

Algebraic Subgraph Counting

QIUYU GUO, University of New South Wales, Australia

JIANYE YANG*, Guangzhou University & PengCheng Laboratory, China

WENJIE ZHANG, University of New South Wales, Australia

HANCHEN WANG, University of Technology Sydney, Australia

YING ZHANG, Zhejiang Gongshang University, China

XUEMIN LIN, The Chinese University of Hong Kong, Shenzhen, China

Subgraph isomorphism counting is a fundamental problem in graph analytics, which aims to find the number of subgraph isomorphisms of a query graph in a data graph. The candidate tree-based framework provides a promising foundation for subgraph counting tasks, offering a unified counting paradigm that can be extended beyond tree patterns. However, supporting subgraph isomorphism within this framework remains challenging, as it requires handling both the non-tree edge constraint and the injective mapping constraint. Although existing solutions employ sampling or learning techniques to address these constraints in this framework, they still either suffer from inherent sampling failures or rely heavily on supervision. In this paper, we propose ASC, an algebraic subgraph counting approach built on the candidate tree-based counting framework. In our method, the non-tree edge constraint is directly incorporated into the candidate tree-based counting process through a matrix-based computation method, enabling subgraph homomorphism counting with high accuracy in polynomial time. Based on the resulting subgraph homomorphism count, we further apply a local sampling method to address the injective mapping constraint, thereby obtaining the final subgraph isomorphism count. Extensive experiments show that ASC can achieve substantially better and more stable performance over the baselines across various datasets, while scaling to billion-edge graphs. Most impressively, as a non-learning method, ASC can even achieve more than an order of magnitude higher average accuracy than the state-of-the-art learning-based method FlowSC with similar efficiency.

CCS Concepts: • **Theory of computation** → **Graph algorithms analysis**; **Approximation algorithms analysis**; • **Mathematics of computing** → *Computations on matrices*.

Additional Key Words and Phrases: Subgraph Counting, Approximation Algorithm, Algebraic Algorithm

ACM Reference Format:

Qiuyu Guo, Jianye Yang, Wenjie Zhang, Hanchen Wang, Ying Zhang, and Xuemin Lin. 2026. Algebraic Subgraph Counting. *Proc. ACM Manag. Data* 4, 4 (SIGMOD), Article 256 (September 2026), 27 pages. <https://doi.org/10.1145/3837094>

1 Introduction

Given a query graph Q and a data graph G , subgraph counting aims to find the number of subgraph isomorphisms of Q in G . It is a fundamental problem with broad applications in query optimization [4, 8, 17, 21], biological network analysis [12, 41, 42, 54], and fraud detection [44, 45]. Despite

*Corresponding Author

Authors' Contact Information: Qiuyu Guo, qiuyu.guo@unsw.edu.au, University of New South Wales, Australia; Jianye Yang, jyyang@gzhu.edu.cn, Guangzhou University & PengCheng Laboratory, China; Wenjie Zhang, University of New South Wales, Australia, wenjie.zhang@unsw.edu.au; Hanchen Wang, University of Technology Sydney, Australia, hanchen.wang@uts.edu.au; Ying Zhang, Zhejiang Gongshang University, China, ying.zhang@zjgsu.edu.cn; Xuemin Lin, The Chinese University of Hong Kong, Shenzhen, China, xuemin.lin@gmail.com.



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 2836-6573/2026/9-ART256

<https://doi.org/10.1145/3837094>

its importance, subgraph counting is computationally challenging due to its #p-hardness [18, 47]. Exact counting via subgraph matching [5, 14, 27, 28, 35, 36, 51, 58] is often intractable due to the inherent complexity of enumeration. Even counting a 32-clique query in a 32-clique graph can exceed a two-hour time limit in our experiments. To broaden applicability, recent efforts focus on approximate counting [18, 30, 39, 47]. In this paper, we focus on approximate subgraph counting.

Existing Solutions and Limitations. Current solutions for approximate subgraph counting can generally be classified into three categories, including summarization-based, sampling-based, and learning-based methods. Summarization-based methods [11, 16, 39, 48] adopt a decomposition-aggregation paradigm, decomposing the query graph into substructures and aggregating their individual counts to estimate the total count. However, their reliance on the independence assumption between substructures often leads to unsatisfactory accuracy in real-world graphs. Sampling-based methods [10, 13, 30, 47] estimate counts via match frequencies in sampled subgraphs. While sampling can achieve high accuracy for simple queries on small graphs, they suffer from inherent sampling failures as the sample space expands. To address this, Fastest [47] employs advanced filtering and candidate tree-based sampling to prune the search space and improve sampling success rates. Nonetheless, eliminating sampling failures remains challenging for complex queries on large graphs. Learning-based methods [18, 24, 55, 60] predict counts by regressing from learned graph features. Earlier works [55, 60] struggle to capture the relationship between the graph features and counts or face training instabilities. The state-of-the-art FlowSC [18] mitigates these issues by simulating data flow within a candidate-tree framework and using a pretraining scheme. However, the performance of these techniques is heavily reliant on the labeled data, which is often impractical for hard query workloads where obtaining labels itself is computationally prohibitive.

Matrix-based Pattern Counting. The connection between matrix computation and pattern counting has deep roots. Early seminal works [3, 25, 33] exploit fast matrix multiplication to exactly count specific structured patterns like small cycles and cliques. The *Functional Aggregate Query* (FAQ) framework [2] provides a unified theoretical foundation for these concepts, formulating general subgraph counting as a sum-product query:

$$\sum_{x_1, \dots, x_k \in V_G} \prod_{(i,j) \in E_Q} E(x_i, x_j), \quad (1)$$

where each variable x_i denotes the mapping of a query vertex to a data vertex, and each factor $E(x_i, x_j)$ enforces edge consistency. Under this framework, matrix multiplication can be seen as a special case of FAQ evaluation, which establishes a natural connection between subgraph counting and matrix-based computation. To solve such queries generally, FAQ proposes InsideOut, an exact algorithm based on variable elimination. Crucially, the FAQ perspective also reveals the fundamental bottleneck shared by all these exact methods: their efficiency is strictly bounded by $O(N^{faqw})$, where N is the data size and $faqw$ is the fractional FAQ-width. While early matrix approaches and InsideOut are effective for low-width topologies, $faqw$ is large for arbitrary or randomly generated subgraphs. Without an exploitable structure, exact evaluation becomes computationally impractical in general, motivating scalable alternatives beyond exact evaluation.

Candidate Tree-based Counting. The candidate tree-based framework supports polynomial-time subgraph homomorphism counting against tree queries [18, 47]. However, there are two major issues in extending it for subgraph isomorphism counting against generic query graphs. *First, it ignores the non-tree edges, i.e., edges in the query graph but not its spanning tree.* Ignoring the matching constraints defined by such edges would lead to significant overestimation. *Second, it cannot handle the injective mapping constraint required by isomorphism.* Because the candidate tree

framework only maintains the matching weights rather than states, it cannot enforce injectivity and thus overestimates the counts, as a data vertex might be used multiple times in a matching.

To apply the candidate tree-based counting framework to subgraph isomorphism counting, existing methods develop effective techniques to handle the non-tree and injective constraints. Fastest treats the tree homomorphism count as an upper bound and adopts sampling to approximate both constraints. FlowSC simulates the data flow pattern in the candidate tree counting process and learns the relationship between graph features and subgraph counts, leveraging query features and supervision to implicitly enforce the constraints. However, these methods often suffer from sampling failures in excessive sampling spaces or from insufficient supervision.

Our Approach and Contributions. In this paper, we propose a candidate tree-based counting method, called Algebraic Subgraph Counting (ASC). Since the candidate tree framework inherently satisfies tree-edge constraints, we decouple the remaining requirements for subgraph isomorphism into *non-tree edge constraints* and *isomorphism (injectivity mapping) constraints*.

To handle the non-tree edges, we divide them into two categories: those connecting vertices with the same parent (Type I) and those connecting vertices with different parents (Type II). We first show that it takes exponential time to compute the exact homomorphism count if the Type-I non-tree edges are preserved in the candidate tree framework. We propose a matrix-based method to incorporate Type-I non-tree edge constraints into candidate tree-based counting, delivering high-quality approximations in polynomial time.

For the Type-II non-tree edge constraint, we have a key observation that the compatibility of the candidates of the two endpoints of a Type-II non-tree edge with a valid matching depends on their lowest common ancestor (LCA) in the tree. We incorporate the Type-II non-tree edge constraints into the candidate tree-based counting process by computing the LCA ratios, which are used to discard the invalid weights accumulated at LCAs in the counting phase. Together with the Type-I non-tree edge constraint computation, this leads to an algebraic method for subgraph homomorphism counting. Based on the resulting homomorphism count, we further apply local sampling to enforce injective mapping constraints, producing the subgraph isomorphism count.

We conduct extensive experiments on 10 real-world datasets, including two billion-edge graphs. The results demonstrate that ASC can achieve consistently better and more stable performance over the existing solutions under the majority of query settings.

2 Preliminary

2.1 Problem Definition

In this paper, we focus on a simple vertex-labeled, undirected, and connected graph $g = (V, E, L, \Sigma)$. Here, V is a set of vertices of g , $E \subseteq V \times V$ is a set of edges of g , L is a labeling function that assigns a label $L(v) \in \Sigma$ to each vertex v in g , and Σ is a set of labels. We use $N(v) = \{v' \in V \mid (v, v') \in E\}$ to denote the neighbors of a vertex v , and $d(v) = |N(v)|$ to denote the degree of v . In this paper, we use $Q = (V_Q, E_Q, L, \Sigma)$ and $G = (V_G, E_G, L, \Sigma)$ to denote the query graph and the data graph, respectively. Besides, we use u to represent a vertex in Q and v to represent a vertex in G .

Definition 2.1 (Subgraph Isomorphism and Homomorphism). Given a query graph Q and a data graph G , subgraph isomorphism is an injective mapping $f : V_Q \mapsto V_G$, such that $\forall u \in V_Q$, $L(u) = L(f(u))$ and $\forall (u, u') \in E_Q$, $(f(u), f(u')) \in E_G$. When injectivity is not required, the mapping f defines a subgraph homomorphism.

In this paper, we focus on subgraph isomorphism. An injective mapping from Q to G is called a *subgraph isomorphic embedding*, also referred to as an *embedding* or a *match* when context is clear.

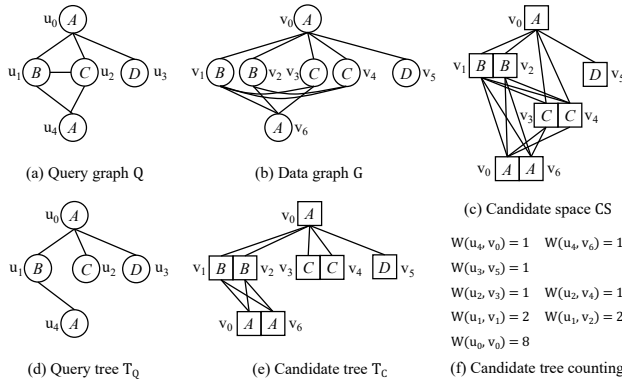


Fig. 1. Running example of candidate tree-based counting.

Definition 2.2 (Subgraph Matching and Counting). Given a query graph Q and a data graph G , subgraph matching is to find all embeddings of Q in G , while subgraph counting is to compute the number of embeddings.

Given the #P-hardness nature of exact subgraph counting, we focus on approximate solutions for subgraph counting in this paper.

Problem Statement. Given a query graph Q and a data graph G , we aim to approximate the number of all embeddings of Q in G .

2.2 Candidate Tree-based Counting

Definition 2.3 (Candidate Space). Given a query graph Q and a data graph G , a candidate space CS maintains a candidate vertex (Resp. edge) set for each vertex (Resp. edge) in Q , such that all matches of Q in G are preserved in the solution space defined by CS .

- Given a vertex $u \in V_Q$, its candidate vertex set $C(u) \subseteq V_G$ satisfies that, for any embedding f , $f(u) \in C(u)$ must hold, but a vertex in $C(u)$ is not necessarily mapped to u in any valid embedding.
- Given an edge $(u, u') \in E_Q$, its candidate edge set $C(u, u') \subseteq E_G$ satisfies that, for any embedding f , $(f(u), f(u')) \in C(u, u')$ must hold, but an edge in $C(u, u')$ is not necessarily mapped to (u, u') in any valid embedding.
- Given an edge $(u, u') \in E_Q$ and a candidate vertex $v \in C(u)$, the conditional candidate vertex set $C(u' \mid u, v) \subseteq C(u')$ satisfies that, for any embedding f with $f(u) = v$, $f(u') \in C(u' \mid u, v)$ must hold, but a vertex in $C(u' \mid u, v)$ is not necessarily mapped to u' in any valid embedding.

Example 2.4. Consider the query graph Q and data graph G shown in Figure 1(a) and (b). We construct the corresponding candidate space CS shown in Figure 1(c). In CS , v_1 and v_2 belong to $C(u_1)$, $(v_0, v_1) \in C(u_0, u_1)$, and $v_1 \in C(u_1 \mid u_2, v_3)$ as u_1 can match v_1 under the condition $u_2 \mapsto v_3$.

To improve the efficiency and accuracy of subgraph counting, existing methods often build a compact candidate space by removing the invalid vertices and edges through candidate filtering algorithms [7, 18, 22, 47]. Then, the subsequent counting process is performed against a more compact candidate space instead of the original data graph, without sacrificing completeness. The candidate tree structure organizes the candidate space hierarchically to assist subgraph counting.

Definition 2.5 (Candidate Tree). Let T_Q be a spanning tree of the query graph Q . The candidate tree T_C is obtained by organizing the candidate space according to the structure of T_Q : each query

vertex u in T_Q is associated with its candidate set $C(u)$, and a tree edge (u, u') in T_Q is replaced by the candidate edges between $C(u)$ and $C(u')$. Consequently, T_C encodes all tree homomorphisms of T_Q in the candidate space.

Example 2.6. For the query graph Q in Figure 1(a), we construct its BFS spanning tree T_Q as shown in Figure 1(d). Its corresponding candidate tree T_C is illustrated in Figure 1(e). In particular, $(\langle u_0, v_0 \rangle, \langle u_1, v_1 \rangle, \langle u_2, v_3 \rangle, \langle u_3, v_5 \rangle, \langle u_4, v_0 \rangle)$ forms a homomorphism embedding of T_Q .

General Idea of Candidate Tree-based Counting. The candidate tree-based counting framework utilizes dynamic programming to compute the number of homomorphisms of T_Q in polynomial time, providing a tight upper bound for subgraph isomorphism counting of Q . In this framework, homomorphism counts are accumulated in a bottom-up manner along the tree edges of T_C . During the processing, each matching pair maintains a weight as the number of homomorphism matches of its subtrees. At the root level, the weights of all root matching pairs are collected to obtain the homomorphism count for the T_Q . More specifically, for a vertex $u \in V_Q$, we use $N_c(u)$ to denote its children in T_Q . For each $v \in C(u)$, we use a weight $W(u, v)$ to represent the number of homomorphisms of the subtree $T_Q(u)$ (rooted at u) contained in the candidate subtree $T_C(v)$ (rooted at v). Then, $W(u, v)$ is updated as follows:

$$W(u, v) = \prod_{u_c \in N_c(u)} \sum_{v_c \in C(u_c | u, v)} W(u_c, v_c). \quad (2)$$

Each matching pair updates its weight with the weights of its child matching pairs, and $W(u, v) = 1$ if u is a leaf vertex. The counting process proceeds from the leaves to the root, with an overall $O(|E_Q||E_C|)$ time [47].

Example 2.7. Continuing Example 2.6, we compute the candidate tree count in Figure 1(e) and the updated weights for each matching pair are shown in Figure 1(f). Candidate tree counting proceeds in a bottom-up manner, starting from the leaves. All leaf matching pairs have a weight of 1, (e.g., $W(u_4, v_0) = W(u_4, v_6) = 1$). By Equation 2, $W(u_1, v_1) = W(u_4, v_0) + W(u_4, v_6) = 2$; similarly, $W(u_1, v_2) = 2$. At the root level, $W(u_0, v_0) = (W(u_1, v_1) + W(u_1, v_2)) \times (W(u_2, v_3) + W(u_2, v_4)) \times W(u_3, v_5) = 8$, so the tree homomorphism count is 8.

Issues for Subgraph Isomorphism. The candidate tree-based counting method offers an efficient paradigm for subgraph homomorphism counting against tree queries. However, there are two major issues in extending it for subgraph isomorphism counting against generic query graphs.

- *Issue 1: Non-tree Edge Constraint.* As shown in Equation 2, for each matching pair (u, v) , we compute, for each child vertex $u_c \in N_c(u)$, the sum of $W(u_c, v_c)$ over all $v_c \in C(u_c | u, v)$, and then take the product of these per-child sums to obtain $W(u, v)$. However, in the processing, the non-tree edges are ignored, which may significantly affect the accumulated weight for the parent vertex, since every edge in the query graph must be matched. Here, a non-tree edge refers to an edge existing in the query graph but not in the query tree.

Example 2.8. Comparing the Q in Figure 1(a) with its BFS tree T_Q in Figure 1(d) reveals 2 non-tree edges, (u_1, u_2) and (u_2, u_4) . As in Example 2.7, the resulting candidate-tree count is 8. However, when graph homomorphism constraints are considered, the true count is 4. The discrepancy is caused by non-tree constraints. Considering the constraints imposed by the non-tree edge (u_1, u_2) , the mappings $u_1 \mapsto v_2$ and $u_2 \mapsto v_3$ are conflicting. Nevertheless, the weight update of $W(u_0, v_0)$ implicitly assumes that every candidate in $C(u_1 | u_0, v_0)$ connects to every candidate in $C(u_2 | u_0, v_0)$, which does not hold in practice. Similarly, the constraint imposed by (u_2, u_4) further restricts valid combinations and should also limit the weights propagation in the candidate tree counting.

• *Issue 2: Isomorphism Constraint.* In the processing of candidate tree-based counting, only the weights of matching pairs are maintained, rather than the explicit matching states. This implies that a data vertex may correspond to multiple query vertices, resulting in a non-injective mapping, which overestimates the number of subgraph isomorphism embeddings. We use the term *isomorphism (iso) constraint* to refer to the injectivity mapping constraint.

Example 2.9. Continuing Example 2.8, when vertex labels and vertex degrees are considered as filtering criteria [7, 53], v_0 is contained in both $C(u_0)$ and $C(u_4)$ as shown in Figure 1(c). Consequently, in the current candidate tree, the weight of leaf v_0 in $W(u_4, v_0)$ is propagated upward to the root v_0 in $W(u_0, v_0)$, causing v_0 to be counted twice and resulting in an overestimation in subgraph isomorphism count.

Remedy of Existing Solutions. To support generic subgraph counting, prior works extend candidate tree counting by approximating non-tree and injective constraints. Fastest [47] estimates the final count by sampling candidate trees and evaluating whether they form valid query matches. However, its performance deteriorates rapidly when the sampling space becomes prohibitively large, which often happens on larger datasets and leads to sampling failures. Alternatively, FlowSC [18] employs supervised learning with cross-graph attention to implicitly emulate matching checks. However, its effectiveness relies on sufficient ground truth, which is costly to obtain for hard query workloads.

3 Our Approach

Overview. Built upon the candidate tree framework, ASC first incorporates Type-I constraints via a matrix-based scheme (section 4), and utilizes an LCA ratio mechanism to filter out invalid combinatorial weights induced by Type-II constraints (section 5). Based on the resulting algebraic homomorphism count, we finally enforce the injectivity constraints via local sampling on subgraphs induced by same-label vertices (section 6).

Algorithm Details. Algorithm 1 summarizes the counting process of ASC. Given a query graph Q and a data graph G , we first construct and refine the candidate space CS using the filtering method BipartitePlus [18] (line 1), and build a BFS spanning tree T_Q of Q . We choose BFS over DFS spanning trees because the former generates more Type-I non-tree edges, whereas the latter generates only Type-II ones. Type-I non-tree edges are easier to handle than Type-II ones, as discussed in the following two sections. Then we compute the LCA ratio for each LCA candidate (line 3).

In the counting phase, the parent weight $W(u, v)$ is updated from its child vertices $N_c(u)$ using a matrix computation-based counting method SequenceCounting that incorporates Type-I non-tree constraints in $Q[N_c(u)]$ (line 7), where $Q[N_c(u)]$ denotes the subgraph of Q induced by $N_c(u)$. The Type-II non-tree edge constraints are handled by applying the LCA ratios at LCA vertices (line 9). We update weights in reversed BFS order and aggregate them at the root (line 12) to obtain the subgraph homomorphism count¹. An isomorphism ratio is then estimated using a local sampling method (line 13), and the final subgraph isomorphism count is obtained by multiplying the homomorphism count by this ratio (line 14).

4 Handling Type-I Non-tree Edge Constraints

In this section, we introduce the key component of ASC, which is a matrix-based counting model that can approximate the Type-I non-tree edge constraint in candidate tree-based counting.

The tree counting formulation in Equation 2 implicitly assumes child vertices are mutually independent, leading to error accumulation when inter-dependencies (i.e., non-tree edge constraints) exist. While exact counting for these constraints typically incurs exponential complexity,

¹As BipartitePlus applies injectivity filtering, this is not a strict homomorphism count.

Algorithm 1: ALGEBRAIC SUBGRAPH COUNTING

Input: A query graph Q , a data graph G
Output: Estimated subgraph count of Q in G

```

1  $CS \leftarrow \text{BipartitePlus}(Q, G);$ 
2  $T_Q \leftarrow \text{BuildTree}(Q);$ 
3  $\text{LCARatio}(CS, T_Q);$ 
4 foreach  $u$  in reversed BFS Traversal of  $T_Q$  do
5    $s \leftarrow \text{BuildSequence}(Q[N_c(u)]);$ 
6   foreach  $v \in C(u)$  do
7      $W(u, v) \leftarrow \text{SequenceCounting}(u, v, CS, s);$ 
8     if  $u$  is a LCA vertex then
9        $W(u, v) \leftarrow W(u, v) \times \text{LCA}(u, v);$ 
10  $\text{HomoCounts} \leftarrow 0;$ 
11 foreach  $v_r \in C(u_r)$  where  $u_r$  is the root of  $T_Q$  do
12    $\text{HomoCounts} += W(u_r, v_r);$ 
13  $\text{IsoRatio} \leftarrow \text{LocalSampling}(CS, Q);$ 
14  $\text{IsoCounts} \leftarrow \text{HomoCounts} \times \text{IsoRatio};$ 
15 return  $\text{IsoCounts}$ 
```

we develop an algebraic method to simulate this process in polynomial time. By transforming candidate connectivity into transition matrices and candidate weights into weight vectors, we propagate weights via matrix multiplications to simulate weight accumulation over Cartesian products (subsection 4.1). For complex structures like cycles among the children, we utilize weight matrices to encode both single and multiple indirect constraints (subsection 4.2). By traversing the children through carefully designed matrix operations, our method incorporates Type-I non-tree constraints into tree counting, with the resulting vector sum yielding the weight for a parent vertex.

4.1 A Simplified Counting Case

Fixing Candidate Tree-based Counting. Equation 2 formulates a tree-based counting paradigm where weight updates are computed by decoupling each child branch, implicitly assuming that child vertices are mutually isolated. To accommodate the case where the child vertices are connected (i.e., non-tree edges), we need to inspect each matching combination for the child vertices. Specifically, given a query vertex u and its matching vertex v , let $\Omega = \times_{u_c \in N_c(u)} C(u_c \mid u, v)$ be the Cartesian product of the conditional candidates of u 's child vertices. Clearly, Ω defines a complete matching space for the child vertices of u when $u \mapsto v$. Based on Ω , we can compute the weight as follows:

$$W(u, v) = \sum_{\mathbf{v} \in \Omega} \left(\sigma(\mathbf{v}) \cdot \prod_{u_c \in N_c(u)} W(u_c, \mathbf{v}[u_c]) \right). \quad (3)$$

Here, each combination $\mathbf{v} \in \Omega$ represents a possible matching for vertices in $N_c(u)$, and $\mathbf{v}[u_c]$ denotes the candidate vertex mapped to u_c . The term σ is an indicator function with value 1 if the subgraph induced by $\mathbf{v}$ is a homomorphism of $Q[N_c(u)]$, and 0 otherwise.

Equation 3 shares the sum-product structure of FAQ-based subgraph counting (Equation 1), where summation enumerates possible vertex mappings and multiplication enforces structural consistency.

Compared with the standard FAQ expression, Equation 3 further incorporates candidate-tree-based weights, transforming the generic sum-product query into a recursive weighted aggregation process.

Direct evaluation of Equation 3 requires enumerating all combinations in the matching space Ω , resulting in $O(n^{|N_c(u)|})$ time, which is exponential in $|N_c(u)|$, assuming that the maximum size of $C(u_c \mid u, v)$ is n . In the FAQ framework, the InsideOut algorithm reduces the cost through optimized variable elimination, but its complexity still depends exponentially on the fractional FAQ-width of the subquery. To address this challenge, we propose a matrix-based approximation that avoids exhaustive enumeration and produces high-quality counting results in polynomial time.

Intuition of Matrix Computation. The main innovation of our method is transforming the subgraph homomorphism counting into a sequence of matrix multiplication operations. Intuitively, adjacency matrix multiplication provides an efficient mechanism for evaluating connectivity between vertex sets. Moreover, its computational rule naturally aligns with Cartesian product operations across vertex sets, where each resulting entry counts the number of connecting paths between the corresponding pair of vertices in the candidate space. With a carefully designed computation strategy, we can incorporate non-tree constraints via algebraic methods.

Constrained Sequence. Given a query vertex u , its child vertices can induce a set of connected components. For example, in the Q of Figure 1(a), $\{u_1, u_2\}$ and $\{u_3\}$ are two connected components. According to the above discussion, each connected component can be processed independently. In the following, we focus on the non-trivial connected components (i.e., containing at least 2 vertices), since the non-tree edges only exist in such components.

Definition 4.1 (Constrained Sequence). Given a query vertex u in the query tree T_Q , let S be a non-trivial connected component induced by a set of vertices in $N_c(u)$ from Q . A constrained sequence s is an ordered arrangement of vertices in S with the following properties.

- Given a vertex $u_c \in S$, a vertex appearing before (Resp., after) u_c in s is called the predecessor (Resp., successor) of u_c .
- Given two vertices $u'_c, u_c \in S$, u'_c is called the direct predecessor (Resp., direct successor) of u_c if it appears immediately before (Resp., after) u_c in s ; Otherwise, u'_c is an indirect predecessor (Resp., indirect successor) of u_c if u'_c is a neighbor of u_c .

Based on the second property, we note that there must be a non-tree edge between the indirect predecessor-successor vertices, while it is not necessary between the direct ones.

Example 4.2. Consider T_Q in Figure 2(a), where $\langle u_1, u_2, u_3, u_4 \rangle$ is a constrained sequence generated by vertices in $N_c(u_0)$. Clearly, u_1 is a direct predecessor of u_2 and an indirect predecessor of u_4 .

Counting via Matrix Multiplication Propagation. For ease of exposition, we start by considering a simplified case where s has no indirect predecessor-successor vertices. Given a query tree T_Q and its corresponding candidate tree T_C , let u and v be a pair of matching vertices. To compute the weight $W(u, v)$ in Equation 3, we maintain a weight vector for each child vertex u_c of u and a transition matrix for each direct predecessor-successor pair of u , which are defined as follows.

Definition 4.3 (Weight Vector). Given a vertex u_c in the constrained sequence s , the weight vector of u_c , denoted as $\mathbf{V}$, is a column vector with $|C(u_c \mid u, v)|$ entries, which are initialized as $W(u_c, v_c)$ for each $v_c \in C(u_c \mid u, v)$.

Definition 4.4 (Transition Matrix). Given a direct predecessor-successor vertex pair u'_c and u_c in the constrained sequence s , the transition matrix between u'_c and u_c , denoted as $\mathbf{A}$, is constructed with rows and columns corresponding to $C(u'_c \mid u, v)$ and $C(u_c \mid u, v)$, respectively. This matrix $\mathbf{A}$ encodes the non-tree constraint between u'_c and u_c within the candidate space. If a non-tree edge

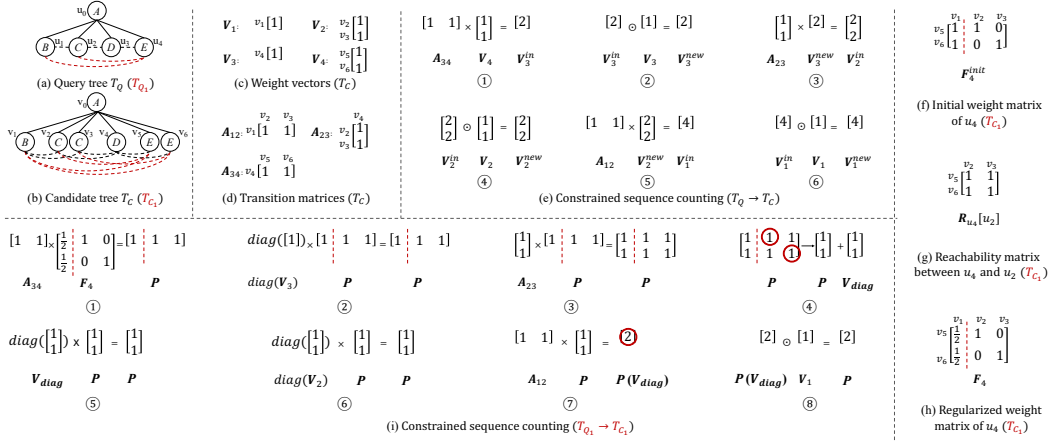


Fig. 2. Example of matrix-based counting: dashed edges denote non-tree edges and red edges present only in T_{Q_1} and T_{C_1} .

exists between u'_c and u_c , then $A_{ij} = 1$ if there exists a candidate edge between the i -th candidate of u'_c and the j -th candidate of u_c , and $A_{ij} = 0$ otherwise. If no non-tree edge exists, A contains all ones, representing that all candidate combinations between u'_c and u_c are considered valid.

Example 4.5. Consider the query tree T_Q in Figure 2(a) and the candidate tree T_C in Figure 2(b), where $\langle u_1, u_2, u_3, u_4 \rangle$ is a constrained sequence. Note here that we ignore the red edges at this moment. The weight vectors of vertices are shown in Figure 2(c). Taking u_2 for example, since it is a leaf vertex and has two candidates, i.e., v_2 and v_3 , its weight vector is initialized as $[[1], [1]]$. The transition matrices are shown in Figure 2(d).

With the weight vector and transition matrix, we can compute $W(u, v)$ as follows. We start from the last vertex in the constrained sequence and propagate backwards. At each step, the weight vector of the successor vertex is multiplied by the transition matrix and then combined with the predecessor's own weight vector via a Hadamard product. Therefore, the counts are propagated along the non-tree candidate edges and accumulated into the predecessor's weights. We repeat this processing until the head is reached, where the sum of the final vector's entries updates $W(u, v)$.

Example 4.6. The constrained sequence counting starts from the end of sequence, proceeds backward along the sequence, as shown in Figure 2(e): in Step ①, the weight vector V_4 of $C(u_4 | u_0, v_0)$ is multiplied by A_{34} to produce the input vector V_3^{in} for $C(u_3 | u_0, v_0)$, which are then multiplied element-wise with the original weight vector V_3 in Step ② to obtain the updated vector V_3^{new} . This process continues backwards along the sequence, following the Steps ③–⑥. Then, at the head u_1 , the sum of the entries in V_1^{new} yields the count of the constrained sequence, which is 4 and exactly equals the homomorphism count of the constrained sequence.

LEMMA 4.7. *Given a constrained sequence s with parent vertex u matched to a candidate v , the matrix-based method returns the exact homomorphism count of the induced subgraph $Q[s]$ in the candidate space conditioned on $u \mapsto v$, assuming there are no indirect predecessor-successor connections and no false positive candidates.*

All proofs in this paper are omitted due to space limitations².

²A full version of this paper is available at <https://arxiv.org/abs/2606.29128>.

4.2 Extending to Generic Case

Motivation. When a query vertex has an indirect predecessor, the indirect and direct relationships along the sequence jointly impose constraints on the candidate matching. For example, in the query tree T_{Q_1} shown in Figure 2(a), where the red edges are included in the tree, u_4 has 2 indirect predecessors u_1 and u_2 . The indirect edges (u_1, u_4) , (u_2, u_4) and the sequence $\langle u_1, u_2, u_3, u_4 \rangle$ jointly impose matching constraints on this sequence.

We propose to use a weight matrix to encode the indirect constraints on a vertex, and further approximate the interactions among these constraints through an enhanced regularized weight matrix. With carefully designed propagation rules, the weight matrix accumulates valid weights during traversal of the constrained sequence to support approximate counting with indirect constraints.

Incorporating Indirect Predecessor Constraints. In the following, we proceed to deal with the indirect predecessor-successor relationships between vertices presented in the constrained sequence. For a vertex with indirect predecessors, we need to properly take the connection constraints brought by such predecessors into consideration during the matrix propagation. To this end, we introduce the concept of *weight matrix*, which is essentially an advanced version of the weight vector by integrating both the candidate weights and the connection status with the indirect predecessors.

Definition 4.8 (Weight Matrix). Given a vertex u_c in the constrained sequence s with m indirect predecessors $\langle u_{p_1}, \dots, u_{p_m} \rangle$ ordered as in s , the weight matrix of u_c , F , consists of m column-wise concatenated blocks, where the k -th block $F[u_{p_k}]$ corresponds to u_{p_k} :

- For an indirect predecessor u_p , the corresponding block $F[u_p]$ is a matrix with $|C(u_c \mid u, v)|$ rows and $|C(u_p \mid u, v)|$ columns, where an entry $F[u_p]_{ij} = 1$ if there is a candidate edge between the i -th candidate of u_c and the j -th candidate of u_p , and 0 otherwise.
- For the block corresponding to the first indirect predecessor u_{p_1} , we further set its nonzero entries to the corresponding candidate weights $W(u_c, v_c)$ of u_c . The weight accumulation in the propagation process is performed through this primary block, while the remaining blocks are used to verify connectivity for non-tree constraints. This ensures that the candidate weights of u_c are accumulated exactly once in the Cartesian-product space.

Example 4.9. For the constrained sequence $\langle u_1, u_2, u_3, u_4 \rangle$ in T_{Q_1} shown in Figure 2(a) and its candidate tree T_{C_1} in Figure 2(b), the initial weight matrix F_4^{init} of u_4 is presented in Figure 2(f), where the red dashed line separates the blocks corresponding to different indirect predecessors. Since u_4 has two indirect predecessors u_1 and u_2 , its weight matrix has two blocks. Considering u_1 , since there is only one vertex v_1 in $C(u_1 \mid u_0, v_0)$ which connects to both candidates v_5 and v_6 in $C(u_4 \mid u_0, v_0)$, the corresponding block is a 2×1 matrix of ones, representing both connectivity and candidate weights. The block for u_2 is constructed similarly based on connectivity only.

Regularizing Weight Matrix. Recall that the overall counting process is formulated as a matrix-based propagation backwards along the constrained sequence. Based on Definition 4.8, the candidate weights of the successor and the non-tree constraint from its first indirect predecessor are embedded in the first block of the successor's weight matrix F , where weight accumulation in propagation takes place. The constraints from other indirect predecessors are encoded in separate blocks. These constraints from different indirect predecessors cannot be applied to the successor's candidates simultaneously for valid weight accumulation during the propagation process.

Example 4.10. In T_{Q_1} in Figure 2(a), both u_1 and u_2 are indirect predecessors of u_4 . In T_{C_1} in Figure 2(b), the weights of u_4 's candidates v_5 and v_6 can propagate backwards along the sequence to v_2 and v_3 . This propagation satisfies the indirect constraint between u_1 and u_4 , since v_1 connects to both v_5 and v_6 , but it violates the indirect constraint between u_2 and u_4 , because v_2 is not adjacent

to v_6 , and v_3 is not adjacent to v_5 . The indirect constraints of (u_1, u_4) and (u_2, u_4) are encoded in different blocks of the weight matrix, which may not be satisfied simultaneously on v_5 and v_6 .

To jointly enforce the constraints from multiple indirect predecessors on a successor without introducing complex multi-constraint checks during propagation, we develop regularization techniques for the weight matrix. Specifically, we apply a scaling ratio matrix to the block of the first indirect predecessor in the weight matrix to approximately offset invalid weights before the propagation begins. Essentially, each ratio estimates how much of the weight propagated from a successor candidate to the corresponding candidate of the first indirect predecessor remains valid after jointly enforcing all indirect-predecessor constraints along the sequence. To facilitate the computation, we make use of the following definition *reachability matrix*, which records the reachability situation between the candidates of two query vertices.

Definition 4.11 (Reachability Matrix). Given u_c and u'_c in the constrained sequence, the reachability matrix between u_c and u'_c , denoted as $\mathbf{R}_{u_c}[u'_c]$, maintains the reachability from vertices in $C(u_c \mid u, v)$ to $C(u'_c \mid u, v)$ along the constrained sequence s , which can be easily obtained via successive multiplying the transition matrices between consecutive vertices in the constrained sequence.

Example 4.12. Figure 2(g) shows the reachability matrix $\mathbf{R}_{u_4}[u_2]$ from candidate of u_4 to the candidates of u_2 , which is obtained by binarizing $(\mathbf{A}_{23} \times \mathbf{A}_{34})^\top$, where all non-zero entries are set to 1. It indicates that both candidates v_5 and v_6 can reach v_2 and v_3 in the candidate space along the constrained sequence.

Now, given a vertex u_c in the constrained sequence with indirect predecessors $\langle u_{p_1}, \dots, u_{p_m} \rangle$ and its initial weight matrix $\mathbf{F}$, we use $\mathbf{F}[u_{p_i}]$ to denote the matrix block corresponding to the indirect predecessor u_{p_i} . With the reachability matrix, we update the weight matrix as follows.

$$\mathbf{F}[u_{p_i}] \leftarrow \mathbf{F}[u_{p_i}] \odot (\text{diag}(\mathbf{R}_{u_c}[u_{p_{i+1}}] \cdot \mathbf{1})^{-1} \cdot ((\mathbf{F}[u_{p_{i+1}}] \odot \mathbf{R}_{u_c}[u_{p_{i+1}}]) \cdot \mathbf{R}_{u_{p_{i+1}}}[u_{p_i}])). \quad (4)$$

In Equation 4, the state at u_{p_i} depends on $u_{p_{i+1}}$. We examine the reachability of each candidate of u_c to candidates at $u_{p_{i+1}}$ with $\mathbf{R}_{u_c}[u_{p_{i+1}}]$. Reachable candidates at $u_{p_{i+1}}$ receive the accumulated weights propagated from u_c 's candidates. Accordingly, multiplying by an all-ones vector $\mathbf{1}$ yields the number of reachable candidates at $u_{p_{i+1}}$ from each candidate of u_c . The resulting vector is broadcast via the *diag*($\cdot$) operation to form a diagonal matrix, whose diagonal entries serve as the denominator for the corresponding row in the ratio computation. The numerator accounts for how much of the accumulated weight propagated from u_c 's candidates satisfies the indirect constraints at $u_{p_{i+1}}$ and can further propagate to u_{p_i} . This is computed in the second line of Equation 4: $\mathbf{F}[u_{p_{i+1}}]$ encodes the indirect constraints of $u_{p_{i+1}}$, while $\mathbf{R}_{u_c}[u_{p_{i+1}}]$ captures reachability from u_c to $u_{p_{i+1}}$. Candidates that are both reachable and indirect constraint-satisfying are propagated to u_{p_i} through $\mathbf{R}_{u_{p_{i+1}}}[u_{p_i}]$, forming the numerator of the ratio computation. The resulting ratio matrix is combined with $\mathbf{F}[u_{p_i}]$ via the Hadamard product, thereby incorporating the indirect constraints at u_{p_i} and producing an updated matrix block, which is then used to update $u_{p_{i-1}}$. Since only the first block of $\mathbf{F}$ contains candidate weights of u_c , we retain only the regularized update of this block in Equation 4. The remaining blocks encode connectivity, where only the zero/non-zero distinction matters, so their original values are kept.

Example 4.13. Based on the initial feature matrix in Example 4.9, we apply a scaling ratio matrix on the block $\mathbf{F}_4^{\text{init}}[u_1]$ of u_1 . For the reachability matrix $\mathbf{R}_{u_4}[u_2]$ shown in Figure 2(g), we compute the row sums, resulting in the vector $[[2], [2]]$, which serves as the per-row denominators of the ratio matrix. Next, we take the Hadamard product of $\mathbf{R}_{u_4}[u_2]$ and $\mathbf{F}_4^{\text{init}}[u_2]$, obtaining $[[1, 0], [0, 1]]$, which

incorporates the indirect constraint on (u_2, u_4) . To evaluate how much of the accumulated weight can further propagate to $C(u_1 \mid u_0, v_0)$, we multiply the obtained matrix by $R_{u_2}[u_1] = [[1], [1]]$, yielding $[[1], [1]]$. Dividing each entry by its corresponding row denominator produces the ratio matrix $[[1/2], [1/2]]$. Finally, the ratio matrix is applied to $F_4^{init}[u_1]$ via Hadamard product, which further incorporates the indirect constraint between (u_1, u_4) . The resulting matrix is the final feature matrix F_4 , as shown in Figure 2(h).

Propagating Weight Matrix. After setting the weight matrix properly, we propagate it via matrix multiplication along the reverse direction of the constrained sequence until reaching the first vertex, where the accumulated weights are extracted as the homomorphism count. Because each vertex in the constrained sequence might have its own weight matrix, to avoid ambiguity, we call the propagated weight matrix as *passing matrix* P . During the propagation, we need to execute the following two types of operations accordingly, including *weight aggregation* and *weight extraction*.

- *Weight Aggregation.* Let u_c be the current vertex in the constrained sequence, and u'_c its direct predecessor during the propagation. We incorporate the weight matrix F of u'_c into the current passing matrix P to update the accumulated weights, as detailed in algorithm 2. Initially, we handle the boundary cases where either matrix acts as a vector because it lacks indirect predecessors.

Case 1: P and F have no associated indirect predecessors. In this scenario, since P has not accumulated any indirect predecessors along the propagation path (line 1), both P and F act purely as weight vectors (Definition 4.3). Consequently, we simply compute their Hadamard product (line 3), identical to step ② of Example 4.6.

Case 2: F has associated indirect predecessors but P does not. F functions as a weight matrix while P remains a weight vector (line 5) under this condition. We row-scale the first block of F by P , and then F takes over as the updated passing matrix (line 7–line 8).

Case 3: P has associated indirect predecessors, but F does not. P has already become a matrix but F remains a vector (line 11) in this scenario, we directly apply F to the first block of P via row-wise scaling to accumulate the candidate weights (line 12).

Case 4: both P and F have associated indirect predecessors. P and F both function as weight matrices (Definition 4.8) in this scenario. We identify their predecessor sets $\mathcal{U}_P$ and $\mathcal{U}_F$, along with their earliest predecessor $u_{P,1}$ and $u_{F,1}$. We first process any overlapping predecessors ($u \in \mathcal{U}_P \cap \mathcal{U}_F$) by performing an element-wise Hadamard product on their corresponding blocks. This step merges their connectivity and weights simultaneously (line 16–line 18). A critical requirement is that only the block corresponding to the earliest indirect predecessor in the constrained sequence acts as the primary block carrying the accumulated weights. To ensure this, we compare the sequence positions of $u_{F,1}$ and $u_{P,1}$. If $u_{F,1}$ appears earlier ($u_{F,1} \prec u_{P,1}$) in the constrained sequence, $F[u_{F,1}]$ becomes the new primary block. We extract the row weights v_{row} from the former primary block $P[u_{P,1}]$, row-broadcast them into $F[u_{F,1}]$, and degrade $P[u_{P,1}]$ into a pure 0/1 connectivity block via binarization (line 19–line 23). Conversely, if $u_{P,1}$ remains the primary block, we extract the weights from $F[u_{F,1}]$, apply them to $P[u_{P,1}]$, and binarize $F[u_{F,1}]$ (line 24–line 29). Finally, any remaining non-overlapping blocks belonging solely to the weight matrix ($u \in \mathcal{U}_F \setminus \mathcal{U}_P$) are appended to the passing matrix P column-wise, expanding its predecessor set accordingly (line 30–line 32).

- *Weight Extraction.* During backward propagation along the constrained sequence, if the predecessor u'_c recorded in the current passing matrix P is encountered, it signifies a *cycle closure*. We perform a weight extraction operation as detailed in algorithm 3.

Algorithm 2: WEIGHTAGGREGATION**Input:** Passing matrix P at u_c , Weight matrix F of current direct predecessor u'_c **Output:** Updated passing matrix P

```

1  if  $P$  has no associated indirect predecessors then
2      if  $u'_c$  has no indirect predecessor then                                // Case 1
3           $P \leftarrow P \odot F$ ;
4          return  $P$ 
5       $\mathcal{U}_F \leftarrow$  indirect predecessors of blocks in  $F$ ;                    // Case 2
6       $u_{F,1} \leftarrow$  the first indirect predecessor in  $\mathcal{U}_F$ ;
7       $F[u_{F,1}] \leftarrow F[u_{F,1}] \otimes_{\text{row}} P$ ;
8      return  $F$ 
9   $\mathcal{U}_P \leftarrow$  indirect predecessors of blocks in  $P$ ;
10  $u_{P,1} \leftarrow$  the first indirect predecessor in  $\mathcal{U}_P$ ;
11 if  $u'_c$  has no indirect predecessor then                                // Case 3
12      $P[u_{P,1}] \leftarrow P[u_{P,1}] \otimes_{\text{row}} F$ ;
13     return  $P$ 
14  $\mathcal{U}_F \leftarrow$  indirect predecessors of blocks in  $F$ ;                        // Case 4
15  $u_{F,1} \leftarrow$  the first indirect predecessor in  $\mathcal{U}_F$ ;
16 foreach  $u \in \mathcal{U}_P \cap \mathcal{U}_F$  do
17      $F[u] \leftarrow P[u] \odot F[u]$ ;
18      $P[u] \leftarrow F[u]$ ;
19 if  $u_{F,1} \prec u_{P,1}$  then
20      $\mathbf{v}_{\text{row}} \leftarrow \text{EXTRACTROWWEIGHTS}(P[u_{P,1}])$ ;
21      $F[u_{F,1}] \leftarrow F[u_{F,1}] \otimes_{\text{row}} \mathbf{v}_{\text{row}}$ ;
22      $P[u_{P,1}] \leftarrow \text{BINARIZE}(P[u_{P,1}])$ ;
23      $P[u_{F,1}] \leftarrow F[u_{F,1}]$ ;
24 else
25      $\mathbf{v}_{\text{row}} \leftarrow \text{EXTRACTROWWEIGHTS}(F[u_{F,1}])$ ;
26      $P[u_{P,1}] \leftarrow P[u_{P,1}] \otimes_{\text{row}} \mathbf{v}_{\text{row}}$ ;
27      $F[u_{F,1}] \leftarrow \text{BINARIZE}(F[u_{F,1}])$ ;
28     if  $u_{F,1} \in \mathcal{U}_P$  then
29          $P[u_{F,1}] \leftarrow F[u_{F,1}]$ ;
30 foreach  $u \in \mathcal{U}_F \setminus \mathcal{U}_P$  do
31      $\text{APPEND}(P, F[u])$ ;
32      $\mathcal{U}_P \leftarrow \mathcal{U}_P \cup \{u\}$ ;
33 return  $P$ 

```

At this stage, the block $P[u'_c]$ forms a square matrix. Its diagonal elements indicate whether the corresponding candidates of u'_c can successfully complete a cycle traversal under the indirect constraints, as in the cycle $(u_2 \rightarrow u_4 \rightarrow u_3 \rightarrow u_2)$ of T_{Q_1} in Figure 2(a). We extract these diagonal elements into a vector $\mathbf{v}_{\text{diag}}$ (line 3) to capture the valid states of u'_c 's candidates. Since the information at u'_c has now been fully evaluated and aggregated, we simplify the passing matrix by

Algorithm 3: WEIGHTEXTRACTION**Input:** Passing matrix $\mathbf{P}$ at u_c , Current direct predecessor u'_c **Output:** Updated passing matrix $\mathbf{P}$

```

1  $\mathcal{U}_P \leftarrow$  indirect predecessors of blocks in  $\mathbf{P}$ ;
2 if  $u'_c \in \mathcal{U}_P$  then
3    $\mathbf{v}_{\text{diag}} \leftarrow \text{EXTRACTDIAG}(\mathbf{P}[u'_c]);$ 
4    $\mathcal{U}_P \leftarrow \mathcal{U}_P \setminus \{u'_c\};$ 
5    $\text{REMOVEBLOCK}(\mathbf{P}, u'_c);$ 
6   if  $\mathcal{U}_P \neq \emptyset$  then                                     // Case 1
7      $u_{P,1} \leftarrow$  the first indirect predecessor in  $\mathcal{U}_P$ ;
8      $\mathbf{v}_{\text{diag}} \leftarrow \text{BINARIZE}(\mathbf{v}_{\text{diag}});$ 
9      $\mathbf{P}[u_{P,1}] \leftarrow \mathbf{P}[u_{P,1}] \otimes_{\text{row}} \mathbf{v}_{\text{diag}};$ 
10  else                                                         // Case 2
11     $\mathbf{P} \leftarrow \mathbf{v}_{\text{diag}};$ 
12 return  $\mathbf{P}$ 

```

completely removing the block corresponding to u'_c (line 5). The subsequent update of $\mathbf{P}$ depends on whether other indirect predecessors remain.

Case 1: Other blocks remain in $\mathbf{P}$. In this scenario (line 6), the presence of remaining blocks implies that u'_c 's block was not the primary carrier of the accumulated weights. Nevertheless, its constraint validation is crucial. Therefore, we binarize the extracted vector to retain connectivity (line 8) and apply it via row-wise scaling to the first block of $\mathbf{P}$ (line 9). This operation effectively filters out the accumulated weights of candidates that fail to satisfy the cycle constraints.

Case 2: u'_c 's block was the only block in $\mathbf{P}$. In this scenario (line 10), the extracted vector $\mathbf{v}_{\text{diag}}$ fully represents the newly accumulated weights. In this case, $\mathbf{v}_{\text{diag}}$ entirely becomes the updated passing matrix $\mathbf{P}$ (line 11) and will continue propagating backward along the constrained sequence.

Algorithm 4 outlines the overall procedure for computing the candidate weight $W(u, v)$ under both sequence constraints and standard tree constraints. The algorithm performs a backward propagation along the constrained sequence s .

We initialize the process at the last vertex u_c of sequence s , setting its weight matrix as the initial state of the passing matrix $\mathbf{P}$ (line 1–line 2). During the reversed traversal of s , for each adjacent pair (u'_c, u_c) , we first propagate the passing matrix $\mathbf{P}$ from u_c to its direct predecessor u'_c by multiplying it with the transition matrix $\mathbf{A}_{u'_c, u_c}$ (line 4). Then, we generate the weight matrix $\mathbf{F}$ for u'_c (line 5). If u'_c closes a cycle—meaning it is an indirect predecessor previously visited—we refine the passing matrix via the WEIGHTEXTRACTION operation to capture cycle constraints (line 7). Subsequently, the state is updated by fusing $\mathbf{P}$ and $\mathbf{F}$ via the WEIGHTAGGREGATION operation (line 8).

Once the propagation reaches the head vertex of s , the finalized passing matrix $\mathbf{P}$ degrades into a vector containing the accumulated valid weights. We sum all elements of this vector to obtain the total sequence-constrained weight w_{con} (line 9). Furthermore, the query vertex u may also possess isolated children that are completely independent of any Type-I non-tree constraints. We compute their normal candidate weight w_{norm} using Equation 2 (line 10). Finally, the product of w_{con} and w_{norm} yields the final matching weight $W(u, v)$ (line 11).

Example 4.14. Continuing Example 4.13, where the weight matrix $\mathbf{F}_4$ is constructed in Figure 2(h). The propagation process is illustrated in Figure 2(i). Starting from u_4 , the weight matrix $\mathbf{F}_4$ is

Algorithm 4: SEQUENCECOUNTING**Input:** u, v, CS, s **Output:** $W(u, v)$

```

1  $u_c \leftarrow$  the last vertex in  $s$ ;
2  $\mathbf{P} \leftarrow \text{WEIGHTMATRIX}(u_c)$ ;
3 foreach  $(u'_c, u_c)$  in the reversed traversal of  $s$  do
4    $\mathbf{P} \leftarrow \mathbf{A}_{u'_c u_c} \mathbf{P}$ ;
5    $\mathbf{F} \leftarrow \text{WEIGHTMATRIX}(u'_c)$ ;
6   if  $u'_c$  closes a cycle then
7      $\mathbf{P} \leftarrow \text{WEIGHTEXTRACTION}(\mathbf{P}, u'_c)$ ;
8    $\mathbf{P} \leftarrow \text{WEIGHTAGGREGATION}(\mathbf{P}, \mathbf{F}, u'_c)$ ;
9  $w_{\text{con}} \leftarrow \text{SUM}(\mathbf{P})$ ;
10  $w_{\text{norm}} \leftarrow \text{ISOLATEDCHILDRENWEIGHT}(u, v)$ ;
11  $W(u, v) \leftarrow w_{\text{con}} \times w_{\text{norm}}$ ;
12 return  $W(u, v)$ 

```

propagated to u_3 by multiplication with $\mathbf{A}_{34}$, illustrated in Step ①. The weight vector of u_3 , which encodes the weights of its candidates, is then incorporated into the first block of $\mathbf{P}$ through row-broadcast as specified in line 12 of algorithm 2, producing the updated passing matrix in Step ②. The passing matrix is subsequently propagated backward to u_2 through $\mathbf{A}_{23}$, obtaining $\mathbf{P}$ in ③. Since u_2 is the indirect predecessor of u_4 , whose constraint on u_4 is enforced in the passing matrix, we extract the block of u_2 . In Step ④, the diagonal (circled in red) of the block corresponding to u_2 is extracted, obtaining $\mathbf{V}_{\text{diag}}$ (line 3 of algorithm 3), and then this block is removed from $\mathbf{P}$ (line 5 of algorithm 3). $\mathbf{V}_{\text{diag}}$ is first binarized (line 8 of algorithm 3) and applied to the first block of $\mathbf{P}$ (line 9 of algorithm 3), yielding the updated $\mathbf{P}$ after weight extraction as shown in Step ⑤. In Step ⑥, the initial weight vector $\mathbf{V}_2$ of u_2 is row-scaled into $\mathbf{P}$, which completes the weight aggregation at u_2 and produces the updated $\mathbf{P}$. After that, the obtained $\mathbf{P}$ continues to propagate backward to u_1 through $\mathbf{A}_{12}$. At this point, u_1 is the only remaining indirect predecessor encoded in the passing matrix, the diagonal (circled in red) of $\mathbf{P}$ is extracted to form the vector $\mathbf{V}_{\text{diag}}$ as shown in Step ⑦, which is then combined with $\mathbf{V}_1$ via a Hadamard product. Finally, summing the entries of $\mathbf{P}$ in Step ⑧ yields the constrained sequence count, which equals to 2. This value is exactly the true homomorphism count of the constrained sequence.

Remark. It should be remarked that the homomorphism counts obtained from the constrained sequence counting are not exact. When the constrained sequence contains indirect constraints, multiple indirect constraints can simultaneously affect a single vertex. Although the regularized weight matrix is introduced to precompute such joint constraint effects before propagation, it can only approximate the proportion of valid weights propagated based on the connected paths. In fact, the valid weight contributed by each connected path is different. The counting rules for such a complicated structure might be further optimized in future work.

4.3 Complexity Analysis

THEOREM 4.15. *Given a connected component S under a query vertex in the query tree, the time complexity of the overall counting process for S is polynomial to the size of S and the candidate space.*

To assess the efficiency of the constrained sequence counting, we conduct a preliminary experiment on a synthetic dataset. Specifically, we use a 32-clique query graph and a 32-clique data

graph, where vertices share the same label. By definition, the exact count is $32!$. We evaluate this instance using two representative subgraph matching algorithms recommended in recent survey papers [50, 57]. None of them can finish within two hours. The state-of-the-art sampling-based subgraph counting method Fastest [47] also fails to return a valid result within 100 seconds. In contrast, our method computes the count in 7.11 seconds with a q-error of 1.98, where candidate space construction and refinement dominate the runtime, and counting takes only 0.52 seconds.

5 Handling Type-II Non-tree Edge Constraints

In this section, we introduce the *lowest common ancestor* (LCA for short) to handle the Type-II non-tree edges constraints.

Recall that a Type-II non-tree edge connects two vertices with different parents in the query tree. However, the two vertices must have an LCA vertex in the tree. Clearly, the paths from the LCA vertex to the two endpoints form a triangle together with this non-tree edge. A necessary condition for a valid match of the query graph is that such a triangle must also exist in the candidate space. However, in the candidate tree-based counting method defined by Equation 2, the weights of candidate vertices are propagated upward along the tree paths without considering such non-tree edge constraints. To remove the false positive counting, we develop an LCA ratio computation method. Given a Type-II non-tree edge, we check if the two endpoints of its matching edge can reach the same LCA candidate. By calculating the positive ratio, we can therefore estimate the valid subgraph homomorphism count.

To compute the LCA ratio, we first find the descendants for each candidate of an LCA vertex using a dynamic programming method. Then, based on the LCA descendants, we compute the LCA ratio for each candidate by using set intersections to count descendant pairs that form valid triangles across endpoints.

5.1 Finding LCA Descendants

For each LCA candidate, we compute its descendant candidates located at the endpoints of a non-tree edge. This is done by propagating reachable candidates downward along the tree path from the LCA to the endpoint. To speed up the computation, we adopt a level-by-level manner where the results computed in the previous level can be reused in the current level. This can be implemented as a dynamic programming on the candidate tree.

Algorithm Details. The computation details are summarized in algorithm 5. For each Type-II non-tree edge, we form two LCA paths, each connecting the LCA to one of its endpoints. We perform a one-pass top-down traversal of the query tree T_Q in BFS order, and compute the descendants of all LCA paths simultaneously. For each LCA path P , we maintain a path state $Current_descendants[P]$, which functions as a mapping from the LCA candidates to their descendants at the current query vertex. As we traverse downwards, the path states at the parent u_p are propagated to the current vertex u . Specifically, when visiting u , we first identify all $v_p \in C(u_p)$ involved in any LCA path passing through its parent u_p (line 5) and cache their child candidates $C(u \mid u_p, v_p)$ (line 7) to ensure each candidate edge is processed only once. Subsequently, for each path P passing through u , we update the path state $Current_descendants[P]$ by replacing the parent candidates v_p with their cached child candidates $Cache[v_p]$ (line 9). If u is the LCA vertex of a path P , its state is initialized as an identity mapping of $C(u)$ (line 11). Once u reaches the endpoint of a path P , the current state is saved (line 13) as the mapping from the LCA candidates to the endpoint descendants.

LEMMA 5.1. *The complexity of Algorithm 5 is $O(|E_C| + |V_Q||E_Q||V_C|)$, where V_Q and E_Q are the query graph vertex and edge sets, and V_C and E_C are the vertex set and edge set in the candidate space.*

Algorithm 5: FIND LCA DESCENDANTS**Input:** The query tree T_Q , corresponding CS **Output:** LCA descendants for each LCA path

```

1 foreach  $u$  in BFS Traversal of  $T_Q$  do
2   if  $u$  is involved in any LCA path then
3      $u_p \leftarrow \text{Parent}(u)$ ;
4     if  $u_p$  exists then
5        $\text{All\_current\_descendants}$ 
6          $\leftarrow \{v_p \mid v_p \in \text{Current\_descendants}[P], \forall P \text{ covering } u_p\}$ ;
7       foreach  $v_p \in \text{All\_current\_descendants}$  do
8          $\text{Cache}[v_p] = C(u \mid u_p, v_p)$ ;
9       foreach LCA path  $P$  passing through  $u$  do
10         $\text{Current\_descendants}[P] \leftarrow \{\text{Cache}[v_p] \mid v_p \in \text{Current\_descendants}[P]\}$ ;
11   if  $u$  is the LCA of path  $P$  then
12      $\text{Current\_descendants}[P] \leftarrow C(u)$ ;
13   if  $u$  is the Endpoint of path  $P$  then
14     Save  $\text{Current\_descendants}[P]$ ;

```

5.2 Computing LCA Ratio

We view the tree paths from the LCA to the two non-tree endpoints as virtual edges. Together with the non-tree edge, these two virtual edges form a triangle. Our goal is to count, for each LCA candidate, how many combinations of endpoint candidates form valid triangles. Specifically, for a given LCA candidate, we first extract the candidate sets of the two endpoints of a Type-II non-tree edge that connected to it through the candidate tree paths. The total number of possible triangles is given by the product of the sizes of these two candidate sets, which aligns with the counting logic of Equation 2. We then explicitly examine how many candidate edges exist between the two endpoint candidate sets. Each such candidate edge, together with the connections from its endpoints to the LCA candidate, forms a valid triangle.

A single LCA vertex may be associated with multiple non-tree edges, some of which share a common endpoint. We group such non-tree edges by their shared endpoint and treat this endpoint as an anchor. Within each group, we fix one candidate path from the LCA to the anchor, and compute the fraction of candidate combinations of the other endpoints that satisfy all triangle constraints induced by the group. The final LCA ratio for an LCA candidate is obtained by multiplying the ratios over all anchor groups.

LEMMA 5.2. *Given an LCA-non-tree triangle, the LCA ratio can be computed in $O(|E_C||V_C|)$ time.*

Remark. By integrating Type-I and Type-II non-tree constraints with the existing tree-edge constraints in the candidate tree framework, our structural module bridges the gap between tree homomorphism and graph homomorphism, extending the candidate tree framework to full subgraph homomorphism counting. This extension is non-trivial, since tree homomorphism counting is polynomial-time solvable, whereas subgraph homomorphism counting itself is a #P-hard problem. Compared with Fastest, which relies on sampling over simple tree homomorphism counts, our approach moves the more complex structural-constraint computation into the counting stage and

leaves only the injectivity constraints to the sampling module. For queries that do not involve injectivity violations, the sampling stage can even be skipped entirely.

6 Local Sampling

While the non-tree edge constraints can be processed by the matrix computation-based method, the isomorphism constraint remains unsolved. This is because the candidate tree-based counting only propagates and aggregates vertex weights, rather than the specific matching states. In this section, we introduce a local sampling approach to estimate the proportion of isomorphism count within the homomorphism count. Unlike prior methods that sample the entire candidate space, we conduct sampling in a local candidate space, where isomorphism constraints arise. In particular, isomorphism violations can only occur among candidates of the same-label query vertices; even within this set, clique-structured matches cannot violate injectivity, since no self-loops exist.

Query vertices with the same label may appear at different positions in the query graph and are not necessarily adjacent. Extracting them in isolation would break the structural connectivity constraints among them. To preserve these inherent structural constraints, we directly identify a connected subgraph within the original query graph that encompasses all these same-label vertices. This connected subgraph naturally consists of both tree edges and non-tree edges from the query graph, seamlessly maintaining the exact topological relationships. As a result, the target same-label vertices form a locally connected structure where each vertex is properly constrained by its neighbors. We then perform sampling exclusively within the candidate space induced by this locally connected subgraph to estimate the isomorphism ratio.

The sampling procedure proceeds by following the traversal order of the connected subgraph. When sampling a query vertex, its feasible sampling region is bounded by its already visited neighbors, meaning that earlier sampled vertices directly constrain the subsequently sampled ones. To optimize this process, we apply a greedy selection rule: among its already visited neighbors, we select the one with the strongest constraint as its tree parent to induce the feasible sampling region, from which a candidate vertex is then uniformly sampled, with the inverse sampling probability accumulated throughout the sampling process. The pre-computed homomorphism weights $W(u, v)$ from the prior counting phase, used in a similar way to [47], are utilized to accurately reflect the matching distribution and continuously monitor the statistical variance of the random walks. The isomorphism ratio is then estimated as the weighted fraction of isomorphic samples among the homomorphic ones. For queries where the empirical samples fail to reach statistical confidence, we approximate this ratio by extrapolating the joint constraint-satisfaction probabilities for homomorphic and isomorphic samples separately from their average marginal survival rates. Additionally, a weight cutoff strategy is applied to truncate overly large sample weights, mitigating extreme variance caused by overly dense homomorphic regions.

7 Experiment

7.1 Experiment Setup

Datasets. We conduct experiments on 10 real-world datasets. The first 8 datasets have publicly available query sets and are widely used in previous works like [18, 47, 55], while the remaining 2 are large-scale graphs where we generate queries via random walks on data graphs following [18, 50]. To obtain the exact subgraph counts for each query, we use the recommended method in [50]. Given the extreme difficulty of subgraph counting, we set a timeout of 30 minutes [18, 55] for computing the ground-truth counts on the first 8 datasets, and a timeout of 2 hours [18] for the two large-scale graphs. Our experiments are evaluated on queries with obtainable ground-truth counts, which are further grouped into subsets by size for evaluation. Dataset statistics are reported in Table 1.

Table 1. Statistics of Datasets.

Dataset	$ V_G $	$ E_G $	$ \Sigma $	d	$ V_Q $
Yeast	3,112	12,519	71	9.0	4 to 32
HPRD	9,460	34,998	307	7.4	4 to 32
Human	4,674	86,282	44	36.9	4 to 20
WordNet	76,853	120,399	5	3.1	4 to 20
DBLP	317,080	1,049,866	15	6.6	4 to 16
Youtube	1,134,890	2,987,624	25	5.3	4 to 32
Eu2005	862,664	16,138,468	40	37.4	4 to 8
Patents	3,774,768	16,518,947	20	8.8	4 to 32
Twitter	41,652,230	1,202,513,344	100	57.7	4
Friendster	65,608,366	1,806,067,135	100	55.1	4 to 8

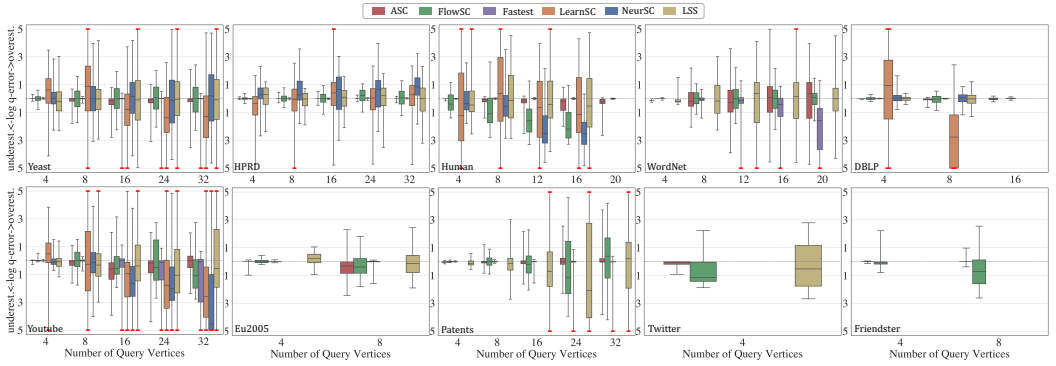


Fig. 3. Evaluating accuracy with the x-axis representing the number of query graph vertices.

Compared Algorithms. We compare ASC with 5 approximate algorithms: (1) FlowSC [18], the state-of-the-art learning-based method; (2) Fastest [47], the state-of-the-art sampling-based method; (3) LearnSC [24]; (4) NeurSC [55]; and (5) LSS [60]. We also evaluate subgraph homomorphism counting against 3 baselines: Fastest [47], Alley [30], and its index-optimized variant AlleyTPI [30]. The source code of FlowSC, Fastest, NeurSC, Alley, AlleyTPI, and LSS are publicly available, and LearnSC is obtained from the authors. All parameter settings follow their default values.

Query Setting. Following the common setting [18, 50, 55], we load a data graph and a query graph into memory each time, and report the elapsed time per data-query pair and the q-error $= \max(\frac{\max(1, c)}{\max(1, \hat{c})}, \frac{\max(1, \hat{c})}{\max(1, c)})$, where $\hat{c}$ and c are the estimated and ground truth counts, respectively. Experiments are conducted on a Ubuntu 22.04.1 system, equipped with an Intel Xeon Silver 4314 CPU @ 2.40GHz with 64 cores.

7.2 Performance Evaluation

Accuracy Comparison. Figure 3 reports the accuracy performance over all datasets. The boxplot presents the distribution of log q-error of a query graph set. The upper bound (Resp. lower bound) of a box is 75% (Resp. 25%) quantile of the q-error, the whiskers contain all q-errors from minimum to maximum, and the line in the box is the median. Values exceeding 10^5 are clipped, with bold red caps indicating truncated whiskers.

Overall, ASC demonstrates consistently higher and more stable accuracy. On the small dataset Yeast, ASC achieves stable performance across query sets of different sizes, attaining the lowest average q-error among all methods and yielding a 3.24× improvement over the state-of-the-art method FlowSC. Although the sampling-based method Fastest performs nearly perfectly on small

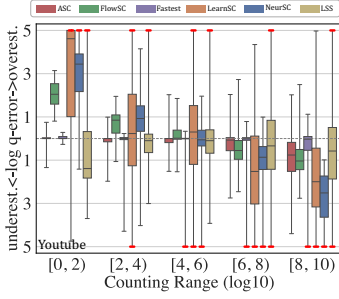


Fig. 4. Effect of counting range.

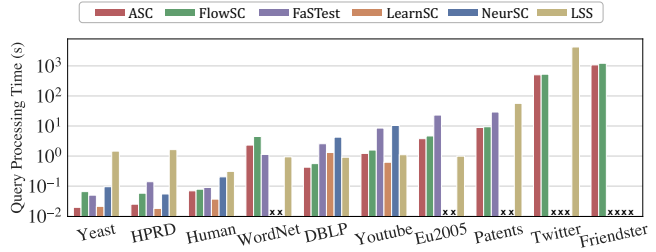


Fig. 5. Efficiency Performance.

queries in Yeast, it begins to suffer from sampling failures when the query size reaches 16, resulting in a sharp degradation in accuracy. On the other two small datasets, HPRD and Human, ASC outperforms all learning-based methods, but is slightly worse than FaSTest, as FaSTest is particularly effective when the sampling space is small. On large datasets such as Youtube and Patents, FaSTest performs well only for small query sizes. In contrast, ASC retains high accuracy on larger datasets and for larger queries. For example, on Youtube₃₂, ASC achieves the best performance with a low q-error, while FlowSC degrades to a substantially higher error and FaSTest fails entirely, yielding a q-error about 7 orders of magnitude larger than that of ASC.

In terms of learning-based methods, NeurSC and LearnSC are difficult to train on large datasets. Although LSS can handle Twitter, it is significantly outperformed by ASC. FlowSC can process all datasets. While it achieves higher accuracy than other learning-based methods, but returns lower accuracy than ASC on most datasets, except for WordNet. The exception is largely due to the low variance of the true counts in WordNet, where many queries share identical counts. Such distributions favor regression-based learning methods because of their tendency to regress toward the mean [18]. Consequently, while LearnSC and NeurSC do not benefit from this property due to their difficulty in training, FlowSC and LSS perform particularly well on WordNet.

In addition, these learning-based methods rely heavily on supervision, where the performance is greatly influenced by the number of training samples. As a result, even FlowSC, which performs best among learning-based methods, obtains poor accuracy on the Human dataset. Moreover, on Human₂₀ and DBLP₁₆, the scarcity of effective training samples prevents the models from learning. In contrast, ASC and FaSTest are not constrained by the number of training samples, performing well even on datasets with limited samples. Moreover, only ASC and FlowSC can scale to the billion-scale dataset, where ASC still achieves higher accuracy than FlowSC.

Counting Range Analysis. We conduct an experiment on a selected dataset to evaluate how the ground truth count of query graphs influences the distribution of q-error. As shown in Figure 4, our algorithm performs consistently across different count ranges. In the range with larger counts, the accuracy becomes slightly worse, mainly due to the scale effect: queries with larger candidate spaces naturally accumulate more approximation error. The learning-based methods FlowSC, LearnSC, and NeurSC tend to overestimate small-count queries, and underestimate large-count queries, which is consistent with the regression-to-the-mean effect commonly observed in regression-based approaches [18]. The active learning strategy of LSS alleviates this issue. The accuracy of FaSTest does not show a clear pattern across different counting ranges. Its performance depends on sampling difficulty, and sampling failures are observed in every interval from 10^2 to 10^{10} .

Table 2. Homomorphism evaluation.

Dataset	ASC		Fastest		Alley		AlleyTPI	
	q-error	Time	q-error	Time	q-error	Time	q-error	Time
Yeast	1.05	0.02	1.02	0.04	1×10^4	0.17	1×10^3	0.17
HPRD	1.01	0.03	1.00	0.11	22.45	0.01	13.15	0.01
Human	1.18	0.08	1.01	0.09	7×10^8	6×10^4	7×10^8	6×10^4
WordNet	1.00	2.23	1.08	0.25	11.85	7×10^3	5.90	4×10^4
DBLP	1.09	0.46	1.01	1.64	-	-	-	-
Youtube	2.74	1.32	1.32	6.10	-	-	-	-
Eu2005	6.48	3.92	1.14	16.69	-	-	-	-
Patents	1.22	8.83	1.05	38.52	-	-	-	-
Twitter	1.04	505.05	-	-	-	-	-	-
Friendster	1.15	1×10^3	-	-	-	-	-	-

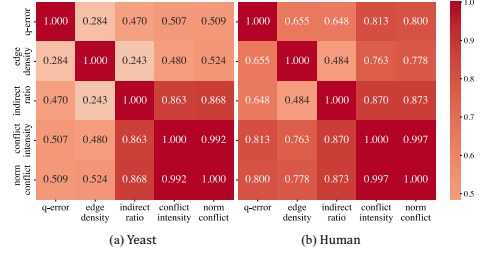


Fig. 6. Correlation heatmap.

Query Processing Time. Figure 5 reports the average processing time. A cross indicates that a method encounters training failure or out-of-memory. In general, ASC is highly competitive on both small and large datasets. On small datasets such as Yeast, HPRD, and Human, ASC is faster than Fastest and achieves runtime comparable to the fastest learning-based method, FlowSC. ASC first incorporates structural constraints into a polynomial-time counting stage, which is efficient in practice. Its sampling stage is then performed only on local subspaces where injectivity violations may occur. If a query does not induce any injectivity violations, the sampling stage can be skipped entirely. Fastest performs sampling based on tree homomorphisms, its sampling is conducted over the full candidate space, which is often much larger than the solution space, making isomorphic samples difficult to obtain. For example, in our preliminary experiment of counting a same-label 32-clique in a same-label 32-clique data graph, Fastest did not return a result within 100 seconds, while ASC skipped sampling entirely because this query does not produce injectivity violations. On the WordNet dataset, NeurSC and LearnSC cannot be applied due to excessive training time. Both ASC and FlowSC are slower than Fastest because their filtering technique BipartitePlus is less efficient on graphs with highly concentrated label distributions. On larger datasets such as Youtube, Eu2005, and Patents, ASC and FlowSC exhibit similar runtime and show a clear advantage over Fastest. As the graph size increases, the full candidate space becomes much larger, causing Fastest to spend more time obtaining isomorphic samples. Due to their complex network architectures, NeurSC and LearnSC cannot scale to large datasets. Although LSS uses a lightweight model and is therefore more scalable, it still cannot process the largest dataset, Friendster. Among learning-based approaches, FlowSC offers the best efficiency and scalability, mainly due to its highly efficient one-pass bottom-up flow-learning model. Overall, ASC achieves efficiency comparable to FlowSC on large and billion-scale datasets such as Twitter and Friendster.

7.3 Testing on Homomorphism

We conduct experiments for subgraph homomorphism counting to evaluate the effectiveness of the main counting module, i.e., matrix computation-based counting (section 4) together with the LCA ratio (section 5). Since computing the exact subgraph homomorphism counts is also computationally expensive, we select those queries with existing subgraph isomorphism ground truth that do not induce isomorphism constraints. Specifically, a query graph will not violate isomorphism constraints if its vertex labels are unique, or if vertices with the same label form a clique. In these cases, the subgraph isomorphism count coincides with the subgraph homomorphism count. We select all such queries from each dataset for homomorphism testing. We compare ASC with Fastest and the homomorphism counting method Alley and its variant AlleyTPI.

As shown in Table 2, ASC provides a better accuracy-efficiency trade-off, it achieves accuracy comparable to Fastest with substantially faster runtime, and clearly outperforms Alley and AlleyTPI

in both accuracy and efficiency. Alley and AlleyTPI frequently suffer from sampling failures, which leads to unsatisfactory performance even on small datasets. In addition, AlleyTPI incurs extra overhead for index construction, making it unsuitable for random data-query pairs. We omit their results on larger datasets. Fastest achieves high accuracy but at the cost of substantial runtime. In contrast, ASC attains accuracy comparable to Fastest on most datasets while requiring significantly less time. Although ASC is less accurate than Fastest on Youtube and Eu2005, where the candidate space is large, its accuracy is acceptable given the substantially lower runtime. Although WordNet also has a larger candidate space, it contains only four 4-size queries without isomorphism constraints, and ASC achieves near-zero error on these queries.

7.4 Evaluation of Individual Techniques

Error Characterization of Type-I Module. We conduct error characterization experiments on small datasets Yeast and Human to analyze the factors that affect algebraic counting. However, error characterization requires exact counting of graph homomorphisms, which is extremely time-consuming even on small datasets. To tackle the computational difficulty, we reduce the exact counting to the child neighborhood of a single candidate v , as defined in Equation 3, and compare $W(u, v)$ with the approximate count produced by the algebraic method. To better investigate the underlying characteristics of our method, we consider 4 scale-insensitive high-order features of the constrained sequences: edge density (the number of internal edges/sequence length), indirect ratio (the number of indirect edges/internal edges), conflict intensity (the number of nested or intersecting cycles/indirect edge), and norm conflict (the number of nested or intersecting cycles/sequence length). The correlations are quantified using Spearman's Rho as shown in Figure 6. As observed, the norm conflict and conflict intensity are the most significant factors of q-error on Yeast and Human, respectively. Since the two factors are highly correlated, the results suggest that the conflict pairs (i.e., nested or intersecting indirect edges within the sequence) are the primary error source.

We also observe that the strongest correlation with q-error reaches around 0.5 on Yeast and 0.8 on Human. This is because our analysis focuses on the structural features of constrained sequences in the query graph, while the corresponding candidate space also substantially affects accuracy. Interestingly, it is observed that, for a sequence s under u , even if we cannot guarantee the exact solution due to nested or indirect edges in s , the weights $W(u, v)$ for some candidates v may still achieve perfect accuracy with q-error = 1. This is attributed to the effort that we introduce a simple penalty based on precomputed reachable weight ratios along the reverse sequence, which can suppress the accumulation of invalid weights during propagation. Although this penalty cannot guarantee counting that exactly conforms to the matching states in the candidate space, it performs very well in practice, which is frequently observed in our experiments.

Impact of Sequence Order. From the perspective of the FAQ framework [2], our constrained sequence defines a variable elimination order for the underlying subquery. Consequently, the resulting matrix chain multiplication can be viewed as a specialized execution plan for the associated sum-product query. We evaluate 6 constrained sequence ordering strategies, namely (1) Random ordering; (2) Min-Candidates, prioritizing query vertices with the smallest candidate set; (3) Max-Degree, prioritizing vertices with the highest degree; (4) Max-Candidates, prioritizing vertices with the largest candidate set; (5) Min-Degree, prioritizing vertices with the lowest degree; and (6) Heuristic, a greedy strategy that minimizes nested or intersecting indirect edges within the sequence. The average q-error of each $W(u, v)$ over queries from Yeast and Human for each strategy is presented in Table 3. The results show that Heuristic performs the best, which aligns with our previous analysis, i.e., fewer conflicting indirect edges bring less accumulated error. Additionally,

Table 3. The impact of different orders.

Order	Random	Min-Candidates	Max-Degree	Max-Candidates	Min-Degree	Heuristic
Yeast	1.23	1.24	1.44	1.47	1.27	1.21
Human	2.23	4.64	9.28	9.61	2.88	2.13

Table 4. Effect of different modules.

Module	(-) Type-I	(-) Type-II	(-) Iso	Iso-only	Fastest	ASC
q-error	6.8×10^9	7.3×10^{15}	1.2×10^6	2.1×10^{16}	7.5×10^6	97.69
Time (s)	1.36	1.11	1.30	1.09	3.37	1.37

all strategies consume comparable query time as expected, since the number of vertices and edges in the sequence remains unchanged across different orders.

Analysis on Type-II module. For the Type-II module, when computing the LCA Ratio, we only account for the number of valid combinations in the LCA-endpoints triangle. For the two virtual edges between the LCA and the endpoints, we consider only reachability and ignore the actual number of paths in the candidate space for efficiency. In other words, regardless of how many tree paths connect an endpoint candidate to an LCA candidate, we treat them as a single path. Due to the infeasibility of exact algorithms, we evaluate the accuracy of the LCA Ratio for each LCA candidate. In particular, for each LCA candidate, we compute the *variance* in the number of paths from endpoint candidates for each non-tree edge to that LCA candidate, and analyze its correlation with estimation accuracy. The experiment results report that Spearman's rho reaches 0.65 and 0.42 on Yeast and Human, respectively, both showing significant p-values, confirming our hypothesis.

Ablative Analysis. We conduct ablation studies on 3 modules for handling Type-I non-tree edge, Type-II non-tree edge, and Iso constraints. All variants follow the framework of candidate tree counting. (-) Type-I denotes the model without Type-I non-tree constraint computation, with (-) Type-II and (-) Iso defined similarly. Iso-only removes both Type-I and Type-II computations. Iso-only removes both Type-I and Type-II non-tree constraint computations. We conduct experiments on 4 representative datasets, namely Yeast, Human, WordNet, and Eu2005, with different characteristics on average degree and label distributions.

Accuracy. It is reported on Table 4 that ASC achieves the best accuracy performance with 97.69 q-error, whereas others perform much worse. For example, Iso-only generates an extremely high q-error of 2.1×10^{16} . Surprisingly, the model without the Iso constraint module (i.e., (-) Iso) performs fairly well and even outperforms Fastest. This implies that our Type-I and Type-II modules can effectively capture structural constraints of the graph. In contrast, without Type-I and Type-II modules, the tree-homomorphism-based Iso module alone cannot achieve satisfactory accuracy.

Query Time. As reported in Table 4, all variants consume almost the same amount of query time. This indicates that removing a specific module does not bring a clear efficiency improvement. The minor time differences are mainly attributed to the distribution of constraints in the query graph.

7.5 Discussion

ASC achieves high accuracy in homomorphism counting as shown in Table 2, benefiting from its precise handling of structural constraints, especially the non-tree ones. While it also performs well

for isomorphism counting, the iso ratio module remains the primary error source, as estimating the isomorphism ratio via sampling is inherently less accurate than the algebraic counting. We see two directions to improve it. First, the current sampling could be replaced by more robust alternatives, since sampling still faces inherent limitations: global sampling (e.g., in Fastest and Alley) often becomes inaccurate or fails in vast search spaces, and although our local sampling alleviates this by skipping query regions that cannot violate injectivity and sampling only over the conflict-prone regions, it does not fundamentally escape the accuracy limitations of sampling. Second, although the structural constraints, including non-tree edges, are already incorporated into the counting, their effect is currently confined to the weight accumulation in the counting stage. Another promising direction is thus to let such constraints act directly on the Cartesian-product space underlying the counting, pruning its enumerable states rather than only correcting the accumulated weights as in the current design.

8 Related Work

Subgraph Matching. Subgraph matching research has been dominated by the filtering-ordering-enumeration paradigm. Recent studies have enhanced this framework through efficient filtering [6, 7, 18–20, 22, 47], optimized ordering strategies [7, 9, 15, 46], and advanced enumeration techniques [50, 57]. Existing enumeration techniques primarily fall into two categories: exploration-based and decomposition-based methods. Within exploration-based approaches, performance is often improved via search space pruning [5, 19, 58], intermediate result reuse [26, 28, 29, 36, 37], and recursion reduction [27, 36, 49, 56]. Alternatively, decomposition-based methods [1, 31, 32, 35, 38, 43, 51] partition the query graph and employ join techniques, such as binary joins [31, 32, 52] and worst-case optimal joins (WCOJ) [1, 38, 40, 51].

Subgraph Counting. Subgraph counting methods primarily employ three approximate paradigms. Summarization-based methods [16, 23, 39, 48] decompose queries and aggregate substructure counts, yet often struggle with accuracy due to independence assumptions. Sampling-based approaches [10, 13, 34, 47] are widely adopted but face inherent sampling failures, despite recent efforts in search-space compression and strategy optimization. Finally, learning-based methods [18, 55, 59, 60] leverage GNNs to capture graph interactions, but their effectiveness is limited by the complex modeling between graph features and counts, high training overhead, and heavy dependence on ground-truth supervision.

9 Conclusion

In this paper, we study the problem of subgraph counting. We propose an algebraic counting method based on the candidate tree-based counting framework. Using fast matrix computation operations, we can obtain accurate subgraph homomorphism counting with high efficiency. Based on this, we obtain the final subgraph isomorphism count via a local sampling strategy. Extensive experiments demonstrate the superior performance of our proposals.

Acknowledgments

This work was supported by the China Scholarship Council (202308440228), the National Natural Science Foundation of China (62572137, U2241211), the Guangdong Basic and Applied Basic Research Foundation (2025A1515011716), the Major Key Project of PCL (PCL2024A05, PCL2025A16), and the Australian Research Council (CE230100001, DP260100689, DE250100226).

References

- [1] Christopher R Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS)* 42, 4 (2017), 1–44.
- [2] Mahmoud Abo Khamis, Hung Q Ngo, and Atri Rudra. 2016. FAQ: questions asked frequently. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 13–28.
- [3] Noga Alon, Raphael Yuster, and Uri Zwick. 1997. Finding and counting given length cycles. *Algorithmica* 17, 3 (1997), 209–223.
- [4] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter Boncz, George Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan Sequeda, et al. 2018. G-CORE: A core for future graph query languages. In *Proceedings of SIGMOD*. 1421–1432.
- [5] Junya Arai, Yasuhiro Fujiwara, and Makoto Onizuka. 2023. Gup: Fast subgraph matching by guard-based pruning. *Proceedings of SIGMOD* 1, 2 (2023), 1–26.
- [6] Bibek Bhattarai, Hang Liu, and H. Howie Huang. 2019. CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching. In *Proceedings of SIGMOD*. 1447–1462.
- [7] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient Subgraph Matching by Postponing Cartesian Products. In *Proceedings of SIGMOD*. 1199–1214.
- [8] Angela Bonifati, Wim Martens, and Thomas Timm. 2020. An analytical study of large SPARQL query logs. *The VLDB Journal* 29, 2 (2020), 655–679.
- [9] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis Shasha, and Alfredo Ferro. 2013. A subgraph isomorphism algorithm and its application to biochemical data. *BMC Bioinformatics* (2013).
- [10] Marco Bressan, Stefano Leucci, and Alessandro Panconesi. 2019. Motivo: Fast Motif Counting via Succinct Color Coding and Adaptive Sampling. In *Proc. VLDB Endow.* 1651–1663.
- [11] Walter Cai, Magdalena Balazinska, and Dan Suciu. 2019. Pessimistic Cardinality Estimation: Tighter Upper Bounds for Intermediate Join Cardinalities. In *Proceedings of SIGMOD*. 18–35.
- [12] Mario Cannataro and Pietro H Guzzi. 2012. *Data management of protein interaction networks*. Vol. 17. John Wiley & Sons.
- [13] Xiaowei Chen and John C. S. Lui. 2016. Mining Graphlet Counts in Online Social Networks. In *Proceedings of ICDM*. 71–80.
- [14] Yunyoung Choi, Kunsoo Park, and Hyunjoon Kim. 2023. Bice: Exploring compact search space by using bipartite matching and cell-wide verification. *Proc. VLDB Endow.* 16, 9 (2023), 2186–2198.
- [15] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento. 2004. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26, 10 (2004), 1367–1372.
- [16] Kyle Deeds, Diandre Sabale, Moe Kayali, and Dan Suciu. 2025. Color: A framework for applying graph coloring to subgraph cardinality estimation. *Proc. VLDB Endow.* 2 (2025), 130–143. doi:10.14778/3705829.3705834
- [17] Kyle B Deeds, Dan Suciu, and Magdalena Balazinska. 2023. Safebound: A practical system for generating cardinality bounds. *Proceedings of SIGMOD* 1, 1 (2023), 1–26.
- [18] Qiuyu Guo, Jianye Yang, Wenjie Zhang, Hanchen Wang, Ying Zhang, and Xuemin Lin. 2025. Efficient and Accurate Subgraph Counting: A Bottom-up Flow-learning Based Approach. *Proc. VLDB Endow.* 18, 8 (2025), 2695–2708. doi:10.14778/3742728.3742758
- [19] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. 2019. Efficient Subgraph Matching: Harmonizing Dynamic Programming, Adaptive Matching Order, and Failing Set Together. In *Proceedings of SIGMOD*. 1429–1446.
- [20] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. Turboiso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of SIGMOD*. 337–348.
- [21] Stephen Harris and Nigel Shadbolt. 2005. SPARQL query processing with conventional relational database systems. In *International Conference on Web Information Systems Engineering*. Springer, 235–244.
- [22] Huahai He and Ambuj K. Singh. 2008. Graphs-at-a-Time: Query Language and Access Methods for Graph Databases. In *Proceedings of SIGMOD*. 405–418.
- [23] Tomaž Hočevar and Janez Demšar. 2014. A combinatorial approach to graphlet counting. *Bioinformatics* 30, 4 (2014), 559–565.
- [24] Wenzhe Hou, Xiang Zhao, and Bo Tang. 2024. LearnSC: An Efficient and Unified Learning-Based Framework for Subgraph Counting Problem. In *Proceedings of ICDE*. IEEE, 2625–2638.
- [25] Alon Itai and Michael Rodeh. 1977. Finding a minimum circuit in a graph. In *Proceedings of the ninth annual ACM symposium on Theory of computing*. 1–10.
- [26] Xun Jian, Zhiyuan Li, and Lei Chen. 2023. Suff: Accelerating subgraph matching with historical data. *Proc. VLDB Endow.* 16, 7 (2023), 1699–1711.

- [27] Tatiana Jin, Boyang Li, Yichao Li, Qihui Zhou, Qianli Ma, Yunjian Zhao, Hongzhi Chen, and James Cheng. 2023. Circinus: Fast redundancy-reduced subgraph matching. *Proceedings of SIGMOD* 1, 1 (2023), 1–26.
- [28] Hyunjoon Kim, Yunyoung Choi, Kunsoo Park, Xuemin Lin, Seok Hee Hong, and Wook Shin Han. 2021. Versatile Equivalences: Speeding up Subgraph Query Processing and Subgraph Matching. In *Proceedings of SIGMOD*. 925–937.
- [29] Hyunjoon Kim, Yunyoung Choi, Kunsoo Park, Xuemin Lin, Seok-Hee Hong, and Wook-Shin Han. 2023. Fast subgraph query processing and subgraph matching via static and dynamic equivalences. *The VLDB journal* 32, 2 (2023), 343–368.
- [30] Kyoungmin Kim, Hyeonji Kim, George Fletcher, and Wook Shin Han. 2021. Combining Sampling and Synopses with Worst-Case Optimal Runtime and Quality Guarantees for Graph Pattern Cardinality Estimation. In *Proceedings of SIGMOD*. 964–976.
- [31] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. 2017. Scalable subgraph enumeration in mapreduce: a cost-oriented approach. *The VLDB Journal* 26, 3 (2017), 421–446.
- [32] Longbin Lai, Lu Qin, Xuemin Lin, Ying Zhang, Lijun Chang, and Shiyu Yang. 2016. Scalable distributed subgraph enumeration. *Proc. VLDB Endow.* 10, 3 (2016), 217–228.
- [33] John M Lewis. 1978. On the complexity of the maximum subgraph problem. In *Proceedings of the tenth annual ACM symposium on Theory of computing*. 265–274.
- [34] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2016. Wander Join: Online Aggregation via Random Walks. In *Proceedings of SIGMOD*. 615–629.
- [35] Qiyang Li, Jeffrey Xu Yu, and Zongyan He. 2025. Subgraph Matching: A New Decomposition Based Approach. *Proc. VLDB Endow.* 18, 11 (2025), 4282–4294.
- [36] Yujie Lu, Zhijie Zhang, and Weiguo Zheng. 2025. B(S)X: Subgraph Matching with Batch Backtracking Search. *Proceedings of SIGMOD* 3, 1 (2025), 1–27.
- [37] Yujie Lu, Zhijie Zhang, Weiguo Zheng, and Lei Zou. 2025. Accelerating Subgraph Matching through Fine-grained and Powerful Equivalences. *Proc. VLDB Endow.* 18, 11 (2025), 3896–3909.
- [38] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *Proc. VLDB Endow.* 12, 11 (2019).
- [39] Thomas Neumann and Guido Moerkotte. 2011. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *Proceedings of ICDE*. 984–994.
- [40] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2012. Worst-Case Optimal Join Algorithms: [Extended Abstract]. In *Proceedings of the ACM PODS*. 37–48.
- [41] Himchan Park and Min-Soo Kim. 2018. EvoGraph: An effective and efficient graph upscaling method for preserving graph properties. In *Proceedings of SIGKDD*. 2051–2059.
- [42] N Pržulj, Derek G Corneil, and Igor Jurisica. 2006. Efficient estimation of graphlet frequency distributions in protein–protein interaction networks. *Bioinformatics* 22, 8 (2006), 974–980.
- [43] Miao Qiao, Hao Zhang, and Hong Cheng. 2017. Subgraph matching: on compression and computation. *Proc. VLDB Endow.* 11, 2 (2017), 176–188.
- [44] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time constrained cycle detection in large dynamic graphs. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1876–1888.
- [45] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2017. The ubiquity of large graphs and surprising challenges of graph processing. In *Proc. VLDB Endow.* 420–431.
- [46] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. In *Proc. VLDB Endow.*, Vol. 1. VLDB Endowment, 364–375.
- [47] Wonseok Shin, Siwoo Song, Kunsoo Park, and Wook-Shin Han. 2024. Cardinality Estimation of Subgraph Matching: A Filtering-Sampling Approach. In *Proc. VLDB Endow.* 1697–1709.
- [48] Giorgio Stefanoni, Boris Motik, and Egor V. Kostylev. 2018. Estimating the Cardinality of Conjunctive Queries over RDF Data Using Graph Summarisation. In *Proceedings of WWW*. 1043–1052.
- [49] Shixuan Sun, Yulin Che, Lipeng Wang, and Qiong Luo. 2019. Efficient parallel subgraph enumeration on a single machine. *Proceedings of ICDE (2019)*, 232–243.
- [50] Shixuan Sun and Qiong Luo. 2020. In-Memory Subgraph Matching: An In-depth Study. In *Proceedings of SIGMOD*. 1083–1098.
- [51] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. 2020. Rapidmatch: A holistic approach to subgraph query processing. In *Proc. VLDB Endow.*, Vol. 14. 176–188.
- [52] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. 2012. Efficient Subgraph Matching on Billion Node Graphs. *Proc. VLDB Endow.* 5, 9 (2012).
- [53] J. R. Ullmann. 1976. An Algorithm for Subgraph Isomorphism. *J. ACM* (1976), 31–42.
- [54] Vladimir Vacic, Lilia M Iakoucheva, Stefano Lonardi, and Predrag Radivojac. 2010. Graphlet kernels for prediction of functional residues in protein structures. *Journal of Computational Biology* 17, 1 (2010), 55–72.

- [55] Hanchen Wang, Rong Hu, Ying Zhang, Lu Qin, Wei Wang, and Wenjie Zhang. 2022. Neural Subgraph Counting with Wasserstein Estimator. In *Proceedings of SIGMOD*. 160–175.
- [56] Rongjian Yang, Zhijie Zhang, Weiguo Zheng, and Jeffrey Xu Yu. 2023. Fast continuous subgraph matching over streaming graphs via backtracking reduction. *Proceedings of SIGMOD* 1, 1 (2023), 1–26.
- [57] Zhijie Zhang, Yujie Lu, Weiguo Zheng, and Xuemin Lin. 2024. A Comprehensive Survey and Experimental Study of Subgraph Matching: Trends, Unbiasedness, and Interaction. In *Proceedings of SIGMOD*, Vol. 2. ACM New York, NY, USA, 1–29.
- [58] Zhijie Zhang and Weiguo Zheng. 2025. BEE: Towards Redundancy Reduction via Block-Separator Decomposition for Subgraph Matching. *Proceedings of SIGMOD* 3, 4 (2025), 1–27.
- [59] Kangfei Zhao, Jeffrey Xu Yu, Qiyang Li, Hao Zhang, and Yu Rong. 2023. Learned sketch for subgraph counting: a holistic approach. *The VLDB Journal* 32, 5 (2023), 937–962.
- [60] Kangfei Zhao, Jeffrey Xu Yu, Hao Zhang, Qiyang Li, and Yu Rong. 2021. A Learned Sketch for Subgraph Counting. In *Proceedings of SIGMOD*. 2142–2155.

Received 17 January 2026; revised 19 May 2026; accepted 12 June 2026