Finding and Counting Patterns in Sparse Graphs

Balagopal Komarath ✉
IIT Gandhinagar, India

Anant Kumar ✉
IIT Gandhinagar, India

Suchismita Mishra ✉
Universidad Andrés Bello, Santiago, Chile

Aditi Sethia ✉
IIT Gandhinagar, India

Abstract

We consider algorithms for finding and counting small, fixed graphs in sparse host graphs. In the non-sparse setting, the parameters treedepth and treewidth play a crucial role in fast, constant-space and polynomial-space algorithms respectively. We discover two new parameters that we call matched treedepth and matched treewidth. We show that finding and counting patterns with low matched treedepth and low matched treewidth can be done asymptotically faster than the existing algorithms when the host graphs are sparse for many patterns. As an application to finding and counting fixed-size patterns, we discover $\widetilde{O}(m^3)$-time[1], constant-space algorithms for cycles of length at most 11 and $\widetilde{O}(m^2)$-time, polynomial-space algorithms for paths of length at most 10.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Theory of computation → Graph algorithms analysis

Keywords and phrases Subgraph Detection and Counting, Homomorphism Polynomials, Treewidth and Treedepth, Matchings

Digital Object Identifier 10.4230/LIPIcs.STACS.2023.40

Related Version *Full Version*: http://bkomarath.rbgo.in/papers/KKMS23.pdf

Supplementary Material *Dataset (Graphs)*: https://github.com/anonymous1203/Spasm

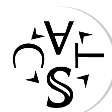
Acknowledgements The research work of S. Mishra is partially funded by Fondecyt Postdoctoral grant 3220618 of Agencia National de Investigatión y Desarrollo (ANID), Chile.

1 Introduction

Given simple graphs G, called the *pattern*, and H, called the *host*, a fundamental computational problem is to find or count occurrences of G in H. What does it mean for G to occur in H? The three most common notions of occurrence are characterized by mappings $\phi : V(G) \mapsto V(H)$. We say:

1. If $\{u, v\} \in E(G)$ implies $\{\phi(u), \phi(v)\} \in E(H)$ and ϕ is one-to-one, then we say that ϕ witnesses a subgraph isomorphic to G in H. The subgraph is obtained by taking the vertices and edges in the image of ϕ. The number of G-subgraphs of H is just the number of such subgraphs G' of H.
2. If $\{u, v\} \in E(G)$ is equivalent to $\{\phi(u), \phi(v)\} \in E(H)$ and ϕ is one-to-one, then ϕ witnesses an induced subgraph isomorphic to G in H. The induced subgraph is obtained by taking the vertices and *all* edges induced by those vertices in the image of ϕ.

[1] $\widetilde{O}$ hides factors that are logarithmic in the input size.

© Balagopal Komarath, Anant Kumar, Suchismita Mishra, and Aditi Sethia;
licensed under Creative Commons License CC-BY 4.0
40th International Symposium on Theoretical Aspects of Computer Science (STACS 2023).
Editors: Petra Berenbrink, Patricia Bouyer, Anuj Dawar, and Mamadou Moustapha Kanté;
Article No. 40; pp. 40:1–40:20


Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

3. If $\{u, v\} \in E(G)$ implies $\{\phi(u), \phi(v)\} \in E(H)$, then we say that ϕ is a homomorphism from G to H. Note that unlike a subgraph isomorphism, ϕ is not required to be one-to-one.

For any of these notions, the detection problem is clearly in NP. All three of them are also straightforward generalizations of the NP-hard problem CLIQUE. Therefore, the existence of efficient algorithms for finding or counting patterns under any of these notions is unlikely in general.

The class of pattern detection and counting problems remain interesting even if we restrict our attention to fixed pattern graphs. Williams [20] showed that the improved algorithms for finding triangles could be used to find faster algorithms for even NP-complete problems such as MAX2SAT. For fixed pattern graphs of size k, the brute-force search algorithm is as follows: Iterate over all k-tuples over $V(H)$ and check whether G occurs in the induced subgraph of H on the vertices in that k-tuple. This algorithm takes $\theta(n^k)$ time and constant space. Therefore, when we restrict our attention to fixed patterns, we seek improvements over this running time preferably keeping the space usage low. There are two broad techniques that reduce the running-time: the usage of fast matrix multiplication algorithms as a sub-routine and the exploitation of structural properties of pattern graphs.

If A is the adjacency matrix of the graph, then Nešetřil and Poljak [16] showed that one can obtain an $O(n^\omega)$-time algorithm for counting triangles using the identity $\mathsf{trace}(A^3) = 6\Delta$, where Δ is the number of triangles in the graph, where $\omega < 2.38$ is the matrix multiplication exponent. Using a simple reduction, they extended this to an algorithm to count $3k$-cliques in $O(n^{k\omega})$-time. They also showed that we can use improved algorithms for counting k-cliques to count any k-vertex pattern. Later, Kloks, Kratsch and Müller [12] showed how to use fast rectangular matrix multiplication to obtain similar improvements to the running time for counting cliques of all sizes, not just multiples of three. Note that the improvements obtained by these algorithms are applicable to all k-vertex patterns. i.e., they do not use the pattern's structure to obtain better algorithms. Since finding a k-clique requires $n^{\Omega(k)}$-time unless ETH is false, we need to exploit the structure of the pattern to obtain significantly better algorithms.

For patterns sparser than cliques, the run-time can be significantly improved over even fast matrix multiplication based (pattern finding) algorithms. The crucial idea is to exploit the structure of the pattern graph. A k-*walk polynomial* is a polynomial where the monomials correspond to walks that are k vertices long. For example, a walk (u, v, w, x) will correspond to the monomial $x_{uv}x_{vw}x_{wx}$ and a walk (u, v, u, v) to the monomial x_{uv}^3 [2]. Williams [21] showed that we can detect k-paths in graphs by (1) computing the k-walk polynomial and (2) checking whether it has multilinear monomials. We can compute the k-walk polynomial in linear-time using a simple dynamic programming algorithm and then multilinear monomials can be detected with high probability by evaluating this polynomial over an appropriate ring where the randomly chosen elements satisfy $a^2 = 0$. This yields is a $O(2^k(n + m))$-time algorithm for finding k-vertex paths as subgraphs in n-vertex, m-edge host graphs.

We now consider the problem of counting sparse patterns. For counting k-paths as subgraphs, the best known algorithm by Curticapean, Dell and Marx [4] takes only $O(f(k)n^{0.174k+o(k)})$-time for some function f. Coming to fixed pattern graphs, Alon, Yuster and Zwick [1] gave $O(n^\omega)$-time algorithms for counting cycle subgraphs of length at most 8 using an algorithm that *combines* fast matrix multiplication and exploitation of the structure of the pattern. Notice that this is the same as the time required for counting triangles (3-cliques).

[2] We write uv to denote the edge $\{u, v\}$.

The notion of graph homomorphisms was shown to play a crucial role in all the above improved algorithms for finding and counting non-clique subgraphs. More specifically, Fomin, Lokshtanov, Raman, Rao, and Saurabh [11] showed how the efficient construction of homomorphism polynomials (see Definition 17), a generalization of k-walk polynomials, can be used to detect subgraphs with small treewidth efficiently. Their algorithm can be seen as a generalization of Williams's algorithm [21] for k-paths to arbitrary graphs. Similarly, Curticapean, Dell and Marx [4] showed that efficient algorithms for counting subgraphs can be derived from efficient algorithms for counting homomorphisms of graphs of small treewidth. Their algorithm can be seen as a generalization of the cycle-counting algorithms of Alon, Yuster, and Zwick [1].

Algorithms for finding and counting patterns in *sparse* host graphs are also studied. An additional parameter, m, the number of edges in the host graph, is taken into account for the design and analysis of these algorithms. In the worst-case, m could be as high as $\binom{n}{2}$, and hence, an $O(n^t)$-time algorithm and an $O(m^{t/2})$-time algorithm for some t have the same asymptotic time complexity. However, it is common in practice that $m = o(n^2)$. For example, if the host graph models a road network, then $m = O(n)$, where the constant factor is determined by the maximum number of roads at any intersection. In such cases, an $O(m^{t/2})$-time algorithm is asymptotically better than an $O(n^t)$-time algorithm.

The broad themes of using fast matrix multiplication and/or structural parameters of the pattern to obtain improved algorithms are still applicable in the setting of sparse host graphs. Using fast matrix multiplication, Eisenbrand and Grandoni [8] showed that we can count k-cliques in $O(m^{k\omega/6})$-time. Kloks, Kratsch and Müller [12] showed that K_4 subgraphs can be counted in $O(m^{(\omega+1)/2})$-time. Again, since $\omega < 3$, this is better than the $O(m^2)$-time given by the brute-force algorithm. Using structural parameters of the pattern, Kowaluk, Lingas, and Lundell [15] obtained many improved algorithms in the sparse host graph setting. For example, their methods obtain an algorithm that runs in $O(m^4)$-time for counting P_{10} as subgraphs. In this work, we obtain an $\widetilde{O}(m^2)$-time algorithm for counting P_{10} (See Theorems 12,13,14,15 for similar improvements).

The model of computation that we consider is the unit-cost RAM model. In particular, we can store labels of vertices and edges in the host graph in a constant number of words[3]. In this model, algorithms based on fast matrix multiplication and/or treewidth mentioned above use polynomial space. However, the brute-force search algorithm uses only constant space as it only needs to store k vertex labels at a time (Recall that we regard k as a constant.). How much speed-up can we obtain while preserving constant space usage? The graph parameter *treedepth* plays a crucial role in answering this question. It is well known that we can count the homomorphisms from a pattern of treedepth d in $O(n^d)$-time while using only constant space (See Komarath, Rahul, and Pandey [13] for a construction of arithmetic formulas counting them. These arithmetic formulas can be implicitly constructed and evaluated in constant space.). Since all k-vertex patterns except k-clique has treedepth strictly less than k, this immediately yields an improvement over the running-time of brute-force while preserving constant space usage. In this work, we improve upon the treedepth-based algorithms for sparse host graphs where the pattern graph is a cycle of length at most 11 (See Theorem 2).

[3] In the TM model or the log-cost RAM model, storing labels of vertices would take $O(\log n)$ space.

1.1 Connection to arithmetic circuits for graph homomorphism polynomials

A popular sub-routine in these algorithms is an algorithm by Diaz, Serna and Thilikos [6] that efficiently counts the number of homomorphisms from a pattern of small treewidth to an arbitrary host graph. Indeed, it can be shown that this algorithm can be easily generalized to *efficiently construct* circuits for homomorphism polynomials instead of counting homomorphisms. Bläser, Komarath and Sreenivasaiah [2] showed that efficient constructions for homomorphism polynomials can even be used to detect *induced* subgraphs in some cases. They also show that many of the faster induced subgraph detection algorithms, such finding four-node subgraphs by Williams et al. [22] and five-node subgraphs by Kowaluk, Lingas, and Lundell [15] can be described as algorithms that efficiently construct these homomorphism polynomials. Therefore, arithmetic circuits for graph homomorphism polynomials provide a unifying framework for describing almost all the fast algorithms that we know for finding and counting subgraphs and finding induced subgraphs. Can we improve these algorithms by finding more efficient ways to construct arithmetic circuits for homomorphism polynomials? Unfortunately, it is known that for the type of circuit that is constructed, i.e., circuits that do not involve cancellations, the existing constructions are the best possible for *all* pattern graphs, as shown by Komarath, Pandey, and Rahul [13]. The situation is similar for constant space algorithms. The best known algorithms can be expressed as divide-and-conquer algorithms that evaluate small formulas constructed by making use of the graph parameter treedepth. Komarath, Pandey, and Rahul[13] also showed that the running-time of these algorithms match the best possible formula size for *all* pattern graphs. These arithmetic circuit lower bounds serve as a technical motivation for considering sparse host graphs, in addition to the practical motivation mentioned earlier.

1.2 Our findings

In this paper, we study algorithms for finding and counting patterns in host graphs that work well especially when the host is sparse. We discover algorithms that are (1) strictly better than the brute-force algorithm, (2) strictly better than the best-known algorithms when the host graph is sparse, (3) close to the best-known algorithms when the host graphs are dense. Our algorithms are based on two new structural graph parameters – the *matched treedepth* and *matched treewidth*. (See 19 and 21 for formal definitions). We show that they can be used to obtain improved running times for algorithms that use constant space and polynomial space respectively. Our algorithms are summarized in Table 1. In the table, the parameter m is the number of edges in the host graph. We denote using mtw the matched treewidth of the pattern and using mtd the matched treedepth of the pattern. The notation $\widetilde{O}$ hides factors that are poly-logarithmic in the input (the host graph) size.

We now explain the relevance of our new parameters; state our algorithms, the relationships between various graph parameters, and some structural characterizations that we prove in this paper in the rest of this section.

Treedepth and matched treedepth

The matched treedepth of a graph is closely related to its treedepth. The constant space algorithm based on treedepth is essentially an divide-and-conquer algorithm over a elimination tree of the pattern graph that executes a brute-force search over each root-to-leaf path in the elimination tree. Therefore, it runs in time $O(n^d)$. We exploit the fact that the elimination tree is matched, which forces an additional constraint that the vertices in each root-to-leaf

Table 1 Pattern counting and detection algorithms for sparse host graphs.

Pattern	Type	Problem	Time	Space	Remarks
C_k	Subgraph	Counting	$\widetilde{O}(m^3)$	$O(1)$	$k \leqslant 11$
P_k	Subgraph	Counting	$\widetilde{O}(m^2)$	$\widetilde{O}(m^2)$	$k \leqslant 10$
C_k	Subgraph	Counting	$\widetilde{O}(m^2)$	$\widetilde{O}(m^2)$	$k \leqslant 9$
Any	Homomorphism	Counting	$\widetilde{O}(m^{\lceil \mathsf{mtd}/2 \rceil})$	$O(1)$	
Any	Homomorphism	Counting	$\widetilde{O}(m^{\lceil (\mathsf{mtw}+1)/2 \rceil})$	$\widetilde{O}(m^{\lceil (\mathsf{mtw}+1)/2 \rceil})$	
C_6	Induced subgraph	Detection	$\widetilde{O}(m^2)$	$\widetilde{O}(m^2)$	
$\overline{P_k}$	Induced subgraph	Detection	$\widetilde{O}(m^{\lceil (k-2)/2 \rceil})$	$\widetilde{O}(m^{\lceil (k-2)/2 \rceil})$	

path has to be covered by a matching. This allows the brute-force part of the algorithm to discover all d vertices on the path using only $d/2$ edges. The central algorithm that we use to obtain constant space algorithms is given below:

▶ **Theorem 1.** *Let G be a graph with $\mathsf{mtd}(G) = d$, then given an m-edge graph H as input, we can count the number of homomorphisms from G to H in $\widetilde{O}(m^{\lceil d/2 \rceil})$-time and constant space.*

It is well-known that the number of G-subgraphs, for any G, can be expressed as a linear combination of the number of homomorphisms from a related set of graphs called the *spasm of* G. The spasm of G contains exactly all graphs that can be obtained by iteratively merging the independent sets in G. Although the treedepth of the spasm of C_{11} is bounded by 6, however, the matched treedepth is not necessarily bounded by the treedepth. We analyze all graphs in the spasm of C_{10} and C_{11} (there are 501 such graphs) and show that the matched treedepth of each graph is at most 6. This yields the following algorithm:

▶ **Theorem 2.** *Given an m-edge graph H as input, we can count the number of C_k, where $k \leqslant 11$, as subgraphs in $\widetilde{O}(m^3)$-time and constant space.*

For comparison, the brute-force algorithm takes $O(m^6)$-time and constant space; and the treedepth based algorithm takes $O(n^6)$-time and constant space.

As seen from the proof of our algorithm for counting C_{11}, the spasm of a pattern can contain a large number of graphs even for relatively small patterns. Therefore, it would be nice to have theorems that upper-bound the matched treedepth. Unfortunately, the property $\mathsf{mtd}(G) \leqslant k$ is not even subgraph-closed unlike treedepth. For example, it can be proved that $\mathsf{mtd}(K_4 - e) = 3$ but $\mathsf{mtd}(C_4) = 4$. However, interesting structural observations can still be made for matched treedepth. The following is a theorem that upper-bounds matched treedepth in terms of treedepth.

▶ **Theorem 3.** *For any graph G, $\mathsf{mtd}(G) \leqslant 2 \cdot \mathsf{td}(G) - 2$.*

Theorem 3 implies that our constant-space algorithms from Theorem 1 for counting homomorphisms are asymptotically faster for *all* patterns, where the inputs are sparse host graphs, when compared to the treedepth-based algorithm.

The following theorem shows that the time complexity for counting homomorphisms of a pattern is lower-bounded by the time complexity for counting all of its induced subgraphs.

▶ **Theorem 4.** *Let G be a graph and G' is a connected, induced subgraph of G, then:*
1. $\mathsf{mtd}(G') \leqslant \mathsf{mtd}(G)$ *if* $\mathsf{mtd}(G)$ *is even.*
2. $\mathsf{mtd}(G') \leqslant \mathsf{mtd}(G) + 1$ *if* $\mathsf{mtd}(G)$ *is odd.*

In light of the importance of matched treedepth, it becomes crucial that we understand this structural parameter as much as possible. The graphs of treedepth 2 are exactly the class of star graphs. This is also the class of graphs with matched treedepth 2. However, for the graph C_4, we have $\mathsf{td}(C_4) = 3$ and $\mathsf{mtd}(C_4) = 4$. So it is interesting to know what are exactly the graphs where treedepth and matched treedepth coincide. The following theorem should be viewed as giving us a preliminary understanding of the relationship between these two parameters.

▶ **Theorem 5.** *Let* G *be a graph such that* $\mathsf{td}(G) = 3$*. Then* $\mathsf{mtd}(G) = 3$ *if and only if* G *is* $(C_4, P_6, T_{3,3})$*-free.*

The graph $T_{3,3}$ is the $(3, 3)$ tadpole graph.

Treewidth and matched treewidth

The treewidth-based dynamic programming algorithm of Díaz, Serna, and Thilikos [6] can be strengthened to output an *arithmetic circuit* that computes the *homomorphism polynomial* for the pattern. An arithmetic circuit is a directed acyclic graph where each internal node is labeled $+$ or $\times$, each leaf is labeled by a variable or a field constant, and there is a designated output node. Such a graph computes a polynomial over the underlying field in a natural fashion. We find that by using a dynamic programming algorithm over matched tree decompositions, we can improve the size of the arithmetic circuit for *sparse* host graphs. Our central theorem is given below:

▶ **Theorem 6.** *Let* G *be a graph with* $\mathsf{mtw}(G) = t$*, then given an* m*-edge host graph* H *as input, we can construct an arithmetic circuit computing the homomorphism polynomial from* G *to* H *in time* $\widetilde{O}(m^{\lceil (t+1)/2 \rceil})$.

For graphs where matched treewidth and treewidth coincide, the running time for counting homomorphisms is a quadratic improvement on the algorithm by Díaz, Serna, and Thilikos [6] for sparse graphs. Therefore, this is also the best possible improvement one can hope to get without improving upon the algorithm by Díaz, Serna and Thilikos [6]. What is the worst case? The following theorem implies that the resulting algorithm *cannot* be worse on sparse host graphs.

▶ **Theorem 7.** *For any graph* G*, we have* $\mathsf{mtw}(G) \leqslant 2 \cdot \mathsf{tw}(G) + 1$.

Unfortunately, unlike for treewidth, the parameter $\mathsf{mtw}(G)$ is not monotone over the subgraph partial order. We first observe an explicit graph family with lower tw and larger mtw. Consider the complete bipartite graph $K_{n,n}$ on n vertices. Notice that $tw(K_{n,n}) = n$.

▶ **Proposition 8.** $\mathsf{mtw}(K_{n,n}) = 2n - 2$ *for all* $n > 1$.

The following observation shows that there exists supergraphs of $K_{n,n}$ with lower mtw than that of $K_{n,n}$.

▶ **Observation 9.** *Consider the supergraph* G *of* $K_{n,n}$ *such that* $V(G) = V(H)$*, and there are edges in one partition of* $K_{n,n}$ *such that the independent set of size* n *becomes a path on* n *vertices. Note that although* $\mathsf{mtw}(K_{n,n}) = 2n - 2$*, but* $\mathsf{mtw}(G) = n$*.*

We show how we can use structural theorems about matched treewidth to prove algorithmic upper bounds. For example, to count the number of P_{10} subgraphs, we only have to show that all graphs in the spasm of P_{10} have low matched treewidth. The spasm of P_{10} is a large set that contains more than 300 graphs. Indeed, it is possible to analyze the matched treewidth for each of these graphs individually. However, it would be better if we have theorems that eliminate such tedious work.

We derive some structural theorems for low values of matched treewidth. Graphs with matched treewidth 1 are exactly trees. We also show $\mathsf{tw}(C_5) = 2$ and $\mathsf{mtw}(C_5) = 3$.

We characterize the matched treewidth of partial 2-trees using forbidden induced minors (See Definition 24) wherever possible. We show that C_5 is exactly the obstruction that forces higher matched treewidth for partial 2-trees.

▶ **Theorem 10.** *For any partial 2-tree* G*, the graph* G *is* C_5*-induced-minor-free if and only if* $\mathsf{mtw}(G) = 2$.

Notice that $\mathsf{tw}(G) = 2$ yields $O(n^3)$-time algorithms for counting homomorphisms. Even if $\mathsf{mtw}(G) = 3$, we obtain $\widetilde{O}(m^2)$-time algorithms for counting homomorphisms which is an improvement for sparse graphs. Does all treewidth 2 graphs have matched treewidth at most 3? No. The graph X in Figure 1 has treewidth 2 and matched treewidth 4. In fact, we can prove that X is exactly the obstruction that forces treewidth 2 graphs to have matched treewidth 4.

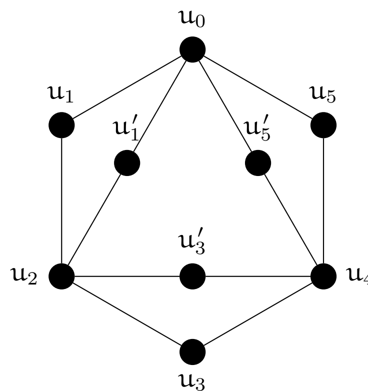


Figure 1 The graph X.

▶ **Theorem 11.** *For any partial 2-tree* G*, the graph* G *is* X*-induced-minor-free if and only if* $\mathsf{mtw}(G) \leqslant 3$.

This theorem implies that all X-induced-minor-free, treewidth 2 patterns have $\widetilde{O}(m^2)$-time homomorphism counting algorithms. This is an improvement for sparse host graph even over the fast matrix multiplication based algorithm given by Curticapean, Dell, and Marx [9] for counting homomorphisms from treewidth 2 graphs that runs in $O(n^\omega)$-time. Since the spasm of P_{10} does not contain any treewidth 4 graph or graph with an X-induced minor, we can show that there is an $\widetilde{O}(m^2)$-time algorithm for counting subgraph isormophisms of all paths on at most 10 vertices by showing that all treewidth 3 graphs in the spasm of P_{10} has matched treewidth 3. There are only 18 such graphs. Analyzing their matched treewidth yields the following theorem:

▶ **Theorem 12.** *Given an* m*-edge graph* H *as input, we can count the number of* P_k *subgraphs, where* $k \leqslant 10$*, in* $\widetilde{O}(m^2)$*-time.*

To the best of our knowledge, the best known path counting algorithms take $\Omega(n^4)$ time for paths on 10 vertices. Therefore, our algorithm is a significant improvement for sparse host graphs and no worse than the best known algorithm for dense host graphs. An easy corollary of the proof of this result is given below:

▶ **Theorem 13.** *Given an* m*-edge graph* H *as input, we can count the number of cycles of length at most* 9 *in* $\widetilde{O}(m^2)$*-time.*

These cycle counting algorithms are an improvement on sparse graphs over the $O(n^\omega)$-time algorithms for cycles of length at most 8 given by Alon, Yuster and Zwick [1].

We also show how to use our improved homomorphism polynomial construction algorithm to speed up detection of induced subgraphs. In particular, we show the following:

▶ **Theorem 14.** *Given an m-edge host graph as input, we can find an induced C_6 or report that none exists in $\widetilde{O}(m^2)$-time.*

This algorithm is no worse than the $O(n^4)$ time algorithm that can be derived using the techniques by Bläser, Komarath, and Sreenivasaiah [2]. For sparse graphs, our algorithm provides a quadratic improvement. We also show the following:

▶ **Theorem 15.** *Given an m-edge host graph as input, we can find an induced $\overline{P_k}$ or report that none exists in $\widetilde{O}(m^{(k-2)/2})$-time.*

This is also a quadratic improvement over the $O(n^{k-2})$ time algorithm given by Bläser, Komarath, and Sreenivasaiah [2] when the host graph is sparse. These algorithms are obtained by analyzing the matched treewidth *and* the automorphism structure of a set of graphs derived from the pattern.

From a technical standpoint, we see our algorithms as a natural combination of pattern detection and counting algorithms that work well on sparse host graph such as the $\widetilde{O}(m)$ algorithm for counting k-walks, the $\widetilde{O}(m^{k/2})$ algorithm for counting k-cliques, and $\widetilde{O}(m^{(k-1)/2})$ time algorithm for detecting induced $K_k - e$ by Vassilevska [18] and insights that improve the running-time on dense graphs by exploiting structural parameters treedepth and treewidth. We do not make use of fast matrix multiplication in any of our algorithms. Such algorithms, called *combinatorial algorithms*, are also of general interest to the community.

1.3 Related work

Algorithms for counting *induced subgraphs* are related to the problems that we consider but we do not consider any algorithms for it. This problem seems to be much harder. It is conjectured by Floderus, Kowaluk, Lingas, Lundell [9] that counting induced subgraphs for any k-vertex pattern graph is as hard as counting k-cliques for sufficiently large k. Several works have considered the parameterized complexity of counting subgraphs (See [5, 4, 17, 10, 7]) where the primary goal is to obtain a dichotomy of easy vs hard based on structural graph parameters. Some works have also considered restrictions on host graphs such as d-degeneracy [3]. The papers on parameterized complexity primarily chooses to focus on the growth-rate of the exponent for a family of patterns such as k-paths, k-cycles, or k-cliques and not the exact exponent for small graphs as we do in this paper.

2 Preliminaries

We consider simple graphs. We refer the reader to Douglas West's textbook [19] for basic definitions in graph theory. We use the following common notations for some well-known graphs: P_k for k-vertex paths, C_k for k-cycles, K_k for k-cliques, $K_k - e$ for k-clique with one edge missing, $K_{m,n}$ for complete bipartite graphs. A k-star is a $(k+1)$-vertex graph with a vertex u adjacent to vertices $v_1, \ldots, v_k$ and no other edges. A *star graph* is a k-star for some k. For a graph G and $S \subseteq V(G)$, we denote by $G[S]$ the subgraph of G induced by the vertices in S.

▶ **Definition 16.** *Given two graphs G and H, a graph homomorphism from G to H is a map $\phi : V(G) \to V(H)$ such that if $uv \in E(G)$, then $\phi(u)\phi(v) \in E(H)$.*

We denote by $\mathrm{Hom}(G, H)$, the set of all homomorphisms from G to H.

▶ **Definition 17.** *Given two graphs* G *and* H*, a homomorphism polynomial is an associated polynomial* $\mathsf{Hom}_{\mathsf{G}}[\mathsf{H}]$ *such that there is a one-to-one correspondence between its monomials and the homomorphisms from* G *to* H*. We define:*

$$\mathsf{Hom}_{\mathsf{G}}[\mathsf{H}] = \sum_{\phi \in \mathrm{Hom}(G,H)} \prod_{u \in V(G)} y_{\phi(u)} \prod_{\{u,v\} \in E(G)} x_{\{\phi(u),\phi(v)\}}$$

Note that H *has a subgraph isomorphic to* G *if and only if* P_G *has a multilinear monomial.*

We say that a graph G' is $(G_1, \ldots, G_m)$-free if no induced subgraph of G' is isomorphic to G_i for any i. We denote the complement of a graph G by $\overline{G}$. We have $V(G) = V(\overline{G})$ and the edges of $\overline{G}$ are exactly the non-edges of G and vice versa.

We assume that all pattern graphs are connected. Since our primary algorithms are all based on counting homomorphisms, this does not lose generality as the number of homomorphisms from a disconnected pattern is just the product of the number of homomorphisms from its components.

▶ **Definition 18.** *An* elimination tree (T, r) *of a connected graph* G *is a tree rooted at* $r \in V(G)$*, where the sub-trees of* r *are recursively elimination trees of the connected components of the graph* $G \setminus r$*. The elimination tree of an empty (no vertices or edges) graph is the empty tree. The depth of an elimination tree* (T, r) *is defined as the maximum number of vertices over all root-to-leaf paths in* T*. The* treedepth *of a graph* G*, denoted* $\mathsf{td}(G)$*, is the minimum depth among all possible elimination trees of* G*.*

Intuitively, treedepth measures the closeness of a given graph to star graphs which are exactly the connected graphs having treedepth 2. We introduce a related notion called matched treedepth that seems to be helpful when designing algorithms for finding or counting patterns in sparse host graphs.

▶ **Definition 19.** *A* matched elimination tree *for a graph* G *is an elimination tree such that the following conditions are true for any root-to-leaf path* $(v_1, \ldots, v_k)$*:*

- *If* k *is even, then* $v_1v_2, v_3v_4, \ldots, v_{k-1}v_k$ *is a matching in* G*.*
- *If* k *is odd, then there is some* i *such that* $E' = \{v_1v_2, \ldots, v_{i-1}v_i, v_iv_{i+1}, \ldots, v_{k-1}v_k\}$ *and* $E' \subseteq E(G)$*. We have that* $E' \setminus \{v_{i-1}v_i, v_iv_{i+1}\}$ *is a matching on* $(k-3)/2$ *vertices.*

The matched treedepth *of a graph* G*, denoted* $\mathsf{mtd}(G)$*, is the minimum depth among all possible matched elimination trees of* G*.*

The matched treedepth is always finite.

▶ **Definition 20.** *A* tree decomposition *of a graph* G *is a pair* $(T, B(t)_{t \in T})$ *where* T *is a tree and* $B(t)$*, called a* bag*, is a collection of subset of vertices of* G *corresponding to every node* $t \in T$*.*

- *(Connectivity Property) For all* $v \in V(G)$*, there is a node* $t \in V(T)$ *such that* $v \in B(t)$ *and all such nodes* t *that contain* v *form a connected component in* T*.*
- *(Edge Property) For all* $e \in E(G)$*, there is a node* $t \in V(T)$ *such that* $e \subseteq B(t)$*.*

The width of a tree decompostion (T, B) *is defined as the maximum bag size minus one, that is,* $\max_{t \in T} |B(t) - 1|$*. The* treewidth *of a graph* G*,* $\mathsf{tw}(G)$*, is the minimum possible width among all possible tree decompositions of* G*.*

Intuitively, treewidth measures the closeness of the given graph to trees which are exactly the graphs with treewidth 1. We introduce a related notion, called matched treewidth, closely related to treewidth, that seems to be useful for designing dynamic programming algorithms over sparse host graphs.

▶ **Definition 21.** *A* matched tree decomposition *for a graph* G *is a tree decomposition where for every bag in the tree decomposition, the subgraph of* G *induced by the vertices in that bag has either a perfect matching or a matching where exactly one vertex* v *in the bag is unmatched and* v *is adjacent to some vertex in the matching. We call such bags* matched*. The* matched treewidth *of a graph* G*,* $\mathrm{mtw}(G)$*, is the minimum possible width among all possible matched tree decompositions of* G*.*

Matched treewidth is finite for all graphs. This is not trivial unlike treewidth because a single bag tree decomposition that contains all the vertices in the graph need not be matched.

We call a tree decomposition *reduced* if no bag B is a subset of another bag. Given any tree decomposition T, we can obtain a reduced tree decomposition T' such that the width of T' is at most the width of T. Moreover, all bags in T' are also bags in T. This implies that if T is matched, then T' is matched as well.

There are several equivalent characterizations for treewidth. Below, we state the ones that we use in this paper.

▶ **Definition 22.** *A* k-tree *is a graph formed by starting with a* $(k+1)$*-clique and repeatedly adding a vertex connected to exactly* k *vertices of the existing* $(k+1)$*-clique. A* partial *of a graph* G *is a graph obtained by deleting edges from* G*. The set of all graphs with treewidth at most* k *is exactly the class of partial* k*-trees.*

We can construct a *standard tree decomposition* T for any k-tree as follows: The root bag of T contains the vertices in the initial $(k+1)$-clique. Let S be a k-sized subset of this clique such that a new vertex v is added to the k-tree by connecting it to all vertices in S. Then, we add a sub-tree to T that will be a standard tree decomposition of the k-tree constructed using $S \cup \{v\}$ as the starting $(k+1)$-clique.

A *chordal completion* of a graph G is a super-graph G' of G such that G' has no induced cycles of length more than 3. A chordal completion that minimizes the size of the largest clique is called *minimum chordal completion.* The treewidth of a graph G is the size of the largest clique in its minimum chordal completion.

Two paths P_1 and P_2 from u to v are *internally disjoint* if and only if P_1 and P_2 do not have any common internal vertex.

A graph G is *2-connected* or *biconnected*, if for any $x \in V(G)$, $G - x$ is connected. Equivalently, for any two vertices in G, there are at least 2 internally disjoint paths in G.

▶ **Definition 23.** *A series-parallel graph is a triple* (G, s, t) *where* s *and* t *are vertices in* G*. This class is recursively defined as follows:*

- *An edge* $\{s, t\}$ *is a series-parallel graph.*
- *(series composition) If* (G_1, s_1, t_1) *and* (G_2, s_2, t_2) *are series-parallel graphs, then the graph obtained by identifiying* s_2 *with* t_1 *is also series-parallel.*
- *(parallel composition) If* (G_1, s_1, t_1) *and* (G_2, s_2, t_2) *are series-parallel graphs, then the graph obtained by identifiying* s_1 *with* s_2 *and* t_1 *with* t_2 *is also series-parallel.*

A graph has treewidth at most 2 *is if and only if all of its biconnected components are series-parallel graphs.*

▶ **Definition 24.** *A graph* G *is said to be a* minor *of a graph* G' *if* G *can be obtained from* G' *either by deleting edges/vertices, or by contracting the edges. (The operation of contraction merges two adjacent vertices* u *and* v *in the graph and removes the edge* (u, v)*.) If* G *is obtained from an induced subgraph of* G' *by contracting the edges, then it is said to be an* induced minor *of* G'*.*

A graph G is called G'-induced-minor-free (G'-minor-free) if G' is not an induced minor (resp. minor) of G.

An edge subdivision is an operation which deletes the edge (u, v) and adds a new vertex w and the edges (u, w) and (w, v). A graph G' obtained from G by a sequence of edge subdivisions is said to be a *subdivision* of G.

2.1 Representation of graphs

We assume the following time complexities for basic graph operations. Any representation that satisfies these is sufficient.

- Given u and v, it can be checked in $\widetilde{O}(1)$-time whether uv is an edge.
- Iterating over all the edges $xy \in E(H)$ ordered by x can be done in $\widetilde{O}(m)$-time, where m is the number of edges in H.

These requirements are satisfied by the following adjacency-list representation. To represent a graph H, we store a red-black tree T that contains non-isolated vertices of H where vertices are ordered according to their labels. Consider a node in this tree that corresponds to a vertex u. This node stores another red-black tree T_u that stores all neighbors of u in H. Now, to check whether uv is an edge or not, we perform a lookup for u in T followed by a lookup for v in T_u if u was found. We can iterate over all edges xy ordered by x by performing an inorder traversal of T where for each node u, we perform an inorder traversal of T_u.

If the pattern does not contain any isolated vertices, then we can ignore isolated vertices in the host graph as well. If the pattern is $G = G' + v$, where v is an isolated vertex and G' is any graph, then the number of homomorphisms from G to H is obtained by multiplying the number of homomorphisms from G' to H by n, where n is the number of vertices in H. This can be calculated by simply storing the number of vertices of H in the data structure.

3 Overview of proofs

Due to space constraints, we cannot include all the proofs in our paper. In this section, we summarize the main ideas involved in our proofs. We refer the reader to the full version for details.

3.1 Homomorphism counting

For any k-vertex pattern, the brute-force algorithm that iterates over all k-tuples of vertices in the host graph can be used to enumerate all homomorphisms from the pattern to the host in $O(n^k)$-time. Suppose the pattern has a perfect matching, then instead of iterating over k-tuples of vertices, we can iterate over all $k/2$-tuples of edges to enumerate all homomorphisms in $O(m^{k/2})$-time. This yields the maximum possible asymptotic improvement for sparse host graphs that does not improve the general algorithm.

The brute-force algorithm can be improved using dynamic programming over tree decompositions of the pattern graph [6]. Broadly, instead of iterating over all k-tuples of vertices, we only have to brute-force over tuples of vertices of size at most b, where b is the bag size in a tree decomposition, for each bag. This yields algorithms that run in $O(n^{t+1})$-time, where t is the treewidth of the pattern. The key observation that helps us to obtain improved algorithms for sparse graphs is that if the subgraph of the pattern induced by every bag in the tree decomposition contains a perfect matching (or a matching plus an, not disjoint, edge for bags with an odd number of vertices), then we can obtain a similar (to the previous paragraph) improvement to the dynamic programming algorithm. i.e., we can

turn the $O(n^{t+1})$-time dynamic programming algorithm to an $O(m^{\lceil (t+1)/2 \rceil})$-time dynamic programming algorithm. However, demanding that each bag contain a perfect matching results in a parameter that is more constrained than treewidth, the matched treewidth. Once matched treewidth is defined, the proof of correctness of the algorithm (See Proof of Theorem 6) is a straightforward generalization of the algorithm in [6].

The dynamic programming algorithm over treewidth requires polynomial space because an intermediate value may be reused any number of times. To reduce this space uses we can use divide-and-conquer algorithm instead of dynamic programming algorithm. The treedepth of the pattern governs the running-time of these algorithms. Broadly, these algorithms iterate over tuples of vertices of size at most d, where d is the length of a root-to-leaf path in the elimination tree. As before, if the subgraph of the pattern induced by the vertices in all paths in the elimination tree contains a perfect matching (or a matching plus an, not disjoint, edge for paths with an odd number of vertices), then, we can turn the $O(n^d)$-time algorithm to an $O(m^{d/2})$-time algorithm. Unlike the generalization for treewidth, this generalization is not as straight-forward as we also want to retain constant-space usage. This means that we require the perfect matching on each root-to-leaf path in the elimination tree occur from top-to-bottom. i.e., suppose a path v_1, v_2, v_3, v_4 is a root-to-leaf path in an elimination tree, then the matching has to be $\{v_1, v_2\}$ and $\{v_3, v_4\}$ and *not* $\{v_1, v_4\}$ and $\{v_2, v_3\}$. This allows the algorithm to recursively discover the vertices in the images of the homomorphism through edges in the host graph while moving from top to bottom in the elimination tree. Additionally, we also need the ability to iterate over all the edges of the host graph ordered by an end-point so that we need not use additional space to know whether or not we have finished processing a vertex.

3.2 Analysis of matched treewidth and matched treedepth

To obtain running-time improvements, we also prove that matched treewidth and matched treedepth are indeed close to treewidth and treedepth respectively for many interesting patterns. First, we prove that the worst-case is not too bad. In the proof of Theorem 3, we prove that half of matched treedepth is always strictly less than treedepth, yielding improvements even in the worst-case. The proof starts with an elimination tree and provides an algorithm that obtains a matched elimination tree by iteratively modifying it. For this proof, we introduce the notion of a connected elimination tree, an elimination tree where each vertex is connected to all of its subtrees. We show a lemma that may be of interest elsewhere, that we can assume wlog that the elimination tree is connected. A top-down iterative algorithm then converts the connected elimination tree into a matched elimination tree. Each iteration consists of two phases: The first phase connects (in the pattern) the current node to all its children, possibly violating connectivity; and the second phase re-establishes connectivity, possibly introducing a non-adjacent chlid (in the pattern) to the current node. By alternating between these two phases, we satisfy the property of matched elimination tree locally, at the current node and then proceed to deeper levels. The proof for the fact that half of matched treewidth is at most treewidth (See proof of Theorem 7) is similar. We start with a tree decomposition and provide an iterative, top-down algorithm that obtains a matched tree decomposition.

For obtaining improved algorithms for fixed patterns, we need some tools to determine when a pattern has small matched treewidth. Towards this end, we are able to completely characterize the matched treewidth of partial 2-trees by using forbidden induced minors. These proofs involve several different ideas specific to the (treewidth, matched treewidth) combination and therefore seems difficult to generalize.

We use contradiction method to prove both Theorem 10 and Theorem 11. The overall idea for the proof of both the theorems are similar. That is, for the backward direction, we assume the existence of a partial 2-tree G, that has the forbidden graph an induced minor. From a matched tree decomposition of G, we construct a matched tree decomposition of the forbidden graph with the same width. This gives a contradiction. For the other direction, we suppose for the contradiction that G is a counterexample with minimum number of vertices. We show the existence of chordal completion of G, whose standard tree decomposition is also a matched tree decomposition of G. This gives a contradiction. More details are discussed below.

Proof Idea of Theorem 10. We prove both directions using proof by contradiction. For the backward direction, we suppose for contradiction that there is a graph G which is a partial 2-tree such that $mtw(G) = 2$ and G has a C_5-induced-minor. We consider a matched tree decomposition T of G with width 2. Note that a C_5 can be obtained from G by deleting some vertices and contracting some edges. For all the deleted vertices we delete it from all the bags of T. Similarly, whenever an edge uv is contracted, we replace u and v in all the bags of T by one of u or v. Note that the obtained tree decomposition T' might not be matched. Since the width of T is two, any bag of T' is of size at most 3. Again any bag of size 3 in T' form a P_3. If a bag B in T' has size 1, then we can remove B by directly connecting all children of B to the parent of B. We show that a bag B of size two in T' does not contain two non-adjacent vertices in C_5. Hence, T' is a matched tree-decomposition of C_5, of width at most 2. This is a contradiction.

For the other direction, we suppose for contradiction that G is a counter example with minimum number of vertices. We prove some properties of G. First, we show that G does not have any cut-vertex. Since G is a partial 2-tree, we also obtain a 2-tree G' that is a super-graph of G and has the same vertex set as G. The standard tree decomposition for G' is matched. Since G is a counter-example, this tree decomposition should not be matched for G. But this means that there is a bag $\{u, v, w\}$ in the tree decomposition where it forms a triangle in G' but (say) v is not adjacent to u and w in G. We now show, using the fact that partial 2-trees are K_4-minor free, that any two adjacent vertices in G' are at a distance of at most two in G. Therefore, we obtain an induced cycle of length at least 5 in G using the vertices u, v, w, and the length-two paths between v and u, v and w, and possibly u and w. ◀

Proof Idea of Theorem 11. For the backward direction, we suppose for the contradiction that, there exists a partial 2-tree that has matched treewidth 3 and has an X-induced-minor. So, X can be obtained from an induced subgraph X' of G by contracting some edges. Let $\mathcal{U}$ be a matched tree decomposition of G and T' be the tree decomposition of X' obtained from $\mathcal{U}$ by deleting vertices in the set $V(G) \setminus V(X')$ from all the bags. Since X is obtained from X' by contracting some edges, there exist three mutually vertex disjoint paths p_0, p_2, p_4 such that contraction of all the edges in p_i gives u_i, for all the $i \in \{0, 2, 4\}$. Suppose X'' be the graph obtained from X' by contracting all the edges in the paths p_0, p_2, p_4 and T'' be the tree decomposition of X'', obtained from T' by replacing any vertex of p_i with u_i (for all the $i \in \{0, 2, 4\}$) in all the bags of T'.

Note that X can also be obtained from X'' by contracting some edges. Hence, there exists internally vertex-disjoints paths Q_{i+1}, Q'_{i+1} from u_i to u_{i+2} in X'', where the index $i \in \{0, 2, 4\}$ is in modulo 6. We define a map $f : V(X'') \mapsto V(X)$ such that $f(u_i) = u_i$ and maps all the internal vertices in the path Q_{i+1} to u_{i+1}, for all $i \in \{0, 2, 4\}$. Let T be the tree decomposition of X from T'' by applying f to all vertices in all bags. Note that there is no size 0 bags in T. We can remove size 1 bags from T using standard techniques. Note that if

uv is an edge in X'', then $f(u)f(v)$ is an edge in X, whenever $f(u) \neq f(v)$. Hence, and bag of size 4 in T is matched, since $\mathcal{U}$ is an matched tree decomposition of G. We show that no bag of size 3 in T is an independent set in X. Furthermore, by using these facts we argue that some modification of T can give a matched tree decomposition of X that has width at most that of T. This gives a contradiction.

For the other direction, we use two facts about partial 2-trees. The first fact is the well-known series-parallel characterization of partial 2-trees and the second one is the following property. Let G be a partial 2-tree and u, v be two vertices in G. Then uv is an edge in any chordal completion of G, whenever there are three internally disjoints paths from u to v in G. We show that for any X-induced-minor-free connected partial 2-tree G, there exists an minimum chordal completion $\tilde{G}$ such that no independent set of size 3 in G is a clique in $\tilde{G}$. Suppose for the contradiction that G is a minimum counterexample for the above statement. Since G is a minimum counterexample, it is not an edge. Furthermore, we show that G is not series composition of two smaller graphs.

Let G with source vertex s and terminal vertex t be the parallel composition of G_1 and G_2. We break our proof into two explicit cases depending on whether G_1 or G_2 is an edge. Before that we prove the following properties that we use in both the case to achieve the contradiction. Then we proved the following claim, that is used in both the cases to achieve a contradiction. If G_1 is a series compositions of two or more G'_i for $1 \leqslant i \leqslant m$. Then, for all $1 \leqslant i \leqslant m$, if G'_i is an edge, then s'_i is not adjacent to t'_i in G'_i.

In the first case we assume that, G_2 is an edge. So, s_1t_1 is not an edge in G_1, as otherwise $G_1 = G$. We showed that G_1 is a parallel composition of two smaller graph then G is not a counterexample. This gives a contradiction. Hence, (G_1, s_1, t_1) is a series composition of smaller graphs, say $(G'_1, s'_1, t'_1), \ldots, (G'_m, s'_m, t'_m)$, for some $m \geqslant 2$. By using the above claim about G_1 and G being minimum counterexample, we showed that there exists $1 \leqslant i \leqslant m$ such that s'_i is not adjacent to t'_j. Furthermore, we prove the following two claims. The first claim says the non-existence of $1 \leqslant j(\neq i) < k(\neq i) \leqslant m$, such that $s'_jt'_j, s'_kt'_k$ are non-edges. Then we prove the second claim, by using the property of partial 2-trees (Let G be a partial 2-tree and u, v be two vertices in G. Then uv is an edge in any chordal completion of G, whenever there are three internally disjoints paths from u to v in G). It gives the existence of an minimum chordal completion of G'_i in which no clique of size 3 is an in dependent set in G'_i and $s'_it'_i$ is an edge. By using these two claims we achieve a contradiction.

In the other case, we have both G_1 and G_2 are non-edges. First, we show that either G_1 or G_2 is not a parallel composition of smaller graphs. Then we decompose G_1 and G_2 into series composed graphs, until we cannot do further. G(with different source or terminal vertex) can be represented as parallel composition of g'_1 and an edge. Now, we follow previous case to achieve a contradiction. ◀

3.3 Algorithms for small patterns

The proofs of improved algorithms for small patterns such as paths and cycles use both homomorphism counting algorithms (Theorem 6 and Theorem 1) and characterizations. *Subgraph counting* algorithms for patterns invoke homomorphism counting on all homomorphic images of the pattern, called the spasm of the pattern. Even for small graphs such as P_{10}, the spasm is a very large set. However, using our characterization theorems for matched treewidth of partial 2-trees, we can conclude that all partial 2-trees in $\mathsf{Spasm}(P_{10})$ has matched treewidth at most 4 immediately yielding $O(m^2)$-time algorithms for counting homomorphisms. We are left with an analysis of only treewidth 3 graphs in the spasm of

which there are only few. However, for constant-space algorithms, we had to analyze a large number of graphs in $\mathsf{Spasm}(C_{11})$ and $\mathsf{Spasm}(C_{10})$. If treedepth is at most 4, then matched treedepth is at most 6 and we immediately obtain an $O(m^3)$-time, constant-space algorithm for counting homomorphisms. For graphs with treedepth 5 or higher, we have to do this analysis manually.

Finally, we consider induced subgraph detection of the patterns C_6 and $\overline{P_k}$. For induced subgraph detection, we have to build arithmetic circuits for homomorphism polynomials of graphs that are related to the pattern. These ideas were used by Bläser, Komarath, and Sreenivasaiah [2], for example to obtain $O(n^4)$-time algorithm for induced C_6 detection and $O(n^{k-2})$-time algorithm for induced $\overline{P_k}$ detection. The overall idea is to express the induced subgraph isomorphism polynomial of any k-vertex pattern G as a linear combination of subgraph isomorphism polynomials of all k-vertex super-graphs G' of G. Then, by choosing a suitable prime p, and by doing arithmetic modulo p, we can eliminate the harder super-graphs G' from the linear combination. Finally, we can check whether the remaining subgraph isomorphism exist or not by checking whether multilinear terms exist or not in an appropriately scaled homomorphism polynomial (See [14]). In adapting these algorithms to sparse graphs, the major challenge is to ensure that that the construction of these appropriately scaled homomorphism polynomials is possible in the required time. First of all, the scaling is done not by multiplying the final result by a constant (since the scaling quite often has to be done by a multiple of p, which is not possible when doing arithmetic modulo p), but by avoiding certain monomials from the polynomial altogether. For this, we need to ensure that the matched tree decompositions we consider are restricted even further. Indeed, we show how to do this for all the relevant super-graphs of C_6 in $\widetilde{O}(m^2)$-time and for $\overline{P_k}$ in $\widetilde{O}(m^{\lceil (k-2)/2 \rceil})$-time to obtain the maximal possible improvement.

4 Conclusion

Due to space constraints, we conclude our extended abstract here. The remaining sections that contain all theorems and proofs are in the appendix of this paper.

References

1. N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, March 1997. doi:10.1007/BF02523189.
2. Markus Bläser, Balagopal Komarath, and Karteek Sreenivasaiah. Graph pattern polynomials. In Sumit Ganguly and Paritosh K. Pandya, editors, *38th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2018, December 11-13, 2018, Ahmedabad, India*, volume 122 of *LIPIcs*, pages 18:1–18:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.FSTTCS.2018.18.
3. Marco Bressan and Marc Roth. Exact and approximate pattern counting in degenerate graphs: New algorithms, hardness results, and complexity dichotomies. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 276–285, 2022. doi:10.1109/FOCS52979.2021.00036.
4. 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*, STOC 2017, pages 210–223, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3055399.3055502.
5. Radu Curticapean and Dániel Marx. Complexity of counting subgraphs: Only the boundedness of the vertex-cover number counts. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 130–139. IEEE Computer Society, 2014. doi:10.1109/FOCS.2014.22.

6. Josep Díaz, Maria J. Serna, and Dimitrios M. Thilikos. Counting h-colorings of partial k-trees. In *Proceedings of the 7th Annual International Conference on Computing and Combinatorics*, COCOON '01, pages 298–307, Berlin, Heidelberg, 2001. Springer-Verlag.
7. Julian Dörfler, Marc Roth, Johannes Schmitt, and Philip Wellnitz. Counting induced subgraphs: An algebraic approach to #w[1]-hardness. *Algorithmica*, 84(2):379–404, 2022. `doi:10.1007/s00453-021-00894-9`.
8. Friedrich Eisenbrand and Fabrizio Grandoni. On the complexity of fixed parameter clique and dominating set. *Theoretical Computer Science*, 326:57–67, October 2004. `doi:10.1016/j.tcs.2004.05.009`.
9. Peter Floderus, Mirosław Kowaluk, Andrzej Lingas, and Eva-Marta Lundell. Induced subgraph isomorphism: Are some patterns substantially easier than others? In Joachim Gudmundsson, Julián Mestre, and Taso Viglas, editors, *Computing and Combinatorics*, pages 37–48, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
10. Jacob Focke and Marc Roth. Counting small induced subgraphs with hereditary properties. In Stefano Leonardi and Anupam Gupta, editors, *STOC '22: 54th Annual ACM SIGACT Symposium on Theory of Computing, Rome, Italy, June 20–24, 2022*, pages 1543–1551. ACM, 2022. `doi:10.1145/3519935.3520008`.
11. Fedor V. Fomin, Daniel Lokshtanov, Venkatesh Raman, Saket Saurabh, and B. V. Raghavendra Rao. Faster algorithms for finding and counting subgraphs. *J. Comput. Syst. Sci.*, 78(3):698–706, 2012. `doi:10.1016/j.jcss.2011.10.001`.
12. Ton Kloks, Dieter Kratsch, and Haiko Müller. Finding and counting small induced subgraphs efficiently. *Inf. Process. Lett.*, 74(3-4):115–121, 2000. `doi:10.1016/S0020-0190(00)00047-8`.
13. Balagopal Komarath, Anurag Pandey, and C. S. Rahul. Graph homomorphism polynomials: Algorithms and complexity. *CoRR*, abs/2011.04778, 2020. `arXiv:2011.04778`.
14. Ioannis Koutis. Faster algebraic algorithms for path and packing problems. In *International Colloquium on Automata, Languages, and Programming*, pages 575–586. Springer, 2008.
15. Miroslaw Kowaluk, Andrzej Lingas, and Eva-Marta Lundell. Counting and detecting small subgraphs via equations. *SIAM J. Discret. Math.*, 27:892–909, 2013.
16. Jaroslav Nešetřil and Svatopluk Poljak. On the complexity of the subgraph problem. *Commentationes Mathematicae Universitatis Carolinae*, 026(2):415–419, 1985. URL: `http://eudml.org/doc/17394`.
17. Marc Roth, Johannes Schmitt, and Philip Wellnitz. Detecting and Counting Small Subgraphs, and Evaluating a Parameterized Tutte Polynomial: Lower Bounds via Toroidal Grids and Cayley Graph Expanders. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming (ICALP 2021)*, volume 198 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 108:1–108:16, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.ICALP.2021.108`.
18. Virginia Vassilevska. *Efficient Algorithms for Path Problems in Weighted Graphs.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA 15213, 2008.
19. Douglas B. West. *Introduction to Graph Theory.* Prentice Hall, September 2000.
20. Ryan Williams. Maximum two-satisfiability. *Encyclopedia of Algorithms*, pages 507–510, 2008. `doi:10.1007/978-0-387-30162-4_227`.
21. Ryan Williams. Finding paths of length k in o*(2k) time. *Inf. Process. Lett.*, 109(6):315–318, February 2009. `doi:10.1016/j.ipl.2008.11.004`.
22. Virginia Vassilevska Williams, Joshua R. Wang, Ryan Williams, and Huacheng Yu. Finding four-node subgraphs in triangle time. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '15, pages 1671–1680, USA, 2015. Society for Industrial and Applied Mathematics.

The following appendices contain the remaining sections of the full paper.

A Matched treedepth

In this section, we introduce algorithms that count homomorphisms and subgraphs efficiently in constant space on sparse host graphs. The central theorem in this section is given below.

▶ **Theorem 1.** *Let G be a graph with $\mathsf{mtd}(G) = d$, then given an m-edge graph H as input, we can count the number of homomorphisms from G to H in $\widetilde{O}(m^{\lceil d/2 \rceil})$-time and constant space.*

Proof. The algorithm is given in Algorithm 1. We can compute the result needed by calling COUNT-HOM-MTD(G, E, r, H, ϕ), where E is an elimination tree for G of depth d, r is the root vertex in E, and ϕ is the empty homomorphism. For simplicity of presentation, we assume that each root-to-leaf path in E has an even number of vertices. Odd number of vertices in a root-to-leaf path is handled similarly.

We assume that the host graph H is represented using a symmetric adjacency list representation. This is mainly to ensure that we can iterate over all edges xy in H ordered by x in Line 3.

First, we prove that the algorithm is correct. We claim that the call COUNT-HOM-MTD (G, E, v, H, σ) where the parameters are as specified in the algorithm returns the number of homomorphisms from G_v to H that extends σ. This is proved by an induction on the height of v in E. Since v is a top node, the base case is when the height is 2. In this case, G_v is a star graph and it is easy to see that the algorithm works. We now prove the inductive case. The variable t computes the final answer. Denote by $s_{u,x}$ for vertex x in H the number of homomorphisms from G_u to H that extends $\tau = \sigma \cup \{v \mapsto x\}$. Notice that since $uv \in E(G)$, to extend τ, the vertex u must be mapped to some y such that $xy \in E(H)$. Therefore, iterating over all such y is sufficient. Notice that we can compute t as $\sum_\tau \prod_u s_{u,x}$. However, this would need storing $|V(G)||V(H)|$ variables. By iterating over the edges of H ordered by x, we can afford to reuse a single s_u for different x instead of keeping a separate $s_{u,x}$ for each x. Now, we show that s_u correctly computes $s_{u,x}$ once the main loop finishes with an x. By the inductive hypothesis, the variable c_w is the number of homomorphisms from G_w to H that extends $\sigma' = \sigma \cup \{v \mapsto x, u \mapsto y\}$. Therefore, we have $s_{u,x} = \sum_y \prod_w c_w$. Line 12 correctly computes this into s_u. Line 15 correctly updates t once an x is finished. Finally, we reset s_u to 0 before processing the next x.

Now, we prove the running-time and space usage of the algorithm. Notice that the depth of the recursion is bounded by the depth of the elimination tree E and each level of recursion stores only constantly many variables. Therefore, the space usage is constant. The main loop in Line 3 runs for $2m$ iterations. The inner-loops only have a constant number of iterations. Therefore, the recursive calls are made only $O(m)$ times. We process two levels of the elimination tree in a level. Therefore, the total running-time is given by $t(d) \leqslant O(m)t(d-2) + \widetilde{O}(1) = \widetilde{O}(m^{d/2})$.

Algorithm 1 COUNT-HOM-MTD(G, E, v, H, σ).

```
Require: G - The pattern graph
Require: E - Matched elimination tree for G
Require: v - A top vertex in E
Require: H - The host graph
Require: σ - A partial homomorphism from the ancestors of v to H
  t ← 0
  s_u ← 0 for all children u of v
  for all edges xy ∈ E(H) ordered by x do
    for all children u of v in E do
      σ' ← σ ∪ {v ↦ x, u ↦ y}
      if σ' is an invalid homomorphism then
        continue
      end if
      for all children w of u do
        c_w ← COUNT-HOM-MTD(G, E, w, H, σ')
      end for
      s_u ← s_u + ∏_w c_w
    end for
    if xy is the last edge on x then
      t ← t + ∏_u s_u
      s_u ← 0 for all u
    end if
  end for
  return t
```

◀

▶ **Definition 25.** *An elimination tree* T *is* connected *if for every node* u *in* T *and a child* v *of* u *in* T, u *is adjacent to some node in* T_v

Now, we show that connected elimination trees are optimal.

▶ **Lemma 26.** *Every connected graph* G *has a connected elimination tree of depth* $\mathsf{td}(G)$.

Proof. We show how to construct a connected elimination tree T' from an elimination tree T without increasing its depth. Let $T' = T$ initially. Suppose there exists some node u in T' that violates the property (If not, we are done). Then, there exists a child v of u in T' such that there is no edge in G between u and any node in T'_v. Let w be a node in T'_v such that w is adjacent to some proper ancestor x of u. Such a w must exist because G is connected. Now, in T', remove the subtree T'_v and make it a subtree of the node x. Repeat this process until all nodes in T' satisfy the required property. This process must terminate since at each step, we reduce the number of nodes violating the property by at least one. This process cannot increase the depth of T' because the only modification is to move a subtree upwards to be a subtree of a proper ancestor of its parent. ◀

▶ **Theorem 3.** *For any graph* G, $\mathsf{mtd}(G) \leqslant 2 \cdot \mathsf{td}(G) - 2$.

Proof. We start with a connected elimination tree T with depth d of G and show how to construct a matched elimination tree of G from T. We use induction on d. For $d = 2$, the tree is already matched and has depth $2 = 2 \cdot 2 - 2$.

Our construction is iterative and top-down. Each iteration ensures that the current top-most node in the elimination tree is adjacent, in the graph G, to all its children and that the elimination tree is connected.

Each iteration consists of two phases iteratively executed until the desired property is satisfied. The first phase ensures that the root node r is adjacent in G to all its children in T. If r has a child v that is not adjacent in G to r, then since T is connected, there is some node w in T_v such that $rw \in E(G)$. Then, we make w a child of r in T, delete w from T_v, and make all children of w children of parent of w. The resulting tree is an elimination tree of depth at most $d+1$. After this phase, the root node is adjacent in G to all its children, the tree's depth has increased by at most one. However, it may not be connected.

In the second phase, we use the construction of Lemma 26 to make the tree connected without increasing the depth. Observe that the construction will keep the existing children of root as is and may add new chlidren to r that are not adjacent to r in G. Suppose a new node u was added as a new child to r in this second phase. The height of subtree rooted at u is at most $d-1$. Therefore, the tree (r, T_u) obtained by attaching u to r has height at most d. We can now execute phase 1 on all the trees (r, T_u) for all such u without increasing depth beyond $d+1$. This process must eventually terminate as we add at least one new child to the root every time.

At the end of the iteration, consider a grandchild x of r. If it is a leaf, since the tree is connected, x must be adjacent in G to its parent in T and we are done. Otherwise, the subtree T_x is a tree of depth at least 2 and at most $d-1$ that is connected. So by the induction hypothesis, we obtain that T_x is a matched elimination tree of depth at most $2(d-1)-2$. This means that the original tree is converted to a matched elimination tree of depth at most $2+2(d-1)-2 = 2d-2$ as required. ◀

▶ **Theorem 4.** *Let G be a graph and G' is a connected, induced subgraph of G, then:*

1. $\mathsf{mtd}(G') \leqslant \mathsf{mtd}(G)$ *if* $\mathsf{mtd}(G)$ *is even.*
2. $\mathsf{mtd}(G') \leqslant \mathsf{mtd}(G)+1$ *if* $\mathsf{mtd}(G)$ *is odd.*

Proof. We start with a matched elimination tree T of even (The odd case is similar) depth d for G and construct a matched elimination tree for G'. First, we delete all nodes in the elimination tree that are in G but not in G'. If a node u in T has parent v that was deleted, then we make u a child of the closest ancestor of v that is still in T. The final forest thus obtained is a tree T' because G' is connected. We assume without loss of generality that T' is a connected elimination tree.

Suppose r' is the root of T'. We will now modify T' into a matched elimination tree. We will first analyze paths in T' that correspond to even length root-to-leaf paths in T. If T' is not matched on this path, then there exists a node u closest to r' such that u is not connected in G' to a child v in T'. This is only possible if u was either matched to a child w of u in T or its parent w in T and w is not in G'. Therefore, we have $\mathsf{dist}_{T'}(r', u) + \mathsf{depth}(T'_v) < \mathsf{depth}(T)$ and this means we can afford to increase the length of any path that passes through edge uv by 1. Since T' is connected, we can now apply the transformation in the proof of Theorem 3 to match u with one of its descendants in T'_v. The depth is still at most d because this transformation increases the depth by at most 1 In effect, the increase in depth in this branch of the tree by pulling up a descendant is compensated by the fact that the unmatched vertex was introduced by deleting a vertex in this branch.

We can iteratively apply the above construction to make T' a matched elimination tree while keeping T' connected. However, applying the above transformation may introduce a node u in T' that is not matched to a child v *because* v's parent w was pulled up in the tree to

match with some other vertex. Such u also satisfy the inequality $\mathsf{dist}_{T'}(r', u) + \mathsf{depth}(T'_v) < \mathsf{depth}(T)$. Why? Any path in the tree T' that passed through both u and v earlier had to pass through w. But, the fact that w was pulled up implies that these paths had length strictly less than $\mathsf{depth}(T)$ by the argument in the previous paragraph. And shifting w to a position earlier in the path cannot increase its length.

For root-to-leaf paths in T of odd length, there might be a matched P_3 on vertices uvw such that v is not in G'. In this case too, by the same argument, the transformations increase the depth to at most $d + 1$ (In this case, we may have to pull up two distinct vertices for matching u and w.). When d is even, increasing the length of such paths by 1 does not increase the depth of the tree. When d is odd, increasing the length of such paths by 1 increases the depth by at most 1. ◀

B Matched treewidth

▶ **Theorem 12.** *Given an m-edge graph H as input, we can count the number of P_k subgraphs, where $k \leqslant 10$, in $\widetilde{O}(m^2)$-time.*

Proof. There are more than 300 graphs in $\mathsf{Spasm}(P_{10})$. We have verified that all of them have mtw at most 3. To minimize the work, we can filter out all graphs in the spasm that has $\mathsf{tw}(G') = 1$ or $\mathsf{tw}(G') = 2$ and G' is X-induced-minor-free. Observe that since X has 9 vertices 12 edges, it cannot be an induced minor in any of the graphs in $\mathsf{Spasm}(P_{10})$. Also, none of the forbidden minors for treewidth 4 can appear in $\mathsf{Spasm}(P_{10})$. Therefore, we only need to analyze graphs of treewidth 3 in $\mathsf{Spasm}(P_{10})$. There are only 18 such graphs. They are listed in a pdf file in the repository associated with this paper (`https://github.com/anonymous1203/Spasm`).

Since $\mathsf{Spasm}(P_k) \subseteq \mathsf{Spasm}(P_{k+1})$ for all $k \geqslant 2$, we can make the same claim for all paths on fewer than 10 vertices. We now use

$$\mathsf{Sub}_G[H] = \sum_{G'} \alpha_{G'} \mathsf{Hom}_{G'}[H] \tag{1}$$

to compute the result. ◀