 (the only orbit in

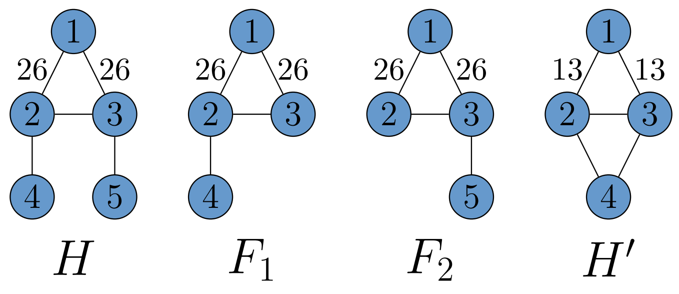


Figure 5.6: Application of Lemma 5.6.8 for vertex orbit 26

the cut set). Every triangle in G incident to v is a match in $\mathcal{M}(v, \hat{\lambda})$. Each such triangle has two mappings to $H|_C$. consider triangle $\{u, v, w\}$ in G. Vertex v has to be matched to vertex 2, therefore one match (A) is $\sigma(u) = 1$, $\sigma(v) = 2$, and $\sigma(w) = 3$, and the other match (B) is $\sigma(u) = 3$, $\sigma(v) = 2$, and $\sigma(w) = 1$. For match (A), $\deg_{F_1}(\sigma) \cdot \deg_{F_2}(\sigma) = (d(v) - 2)(d(w) - 2)$, and $\deg_{F_1}(\sigma) \cdot \deg_{F_2}(\sigma) = (d(v) - 2)(d(u) - 2)$ for match (B).

The only possible shrinkage of H is to a diamond H', as shown in Fig. 5.6. Let orbit $\theta_{13} = (H', S')$, where $S' = \{2, 3\}$, show orbit 13. We can see that in any C-shrinkage of H into H', $\tau(2) \in S'$. Let $\lambda_{13} = (H', 2)$ be a representative of θ_{13}. Notice that $\text{numSh}_C(\lambda_{26}, \lambda_{13}) = 2$. In one case we set $\tau(1) = 1$, $\tau(2) = 2$, $\tau(3) = 3$, $\pi_1(4) = 4$, and $\pi_2(5) = 4$. In the other case, we set $\tau(1) = 4$, $\tau(2) = 2$, $\tau(3) = 3$, $\pi_1(4) = 1$, and $\pi_2(5) = 1$. The set of maps $\{\tau, \pi_1, \pi_2\}$ in both cases forms a C-shrinkage of H into H' where $\tau(2) = 2$.

$$
\begin{aligned}
\text{M}(v, \lambda_{26}) = \sum_{t=\langle u,v,w\rangle \text{ triangle}} & [(d(v) - 2)((d(u) - 2) \\
& + (d(w) - 2))] - 2 \cdot \text{DM}(v, \lambda_{13}) \qquad (5.2)
\end{aligned}
$$

Note that $\text{sz}(\theta_{26}) = 2$ and H has two automorphisms, so (by Corollary 5.6.9) $\text{DM}(v, \theta_{26}) = \text{M}(v, \lambda_{26})$.

5.7 Getting orbit counts

Lemma 5.6.8 gives us a collection of more than fifty equations similar to (5.2). For each of them, we verify that they can be computed through an enumeration of the patterns in Fig. 5.5, assuming that all edge orbits of Fig. 5.7 are available. For readability, we move the details of equations for computing 5-VOC, their runtime analysis, and the final proof of Theorem 5.4.4 to §5.7.1 of the Appendix, but we give the details of 4 vertex and edge orbit counts and a few example of 5-VOCs equations in this section. We will prove the following theorem for 4-vertex orbit counting.

Theorem 5.7.1. *All vertex and edge orbit counts for 4-vertex patterns can be obtained in time $O(W(G) + D(G) + m + n)$.*

Getting 4-VOCs: The easiest way to demonstrate our framework, is to apply it to orbits in 4-vertex patterns with up to 4 vertex (orbits 0-14). For each vertex v in G, let $T(v)$, $C_4(v)$, and $K_4(v)$ denote the number of triangles incident to v, the number of 4-cycles incident to v, and the number of 4-cliques incident to v, respectively. For each edge $e = (u, v)$ in G, let $T(e)$, $C_4(e)$, and $K_4(e)$ denote the number of triangles incident to e, the number of 4-cyles incident to e, and the number of 4-cliques incident to e, respectively. For each triangle t, let $K_4(t)$ denote the number of 4-cliques including t. In [137], Pinar-Seshadhri-Vishal have shown that there is an algorithm that in time $O(W(G)+D(G)+m+n)$, computes (for all vertices u, edges $e = (v, w)$, and triangles t): all $T(v)$, $T(e)$, $C_4(v)$, $C_4(e)$, $K_4(v)$, $K_4(e)$, and $K_4(t)$. Their algorithm also obtains for every edge e, the list of triangles incident to e. Vertex orbit counts of patterns with up to 4 vertices can be computed using the equations presented in Lemma 5.7.2.

Lemma 5.7.2. *For $i \in 0, \ldots, 14$, let $\lambda_i = r(\theta_i)$. Then, for each vertex $u \in V$,*

$DM(u, \lambda_0) = d(u)$

$DM(u, \lambda_1) = \sum_{v \in N(u)} d(v) - 1$

$DM(u, \lambda_2) = \binom{d(u)}{2}$

$DM(u, \lambda_3) = T(u)$

$DM(u, \lambda_4) = \sum_{v \in N(u)} DM(v, \lambda_1) - 2DM(u, \lambda_2) - 2DM(u, \lambda_3)$

$DM(u, \lambda_5) = DM(u, \lambda_1)(d(u) - 1) - 2DM(u, \lambda_3)$

$DM(u, \lambda_6) = \sum_{v \in N(u)} \binom{d(v)-1}{2}$

$DM(u, \lambda_7) = \binom{d(u)}{3}$

$DM(u, \lambda_8) = C_4(u)$

$DM(u, \lambda_9) = \sum_{v \in N(u)} DM(v, \lambda_3) - T(u, v)$

$DM(u, \lambda_{10}) = \sum_{v \in N(u)} T(u, v)(d(v) - 2)$

$DM(u, \lambda_{11}) = DM(u, \lambda_3)(d(u) - 2)$

$DM(u, \lambda_{12}) = \sum_{t=(u,v,x)} T(v, x) - 1$

$DM(u, \lambda_{13}) = \sum_{v \in N(u)} \binom{T(u,v)}{2}$

$DM(u, \lambda_{14}) = K_4(u)$

Proof. For θ_{12}, we use a triangle as the cut set. The remaining component is a vertex, which forms a triangle with the edge (v, x). After mapping a triangle $t = (u, v, x)$ to the cut set, we need to select a vertex to extend the copy of the cut set to a copy of θ_{12}. The number of such vertices are equal to $T(v, x) - 1$, which is the number of all triangles incident to the edge (v, x) except t.

The rest of the equations, either have a vertex or an edge as the cut set, or are computed directly, such as θ_2, which are easy to follow. □

Getting edge orbit counts of 4-vertex subgraphs: There are eleven edge orbits for 4-vertex subgraphs as shown in Fig. 5.7. For an edge (u, v), we use $E_i((u, v))$ to denote the count of the ith edge orbit (where i is from Fig. 5.7).

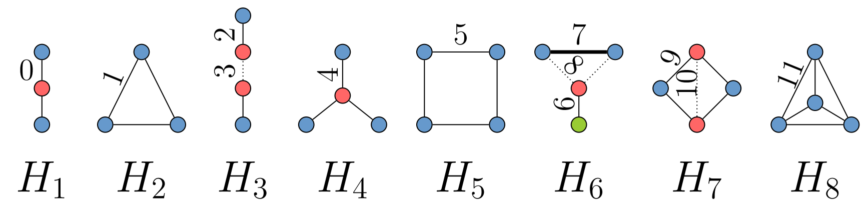


Figure 5.7: All edge orbits of 4-vertex patterns. Within each pattern, edges of the same line style form an edge orbit.

Definition 5.7.3. Given an edge $e = (v, u)$ in graph G and edge orbit i which lies in pattern H, a match of edge orbit i involving edge (v, u), is a non-induced copy of H in G such that e is mapped to an edge in edge orbit i.

Let the vertex orbits of the two end points of edge orbit i, be $\theta_a = (H, S_a)$ and $\theta_b = (H, S_b)$ where $a \geqslant b$. From the definition of automorphism, it is clear that a match of edge orbit i, involving edge $e = (v, u)$, maps v to a vertex in S_a and u to a vertex in S_b, or vice versa. Similar to vertex orbits, we call two matches of an edge orbits equivalent if one can be obtained from the other by applying an automorphism. We use $E_i(\langle v, u\rangle)$ to denote the number of distinct matches of edge orbit i involving $e = (v, u)$, where v is mapped to a vertex in S_a and u is mapped to a vertex in S_b. If $a = b$, then $E_i(\langle v, u\rangle) = E_i(\langle u, v\rangle)$, thus we use $E_i((v, u))$ to denote the number of distinct matches of orbit i involving e.

Edge orbit counts of patterns with up to 4 vertex can be computed using the equations presented in Lemma 5.7.4.

Lemma 5.7.4. *Let* $\lambda_3 = r(\theta_3)$. *For each edge* $(u, v) \in E(G)$,

$E_0(\langle u, v\rangle) = d(u) - 1$

$E_1((u, v)) = T(u, v)$

$E_2(\langle u, v\rangle) = \sum_{x \in N(u) \setminus v} [d(x) - 1] - E_1(u, v)$

$E_3((u, v)) = (d(u) - 1)(d(v) - 1) - E_1(u, v)$

$E_4(\langle u, v\rangle) = \binom{d(u)-1}{2}$

$E_5((u,v)) = C_4(u,v)$

$E_6(\langle u,v\rangle) = DM(u,\lambda_3) - E_1(u,v)$

$E_7((u,v)) = \sum_{t=(u,v,x)} d(x) - 2$

$E_8(\langle u,v\rangle) = E_1(u,v)(d(u)-2)$

$E_9(\langle u,v\rangle) = \sum_{t=(u,v,x)} E_1(u,x) - 1$

$E_{10}((u,v)) = \binom{T(u,v)}{2}$

$E_{11}((u,v)) = K_4(u,v)$

Proof. For E_7 and E_9, we need to enumerate the triangles incident to (u,v). This could be obtained by the algorithm presented in [137] in time $O(W(G)+m+n)$ for all edges. The rest of the edge orbits are either computed directly or have a vertex or an edge as a cut set and are easy to follow. □

Finally, we can prove Theorem 5.7.1.

Proof of Theorem 5.7.1. All vertex and edge orbits of patterns with up to 4-vertices could be obtained from equations in Lemma 5.7.2 and Lemma 5.7.4. For all vertices v, all edges e, and all triangles t, we can get $T(v)$, $T(e)$, $C_4(v)$, $C_4(e)$, $K_4(v)$, $K_4(e)$, $K_4(t)$, and also for all edges e we can obtain the list of triangles incident to e in $O(W(G)+D(G)+m+n)$ [137]. Assuming we have these counts, the rest of the vertex and edge orbit counts are either computed directly, or use a vertex, edge, or a triangle a cut set. Therefore, we can obtain all the other orbit counts for 4-vertex patterns in $O(W(G)+m+n)$ extra time. Overall, it takes $O(W(G)+D(G)+m+n)$ time to get all vertex and edge orbit counts of 4-vertex patterns. □

Getting 5-VOCs: We demonstrate the main ideas through a number of examples.

- Orbit 26: The pattern cutting framework gives (5.2). We can precompute and store degrees at all vertices. During an enumeration of all triangles, one can compute the summand for each triangle. The triangles can be enumerated in $O(W(G))$ time (indeed, it can be done even faster using orientations). Orbit 13 belongs to a 4-vertex pattern, so $\mathrm{DM}(2, \theta_{13})$ is obtained from Theorem 5.7.1.

- Orbit 37: let $\lambda_{37} = r(\theta_{37})$ and $\lambda_{12} = r(\theta_{12})$, then

$$\mathrm{DM}(u, \lambda_{37}) = \sum_{v \in N(u)} [E_5((u, v))(d(v) - 2)] - 2\mathrm{DM}(u, \lambda_{12}). \tag{5.3}$$

After storing E_5-values on each edge, one can get this VOC by a triangle enumeration. Orbit 12 belongs to a 4-vertex pattern.

- Orbit 68:

$$\mathrm{DM}(u, \theta_{68}) = \sum_{\substack{v,w \text{ where} \\ \langle u,v,w \rangle \text{ is a wedge}}} \binom{D(u, v, w)}{2} \tag{5.4}$$

This is a challenging orbit to count. The value $D(u, v, w)$ is the number of diamonds (H_7) that involves the vertices u, v, w. It is too expensive to precompute and store all these values, but we can do it piecemeal. With knowledge of triangles, we can enumerate all diamonds involving a fixed vertex u. This can be used to find all the relevant values. Overall, the total time is a diamond enumeration and a wedge enumeration.

Overall, this technique can analogously handle all orbits, barring the 5-cycle and 5-clique (each of which as a single orbit). The 5-clique can be directly enumerated in time $O(DBP(G^{\rightarrow}))$, a consequence of the classic Chiba-Nishizeki algorithm [39] and explicitly proven in [137].

Dealing with 5-cycles: This is a special case, and handled in the following theorem. This is a significant strengthening of 5-cycle counter in ESCAPE, which only gave a global count in the same running time.

Theorem 5.7.5. *Vertex orbit counts for the 5-cycle can be computed in time $O(W(G) + DP(G^{\rightarrow}) + m + n)$.*

Proof. As shown in Fig. 5.8, there are three different 5-cycles DAGs up to isomorphism. Each 5-cycle has exactly one directed 3-path as shown in Fig. 5.8, such that the remaining wedge is not an in-in wedge. In the figure, this directed 3-path is labeled i, j, k, l, and w is the center vertex of the wedge. By a directed wedge enumeration, we can precompute the number of such wedges between all pairs of vertices. We enumerate over the directed 3-paths: for every directed 3-path we get between vertices i and l, we already know the number of relevant directed wedges between i and l as shown in Fig. 5.8. This allows us to increment the orbit counts for the vertices i, j, k, l, by the number of wedges. Notice that an edge between i and k or between j and l could result in such a directed wedge. We can check the existence of these two edges using hashed edges of G. For each such edge, we should decrement the orbit counts for the vertices i, j, k, l by one.

This process does not update the orbit count for vertex w. Let $P(i, l)$ be the number of directed 3-paths from i to l as shown in Fig. 5.8. To compute the orbit counts for vertex w, we enumerate in-out and out-out wedges between i and l, and add $P(i, l)$ to the 5-cycle orbit count of vertex w. Notice that the 3-paths (corresponding to $P(i, l)$) potentially intersect with the wedge under consideration. Any such intersection would result in couting a tailed triangle instead of a 5-cycle. We need to subtract out the count of these tailed-triangles. Any in-out wedge from i to l corresponds to a 5-cycle of type (c), in which case we count correctly. An in-out wedge from l to i corresponds to a 5-cycle of type (b); in this case we will count each tailed triangle of type (1), shown in Fig. 5.9, as a 5-cycle while passing over (l, j, i) wedge. An out-out wedge between i and l corresponds to a 5-cycle

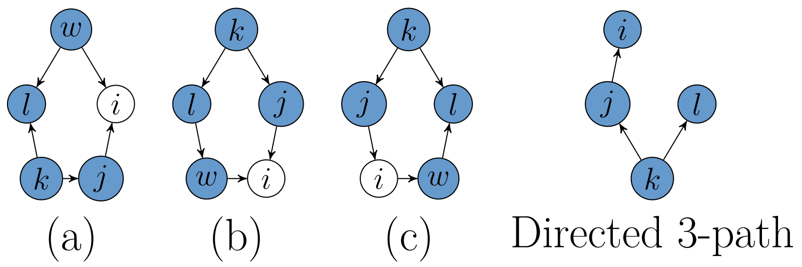


Figure 5.8: All different 5-cycle DAGs up to isomorphism. There is only one directed 3-path as shown on the right side in each 5-cycle DAG where the remaining wedge is not an in-in wedge.

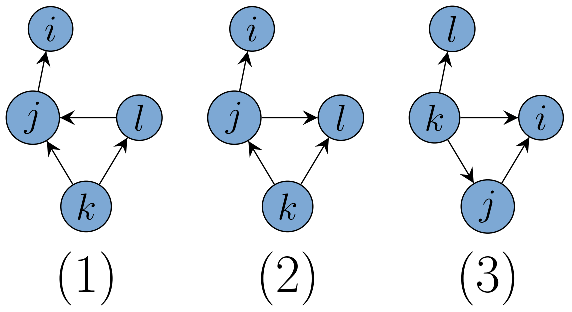


Figure 5.9: Directed tailed triangles counted while counting 5-cycles

type (a); in this case we will count each tailed triangle of type (2) as a 5-cycle while passing over (i, j, l), and count each tailed triangle of type (3) while passing over (i, k, l).

We can easily get the tailed triangle counts corresponding to each wedge using the per-edge tailed triangle counts that we already have. All in all, we can get VOCs for the 5-cycle in the stated time. □

5.7.1 Details of Getting 5-VOCs

In this section we provide the formulas for computing 5-VOCs derived from Lemma 5.6.8 and also analysis of run time for computing 5-VOCs using this equations. This will also prove Theorem 5.4.4. In the run time analysis for computing orbit θ_i, we assume that we have already obtained the counts for θ_0-θ_{i-1} and all edge orbit counts E_0-E_{11}. Most of the equations in Theorem 5.7.6 have a vertex, an edge, or a triangle as the cut set and are straightforward to follow. we give a proof sketch for the rest of the equations and how they are obtained from Lemma 5.6.8.

Theorem 5.7.6. *For $i \in 0, \dots, 72$, let $\lambda_i = r(\theta_i)$. Let $TT(u,v)$ denote the count of tailed triangles incident to edge (u,v), where u is the tail vertex (θ_9) and v is in θ_{10}. The value $D(u,v,w)$ denotes the number of diamonds (H_7) that involves the vertices u, v, w such that u and w are the vertices incident to the chord. Let $D(u,v)$ be the number of diamonds where u and v are not incident to the chord. And finally, let $W(u,v)$ be the number of wedges between vertices u and v. Then, for each vertex $u \in V$,*

$$DM(u,\lambda_{15}) = \sum_{v \in N(u)} [DM(v,\lambda_4)] - DM(u,\lambda_5) - 2DM(u,\lambda_{11}) - 2DM(u,\lambda_8)$$

$$DM(u,\lambda_{16}) = DM(u,\lambda_4)(d(u)-1) - DM(u,\lambda_{10}) - 2DM(u,\lambda_8)$$

$$DM(u,\lambda_{17}) = \binom{DM(u,\lambda_1)}{2} - DM(u,\lambda_3) - DM(u,\lambda_6)$$
$$- DM(u,\lambda_8) - DM(u,\lambda_{10})$$

$$
\begin{aligned}
DM(u,\lambda_{18}) &= \sum_{v\in N(u)} [DM(v,\lambda_6)] - 3DM(u,\lambda_7) - DM(u,\lambda_{10}) \\
&= \sum_{v} \left[W(u,v)\binom{d(v)-1}{2} \right] - DM(u,\lambda_{10})
\end{aligned}
$$

$$
DM(u,\lambda_{19}) = \sum_{v\in N(u)} [DM(v,\lambda_5)] - DM(u,\lambda_4) - DM(u,\lambda_u) - DM(u,\lambda_{10})
$$

$$
DM(u,\lambda_{20}) = \sum_{v\in N(u)} \left[(d(u)-1)\binom{d(v)-1}{2} \right] - 2DM(u,\lambda_{10})
$$

$$
DM(u,\lambda_{21}) = \sum_{v\in N(u)} \left[(d(v)-1)\binom{d(u)-1}{2} \right] - DM(u,\lambda_{11})
$$

$$
DM(u,\lambda_{22}) = \sum_{v\in N(u)} \binom{d(v)-1}{3}
$$

$$
DM(u,\lambda_{23}) = \binom{d(u)}{4}
$$

$$
DM(u,\lambda_{24}) = \sum_{v\in N(u)} [DM(v,\lambda_{10})] - DM(u,\lambda_{10}) - 2DM(u,\lambda_{11}) - 2DM(u,\lambda_{12})
$$

$$
DM(u,\lambda_{25}) = \sum_{t=(u,v,x)} [(d(v)-2)(d(x)-2)] - DM(u,\lambda_{12})
$$

$$DM(u,\lambda_{26}) = \sum_{t=(u,v,x)} [(d(u)-2)((d(x)-2)+(d(v)-2))] - 2DM(u,\lambda_{13})$$

$$\begin{aligned}DM(u,\lambda_{27}) &= \sum_{v\in N(u)} [DM(v,\lambda_9)] - DM(u,\lambda_{11}) - 2DM(u,\lambda_{13})\\ &= \sum_{v} [W(u,v)DM(v,\lambda_3)] - DM(u,\lambda_{13}) - DM(u,\lambda_9) - DM(u,\lambda_3)\end{aligned}$$

$$\begin{aligned}DM(u,\lambda_{28}) &= \sum_{v\in N(u)} [(d(u)-1)DM(v,\lambda_3)]\\ &- 2DM(u,\lambda_3) - 2DM(u,\lambda_{11}) - 2DM(u,\lambda_{12})\end{aligned}$$

$$\begin{aligned}DM(u,\lambda_{29}) &= \sum_{t=(u,v,x)} [DM(v,\lambda_1) + DM(x,\lambda_1)]\\ &- 4DM(u,\lambda_3) - DM(u,\lambda_{10}) - 2DM(u,\lambda_{11})\\ &- 2DM(u,\lambda_{12}) - 2DM(u,\lambda_{13})\end{aligned}$$

$$\begin{aligned}DM(u,\lambda_{30}) &= \sum_{v\in N(u)} [(d(v)-1)DM(u,\lambda_3)]\\ &- 2DM(u,\lambda_3) - DM(u,\lambda_{10}) - 2DM(u,\lambda_{13})\end{aligned}$$

$$\begin{aligned}DM(u,\lambda_{31}) &= \sum_{v\in N(u)} [(d(v)-1)DM(v,\lambda_3)]\\ &- 2DM(u,\lambda_9) - DM(u,\lambda_{10}) - 2DM(u,\lambda_3)\end{aligned}$$

$$DM(u,\lambda_{32}) = \sum_{t=(u,v,x)} \left[\binom{d(v)-2}{2} + \binom{d(x)-2}{2} \right]$$

$$DM(u,\lambda_{33}) = DM(u,\lambda_3)\binom{d(u)-2}{2}$$

$$DM(u,\lambda_{35}) = \sum_{v\in N(u)} [DM(v,\lambda_8)] - 2R_8(u) - R_{13}(u)$$

$$DM(u,\lambda_{36}) = \sum_{v} \binom{W(u,v)}{2}(d(v)-2) - DM(u,\lambda_{13})$$

$$DM(u,\lambda_{37}) = \sum_{v\in N(u)} [E_5(u,v)(d(v)-2)] - 2DM(u,\lambda_{12})$$

$$DM(u,\lambda_{38}) = DM(u,\lambda_8)(d(u)-2) - DM(u,\lambda_{13})$$

$$DM(u,\lambda_{39}) = \sum_{v\in N(u)} [DM(v,\lambda_{13})] - DM(u,\lambda_{13}) - 2DM(u,\lambda_{12})$$

$$DM(u,\lambda_{40}) = \sum_{v\in N(u)} E_9(u,v)(d(v)-3)$$

$$DM(u,\lambda_{41}) = \sum_{v\in N(u)} \binom{E_1(u,v)}{2}(d(v)-3)$$

$$DM(u,\lambda_{42}) = \sum_{v\in N(u)} \binom{E_1(u,v)}{2}(d(u)-3)$$

$$\begin{aligned}DM(u,\lambda_{43}) = &\sum_{t=(u,v,x)} [(DM(v,\lambda_3)-1)+(DM(x,\lambda_3)-1)]\\ &-2DM(u,\lambda_{12})-2DM(u,\lambda_{13})\end{aligned}$$

$$DM(u,\lambda_{44}) = \binom{DM(u,\lambda_3)}{2} - DM(u,\lambda_{13})$$

$$DM(u,\lambda_{45}) = \sum_{v\in N(u)} [DM(v,\lambda_{12})] - 2DM(u,\lambda_{13}) - 3DM(u,\lambda_{14})$$

$$DM(u,\lambda_{46}) = \sum_{t=(u,v,x)} [E_7(v,x)] - DM(u,\lambda_{11}) - 3DM(u,\lambda_{14})$$

$$DM(u,\lambda_{47}) = DM(u,\lambda_{12})(d(u)-2) - 3DM(u,\lambda_{14})$$

$$\begin{aligned}DM(u,\lambda_{48}) = &\sum_{t=(u,v,x)} [(E_1(u,v)-1)(d(x)-2)\\ &+(E_1(u,x)-1)(d(v)-2)] - 6DM(u,\lambda_{14})\end{aligned}$$

$$DM(u,\lambda_{49}) = \sum_{v,x\in N(u)} \binom{W(v,x)-1}{2}$$

$$DM(u, \lambda_{50}) = \sum_{v} \binom{W(u,v)}{3}$$

$$DM(u, \lambda_{51}) = \sum_{v} TT(u,v)(W(u,v) - 1) - 2DM(u, \lambda_{12})$$

$$DM(u, \lambda_{52}) = \sum_{t=(u,v,x)} [E_5(v,x)] - 2DM(u, \lambda_{13})$$

$$DM(u, \lambda_{53}) = \sum_{t=(u,v,x)} [E_5(u,v) + E_5(u,x)] - 2DM(u, \lambda_{13}) - 2DM(u, \lambda_{12})$$

$$DM(u, \lambda_{54}) = \sum_{t=(u,v,x)} \binom{E_1(v,x) - 1}{2}$$

$$DM(u, \lambda_{55}) = \sum_{v \in N(u)} \binom{E_1(u,v)}{3}$$

$$DM(u, \lambda_{56}) = \sum_{v \in N(u)} [DM(v, \lambda_{14})] - 3DM(u, \lambda_{14})$$

$$DM(u, \lambda_{57}) = \sum_{v \in N(u)} E_{11}(u,v)(d(v) - 3)$$

$$DM(u, \lambda_{58}) = DM(u, \lambda_{14})(d(u) - 3)$$

$$DM(u,\lambda_{59}) = \sum_{t=(u,v,x)} [E_9(\langle x,v\rangle) + E_9(\langle v,x\rangle)] - 2DM(u,\lambda_{13}) - 6DM(u,\lambda_{14})$$

$$DM(u,\lambda_{60}) = \sum_{v\in N(u)} E_9(\langle v,u\rangle)(E_1(u,v)-1) - 6DM(u,\lambda_{14})$$

$$\begin{aligned} DM(u,\lambda_{61}) = &\sum_{t=(u,v,x)} [(E_1(u,v)-1)(E_1(u,x)-1)] \\ &- 3DM(u,\lambda_{14}) \end{aligned}$$

$$DM(u,\lambda_{62}) = \sum_{v,x\in N(u)} [D(v,x)] - DM(u,\lambda_{13})$$

$$DM(u,\lambda_{63}) = \sum_{v} D(u,v)(W(u,v)-2)$$

$$DM(u,\lambda_{64}) = \sum_{v,x\in N(u)} D(v,u,x)(W(v,x)-2)$$

$$DM(u,\lambda_{65}) = \sum_{t=(u,v,x)} [E_{11}(v,x)] - 3DM(u,\lambda_{14})$$

$$DM(u,\lambda_{66}) = \sum_{\langle u,v,x,y\rangle \textit{ is 4-clique}} [E_1(v,x) + E_1(v,y) + E_1(x,y)]$$

$$DM(u, \lambda_{67}) = \sum_{v \in N(u)} E_{11}(u, v)(E_1(u, v) - 2)$$

$$DM(u, \lambda_{68}) = \sum_{\substack{v,x \text{ where} \\ \langle u,v,x \rangle \text{ is a wedge}}} \binom{D(u, v, x)}{2}$$

$$DM(u, \lambda_{69}) = \sum_{v,x \in N(u)} \binom{D(u, v, x)}{2}$$

$$DM(u, \lambda_{70}) = \sum_{\langle u,v,x,y \rangle \text{ is 4-clique}} K_4(v, x, y) - 1$$

$$DM(u, \lambda_{71}) = \sum_{t=(u,v,x)} \binom{K_4(t)}{2}$$

Proof. For orbits θ_{36}, θ_{49}, θ_{50}, θ_{51}, θ_{63}, and θ_{64}, we need the counts of wedges which have the vertex at hand in the middle as in the equation for θ_{49}, or at one of the ends, as in the equation for θ_{36}. We do not precompute and store these counts for all vertices as it could be expensive. But we can get these counts while counting, by enumerating wedges in time $O(W(G))$ [137]. In the equation for θ_{51}, we need the counts of $TT(u, v)$. But this is easy to get while enumerating the wedges between u and v, and using the triangle per-edge counts for edge (x, v), where (u, x, v) is a wedge. Equation of orbits θ_{62}, θ_{63}, θ_{64}, θ_{68}, and θ_{69} require the counts of diamonds. These counts are too expensive to precompute and store for all the vertices, so we do it while computing the counts for each vertex, using triangle counts for each edge. To compute the coutns of θ_{70} and θ_{71}, we need to use the counts of 4-cliques incident to each triangle t, which we can get in $O(W(G) + D(G) + m + n)$ [137]. $\square$

Finally, we can prove Theorem 5.4.4.

Proof of Theorem 5.4.4. By Theorem 5.7.1, we know that we can obtain all the counts for orbits θ_0-θ_{14} and E_0-E_{11} in time $O(W(G)+D(G)+m+n)$. Also for each triangle t, we can get $K_4(t)$ and for each edge e, the list of triangles incident to e in time $O(W(G) + D(G) + m + n)$ [137]. We need to show that computing orbit counts for θ_{15}-θ_{72} takes time $O(W(G) + D(G) + DP(G^{\rightarrow}) + DBP(G^{\rightarrow}) + m + n)$. By Theorem 5.7.5, θ_{34} (the only orbit in 5-cycle) counts can be obtained in time $O(W(G) + DP(G^{\rightarrow}) + m + n)$ and the counts for θ_{72} (the only orbit in 5-clique) takes $O(DP(G^{\rightarrow}))$ to compute [137]. So, we only need to show that computing 5-VOCs except 5-cycle and 5-clique, using each of the equations in Theorem 5.7.6 takes time $O(W(G) + D(G) + DP(G^{\rightarrow}) + DBP(G^{\rightarrow}) + m + n)$.

We divide the set of orbits of 5-vertex patterns to categories with different runtime. When analysing the runtime for equation Of θ_i, we assume that we have access to the counts for θ_0-θ_{i-1} and all edge orbit counts E_0-E_{11}, as we have stored them previously. Orbit in each category are shown in Tab. 5.1.

- Orbits that we can count in time $O(n)$ for all vertices:

Computing these vertex orbits, we only need to pass over vertices in G, and then it is straightforward to get the counts for each vertex in constant time using the equations in Theorem 5.7.6.

- Orbits that we can count in time $O(m + n)$ for all vertices: In this category, to compute the counts for each vertex in G, we enumerate its neighborhood. This takes time $O(m + n)$ overall.

- Orbits that we can count in time $O(W(G) + m + n)$: Enumerating all the wedges suffices to compute the counts for θ_{36}, θ_{49}, and θ_{50} using their equations.

Table 5.1: Time for computing 5-VOCs for all vertices using equations in Theorem 5.7.6

5-VOC runtime	Orbits
$O(n)$	θ_{16}, θ_{17}, θ_{23}, θ_{33}, θ_{38}, θ_{44}, θ_{47}, θ_{58}
$O(m+n)$	θ_{15}, θ_{18}, θ_{19}, θ_{20}, θ_{21}, θ_{22}, θ_{24}, θ_{27} θ_{28}, θ_{30}, θ_{31}, θ_{35}, θ_{37}, θ_{39}, θ_{40}, θ_{41}, θ_{42}, θ_{45}, θ_{55}, θ_{56}, θ_{57}, θ_{60}, θ_{67}
$O(W(G)+m+n)$	θ_{25}, θ_{26}, θ_{29}, θ_{32}, θ_{36}, θ_{43}, θ_{46}, θ_{48}, θ_{49}, θ_{50}, θ_{51}, θ_{52}, θ_{53}, θ_{54}, θ_{59}, θ_{61}, θ_{65}

While enumerating wedges to get the counts of orbit θ_{51} for vertex u, we need the count of tailed triangles incident to edge (u, v), where u is the tail vertex (θ_9) and v is in θ_{10}. But this is easy to get using triangle counts, while (u, v) is the wedge at hand during the wedge enumeration.

The rest of the orbits in this category could be obtained by enumerating all the triangles, which is possible in $O(W(G))$.

- Orbits that we can count in time $O(W(G) + D(G) + m + n)$: For θ_{66} and θ_{70}, we need to enumerate 4-cliques, which takes time $O(W(G) + D(G) + m + n)$ [137]. Getting counts of orbit θ_{71} requires enumeration of triangles, but for each triangle t at hand, we need to get $K_4(t)$, which is overall possible in time $O(W(G) + D(G) + m + n)$.

To get the rest of the orbit counts in this category, we need to enumerate diamonds. Similar to the way we enumerate wedges while enumerating neighbors of a vertex, instead of precomputing and storing all the wedge counts, we enumerate diamonds while enumerating wedges, using triangle counts that we already have.

□

5.8 Experimental Results

We implement EVOKE in C++. We ran experiments on a commodity machine from AWS EC2: R5d.2xlarge, which has Intel Xeon Platinum 8175M CPU @ 2.50GHz with 4 cores and 1024K L2 cache (per core), 34MB L3 cache, and 64GB memory. For running EVOKE on the com-orkut graph (117M edges), we used the more powerful R5d.12xlarge EC2 instance (with 384GB RAM). We actually run ORCA for 5-vertex patterns on the larger machine for any instance with more than 1M edges. The EVOKE package is available at [2] as open source code.

We used large graph datasets from the Network Repository [149], SNAP [101], and Citation Network Dataset [1, 172]. We removed directions from edges, and omitted duplicates and self loops. Tab. 5.2 includes the number of nodes, edges, and triangles for all the graphs we used. We also run EVOKE on wiki-en-cat, a bipartite graph from the KONECT network repository [3, 4, 193].

As mentioned earlier, we compare our results with ORCA [75] which is the state of the art algorithm for computing all 5-VOCs. The runtimes of ESCAPE, EVOKE, and ORCA is given in Tab. 5.2. We also state the time for just counting 4-VOCs. When we do not report a time for ORCA, it implies that either ORCA ran out of memory or ran more than 1000 times the EVOKE running time. In all the results, the time includes the I/O, so we account for the time required to print the (large) output into files. As mentioned later, there is a parallel implementation of EVOKE, but all run times reported are of the sequential implementation (to have a fair comparison with ORCA).

Running time of EVOKE: As seen in Tab. 5.2, for many instances of counting 5-VOCs, we simply cannot get results with ORCA. For all graphs larger than web-google-dir, ORCA-5 runs out of memory even on the more powerful EC2 instance, or was stopped after a thousand times the corresponding EVOKE running

Table 5.2: Properties of the graphs and runtime of ESCAPE, EVOKE, and ORCA

				Runtimes in seconds					
Dataset (sorted by increasing $\lvert E\rvert$)	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert T\rvert$	ESC-4	EVOKE-4	ORCA-4	ESC-5	EVOKE-5	ORCA-5
soc-brightkite	56.7K	213K	494K	0.43	0.59	1.77	4.69	7.74	562.84
ia-email-EU-dir	265K	364K	267K	0.49	1.29	9.38	5.91	13.18	17.36K
tech-RL-caida	191K	607K	455K	0.68	1.29	2.99	4.65	10.03	595.44
Citation-network V1	2.17K	631K	248K	0.69	2.57	42.91	2.89	8.93	275.15
ca-coauthors-dblp	540K	1.52M	444M	266.81	287.89	510.77	20.69K	26.91K	171.32K
DBLP-Citation-network V5	470K	2.08M	1.38M	2.59	10.18	13.04	19.17	40.76	2.92K
Citation-network V2	660K	3.02M	1.9M	4.11	11.57	28.42	32.78	69.36	7.52K
wiki-en-cat	2.04M	3.8M	0	3.13	12.61	114.31	22.85	86.58	-
web-google-dir	876K	4.32M	13.4M	4.76	10.03	45.40	45.86	104.88	76.37K
web-wiki-ch-internal	1.93M	8.95M	18.19M	30.11	65.45	655.15	1.22K	1.87K	-
tech-as-skitter	1.69M	11.1M	28.8M	28.91	68.25	827.46	853.21	1.46K	-
web-hudong	1.98M	14.43M	21.61M	48.20	85.83	1.78K	2.41K	3.45K	-
web-baidu-baike	2.14M	17.01M	25.2M	61.4	148.11	2.92K	2.66K	4.27K	-
tech-ip	2.25M	21.64M	2.3M	92.03	277.87	79.96K	18.14K	40.57K	-
soc-LiveJournal1	4.85M	42.85M	285.73M	401.07	599.43	1.30K	28.46K	36.57K	-
com-orkut	3.72M	117.18M	627.58M	1.23K	2.77K	7.37K	137.73K	143.41K	-

time has passed (shown by blue bars in Fig. 5.1). When ORCA does give results, the speedup of EVOKE is easily in the orders of hundreds. Fig. 5.1 gives the speedup as a chart. EVOKE makes 5-VOCs computation feasible, for graphs with tens of millions of edges. ORCA is unable to process any graph in that size range. Even for the large com-orkut graph with over 100M edges, EVOKE gets all counts in two days.

As an aside, for counting 4-VOCs, EVOKE runs typically in minutes, consistent with previous work [125, 137].

Comparison with ESCAPE: Theorem 5.4.4 shows that the asymptotic upper bound given for ESCAPE in [137] is also an asymptotic upper bound for EVOKE run time. We are able to validate this in practice. Fig. 5.10a shows the ratio of runtime of EVOKE over ESCAPE for 5-vertex patterns. Note that ESCAPE counts subgraphs and EVOKE computes orbit counts for orbits in those subgraphs. As we can see in Fig. 5.10a, in all our experiments the ratio is typically below 2 and never more than 4. We believe this finding to be significant, since obtaining the richer information of 5-VOCs is just as feasible as getting exact total counts.

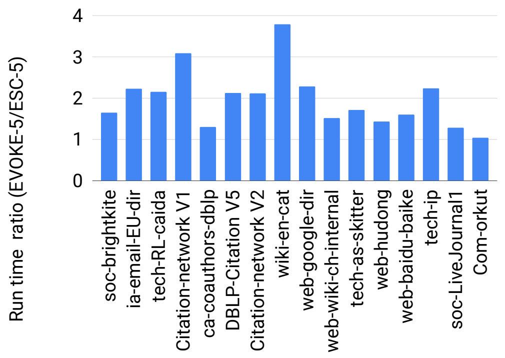
Runtime distribution and parallel speedup: Typically, a few orbits take the lion's share of the running time. Fig. 5.10b shows the split-up of running time over the various orbits. We group them into four classes: the 5-clique, the 5-cycle, the orbits of H_{25} and H_{27} (these require diamond enumerations), and everything else. By and large, just the 5-cycle and 5-clique orbits account for half the time.

It is straightforward to parallelize the computation of these different groups of orbits. For the non-induced setting, these are simply independent computations. We perform this parallelism, and present the speedup in Fig. 5.10c. As expected, there is roughly a 1.5-2 factor speedup, corresponding to the most expensive orbit to compute.

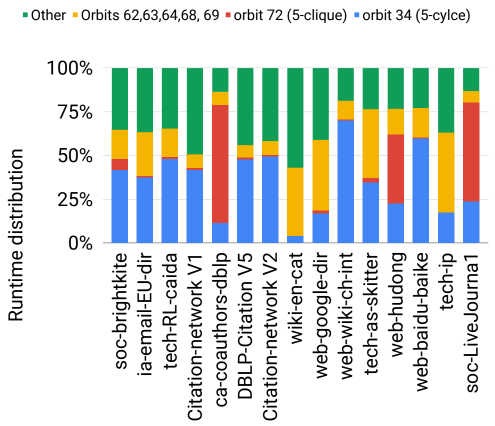
VOC distributions: As a demonstration of `EVOKE`, we plot the VOC distribution (also called graphlet degree distribution) of various graphs. To get cleaner figures, we plot the Complementary Cumulative Distribution (CCD): for x, we plot the fraction of vertices whose orbit count is at least x. This is plotted for Orbit 70 (in induced 5-clique minus edge) in Fig. 5.11a and for Orbit 17 (center of induced 4-path) in Fig. 5.11b. We stress that these induced counts are typically harder to obtain than the non-induced counts.

For Orbit 17, we observe that the largest count is more than trillions, showing the challenges in exact counting. Also the distribution of `tech-as-skitter` has a bigger dropoff in the tail, which may be indicative of the path structures in AS networks. The `web-google-dir` graph has a sharp dropoff at the end as well. We see that Orbit 70 distributions are quite different over the graphs, unlike Orbit 17, where the tails are similar for three of the graphs. The counts in `Citation-network V2` are much smaller, suggesting there are not many 5-cliques missing edges.

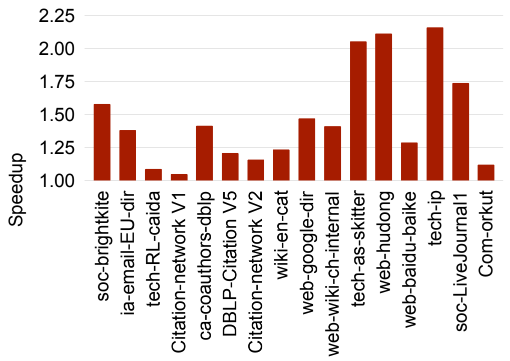
In Fig. 5.11c, for the graph `web-google-dir`, we plot the VOC of the three



(a) Ratio of runtime represented as EVOKE-5/ESC-5 demonstrates Theorem 5.4.4



(b) Runtime distribution over 5-vertex orbits

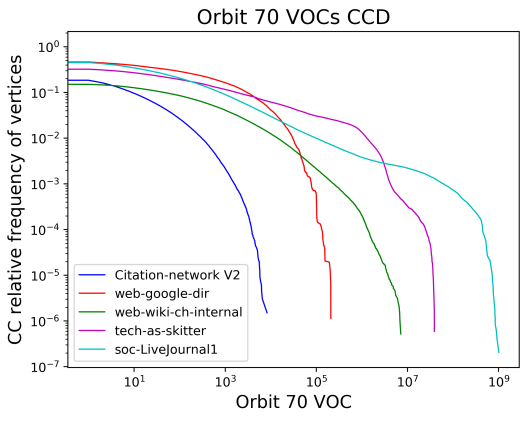


(c) Speedup achieved by parallel computation of 5-VOCs

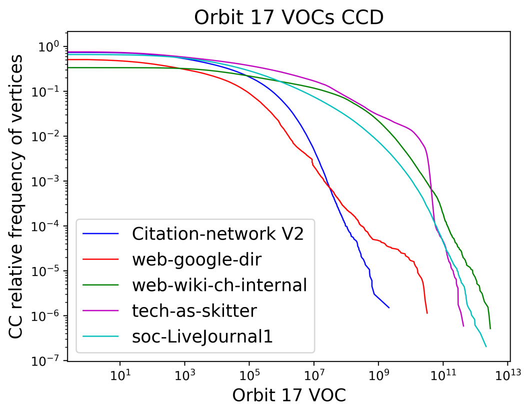
Figure 5.10: Empirical analysis of `EVOKE` runtime

different orbits (15-17) of the induced 4-path. Observe how the distribution for Orbit 15 (the start/end) is significantly different from Orbit 17 (the center), underscoring the fine-grained information that orbits provide over vanilla counts.

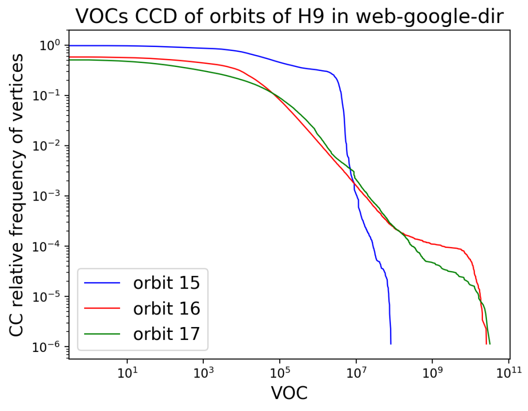
Graph mining through orbit counts: As another demonstration, we focus on the citation network `DBLP-Citation-network V5`, where we have metadata associated with vertices (papers). We found that the paper with the largest count of Orbit 17 (center of induced 4-path) is the classic book “C4.5: Programs for Machine Learning” by Ross Quinlan. Furthermore, the paper participating in the



(a) Orbit 70 VOCs CCD



(b) Orbit 17 VOCs CCD



(c) VOCs CCD of orb(H_9)

Figure 5.11: (a), (b): VOCs comp. cum. distribution (CCD) of orbits. For count x, we plot the fraction of vertices with orbit count at least x. (c) For `web-google-dir`, we plot the VOC CCD for all orbits of the 4-path. Observe that the distributions for the start/end (orbit 15) and the center (orbit 17) behave differently.

most 5-cliques is the highly cited VLDB 94 paper "Fast Algorithms for Mining Association Rules in Large Databases" by Agarwal and Srikant. It is interesting that the orbit counts can immediately give us semantically significant vertices.

Chapter 6

Generalized Temporal Triangle Counting

In this chapter we introduce $(\delta_{1,3}, \delta_{1,2}, \delta_{2,3})$-temporal triangles, a generalized notion of temporal triangle counting. Our main contribution is `DOTTT`, an algorithm for counting $(\delta_{1,3}, \delta_{1,2}, \delta_{2,3})$-temporal triangles that runs in $O(m\kappa \log m)$. `DOTTT` is up to twice as fast as the state-of-the-art and has an asymptotic running time closer to that of static triangle counting.

6.1 Problem Description

The input is a directed temporal graph $T = (V, E)$. Each edge is a tuple of the form (u, v, t) where u and v are vertices in the temporal graph, and t is a timestamp. For notational convenience, we assume all timestamps in a temporal network are unique integers.

We introduce our notion of $(\delta_{1,3}, \delta_{1,2}, \delta_{2,3})$-temporal triangles.

Definition 6.1.1. Let $e_1 = (u_1, v_1, t_1), e_2 = (u_2, v_2, t_2)$, and $e_3 = (u_3, v_3, t_3)$, be three directed temporal edges where the induced static graph on them is a triangle, and $t_1 < t_2 < t_3$.

(e_1, e_2, e_3) is a $(\delta_{1,3}, \delta_{1,2}, \delta_{2,3})$-*temporal triangle* if $t_2 - t_1 \leqslant \delta_{1,2}$, $t_3 - t_2 \leqslant \delta_{2,3}$, and $t_3 - t_1 \leqslant \delta_{1,3}$.

Thus, we specify timestamp differences between *every* pair of edges. When one also considers the direction of edges, there exist eight different types of temporal triangles as shown in Fig. 6.1. These types are distinguished by temporal ordering of edges and their direction. Thus, for any choice of $(\delta_{1,3}, \delta_{1,2}, \delta_{2,3})$, there are eight different types of temporal triangles (one corresponding to each figure in Fig. 6.1).

We observe that the notion in Definition 6.1.1 subsumes most existing temporal triangle definitions. Specifically, a $(\delta_{1,3}, \delta_{1,3}, \delta_{1,3})$-temporal triangle becomes a $\delta_{1,3}$-temporal triangles as defined in PBL [128]. Temporal triangles with respect to the temporal motif definition by by Kovanen et al. in [96] consider timestamp differences between consecutive edges in temporal ordering. By our definition, $(2\Delta, \Delta, \Delta)$-temporal triangles capture these types of temporal triangles. Although, the definition in [96] is more restrictive and requires that all edges incident to a node are consecutive events of that node. Most existing temporal triangle counting literature uses these definitions [104, 108, 168, 185].

We describe a simple example to see how Definition 6.1.1 offers richer temporal information. Let us measure time in hours, so $(2, 1, 1)$-temporal triangle is one where the first and second edge (of the triangle) are at most 1 hour apart, and similarly for the second and third edge. Now consider $(1.5, 1, 1)$-temporal triangles. The time gap between the first and second edge (as well as the second and third) is again 1 hour, but the entire triangle must occur within 1.5 hours. There is a significant difference between these cases, but previous definitions of temporal triangles would not distinguish these.

We note that more general temporal motifs, beyond triangles, have been defined. Yet, to the best of our knowledge, most fast algorithms that scale to millions of edges have been designed for triangles. Paranjape et al. specialized algorithm for 3-edge triangle motifs (temporal triangles) is up to 56x faster than their general motif counting algorithm [128]. Our focus was on scalable algorithms, and hence, on triangle counting. We believe that generalizing Definition 6.1.1 (and our DOTTT algorithm) for general motifs would be compelling future work.

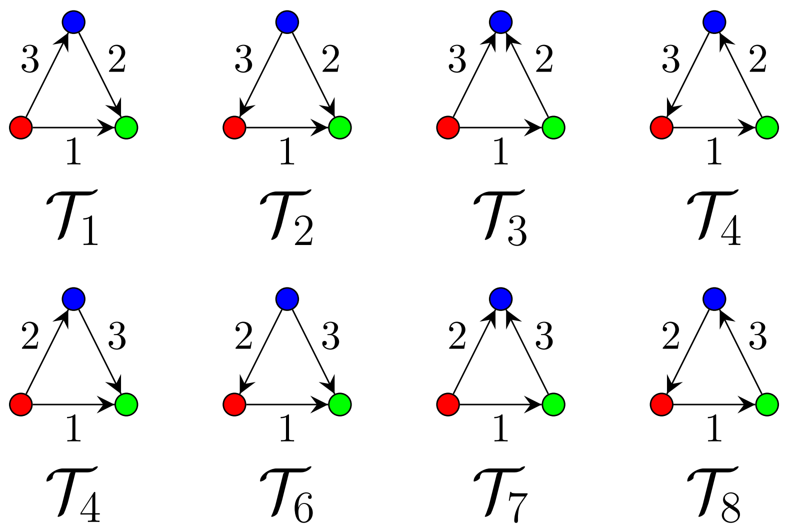


Figure 6.1: All possible temporal triangle types. The start point of the first edge (in temporal ordering of edges) is shown in red and the end point in green.

6.2 Main Contributions

Our main result is the *Degeneracy Oriented Temporal Triangle Totaler* algorithm, DOTTT that counts $(\delta_{1,3}, \delta_{1,2}, \delta_{2,3})$-temporal triangles as defined in Definition 6.1.1. The running time is only a logarithmic overhead over static triangle counting. We detail our contributions below.

Theoretically bridging gap between temporal and static triangle counting: Our main theorem is the following.

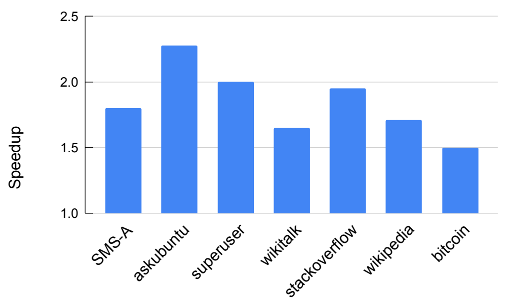
Theorem 6.2.1. *Given $\delta_{1,3}, \delta_{1,2}$ and $\delta_{2,3}$, the* `DOTTT` *algorithm exactly counts each of the eight types of $(\delta_{1,3}, \delta_{1,2}, \delta_{2,3})$-temporal triangles (Fig. 6.1) in a temporal graph in $O(m\kappa \log m)$ time. (Here, m is the total number of temporal edges, and κ is the degeneracy of the underlying static graph.)*

Observe that, up to a logarithmic factor, our theoretical running time for temporal triangle counting matches the $O(m\kappa)$ bound for static triangle counting. As mentioned earlier, the previous best bound was $O(m\tau^{1/2})$. We stress that there is no dependence on the time intervals $(\delta_{1,3}, \delta_{1,2}, \delta_{2,3})$.

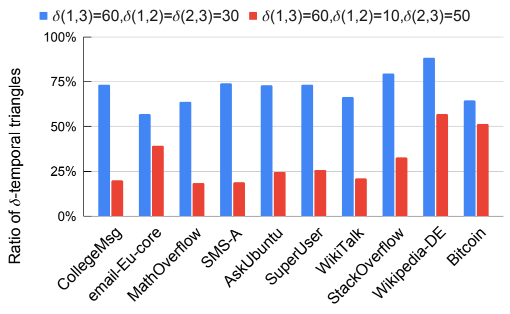
The idea of degeneracy orientations is tailored to static graphs, and one of our contributions is to show it can help for temporal triangle counting. A key insight in `DOTTT` is to process (underlying) static edges in the exact order of the Chiba-Nishizeki algorithm, but carefully consider neighboring edges to capture all temporal triangles. By a non-trivial combinatorial analysis, we can prove that number of times that a temporal edge is processed is upper bounded by κ. We need additional data structure tricks to get the counts efficiently, leading to an extra logarithmic factor.

Excellent practical behavior of `DOTTT`: `DOTTT` consistently determines temporal triangle counts in less than ten minutes for datasets with tens of millions of edges. We only use a single commodity machine with 64GB memory, without any parallelization. We directly compare `DOTTT` with the state-of-the-art PBL algorithm. Our algorithm is consistently faster, and as illustrated in Fig. 6.2a we typically get a factor 1.5 speedup for larger graphs. (We note that `DOTTT` can count a more general class of temporal triangles.)

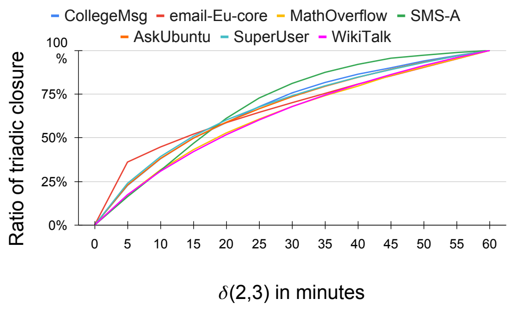
We note that for the largest dataset in our experiments, Bitcoin (515.5M edges), `DOTTT` only uses 64GB memory and runs in less than an hour, while existing methods ran out of memory (details in §6.8).



(a) Speedup



(b) Expressivity



(c) Triadic closure over time

Figure 6.2: (a): The Speedup of DOTTT for counting $(\delta_{1,3}, \delta_{1,2}, \delta_{2,3})$-temporal triangles over the PBL algorithm for counting $\delta_{1,3}$-temporal triangles. (b): We fix $\delta_{1,3}$ to 1 hr. Blue bars show the ratio of (1 hr, 30 mins, 30 mins)-temporal triangles to (1 hr, 1 hr, 1 hr)-temporal triangle. The red bars illustrate the ratio for the case of (1 hr, 10 mins, 50 mins)-temporal triangles and is more restrictive. (c): We fix $\delta_{1,3} = 2$ hrs and $\delta_{1,2} = 1$ hr. At t we plot the ratio of (2 hrs, 1 hr, t)-temporal triangles to (2 hrs, 1 hr, 1hr)-temporal triangles.

Richer triadic information from $(\delta_{1,3}, \delta_{1,2}, \delta_{2,3})$-temporal triangles: We demonstrate how $(\delta_{1,3}, \delta_{1,2}, \delta_{2,3})$-temporal triangles can give a richer network analysis method. Consider Fig. 6.2b. For a collection of temporal datasets, we generate the counts of (1 hr, 30 min, 30 min)-temporal triangle counts, as well as those for (1 hr, 10 min, 50 min)-temporal triangles. We plot these numbers as a ratio of (1 hr, 1 hr, 1 hr)-temporal triangles. Across the datasets, the ratios are at most 75%. The red bars are typically at most 25%, showing the extra power of Definition 6.1.1 in distinguishing temporal triangles.

We note here that for each dataset, `DOTTT` has the same running time for obtaining the counts for (1 hr, 30 min, 30 min)-temporal triangle and (1 hr, 10 min, 50 min)-temporal triangles, as it has no dependency on the time intervals $(\delta_{1,3}, \delta_{1,2}, \delta_{2,3})$.

An interesting study is presented in Fig. 6.2c. The transitivity and clustering coefficients are fundamental quantities of study in network science. In temporal graphs, in addition to these measure, the time it takes for a wedge (2-path) to close could also be of importance. (Zingnani et al. proposed the triadic closure delay metric that capture the time delay between when a triadic closure is first possible, and when they occur [200].) In Fig. 6.2c, we fix $\delta_{1,3} = 2$ hrs and $\delta_{1,2} = 1$ hr. We then vary $\delta_{2,3}$ from zero to 60 minutes, and plot the ratio of $(\delta_{1,3}, \delta_{1,2}, \delta_{2,3})$-temporal triangles to (2hrs, 1hr, 1hr)-temporal triangles. We can see the trends in triadic closure with respect to the time for the third edge. We observe that, by and large, half the triangles are formed within 20 minutes of the first two edges appearing. And by 30 minutes, almost 75% of these triangles are formed. These are examples of triadic analyses enabled by `DOTTT`.

6.3 Main challenges

In a temporal graph, the number of temporal edges is typically two to three times the number of underlying static edges. Since most triangle counting algorithms are based on some form of wedge enumerations, this leads to a significant increase in the number of edges. One method used for temporal triangle counting is to simply prune the temporal edges based on the time period [108, 167]. But such algorithms have a dependency on the time period and are inefficient for large time periods.

Another significant challenge is the multiplicity of an individual edge can be extremely large. The same edge often occurs many hundreds to thousands of times in a temporal network (in the BitCoin network, there is an edge appearing 447K times Tab. 6.2). These edges create significant bottlenecks for enumeration methods. It is not clear how efficient methods on the underlying static graphs (which ignores multiplicities) can help with this problem. Triangle counting often works by finding a wedge (2-path) and checking for the third edge. With multiple temporal edges between the same pair of vertices, this method requires many edge lookups. Paranjape et al. used a clever idea to process edges on a pair of vertices $O(\tau^{1/2})$ times. The challenge is to bound it by the degeneracy of the graph.

The time constraints expressed by $(\delta_{1,3}, \delta_{1,2}, \delta_{2,3})$-temporal triangles create additional challenges. A clever wedge enumeration exploiting the degeneracy may produce wedges containing the first and second edges of the triangle, the first and third, or the second and third. This makes the lookup (or counting) of possible "matches" for the remaining edge challenging, since it appears we need to look at all multiple edges. On the other hand, if we enumerated wedges that only involved the first and second edge, we cannot benefit for the efficiencies of degeneracy-based methods. Some of these problems can be circumvented for (δ, δ, δ)-temporal triangles, but the general case is challenging.

Overall, we can state the main challenge as follows. Fast triangle counting methods (such as degeneracy based methods) necessarily ignore time constraints while generating wedges, making it hard to look for the "closing" edge. On the other hand, a method that exploits the timestamps by (say) pruning cannot get the efficiency gains of degeneracy based methods. One of the insights of `DOTTT` is a resolution of this tension.

6.4 Related Work

There is rich history of work on triangle counting in static graphs. Various algorithm for triangle and motif counting in attributed graphs have also been proposed [69, 120, 136, 145, 150, 191]. Here we only focus on temporal networks and refer the reader to [10] and the tutorial [160] for a more detailed list of related work.

Graph orientation, in particular degeneracy ordering, is a classic idea in counting triangles and motifs in static graphs, pioneered by Chiba-Nizhizeki [38]. Recently, there has been a number of triangle counting and motif counting algorithms inspired by these techniques [51, 81, 83, 125, 129, 137]. The main benefit of degeneracy ordering is that the out-degree of each vertex becomes small when we orient the static graph based on this ordering.

Kovanen et al. called two temporal edges ΔT-adjacent if they share a vertex and the difference of their timestamps are at most ΔT [96]. In their definition of temporal motifs, temporal edges must represent consecutive events for a node. Redmond et al. gave an algorithm for counting δ-temporal motifs but their algorithm does not take the temporal ordering of edges into account [142], and only counts motifs where incoming edges occur before outgoing edges. Gurukar et al. present a heuristic for counting temporal motifs [70].

More related to our work, Paranjape-Benson-Leskovec defined the δ-temporal motifs where all edges occur inside a time period δ and also the temporal ordering of edges are taken into account [128]. They gave a general algorithm for counting k-node ℓ-edge motifs in temporal networks. The main idea behind their algorithm is a moving time window of size δ over the sequence of all temporal edges for each static motif matching the underlying static motif of the temporal motif of interest. For temporal triangles, their algorithm runs in $O(\tau m)$ time where τ is the number of triangles in the underlying static graph of the input temporal graph T, as it might enumerate temporal edges on static edges with high multiplicity $O(\tau)$ time. They also presented a specialized, more efficient algorithm for counting 3-edge temporal triangles that runs in time $O(\tau^{1/2}m)$. We call their algorithm PBL and use it as our baseline.

Mackey et al. presented a backtracking algorithm for counting δ-temporal motifs that maps edges of the motif to the edges of the host graph one by one in temporal (chronological) ordering. For each edge, it only searches through edges that occur in the correct temporal ordering and respect the time gap restriction. [108]. Unlike the PBL algortihm and ours, this algorithm could be inefficient for large values of δ as its runtime depends on the value of δ.

Liu et al. [105] introduced a comparative survey of temporal motif models. Boekhout et al. gave an algorithm for counting δ-temporal multi-layer temporal motifs [30]. Li et al., developed an algorithm for counting temporal motifs in heterogeneous information networks [102]. Petrovic et al. gave an algorithm for counting causal paths in time series data on networks [134].

There has also been recent progress on approximating the counts of temporal motifs and triangles [168, 185]. Particularly , Liu et al. presented a sampling framework for approximating the counts of $\delta_{1,3}$-temporal motifs [104].

6.5 Preliminaries

The input graph is a directed temporal graph that we denote by $T(V, E)$. Let $|V| = n$ and $|E| = m$. Temporal graph T is presented as a collection of m directed temporal edges $e = (u, v, t)$ where $u, v \in V$, and t is the timestamp for edge e where $t \in \mathbb{R}$. We use $t(e)$ to denote the timestamp of a temporal edge e. Note that there could be multiple temporal edges on the same pair of nodes.

We assume that all the timestamps in T are unique. This assumption leads to the clean definition of different types of temporal triangles (Fig. 6.1), but is not a necessity of our algorithm. To be more specific, our algorithm also works for temporal graph including temporal edges with equal timestamp.

We denote the underlying undirected static graph of T as $G = (V, E_s)$ and put $|E_s| = m_s$. Two vertices in the static graph G are connected if there is at least one temporal edge between them. Formally, $E_s = \{\{u, v\} \mid \exists t : (u, v, t) \in E \vee (v, u, t) \in E\}$. For $v_1, v_2 \in V$, let $\sigma((v_1, v_2))$ denote the temporal multiplicity, that is the number of temporal edges on $\{v_1, v_2\}$ directed from v_1 to v_2.

As shown in Fig. 6.1, there are eight different types of temporal triangles. The time restrictions $\delta_{1,3}$, $\delta_{1,2}$, and $\delta_{2,3}$ is not involved in definition of these types and could be applied to each of them. Note that these different types account for all possible ordering of temporal edges in the triangle in addition to their directions.

6.6 Main Ideas

Our algorithm first enumerates static triangles in G, the underlying static graph of the input temporal graph T. Let $\{u, v, w\}$ be a static triangle. We consider all possible temporal orderings as shown in Fig. 6.3, and all possible orientations as shown in Fig. 6.4, for a temporal triangle corresponding to $\{u, v, w\}$. A temporal ordering and a temporal orientation together determine the type of the temporal triangle. For example π_1 and ρ_8 correspond to $\mathcal{T}_1$. Tab. 6.1 lists all possible pairs of temporal ordering and orientations and their corresponding type of temporal triangle as a function ψ.

We store the input temporal edges of the input temporal graph T in a data structure in the CSR format. Thus, we can assume that we have constant time access to temporal edges on each pair of vertices for each direction in the order of increasing timestamps. Let π denote the temporal ordering, and ρ denote the orientation for which we want to count the temporal triangles. In this section, from here we only consider temporal edges that follow the orientation ρ.

Assume that the timestamps of two of the edges of a temporal triangle corresponding to the static triangle $\{u, v, w\}$ is given. WLOG, assume that these temporal edges correspond to $\{u, v\}$ and $\{u, w\}$. We can use a binary search to find the number of temporal edges on the pair $\{v, w\}$ that are compatible with the two given temporal edges. Note that compatibility of timestamps is determined by the timestamp of edges and the temporal ordering π. Thus, all we need is to enumerate all possible pairs of temporal edges on $\{u, v\}$ and $\{u, w\}$. This could be an expensive enumeration if both these static edges have high multiplicity of temporal edges.

We show that we can obtain the counts of temporal triangles on $\{u, v, w\}$ without enumeration of all possible pairs of temporal edges on $\{u, v\}$ and $\{u, w\}$. Let $e_1, \ldots, e_{\sigma(\rho(\{u,v\}))}$ denote the sequence of temporal edges in the order of increasing timestamp on $\{u, v\}$, and $e'_1, \ldots, e'_{\sigma(\rho(\{u,w\}))}$ denote that of $\{u, w\}$. We enumerate each of these sequences of temporal edges separately, and for each edge we store cumulative counts of compatible temporal edge on $\{v, w\}$. In other words, for each edge e in these two sequence, we store the counts of edges e_3 on $\{v, w\}$ that are compatible with e or any other temporal edge in the same sequence with a smaller timestamp.

Then we enumerate the temporal edges on $\{u, v\}$, and for each edge e_i we use binary search to find the sequence of temporal edges $e'_j, \ldots, e'_k$, in increasing order of timestamp, on $\{v, w\}$ that are compatible with e_i. We use the cumulative counts of compatible edges on $\{v, w\}$ that we stored for e_i, e'_k, and e'_j to compute the counts of all temporal triangles on $\{u, v, w\}$ that include e_i.

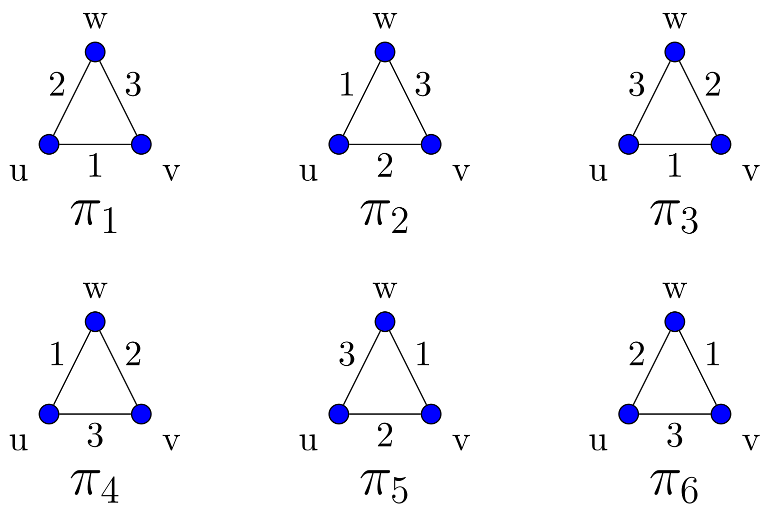


Figure 6.3: All possible ordering of temporal edges of a temporal triangle corresponding to a static triangle $\{u, v, w\}$.

Although we avoid the enumeration of pairs of temporal edges on $\{u, v\}$ and $\{u, w\}$, our algorithm could still be inefficient. The reason is that static edges $\{u, v\}$ could have high multiplicity of temporal edges and also participate in a large number of static triangles. Here is where we use the power of vertex ordering and graph orientation techniques.

`DOTTT` enumerates static triangles in $G_{\prec}$, and when processing a static triangle $\{u, v, w\}$ where u comes first in the degeneracy ordering $\prec$ of G, it only enumerates temporal edges on $\{u, v\}$ and $\{u, w\}$. Thus, each temporal edge on a pair $\{x, y\}$ where $x \prec y$, is processed only for static triangles where the third vertex is in the out-neighborhood of x. But we know that the out-degree of each vertex is bounded by κ in $G_{\prec}$. Therefore, each such temporal edge on $\{x, y\}$ is processed $O(\kappa)$ times.

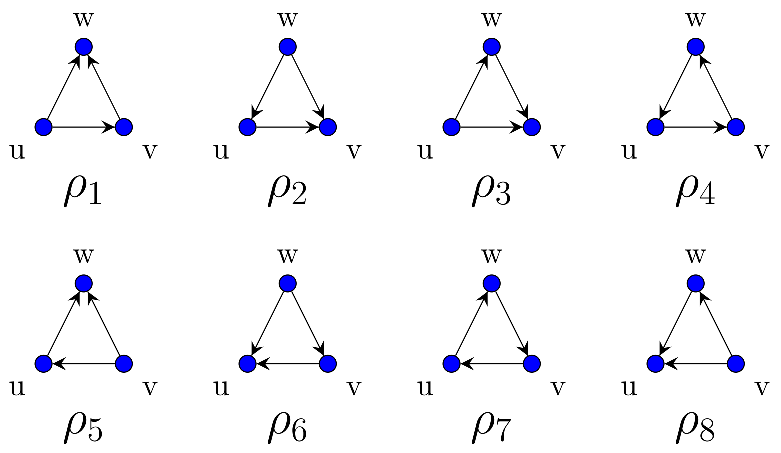


Figure 6.4: All possible orientations of temporal edges of a temporal triangle corresponding to a static triangle $\{u, v, w\}$.

Table 6.1: Conversion from temporal ordering and orientation to temporal triangle type.

ψ	ρ_1	ρ_2	ρ_3	ρ_4	ρ_5	ρ_6	ρ_7	ρ_8
π_1	$\mathcal{T}_7$	$\mathcal{T}_6$	$\mathcal{T}_5$	$\mathcal{T}_8$	$\mathcal{T}_3$	$\mathcal{T}_2$	$\mathcal{T}_4$	$\mathcal{T}_1$
π_2	$\mathcal{T}_5$	$\mathcal{T}_3$	$\mathcal{T}_7$	$\mathcal{T}_4$	$\mathcal{T}_6$	$\mathcal{T}_1$	$\mathcal{T}_8$	$\mathcal{T}_2$
π_3	$\mathcal{T}_3$	$\mathcal{T}_2$	$\mathcal{T}_1$	$\mathcal{T}_4$	$\mathcal{T}_7$	$\mathcal{T}_6$	$\mathcal{T}_8$	$\mathcal{T}_5$
π_4	$\mathcal{T}_1$	$\mathcal{T}_7$	$\mathcal{T}_3$	$\mathcal{T}_8$	$\mathcal{T}_2$	$\mathcal{T}_5$	$\mathcal{T}_4$	$\mathcal{T}_6$
π_5	$\mathcal{T}_6$	$\mathcal{T}_1$	$\mathcal{T}_2$	$\mathcal{T}_8$	$\mathcal{T}_5$	$\mathcal{T}_3$	$\mathcal{T}_4$	$\mathcal{T}_7$
π_6	$\mathcal{T}_2$	$\mathcal{T}_5$	$\mathcal{T}_6$	$\mathcal{T}_4$	$\mathcal{T}_1$	$\mathcal{T}_7$	$\mathcal{T}_8$	$\mathcal{T}_3$

6.7 Our Main Algorithm

In this section we describe our algorithm for getting $(\delta_{1,3}, \delta_{1,2}, \delta_{2,3})$-temporal triangles counts. Let $T = (V, E)$ be the input directed temporal graph given as a list of temporal edges sorted by timestamps. Although not necessary for our algorithm, assuming that edges are given in increasing order of timestamp is common in temporal networks as the edges are recorded in their order of occurrence [128].

We first extract the static graph $G(V, E_s)$ from T. Then, we obtain the degeneracy ordering of G, denoted by $\prec$ using the algorithm by Matula and Beck [112], and orient the edges of G with respect to $\prec$ to get the DAG $G_\prec$. We start by enumerating static triangles in $G_\prec$. This can be done in $O(m_s\kappa)$ where κ is the degeneracy of G [38, 137].

Note that all triangles in $G_\prec$ are acyclic as $G_\prec$ is a DAG, so each triangle in G correspond to an acyclic triangle in $G_\prec$. In order to enumerate all triangles in G, we enumerate all directed edges in $G_\prec$, and for each directed edge (u, v) we enumerate $N^+(u)$. For each vertex $w \in N^+(u)$, we check whether $\{u, v, w\}$ is a triangle by checking the existence of an edge between v and w.

We call vertex u in a static triangle $\{u, v, w\}$ the *source vertex* if $u \prec v$ and $u \prec w$. Let $\{u, v, w\}$ be the triangle being processed while enumerating triangles in $G_\prec$. WLOG, assume u is the source vertex in $\{u, v, w\}$. Thus, the number of times we visit $\{u, v\}$ or $\{u, w\}$ in a static triangle are limited by $d^+_{G_\prec}(u)$ that is bounded by κ. But the number of times we visit the static edge $\{v, w\}$ is not bounded by κ, so we want to avoid enumerating temporal edges on $\{v, w\}$. Next, we show how to count the number of $(\delta_{1,3}, \delta_{1,2}, \delta_{2,3})$-temporal triangles corresponding to the static triangle $\{u, v, w\}$.

We define the temporal ordering of a temporal triangle corresponding to a static triangle $\{u, v, w\}$ as a mapping $\pi : \{1, 2, 3\} \to \{\{u, v\}, \{u, w\}, \{v, w\}\}$. There are six different possible temporal orderings as shown in Fig. 6.3.

We define the orientation of a temporal triangle corresponding to a static triangle $\{u, v, w\}$ as a mapping ρ from each pair of vertices of $\{u, v, w\}$ to one of the two possible ordered pairs of the same pair of vertices. For example, $\rho_1(\{u, v\}) = (u, v)$ for ρ_1 in Fig. 6.4. The orientation of a temporal triangle simply determines the direction of its temporal edges. Each such temporal edge can take two possible directions, so there are eight types of orientation such a temporal triangle can take as shown in Fig. 6.4. Note that orientation of a temporal triangle is independent of its temporal ordering.

It is easy to see that the temporal ordering and orientation determine the type of the temporal triangle. But different combinations of temporal orderings and orientation could result in the same type. The temporal triangle type for all possible pairs of temporal ordering and orientation are shown in Tab. 6.1.

For a temporal ordering π and for $i \in \{1,2,3\}$, we use $S_i(\pi,\rho)$ to denote the sequence of temporal edges between the pair of vertices $\pi(i)$ that have the direction $\rho(\pi(i))$, in sorted order of timestamp. When π and ρ are clear from the context, we use S_i instead of $S_i(\pi,\rho)$. We assume that we have access to S_1, S_2, and S_3 in constant time. Let σ_i denote the length of S_i. We use $S_i[\ell]$ to denote the ℓ-th edge in the sequence S_i, and $S_i[\ell : \ell']$ to denote the consecutive subsequence of S_i ranging from $S_i[\ell]$ to $S_i[\ell']$.

For a sequence S of temporal edges in increasing order of timestamp and timestamps t and t' where $t \leqslant t'$, let $\mathrm{EC}([t,t'],S)$ denote the number of edges in S with a timestamp in the time window $[t,t']$. For given $\delta_{1,3}, \delta_{1,2}$, and $\delta_{2,3}$, let $\mathrm{TTC}(\{u,v,w\},\pi,\rho)$ denote the number of $(\delta_{1,3},\delta_{1,2},\delta_{2,3})$-temporal triangles corresponding to the static triangle $\{u,v,w\}$, temporal ordering π, and orientation ρ.

Lemma 6.7.1. *For a static triangle $\{u,v,w\}$, a temporal ordering π, and an orientation ρ,*

$$\mathrm{TTC}(\{u,v,w\},\pi,\rho) = \sum_{e_2 \in S_2} \sum_{\substack{e_1 \in S_1 \\ t(e_1) \in [t(e_2)-\delta_{1,2}, t(e_2)]}}$$
$$\mathrm{EC}([t(e_2), \min(t(e_2)+\delta_{2,3}, t(e_1)+\delta_{1,3})], S_3)$$

Proof. If temporal edge $e_1 \in S_1$ is in a $(\delta_{1,3},\delta_{1,2},\delta_{2,3})$-temporal triangle with edge $e_2 \in S_2$, then $t(e_1) \in [t(e_2)-\delta_{1,2}, t(e_2)]$. Fix a pair of temporal edges (e_1,e_2)

in $S_1 \times S_2 = \{(e_1, e_2) \mid e_1 \in S_1 \wedge e_2 \in S_2\}$ where $t(e_2) \in [t(e_1), t(e_1) + \delta_{1,2}]$. A temporal edge $e_3 \in S_3$ composes a $(\delta_{1,3}, \delta_{1,2}, \delta_{2,3})$-temporal triangle with e_1 and e_2 iff $t(e_3) \in [t(e_2), \min(t(e_2) + \delta_{2,3}, t(e_1) + \delta_{1,3})]$. $\square$

For a triangle $\{u, v, w\}$ where u is the source vertex, we divide all six possible temporal orderings into three categories based on the place of $\{v, w\}$ in them. Recall that we want to avoid enumerating temporal edges on $\{v, w\}$. In π_1 and π_2, $\{v, w\}$ is assigned to the third place. $\{v, w\}$ is assigned to the second place in π_3 and π_4, and finally to the first place in temporal ordering π_5 and π_6.

Temporal orderings π_1 and π_2: Using Lemma 6.7.1, one can compute $\mathrm{TTC}(\{u, v, w\}, \pi, \rho)$ by enumerating pairs of temporal edges in $S_1 \times S_2 = \{(e_1, e_2) \mid e_1 \in S_1 \wedge e_2 \in S_2\}$. For each pair we compute $\mathrm{EC}([t(e_2), \min(t(e_2) + \delta_{2,3}, t(e_1) + \delta_{1,3}), S_3)$ using binary search. To get the final counts we sum $\mathrm{EC}([t(e_2), \min(t(e_2) + \delta_{2,3}, t(e_1) + \delta_{1,3}), S_3)$ over all pairs $(e_1, e_2) \in S_1 \times S_2$. But enumerating $S_1 \times S_2$ could be expensive and this process overall runs in time $O(\sigma_1 \sigma_2 \log(\sigma_3))$. Next, we show that we can compute the same count by enumerating edges in S_1 and S_2 separately and storing cumulative counts of compatible edges on S_3 for each edge.

For $i, j \in \{1, 2, 3\}$ where $i \neq j$, and $\ell, \ell' \in \{1, \ldots, \sigma_i\}$ where $\ell \leqslant \ell'$ we use $\mathrm{CEC}_{+\delta_{1,3}}(S_i[\ell : \ell'], S_j)$ to denote the cumulative count of edges in S_j with a timestamp in $[t(e), t(e) + \delta_{1,3}]$ for edges e in the sequence $S_i[\ell : \ell']$. Formally

$$\mathrm{CEC}_{+\delta_{1,3}}(S_i[\ell : \ell'], S_j) = \sum_{\ell \leqslant r \leqslant \ell'} \mathrm{EC}([t(S_i[r]), t(S_i[r]) + \delta_{1,3}], S_j).$$

Cumulative counts CEC_{∞}, $\mathrm{CEC}_{-\delta_{1,3}}$, and $\mathrm{CEC}_{-\infty}$, are defined the same way with time intervals $[t(e), \infty)$, $[t(e) - \delta_{1,3}, t(e)]$, and $(-\infty, t(e)]$, respectively. Counts $\mathrm{CEC}_{-\delta_{1,2}}$, $\mathrm{CEC}_{+\delta_{1,2}}$, $\mathrm{CEC}_{-\delta_{2,3}}$, and $\mathrm{CEC}_{+\delta_{2,3}}$ are defined similarly. Note that we can compute $\mathrm{CEC}(S_i[1 : \ell], S_j)$ for each $\ell \in \{1, \ldots, \sigma_i\}$ with one pass over S_i, and once we have these counts, we can get the cumulative counts $\mathrm{CEC}(S_i[\ell' : \ell''], S_j)$, for each consecutive subsequence $S_i[\ell' : \ell'']$ of S_i as follows.

$$\begin{aligned}\mathrm{CEC}_{+\delta_{1,3}}(S_i[\ell' : \ell''], S_j) = \mathrm{CEC}_{+\delta_{1,3}}(S_i[1 : \ell''], S_j)\\ - \mathrm{CEC}_{+\delta_{1,3}}(S_i[1 : \ell' - 1], S_j).\end{aligned}$$

where $\mathrm{CEC}_{+\delta_{1,3}}(S_i[1 : 0], S_j) = 0$.

We first enumerate edges in S_1. For each edge $e_1 \in S_1$ we compute the counts $\mathrm{CEC}_{+\delta_{1,3}}(S_1[1 : \ell], S_3)$ and $\mathrm{CEC}_{\infty}(S_1[1 : \ell], S_3)$ for each $\ell \in \{1, \ldots, \sigma_1\}$ and store them for e_1. Next, we enumerate edges in S_2 and compute $\mathrm{CEC}_{\infty}(S_2[1 : \ell], S_3)$ for each $\ell \in \{1, \ldots, \sigma_2\}$.

Fix an edge $e_2 \in S_2$. Let ℓ_f and ℓ_ℓ be the indices of the first and last edges in S_1 with a timestamp in $[t(e_2) - \delta_{1,2}, t(e_2)]$. Also let $\ell_{\delta_{2,3}}$ be the index of the last edge in S_1 with a timestamp at most $t(e_2) - \delta_{1,3} + \delta_{2,3}$. We can find $\ell_f, \ell_{\delta_{2,3}}$, and ℓ_ℓ using a binary search on S_1. Note that $\delta_{1,3} \leqslant \delta_{1,2} + \delta_{2,3}$, thus $\ell_f \leqslant \ell_{\delta_{2,3}} \leqslant \ell_\ell$.

First consider the temporal edges $S_1[i]$ where $\ell_f \leqslant i \leqslant \ell_{\delta_{2,3}}$. For any such edge $t(S_1[i]) + \delta_{1,3} \leqslant t(e_2) + \delta_{2,3}$, so the timestamp of compatible edges in S_3 lie in the interval $[t(e_2), t(S_1[i])+\delta_{1,3}]$. Having stored the cumulative counts described above, we can compute the number of pairs of temporal edges $(e_1, e_3) \in S_1[\ell_f : \ell_{\delta_{2,3}}] \times S_3$ that compose a $(\delta_{1,3}, \delta_{1,2}, \delta_{2,3})$-temporal triangle on $\{u, v, w\}$ with e_2, complying with π_1 and ρ, as follows.

$$\begin{aligned}&\sum_{\ell \leqslant i \leqslant \ell_{\delta_{2,3}}} \mathrm{EC}([t(e_2), t(S_1(i)) + \delta_{1,3}], S_3) =\\ &\mathrm{CEC}_{+\delta_{1,3}}(S_1[\ell_f : \ell_{\delta_{2,3}}], S_3) - \mathrm{CEC}_{\infty}(S_1[\ell_f : \ell_{\delta_{2,3}}], S_3)\\ &+(\ell_{\delta_{2,3}} - \ell_f + 1) \cdot \mathrm{EC}([t(e_2), \infty), S_3)\end{aligned}$$

Now, we count the number of temporal edges in S_3 that compose a triangle with

e_2 and $S_1[i]$, where $\ell_{\delta_{2,3}} < i \leqslant \ell_\ell$. For a temporal edge $S_1[i]$ where $\ell_{\delta_{2,3}} < i \leqslant \ell_\ell$, we have $t(S_1[i]) + \delta_{1,3} > t(e_2) + \delta_{2,3}$. Thus, there are $\mathrm{EC}([t(e_2), t(e_2) + \delta_{2,3}], S_3)$ edges on S_3 that compose a triangle with e_2 and $S_1[i]$. So the final count of pairs $(e_1, e_3) \in S_1 \times S_3$ that are in a temporal triangle with e_2 corresponding to the static triangle $\{u, v, w\}$ can be computed as follows.

$$\begin{aligned}
&\sum_{\substack{e_1 \in S_1, t(e_1) \in \\ [t(e_2) - \delta_{1,2}, t(e_2)]}} \mathrm{EC}([t(e_2), \min(t(e_2) + \delta_{2,3}, t(e_1) + \delta_{1,3})], S_3) \\
= &\sum_{\ell_f \leqslant i \leqslant \ell_{\delta_{2,3}}} \mathrm{EC}([t(e_2), t(S_1(i)) + \delta_{1,3}], S_3) \\
&+ (\ell_\ell - \ell_{\delta_{2,3}}) \cdot \mathrm{EC}([t(e_2), t(e_2) + \delta_{2,3}], S_3)
\end{aligned}$$

Algorithm 3 Counting $(\delta_{1,3}, \delta_{1,2}, \delta_{2,3})$-temporal triangles corresponding to a static triangle and temporal orientation π_1 or π_2

```
 1: procedure TTC-vw3($\delta_{1,3}$, $\delta_{1,2}$, $\delta_{2,3}$,$\langle u, v, w\rangle$, $\pi$, $\rho$)
    ▷ $\pi(\{v, w\}) = 3$
 2:     Enumerate $S_1$ and compute $\mathrm{CEC}_{+\delta_{1,3}}$ and $\mathrm{CEC}_\infty$ on $S_3$.
 3:     count = 0
 4:     for $i = 1, \ldots, \sigma_2$ do
 5:         Let $\ell_f$ = lowerBound($t(S_2[i]) - \delta_{1,2}, S_1$)
 6:         Let $\ell_{\delta_{2,3}}$ = upperBound($t(S_2[i]) - \delta_{1,3} + \delta_{2,3}, S_1$)
 7:         Let $\ell_\ell$ = upperBound($t(S_2[i]), S_1$)
    ▷ Edges in $S_1[\ell_f : \ell_{\delta_{2,3}}]$
 8:         count + = $\mathrm{CEC}_{+\delta_{1,3}}(S_1[\ell_f : \ell_{\delta_{2,3}}], S_3)$
 9:         count − = $\mathrm{CEC}_\infty(S_1[\ell_f : \ell_{\delta_{2,3}}], S_3)$
10:         count + = $(\ell_{\delta_{2,3}} - \ell_f + 1) \cdot \mathrm{EC}([t(S_2[i]), \infty), S_3)$
    ▷ Edges in $S_1[\ell_{\delta_{2,3}} + 1 : \ell_\ell]$
11:         count + = $(\ell_\ell - \ell_{\delta_{2,3}}) \cdot \mathrm{EC}([t(S_2[i]), t(S_2[i]) + \delta_{2,3}], S_3)$
12:     return count
```

By Lemma 6.7.1, to get $\text{TTC}(\{u, v, w\}, \pi, \rho)$, we only need to sum these counts over edges in S_2. Let $\langle u, v, w\rangle$ denote a static triangle where $u \prec v \prec w$. Alg. 3 formalizes the procedure described above for computing $\text{TTC}(\langle u, v, w\rangle, \pi, \rho)$ where π is either π_1 or π_2.

Similar to Alg. 3, in algorithms for the remaining temporal orderings, for each static triangle $\{u, v, w\}$ where u is the source vertex, we only enumerate temporal edges on $\{u, v\}$ and $\{u, w\}$. And for each temporal edge we perform a constant number of binary searches on temporal edges on the other two static edges of the static triangle $\{u, v, w\}$.

Temporal orderings π_3 and π_4: Consider an orientation ρ, and a static triangle on vertices $\{u, v, w\}$ enumerated in $G_\prec$, where u is the source vertex. The category of temporal orderings π_3 and π_4 is more intricate because $\pi_3(\{v, w\}) = \pi_4(\{v, w\}) = 2$. Recall that we want to avoid enumerating temporal edges on $\{v, w\}$, so we do not enumerate edges on S_2 as in the case of π_1 and π_2. Instead, we enumerate edges on S_1, and compute the counts of edges on S_2 that form a temporal triangle with compatible edge in S_3.

We start by enumerating edges on S_1. Consider an edge $e_1 \in S_1$. Let ℓ_f and ℓ_ℓ denote the indices of first and last edge in S_3 with a timestamp in $[t(e_1), t(e_1)+\delta_{1,3}]$. Also, let $\ell_{\delta_{1,2}}$ denote the index of the first edge in S_3 with a timestamp greater than $t(e_1)+\delta_{1,2}$, and $\ell_{\delta_{2,3}}$ be the index of the first edge in S_3 that has a timestamp greater than $t(e_1) + \delta_{2,3}$. Note that $\ell_{\delta_{1,2}}$ and $\ell_{\delta_{2,3}}$ divide $S_3[\ell : \ell_\ell]$ into three consecutive subsequences. We show how to count temporal triangles that involve temporal edges in each of these three subsequences.

For a temporal edge $S_3[i]$ where $\ell_f \leqslant i < \min(\ell_{\delta_{1,2}}, \ell_{\delta_{2,3}})$, each edge $e_2 \in S_2$ where $t(e_2) \in [t(e_1), t(S_3[i])]$ form a $(\delta_{1,3}, \delta_{1,2}, \delta_{2,3})$-temporal triangle with e_1 and $S_3[i]$. To obtain the counts of these edges in S_2, it suffices to store $\text{CEC}_{-\infty}$ on S_2

for each edge $e_3 \in S_3$.

Now, consider the temporal edges $S_3[i]$ where $\max(\ell_{\delta_{1,2}}, \ell_{\delta_{2,3}}) < i \leqslant \ell_\ell$. The timestamp of these edges are in time window $[t(e_1) + \max(\delta_{1,2}, \delta_{2,3}), t(e_1) + \delta_{1,3}]$, and together with temporal edge e_1 form a $(\delta_{1,3}, \delta_{1,2}, \delta_{2,3})$-temporal triangle with each temporal edge $e_2 \in S_2$ where $t(e_2) \in [t(S_3[i]) - \delta_{2,3}, t(e_1) + \delta_{1,2}])$. To count the number of such temporal edges in S_2 we only need to store $\mathrm{CEC}_{-\delta_{2,3}}$ and $\mathrm{CEC}_{-\infty}$ on S_2 for each temporal edge e_3 in S_3.

Algorithm 4 Counting $(\delta_{1,3}, \delta_{1,2}, \delta_{2,3})$-temporal triangles corresponding to a static triangle and temporal orientation π_3 or π_4

```
 1: procedure CTT-VW2($\delta_{1,3}$, $\delta_{1,2}$, $\delta_{2,3}$,$\langle u, v, w\rangle$, $\pi$, $\rho$)
    ▷ $\pi(\{v, w\}) = 2$
 2:     count = 0
 3:     Enumerate $S_3$ and compute $\mathrm{CEC}_{-\infty}$ and $\mathrm{CEC}_{-\delta_{2,3}}$ on $S_2$
 4:     for $i = 1, \ldots, \sigma_1$ do
 5:         Let $\ell_f = \textsc{lowerBound}(t(S_1[i]), S_3)$
 6:         Let $\ell_{\delta_{1,2}} = \textsc{upperBound}(t(S_1[i]) + \delta_{1,2}, S_3)$
 7:         Let $\ell_{\delta_{2,3}} = \textsc{lowerBound}(t(S_1[i]) + \delta_{2,3}, S_3)$
 8:         Let $\ell_\ell = \textsc{upperBound}(t(S_1[i]) + \delta_{1,3}, S_3)$
 9:         Let $\ell_{\min} = \min(\ell_{\delta_{1,2}}, \ell_{\delta_{2,3}})$
10:         Let $\ell_{\max} = \max(\ell_{\delta_{1,2}}, \ell_{\delta_{2,3}})$
    ▷ Edges in $S_3[\ell_f : \ell_{\min}]$
11:         count + = $\mathrm{CEC}_{-\infty}(S_3[\ell_f : \ell_{\min}], S_2)$
12:         count − = $(\ell_{\min} - \ell_f + 1) \cdot \mathrm{EC}((-\infty, t(S_1[i])], S_2)$
    ▷ Edges in $S_3[\ell_{\min} + 1 : \ell_{\max} - 1]$
13:         if $\delta_{1,2} \leqslant \delta_{2,3}$ then
14:             count + = $(\ell_{\delta_{2,3}} - \ell_{\delta_{1,2}})$
15:                 $\cdot\, \mathrm{EC}([t(S_1[i]), t(S_1[i]) + \delta_{1,2}], S_2)$
16:         else if $\delta_{2,3} \leqslant \delta_{1,2}$ then
17:             count + = $\mathrm{CEC}_{-\delta_{2,3}}(S_3[\ell_{\delta_{2,3}} : \ell_{\delta_{1,2}}], S_2)$
    ▷ Edges in $S_3[\ell_{\max} : \ell_\ell]$
18:         count + = $\mathrm{CEC}_{-\delta_{2,3}}(S_3[\ell_{\max} : \ell_\ell], S_2)$
19:         count − = $\mathrm{CEC}_{-\infty}(S_3[\ell_{\max} : \ell_\ell], S_2)$
20:         count += $(\ell_\ell - \ell_{\max}) \cdot \mathrm{EC}((-\infty, t(S_1[i]) + \delta_{1,2}], S_2)$
21:     return count
```

Finally, consider an edge $S_3[i]$ where $\min(\ell_{\delta_{1,2}}, \ell_{\delta_{2,3}}) \leqslant i \leqslant \max(\ell_{\delta_1}, \ell_{\delta_{2,3}})$. The number of compatible edges in S_2 depend on how $\delta_{1,2}$ compares to $\delta_{2,3}$. There are two cases: $(a) : \delta_{1,2} < \delta_{2,3}$ and $(b) : \delta_{2,3} < \delta_{1,2}$. In case (a) edges in S_2 have to have a timestamp in $[t(e_1), t(e_1)+\delta_{1,2}]$ to form a triangle with e_1 and $S_3[i]$, and in case (b) their timestamps should be in the time window $[t(S_3[i]) - \delta_{2,3}, t(S_3[i])]$. Alg. 4 give the step by step procedure for counting temporal triangles for temporal orderings π_3 and π_4.

Algorithm 5 Counting $(\delta_{1,3}, \delta_{1,2}, \delta_{2,3})$-temporal triangles corresponding to a static triangle and temporal orientation π_5 or π_6

```
1: procedure TTC-vw1($\delta_{1,3}$, $\delta_{1,2}$, $\delta_{2,3}$,$\langle u, v, w\rangle$, $\pi$, $\rho$)
   ▷ $\pi(\{v, w\}) = 1$
2:     count = 0
3:     Enumerate $S_3$ and compute $\mathrm{CEC}_{-\delta_{1,3}}$ and $\mathrm{CEC}_{-\infty}$ on $S_1$
4:     for $i = 1, \ldots, \sigma_2$ do
5:         Let $\ell_f$ = lowerBound$(t(S_2[i]), S_3)$
6:         Let $\ell_{\delta_{1,2}}$ = lowerBound$(t(S_2[i]) + \delta_{1,3} - \delta_{1,2}, S_3)$
7:         Let $\ell_\ell$ = upperBound$(t(S_2[i]) + \delta_{2,3}, S_3)$
   ▷ Edges in $S_3[\ell_{\delta_{1,2}} : \ell_\ell]$
8:         count + = $\mathrm{CEC}_{-\delta_{1,3}}(S_3[\ell_{\delta_{1,2}} : \ell_\ell], S_1)$
9:         count − = $\mathrm{CEC}_{-\infty}(S_3[\ell_{\delta_{1,2}} : \ell_\ell], S_1)$
10:        count + = $(\ell_\ell - \ell_{\delta_{1,2}} + 1) \cdot \mathrm{EC}((-\infty, t(S_2[i])], S_1)$
   ▷ Edges in $S_3[\ell_f : \ell_{\delta_{1,2}} - 1]$
11:        count + = $(\ell_{\delta_{1,2}} - \ell_f) \cdot \mathrm{EC}([t(S_2[i]) - \delta_{1,2}, t(S_2[i])], S_1)$
12:    return count
```

Temporal orderings π_5 and π_6: This case is similar to the case of temporal ordering π_1 and π_2. The difference is that while enumerating edges in S_2, we will first find the compatible edges in S_3 instead of S_1, and then count edges in S_1 that complete a temporal triangle. Consider a static triangle $\{u, v, w\}$ and orientation ρ. Fix a temporal edge e_2 in S_2. Let ℓ_f and ℓ_ℓ denote the indices of the first and last temporal edges in S_3 with a timestamp in the time period $[t(e_2), t(e_2) + \delta_{2,3}]$. Let $\ell_{\delta_{1,2}}$ be the first temporal edge in S_3 such that $t(S_3(\ell_{\delta_{1,2}})) > t(e_2)+\delta_{1,3}-\delta_{1,2}$. Since $\delta_{1,3} \leqslant \delta_{1,2} + \delta_{2,3}$, we have $\ell_f \leqslant \ell_{\delta_{2,3}} \leqslant \ell_\ell$. Consider a temporal edge $S_3[i]$ where $\ell_f \leqslant i < \ell_{\delta_{1,2}}$. The number of edges in S_1 that form a triangle with e_2 and $S_3[i]$

is $\mathrm{EC}(S_1, [t(e_2) - \delta_{1,2}, t(e_2)])$. For each temporal edge $S_3[i]$ where $\ell_{\delta_{1,2}} \leqslant i \leqslant \ell_\ell$, there are $\mathrm{EC}(S_1, [t(S_3[i]) - \delta_{1,3}, t(e_2)])$ edges that complete a temporal triangle. This is the same as in the case of π_1 and π_2 with different time windows. We need to get $\mathrm{CEC}_{-\infty}$ and $\mathrm{CEC}_{-\delta_{1,3}}$ on S_1 for each edge $e_3 \in S_3$.

6.7.1 Getting the Counts for All Temporal Triangle Types

Now that we can count $(\delta_{1,3}, \delta_{1,2}, \delta_{2,3})$-temporal triangles for each combination of temporal ordering and orientation, it only remains to get the counts for each temporal triangle type (Fig. 6.1). Let $\psi(\pi, \rho)$ denote the triangle type for π and ρ. Alg. 6 gets the counts for all eight types. Now, we can finally prove Theorem 6.2.1.

Algorithm 6 Counting $(\delta_{1,3}, \delta_{1,2}, \delta_{2,3})$-temporal triangles for each temporal triangle types

```
1: procedure COUNT-TEMPORAL-TRIANGLES($T$, $\delta_{1,3}$, $\delta_{1,2}$, $\delta_{2,3}$)
2:     Extract the static graph $G$ of $T$.
3:     Find the degeneracy ordering $\prec$ of $G$.
4:     Derive $G_\prec$ by orienting $G$ with respect to $\prec$.
5:     Initialize Counts to 0 for $\mathcal{T}_1, \ldots, \mathcal{T}_8$.
6:     for all Static triangles $\{u, v, w\}$ do
                                                        ▷ WLOG let $u \prec v \prec w$
7:         for all Temporal ordering $\pi$ and orientation $\rho$ do
8:             Counts($\psi(\pi, \rho)$) += TTC($\langle u, v, w\rangle, \pi, \rho$)     ▷ Tab. 6.1
                                              ▷ Using Alg. 3, Alg. 4, and Alg. 5
```

Proof of Theorem 6.2.1. Extracting the static graph G from T can be done in $O(m)$ time. We simply enumerate all temporal edges of T and for each temporal edge $e = (v_1, v_2, t)$, we add an static edge between $\{v_1, v_2\}$ in G if they are not connected already. The degeneracy ordering of G could be obtained in $O(m_s)$ time [112], and $G_\prec$ could also be derived in time $O(m_s)$. For enumerating static triangles we first enumerate each edge in $G_\prec$. For each edge (u, v), we enumerate

$N^+(u)$ which takes $O(\kappa)$ as $d^+_{G_\prec}(u) \leqslant \kappa$. We can lookup if there is an edge between v and w in constant time. Thus, enumerating triangles take $O(m_s \cdot \kappa)$ time overall.

Note that for each static triangle we only enumerate temporal edges on static edges incident to the source vertex. So for the static triangle $\langle u, v, w\rangle$, we only enumerate temporal edges on the pairs $\{u, v\}$ and $\{u, w\}$. While processing a temporal edge during enumeration of temporal edges on $\{u, v\}$ or $\{u, w\}$, we either perform a constant time operation, or spend $O(\log(\sigma_{\max}))$ time for a constant number of binary searches over the temporal edges of the other two static edges in the static triangle $\langle u, v, w\rangle$. Thus,

$$T(\mathcal{A}) = O(m_s \cdot \kappa + \sum_{\langle u,v,w\rangle} (\sigma(u, v) + \sigma(u, w)) \log(\sigma_{\max}))$$

where $\mathcal{A}$ denotes Alg. 6, and $T(\mathcal{A})$ denotes the worst case time complexity of $\mathcal{A}$.

For each vertex $u \in V$, $d^+_{G_\prec}(u) \leqslant \kappa$, so each edge (u, v) in $G_\prec$, is a part of at most κ static triangles where u is the source vertex. Therefore, the temporal edges on each edge $\{u, v\}$ in G are enumerated at most $O(\kappa)$ times. Thus,

$$T(\mathcal{A}) = O(m_s \cdot \kappa + \sum_{\{u,v\}\in E_s} (\sigma(u, v) + \sigma(v, u)) \cdot \kappa \log(\sigma_{\max})).$$

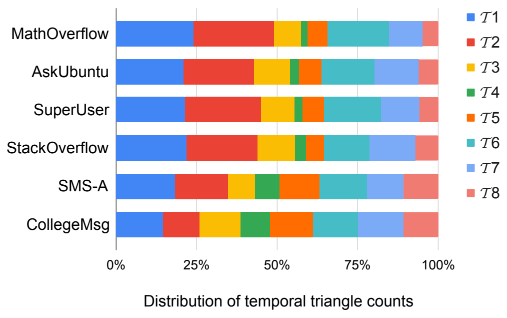
Hence,

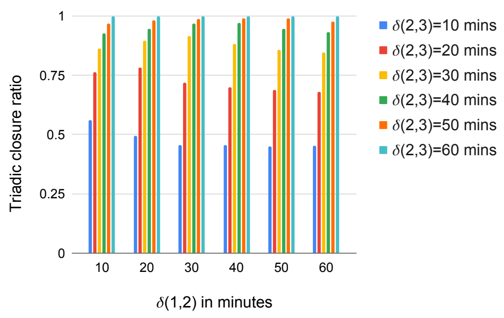
$$T(\mathcal{A}) = O(m\kappa \log(\sigma_{\max})).$$

□

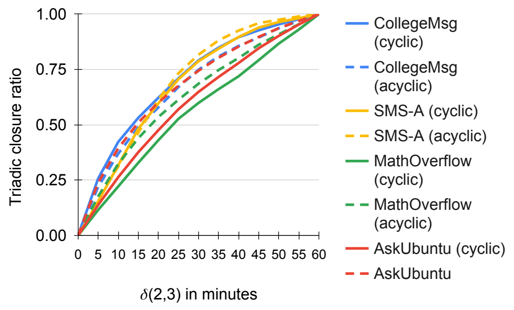
Table 6.2: Descriptions of the datasets and runtime of `DOTTT` and PBL .

dataset	#vertices	#edges	#static edges	#static triangles	degeneracy	max multiplicity	time span (years)	`DOTTT` runtime	PBL runtime
CollegeMsg	1.9K	59.8K	13.8K	14.3K	20	98	0.51	0.09	0.07
email-Eu-core	986	332K	16.1K	105K	34	2.8K	2.2	2.31	3.37
MathOverflow	24.7K	390K	188K	1.4M	78	225	6.46	3.17	3.6
SMS-A	44.1K	545K	52.22K	10K	9	5.3K	0.92	0.45	0.81
AskUbuntu	157K	727K	456K	680K	48	154	7.09	2.23	5.08
SuperUser	192K	1.11M	715K	1.54M	61	78	7.59	4.41	8.84
WikiTalk	1.09M	6.11M	2.79M	8.12M	124	1.1K	6.21	34	56
StackOverflow	2.58M	47.9M	28.18M	114.2M	198	549	7.60	347	678
Wikipedia-DE	2.17M	86.21M	39.71M	169.9M	265	347	10.18	576	987
Bitcoin	59.61M	515.5M	366.4M	706.2M	604	447K	5.98	2923	4374





(a) Temporal triangle count distribution **(b)** Effect of $\delta_{1,2}$ and $\delta_{2,3}$ on triadic closure



(c) Triadic closure in cyclic and acyclic cases

Figure 6.5: (a):The distribution of (1 hr, 1 hr, 1 hr)-temporal triangle counts over all eight temporal triangle types as shown in Fig. 6.1. (b):We fix $\delta_{1,3}$ to 2 hrs. We vary $\delta_{1,2}$ from 0 to 60 minutes and plot the ratio of (2 hrs, $\delta_{1,2}, \delta_{2,3}$)-temporal triangles to (2hrs, $\delta_{1,2}$, 1hr)-temporal triangles for $\delta_{2,3}$ ranging from 0 to 60 minutes. (c) We plot the ratio of (2 hrs, 1 hr, $\delta_{2,3}$)-temporal triangles to (2hrs, 1hr, 1hr)-temporal triangles for $\delta_{2,3}$ ranging from 0 to 60 minutes, for cyclic and acyclic triangles.

6.8 Experimental Evaluations

We implemented our algorithm in C++ and used a commodity machine from AWS EC2: R5d.2xlarge to run our experiments. This EC2 instance has Intel(R) Xeon(R) Platinum 8175M CPU @ 2.50GHz and 64GB memory. On this AWS machine, PBL runs out of memory for the Bitcoin graph, so we used one with more than 256GB memory for this case. The implementation of DOTTT is available at [5].

We performed our experiments on a collection of temporal input graphs from SNAP [101], KONECT [98], and the Bitcoin transaction dataset from [93], consisting of all transactions up to Feb 9, 2018. The timestamp of each transaction is the creation time of the block on the blockchain that contains it [143].

Running time: All the running times are shown in Tab. 6.2. We ran all experiments on a single thread. In most instances, DOTTT takes a few seconds to run. For graphs with tens of millions of temporal edges, DOTTT runs in less than ten minutes. Even for the Bitcoin graph with 515M edges, DOTTT takes less than an hour.

Running time independent of time periods: The running time of both DOTTT and PBL algorithms are independent of the time periods. DOTTT has the same running time for time restrictions ranging from 0 to the time span of the input dataset. For comparison with PBL, we set $\delta_{1,3} = \delta_{1,2} = \delta_{2,3} = 1$ hr.

Comparison with PBL : We compare our algorithm with the PBL algorithm that counts $\delta_{1,3}$-temporal triangles, as it is the closest to our work. We typically get a 1.5x-2x speedup over PBL for large graphs (more than 0.5M edges) as shown in Fig. 6.2a. Note that DOTTT computes $(\delta_{1,3}, \delta_{1,2}, \delta_{2,3})$-temporal triangle counts while PBL only gets the counts of $\delta_{1,3}$-temporal triangles.

Distribution of counts over types of triangles: The distribution of (1 hr, 1 hr, 1 hr)-temporal triangle counts for our datasets are shown in Fig. 6.5a. As we expected [119, 128, 183, 195], networks from similar domains have similar distributions. It is easy to see in Fig. 6.5a, that all the stack exchange networks have similar distributions. The same holds for the message networks CollegeMsg and SMS-A.

We observe that cyclic temporal triangles, $\mathcal{T}_4$ and $\mathcal{T}_8$, have a larger share in temporal triangle counts in messaging networks than in stack exchange networks.

Triadic closures in temporal networks: In static triangles, the transitivity measures the ratio of number of static triangles to the number of all wedges. In temporal graphs, in addition to transitivity, the time it takes for a wedge to appear and close is of importance [200]. In Fig. 6.5b, we study the effect of the time it takes for a wedge to appear from an edge, on the time it takes to close for CollegeMsg graph. We fix $\delta_{1,3} = 2$ hrs. For $\delta_{1,2}$ ranging from zero to 60 minutes (10 minute steps), we vary $\delta_{2,3}$ from zero to 60 minutes and plot the ratio of $(\delta_{1,3}, \delta_{1,2}, \delta_{2,3})$-temporal triangles over (2hrs, $\delta_{1,2}$, 1hr)-temporal triangles. We observe that the set of ratios for all values of $\delta_{2,3}$ are almost identical for different values of $\delta_{1,2}$. For instance, for all values of $\delta_{1,2}$, roughly half the triangles are formed in 10-20 minutes. This implies that once a wedge is formed, the time it took to appear does not affect the time it takes to close.

As another demonstration of `DOTTT`, for $\delta_{1,3} = 2$ hrs and $\delta_{1,2} = 1$ hrs, we plot the ratio of $(\delta_{1,3}, \delta_{1,2}, \delta_{2,3})$-temporal triangles to (2hrs, 1hr, 1hr)-temporal triangles, this time separately for cyclic and acyclic temporal triangles in Fig. 6.5c. We observe that for stack exchange networks, acyclic temporal triangles tend to take a shorter time to close from the moment their second edge appears than cyclic temporal triangles. As we see in Fig. 6.5c, this is not the case for message networks.

Chapter 7

Conclusion

In this thesis we studied the problem of subgraph counting and the role of graph orientation and degeneracy, both in theory and in practice. We gained a better theoretical understanding of the problem by giving a linear time algorithm for $\textsc{sub-cnt}_k$ in bounded degenreacy graphs for $k < 6$, and proving that it does not admit a linear time algorithm, assuming a standard conjecture in fine grained complexity. Moreover, we discovered a near-linear time algorithm dichotomy for homomorphism counting in bounded degenreacy grpahs. We gave a clean characterization of patterns for which near linear time homomorphism counting algorithms are possible in bounded degenreacy graphs.

Our results on subgraph and homomorphism counting in bounded degeneracy graphs advanced our theoretical understanding of this problem and the limits of degenracy based methods. We pose the following open problems as possible future research direction.

- In Chapter 4, we gave a clean characterization of pattern graphs H for which $\text{Hom}_G(H)$ is computable in near-linear time when G has bounded degeneracy. Our lower bound result (Theorem Theorem 4.5.1) also holds for SUB-CNT$_H$ (see Observation Observation 4.5.8). However, our argument for the other direction does not follow to the subgraph counting version of the problem. Can we characterize pattern graphs H where SUB-CNT$_H$ admits a (near) linear time algorithm in bounded degeneracy graphs?

- Can we describe the restricted class of graphs $\mathcal{G}$, for each pattern H, where $\mathrm{Hom}_G(H)$ for $G \in \mathcal{G}$ is countable in (near) linear time?

We also contributed to the practical world of subgraph counting. We gave practical algorithm for getting all 5-VOCs, based on graph orientation and vertex ordering, that is typically hundreds of times faster than the state-of-the-art. This is an exemplary result, showing the power of these techniques.

Another practical subgraph counting subfield we explored is counting patterns in temporal networks. We introduced $(\delta_{1,3}, \delta_{1,2}, \delta_{2,3})$-temporal triangles, a generalized notion of temporal triangles that considers time gaps between any pair of edges of the triangle. We presented `DOTTT`, an efficient algorithm based on graph orientation and degeneracy ordering, for counting $(\delta_{1,3}, \delta_{1,2}, \delta_{2,3})$-temporal triangles. `DOTTT` improves on the state-of-the-art temporal triangle counting algorithms and has an asymptotic running time closer to that of static triangle counting. It would be interesting to study the possiblity of extension of the notion of $(\delta_{1,3}, \delta_{1,2}, \delta_{2,3})$-temporal triangles to other temporal patterns.

Bibliography

[1] Citation network dataset. Available at https://aminer.org/citation.

[2] Evoke. https://bitbucket.org/nojan-p/orbit-counting.

[3] The koblenz network collection. Available at http://konect.uni-koblenz.de/.

[4] Wikipedia (en) network dataset – KONECT, October 2016.

[5] DOTTT. https://github.com/nojanp/temporal-triangle-counting, 2021.

[6] Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *Proc. 55th Annual IEEE Symposium on Foundations of Computer Science*, 2014.

[7] Monica Agrawal, Marinka Zitnik, and Jure Leskovec. Large-scale analysis of disease pathways in the human interactome. In *Pacific Symposium on Biocomputing*, volume 23, page 111. World Scientific, 2018.

[8] Nesreen K. Ahmed, Jennifer Neville, Ryan A. Rossi, and Nick Duffield. Efficient graphlet counting for large networks. In *International Conference on Data Mining*, 2015.

[9] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Graph sketches: sparsification, spanners, and subgraphs. In *Proc. 31st ACM Symposium on Principles of Database Systems*, pages 5–14. ACM, 2012.

[10] 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.

[11] Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *J. ACM*, 42(4):844–856, 1995.

[12] Noga Alon, Raphael Yuster, and Uri Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997.

[13] Sepehr Assadi, Michael Kapralov, and Sanjeev Khanna. A simple sublinear-time algorithm for counting arbitrary subgraphs via edge sampling. In *Proc. 10th Conference on Innovations in Theoretical Computer Science*, 2018.

[14] Ziv Bar-Yossef, Ravi Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2002.

[15] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *Annual SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 16–24, 2008.

[16] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 16–24, 2008.

[17] A. Benson, D. F. Gleich, and J. Leskovec. Higher-order organization of complex networks. *Science*, 353(6295):163–166, 2016.

[18] Suman K Bera and Amit Chakrabarti. Towards tighter space bounds for counting triangles and other substructures in graph streams. In *Proc. 34th International Symposium on Theoretical Aspects of Computer Science*, 2017.

[19] Suman K Bera, Amit Chakrabarti, and Prantar Ghosh. Graph coloring via degeneracy in streaming and other space-conscious models. In *Proc. 47th International Colloquium on Automata, Languages and Programming*, 2020.

[20] Suman K Bera, Noujan Pashanasangi, and C Seshadhri. Linear time subgraph counting, graph degeneracy, and the chasm at size six. In *Proc. 11th Conference on Innovations in Theoretical Computer Science*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[21] Suman K Bera, Noujan Pashanasangi, and C Seshadhri. Near-linear time homomorphism counting in bounded degeneracy graphs: the barrier of long induced cycles. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2315–2332. SIAM, 2021.

[22] Suman K Bera and C Seshadhri. How the degeneracy helps for triangle counting in graph streams. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 457–467, 2020.

[23] Jonathan W. Berry, Bruce Hendrickson, Randall A. LaViolette, and Cynthia A. Phillips. Tolerating the community detection resolution limit with edge weighting. *Phys. Rev. E*, 83:056119, May 2011.

[24] Umberto Bertele and Francesco Brioschi. On non-serial dynamic programming. *J. Comb. Theory, Ser. A*, 14(2):137–148, 1973.

[25] Nadja Betzler, Rene Van Bevern, Michael R Fellows, Christian Komusiewicz, and Rolf Niedermeier. Parameterized algorithmics for finding connected motifs in biological networks. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 8(5):1296–1308, 2011.

[26] M. Bhuiyan, M. Rahman, M. Rahman, and M. Al Hasan. Guise: Uniform sampling of graphlets for large graph analysis. In *International Conference on Data Mining*, pages 91–100, 2012.

[27] Etienne Birmele et al. Detecting local network motifs. *Electronic Journal of Statistics*, 6:908–933, 2012.

[28] Andreas Björklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto. Counting paths and packings in halves. In *Proc. 17th Annual European Symposium on Algorithms*, pages 578–586, 2009.

[29] Andreas Björklund, Petteri Kaski, and Łukasz Kowalik. Counting thin subgraphs via packings faster than meet-in-the-middle time. *ACM Transactions on Algorithms (TALG)*, 13(4):48, 2017.

[30] Hanjo D Boekhout, Walter A Kosters, and Frank W Takes. Efficiently counting complex multilayer temporal motifs in large-scale networks. *Computational Social Networks*, 6(1):1–34, 2019.

[31] JA Bondy and USR Murty. Graph theory (2008). *Grad. Texts in Math*, 2008.

[32] Christian Borgs, Jennifer Chayes, László Lovász, Vera T Sós, and Katalin Vesztergombi. Counting graph homomorphisms. In *Topics in discrete mathematics*, pages 315–371. Springer, 2006.

[33] Giorgos Bouritsas, Fabrizio Frasca, Stefanos Zafeiriou, and Michael M Bronstein. Improving graph neural network expressivity via subgraph isomorphism counting. *arXiv preprint arXiv:2006.09252*, 2020.

[34] Marco Bressan. Faster subgraph counting in sparse graphs. In *14th International Symposium on Parameterized and Exact Computation (IPEC 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[35] Graham R Brightwell and Peter Winkler. Graph homomorphisms and phase transitions. *Journal of combinatorial theory, series B*, 77(2):221–262, 1999.

[36] R. Burt. Structural holes and good ideas. *American Journal of Sociology*, 110(2):349–399, 2004.

[37] Ashok K Chandra and Philip M Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proc. 9th Annual ACM Symposium on the Theory of Computing*, pages 77–90, 1977.

[38] Norishige Chiba and Takao Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on computing*, 14(1):210–223, 1985.

[39] Norishige Chiba and Takao Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. Comput.*, 14:210–223, 1985.

[40] Jonathan Cohen. Graph twiddling in a mapreduce world. *Computing in Science & Engineering*, 11(4):29, 2009.

[41] J. Coleman. Social capital in the creation of human capital. *American Journal of Sociology*, 94:S95–S120, 1988.

[42] Graham Cormode and Hossein Jowhari. A second look at counting triangles in graph streams (corrected). *Theor. Comput. Sci.*, 683:22–30, 2017.

[43] Radu Curticapean, Holger Dell, and Dániel Marx. Homomorphisms are a good basis for counting small subgraphs. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 210–223, 2017.

[44] Radu Curticapean and Dániel Marx. Complexity of counting subgraphs: Only the boundedness of the vertex-cover number counts. In *Proc. 55th Annual IEEE Symposium on Foundations of Computer Science*, pages 130–139, 2014.

[45] Víctor Dalmau and Peter Jonsson. The complexity of counting homomorphisms seen from the other side. *Theor. Comput. Sci.*, 329(1-3):315–323, 2004.

[46] Holger Dell, Marc Roth, and Philip Wellnitz. Counting answers to existential questions. In *Proc. 46th International Colloquium on Automata, Languages and Programming*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[47] Josep Díaz, Maria Serna, and Dimitrios M Thilikos. Counting h-colorings of partial k-trees. *Theor. Comput. Sci.*, 281(1-2):291–309, 2002.

[48] Reinhard Diestel. *Graph Theory, Fourth Edition*. Springer, 2010.

[49] Stavros I Dimitriadis, Nikolaos A Laskaris, Vasso Tsirka, Michael Vourkas, Sifis Micheloyannis, and Spiros Fotopoulos. Tracking brain dynamics via time-dependent network analysis. *Journal of neuroscience methods*, 193(1):145–155, 2010.

[50] Martin Dyer and Catherine Greenhill. The complexity of counting graph homomorphisms. *Random Structures & Algorithms*, 17(3-4):260–289, 2000.

[51] Talya Eden, Amit Levi, Dana Ron, and C Seshadhri. Approximately counting triangles in sublinear time. *SIAM Journal on Computing*, 46(5):1603–1646, 2017.

[52] Talya Eden, Dana Ron, and C Seshadhri. On approximating the number of k-cliques in sublinear time. In *Proc. 50th Annual ACM Symposium on the Theory of Computing*, pages 722–734, 2018.

[53] Talya Eden, Dana Ron, and C Seshadhri. Faster sublinear approximations of k-cliques for low arboricity graphs. In *Annual ACM-SIAM Symposium on Discrete Algorithms*, 2020.

[54] Friedrich Eisenbrand and Fabrizio Grandoni. On the complexity of fixed parameter clique and dominating set. *Theor. Comput. Sci.*, 326(1-3):57–67, 2004.

[55] Ethan R Elenberg, Karthikeyan Shanmugam, Michael Borokhovich, and Alexandros G Dimakis. Beyond triangles: A distributed framework for estimating 3-profiles of large graphs. In *Proc. 12th Annual SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 229–238. ACM, 2015.

[56] Ethan R. Elenberg, Karthikeyan Shanmugam, Michael Borokhovich, and Alexandros G. Dimakis. Beyond triangles: A distributed framework for estimating 3-profiles of large graphs. In *Annual SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 229–238, 2015.

[57] Ethan R Elenberg, Karthikeyan Shanmugam, Michael Borokhovich, and Alexandros G Dimakis. Distributed estimation of graph 4-profiles. In *Proc. 25th Proceedings, International World Wide Web Conference (WWW)*, pages 483–493. International World Wide Web Conferences Steering Committee, 2016.

[58] Ethan R. Elenberg, Karthikeyan Shanmugam, Michael Borokhovich, and Alexandros G. Dimakis. Distributed estimation of graph 4-profiles. In *Proceedings, International World Wide Web Conference (WWW)*, pages 483–493, 2016.

[59] David Eppstein. Arboricity and bipartite subgraph listing algorithms. *Information processing letters*, 51(4):207–211, 1994.

[60] G. Fagiolo. Clustering in complex directed networks. *Phys. Rev. E*, 76:026107, Aug 2007.

[61] Mehrdad Farajtabar, Manuel Gomez-Rodriguez, Yichen Wang, Shuang Li, Hongyuan Zha, and Le Song. Coevolve: A joint point process model for information diffusion and network co-evolution. In *Companion Proceedings of the The Web Conference 2018*, 2018.

[62] K. Faust. A puzzle concerning triads in social networks: Graph constraints and the triad census. *Social Networks*, 32(3):221–233, 2010.

[63] Jörg Flum and Martin Grohe. The parameterized complexity of counting problems. *SIAM J. Comput.*, 33(4):892–922, 2004.

[64] Noé Gaumont, Clémence Magnien, and Matthieu Latapy. Finding remarkably dense sequences of contacts in link streams. *Social Network Analysis and Mining*, 6(1):1–14, 2016.

[65] Lior Gishboliner, Yevgeny Levanzov, and Asaf Shapira. Counting subgraphs in degenerate graphs, 2020.

[66] G. Goel and J. Gustedt. Bounded arboricity to determine the local structure of sparse graphs. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 159–167. Springer, 2006.

[67] Mira Gonen and Yuval Shavitt. Approximating the number of network motifs. *Internet Mathematics*, 6(3):349–372, 2009.

[68] Martin Grohe. The complexity of homomorphism and constraint satisfaction problems seen from the other side. *Journal of the ACM (JACM)*, 54(1):1–24, 2007.

[69] Shawn Gu, John Johnson, Fazle E Faisal, and Tijana Milenković. From homogeneous to heterogeneous network alignment via colored graphlets. *Scientific reports*, 8(1):1–16, 2018.

[70] Saket Gurukar, Sayan Ranu, and Balaraman Ravindran. Commit: A scalable approach to mining communication motifs from dynamic networks. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015.

[71] Rudolf Halin. S-functions for graphs. *Journal of geometry*, 8(1-2):171–186, 1976.

[72] Pavol Hell and Jaroslav Nešetřil. On the complexity of h-coloring. *Journal of Combinatorial Theory, Series B*, 48(1):92–110, 1990.

[73] Tomaž Hočevar and Janez Demšar. A combinatorial approach to graphlet counting. *Bioinformatics*, 30(4):559–565, 2014.

[74] Tomaž Hočevar and Janez Demšar. Combinatorial algorithm for counting small induced graphs and orbits. *PloS ONE*, 12(2):e0171428, 2017.

[75] Tomaž Hočevar, Janez Demšar, et al. Computation of graphlet orbits for nodes and edges in sparse graphs. *Journ. Stat. Soft*, 71, 2016.

[76] P. Holland and S. Leinhardt. A method for detecting structure in sociometric data. *American Journal of Sociology*, 76:492–513, 1970.

[77] F. Hormozdiari, P. Berenbrink, N. Prulj, and S. Cenk Sahinalp. Not all scale-free networks are born equal: The role of the seed graph in ppi network evolution. *PLoS Computational Biology*, 118, 2007.

[78] Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? In *Proc. 39th Annual IEEE Symposium on Foundations of Computer Science*, pages 653–662, 1998.

[79] Alon Itai and Michael Rodeh. Finding a minimum circuit in a graph. *SIAM Journal on Computing*, 7(4):413–423, 1978.

[80] Shweta Jain and C. Seshadhri. A Fast and Provable Method for Estimating Clique Counts Using Turán's Theorem. In *Proceedings, International World Wide Web Conference (WWW)*, pages 441–449, 2017.

[81] Shweta Jain and C Seshadhri. A fast and provable method for estimating clique counts using turán's theorem. In *Proc. 26th Proceedings, International World Wide Web Conference (WWW)*, pages 441–449. International World Wide Web Conferences Steering Committee, 2017.

[82] Madhav Jha, C Seshadhri, and Ali Pinar. A space efficient streaming algorithm for triangle counting using the birthday paradox. In *Proc. 19th Annual SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 589–597, 2013.

[83] Madhav Jha, C Seshadhri, and Ali Pinar. Path sampling: A fast and provable method for estimating 4-vertex subgraph counts. In *Proc. 24th Proceedings, International World Wide Web Conference (WWW)*, pages 495–505. International World Wide Web Conferences Steering Committee, 2015.

[84] Madhav Jha, C. Seshadhri, and Ali Pinar. Path sampling: A fast and provable method for estimating 4-vertex subgraph counts. In *Proceedings, International World Wide Web Conference (WWW)*, 2015.

[85] H. Jowhari and M. Ghodsi. New streaming algorithms for counting triangles in graphs. In *Computing and Combinatorics Conference (COCOON)*, pages 710–716, 2005.

[86] Hossein Jowhari and Mohammad Ghodsi. New streaming algorithms for counting triangles in graphs. In *Computing and Combinatorics*, pages 710–716, 2005.

[87] Daniel M Kane, Kurt Mehlhorn, Thomas Sauerwald, and He Sun. Counting arbitrary subgraphs in data streams. In *Proc. 39th International Colloquium on Automata, Languages and Programming*, pages 598–609, 2012.

[88] Arijit Khan, Nan Li, Xifeng Yan, Ziyu Guan, Supriyo Chakraborty, and Shu Tao. Neighborhood based fast graph search in large networks. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, 2011.

[89] Ton Kloks, Dieter Kratsch, and Haiko Müller. Finding and counting small induced subgraphs efficiently. *Information Processing Letters*, 74(3-4):115–121, 2000.

[90] Tamara G Kolda, Ali Pinar, Todd Plantenga, C Seshadhri, and Christine Task. Counting triangles in massive graphs with mapreduce. *SIAM Journal on Scientific Computing*, 36(5):S48–S77, 2014.

[91] Tamara G. Kolda, Ali Pinar, Todd Plantenga, C. Seshadhri, and Christine Task. Counting triangles in massive graphs with mapreduce. *SIAM J. Scientific Computing*, 36(5), 2014.

[92] M. N. Kolountzakis, G. L. Miller, R. Peng, and C. Tsourakakis. Efficient triangle counting in large graphs via degree-based vertex partitioning. In *WAW'10*, 2010.

[93] Dániel Kondor, István Csabai, János Szüle, Márton Pósfai, and Gábor Vattay. Inferring the interplay between network structure and market effects in bitcoin. *New Journal of Physics*, 16(12):125003, 2014.

[94] Tsvi Kopelowitz, Seth Pettie, and Ely Porat. Higher lower bounds from the 3sum conjecture. In *Proc. 27th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2016.

[95] Ioannis Koutis and Ryan Williams. Limits and applications of group algebras for parameterized problems. In *International Colloquium on Automata, Languages and Programming*, pages 653–664, 2009.

[96] Lauri Kovanen, Márton Karsai, Kimmo Kaski, János Kertész, and Jari Saramäki. Temporal motifs in time-dependent networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2011(11):P11005, 2011.

[97] Mirosław Kowaluk, Andrzej Lingas, and Eva-Marta Lundell. Counting and detecting small subgraphs via equations. *SIAM Journal on Discrete Mathematics*, 27(2):892–909, 2013.

[98] Jérôme Kunegis. KONECT – The Koblenz Network Collection. In *Proc. Int. Conf. on World Wide Web Companion*, pages 1343–1350, 2013.

[99] Konstantin Kutzkov and Rasmus Pagh. On the streaming complexity of computing local clustering coefficients. In *ACM International Conference on Web Search and Data Mining*, pages 677–686, 2013.

[100] Matthieu Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical computer science*, 407(1-3):458–473, 2008.

[101] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[102] Yuchen Li, Zhengzhi Lou, Yu Shi, and Jiawei Han. Temporal motifs in heterogeneous information networks. In *MLG Workshop@ KDD*, 2018.

[103] Yongsub Lim and U. Kang. MASCOT: memory-efficient and accurate sampling for counting local triangles in graph streams. In *Annual SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 685–694, 2015.

[104] Paul Liu, Austin Benson, and Moses Charikar. A sampling framework for counting temporal motifs. *arXiv preprint arXiv:1810.00980*, 2018.

[105] Penghang Liu, Valerio Guarrasi, and A Erdem Sariyuce. Temporal network motifs: Models, limitations, evaluation. *IEEE Transactions on Knowledge and Data Engineering*, 2021.

[106] László Lovász. Operations with structures. *Acta Mathematica Academiae Scientiarum Hungarica*, 18(3-4):321–328, 1967.

[107] László Lovász. *Large networks and graph limits*, volume 60. American Mathematical Soc., 2012.

[108] Patrick Mackey, Katherine Porterfield, Erin Fitzhenry, Sutanay Choudhury, and George Chin. A chronological edge-driven approach to temporal subgraph isomorphism. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 3972–3979. IEEE, 2018.

[109] Madhusudan Manjunath, Kurt Mehlhorn, Konstantinos Panagiotou, and He Sun. Approximate counting of cycles in streams. In *Proc. 19th Annual European Symposium on Algorithms*, pages 677–688, 2011.

[110] Dror Marcus and Yuval Shavitt. Efficient counting of network motifs. In *IEEE 30th International Conference on Distributed Computing Systems Workshops*, pages 92–98. IEEE, 2010.

[111] Dror Marcus and Yuval Shavitt. Efficient counting of network motifs. In *ICDCS Workshops*, pages 92–98, 2010.

[112] David W Matula and Leland L Beck. Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM*, 30(3):417–427, 1983.

[113] Andrew McGregor, Sofya Vorotnikova, and Hoa T. Vu. Better algorithms for counting triangles in data streams. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 401–411, 2016.

[114] Andrew McGregor, Sofya Vorotnikova, and Hoa T. Vu. Better algorithms for counting triangles in data streams. In *ACM Symposium on Principles of Database Systems*, pages 401–411, 2016.

[115] Ine Melckenbeeck, Pieter Audenaert, Didier Colle, and Mario Pickavet. Efficiently counting all orbits of graphlets of any order in a graph using autogenerated equations. *Bioinformatics*, 34(8):1372–1380, 2018.

[116] Ine Melckenbeeck, Pieter Audenaert, Tom Michoel, Didier Colle, and Mario Pickavet. An algorithm to automatically generate the combinatorial orbit counting equations. *PLoS ONE*, 11(1):1–19, 01 2016.

[117] Tijana Milenković and Nataša Pržulj. Uncovering biological network function via graphlet degree signatures. *Cancer informatics*, 6:CIN–S680, 2008.

[118] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: Simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.

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

[120] Misael Mongioví, Giovanni Micale, Alfredo Ferro, Rosalba Giugno, Alfredo Pulvirenti, and Dennis Shasha. glabtrie: A data structure for motif discovery with constraints. In *Graph Data Management*, pages 71–95. Springer, 2018.

[121] Burkhard Monien. How to find long paths efficiently. In *North-Holland Mathematics Studies*, volume 109, pages 239–254. Elsevier, 1985.

[122] C. St. J. A. Nash-Williams. Decomposition of finite graphs into forests. *Journal of the London Mathematical Society*, 39(1):12, 1964.

[123] Jaroslav Nešetřil and Svatopluk Poljak. On the complexity of the subgraph problem. *Commentationes Mathematicae Universitatis Carolinae*, 26(2):415–419, 1985.

[124] Derek O'Callaghan, Martin Harrigan, Joe Carthy, and Pádraig Cunningham. Identifying discriminating network motifs in youtube spam. *arXiv preprint arXiv:1202.5216*, 2012.

[125] Mark Ortmann and Ulrik Brandes. Efficient orbit-aware triad and quad census in directed and undirected graphs. *Applied network science*, 2(1), 2017.

[126] R. Pagh and C. Tsourakakis. Colorful triangle counting and a mapreduce implementation. *Information Processing Letters*, 112:277–281, 2012.

[127] Gergely Palla, Imre Derényi, Illés Farkas, and Tamás Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *nature*, 435(7043):814–818, 2005.

[128] Ashwin Paranjape, Austin R Benson, and Jure Leskovec. Motifs in temporal networks. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, pages 601–610, 2017.

[129] Noujan Pashanasangi and C Seshadhri. Efficiently counting vertex orbits of all 5-vertex subgraphs, by evoke. In *Proc. 13th ACM International Conference on Web Search and Data Mining*, pages 447–455, 2020.

[130] Noujan Pashanasangi and C. Seshadhri. Faster and generalized temporal triangle counting, via degeneracy ordering. In *Proc. 27th Annual SIGKDD International Conference on Knowledge Discovery and Data Mining*, page 1319–1328, New York, NY, USA, 2021. Association for Computing Machinery.

[131] Mihai Patrascu. Towards polynomial lower bounds for dynamic problems. In *Proc. 42nd Annual ACM Symposium on the Theory of Computing*, 2010.

[132] A. Pavan, K. Tangwongsan, S. Tirthapura, and K.-L. Wu. Counting and sampling triangles from a graph stream. In *International Conference on Very Large Data Bases*, 2013.

[133] Aduri Pavan, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. Counting and sampling triangles from a graph stream. *Proceedings of the VLDB Endowment*, 6(14):1870–1881, 2013.

[134] Luka V Petrovic and Ingo Scholtes. Counting causal paths in big times series data on networks. *arXiv preprint arXiv:1905.11287*, 2019.

[135] Joseph J Pfeiffer, Timothy La Fond, Sebastian Moreno, and Jennifer Neville. Fast generation of large scale social networks while incorporating transitive closures. In *2012 International Conference on Privacy, Security, Risk and Trust and 2012 International Confernece on Social Computing*, pages 154–165. IEEE, 2012.

[136] Joseph J Pfeiffer III, Sebastian Moreno, Timothy La Fond, Jennifer Neville, and Brian Gallagher. Attributed graph models: Modeling network structure with correlated attributes. In *Proceedings of the 23rd international conference on World wide web*, 2014.

[137] Ali Pinar, C Seshadhri, and Vaidyanathan Vishal. Escape: Efficiently counting all 5-vertex subgraphs. In *Proceedings, International World Wide Web Conference (WWW)*. International World Wide Web Conferences Steering Committee, 2017.

[138] Alejandro Portes. Social capital: Its origins and applications in modern sociology. *Annual Review of Sociology*, 24(1):1–24, 1998.

[139] Natasa Przulj. Biological network comparison using graphlet degree distribution. *Bioinformatics*, 23(2):177–183, 2007.

[140] Natasa Przulj, Derek G. Corneil, and Igor Jurisica. Modeling interactome: scale-free or geometric?. *Bioinformatics*, 20(18):3508–3515, 2004.

[141] M. Rahman, M. A. Bhuiyan, and M. Al Hasan. Graft: An efficient graphlet counting method for large graph analysis. *IEEE Transactions on Knowledge and Data Engineering*, PP(99), 2014.

[142] Ursula Redmond and Pádraig Cunningham. Temporal subgraph isomorphism. In *2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2013)*, pages 1451–1452. IEEE, 2013.

[143] Fergal Reid and Martin Harrigan. An analysis of anonymity in the bitcoin system. In *Security and privacy in social networks*, pages 197–223. Springer, 2013.

[144] Pedro Ribeiro, Pedro Paredes, Miguel EP Silva, David Aparicio, and Fernando Silva. A survey on subgraph counting: concepts, algorithms and applications to network motifs and graphlets. *arXiv preprint arXiv:1910.13011*, 2019.

[145] Pedro Ribeiro and Fernando Silva. Discovering colored network motifs. In *Complex Networks V*, pages 107–118. Springer, 2014.

[146] Neil Robertson and Paul D Seymour. Graph minors. i. excluding a forest. *Journal of Combinatorial Theory, Series B*, 35(1):39–61, 1983.

[147] Neil Robertson and Paul D Seymour. Graph minors. iii. planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49–64, 1984.

[148] Neil Robertson and Paul D. Seymour. Graph minors. ii. algorithmic aspects of tree-width. *Journal of algorithms*, 7(3):309–322, 1986.

[149] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[150] Ryan A Rossi, Nesreen K Ahmed, Aldo Carranza, David Arbour, Anup Rao, Sungchul Kim, and Eunyee Koh. Heterogeneous network motifs. *arXiv preprint arXiv:1901.10026*, 2019.

[151] Rahmtin Rotabi, Krishna Kamath, Jon M. Kleinberg, and Aneesh Sharma. Detecting strong ties using network motifs. In *Proceedings, International World Wide Web Conference (WWW)*, pages 983–992, 2017.

[152] Marc Roth and Philip Wellnitz. Counting and finding homomorphisms is universal for parameterized complexity theory. In *Proc. 31st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2161–2180, 2020.

[153] Seyed-Vahid Sanei-Mehri, Ahmet Erdem Sariyüce, and Srikanta Tirthapura. Butterfly counting in bipartite networks. In *Annual SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2150–2159, 2018.

[154] Ahmet Erdem Sariyuce, C. Seshadhri, Ali Pinar, and Umit V. Catalyurek. Finding the hierarchy of dense subgraphs using nucleus decompositions. In *Proceedings, International World Wide Web Conference (WWW)*, pages 927–937, 2015.

[155] T. Schank and D. Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *Experimental and Efficient Algorithms*, pages 606–609. Springer, 2005.

[156] C. Seshadhri, Tamara G. Kolda, and Ali Pinar. Community structure and scale-free collections of Erdös-Rényi graphs. *Physical Review E*, 85(5):056109, May 2012.

[157] C. Seshadhri, Ali Pinar, and Tamara G. Kolda. Fast triangle counting through wedge sampling. In *Proceedings of the SIAM Conference on Data Mining*, 2013.

[158] C. Seshadhri, Ali Pinar, and Tamara G. Kolda. Wedge sampling for computing clustering coefficients and triangle counts on large graphs. *Statistical Analysis and Data Mining*, 7(4):294–307, 2014.

[159] C. Seshadhri and Srikanta Tirthapura. Scalable subgraph counting: The methods behind the madness: WWW 2019 tutorial. In *Proceedings, International World Wide Web Conference (WWW)*, 2019.

[160] C Seshadhri and Srikanta Tirthapura. Scalable subgraph counting: The methods behind the madness: Www 2019 tutorial. In *Proceedings of the Web Conference (WWW)*, volume 2, page 75, 2019.

[161] Comandur Seshadhri, Ali Pinar, and Tamara G Kolda. Triadic measures on graphs: The power of wedge sampling. In *Proceedings of the 2013 SIAM international conference on data mining*, pages 10–18. SIAM, 2013.

[162] Nino Shervashidze, S. V. N. Vishwanathan, Tobias Petri, Kurt Mehlhorn, and Karsten M. Borgwardt. Efficient graphlet kernels for large graph comparison. In *AISTATS*, pages 488–495, 2009.

[163] K. Shin, T. Eliassi-Rad, and C. Faloutsos. Patterns and anomalies in k-cores of real-world graphs with applications. *Knowledge and Information Systems*, 54(3):677–710, 2018.

[164] Kijung Shin. WRS: waiting room sampling for accurate triangle counting in real graph streams. In *International Conference on Data Mining*, pages 1087–1092, 2017.

[165] S. Son, A. Kang, H. Kim, T. Kwon, J. Park, and H. Kim. Analysis of context dependence in social interaction networks of a massively multiplayer online role-playing game. *PLoS ONE*, 7(4):e33918, 04 2012.

[166] Alina Stoica and Christophe Prieur. Structure of neighborhoods in a large social network. In *International Conference on Computational Science and Engineering*, volume 4, pages 26–33. IEEE, 2009.

[167] Xiaoli Sun, Yusong Tan, Qingbo Wu, Baozi Chen, and Changxiang Shen. Tm-miner: Tfs-based algorithm for mining temporal motifs in large temporal network. *IEEE Access*, 7, 2019.

[168] Xiaoli Sun, Yusong Tan, Qingbo Wu, Jing Wang, and Changxiang Shen. New algorithms for counting temporal graph pattern. *Symmetry*, 11(10):1188, 2019.

[169] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th international conference on World wide web*, pages 607–614, 2011.

[170] George Szekeres and Herbert S Wilf. An inequality for the chromatic number of a graph. *Journal of Combinatorial Theory*, 4(1):1–3, 1968.

[171] M. Szell and S. Thurner. Measuring social dynamics in a massive multiplayer online game. *Social Networks*, 32:313–329, 2010.

[172] Jie Tang, Jing Zhang, Limin Yao, Juanzi Li, Li Zhang, and Zhong Su. Arnetminer: extraction and mining of academic social networks. In *Annual SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 990–998. ACM, 2008.

[173] Charalampos E. Tsourakakis. The k-clique densest subgraph problem. In *Proceedings, International World Wide Web Conference (WWW)*, pages 1122–1132, 2015.

[174] Charalampos E. Tsourakakis, U. Kang, Gary L. Miller, and Christos Faloutsos. Doulion: counting triangles in massive graphs with a coin. In *Annual SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 837–846, 2009.

[175] Charalampos E. Tsourakakis, Jakub Pachocki, and Michael Mitzenmacher. Scalable motif-aware graph clustering. In *Proceedings, International World Wide Web Conference (WWW)*, pages 1451–1460, 2017.

[176] Kun Tu, Jian Li, Don Towsley, Dave Braines, and Liam D Turner. Network classification in temporal networks using motifs. *arXiv preprint arXiv:1807.03733*, 2018.

[177] Kun Tu, Jian Li, Don Towsley, Dave Braines, and Liam D Turner. gl2vec: Learning feature representation using graphlets for directed networks. In *Proceedings of the 2019 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, pages 216–221, 2019.

[178] Ata Turk and Duru Turkoglu. Revisiting wedge sampling for triangle counting. In *Proceedings, International World Wide Web Conference (WWW)*. ACM, 2019.

[179] Johan Ugander, Lars Backstrom, and Jon M. Kleinberg. Subgraph frequencies: mapping the empirical and extremal geography of large graph collections. In *Proceedings, International World Wide Web Conference (WWW)*, pages 1307–1318, 2013.

[180] Sergi Valverde and Ricard V Solé. Network motifs in computational graphs: A case study in software architecture. *Physical Review E*, 72(2):026107, 2005.

[181] Virginia Vassilevska. Efficient algorithms for clique problems. *Information Processing Letters*, 109(4):254–257, 2009.

[182] Virginia Vassilevska and Ryan Williams. Finding, minimizing, and counting weighted subgraphs. In *Proc. 41st Annual ACM Symposium on the Theory of Computing*, pages 455–464, 2009.

[183] A Vazquez, R Dobrin, D Sergi, J-P Eckmann, Zoltan N Oltvai, and A-L Barabási. The topological relationship between the large-scale attributes and local interaction patterns of complex networks. *Proceedings of the National Academy of Sciences*, 101(52):17940–17945, 2004.

[184] Davide Vega and Matteo Magnani. Foundations of temporal text networks. *Applied network science*, 3(1):1–26, 2018.

[185] Jingjing Wang, Yanhao Wang, Wenjun Jiang, Yuchen Li, and Kian-Lee Tan. Efficient sampling algorithms for approximate temporal motif counting. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, 2020.

[186] Pinghui Wang, Junzhou Zhao, Xiangliang Zhang, Zhenguo Li, Jiefeng Cheng, John C. S. Lui, Don Towsley, Jing Tao, and Xiaohong Guan. MOSS-5: A fast method of approximating counts of 5-node graphlets in large graphs. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 30(1):73–86, 2018.

[187] Pinghui Wang, Junzhou Zhao, Xiangliang Zhang, Zhenguo Li, Jiefeng Cheng, John CS Lui, Don Towsley, Jing Tao, and Xiaohong Guan. Moss-5: A fast method of approximating counts of 5-node graphlets in large graphs. *IEEE Transactions on Knowledge and Data Engineering*, 30(1):73–86, 2017.

[188] Stanley Wasserman, Katherine Faust, et al. *Social network analysis: Methods and applications*. Cambridge university press, 1994.

[189] Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world'networks. *nature*, 393(6684):440–442, 1998.

[190] B. Welles, A. Van Devender, and N. Contractor. Is a friend a friend?: Investigating the structure of friendship networks in virtual worlds. In *CHI-EA'10*, pages 4027–4032, 2010.

[191] S. Wernicke and F. Rasche. Fanmod: a tool for fast network motif detection. *Bioinformatics*, 22(9):1152–1153, 2006.

[192] Sebastian Wernicke. Efficient detection of network motifs. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 3(4):347–359, 2006.

[193] Wikimedia Foundation. Wikimedia downloads. http://dumps.wikimedia.org/, January 2010.

[194] Virginia Vassilevska Williams, Joshua R Wang, Ryan Williams, and Huacheng Yu. Finding four-node subgraphs in triangle time. In *Proc. 26th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1671–1680, 2014.

[195] Ömer Nebil Yaveroğlu, Noël Malod-Dognin, Darren Davis, Zoran Levnajic, Vuk Janjic, Rasa Karapandza, Aleksandar Stojmirovic, and Nataša Pržulj. Revealing the hidden language of complex networks. *Scientific reports*, 4(1):1–9, 2014.

[196] Hao Yin, Austin R. Benson, and Jure Leskovec. Higher-order clustering in networks. *Phys. Rev. E*, 97:052306, 2018.

[197] Hao Yin, Austin R. Benson, and Jure Leskovec. The local closure coefficient: A new perspective on network clustering. In *ACM International Conference on Web Search and Data Mining (WSDM)*, pages 303–311, 2019.

[198] Jin-Hyun Yoon and Sung-Ryul Kim. Improved sampling for triangle counting with MapReduce. In *Convergence and Hybrid Information Technology*, volume 6935, pages 685–689. 2011.

[199] Zhao Zhao, Guanying Wang, Ali R Butt, Maleq Khan, VS Anil Kumar, and Madhav V Marathe. Sahad: Subgraph analysis in massive networks using hadoop. In *IEEE 26th International Parallel and Distributed Processing Symposium*, pages 390–401. IEEE, 2012.

[200] Matteo Zignani, Sabrina Gaito, Gian Paolo Rossi, Xiaohan Zhao, Haitao Zheng, and Ben Zhao. Link and triadic closure delay: Temporal metrics for social network dynamics. In *Proceedings of the International AAAI Conference on Web and Social Media*, 2014.