Fast Local Subgraph Counting

Qiyan Li
The Chinese University of Hong Kong
Hong Kong, China
qyli@se.cuhk.edu.hk

Jeffrey Xu Yu
The Chinese University of Hong Kong
Hong Kong, China
yu@se.cuhk.edu.hk

ABSTRACT

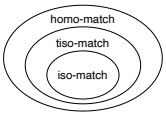
We study local subgraph counting queries, $Q = (p, o)$, to count how many times a given k-node pattern graph p appears around every node v in a data graph G when the given center node o in p maps to v. Such local subgraph counting becomes important in GNNs (Graph Neural Networks), where incorporating such counts for every node in G into the GNN architecture enhances the model's ability to capture complex relationships within the graph G. It is challenging to count by subgraph isomorphism, which is known to be NP-hard. In this paper, we propose a novel approach by tree-decomposition-based counting. For a complex pattern graph p in Q, we find its best tree decomposition T, where a node in T represents a subgraph of p, and a node in p may appear in multiple nodes in T. Let $p(T)$ be the pattern represented by T. Our approach is to count $p(T)$ by homomorphism with a constraint to count the subgraph in every tree node by subgraph isomorphism. We apply symmetry-breaking rules to reduce the cost of counting by subgraph isomorphism for every node in T, and we develop a new multi-join algorithm to compute such counts. We confirm that our approach on a single machine using a single core can outperform the others significantly.

PVLDB Reference Format:
Qiyan Li and Jeffrey Xu Yu. Fast Local Subgraph Counting. PVLDB, 17(8): XXX-XXX, 2024.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/magic62442/subgraph-counting.

1 INTRODUCTION

A graph is a complex structure that has been widely used to support various real-world applications such as social network analysis. Given a large data graph G, a local subgraph counting query $Q = (p, o)$ is to count how many times a given k-node pattern graph p that appears around every node v in G when the center node o in p maps to v. Local subgraph counting has many applications in data mining, such as community detection, graph clustering, network comparison and alignment, anomaly detection, and detecting strong ties in social networks [80, 101], and becomes important in GNNs (Graph Neural Networks), as it provides a large number of topological features. For example, when $k = 5, 6$ and 7, there are 58,407 and 4,306 such pattern graphs respectively. Such local subgraph counts infuse GNNs with higher-order graph structural information. This is useful in tasks where the presence of specific subgraph patterns is indicative of certain properties or labels. In [9], Barceló et al. propose local graph parameter enabled GNNs, and in [76], Qian et al. further study subgraph-enhanced GNNs.



(a) Subgraph counts

	WS	BY	CA	SG	WW
$\sum_v \lvert \mathrm{HOM}_{p,o}(v) \rvert$	86.7	1,767	3,949	1,839	23,936
$\sum_v \lvert \mathrm{tISO}_{p,o,T}(v) \rvert$	78.3	1,620	3,817	1,584	16,461
$\sum_v \lvert \mathrm{ISO}_{p,o}(v) \rvert$	70.0	1,491	3,547	1,426	14,899
#enumerated match	0.6	6.7	8.1	12.9	232

(b) number of matches($\times 10^8$) of the pattern p in Fig. 4

Figure 1: Comparing three types of matches

Local subgraph counting is challenging, particularly when it is to count by subgraph isomorphism, which is known to be NP-hard. To compute local subgraph counting queries, there are enumeration-based [72, 96], matrix-based [23, 41, 42, 63–65], and decomposition-based [73, 101] approaches. A survey can be found in [80]. Among the three categories, the state-of-the-art is the decomposition approach (e.g., EVOKE [73] and DISC [101]), which decomposes p into smaller pattern graphs to count. Here, EVOKE can only support pattern graphs up to 5 nodes [73], whereas DISC is an approach that can handle any k-node pattern graphs. In brief, DISC is to compute local subgraph counting queries under subgraph isomorphism by homomorphism counting [5, 13, 20, 22, 25, 28, 34, 40]. It computes a query by eliminating homomorphism counts for those that are not subgraph matches from the total homomorphism count for any possible homomorphism matches [101]. As shown in Fig. 1(a), to count by subgraph isomorphism or iso-match (the innermost circle), it counts by homomorphism or homo-matches (the outermost circle) and substrates the counts of the homo-matches between the innermost and outermost circles. The hardness of counting by homomorphism is the same as counting by subgraph isomorphism in general. But, homomorphism can be done efficiently as there are less constraints to check. However, DISC is inefficient in practice. It can only process some selected 6-node pattern graphs in a batch [101]. To the best of our knowledge, there is no reported result that can compute all 407 6-node pattern graphs in a batch together over real large graphs in a reasonable time.

In this paper, we propose a new decomposition approach, called *tree-decomposition-based counting*, to compute local subgraph counting queries for any k-node pattern graph p in a novel way. For a complex pattern p, we find its tree-shaped pattern, T, by tree decomposition [32], where a node in T represents a subgraph of p, and a node in p may appear in multiple nodes in T. By tree-decomposition-based counting, we count the entire tree T by homomorphism counting, where every node in p that appears in different nodes in T must map to the same node in the data graph G, and that every subgraph p' of p represented by a node in T is counted by subgraph isomorphism. We call it tiso-match and illustrate it

Proceedings of the VLDB Endowment, Vol. 17, No. 8 ISSN 2150-8097.
doi:XX.XX/XXX.XX

by the middle circle in Fig. 1(a). We subtract the counts of the tiso-matches between the innermost and the middle circle. To show the performance achievement, we show the number of matches for a local subgraph counting query $Q = (p, o)$, where p is a 6-node pattern p in Fig. 4 and $o = u_1$ in p, using five datasets (Table 2) in Fig. 1(b). In Fig. 1(b), the first three rows are the total number of homo-matches, tiso-matches, and iso-matches, to be enumerated for the pattern p. We get such numbers using an enumeration-based approach. The number of enumerations by tiso-matches is significantly less than that by homo-matches since tiso-matches have more constraints than homo-matches. The actual number of enumerations by our approach is given in the last row in Fig. 1(b). We reduce the unnecessary enumerations significantly since we only enumerate subgraphs represented by tree nodes instead of p, and we apply symmetry-breaking rules to further reduce the enumeration of these subgraphs.

Main Contributions: First, we propose a novel tree decomposition-based counting and prove its correctness. Second, we explore automorphism orbits (structurally equivalent nodes) with symmetry-breaking rules to reduce the cost of finding iso-matches. Symmetry-breaking is an efficient technique for enumerating the iso-matches of an entire pattern graph. Different from existing works, we give a solution that can apply symmetry-breaking rules for tree nodes in a tree decomposition to compute aggregations. In addition, we propose an optimization technique to further reduce the cost. Third, we propose a new multi-join algorithm to compute each tree. Fourth, we have implemented our approach on a single machine, and we confirm that our approach SCOPE can outperform the state-of-the-art approach DISC significantly. We can compute the batch of all 407 6-node queries over real large graphs.

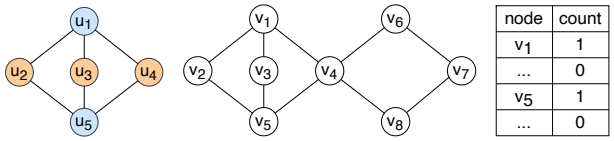
Organizations. We give preliminaries and the problem statement in Section 2, and discuss the existing homomorphism-based approach in Section 3. We propose a new decomposition-based approach in Section 4, discuss the new counting in Section 5, and introduce a new multi-join algorithm in Section 6. We discuss related works in Section 7, and confirm the efficiency of our approach in Section 8. We conclude our work in Section 9.

2 PRELIMINARIES

Following the notations in [101], we model a simple undirected graph as $G = (V, E)$, where V and E are the sets of nodes and edges in G, respectively. A graph $G' = (V', E')$ is a subgraph of G if $V' \subseteq V$ and $E' \subseteq E$, and is an induced subgraph of G if E' contains all the edges in G such that both endpoints belonging to V'. We call a graph a k-node graph (or k-graph) if it has k nodes. The number of nodes and the number of edges are denoted as $n = |V|$ and $m = |E|$.

Homomorphism & Subgraph Isomorphism: Let $G = (V, E)$ be a data graph and $p = (V_p, E_p)$ be a pattern graph. A function $f : V_p \mapsto V$ is called a homomorphism of p if for each edge $(u, u') \in E_p$, we have $(f(u), f(u')) \in E$, and a homomorphism f of p is called a subgraph isomorphism of p if f is injective. A subgraph $G_f = (V_f, E_f)$ in a data graph G is an *iso-match* (*hom-match*) of a pattern graph p if f is a subgraph isomorphism (homomorphism).

Automorphism Orbit: For a given graph $G = (V, E)$, an automorphism is a bijective function $\gamma : V \mapsto V$ such that $(v, v') \in E$ iff $(\gamma(v), \gamma(v')) \in E$. Here, an automorphism γ maps G to itself in a structure-preserving manner. The set of automorphisms, denoted as $\mathsf{Aut}(G)$, forms a group called the automorphism group of G. With $\mathsf{Aut}(G)$, two nodes $v, v' \in V$ are in an equivalence relation iff there exists an automorphism γ such that $\gamma(v) = v'$ (or $v \rightarrow v'$) for $\gamma \in \mathsf{Aut}(G)$. Such equivalence relations partition nodes into equivalence classes, where an equivalence class is called an automorphism orbit, and is denoted as ϑ. Below, we use o to denote a representative node in ϑ, which we call an orbit.



node	count
v_1	1
...	0
v_5	1
...	0

(a) pattern graph p (b) data graph G (c) SubG count

Figure 2: An Example of $Q = (p, o)$ for $o = u_1$

Problem Statement: In this work, we study local subgraph counting queries by subgraph isomorphism. Here, a local subgraph counting query Q is defined as $Q = (p, o)$, where p is a connected pattern graph and o is an orbit of p. The local subgraph count for a specific node v in a data graph G, denoted as $|\mathsf{SubG}_{p,o}(v)|$, is the count of all iso-matches of p that match v to o such that $f(o) = v$, where it only counts one for the iso-matches of p that are induced from the same edges. We compute $|\mathsf{SubG}_{p,o}(v)|$ for every node v in G.

It is known that $|\mathsf{SubG}_{p,o}(v)|$ can be processed by *local subgraph isomorphism counting* ($|\mathsf{ISO}_{p,o}(v)|$) or *local homomorphism counting* ($|\mathsf{HOM}_{p,o}(v)|$), where $\mathsf{ISO}_{p,o}(v)$ and $\mathsf{HOM}_{p,o}(v) = \{f \mid f$ are subgraph isomorphism and homomorphism of p with $f(o) = v\}$, respectively. It is important to mention that $|\mathsf{SubG}_{p,o}(v)| = |\mathsf{ISO}_{p,o}(v)| \cdot |\vartheta|/|\mathsf{Aut}(p)|$, as shown in [101]. We explain it in Example 2.1.

Example 2.1: Fig. 2 shows an example with a pattern graph $p = (V_p, E_p)$, a data graph $G = (V, E)$, and the $\mathsf{SubG}_{p,o}(v_i)$ for every v_i in G for a given subgraph counting query $Q = (p, o)$, where $o = u_1$. The following two mappings, f_i and f_j from V_p to V, $f_i = \{u_1 \rightarrow v_1, u_2 \rightarrow v_2, u_3 \rightarrow v_3, u_4 \rightarrow v_4, u_5 \rightarrow v_5\}$ and $f_j = \{u_1 \rightarrow v_1, u_2 \rightarrow v_2, u_3 \rightarrow v_3, u_4 \rightarrow v_4, u_5 \rightarrow v_1\}$, are homomorphisms, and only f_i is subgraph isomorphism, as both u_1 and u_5 in f_j map to the same node v_1 in G. With f_i and f_j, there is an iso-match, G_{f_i}, and a homo-match, G_{f_j}, in G.

The automorphism group of p, $\mathsf{Aut}(p)$, is in size of $|\mathsf{Aut}(p)| = 12$. As an example, one automorphism γ is $(u_1, u_5)(u_2, u_3, u_4)$ based on the representation of the product of disjoint cycles regarding permutation. For example, consider the disjoint cycle of (u_2, u_3, u_4), it refers to $u_2 \rightarrow u_3$, $u_3 \rightarrow u_4$, and $u_4 \rightarrow u_2$. Some other examples are $(u_1)(u_2)(u_3)(u_4)(u_5)$ and $(u_1, u_5)(u_2)(u_3)(u_4)$, where the former is an identity automorphism (or trivial automorphism) by which $u_i \rightarrow u_i$ for every u_i. By $\mathsf{Aut}(p)$, we have two automorphism orbits, $\vartheta_1 = \{u_1, u_5\}$ and $\vartheta_2 = \{u_2, u_3, u_4\}$.

There are 6 iso-matches, G_{f_k} of p in G that map u_1 to v_1 based on 6 subgraph isomorphisms, f_k, for $1 \leq k \leq 6$, in which u_1 maps to v_1, u_5 maps to v_5, u_2 maps to any of the 3 nodes in $\{v_2, v_3, v_4\}$, u_3 maps to any of the 2 nodes in $\{v_2, v_3, v_4\}$ that u_2 does not map to, and u_4 maps to one left that both u_2 and u_3 do not map to. Therefore, $|\mathsf{ISO}_{p,o}(v_1)| = 6$, where the 6 iso-matches are induced from the same node set $\{v_1, v_2, v_3, v_4, v_5\}$. As observed in Fig. 2, we have $|\mathsf{SubG}_{p,o}(v_1)| = 1$ by $|\mathsf{SubG}_{p,o}(v_1)| = |\mathsf{ISO}_{p,o}(v_1)| \cdot |\vartheta|/|\mathsf{Aut}(p)|$.

Here, $|\mathsf{ISO}_{p,o}(v_1)| = 6$, $|\vartheta| = 2$ for $\vartheta = \{u_1, u_5\}$ where $o = u_1$ is an orbit that maps to v_1, and $|\mathsf{Aut}(p)| = 12$. □

3 AN APPROACH BY HOMOMORPHISM

HOM queries are considered faster to process than ISO queries due to the fact that homomorphisms are not required to be injective. In [101], it shows that, for a pattern graph p and a node orbit o of p, an ISO query Q can be systematically divided into a set of HOM queries $(p' = (V_{p'}, E_{p'}), o')$ with $|V_{p'}| \leq |V_p|$ to process.

We present DISC [101], which is a general approach by homomorphism to compute SubG queries in brief. First, the count of $\mathsf{ISO}_{p,o}(v)$ is less than or equal to the count of $\mathsf{HOM}_{p,o}(v)$, as HOM is less restrictive than ISO. Hence, the count of $\mathsf{HOM}_{p,o}(v)$ is equal to the count of $\mathsf{ISO}_{p,o}(v)$ plus the count of $\nabla_{p,o}(v)$, where $\nabla_{p,o}(v)$ is the set of all non-injective homomorphisms f of p with $f(o) = v$. Note that by non-injective homomorphism, two nodes in p map to the same node in data graph G. Second, it shows that $|\nabla_{p,o}(v)|$ is the sum of $|\mathsf{ISO}_{p',o(p')}(v)|$ for any subpattern p' of p, where $o(p')$ denotes the node in p' that contains o, as there is a 1:1 connection between a subpattern p' of p and a non-injective homomorphism. Third, the count of $\mathsf{HOM}_{p,o}(v)$ is the sum of $|\mathsf{ISO}_{p',o(p')}(v)|$ for any subpattern p' of p and p itself.

$$|\mathsf{HOM}_{p,o}(v)| = \sum_{p' \in \{p\} \cup \mathsf{Sub}(p)} |\mathsf{ISO}_{p',o(p')}(v)| \tag{1}$$

With the assistance of the Möbius inversion formula, |ISO| can be computed by |HOM| as follows.

$$|\mathsf{ISO}_{p,o}(v)| = \sum_{p' \in \{p\} \cup \mathsf{Sub}(p)} \mu(p, p') \cdot |\mathsf{HOM}_{p',o(p')}(v)| \tag{2}$$

where $\mu(p, p')$ is the corresponding Möbius function and $\mathsf{Sub}(p)$ is the set of all subpatterns of p.

The key idea behind DISC is to process $|\mathsf{HOM}_{p,o}(v)|$ for a pattern graph $p = (V_p, E_p)$ by relational algebra efficiently using database techniques as follows, supposing that an undirected data graph G is stored in an edge relation $R_G(from, to)$.

$$_o\Upsilon_{\mathsf{count}(*)}(R_1 \bowtie R_2 \bowtie \cdots \bowtie R_{|E_p|}) \tag{3}$$

In Eq. (3), R_i is a renamed relation of R_G for an edge $e_i \in E_p$ of p, for $1 \leq i \leq |E_p|$, and $_o\Upsilon_F(\cdot)$ is to aggregate using the function F over each group grouped by the group-by attribute o.

To process Eq. (3) for a specific pattern graph p by HOM, DISC decomposes a complex cyclic join $(R_1 \bowtie R_2 \bowtie \cdots \bowtie R_{|E_p|})$ to a join tree based on tree decomposition [32]. With the tree decomposition T, DISC processes the complex joins $(R_1 \bowtie R_2 \bowtie \cdots \bowtie R_{|E_p|})$ in two steps. In the first step, it processes the joins for each node τ in the join tree, which results in an intermediate relation $\mathsf{rel}(\tau)$, and in the second step it processes the joins over all the intermediate relations. Following the *AGM* bound [8], the size of the intermediate result $(\mathsf{rel}(\tau))$ for a node τ is bounded by $|R_G|^{\mathsf{fhw}}$, where fhw is called the fractional hypertree width and is the minimum real number such that every node τ in the join tree T has a fractional edge cover of weight fhw. By pushing down group-by and aggregations, the time complexity of processing T is $O(|V_p| \cdot |R_G|^{\mathsf{fhw}})$ in [101].

The workflow of DISC [101] is depicted in the upper part in Fig. 3. ❶ The ISO count of p equals to the HOM count of p minus the ISO count of a set of subpatterns $(\mathsf{Sub}(p))$. ❷ The ISO count for a subpattern, p', in $\mathsf{Sub}(p)$ is counted in a similar manner recursively. Hence, $\mathsf{ISO}_p(o, v)$ becomes a linear combination of the HOM counts of patterns. ❸ DISC uses tree decomposition [32] to compute HOM count for each of such patterns with joins and aggregations. ❹ DISC proposes a multi-join algorithm Disc to compute joins and aggregations for each tree decomposition.

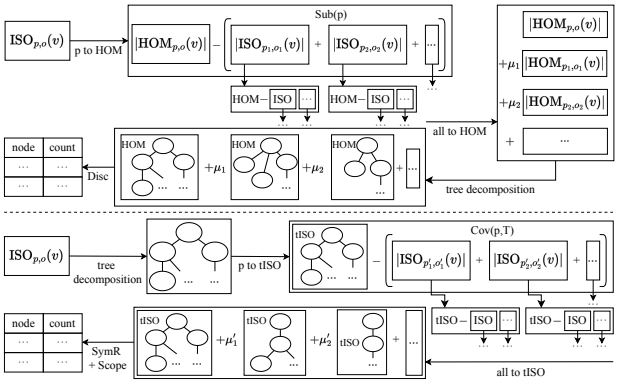


Figure 3: An overview of DISC and SCOPE

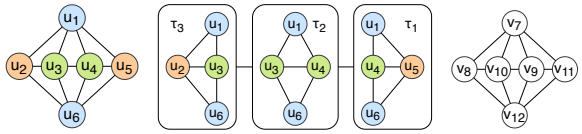
4 A NEW APPROACH BY PARAMETERIZED SUBGRAPH ISOMORPHISM

Subgraph isomorphism is studied in parameterized complexity [21, 61]. Here, a fixed-parameter algorithm is to deal with an NP-hard problem in running time $f(\kappa) \cdot n^c$ where κ is a parameter or a set of parameters, $f(\cdot)$ is a computable function which can be exponential on κ, and c is a constant which is independent of κ and n. Mark and Phlipczuk in [61] investigate the different parameters on the complexity of parameterized subgraph isomorphisms. In this paper, we focus on a parameterized algorithm by treewidth based on tree decomposition on the pattern graph p for local subgraph counting.

Definition 4.1: (**Tree Decomposition**) Given an undirected pattern graph $p = (V_p, E_p)$, a tree decomposition of p is a tree $T = (V_T, E_T)$. We use τ_i (simply τ) to denote a node in V_T, where τ_i maintains a nonempty subset of V_p, denoted as $V(\tau_i)$ $(\subseteq V_p)$, with which an induced subgraph of p can be constructed, denoted as $p(\tau_i)$. We say a node v in V_p appears in τ_i if $v \in V(\tau_i)$. The three conditions on T are as follows. (1) Every node in p is covered by T such that $V_p = \bigcup_{\tau_i \in V_T} V(\tau_i)$. (2) Every edge in p is covered by T such that for every edge $(u, v) \in E_p$, both u and v appear in at least one τ_i. (3) Nodes in T are connected if they all contain a pattern node. That is, if a node $v \in V_p$ appears in both τ_i and τ_j, then v appears in every τ_k on the path that connects τ_i and τ_j in T.

Given a tree decomposition T for a query $Q = (p, o)$, the root node of T is selected from a node that contains o. We denote a subtree rooted at node τ_i as T_i. The subtree T_i represents a subgraph of p, denoted as p_i (or $p(T_i)$), which is an induced subgraph of p over $\bigcup_{\tau_j \in T_i} V(\tau_j)$. We also use $\mathsf{parent}(\tau_i)$ to denote the parent node of τ_i in T, and $V_C(\tau_i)$ to denote the common nodes that appear in both $V(\tau_i)$ and $V(\mathsf{parent}(\tau_i))$, for a node τ_i in T. An induced subgraph $G(V_C(\tau_i))$, denoted as $G_C(\tau_i)$, can be constructed on $V_C(\tau_i)$.

Treewidth: The width of a tree decomposition T is the largest size



(a) pattern p (b) T (c) G

Figure 4: A pattern p, a tree decomposition T, and a graph G

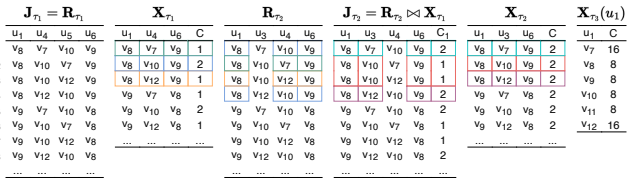


Figure 5: $\mathsf{tISO}_{p,o,T}|v|$ Computing

of $|V(\tau_i)|$ minus 1. There are several other metrics defined, including generalized hypertree width and fractional hypertree width [32]. Our approach can use any width. The treewidth of p, denoted as $\mathsf{tw}(p)$, is the smallest width over all possible tree decomposition for G. Below, we use $\mathsf{tw}(p)$ to denote a tree decomposition T whose width is $\mathsf{tw}(p)$. It is known that subgraph isomorphism for a pattern graph $p = (V_p, E_p)$ and a data graph $G = (V, E)$ can be solved in time $2^{O(|V_p|)} \cdot |V|^{O(\mathsf{tw}(p))}$ in [31, 61].

Example 4.1: Consider $\mathsf{ISO}_{p,o}(v)$ where p is a 6-node pattern shown in Fig. 4(a) and $o = u_1$. The tree decomposition T for p is shown in Fig. 4(b). In T, there are 3 nodes, τ_i, for $1 \le i \le 3$, that represent three subgraphs, $p(\tau_i)$. The root of T is τ_3 as it contains the orbit u_1. The subtree T_1 is τ_1, the subtree T_2 is the subtree of T rooted at τ_2, and the subtree $T_3 = T$. Here, $\mathsf{parent}(\tau_1) = \tau_2$, $\mathsf{parent}(\tau_2) = \tau_3$. The common nodes of τ_3 and τ_2 with its parent are $V_C(\tau_1) = \{u_1, u_4, u_6\}$, and $V_C(\tau_2) = \{u_1, u_3, u_6\}$, respectively; and $G_C(\tau_1)$ and $G_C(\tau_2)$ are simple paths, u_1–u_4–u_6 and u_1–u_3–u_6, respectively. The data graph G is also shown in Fig. 4(c).

TISO-based counting: We propose *tree-decomposition-based counting*. Let T_i be a subtree in a tree decomposition T for a pattern graph p. A tiso-match over $p(T_i)$ is a homo-match of $p(T_i)$ in G on the condition that $p(\tau_j)$ are iso-matches for every $\tau_j \in T_i$. Below, we use $|\mathsf{tISO}_{p,o,T}(v)|$ to denote the number of tiso-matches that match a node v in G to o, given a tree decomposition T.

Lemma 4.1: *For a query $Q = (p, o)$, $|\mathsf{ISO}_{p,o}(v)| \le |\mathsf{tISO}_{p,o,T}(v)| \le |\mathsf{HOM}_{p,o}(v)|$ for every node v in G.*

Proof Sketch: First, some nodes, u and u', in p that map to the same node in G by HOM cannot map to the same node by tISO if both appear in a node τ in T due to iso-matches in τ. We have $|\mathsf{tISO}_{p,o,T}(v)| \le |\mathsf{HOM}_{p,o}(v)|$. Second, some nodes, u and u', that do not appear in any node τ in T together may map to the same node in G by tISO, which cannot happen by iso-matches. We have $|\mathsf{ISO}_{p,o}(v)| \le |\mathsf{tISO}_{p,o,T}(v)|$. □

Next, we discuss how to compute $|\mathsf{tISO}|$ by enumerating iso-matches for every tree node τ in T. Here, for every τ in T, we find iso-matches of τ in G, and maintain it in a relation $\mathbf{R}_\tau = \mathsf{ISO}(p(\tau))$. For a leaf node τ in T, we keep the count of tiso-matches in a relation $\mathbf{X}(V_C(\tau), C)$ as follows.

$$\mathbf{X}_\tau(V_C(\tau), C) \;=\; {}_{V_C(\tau)}\Upsilon_{count(*)\to C}(\mathbf{R}_\tau) \tag{4}$$

Here ${}_{\nu}\Upsilon_{h\to A}(\cdot)$ is to apply an aggregate function h for a group ν and rename the output of h as A. For a non-leaf node τ in T, let $\mathbf{J}_\tau$ be as follows.

$$\begin{aligned}\mathbf{J}_\tau \;=\; & \mathbf{R}_\tau \bowtie_{V_C(\tau_1)} \rho_{C\to C_1}(\mathbf{X}_{\tau_1}) \bowtie_{V_C(\tau_2)} \rho_{C\to C_1}(\mathbf{X}_{\tau_2}) \\ & \cdots \bowtie_{V_C(\tau_k)} \rho_{C\to C_k}(\mathbf{X}_{\tau_k}))\end{aligned} \tag{5}$$

where τ_i for $1 \le i \le k$ is a child of τ such that $\tau = \mathsf{parent}(\tau_i)$, a join is a join on the common nodes of $V_C(\tau_i)$, and $\rho_{C\to C'}(X)$ is a rename operator to rename C in X to be C'. Then, for a non-leaf node τ that is not the root of T, we have

$$\mathbf{X}_\tau(V_C(\tau), C) \;=\; {}_{V_C(\tau)}\Upsilon_{sum(C_1\times C_2\times\cdots\times C_k)\to C}(\mathbf{J}_\tau) \tag{6}$$

For the root τ of of T, we have

$$\mathbf{X}_\tau(o, C) \;=\; {}_{o}\Upsilon_{sum(C_1\times C_2\times\cdots\times C_k)\to C}(\mathbf{J}_\tau) \tag{7}$$

where o is the orbit o of the given query $Q = (p, o)$. It is important to note that $|\mathsf{tISO}_{p,o,T}(v)|$ for every node v in G is the tuple in the table $\mathbf{X}_\tau(o, C)$ for $v = f(o)$ when τ is the root of T. As a special case, when there is only one node τ in T which is both the root and the leaf, we compute it as $\mathbf{X}_\tau(o, C) = {}_{o}\Upsilon_{count(*)\to C}(\mathbf{R}_\tau)$.

Example 4.2: Reconsider Example 4.1. We show $|\mathsf{tISO}_{p,o,T}(v)|$ computing for every node v in G in Fig. 5, focusing on when $f(u_1) = v_8$ or $f(u_1) = v_9$. ① For τ_1, $\mathbf{R}_{\tau_1}$ shows the iso-matches of V_{τ_1}, $\mathbf{X}_{\tau_1}$ shows the tISO-counts of $p(\tau_1)$ that have the same matches to $V_C(\tau_1) = \{u_1, u_4, u_6\}$ based on Eq. (4) as τ_1 is the leaf node in T. For example, the 2nd tuple of $(v_8, v_{10}, v_9, 2)$ in $\mathbf{X}_{\tau_1}$ shows that there are 2 iso-matches of $p(\tau_1)$ that contain the same three nodes in G by $\{u_1 \to v_8, u_4 \to v_{10}, u_6 \to v_9, u_5 \to v_7\}$ and $\{u_1 \to v_8, u_4 \to v_{10}, u_6 \to v_9, u_5 \to v_{12}\}$. ② For τ_2, $\mathbf{R}_{\tau_2}$ shows the iso-matches of $p(\tau_2)$ in G. In $\mathbf{R}_{\tau_2}$, there are 8 iso-matches of $p(\tau_2)$ in G when u_1 matches either v_8 or v_9. Following Eq. (5), $\mathbf{J}_{\tau_2} = \mathbf{R}_{\tau_2} \bowtie \mathbf{X}_{\tau_1}$. In a similar manner, $\mathbf{X}_{\tau_2}$ shows the tiso-counts of $p(T_2)$ grouped by $V_C(\tau_2) = \{u_1, u_3, u_6\}$ based on Eq. (6) as τ_2 is the non-root/non-leaf node in T. ③ In a similar manner we can compute τ_3.

We discuss how to compute $|\mathsf{ISO}|$ by $|\mathsf{tISO}|$ given a tree decomposition T for a pattern graph p. As shown in Eq. (1), $|\mathsf{HOM}_{p,o}(v)|$ is the sum of $|\mathsf{ISO}_{p,o}(v)|$ and $|\mathsf{ISO}_{p',o(p')}(v)|$ for any subpattern p' of p, where the set of subpatterns of p, $\mathsf{Sub}(p)$, is defined in [101] (Definition 4.2). In brief, a subpattern of $p = (V_p, E_p)$, $p' = (V_{p'}, E_{p'})$, is a pattern with less number of nodes ($|V_{p'}| < |V_p|$), due to the reason that some nodes in p map to the same node in p' by homomorphism. We follow [101] to describe such homomorphism below. In [101], it specifies $V_{p'}$ by a partition of nodes in V_p, denoted as $\mathcal{I} = \{I_1, I_2, \ldots, I_{|V_{p'}|}\}$, where I_k, for $1 \le k \le |V_{p'}|$, is an independent subset of V_p so that there are no edges in p between any two nodes in I_k. In other words, all the nodes in an I_k may map to the same node by homomorphism. A node v_k in $V_{p'}$ corresponds to one distinct I_k. An edge (v_i, v_j) exists in p' if there is an edge in p between some node in I_i and some node in I_j. As there is a one-to-one connection between a subpattern p' and a partition $\mathcal{I}$, it becomes possible to discuss subpatterns by such partitions. Below, we use $\mathbb{I} = \{\mathcal{I}_1, \mathcal{I}_2, \cdots\}$ to denote all subpatterns, p_i, in $\mathsf{Sub}(p)$.

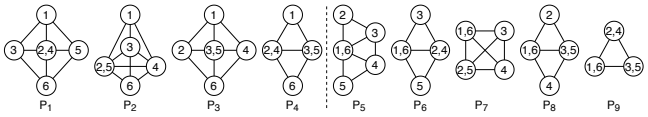


Figure 6: Cov **and non-**Cov **of** p **with** T **in Fig. 4**

Definition 4.2: Given a tree decomposition T on p, let p_i be a subpattern of p, where its corresponding partition is $\mathcal{I}_i = \{I_1, I_2, \cdots\}$ in $\mathbb{I}$. We say T covers p_i, if any two nodes, u and v, in the same I_k do not appear together in any node τ in T.

By Definition 4.2, the set of subpatterns of p covered by T is a subset of $\mathsf{Sub}(p)$, which we denote as $\mathsf{Cov}(p, T)$ ($\subseteq \mathsf{Sub}(p)$).

Example 4.3: Given a pattern p and its tree decomposition T in Fig. 4, the 9 subpatterns of p, $\mathsf{Sub}(p)$, are shown in Fig. 6, and the first 4 subpatterns are covered by T such that $\mathsf{Cov} = \{p_1, p_2, p_3, p_4\}$. Such 4 subpatterns covered by T are induced by the partitions $\mathcal{I}_1 = \{\{u_2, u_4\}, \{u_1\}, \{u_3\}, \{u_5\}, \{u_6\}\}$, $\mathcal{I}_2 = \{\{u_2, u_5\}, \{u_1\}, \{u_3\}, \{u_4\}, \{u_6\}\}$, $\mathcal{I}_3 = \{\{u_2\}, \{u_1\}, \{u_3, u_5\}, \{u_4\}, \{u_6\}\}$, $\mathcal{I}_4 = \{\{u_2, u_4\}, \{u_1\}, \{u_3, u_5\}, \{u_6\}\}$, respectively.

Proposition 4.1: *For a given pattern graph p, an orbit o, and a tree decomposition T, we have*

$$|\mathsf{HOM}_{p,o}(v)| - |\mathsf{tISO}_{p,o,T}(v)| = \sum_{p' \in \mathsf{Sub}(p) \setminus \mathsf{Cov}(p,T)} |\mathsf{ISO}_{p',o(p')}(v)| \quad (8)$$

Proof Sketch: It can be proved in a similar way as to prove Proposition 4.2 in [101]. We give the proof sketch below. To prove the $\leq$ part, we consider a homomorphism f that is not by tISO. First, with f, we can find a matching G_f in G. By G_f, we can find a subpattern p' in $\mathsf{Sub}(p) \setminus \mathsf{Cov}(p, T)$. Second, given p', we can find an injective homomorphism (or subgraph isomorphism) f' with which we find a matching $G_{f'}$ that is isomorphic to G_f. Hence, its count is included on the right by subgraph isomorphism regarding p'. To prove the $\geq$ part, we consider an subgraph isomorphism f' of a subpattern $p' \in \mathsf{Sub}(p) \setminus \mathsf{Cov}(p, T)$, and we can find a homomorphism f that is not by tISO for the pattern p. Hence its count is included on the left. The proposition is proved by the two inequalities. □

By combing Eq. (1) and Eq. (8), we have

$$|\mathsf{tISO}_{p,o,T}(v)| = \sum_{p' \in \{p\} \cup \mathsf{Cov}(p,T)} |\mathsf{ISO}_{p',o(p')}(v)| \quad (9)$$

Based on Eq. (9), we can compute $|\mathsf{ISO}_{p,o}(v)|$ by tISO using Algorithm 1 with which we have the following formula.

$$|\mathsf{ISO}_{p,o}(v)| = \sum_{p_i \in \mathcal{P}} \mu_i \cdot |\mathsf{tISO}_{p_i,o(p_i),T_i}(v)| \quad (10)$$

Here, $\mathcal{P}$ is the set of all distinct patterns in Λ, $\mu_i = \sum_{x=p_i} (-1)^{d_x+1}$, d_x is the depth of x in Λ, and the depth of the root is 1.

The workflow of our approach SCOPE is presented in the lower part in Fig. 3. ❶ We first get the tree decomposition for a given pattern p. ❷ We show that ISO count of p can be obtained by subtracting the tISO count with the ISO count of $\mathsf{Cov}(p, T)$, a subset of $\mathsf{Sub}(p)$ (Eq. (9)). ❸ With Algorithm. 1, we recursively apply Eq. (9) to compute ISO counts by tISO counts (Eq. (10)). ❹ We process tree decompositions by symmetry-breaking rules (Section 5) and our new multi-join algorithm Scope (Section 6). Notably, symmetry-breaking rules are exclusive to ISO with which we define tISO. In particular, we use ISO for nodes in T.

Algorithm 1: tISOToISO (p)

Input: pattern graph p
Output: A formula to compute $|\mathsf{ISO}_{p,o}(v)|$ by tISO

```
Λ ← p, push p to Q;
while Q ≠ ∅ do
    pop Q to p'; compute T_{p'} and Cov(p', T_{p'});
    foreach p'' ∈ Cov(p', T_{p'}) do
        add a node p'' and an edge (p', p'') into Λ; push p'' to Q;
return GenEq(Λ.root);
Procedure GenEq(x)
    return tISO_{x,o(x),T_x} − Σ_{y∈x.child} GenEq(y);
```

5 TISO-BASED COUNTING

As shown in Eq. (4)-Eq. (7), a main cost in tISO-based counting is to compute ISO-matches ($\mathbf{R}_\tau$) for every tree node τ in a tree decomposition T. And the key issue is how to compute ISO-matches $\mathbf{R}_\tau$ using automorphism orbits. A common technique to compute ISO for a given pattern graph p is by symmetry-breaking, which is used in subgraph enumeration to reduce the number of iso-matches of p in G if there exist automorphism orbits in p [33, 57, 77, 96]. This is due to the fact that a subgraph in G can be iso-matched multiple times given automorphism orbits. We introduce symmetry-breaking rules below.

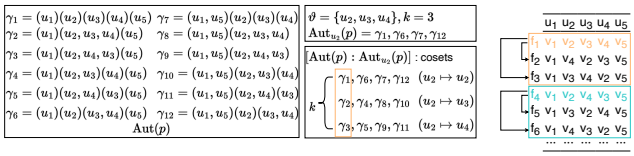
Symmetry-breaking rules: For a pattern graph p with automorphisms $\mathsf{Aut}(p)$, a symmetry-breaking is an automorphism orbit of $\mathsf{Aut}(p)$, namely, $\vartheta = \{u_1, u_2, \cdots, u_k\}$. For each pair (u_1, u_i) in ϑ, for $2 \leq i \leq k$, a partial order $\prec$ is imposed such that $u_1 \prec u_i$. A symmetry-breaking rule (*SymR*) for a given ϑ is presented in the form of $\theta = \{u_1 \prec u_2, \cdots, u_1 \prec u_k\}$. Assume that there is a total order ($<$) on nodes in the data graph G. By a symmetry-breaking rule, it enforces $f(u_1) < f(u_i)$ in the data graph G, if $u_1 \prec u_i$ in θ. In other words, An iso-match of p in G is valid if $f[u_1] < f[u_i]$ for all $i \in [2, k]$ in a *SymR* θ.

Existing works enumerate subgraphs using a set of *SymR*s [6, 26, 33, 35, 45, 57–59, 77, 79, 84, 96, 100] for a pattern graph without tree decomposition. In this work, we study how to compute ISO for every tree node τ using automorphism orbits and its *SymR*s, given a tree decomposition. It is important to note that this technique cannot be used for homomorphisms.

5.1 More about Automorphism Orbits

Given a pattern graph p, consider an automorphism orbit $\vartheta = \{u_1, u_2, \cdots, u_k\}$ in $\mathsf{Aut}(p)$ which can be represented by *SymR* $\theta = \{u_1 \prec u_2, \cdots, u_1 \prec u_k\}$. We define a stabilizer subgroup of $\mathsf{Aut}_\vartheta(p) = \{\gamma | \gamma(u_1) = u_1 \text{ for } \gamma \in \mathsf{Aut}(p)\}$ [24]. Note that $\mathsf{Aut}_\vartheta(p)$ is defined by fixing u_1 in ϑ.

With the stabilizer, the (left) cosets of $\mathsf{Aut}_\vartheta(p)$ in $\mathsf{Aut}(p)$, denoted as $[\mathsf{Aut}(p) : \mathsf{Aut}_\vartheta(p)]$, are disjoint and are in the same size obtained by composing each automorphism of $\mathsf{Aut}_\vartheta(p)$ by a γ in $\mathsf{Aut}(p)$ such as $[\mathsf{Aut}(p) : \mathsf{Aut}_\vartheta(p)] = \cup_{\gamma \in \mathsf{Aut}(p)} \{\gamma \circ \gamma' | \gamma' \in \mathsf{Aut}_\vartheta(p)\}$. By the orbit-stabilizer theorem in group theory, the number of cosets is $|\vartheta|$



(a) automorphisms, stabilizer subgroup and cosets (b) two *msets*

Figure 7: automorphisms and the θ mapping set

and $|\text{Aut}_\vartheta(p)| = |\text{Aut}(p)|/k$ for $k = |\vartheta|$. That is, there are k disjoint cosets in the same size, and in the i-th coset the automorphisms send u_1 to the same u_i in ϑ.

Example 5.1: Consider the pattern graph p in Fig. 2(a). Its $\text{Aut}(p)$ is in Fig. 7(a) where $|\text{Aut}(p)| = 12$. There is an automorphism orbit $\vartheta = \{u_2, u_3, u_4\}$ in p, which can serve the role of stabilizer as $\text{Aut}_\vartheta(p) = \{\gamma_1, \gamma_6, \gamma_7, \gamma_{12}\}$. Here, $|\text{Aut}_\vartheta(p)| = |\text{Aut}(p)|/|\vartheta| = 12/3 = 4$. We explain how a coset in $[\text{Aut}(p) : \text{Aut}_\vartheta(p)]$ is constructed. Consider $\gamma_6 \in \text{Aut}(p)$. By composing γ_6 with each of $\text{Aut}_\vartheta(p)$, we have the first coset, $\{\gamma_1, \gamma_6, \gamma_7, \gamma_{12}\}$. In detail, $\gamma_6 \circ \gamma_1 = \gamma_6$, $\gamma_6 \circ \gamma_6 = \gamma_1$, $\gamma_6 \circ \gamma_7 = \gamma_{12}$, $\gamma_6 \circ \gamma_{12} = \gamma_7$. Hence γ_6 is one that produces the first coset. Note that in the 1st coset all automorphisms send u_2 to u_2. The 2nd coset sends u_2 to u_3, and the 3rd coset sends u_2 to u_4 for $\vartheta = \{u_2, u_3, u_4\}$.

The stabilizer and the corresponding cosets are discussed regarding a pattern graph p. Given the k cosets by the stabilizer $\text{Aut}_\vartheta(p)$ where $|\vartheta| = k$, we define a set of iso-mappings from p to G called an ϑ mapping set (ϑ-mapset).

Definition 5.1: (ϑ-mapset) For a stabilizer $\text{Aut}_\vartheta(p)$ over $\vartheta = \{u_1, u_2, \cdots, u_k\}$, there are k cosets. Let $\tilde{\gamma}_i$ be the automorphism selected from the i-th cosets for $1 \le i \le k$. An ϑ mapping set (ϑ-mapset) is a set of k iso-mappings $\Phi = \{\phi_1, \phi_2, \cdots, \phi_k\}$ where ϕ_i is an iso-mapping by $\tilde{\gamma}_i$ over the same set of nodes in G. For the *SymR* $\theta = \{u_1 < u_2, \cdots, u_1 < u_k\}$ over ϑ, $\phi_1 \in \Phi$ is called the representative of ϑ-mapset (Φ), if it satisfies *SymR* θ.

Reconstruction of ϑ mapping set: It is obvious that ϕ_1 is the only iso-mapping in Φ that satisfies *SymR* θ. Given ϕ_1 for ϑ-mapset, we can reconstruct iso-mappings ϕ_i as $\phi_i(u_j) = \phi_1(\tilde{\gamma}_i(u_j))$ for $u_j \in V_p$. It is important to note that we do not need to find other iso-mappings if we find ϕ_1, we can reconstruct the other mappings.

Example 5.2: Reconsider Example 5.1. In Fig. 7(a), there are 3 cosets for $\vartheta = \{u_2, u_3, u_4\}$ in Fig. 7(a), and we have $\tilde{\gamma}_1 = \gamma_1$, $\tilde{\gamma}_2 = \gamma_2$, and $\tilde{\gamma}_3 = \gamma_3$. Over ϑ, the *SymR* $\theta = \{u_2 < u_3, u_2 < u_4\}$. Given the data graph G in Fig. 2(b), we show two ϑ-mapsets in Fig. 7(b) where one is formed by the first 3 iso-mappings, and one is formed by the second 3 iso-mappings. Consider the 2nd ϑ-mapset in which f_4 (ϕ_1) is the representative that satisfies the *SymR*, and we can construct f_5 (ϕ_2) and f_6 (ϕ_3) by composing f_4 with $\tilde{\gamma}_2$ and $\tilde{\gamma}_3$, respectively.

Generating multiple *SymR*s: There are multiple sets of *SymR*s for a pattern graph p. An algorithm in [33] generates one set of *SymR*s for p randomly. Existing works show the efficiency of enumerating subgraphs using such a set of *SymR*s [3, 6, 26, 33, 35, 45, 57–59, 77, 79, 84, 96, 100]. Different from the existing work, we explore how to utilize *SymR*s for every tree node τ in a tree decomposition T, and we need to select *SymR*s for τ's and for the entire T. There exists

Algorithm 2: GenAllRules (p)

```
Input: a pattern graph p
Output: all sets of SymRs for p
1  A ← Aut(p); R ← ∅;
2  GenRules (p, A, ∅, ∅);
3  return R;
4  Procedure GenRules(p, A, F, R)
5      if |A| = 1 then R ← R ∪ {R};
6      else
7          Θ ← the set of equivalent classes in A constrained by F;
8          foreach ϑ ∈ Θ do
9              let u_i be the one with the smallest id in ϑ;
10             F' ← F ∪ {u_i};
11             let θ be the set of SymRs by ϑ;
12             R' ← R ∪ {θ};
13             A' ← A constrained by F;
14             GenRules (p, A', F', R');
```

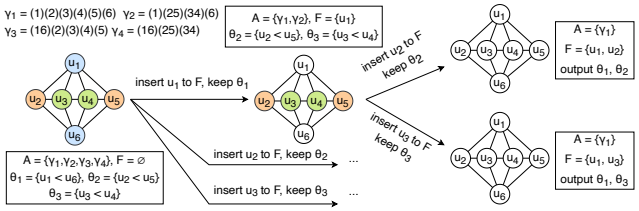
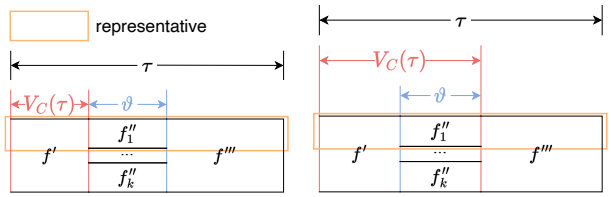


Figure 8: Generating *SymR*s

some *SymR* that cannot be efficiently used with T, which we will discuss later. Such an issue does not occur when applying *SymR*s to p without T.

We give an algorithm GenAllRules (Algorithm 2) to generate all sets of *SymR*s based on the algorithm given in [33]. We explain GenAllRules using the pattern graph p in Fig. 4(a) as an example, where we consider p as τ in T. There are 4 automorphisms such that $\text{Aut}(p) = \{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$, where $\gamma_1 = (u_1)(u_2)(u_3)(u_4)(u_5)(u_6)$, $\gamma_2 = (u_1)(u_2u_5)(u_3u_4)(u_6)$, $\gamma_3 = (u_1u_6)(u_2)(u_3)(u_4)(u_5)$, and $\gamma_4 = (u_1u_6)(u_2u_5)(u_3u_4)$. With $\text{Aut}(p)$, there are 3 symmetry-breakings, $\vartheta_1 = \{u_1, u_6\}$, $\vartheta_2 = \{u_2, u_5\}$, and $\vartheta_3 = \{u_3, u_4\}$. In GenAllRules, $\mathcal{R}$ is the set of *SymR*s to be generated, which is initialized to be empty. It initially calls the procedure GenRules with the inputs of the pattern graph p and $A = \text{Aut}(p)$. In the procedure, F is the constraints when selecting *SymR*s based on the automorphisms A, and R is one set of *SymR*s to be generated. Initially, there are no constraints, so $F = \emptyset$, $\Theta = \{\vartheta_1, \vartheta_2, \vartheta_3\}$ (line 7). Suppose that $\vartheta_1 = \{u_1, u_6\}$ is selected (line 8), u_1 is added into F' as a constraint, with which it constrains that only an automorphism, γ_i, with (u_1) can be further explored next. In other words, u_1 is fixed. Here, the *SymR* $\theta = \{u_1 < u_6\}$ is added into R' (lines 11-12), the automorphism in A constrained by F' is $A' = \{\gamma_1, \gamma_2\}$ in which (u_1) appears as constrained. It recursively calls GenRules. The procedure is illustrated in Fig. 8. The output of GenAllRules is $\mathcal{R} = \{\mathcal{R}_1, \mathcal{R}_2\}$, where $\mathcal{R}_1 = \{\theta_1, \theta_2\}$ and $\mathcal{R}_2 = \{\theta_1, \theta_3\}$. Below, we use $\mathcal{R}_\tau$ to denote the set of sets of *SymR*s for a tree node τ.

The ϑ independency: Let ϑ and ϑ' be any two automorphism orbits in the same set of *SymR*s (e.g., $\mathcal{R}_i$) generated by GenAllRules



(a) $\vartheta \cap V_C(\tau) = \emptyset$ (b) $\vartheta \subset V_C(\tau)$

Figure 9: *SymR* with τ and T

$\mathbf{R}'_{\tau_1}$

u_1	u_4	u_5	u_6
v_8	v_7	v_{10}	v_9
v_8	v_{10}	v_7	v_9
v_8	v_{10}	v_{12}	v_9
v_8	v_{12}	v_{10}	v_9
...	...	...	...

$\mathbf{X}'_{\tau_1}$

u_1	u_4	u_6	C
v_8	v_7	v_9	1
v_8	v_{10}	v_9	2
v_8	v_{12}	v_9	1
...	...	...	...

(a) $\{u_1 < u_6\}, \tau_1$

$\mathbf{R}'_{\tau_2}$

u_1	u_3	u_4	u_6
v_8	v_7	v_{10}	v_9
v_8	v_{10}	v_7	v_9
v_8	v_{10}	v_{12}	v_9
v_8	v_{12}	v_{10}	v_9
...	...	...	...

$\mathbf{J}'_{\tau_2} = \mathbf{R}'_{\tau_2} \bowtie \mathbf{X}'_{\tau_1}$

u_1	u_3	u_4	u_6	C
v_8	v_7	v_{10}	v_9	2
v_8	v_{10}	v_7	v_9	1
v_8	v_{10}	v_{12}	v_9	1
v_8	v_{12}	v_{10}	v_9	2
...	...	...	...	...

$\mathbf{X}'_{\tau_2}$

u_1	u_3	u_6	C
v_8	v_7	v_9	2
v_8	v_{10}	v_9	2
v_8	v_{12}	v_9	2
...	...	...	...

(b) $\{u_1 < u_6\}, \tau_2$

Figure 10: Apply *SymR*s to τ's given p and T in Fig. 4(a)

(Algorithm 2). There are only two cases between ϑ and ϑ': one is contained in another, and the other is the two are disjoint. ϑ and ϑ' are independent since ϑ' is an automorpihsm orbit when we fix a node in ϑ or vice-versa. The corresponding $\tilde{\gamma}$ of the ϑ-mapsets are independent. Therefore, any subset of a set of *SymR*s can be used. We call this ϑ independency.

5.2 Automorphism Orbits in Tree Nodes

Following the discussion on automorphisms for a pattern graph, with a tree decomposition T, we compute iso-matches ($\mathbf{R}_\tau$) for each tree node $\tau \in T$ using *SymR*s by considering τ as a pattern graph. Here, a key issue is the relationship between *SymR* θ for τ and $V_C(\tau)$. Note that $V_C(\tau)$ is used as the group-by attributes in Eq. (4) and Eq. (6) to compute tISO counts. We need to ensure that the application of *SymR*s for a tree node τ does not affect the aggregation by the group-by attributes $V_C(\tau)$.

There are 4 cases for an automorphism ϑ (or its *SymR* θ) regarding a tree node τ, namely, ❶ $\vartheta \not\subset \tau$, ❷ $\vartheta \cap V_C(\tau) = \emptyset$, ❸ $\vartheta \subset V_C(\tau)$, and ❹ $\vartheta \cap V_C(\tau) \neq \emptyset$ and $\vartheta \not\subset V_C(\tau)$. The last 3 cases are the cases when $\vartheta \subset V(\tau)$. Assume there is an iso-match f of the entire τ such that $f = f' \| f'' \| f'''$, where f' is for the part of mapping by $V_C \setminus \vartheta$, f'' is the part of mapping by ϑ, and f''' is for the part of mapping by $V_\tau \setminus (V_C \cup \vartheta)$. By *SymR* θ for ϑ, it affects f'', or more precisely, its ϑ-mapset of size $k = |\vartheta|$. For ❶, as $\vartheta \not\subset \tau$, ϑ has no impacts on V_C. For ❷, it only finds the representative f''_1 using *SymR* θ. As ϑ and $V_C(\tau)$ is disjoint, its count without *SymR* θ is its count with *SymR* θ multiplied by k. For ❸, it only finds the representative f''_1 using *SymR* θ for ϑ. As ϑ is included in $V_C(\tau)$, it needs to reconstruct the other mappings, f''_i, in the ϑ-mapset. For ❹, it is prohibited for the reason that there may exist two different ϑ-mapsets whose match to V_C overlap but are not the same. The count by such V_C is incorrect. We show ❷ and ❸ in Fig. 9.

Example 5.3: Consider Example 5.2 where we take this pattern graph p as τ in a tree decomposition T. As shown in Fig. 7(a), there are 3 cosets for $\vartheta = \{u_2, u_3, u_4\}$ in Fig. 7(a), where $k = |\vartheta| = 3$. Over ϑ, *SymR* $\theta = \{u_2 < u_3, u_2 < u_4\}$. For ❷, suppose $V_C(\tau) = \{u_1\}$, and we only get f_4 for the 2nd θ mapping set. There should be additional $k-1$ iso-mappings. The count for $V_C = \{u_1\}$ needs to by multiplied by k for a ϑ-mapset. For ❸, suppose $V_C(\tau) = \{u_1, u_2, u_3, u_4\}$, and we only get f_4 for the 2nd ϑ-mapset. There should be additional $k-1$ iso-mappings to be reconstructed.

We discuss how to do count-correction/reconstruction. Here, we discuss it by assuming that $\mathbf{R}_\tau$ is for a single ϑ-mapset, which is a set of k iso-mappings $\Phi = \{\phi_1, \phi_2, \cdots, \phi_k\}$ where ϕ_i is an iso-mapping by $\tilde{\gamma}_i$ over the same set of nodes in G (Definition 5.1). Among all in Φ, ϕ_1 is the one that satisfies the *SymR* θ.

First, for ❷, we only need to do count-correction, as illustrated in Fig. 9(a). We give the details for a non-leaf node τ that is not the root (Eq. (5) and Eq. (6)) regarding a given ϑ where $\vartheta \cap V_C(\tau) = \emptyset$. The others can be dealt with in a similar manner. We show how we correct it for a single ϑ-mapset. If we can do it for a single ϑ-mapset, the overall sum in Eq. (6) is correct as it is the sum of the counts for all ϑ-mapsets by the group-by attributes $V_C(\tau)$. It is worth noting that the only place that changes is $\mathbf{R}_\tau$ in Eq. (5) if we enforce *SymR* θ w.r.t ϑ, where every child $\mathbf{X}_{\tau_i}$ remains unchanged. Without the *SymR* θ, the size of $\mathbf{R}_\tau$ is $k = |\vartheta|$, and the size of the corresponding $\mathbf{J}_\tau$ is k because it can only join one tuple from $\mathbf{X}_{\tau_i}$. By enforcing *SymR* θ, the size of $\mathbf{R}_\tau$ becomes 1 for the representative ϕ_1, and its count C in $\mathbf{X}_\tau(V_C(\tau), C)$ in Eq. (6) becomes C/k. We correct its count by multiplying it by k. There is no need to do reconstruction.

Second, for ❸, there is no need to do count-correction. We explain it using Fig. 9(b). The count for f, where f''_1 is the representative of a ϑ-mapset, is reduced by $1/k$ for $k = |\vartheta|$. That is the correct count of f with f''_1. We only need to reconstruct the other mappings following the discussion of reconstruction given in Section 5.1. That is, we reconstruct the other f''_i in Φ by f''_1.

Example 5.4: Consider Example 4.1 where its pattern graph p, tree decomposition T, and data graph G are in Fig. 4. There are 3 tree nodes, τ_1, τ_2, and τ_3, and the root is τ_3. For the leaf node τ_1 with $V_C(\tau_1) = \{u_1, u_4, u_6\}$, there are two *SymR*s $\theta_{11} = \{u_1 < u_6\}$ over $\vartheta_{11} = \{u_1, u_6\}$, and $\theta_{12} = \{u_4 < u_5\}$ over $\vartheta_{12} = \{u_4, u_5\}$. Each has 2 cosets. For θ_{11}, $\tilde{\gamma}_1(\theta_{11}) = (u_1)(u_4)(u_5)(u_6)$, and $\tilde{\gamma}_2(\theta_{11}) = (u_1, u_6)(u_4)(u_5)$. For θ_{12}, $\tilde{\gamma}_1(\theta_{12}) = (u_1)(u_4)(u_5)(u_6)$, and $\tilde{\gamma}_2(\theta_{12}) = (u_1)(u_6)(u_4, u_5)$. We explain ϑ_{11} which is the case ❸, because $\vartheta_{11} \subset V_C(\tau_1)$, for the leaf node τ_1. For a leaf-node τ_1, $\mathbf{J}_{\tau_1} = \mathbf{R}_{\tau_1}$. First, $\mathbf{R}_{\tau_1}$ with $\mathbf{X}_{\tau_1}$ are shown in Fig. 5 without *SymR*s. Second, consider $\mathbf{R}_{\tau_1}$ with *SymR* $\theta_{11} = \{u_1 < u_6\}$. One of its ϑ-mapset is $\{f_1, f_5\}$, where for example $f_1 = \{u_1 \to v_8, u_4 \to v_7, u_5 \to v_{10}, u_6 \to v_9\}$, and the other ϑ-mapsets are $\{f_2, f_6\}$, $\{f_3, f_7\}$, and $\{f_4, f_8\}$. Third, we have $\mathbf{R}'_{\tau_1}$ by enforcing *SymR* θ_{11} as shown in Fig. 10(a), where it only keeps the representatives of ϑ-mapsets, $\{f_1, f_2, f_3, f_4\}$. The corresponding $\mathbf{X}'_{\tau_1}$ is also shown in Fig. 10(a). Comparing $\mathbf{R}_{\tau_1}$ in Fig. 5, the number of iso-mappings in $\mathbf{R}'_{\tau_1}$ in Fig. 10(a) is reduced by a half as $|\vartheta_{11}| = 2$. And $\mathbf{X}'_{\tau_1}$ is also reduced by a half comparing $\mathbf{X}_{\tau_1}$. We can reconstruct $\mathbf{X}_{\tau_1}$ from $\mathbf{X}'_{\tau_1}$ using the representatives $\{f_1, f_2, f_3, f_4\}$. For example, with f_1, we can get f_5 (e.g., (v_9, v_7, v_8) from (v_8, v_7, v_9)) by swapping the node mapped by u_1 and u_6, and keep the count as the one in f_1. □

The ϑ independency in a tree node τ: We have discussed the ϑ independency for a pattern graph in Section 5.1. We also need to impose such dependency for a tree node τ in T. We need certain conditions regarding $V_C(\tau)$ for τ and $V_C(\tau_i)$ for each child τ_i of

τ in T. First, for $V_C(\tau)$, the condition is that for any single ϑ in τ, the automorphism $\tilde{\gamma}_l$ selected from the l-th coset must satisfy $\tilde{\gamma}_l(u_j) = u_j$ for any $u_j \in V_C(\tau) \setminus \vartheta$. The main idea is to ensure that group-by attributes not in ϑ are fixed when computing the aggregation so that the aggregation result can be correctly used in its parent. For example, in Fig. 7(a), $\vartheta = \{u_2, u_3, u_4\}$ for a τ over the set of nodes $\{u_1, u_2, u_3, u_4, u_5\}$. Suppose it is a leave node τ in T, and $V_C(\tau) = \{u_2, u_3, u_4, u_5\}$, we have $V_C(\tau) \setminus \vartheta = \{u_5\}$. Consider the 2nd coset, γ_2 can be $\tilde{\gamma}_2$ as $\gamma_2(u_5) = u_5$, and γ_8 cannot be selected as $\tilde{\gamma}_2$. Second, for $V_C(\tau_i)$, we ensure that $\tilde{\gamma}_l(u_j) = u_j$ for any $u_j \in V_C(\tau_i) \setminus \vartheta$, and that $\vartheta \cap V_C(\tau_i) = \emptyset$ or $\theta \subset \mathcal{R}_{\tau_i}$. Note that $\theta \subset \mathcal{R}_{\tau_i}$ implies $\vartheta \subset V_C(\tau_i)$. These conditions ensure that in Eq. (5), mappings in the same ϑ-mapset of τ have the same $C_1, \cdots C_w$, therefore establishing the correctness of computing aggregation in τ. If such a $\tilde{\gamma}_i$ does not exist, we do not use this ϑ. If two orbits ϑ and ϑ' both satisfy these conditions, they are independent and both can be used.

The automorphism orbit at the root of T: We discuss how we use *SymR*s at the root node τ in T, where its $V_C(\tau) = \emptyset$ as it does not have any parent. The group-by attribute in the root is o, a single node, which is the orbit in a local subgraph counting query $Q = (p, o)$. There are two cases. One is $o \notin \vartheta$, and one is $o \in \vartheta$. For the former, it is a similar case that ϑ and $V_C(\tau)$ are disjoint to be dealt with. For the latter, for any *SymR* $\theta = (o < u_i)$, we only need to reconstruct by swapping o with u_i w.r.t the mapping by *SymR* θ.

5.3 The Optimizations

We have discussed how to correct the count and reconstruct a single ϑ-mapset for a tree node τ_i in T. We can ensure all the counts are correct if we do so for every τ_i in T. We propose an optimization technique in a way that we do not need to do so for every τ_i in T, and we can delay it from τ_i to its parent τ under certain conditions.

First, consider the case ❷ for τ_i where $\vartheta \cap V_C(\tau_i) = \emptyset$, its parent $\tau = \text{parent}(\tau_i)$ must not have ϑ by tree decomposition. The count-correction can be delayed from τ_i to τ. We explain it below. Suppose we have to do count-correction by multiplying k for a ϑ-mapset in τ_i when a *SymR* θ is used. That is we have to correct $\mathbf{X}_{\tau_i}(V_C(\tau_i), C)$ with the ϑ-mapset to be $\mathbf{X}_{\tau_i}(V_C(\tau_i), C \cdot k)$ by Eq. (6). Furthermore, consider τ which is the parent of τ_i. As shown in Eq. (6), its C_i before correction under τ_i becomes $C_i \cdot k$ after correction, for the parent τ. We have ${}_{V_C(\tau)}\Upsilon_{sum(C_1 \times \cdots \times C_i \cdot k \times \cdots) \to C} = {}_{V_C(\tau)}\Upsilon_{sum(C_1 \times \cdots \times C_i \times \cdots) \cdot k \to C}$ w.r.t Eq. (6),where C_i is the count before count-correction in τ_i. In other words, it is possible that we do not do count-correction by multiplying k in τ_i but do it in its parent τ.

Second, for the case ❸ where $\vartheta \subset V_C(\tau_i)$, its parent τ must have the same ϑ. There are 3 sub-cases with τ, ϑ is an automorphism orbit in τ (❷, ❸), and ϑ is not an automorphism orbit in τ. When ϑ is an automorphism in τ for the sub-cases ❷/❸, the counts by $V_C(\tau)$ will not be affected by ϑ-mapsets in τ_i, because, for any ϑ-mapset in τ_i, there will be one and only one ϑ-mapset in τ that map to the same nodes in G by the same ϑ. This is similar to the discussion in Section 5.2. We can delay reconstruction. Third, for the case that ϑ is not an automorphism orbit in τ, we have to do reconstruction for τ_i, and we cannot delay it.

Example 5.5: Continue Example 5.4. As given in Example 5.4, for the leaf node τ_1 with $V_C(\tau_1) = \{u_1, u_4, u_6\}$, there are two *SymR*s $\theta_{11} = \{u_1 < u_6\}$ over $\vartheta_{11} = \{u_1, u_6\}$, and $\theta_{12} = \{u_4 < u_5\}$ over $\vartheta_{12} = \{u_4, u_5\}$. For τ_2, the *SymR*s are $\theta_{21} = \{u_1 < u_6\}$ and $\theta_{22} = \{u_3 < u_4\}$. Here, $V_C(\tau_2) = \{u_1, u_3, u_6\}$. For τ_3 (the root), the *SymR* is $\theta_{31} = \{u_1 < u_6\}$, and $o = u_1 \in \tau_3$ where o is the orbit of the given query $Q = (p, o)$. Note that θ_{11} in τ_1, θ_{21} in τ_2, and θ_{31} are identical.

The results by count-correction/reconstruction for every tree node are shown in Fig. 5. In Fig. 10(a), we show the result in $\mathbf{X}'_{\tau_1}$ by enforcing the *SymR* θ_{11} in τ_1, together with its $\mathbf{R}'_{\tau_1}$. Here, in τ_1, it is the case ❸, for $\vartheta_{11} \subset V_C(\tau_1)$, and in τ, it is the case of ❸ as well, such that $\vartheta_{21} \subset V_C(\tau_2)$. Note $\vartheta_{21} = \vartheta_{11}$. We can delay reconstruction from τ_1 to τ_2. The result is presented in Fig. 10(b).

We can also delay reconstruction to τ_3. Note that τ_3 is the root of T, which does not have $V_C(\tau_3)$, and we can treat it in a specific way as discussed. That is, for the *SymR* θ_{31}, we reconstruct $\phi_2(u_1)$ from $\phi_1(u_6)$ for each ϑ-mapset. □

Selecting *SymR*s: We select *SymR*s for T as follows. First, we obtain $\mathcal{R}_\tau$ for each τ in T by GenAllRules. Second, we check each combination of all rule sets in all tree nodes. Let $\mathcal{R}'_\tau \in \mathcal{R}_\tau$ be the selected rule set of τ. We check the constraints for ϑ-mapsets independency and remove invalid rules in $\mathcal{R}'_\tau$. Third, we use a simple cost function $\sum_{\tau \in T, \theta \in \mathcal{R}'_\tau} |\theta| * |V_\tau|$ to select one set of rules that maximizes this function. Here, we prefer rules with larger $|\theta|$ since they can reduce more matches for a given tree node. Additionally, we prefer rules applied to larger tree nodes because the induced subgraphs are more challenging to enumerate.

We show some statistics about *SymR*s in Table 1. The 1st column is k for k-node pattern graph p, the 2nd column is the total number of pattern graphs with such k, the 3rd column is the total number of patterns that have *SymR*s, the 4th column is the number of tree decompositions that can use all *SymR*s that appear in p in their nodes, the 5th column is the number of tree decompositions that can use at least one *SymR* that appears in p in their nodes, and the last column is the percentage of *SymR*s that appear in p to be used in T on average. A majority of pattern graphs are with *SymR*s with tree decomposition.

Table 1: Some statistics about *SymR*s

k-node	# of p	Symmetry(p)	All-in-T	$\geq$1-in-T	%
5	58	58	24	36	51.1%
6	407	359	144	238	53.1%
7	4,306	3,298	1,405	2,171	54.3%

6 MULTI-JOIN ALGORITHMS

In this section, we discuss how to process a pattern graph $p = (V_p, E_p)$ given its tree decomposition T based on the multi-join algorithm Leapfrog [91], which is a state-of-the-art worst-case optimal algorithm that is also used in [101]. In brief, Leapfrog is to process a join query over m relations based on the join attribute order using iterators. To process it for p by Leapfrog, we can represent an edge $e_i \in E_p$ as a relation for $|E_p| = m$.

There are several ways to process the aggregations given T. First, following Eq. (4)-Eq. (7), we can process every tree node $\tau \in T$ using Leapfrog, maintain its result in $\mathbf{X}_\tau$, and join all such $\mathbf{X}_\tau$ relations. As pointed out in [101], this approach cannot be taken when the

sizes of such relations are too large to keep in the main memory. For example, in Fig. 4(b), there are 3 tree nodes, and for $i = 1$ or $i = 2$, the relation $\mathbf{X}_{\tau_i}(V_C(\tau_i), C)$ is a 4 attribute relation where $V_C(\tau_i)$ is the group-by attributes and C maintains its aggregation. In this example, the size of $\mathbf{X}_{\tau_i}(V_C(\tau_i), C)$ is $O(n^k)$ for $k = |V_C(\tau_i)|$ where n is the number of nodes in the data graph G. We call it an $O(n^k)$-approach, which is related to the space complexity.

In [101], DISC proposed an $O(1)$-approach regarding the memory, which we show in Algorithm 3. We call it Disc and explain it using an example decomposition T.

Example 6.1: Consider a tree decomposition T for a pattern graph p with 3 tree nodes, τ_1, τ_2, and τ_3, where τ_3 is the root as shown at the top in Fig. 11. We assume that τ_1 is over two subgraphs B and D with $V_C(\tau_1) = B$, τ_2 is over three subgraphs A, E, and B with $V_C(\tau_2) = A$, and τ_3 is over two subgraphs F and A. Furthermore, we assume that $V_C(\tau_1) = B$ can be divided into two disjoint sets, $V_C^1(\tau_1) = B_1$ and $V_C^2(\tau_1) = B_2$, and $V_C(\tau_2) = A$ can be divided into two disjoint sets, $V_C^1(\tau_2) = A_1$ and $V_C^2(\tau_2) = A_2$.

For T in Example 6.1, Disc (Algorithm 3) processes it as follows starting from the root τ_3 in T. ❶ It finds a partial match $f_{AB} = f_A \| f_B$ that matches $V_C(\tau_2) = A$ in τ_2 and then $V_C(\tau_1) = B$ in τ_1. ❷ Given f_{AB}, it enumerates all D in τ_1, and stores its count in $\mathbf{X}_{\tau_1}(B, C)$. ❸ Given f_{AB}, it enumerates all E in τ_2, and updates its count in $\mathbf{X}_{\tau_1}(A, C)$ regarding f_{AB} using the count done in ❷. ❹ For f_A in τ_2, it repeats ❷ and ❸ to compute its final count in $\mathbf{X}_{\tau_2}(A, C)$ regarding f_A. ❺ It indicates that the final count is stored in $\mathbf{X}_{\tau_2}(A, C)$ for f_A. ❻ In τ_3, with the count of $\mathbf{X}_{\tau_2}(A, C)$ for f_A done in ❺, it enumerates F and updates its count regarding the orbit of p. ❼ By repeating ❶-❻ for all possible matches of f_A, it gets the final count for the orbit o. Disc repeats it for every possible match of f_A by maintaining it with 2 counts. Here, the number of counts need to be maintained is $|V_T| - 1$, which is considered as a constant.

Disc is space-efficient. However, the cost of computing T is high, which is related to the number of iterations. The number of iterations depends on $\alpha = |\bigcup_i V_C(\tau_i)|$ over a path in T. In Example 6.1, it is $|V_C(\tau_2) \cup V_C(\tau_1)| = |A \cup B|$, and the number of exploring τ_1 in the data graph G is $O(n^\alpha)$. We propose a new $O(m)$-approach regarding the memory, which we call Scope. The main idea is to reduce α by only using a part of $V_C(\tau_i)$, while keeping the other part of $V_C(\tau_i)$ in memory. The memory used is bounded by $O(m)$. We give the algorithm in Algorithm 4, and explain it using Example 6.1.

As given in Example 6.1, we have $A = A_1, A_2$, and $B = B_1, B_2$. Suppose $|A_1 \cup B_1| < |A \cup B|$. Here, we take A_2 and B_2 as an edge to ensure $O(m)$ memory. Scope (Algorithm 4) processes it as follows starting from the root τ_3 in T. ❶ It finds a partial match $f_{A_1B_1} = f_{A_1} \| f_{B_1}$ that matches $A_1 \subset V_C(\tau_2)$ in τ_2 and then $B_1 \subset V_C(\tau_1)$ in τ_1. ❷ Given $f_{A_1B_1}$, it enumerates all B_2 and D in τ_1, and stores a count for every f_{B_2} that can expand from $f_{A_1B_1}$ in $\mathbf{X}_{\tau_1}(B_1B_2, C)$. ❸ Given $f_{A_1B_1}$, it enumerates all A_2, B_2, and E in τ_2, and updates the counts in $\mathbf{X}_{\tau_2}(A_1A_2, C)$ regarding f_{A_1}. To update, it needs to find the count done in ❷ by hash-join. As all the matches f_{B_2} given $f_{A_1B_1}$ are distinct, the join cost is constant. ❹ For f_{A_1} in τ_2, it repeats ❷ and ❸ to compute its final count in $\mathbf{X}_{\tau_2}(A_1A_2, C)$ regarding f_{A_1}. ❺ It indicates that the final count is stored in $\mathbf{X}_{\tau_2}(A_1A_2, C)$ regarding f_{A_1}. ❻ In τ_3, with the count of $\mathbf{X}_{\tau_2}(A_1A_2, C)$ regarding f_{A_1} done in ❺, it enumerates A_2 and F and updates its count regarding the orbit of p. ❼ By repeating ❶-❻ for all possible matches of f_{A_1}, it gets the final count for the orbit o. Scope repeats it for every possible match by maintaining $O(m)$ matches and counts supposing A_2 and B_2 are for an edge. As indicated in Fig. 11, when we start processing τ_2 in ❹, we can release the memory used for its child in ❸.

Algorithm 3: Disc(f_i, τ, T)

Input: an i-mapping f_i, pattern graph $\tau = (V_\tau, E_\tau)$, a tree decomposition T
Output: $\mathbf{X}_\tau$

Disc(f_i, τ_j, T) if i is the smallest number for f_i to contain $V_C(\tau_j)$ for every child τ_j of τ in T;
if $i = |V_\tau|$ **then**
  update $\mathbf{X}_\tau$ based on $V_C(\tau)$;
else
  let u_{i+1} be the $(i+1)$-th node in V_τ in order;
  find all $f(u_{i+1})$ matches that can expand from f_i constrained by τ, denoted as $\text{val}(f_i \to u_{i+1})$;
  for *each* v *in* $\text{val}(f_i \to u_{i+1})$ **do**
    Disc($f_i \| v$, τ, T);

Algorithm 4: Scope(f_i, τ, T)

Input: an i-mapping f_i, pattern graph $\tau = (V_\tau, E_\tau)$, a tree decomposition T
Output: $\mathbf{X}_\tau$

let $V_C(\tau_j) = V_C^1(\tau_j) \cup V_C^2(\tau_j)$ where $V_C^1(\tau_j) \cap V_C^2(\tau_j) = \emptyset$ for every child τ_j of τ;
Scope(f_i, τ_j, T) if i is the smallest number for f_i to contain $V_C^1(\tau_j)$ for every child τ_j of τ in T;
if $i = |V_\tau|$ **then**
  update $\mathbf{X}_\tau$ based on $V_C^2(\tau)$ using $\mathbf{X}_{\tau_j}$ for every child τ_j of τ;
else
  let u_{i+1} be the $(i+1)$-th node in V_τ in order;
  find all $f(u_{i+1})$ matches that can expand from f_i constrained by τ, denoted as $\text{val}(f_i \to u_{i+1})$;
  for *each* v *in* $\text{val}(f_i \to u_{i+1})$ **do**
    Scope($f_i \| v$, τ, T);

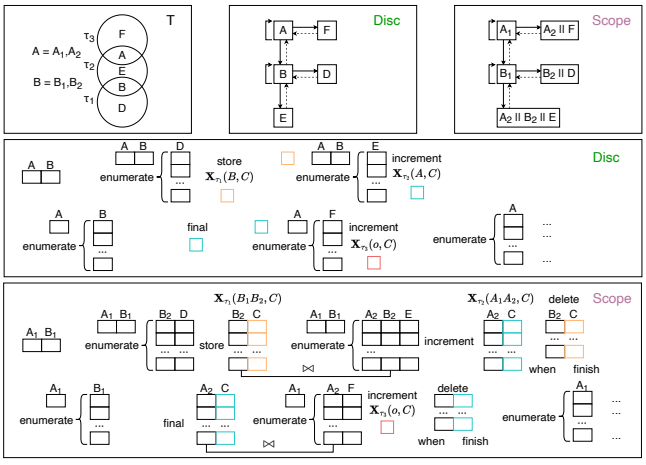


Figure 11: Disc **vs** Scope

Scope is more efficient than Disc. For this example, we have $|A_1 \cup B_1| < |A \cup B|$. As an optimization technique, we share the V_C^1 of a child with the V_C^1 of its parent. In this example, if we make it as $A_1 = B_1$, we have $|A_1 \cup B_1| = |A_1|$, and we only have $O(n^{|A_1|})$ iterations instead of $O(n^{|A_1 \cup B_1|})$. Recall that a similar optimization is to share *SymR*s between a child and its parent to reduce the number of reconstructions. Consider the pattern graph p and the tree decomposition T in Fig. 4. Here, $A_1 = B_1 = \{u_1\}$, $A_2 = \{u_3, u_6\}$, $B_2 = \{u_4, u_6\}$, $D = \{u_5\}$, $E = \emptyset$, and $F = \{u_2\}$. The number of iterations is $O(n^{|A_1|}) = O(n)$, and τ_1 can be processed efficiently. We put the complexity analysis of the three multi-join algorithms in the full version in our GitHub repository.

7 RELATED WORK

Subgraph Counting. The recent survey on subgraph counting [80] outlines three primary approaches to exact subgraph counting: enumeration-based, matrix-based, and decomposition-based. Enumeration-based approaches [33, 43, 44, 50, 52, 67, 71, 81, 97, 98] count by enumerating all matches of the pattern graph in the data graph. Matrix-based approaches [23, 41, 42, 63–65] rely on resolving linear algebra equations, which are grounded in the enumeration of other pattern graphs. JESSE [63–65] is a representative matrix-based approach that can automatically generate and select equations, but it is limited to computing the local counts of all k-node patterns collectively for a specified k. Decomposition-based approaches [4, 60, 62, 73, 74, 101] count p based on enumerating smaller graphs that are obtained by decomposing p. DISC [101], EVOKE [73] and SCOPE are in this category. For approximate subgraph counting, there are sampling-based approaches [11, 14, 15, 17, 18, 29, 30, 37, 46, 51, 56, 75, 78, 94, 95, 99] and learning-based approaches [93, 102]. Our approach is a new decomposition-based approach for general local subgraph counting.

Subgraph Matching. Subgraph matching enumerates iso-matches from the pattern graph to the data graph, with many works founded on Ullmann's backtracking [90] or Leapfrog [91]. Research efforts have been devoted to filtering candidates [10, 12, 19, 33, 38, 39, 53, 54, 87, 88, 103], optimizing matching order [12, 38, 39, 53, 54, 66, 83, 85, 87, 88], and using previous matching results for pruning [7, 38, 47, 53, 54]. Different backtracking algorithms have been proposed [48, 85] to reduce set intersections. There are also distributed approaches that decompose the query into sub-structures and assembly the matches of sub-structures to obtain the pattern's results [58, 77, 79, 82, 96, 100]. Experimental studies for subgraph matching can be found in [59, 86]. We extend symmetry-breaking [33] in subgraph matching to decomposition-based counting.

Worst-case optimal join. The *AGM* bound [8] gives the worst-case output size of a multi-join. In worst-case scenarios, executing a series of binary joins is inefficient since their complexity exceeds the *AGM* bound. Conversely, worst-case optimal join algorithms such as NPRR [69], GenericJoin [70], and Leapfrog [91] have a time complexity that matches this *AGM* bound. Owing to its remarkable efficiency, Leapfrog finds extensive application in both subgraph matching and subgraph counting. Various studies [3, 49, 66, 89] have amalgamated the worst-case optimal join with a binary join, steered by the principles of tree decomposition for better performance. Based on Leapfrog, Disc [101] handles aggregations in tree decompositions, and our Scope improves Disc.

Table 2: The 12 datasets

Graph	Notation	|V|	|E|	avg. degree
web-spam	WS	4.8×10^3	3.7×10^4	15.7
rec-movielens-user-movies-10m	RM	7.6×10^3	5.5×10^4	14.6
bio-grid-yeast	BY	6.0×10^3	1.6×10^5	52.2
ca-AstroPh	CA	1.9×10^4	2.0×10^5	21.1
rec-github	RG	1.2×10^5	4.4×10^5	7.2
soc-gowalla	SG	2.0×10^5	9.5×10^5	9.7
soc-youtube	SY	1.1×10^6	3.0×10^6	5.3
web-wiki-ch-internal	WW	1.9×10^6	9.0×10^6	9.3
web-hudong	WH	2.0×10^6	1.4×10^7	14.6
ca-coauthors-dblp	CC	5.4×10^5	1.5×10^7	56.4
soc-livejournal1	SL	4.8×10^6	4.3×10^7	17.7
soc-orkut-dir	SO	3.1×10^6	1.2×10^8	76.3

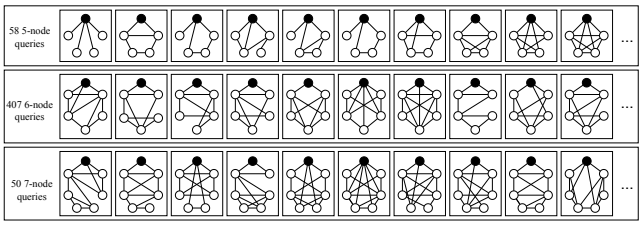


Figure 12: The 515 queries

8 EXPERIMENTS

Algorithms: We implemented SCOPE to compute a local subgraph counting query by tISO and *SymR*s, and evaluate tree decompositions by the Scope (Algorithm 4). To fully understand SCOPE, we have implemented its variants: SCOPE-Td, SCOPE-TSd, and SCOPE-Ts. Here, SCOPE-Td is by tISO without *SymR*s and is evaluated by the Disc algorithm (Algorithm 3), SCOPE-TSd is by both tISO and *SymR*s and is evaluated by the Disc algorithm, and SCOPE-Ts is by tISO without *SymR*s and is evaluated by the Scope algorithm. It is important to note that the Disc algorithm (Algorithm 3) is the algorithm used in DISC [101] to compute the aggregations in a tree decomposition, whereas DISC is the overall algorithm to compute subgraph counts. We also implemented 2 baselines, isoS and DISC1. Here, isoS is to count by directly enumerating iso-matches with *SymR*s. DISC1 is our implementation of DISC [101]. We also compare with EVOKE [73], DISC [101] and JESSE [63–65]. Like SCOPE, EVOKE and DISC are decomposition-based approaches, where EVOKE can only handle k-node pattern graphs for $k \leq 5$, and DISC is a general approach. JESSE is a matrix-based approach on a single machine that can handle any k-node pattern graphs. JESSE can only count all k-node patterns collectively for a specified k. It can not count selected queries. We omit other algorithms since they either only support global counting or are outperformed by DISC and EVOKE, as reported in [73, 101].

12 Datasets: We use 12 data graphs, including web graphs, recommendation networks, biological networks, collaboration networks, and social networks (Table 2). All data graphs are taken from [1, 2]. We deal with all graphs as simple undirected graphs.

515 Queries: We conduct testing using 515 local subgraph counting queries in total, as shown in Fig. 12: all 58 5-node queries, all 407 6-node queries, and 50 7-node queries. We randomly select 50 7-node queries from 2,423 7-node queries whose tw = 3. On average, each selected pattern has 11.7 edges, and uses 10.9 HOM counts

Table 3: Results of the 6-node pattern in Fig. 4

	Enumerated matches ($\times 10^8$)					Elasped Time(seconds)				
	WS	BY	CA	SG	WW	WS	BY	CA	SG	WW
isoS	17.5	372	886	356	3,724	36.3	710	1,286	851	15,200
DISC1	4.6	65.4	113	89.8	1,127	15.8	237	153	572	16,400
SCOPE-Td	4.1	59.8	107	78.7	934	16.0	257	172	542	17,596
SCOPE-TSd	1.9	28.8	51.0	37.7	458	7.4	115	81.3	265	8,871
SCOPE-Ts	1.4	15.7	21.6	29.2	483	6.8	80.3	69.4	159	5,011
SCOPE	0.6	6.7	8.1	12.9	232	3.3	38.2	32.9	80.6	2,536

or 9.6 tISO counts to compute its ISO count. It is challenging. As an indication, the general approach DISC can only handle up to some simple 6-node queries [101], and there is no report published to test all 6-node queries in real-world data graphs.

Settings: We conduct all experiments on a single machine running CentOS 8 with Intel Xeon Silver 4215 32-core 2.5GHz CPU and 128GB memory. SCOPE and the variants, isoS, DISC1 and EVOKE are in C++. All these C++ implementations are compiled by g++ 8.5.0 with -O3 enabled and run with one thread. JESSE is in Java (Java 1.8). We use the default configuration and run it with one thread. DISC is a distributed system built on *Spark* (Spark 2.4.3). To remedy the difference in the programming language, we use the single machine configuration in [101] and run DISC with 32 threads. We also show the results of our C++ implementation DISC1. Like the previous works [63, 73, 101], we report the total time of running all queries in a batch. The time limit is 1 day. We also present the number of matches enumerated. We compare two algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$ by the speedup of $\mathcal{A}_2$ relative to $\mathcal{A}_1$, defined as the ratio of $\mathcal{A}_1$'s execution time to that of $\mathcal{A}_2$, and the reduction in enumerated matches, defined as the ratio of the number of matches enumerated by $\mathcal{A}_1$ to the number by $\mathcal{A}_2$. We put the results for memory usage, preprocessing time and scalability tests in the full version.

8.1 A Simple Case Study: HOM, ISO, or tISO

As a case study to start, we consider a local subgraph counting query $Q = (p, o)$, where p is the 6-node pattern graph in Fig. 4 and $o = u_1$. We test 3 different approaches, namely, ISO, HOM, and tISO. For ISO, we use isoS which takes an ISO-based approach on p by directly enumerating iso-matches with *SymR*s. For HOM, we use DISC1, which generates 8 distinct trees for p by tree decomposition and processes each of the 8 trees by Disc [101]. For tISO, we use SCOPE and its variants, which generate 6 distinct trees by the tISOToISO algorithm (Algorithm 1). This results in 11 tree nodes, and each of them is a 3/4-graph. The results are presented in Table 3. Decomposition approaches (DISC1, SCOPE, and the variants) enumerate much less matches compared to the enumeration approach isoS. Our SCOPE that combines tree decomposition, *SymR*s, and the Scope algorithm outperforms the others significantly.

8.2 The Three Batches of Queries

We have conducted testing using three batches of queries, namely, all 58 5-node queries, all 407 6-node queries, and 50 selected 7-node queries. The results are shown in Fig. 13, Fig. 14(a), and Fig. 14(c), respectively. Cases where the algorithm exceeded the time or memory limits are excluded from the figures.

All 5-node Queries: As shown in Fig. 13, JESSE can only complete all 5-node queries on WS and CA, and SCOPE outperforms JESSE by more than 2 orders. SCOPE also outperforms DISC. Here, on the one machine setting SCOPE runs using one thread, whereas DISC runs using 32 threads on *Spark*. SCOPE is 133× faster than DISC on average across 6 datasets, while DISC cannot compute the other 6 datasets. The maximum speedup observed is 227× on the WS graph. EVOKE outperforms SCOPE in many cases. The main reason is that EVOKE does its best to deal with each of the 5-node queries in implementation, even though EVOKE takes a simple way to construct a 2-level tree for each p. However, the differences between EVOKE and SCOPE are not big. On average, EVOKE is about 1.9× faster than SCOPE. But note that SCOPE is better than EVOKE in the three largest data graphs. SCOPE can compute all 5-node queries on SO, but EVOKE cannot. Also, EVOKE cannot support k-node queries when $k > 5$.

All 6-node Queries: For all 407 6-node queries in Fig. 14(a), only SCOPE can compute the batch of all 407 6-node queries. EVOKE does not support $k = 6$. JESSE and DISC run out of memory. To compare with DISC, we referred to our implementation DISC1 in Fig. 14(a). SCOPE outperforms all variants and DISC1 significantly. We also select 50 of the 6-node queries whose tw are 3. The results are in Fig. 14(b). SCOPE consistently outperforms DISC by more than 1 order of magnitude, in terms of both the elapsed time and the number of enumerated matches.

The 50 7-node Queries: The results are shown in Fig. 14(c). Like in the batch of 6-node queries, DISC can not run these 7-node queries due to memory exhaustion, so we use DISC1 in Fig. 14(c). SCOPE significantly outperforms the other variants and DISC1.

8.3 Effect of Proposed Techniques

tISO-based counting: We compare DISC1 and SCOPE-Td. Here, the former counts by HOM and the latter counts by tISO. Both use tree decomposition and use Disc to process each tree. SCOPE-Td does not use *SymR*. SCOPE-Td consistently performs better, showing average speedups of 1.1×. Also, the average reductions in enumerated matches are 1.3×, 1.4×, and 1.3×, respectively for $k = 5, 6, 7$. On one hand, ISO has an overhead for checking the injectivity of mappings. On the other hand, it benefits from having fewer iso-matches, and there are less trees to compute in tISO. Also, note that tISO can be used with *SymR*s, whereas HOM cannot. Overall, our experiments show that tISO performs better than HOM.

The symmetry rules: We investigate the benefit of symmetry rules by comparing SCOPE-Ts (without *SymR*s) and SCOPE (with *SymR*s). SCOPE significantly outperforms SCOPE-Ts. Take the CA graph as an example. SCOPE is 18.3× faster than SCOPE-Ts for the total time of the 5-node batch query, 10.9× faster for the 6-node batch query, and 5.7× faster for the 7-node batch query. To further investigate the effectiveness of *SymR*s in different queries, we run all 6-node queries separately and compare SCOPE-Ts and SCOPE in Fig. 15. Here, we use a scatter plot where the x-axis is the density of each pattern, the colored points denote the tw, and the y-axis is the speedup/reduction in enumerated matches. The average speedup of the 407 individual queries in BY, CA, and RG are 7.2×, 5.8×, and 7.0×, respectively, and the average reduction in enumerated matches are 5.8×, 6.4×, and 5.3×, respectively. We observe that the speedup and reduction in enumerated matches are more significant for queries with larger tw. A tree node is likely to

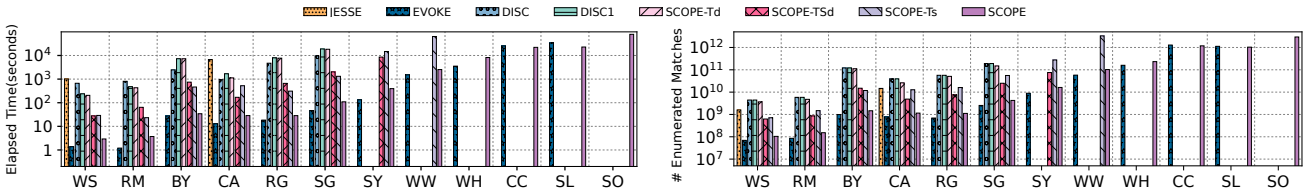
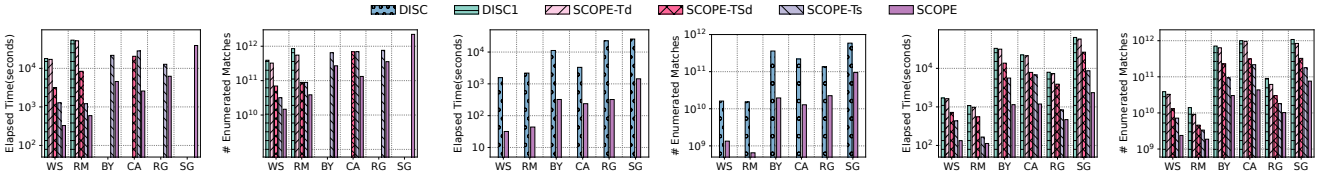
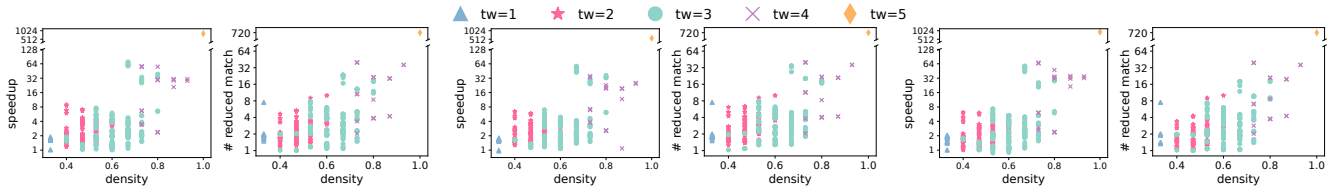


Figure 13: The batch with 58 5-node queries



(a) The batch with 407 6-node queries (b) The batch with 50 6-node queries (c) The batch with 50 selected 7-node queries

Figure 14: 6-node and 7-node queries



(a) BY (b) CA (c) SG

Figure 15: SCOPE **vs** SCOPE-Ts **over the 407 individual 6-node queries**

Table 4: Mean absolute error (MAE) for the ZINC dataset

model	GCN	GraphSage	GAT	MoNet	GatedGCN
Baseline	.356±.011	.455±.023	.464±.005	.260±.008	.340±.006
$\mathcal{F}$-MPNN	.198±.003	.235±.005	.209±.006	.190±.002	.135±.010
$\mathcal{F}^+$-MPNN	**.190±.021**	**.226±.014**	**.200±.003**	**.168±.011**	**.126±.009**

induce a dense subgraph since it can not be further partitioned. If the tw is large, then there exist some large and dense tree nodes in the tree decomposition, and there are many automorphism orbits and *SymR*s to be used. Note that a large tw means that the query is difficult to compute, so hard queries benefit more from *SymR*s.

The multi-join algorithms: We discussed two algorithms, namely, Disc (Algorithm 3) and Scope (Algorithm 4). We study the efficiency of the two by comparing SCOPE-TSd and SCOPE. The primary distinction lies in the algorithmic choice; SCOPE-TSd uses Disc, while SCOPE uses the Scope algorithm. In the three batch queries, SCOPE is more effective than SCOPE-TSd. For example, for the 6-node queries, SCOPE-TSd can only finish in 3 graphs, where SCOPE is 10.6× faster on average. This is due to the fact that Scope enumerates much less matches. SCOPE-TSd enumerates more matches since it can repeatedly enumerate the same tree nodes.

8.4 Applying Subgrpah Counts to GNN

In [9], the authors show that augmenting node feature with local subgraph counts can increase the expressive power of GNNs theoretically and empirically. Here, we further augment node features with 5-node and 6-node local subgraph counts that they do not use. We conducted extensive experimental studies to study the two tasks conducted in [9], with five GNN architectures: GCN [55], GraphSage [36], GAT [92], MoNet [68], and GatedGCN [16], following the settings [9]. We study predicting the solubility of molecules in the ZINC dataset [27]. It has 12,000 graphs and each graph is a particular molecule. Table 4 shows the results. Here, Baseline uses atom types as node features. $\mathcal{F}$-MPNN [9] adds 3-10 cycle counts to node features, and we further add 95 non-zero counts taken from all 5/6-node patterns that $\mathcal{F}$-MPNN does not use, denoted as $\mathcal{F}^+$-MPNN. $\mathcal{F}^+$-MPNN has the smallest mean absolute error in all cases, and the improvement over the Baseline is significant. We also studied the node classification task in the full version.

9 CONCLUSION

We propose a novel decomposition-based approach for local subgraph counting, $Q = (p, o)$, by tree-decomposition-based counting (tISO-based counting), which can handle any k-node pattern graph p with the node orbit o. We confirm the efficiency of tISO-based counting by comparing our SCOPE with two state-of-the-art approaches, EVOKE and DISC, using 12 large datasets. EVOKE only supports pattern graphs up to 5 nodes, and SCOPE outperforms EVOKE in 3 large data graphs. For the batch of 5-node queries, SCOPE is 133 times faster than DISC on average over 6 datasets, while DISC cannot compute the other 6 datasets in the given time limit. For the batch of all 407 6-node queries, SCOPE is the only one that can compute on real large graphs in the given time limit.

ACKNOWLEDGEMENT

This work was supported by the Research Grants Council of Hong Kong, China, No.14205520.

REFERENCES

[1] Network Repository. last accessed: 18/04/2024. https://networkrepository.com.

[2] SNAP: Large Network Dataset Collection. last accessed: 18/04/2024. https://snap.stanford.edu/data/index.html.

[3] C. R. Aberger, S. Tu, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. In *SIGMOD*, pages 431–446. ACM, 2016.

[4] N. K. Ahmed, J. Neville, R. A. Rossi, and N. G. Duffield. Efficient graphlet counting for large networks. In *2015 IEEE International Conference on Data Mining, ICDM 2015, Atlantic City, NJ, USA, November 14-17, 2015*, pages 1–10. IEEE Computer Society, 2015.

[5] O. Amini, F. V. Fomin, and S. Saurabh. Counting subgraphs via homomorphisms. In *Proc. of ICALP'09*, pages 71–82, 2009.

[6] K. Ammar, F. McSherry, S. Salihoglu, and M. Joglekar. Distributed Evaluation of Subgraph Queries Using Worst-case Optimal Low-memory Dataflows. *Proc. VLDB Endow.*, 11(6):691–704, 2018.

[7] J. Arai, Y. Fujiwara, and M. Onizuka. Gup: Fast subgraph matching by guard-based pruning. *Proceedings of the ACM on Management of Data*, 1(2):1–26, 2023.

[8] A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, October 25-28, 2008, Philadelphia, PA, USA*, pages 739–748. IEEE Computer Society, 2008.

[9] P. Barceló, F. Geerts, J. L. Reutter, and M. Ryschkov. Graph neural networks with local graph parameters. In *NeurIPS*, pages 25280–25293, 2021.

[10] B. Bhattarai, H. Liu, and H. H. Huang. CECI: compact embedding cluster index for scalable subgraph matching. In *SIGMOD*, pages 1447–1462. ACM, 2019.

[11] M. Bhuiyan, M. Rahman, M. Rahman, and M. A. Hasan. GUISE: uniform sampling of graphlets for large graph analysis. In *12th IEEE International Conference on Data Mining, ICDM 2012, Brussels, Belgium, December 10-13, 2012*, pages 91–100. IEEE Computer Society, 2012.

[12] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang. Efficient subgraph matching by postponing cartesian products. In *SIGMOD*, pages 1199–1214. ACM, 2016.

[13] C. Borgs, J. Chayes, L. Lovász, V. T. Sós, and K. Vesztergombi. Counting graph homomorphisms. In *Topics in Discrete Mathematics*, pages 315–371. 2006.

[14] M. Bressan, F. Chierichetti, R. Kumar, S. Leucci, and A. Panconesi. Motif counting beyond five nodes. *ACM Trans. Knowl. Discov. Data*, 12(4):48:1–48:25, 2018.

[15] M. Bressan, S. Leucci, and A. Panconesi. Motivo: Fast motif counting via succinct color coding and adaptive sampling. *Proc. VLDB Endow.*, 12(11):1651–1663, 2019.

[16] X. Bresson and T. Laurent. Residual gated graph convnets. *CoRR*, abs/1711.07553, 2017.

[17] X. Chen, Y. Li, P. Wang, and J. C. S. Lui. A general framework for estimating graphlet statistics via random walk. *Proc. VLDB Endow.*, 10(3):253–264, 2016.

[18] X. Chen and J. C. S. Lui. Mining graphlet counts in online social networks. *ACM Trans. Knowl. Discov. Data*, 12(4):41:1–41:38, 2018.

[19] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10):1367–1372, 2004.

[20] R. Curticapean, H. Dell, and D. 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, Montreal, QC, Canada, June 19-23, 2017*, pages 210–223. ACM, 2017.

[21] M. Cygan, F. V. Fomin, L. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. *Parameterized Algorithms*. Springer, 2015.

[22] V. Dalmau and P. Jonsson. The complexity of counting homomorphisms seen from the other side. *Theoretical Computer Science*, 329(1-3):315–323, 2004.

[23] V. S. Dave, N. K. Ahmed, and M. A. Hasan. E-clog: Counting edge-centric local graphlets. In *2017 IEEE International Conference on Big Data (IEEE BigData 2017), Boston, MA, USA, December 11-14, 2017*, pages 586–595. IEEE Computer Society, 2017.

[24] R. M. F. David S. Dummit. *Abstract Algebra*. Wiley, 3 edition, 2003.

[25] J. Díaz, M. Serna, and D. M. Thilikos. Counting h-colorings of partial k-trees. *Theoretical Computer Science*, 281(1-2):291–309, 2002.

[26] V. V. dos Santos Dias, C. H. C. Teixeira, D. O. Guedes, W. M. Jr., and S. Parthasarathy. Fractal: A general-purpose graph pattern mining system. In *SIGMOD*, pages 1357–1374. ACM, 2019.

[27] V. P. Dwivedi, C. K. Joshi, A. T. Luu, T. Laurent, Y. Bengio, and X. Bresson. Benchmarking graph neural networks. *J. Mach. Learn. Res.*, 24:43:1–43:48, 2023.

[28] M. Dyer and C. Greenhill. The complexity of counting graph homomorphisms. In *Proc. of SODA'00*, pages 246–255, 2000.

[29] E. R. Elenberg, K. Shanmugam, M. Borokhovich, and A. G. Dimakis. Beyond Triangles: A Distributed Framework for Estimating 3-profiles of Large Graphs. In *SIGKDD*, pages 229–238, 2015.

[30] E. R. Elenberg, K. Shanmugam, M. Borokhovich, and A. G. Dimakis. Distributed Estimation of Graph 4-Profiles. In *Proc. of WWW'16*, pages 483–493, 2016.

[31] J. Flum and M. Grohe. *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2006.

[32] G. Gottlob, G. Greco, N. Leone, and F. Scarcello. Hypertree Decompositions: Questions and Answers. In *Proc. of PODS'16*, pages 57–74, 2016.

[33] J. A. Grochow and M. Kellis. Network motif discovery using subgraph enumeration and symmetry-breaking. In *Research in Computational Molecular Biology, 11th Annual International Conference, RECOMB 2007, Oakland, CA, USA, April 21-25, 2007, Proceedings*, volume 4453 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2007.

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

[35] W. Guo, Y. Li, M. Sha, B. He, X. Xiao, and K. Tan. Gpu-accelerated subgraph enumeration on partitioned graphs. In *SIGMOD*, pages 1067–1082. ACM, 2020.

[36] W. L. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 1024–1034, 2017.

[37] G. Han and H. Sethu. Waddling random walk: Fast and accurate mining of motif statistics in large graphs. In *IEEE 16th International Conference on Data Mining, ICDM 2016, December 12-15, 2016, Barcelona, Spain*, pages 181–190. IEEE Computer Society, 2016.

[38] M. Han, H. Kim, G. Gu, K. Park, and W.-S. Han. Efficient Subgraph Matching: Harmonizing Dynamic Programming, Adaptive Matching Order, and Failing Set Together. In *SIGMOD*, pages 1429–1446, 2019.

[39] W.-S. Han, J. Lee, and J.-H. Lee. Turboiso: Towards Ultrafast and Robust Subgraph Isomorphism Search in Large Graph Databases. In *SIGMOD*, pages 337–348, 2013.

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

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

[42] T. Hočevar and J. Demšar. Combinatorial algorithm for counting small induced graphs and orbits. *PloS one*, 12(2):e0171428, 2017.

[43] M. Houbraken, S. Demeyer, T. Michoel, P. Audenaert, D. Colle, and M. Pickavet. The index-based subgraph matching algorithm with general symmetries (ismags): exploiting symmetry for faster subgraph enumeration. *PloS one*, 9(5):e97896, 2014.

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

[45] K. Jamshidi, R. Mahadasa, and K. Vora. Peregrine: a pattern-aware graph mining system. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 13:1–13:16. ACM, 2020.

[46] M. Jha, C. Seshadhri, and A. Pinar. Path sampling: A fast and provable method for estimating 4-vertex subgraph counts. In *Proceedings of the 24th International Conference on World Wide Web, WWW 2015, Florence, Italy, May 18-22, 2015*, pages 495–505. ACM, 2015.

[47] X. Jian, Z. Li, and L. Chen. Suff: Accelerating subgraph matching with historical data. *Proc. VLDB Endow.*, 16(7):1699–1711, 2023.

[48] T. Jin, B. Li, Y. Li, Q. Zhou, Q. Ma, Y. Zhao, H. Chen, and J. Cheng. Circinus: Fast redundancy-reduced subgraph matching. *Proceedings of the ACM on Management of Data*, 1(1):1–26, 2023.

[49] O. Kalinsky, Y. Etsion, and B. Kimelfeld. Flexible caching in trie joins. *arXiv preprint arXiv:1602.08721*, 2016.

[50] Z. R. M. Kashani, H. Ahrabian, E. Elahi, A. Nowzari-Dalini, E. S. Ansari, S. Asadi, S. Mohammadi, F. Schreiber, and A. Masoudi-Nejad. Kavosh: a new algorithm for finding network motifs. *BMC Bioinform.*, 10:318, 2009.

[51] N. Kashtan, S. Itzkovitz, R. Milo, and U. Alon. Efficient sampling algorithm for estimating subgraph concentrations and detecting network motifs. *Bioinform.*, 20(11):1746–1758, 2004.

[52] S. Khakabimamaghani, I. Sharafuddin, N. Dichter, I. Koch, and A. Masoudi-Nejad. Quatexelero: an accelerated exact network motif detection algorithm. *PloS one*, 8(7):e68073, 2013.

[53] H. Kim, Y. Choi, K. Park, X. Lin, S. Hong, and W. Han. Versatile equivalences: Speeding up subgraph query processing and subgraph matching. In *SIGMOD*, pages 925–937. ACM, 2021.

[54] H. Kim, Y. Choi, K. Park, X. Lin, S.-H. Hong, and W.-S. Han. Fast subgraph query processing and subgraph matching via static and dynamic equivalences. *The VLDB journal*, 32(2):343–368, 2023.

[55] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[56] T. G. Kolda, A. Pinar, and C. Seshadhri. Triadic measures on graphs: The power of wedge sampling. In *Proceedings of the 13th SIAM International Conference on Data Mining, May 2-4, 2013. Austin, Texas, USA*, pages 10–18. SIAM, 2013.

[57] L. Lai, L. Qin, X. Lin, and L. Chang. Scalable Subgraph Enumeration in MapReduce. *Proc. VLDB Endow.*, 8(10):974–985, 2015.

[58] L. Lai, L. Qin, X. Lin, and L. Chang. Scalable subgraph enumeration in mapreduce: a cost-oriented approach. *VLDB J.*, 26(3):421–446, 2017.

[59] L. Lai, Z. Qing, Z. Yang, X. Jin, Z. Lai, R. Wang, K. Hao, X. Lin, L. Qin, W. Zhang,

Y. Zhang, Z. Qian, and J. Zhou. Distributed subgraph matching on timely dataflow. *Proc. VLDB Endow.*, 12(10):1099–1112, 2019.

[60] D. Marcus and Y. Shavitt. Efficient counting of network motifs. In *30th IEEE International Conference on Distributed Computing Systems Workshops (ICDCS 2010 Workshops), 21-25 June 2010, Genova, Italy*, pages 92–98. IEEE Computer Society, 2010.

[61] D. Marx and M. Pilipczuk. Everything you always wanted to know about the parameterized complexity of subgraph isomorphism (but were afraid to ask). In *31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014), STACS 2014, March 5-8, 2014, Lyon, France*, volume 25 of *LIPIcs*, pages 542–553, 2014.

[62] L. A. A. Meira, V. R. Máximo, Á. L. Fazenda, and A. F. da Conceição. acc-motif: Accelerated network motif detection. *IEEE ACM Trans. Comput. Biol. Bioinform.*, 11(5):853–862, 2014.

[63] I. Melckenbeeck, P. Audenaert, D. Colle, and M. Pickavet. Efficiently counting all orbits of graphlets of any order in a graph using autogenerated equations. *Bioinform.*, 34(8):1372–1380, 2018.

[64] I. Melckenbeeck, P. Audenaert, T. Michoel, D. Colle, and M. Pickavet. An algorithm to automatically generate the combinatorial orbit counting equations. *PloS one*, 11(1):e0147078, 2016.

[65] I. Melckenbeeck, P. Audenaert, T. V. Parys, Y. V. de Peer, D. Colle, and M. Pickavet. Optimising orbit counting of arbitrary order by equation selection. *BMC Bioinform.*, 20(1):27:1–27:13, 2019.

[66] A. Mhedhbi and S. Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proc. VLDB Endow.*, 12(11):1692–1704, 2019.

[67] 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.

[68] F. Monti, D. Boscaini, J. Masci, E. Rodolà, J. Svoboda, and M. M. Bronstein. Geometric deep learning on graphs and manifolds using mixture model cnns. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 5425–5434. IEEE Computer Society, 2017.

[69] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case Optimal Join Algorithms: [Extended Abstract]. In *Proc. of PODS'12*, pages 37–48, 2012.

[70] H. Q. Ngo, C. Ré, and A. Rudra. Skew strikes back: new developments in the theory of join algorithms. *ACM SIGMOD Record*, 42(4):5–16, 2014.

[71] P. Paredes and P. M. P. Ribeiro. Towards a faster network-centric subgraph census. In *Advances in Social Networks Analysis and Mining 2013, ASONAM '13, Niagara, ON, Canada - August 25 - 29, 2013*, pages 264–271. ACM, 2013.

[72] H.-M. Park, S.-H. Myaeng, and U. Kang. Pte: Enumerating trillion triangles on distributed systems. In *SIGKDD*, pages 1115–1124, 2016.

[73] N. Pashanasangi and C. Seshadhri. Efficiently counting vertex orbits of all 5-vertex subgraphs, by EVOKE. In *WSDM '20: The Thirteenth ACM International Conference on Web Search and Data Mining, Houston, TX, USA, February 3-7, 2020*, pages 447–455. ACM, 2020.

[74] A. Pinar, C. Seshadhri, and V. Vishal. ESCAPE: efficiently counting all 5-vertex subgraphs. In *Proceedings of the 26th International Conference on World Wide Web, WWW 2017, Perth, Australia, April 3-7, 2017*, pages 1431–1440. ACM, 2017.

[75] N. Przulj, D. G. Corneil, and I. Jurisica. Efficient estimation of graphlet frequency distributions in protein-protein interaction networks. *Bioinform.*, 22(8):974–980, 2006.

[76] C. Qian, G. Rattan, F. Geerts, M. Niepert, and C. Morris. Ordered subgraph aggregation networks. In *NeurIPS*, 2022.

[77] M. Qiao, H. Zhang, and H. Cheng. Subgraph Matching: On Compression and Computation. *Proc. VLDB Endow.*, 11(2):176–188, 2017.

[78] M. Rahman, M. A. Bhuiyan, and M. A. Hasan. Graft: An efficient graphlet counting method for large graph analysis. *IEEE Trans. Knowl. Data Eng.*, 26(10):2466–2478, 2014.

[79] X. Ren, J. Wang, W. Han, and J. X. Yu. Fast and robust distributed subgraph enumeration. *Proc. VLDB Endow.*, 12(11):1344–1356, 2019.

[80] P. Ribeiro, P. Paredes, M. E. P. Silva, D. Aparício, and F. M. A. Silva. A survey on subgraph counting: Concepts, algorithms, and applications to network motifs and graphlets. *ACM Comput. Surv.*, 54(2):28:1–28:36, 2022.

[81] P. M. P. Ribeiro and F. M. A. Silva. g-tries: an efficient data structure for discovering network motifs. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26, 2010*, pages 1559–1566. ACM, 2010.

[82] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *SIGMOD*, pages 23–34, 1979.

[83] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming Verification Hardness: An Efficient Algorithm for Testing Subgraph Isomorphism. *Proc. VLDB Endow.*, 1(1):364–375, 2008.

[84] T. Shi, M. Zhai, Y. Xu, and J. Zhai. Graphpi: high performance graph pattern matching through effective redundancy elimination. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, page 100. IEEE/ACM, 2020.

[85] S. Sun, Y. Che, L. Wang, and Q. Luo. Efficient parallel subgraph enumeration on a single machine. In *ICDE*, pages 232–243. IEEE, 2019.

[86] S. Sun and Q. Luo. In-memory subgraph matching: An in-depth study. In *SIGMOD*, pages 1083–1098. ACM, 2020.

[87] S. Sun and Q. Luo. Subgraph matching with effective matching order and indexing. *IEEE Trans. Knowl. Data Eng.*, 34(1):491–505, 2022.

[88] S. Sun, X. Sun, Y. Che, Q. Luo, and B. He. Rapidmatch: A holistic approach to subgraph query processing. *Proc. VLDB Endow.*, 14(2):176–188, 2020.

[89] S. Tu and C. Ré. Duncecap: Query plans using generalized hypertree decompositions. In *SIGMOD*, pages 2077–2078, 2015.

[90] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.

[91] T. L. Veldhuizen. Leapfrog triejoin: a worst-case optimal join algorithm. *CoRR*, abs/1210.0481, 2012.

[92] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph attention networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[93] H. Wang, R. Hu, Y. Zhang, L. Qin, W. Wang, and W. Zhang. Neural subgraph counting with wasserstein estimator. In *SIGMOD*, pages 160–175. ACM, 2022.

[94] P. Wang, Y. Qi, J. C. S. Lui, D. Towsley, J. Zhao, and J. Tao. Inferring higher-order structure statistics of large networks from sampled edges. *IEEE Trans. Knowl. Data Eng.*, 31(1):61–74, 2019.

[95] P. Wang, J. Zhao, X. Zhang, Z. Li, J. Cheng, J. C. S. Lui, D. Towsley, J. Tao, and X. Guan. MOSS-5: A fast method of approximating counts of 5-node graphlets in large graphs. *IEEE Trans. Knowl. Data Eng.*, 30(1):73–86, 2018.

[96] Z. Wang, R. Gu, W. Hu, C. Yuan, and Y. Huang. BENU: distributed subgraph enumeration with backtracking-based framework. In *ICDE*, pages 136–147. IEEE, 2019.

[97] S. Wernicke. A faster algorithm for detecting network motifs. In *Algorithms in Bioinformatics, 5th International Workshop, WABI 2005, Mallorca, Spain, October 3-6, 2005, Proceedings*, volume 3692 of *Lecture Notes in Computer Science*, pages 165–177. Springer, 2005.

[98] S. Wernicke and F. Rasche. FANMOD: a tool for fast network motif detection. *Bioinform.*, 22(9):1152–1153, 2006.

[99] C. Yang, M. Lyu, Y. Li, Q. Zhao, and Y. Xu. SSRW: A scalable algorithm for estimating graphlet statistics based on random walk. In *Database Systems for Advanced Applications - 23rd International Conference, DASFAA 2018, Gold Coast, QLD, Australia, May 21-24, 2018, Proceedings, Part I*, volume 10827 of *Lecture Notes in Computer Science*, pages 272–288. Springer, 2018.

[100] Z. Yang, L. Lai, X. Lin, K. Hao, and W. Zhang. HUGE: an efficient and scalable subgraph enumeration system. In *SIGMOD*, pages 2049–2062. ACM, 2021.

[101] H. Zhang, J. X. Yu, Y. Zhang, K. Zhao, and H. Cheng. Distributed subgraph counting: A general approach. *Proc. VLDB Endow.*, 13(11):2493–2507, 2020.

[102] K. Zhao, J. X. Yu, H. Zhang, Q. Li, and Y. Rong. A learned sketch for subgraph counting. In *SIGMOD*, pages 2142–2155. ACM, 2021.

[103] P. Zhao and J. Han. On Graph Query Optimization in Large Networks. *Proc. VLDB Endow.*, 3(1-2):340–351, 2010.

Appendix

COMPLEXITY ANALYSIS

Table 5: Comparing the complexity of three algorithms

Algorithm	Time complexity	Space complexity
Eq. (4)-Eq. (7)	$O(n^{\lvert V(\tau)\rvert})$	$O(n^{\lvert V_C(\tau)\rvert})$
Disc	$O(n^{\lvert V(\tau)\cup_{\tau'\in anc(\tau)} V_C(\tau')\rvert})$	$O(1)$
Scope	$O(n^{\lvert V(\tau)\cup_{\tau'\in anc(\tau)} V_C^1(\tau')\rvert})$	$O(m)$

We summarize the time and space complexity of the three multi-join algorithms discussed in this paper in Table 5. Here, the complexity is for computing a single tree node τ, and the complexity for computing a tree or all trees in an equation is the largest complexity of all tree nodes. $anc(\tau)$ are the ancestors of τ. The complexity of Eq. (4)-Eq. (7) and the space complexity of all algorithms follow the previous discussions in Section 6. In Algorithm 3, we call the algorithm for τ only when the V_C of the ancestors and itself are matched. There are $O(n^{\sum_{\tau'\in anc(\tau)\cup\tau} |V_C(\tau')|})$ matches of the V_C of ancestors and itself. For each match, we need to compute matches for the vertices in $V(\tau) \setminus V_C(\tau)$ whose time complexity is $O(n^{|V(\tau)\setminus V_C(\tau)|})$. Combining the two parts, the time complexity of Disc is $O(n^{|V(\tau)\cup_{\tau'\in anc(\tau)} V_C(\tau')|})$. Similarly, the time complexity of Scope is $O(n^{|V(\tau)\cup_{\tau'\in anc(\tau)} V_C^1(\tau')|})$. Since $\cup_{\tau'\in anc(\tau)} V_C^1(\tau') \subset \cup_{\tau'\in anc(\tau)} V_C(\tau')$, Scope has a smaller time complexity than Disc. Besides, by sharing the V_C^1 of τ and ancestors, we can achieve the optimal time complexity $O(n^{|V(\tau)|})$.

ADDITIONAL EXPERIMENTS

Table 6: The peak memory usage(GB) of 5-node queries

	WS	RM	BY	CA	RG	SG	SY	WW	WH	CC	LJ	SO
EVOKE	0.05	0.07	0.2	0.27	0.59	1.3	4.2	12	18	27	58	-
DISC	61	61	68	62	71	84	-	-	-	-	-	-
SCOPE	0.02	0.03	0.06	0.09	0.21	0.42	1.6	3.9	5.5	9.3	18	40

Memory usage. Table 6 shows the peak memory usage of SCOPE, DISC and EVOKE using 5-node queries. DISC has the highest memory usage and it runs out of memory on 6 graphs. EVOKE has a relatively high memory cost and it runs out of memory on the SO graph, although its code is carefully optimized. Our SCOPE has the smallest space cost. For the 407 6-node queries and 50 7-node queries, SCOPE's memory costs are 0.94GB, 1.42GB, respectively, on the SG graph. We note that DISC's high memory cost is an implementation issue. As in Table 5, the main memory cost of Disc comes from storing the input graph and the counts for each tree since it only needs constant space for intermediate counts. Scope needs to store additional intermediate counts with $O(m)$ space. Our implementation DISC1 has a smaller memory cost than SCOPE.

Preprocessing time. Table 7 shows the preprocessing time of SCOPE. We generate a query plan for each query graph, which consists of an equation and tree decompositions for each pattern in the equation(Algorithm 1), compute *SymR*s, and choose V_C and attribute order for each tree node. It is much shorter than the total time in Fig. 13 and Fig. 14. Note that the plans only depend on the query, so we can also pre-compute and reuse the plans for different data graphs to eliminate this preprocessing time. We also note that our planning is much more efficient than DISC. DISC takes 56 seconds for 5-node queries, 1,255 seconds for 6-node queries, and 13,533 seconds for 7-node queries. We observe that the time for tree decomposition dominates when $k = 6$ or 7. This is because we use a brute-force approach to enumerate and check all tree decompositions. To scale to larger query graphs, we can reduce this time by only finding one decomposition with minimum width instead of checking all decompositions.

Table 7: The preprocessing time of 3 query sets

		time (seconds)				
k	# queries	TD	equation	symmetry	V_C, attr order	total
5	58	0.045	0.015	0.002	0.067	0.129
6	407	10.449	0.193	0.029	0.728	11.399
7	50	52.258	0.156	0.019	0.425	52.859

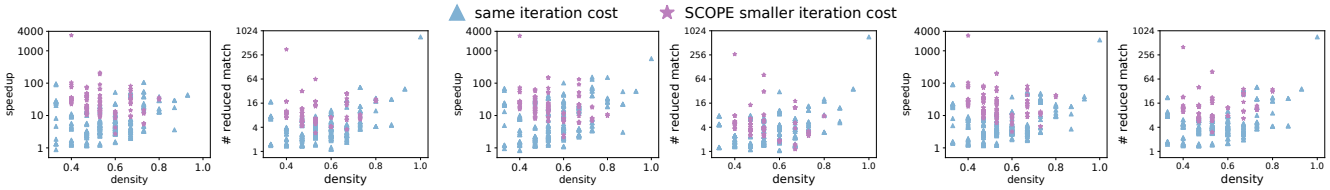
Table 8: Accuracy for the PATTERN dataset

model	GCN	GraphSage	GAT	MoNet	GatedGCN
Baseline	73.67±0.22	70.92±0.123	78.95±0.12	85.78±0.04	85.61±0.05
$\mathcal{F}$-MPNN	79.66±1.65	85.74±0.37	84.87±0.22	86.54±0.02	85.72±0.38
$\mathcal{F}^+$-MPNN	**84.17±1.25**	**86.20±0.04**	**86.10±0.33**	**86.56±0.03**	**86.03±0.04**

Scalability Test. Fig. 17(a) shows the running time of three approaches with respect to the number of edges in the SO graph. We randomly divided the edges of SO into 5 equally sized groups and created 5 graphs for the cases of 20%, 40%, 60%, 80%, and 100% edges. The $i+1$-th graph contains the i-th graph. DISC runs out of memory in all cases, and EVOKE runs out of memory when we use 80% or all edges. SCOPE scales better than EVOKE when increasing the data size. Fig. 17(b) shows the running time of SCOPE in three types of queries with sizes 4-9. For cycles, we randomly select o since all nodes are in the same orbit. For paths, we select the end node as o. Wheels are graphs made of a k-1 cycle and one center node connecting to other k-1 nodes. we select the center node as o. While existing approaches only support (unlabeled) queries with 6 nodes in practice, SCOPE supports queries with 9 nodes. Due to the hard nature of the subgraph counting problem, the running time grows exponentially with respect to the query size.

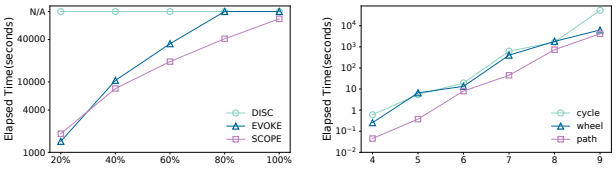
Applying subgraph counts to vertex classification. We study the effect of augmenting node features with local subgraph counts for the vertex classification task. The objective is to predict for every node whether it belongs to a fixed pattern P using PATTERN dataset [27] which has 14,000 graphs. Table 8 shows the results. Baseline uses a random value from $\{0, 1, 2\}$ as a feature. $\mathcal{F}$-MPNN [9] adds 3-5 clique counts, and we further add 464 non-zero counts taken from all 5/6-node patterns that $\mathcal{F}$-MPNN does not use, denoted as $\mathcal{F}^+$-MPNN. $\mathcal{F}^+$-MPNN has a much higher accuracy, outperforming Baseline and $\mathcal{F}$-MPNN in all cases.

SCOPE **vs** DISC1 **over the individual queries.** We compare SCOPE and DISC1 by running each 6-node query separately. The results are shown in Fig. 16. SCOPE has one to two orders of magnitude speedup in most queries, and can be more than three orders of



(a) WS (b) RM (c) CA

Figure 16: SCOPE **vs** DISC1 **over the 407 individual 6-node queries**



(a) varying data size, SO (b) varying query size

Figure 17: Scalability test

magnitude for some queries. the average number of reduction in WS, RM, and CA are 8.0×, 6.6×, and 16.3×, respectively. There are 123 queries where SCOPE has a better time complexity. They are marked with purple stars and other queries are marked with blue triangles. While all queries get enhanced performance due to the tISO and *SymR*s, such queries gain additional advantages from reduced iteration costs. They have a more significant speedup. The average reduction in enumerated matches for them are 19.4×, 14.4×, and 23.5× for WS, RM, and CA, respectively.