Efficient and Accurate Subgraph Counting: A Bottom-up Flow-learning Based Approach

Qiuyu Guo
Guangzhou University
University of New South Wales
qiuyu.guo@unsw.edu.au

Jianye Yang*
Guangzhou University
PengCheng Laboratory
jyyang@gzhu.edu.cn

Wenjie Zhang
University of New South Wales
wenjie.zhang@unsw.edu.au

Hanchen Wang
University of New South Wales
hanchen.wang@unsw.edu.au

Ying Zhang
Zhejiang Gongshang University
ying.zhang@zjgsu.edu.cn

Xuemin Lin
Shanghai Jiao Tong University
xuemin.lin@sjtu.edu.cn

ABSTRACT

Subgraph counting is a fundamental problem in graph analytics with broad applications, yet remains computationally intractable due to its #P-hardness. To address this, numerous approximate solutions have been proposed, though they often suffer from limited efficiency and accuracy. In this paper, we introduce FlowSC, a novel approach that achieves both high accuracy and efficiency in subgraph counting. Our method starts with an enhanced candidate filtering algorithm, which significantly improves the pruning capability of bipartite graph-based techniques with minimal overhead. Building on this, we propose a bottom-up flow-learning model based on a new Graph Neural Network (GNN) architecture. By employing a carefully designed message-passing mechanism, the model explicitly controls the direction, range, and iterations of information flow, enabling a simulation of the candidate tree-based counting process. This mechanism is further empowered by a customized message aggregation technique, alongside a pretraining strategy that facilitates model training. Extensive experiments show that FlowSC can achieve up to 4 orders of magnitude improvement in accuracy and 3× improvement in efficiency over the baselines across datasets, while scaling to billion-edge graphs.

PVLDB Reference Format:
Qiuyu Guo, Jianye Yang, Wenjie Zhang, Hanchen Wang, Ying Zhang, and Xuemin Lin. Efficient and Accurate Subgraph Counting: A Bottom-up Flow-learning Based Approach. PVLDB, 18(8): 2695 - 2708, 2025.
doi:10.14778/3742728.3742758

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/EricaGuoQiuyu/FlowSC.

1 INTRODUCTION

Subgraph isomorphism is an essential concept in analyzing graph-structured data. Among its related problems, subgraph matching and subgraph counting have attracted extensive research attention. Given a query graph Q and a data graph G, subgraph matching aims to find all isomorphic matches of Q in G, while subgraph counting focuses on computing the total number of such matches. They are both important tasks with a wide range of real-world applications, including analyzing social networks [13, 17, 48], understanding biological networks [11, 44, 45, 54], and optimizing graph query plans [2, 22]. This paper focuses on subgraph counting.

Despite its importance, subgraph counting is computationally challenging due to its #P-hardness [47]. This challenge is further amplified by the worst-case exponential number of possible matches. For example, in the dataset Yeast, which only contains 3112 vertices, the number of matches for a single query with 32 vertices can exceed 10^8. Clearly, exact solutions are limited to query-data graph pairs that yield a feasible number of matches, even when data graphs are relatively small. To broaden the application range, recent research efforts have been devoted to developing approximate solutions [9, 10, 13, 26, 29, 35, 36, 42, 47, 49, 55, 72].

Existing Solutions and Limitations. Existing approximate solutions for subgraph counting can be classified into three categories: summarization-based [10, 42, 49], sampling-based [9, 13, 29, 35, 47], and learning-based methods [26, 36, 55, 72]. Summarization-based methods follow a decomposition-aggregation framework. The query graph is first decomposed into substructures, and the counts of these substructures are then aggregated to obtain the final count. However, these methods assume that the counts of these substructures are independent, which is often impractical for real-world graphs, and thus offer inaccurate estimation.

Sampling-based methods infer subgraph counts by evaluating the frequency of isomorphic matches among the subgraphs sampled from the data graph. Although these methods can achieve high efficiency, they often suffer from sampling failures caused by the vast sample space, especially when handling complex queries or large graphs. The state-of-the-art algorithm FaSTest [47] mitigates this issue by reducing the sample space through effective candidate space refinement, albeit at the cost of expensive precomputed indexes. Nevertheless, inherent sampling failures are still not eliminated, leading to low estimation accuracy.

Recently, learning-based models have attracted growing attention in subgraph counting. NSIC [36] first encodes the query and data graphs using neural networks, followed by an attention-based network to model their interactions. Due to the large parameter space and the lack of effective isomorphism capturing, NSIC suffers from severe efficiency and accuracy issues. LSS [72] adopts

*Corresponding author

Proceedings of the VLDB Endowment, Vol. 18, No. 8 ISSN 2150-8097.
doi:10.14778/3742728.3742758

a learning-based decomposition-aggregation approach: it decomposes the query graph and encodes the substructures with data graph information, then aggregates the features through a self-attention mechanism. However, it relies on simplistic representations of the data graph and lacks effective modeling of query-data correlations. NeurSC [55] reduces redundancy in the data graph through candidate filtering and enhances query-data correlation learning with a Wasserstein estimator, yet it does not capture the relationship between structural features and isomorphic counts. The latest method LearnSC [26] jointly decomposes the query and data graphs, and applies contrastive interaction to model their relationship. However, it still fails to capture the connection between structural features and isomorphic counts and exhibits instability during training. In summary, despite their diverse strategies, these methods still have three main limitations: (1) insufficient reduction of redundancy on data graph, (2) inability to learn isomorphic patterns from structural features, and (3) lack of explicit modeling between structural features and subgraph counts. These limitations lead to unsatisfactory performance in both efficiency and accuracy.

Contributions. In this paper, we propose a new learning-based method, termed FlowSC. To alleviate the limitations of existing solutions, we propose two techniques, including a candidate space refinement algorithm and a novel bottom-up flow-learning model.

First, we propose an efficient and effective candidate space (CS) refinement algorithm, termed BipartitePlus. The state-of-the-art CS refinement method [47] enhances the pruning power of the bipartite graph-based approach via precomputed indexes of substructures (e.g., triangles and four-cycles). We observe that the computation of such substructures is costly, especially for large data graphs. Motivated by this, we improve the filtering performance of the bipartite graph by considering the connectivity between neighbors of the matching vertex with little overhead. Our empirical study suggests that BipartitePlus achieves similar pruning capacity but much better efficiency than the precomputed indexes.

Second, we propose a novel bottom-up flow-learning model, which is inspired by the candidate tree-based counting approach. This model includes a new GNN architecture, which can accurately simulate the information flow in the candidate trees via a one-pass bottom-up message-passing mechanism. Via this carefully designed message-passing mechanism, we can control the direction, range, and iterations of information propagation, so as to avoid the limitations existing in generic GNNs. To deal with the complex matching conditions between the query graph and the data graph, we propose a customized message aggregating method via a cross-graph attention mechanism. This method leverages the structural information of the query graph to guide the information aggregation of vertices in the candidate space, and thus offers a more accurate simulation of the matching condition checks. Moreover, we introduce a pretraining technique to improve training stability.

We conduct extensive experiments on 10 real-world datasets to evaluate FlowSC. The experimental results demonstrate that FlowSC can achieve up to 4 orders of magnitude accuracy and 3× query efficiency improvement over its counterparts, averaged across datasets. Compared to the state-of-the-art method FaSTest, FlowSC performs slightly worse on simple query-data pairs but achieves much better accuracy and stability on complex ones.

- We propose a new candidate space filtering algorithm BipartitePlus, which improves the pruning power of the bipartite graph-based method with little overhead.
- We propose a novel bottom-up flow-learning model, which utilizes a one-pass message-passing mechanism, optimized by customized message aggregation and pretraining techniques.
- Extensive experimental results on real graphs demonstrate the outstanding performance of our method FlowSC.

2 PRELIMINARIES

2.1 Problem Definition

In this paper, we focus on a connected, undirected, and vertex-labeled graph $g = (V, E, L)$, where V is a set of vertices, $E \subseteq V \times V$ is a set of edges, and L is a labeling function that maps each vertex $v \in V$ to a label in a label set Σ, denoted as $L(v)$. Given a vertex $v \in V$, the neighbors of v, denoted as $N(v)$, are all adjacent vertices of v, i.e., $N(v) = \{v' \in V \mid (v, v') \in E\}$. The degree of v, denoted as $d(v)$, is the number of neighbors of v, i.e., $d(v) = |N(v)|$. For presentation clarity, we use $Q = (V_Q, E_Q, L)$ and $G = (V_G, E_G, L)$ to denote the query graph and data graph, respectively. Besides, we use u and v to represent a vertex in the query graph and data graph, respectively.

Definition 2.1 (Subgraph Isomorphism). *Given a query graph $Q = (V_Q, E_Q, L)$ and a data graph $G = (V_G, E_G, L)$, Q is subgraph isomorphic to G if and only if there exists an injective mapping f from V_Q to V_G such that $\forall u \in V_Q$, $L(u) = L(f(u))$ and $\forall (u, u') \in E_Q$, $(f(u), f(u')) \in E_G$.*

We call an injective mapping from vertices in Q to vertices in G as a *subgraph isomorphic embedding* of Q in G. We use the term *embedding* or *match* to simply refer to subgraph isomorphic embedding when the context is clear.

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

It is clear that all subgraph matching algorithms can also be used for subgraph counting. However, exact subgraph matching or counting is computationally challenging or even infeasible, especially for complex queries or large data graphs. In this paper, we focus on approximate solutions for subgraph counting.

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

2.2 Candidate Space

To improve the efficiency of subgraph matching or counting, an important task is to remove the unnecessary vertices and edges beforehand. The surviving vertices and edges are organized by an auxiliary data structure, called *candidate space (CS)*. We first give the necessary concepts relevant to CS, and then introduce useful filtering techniques for constructing a compact CS.

Definition 2.3 (Candidate Vertex). *Given a vertex $u \in V_Q$, a vertex $v \in V_G$ is a candidate vertex of u, if there exists a match of Q in G that maps v to u. We use $C(u)$ to denote a set of candidate vertices of u.*

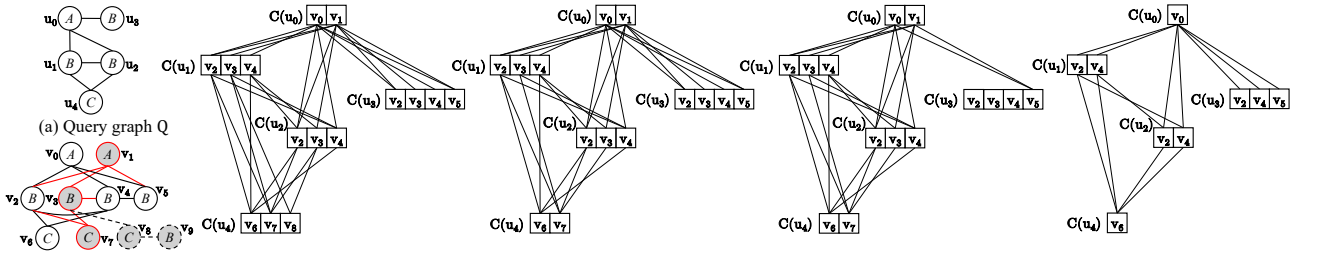


Figure 1: A running example of candidate space refinement processing for a query graph Q and data graph G.

Definition 2.4 (Conditional Candidate Vertex). *Given two vertices $u \in V_Q$ and $v \in V_G$, for a vertex $u' \in N(u)$, a vertex $v' \in N(v)$ is a conditional candidate vertex of u' if there exists a match of Q in G that maps v' to u' under the condition that u matches v. We use $C(u' \mid u, v)$ to denote all conditional candidate vertices of u'.*

Definition 2.5 (Candidate Edge). *Given an edge $(u, u') \in E_Q$, an edge $(v, v') \in E_G$ is a candidate edge of (u, u'), if there exists a match of Q in G that maps u to v and u' to v'. We use $C(u, u')$ to denote a set of candidate edges of (u, u').*

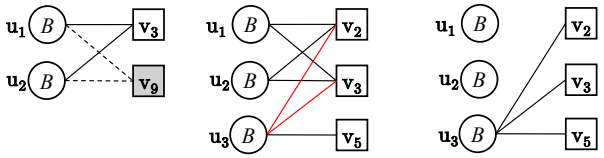
The candidate vertex set of each vertex and the candidate edge set of each edge in Q collectively define the candidate space CS. According to the above definitions, it is evident that all embeddings of Q in G are preserved in CS. We call this property of CS as *complete*. This implies that, to find all embeddings of Q, it is sufficient to only consider CS rather than G. To improve efficiency and accuracy, the key is to refine the candidate space as much as possible while retaining completeness.

Initializing CS. We use local filtering, such as LDF [53] and NLF [7], to initialize the candidate vertex set. For a vertex $u \in V_Q$, LDF adds a vertex $v \in V_G$ into $C(u)$ if $L(u) = L(v)$ and $d(u) \leq d(v)$. For each label l, NLF checks whether a candidate vertex v has not fewer neighbors with label l than vertex u. To initialize candidate edges, we add an edge $(v, v') \in E_G$ into $C(u, u')$ if $v \in C(u)$ and $v' \in C(u')$.

Example 2.6. *Consider the query graph Q and data graph G in Figure 1(a) and (b). Take u_0 for example. Because v_0 and v_1 pass the check of LDF and NLF, we have that $C(u_0) = \{v_0, v_1\}$. The candidate set of other vertices in V_Q can be computed analogously. Since v_9 does not satisfy the LDF and NLF matching conditions of any query vertex, it is removed from the initial CS. Figure 1(c) shows the initial CS after creating the candidate edges accordingly.*

Refining CS. To refine CS, recent work mainly focuses on utilizing the neighborhood for filtering. The intuition is that vertex v matches u only if v's neighbors match u's neighbors.

• **GQL [23]**. For each $v \in C(u)$, we build a bipartite graph B_u^v between $N(u)$ and $N(v)$. First, for any two vertices $u' \in N(u)$ and $v' \in N(v)$, a bipartite edge $e_b(u', v')$ is created if $v' \in C(u' \mid u, v)$. Then, we check whether there is a maximum matching in B_u^v, i.e., each vertex in $N(u)$ is connected to a unique neighbor in $N(v)$, with no two bipartite edges sharing a vertex. If not, v is removed from $C(u)$. We refer to the refinement process based on a bipartite graph as a bipartite check.



(a) B_{u4}^{v8} of GQL (b) B_{u0}^{v1} of EdgeBipartite (c) B_{u0}^{v1} of BipartitePlus

Figure 2: Different bipartite graph based methods.

Example 2.7. *Following Example 2.6, to further reduce the candidate space, GQL removes false positive candidates via bipartite checks. For example, to validate $v_8 \in C(u_4)$, GQL builds a bipartite graph B_{u4}^{v8} between $N(u_4)$ and $N(v_8)$ as shown in Figure 2(a). Since the v_9 has already been filtered out by NLF, B_{u4}^{v8} cannot form a size-2 match, thus v_8 can be removed from $C(u_4)$, and edges incident on it can also be discarded. The CS after GQL filtering is shown in Figure 1(d).*

• **EdgeBipartite [47]**. FaSTest [47] further enhances the bipartite check and proposes an advanced filtering method, termed EdgeBipartite. It points out that if a bipartite edge $e_b(u', v')$ is not included in any maximum matching in B_u^v, v' should be removed from $C(u' \mid u, v)$. Meanwhile, to maintain a compact candidate edge set, EdgeBipartite removes an edge (v, v') from the set if it cannot match any query edges, even though both v and v' remain valid in the candidate vertex set.

Example 2.8. *Continuing Example 2.7, EdgeBipartite performs an advanced bipartite check. Considering the candidate matching pair $\langle u_0, v_1 \rangle$, EdgeBipartite builds a bipartite graph $B_{u_0}^{v_1}$, as shown in Figure 2(b). There are two maximum matchings in $B_{u_0}^{v_1}$, i.e., $\{\langle u_1, v_2 \rangle, \langle u_2, v_3 \rangle, \langle u_3, v_5 \rangle\}$ and $\{\langle u_1, v_3 \rangle, \langle u_2, v_2 \rangle, \langle u_3, v_5 \rangle\}$. However, the two bipartite edges $e_b(u_3, v_2)$ and $e_b(u_3, v_3)$ shown in red lines do not appear in any maximum matching, so v_2 and v_3 are removed from $C(u_3 \mid u_0, v_1)$. Similarly, v_2 and v_4 are also removed from $C(u_3 \mid u_0, v_0)$. In this way, EdgeBipartite checks all query-candidate pairs. The CS after being refined by EdgeBipartite is shown in Figure 1(e).*

• **Triangle Safety and Four-cycle Safety [47]**. FaSTest also introduces two substructure filtering techniques, namely triangle safety (*3C*) and four-cycle safety (*4C*). A candidate edge (v, v') is triangle safe with respect to a query edge (u, u') if the following conditions are met: (1) The number of triangles formed by (v, v') in G is not fewer than that of triangles formed by (u, u') in Q; and (2) the corresponding vertices that form triangles with (v, v') and (u, u') should also be matched. A candidate edge can be removed from $C(u, u')$ if it is not triangle-safe regarding (u, u'). Similarly, *4C* considers an edge together with two other vertices to form a

four-cycle. It is easy to verify that the redundant vertices and edges outlined in red in Figure 1(b) can be removed by *3C* check. The CS refined by *3C* is illustrated in Figure 1(f).

2.3 Candidate Tree-based Counting

The candidate tree structures organize the candidate space hierarchically to assist subgraph counting.

Definition 2.9 (Candidate Tree). *Let T_Q be a spanning tree of the query graph Q, a candidate tree T_C for T_Q is a homomorphism of T_Q in the candidate space.*

Given a candidate tree T_C, a bottom-up dynamic programming approach is developed to calculate the exact number of matches of T_Q in T_C [20, 47, 73]. In particular, 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 $W(u, v)$ to denote the number of matches of $T_Q(u)$ contained in the candidate tree $T_C(v)$, where $T_Q(u)$ (Resp. $T_C(v)$) is the subtree rooted at u (Resp. v). Then, $W(u, v)$ is computed as follows.

$$W(u,v) = \prod_{u_c \in N_c(u)} \sum_{v_c \in C(u_c|u,v)} W(u_c, v_c) \quad (1)$$

Clearly, $W(u, v) = 1$ if u is a leaf vertex. Based on Equation 1, the number of matches in a candidate tree can be obtained by iteratively updating the $W(u, v)$ value of vertices in a bottom-up manner until to the root of the tree. The final count for T_Q is the sum of counts returned by all candidate trees for T_Q in CS. The overall computation can be efficiently done in $O(|E_Q||E_G|)$ time [47].

However, the above bottom-up dynamic programming approach still bears two main issues. First, it only supports homomorphism, isomorphism counting remains NP-hard even for tree queries [41]. Second, the counting for the original query graph Q is still very difficult, because there exist non-tree edges, which pose complex matching condition checks. To address this, FaSTest [47] treats a tree count as an upper bound and builds upon it with sampling strategies. By calculating the isomorphic matching success probability of the sampled trees, one can estimate query graph matches.

3 OUR APPROACH

Motivation. The candidate tree-based method offers a promising computation framework for subgraph counting. However, existing solutions still suffer from the following two limitations.

• *Limitation 1: Unsatisfactory Candidate Space Filtering.* FaSTest employs EdgeBipartite to refine the candidate space, which however still has limited filtering power. Although additional filtering techniques (i.e., triangle safety and four-cycle safety) are developed to further refine the CS, the computation of such substructures itself is computationally costly, especially for large data graphs. In fact, it is observed that the time cost of computing triangles dominates the overall cost of subgraph counting for FaSTest. As a result, there is a pressing need to develop more efficient and effective candidate space refinement techniques.

• *Limitation 2: Inaccurate Subgraph Counting.* The candidate tree-based counting method adopts a bottom-up dynamic programming approach. However, this method is only feasible for tree structures (e.g., the spanning tree of the query graph) under homomorphism semantics. This is because there exist non-tree edges for generic graphs, and it is NP-hard for isomorphism, which poses great computational challenges. Although FaSTest proposes a sampling-based algorithm to address these challenges, we observe that it performs reliably only when the data graph is relatively small and the query graph has a simple structure. For the complex query-data pairs with a large sample space, it is difficult to obtain successful samples, which significantly affects accuracy.

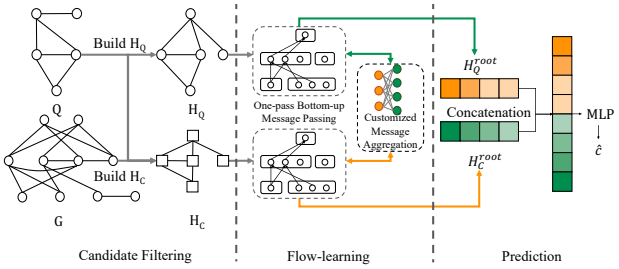


Figure 3: Framework of FlowSC.

Approach Overview. Motivated by the above analysis, we propose a novel approach, termed FlowSC. First, we develop an efficient and effective candidate space filtering algorithm, termed BipartitePlus, which improves the filtering performance of GQL and EdgeBipartite by considering the connectivity between neighbors of the matching vertex with little overhead. Second, to avoid large estimation errors brought by sampling failure, we resort to the learning-based framework. In particular, we develop a novel hierarchical graph-oriented GNN architecture, which can control the information flow in the message-passing process, so as to accurately estimate the matching count in the candidate space.

The overall framework of FlowSC is presented in Figure 3, which consists of three phases. First, we use BipartitePlus to construct a compact candidate space (see Section 4 for the details). Then, we transform the candidate space to a hierarchical graph, where a novel bottom-up GNN architecture is developed to learn the information flow (see Section 5 for the details). Last, the learned features of both the query graph and candidate space are fed to a multi-layer perception (MLP) to obtain the final estimation of the count.

4 ADVANCED CANDIDATE FILTERING

4.1 Motivation

Recall that GQL and EdgeBipartite utilize the neighborhood to enhance the filtering performance. However, we observe that these two methods failed to consider the matching edge between the neighbors, resulting in redundant candidate vertices or edges. More specifically, consider a bipartite graph B_u^v for a matching pair $\langle u, v \rangle$. Let u' and u'' be two neighbors of u, and v' and v'' be two neighbors of v matching u' and u'', respectively. Now, suppose there is an edge between u' and u'', while no edge between v' and v''. GQL and EdgeBipartite will still consider u and v a successful match.

Example 4.1. *Continuing Example 2.8, in $B_{u_0}^{v_1}$ shown in Figure 2(b), although v_2 and v_3 are removed from $C(u_3 \mid u_0, v_1)$, EdgeBipartite still considers v_1 as a candidate of u_0 because there exists a maximum matching in $B_{u_0}^{v_1}$. However, as shown in Figure 1(a), there exists an edge between the neighbors of u_0, i.e., (u_1, u_2), while no edge exists between neighbors of v_1 in Figure 1(b). Thus, v_1 cannot be a candidate*

for u_0. Actually, all invalid vertices and edges depicted by the red solid circles or lines in Figure 1(b) cannot be filtered out by EdgeBipartite.

Although the invalid candidates can be further filtered out by cyclic substructures, e.g., triangles or four-cycles [47], the computation for such substructures itself is time cost expensive. For example, it is $O(|E|^{3/2})$ to find all triangles in a graph, not to mention the computation of four-cycles. In fact, it is noticed that the time cost of computing triangles dominates the overall cost of subgraph counting in large data graphs. Instead of utilizing such expensive pre-computed cyclic substructures, BipartitePlus conducts a light-weight connectivity check for the neighbors of the matching vertex pair to enhance the filtering performance.

4.2 BipartitePlus Algorithm

Technical Details. We focus on a candidate matching pair $\langle u, v\rangle$ and its corresponding bipartite graph B_u^v. We begin by introducing the concept of *triangle edge*.

Definition 4.2 (Triangle Edge). *Given a vertex v in a graph g, and two vertices $v', v'' \in N(v)$, the edge $(v', v'') \in E_g$ is called a triangle edge with respect to v.*

Clearly, a triangle edge forms a triangle with v since its two endpoints are both neighbors of v. We use $\Delta(v)$ to denote all triangle edges with respect to v, and $\Delta(v, v')$ to denote a set of vertices that can form a triangle with edge (v, v'). Below, we formally give the triangle edge-based filtering rule.

Lemma 4.3. *Given a bipartite graph B_u^v, if there exists a triangle edge in $(u', u'') \in \Delta(u)$ having no matching edge in $\Delta(v)$, i.e., $\nexists (v', v'') \in \Delta(v)$ where $v' \in C(u' \mid u, v)$ and $v'' \in C(u'' \mid u, v) \cap C(u'' \mid u', v')$, then v cannot be a candidate vertex for u.*

Proof. The lemma is immediate since the neighborhood matching is a necessary condition for the matching of two vertices. □

Note that Lemma 4.3 only offers a necessary condition to refine the candidate space, rather than a sufficient one, because multiple triangle edges in $\Delta(u)$ might be matching to the same edge in $\Delta(v)$. Nevertheless, by incorporating injectivity checks in GQL, the triangle edge can still be utilized to refine B_u^v.

Algorithm Details. To strengthen the bipartite check, we construct a refined B_u^v by imposing stricter conditions. In particular, for any $u' \in N(u)$ and $v' \in N(v)$ such that $v' \in C(u' \mid u, v)$, a bipartite edge $e_b(u', v')$ is created if

- $\Delta(u, u')$ is empty, or
- $\Delta(u, u')$ is not empty and $\forall u'' \in \Delta(u, u')$, $\exists v'' \in C(u'' \mid u, v) \cap C(u'' \mid u', v')$ such that $(v', v'') \in E_G$.

Otherwise, v' is removed from $C(u' \mid u, v)$. We call the second condition as the *triangle match* between u' and v'. After refining B_u^v, we check whether there exists a maximum matching, which confirms that v is a candidate for u.

Example 4.4. *Following the discussion in Example 4.1, BipartitePlus considers edge $(u_1, u_2) \in \Delta(u_0)$ in its bipartite check of $B_{u_0}^{v_1}$. Specifically, consider $u_1 \in N(u_0)$ and $v_2 \in N(v_1)$. Although $v_2 \in C(u_1 \mid u_0, v_1)$, $\Delta(u_0, u_1) = \{u_2\}$ is not empty. We cannot find a vertex in $C(u_2 \mid u_0, v_1) \cap C(u_2 \mid u_1, v_2)$ having an edge with v_2, indicating that u_1 cannot match v_2 in $B_{u_0}^{v_1}$. Thus, we do not create a bipartite edge between u_1 and v_2. Figure 2(c) presents the $B_{u_0}^{v_1}$ after refining. Since there is no maximum matching in $B_{u_0}^{v_1}$, we remove v_1 from $C(u_0)$. The CS refined by BipartitePlus is illustrated in Figure 1(f).*

Algorithm 1: BipartitePlus

```
Input: Query graph Q and initial candidate space CS
Output: Refined CS
1  foreach u ∈ V_Q do
2      foreach v ∈ C(u) do
3          foreach u' ∈ N(u) do
4              foreach v' ∈ C(u' | u, v) do
5                  if Δ(u, u') = ∅ or
                     ∃v'' ∈ C(u'' | u, v) ∩ C(u'' | u', v') s.t.
                     (v', v'') ∈ E_G for each u'' ∈ Δ(u, u') then
6                      Create a bipartite edge e_b(u', v') in B_u^v;
7                  else
8                      Remove v' from C(u' | u, v);
9          if ∄ a maximum match in B_u^v then
10             Remove v from C(u);
```

Algorithm 1 illustrates the details of BipartitePlus. For each candidate matching pair $\langle u, v\rangle$, we go through all its neighboring pairs (Lines 3-8). For a neighboring pair $\langle u', v'\rangle$, we created a bipartite edge $e_b(u', v')$ in B_u^v if it satisfies the triangle match (Lines 5-6). After considering all neighboring vertices, we remove v from $C(u)$ if there does not exist a maximum matching in B_u^v (Lines 9-10). Besides, we also remove all candidate edges between v and $N(v)$ that were introduced to match the query edges between u and $N(u)$.

Complexity Analysis. For each vertex pair $u' \in N(u)$ and $v' \in C(u' \mid u, v)$ in B_u^v, it is clear the time complexity of a single triangle matching check is $O(d(u)d(v))$ since $|\Delta(u, u')|$ and $|C(u'' \mid u, v) \cap C(u'' \mid u', v')|$ are bounded by $d(u)$ and $d(v)$, respectively. Given that we can determine whether B_u^v has a maximum match in $O(d^2(u)d(v))$ time using Ford-Fulkerson algorithm [18]. The overall time complexity of validating v to u is bounded by $O(d^2(u)d^2(v))$. We remark that although EdgeBipartite and GQL have a lower complexity of $O(d^2(u)d(v))$, BipartitePlus is not necessarily slower in overall efficiency. A stronger filter can eliminate more vertices and edges in an earlier refinement stage, thus significantly reducing the number of checks needed in subsequent steps.

Discussion. Unlike the *3C* filter in FaSTest, our filter has two key differences: (1) We do not build a triangle index for all edges in the data graph. Instead, we perform a triangle matching check only for the limited candidate edges corresponding to a query edge forming a triangle in the query graph; and (2) The check on $(v, v') \in \Delta(v)$ is conducted within a reduced search space, i.e., $C(u'' \mid u, v) \cap C(u'' \mid u', v')$, rather than by pre-storing all common neighbors of v and v' in the original data graph. In other words, we avoid constructing triangles across the entire data graph and instead construct them in a sufficiently refined scope with minimal overhead.

4.3 Effective Filtering Strategy

Postponing BipartitePlus Filtering. Our complexity analysis shows that BipartitePlus has a higher worst-case time complexity

than GQL/EdgeBipartite, although it exhibits stronger pruning power. To further mitigate the computation overhead, we also adopt a postponing filtering strategy, which is motivated by the following empirical observations. At the beginning of refinement, GQL alone can filter out the majority of vertices. For example, on the dataset Human, a single execution of GQL can eliminate about 94% vertices. Moreover, on 7 out of the 8 datasets we tested, more than 90% of the vertices can be removed by a single GQL refinement.

Therefore, we improve the filtering efficiency by postponing the activation of BipartitePlus. At the beginning, GQL is employed, the right side of each bipartite graph can be significantly refined after that. Thus, the number of candidate triangle edges is very limited. We then execute BipartitePlus on the reduced bipartite graphs to further filter out the invalid candidates with little overhead.

Details of Adaptive Refinement. During the filtering process, the removal of a candidate vertex may immediately invalidate some of its neighbors, triggering a cascading effect. Therefore, it is important to consider the traverse order as well as the number of refinement rounds as below.

• *Traverse Order*. In the GQL refinement round, we start from the query vertex with a maximum degree based on the intuition that it is difficult for a large-degree vertex to satisfy the matching conditions. The subsequent filtering follows a BFS order by always selecting the neighbor with the maximum degree. In the following BipartitePlus refinement, we prioritize filtering the query vertex that is most reduced in the previous round. We also use a BFS order to refine the candidate space of the remaining query vertices.

• *Refinement Rounds*. As more rounds of refinement are conducted, more vertices and edges can be filtered out. However, the incremental benefit also diminishes [70], making further refinements less effective. Thus, it is important to stop the refinement process immediately when the gain brought by the enhanced pruning power cannot be well paid off. To this end, we establish the conditions for activating BipartitePlus and stopping the refinement process as follows, where ρ denotes the proportion of vertices survived after one refinement iteration.

(i) Employ GQL at the beginning of refinement;
(ii) Activate BipartitePlus in the following when $\rho > \tau_1$;
(iii) Stop BipartitePlus when $\rho > \tau_2$.
We empirically set $\tau_1 = 0.6$ and $\tau_2 = 0.9$.

5 BOTTOM-UP FLOW-LEARNING

5.1 Motivation

We observe that the summation operation in the candidate tree-based method can be well simulated by the summation-based message aggregation operation of GNNs. However, the direct use of classic GNNs on subgraph counting would face the following issues.

- **Issue 1**: In a classic GNN, each vertex indiscriminately sends and receives messages in its neighborhood. However, the candidate tree-based framework favours conducting message-passing in a limited scope and in a bottom-up manner.
- **Issue 2**: In a classic GNN, a vertex aggregates information from its neighbors in each iteration. In the context of candidate tree counting, this would make the vertex weight be counted and accumulated repeatedly.
- **Issue 3**: In the context of subgraph counting, the matching conditions defined by non-tree edges are critical to counting accuracy. A small change in query graph would cause a significant change in the final result.

Clearly, the above issues pose great challenges in utilizing GNN techniques. To address these challenges, we propose a new GNN architecture that can control the information flow in the message-passing process. To improve the ability of GNNs to handle the complex matching conditions, we propose a customized aggregation strategy by using a cross-graph attention mechanism. This approach leverages the structural information of the query graph to guide the information aggregation of vertices in the candidate space, and thus offers a more accurate simulation of the matching condition checks.

5.2 Overall Architecture of Our Model

The bottom-up dynamic programming of the candidate tree-based method offers a promising computation framework. However, this method cannot well handle the non-tree edges for generic graphs. To get rid of this limitation, we propose a hierarchical graph-based method. Our model consists of the following main components.

Constructing Hierarchical Graphs. To accurately simulate the bottom-up information flow, we transform the query graph into a hierarchical graph, as defined below.

Definition 5.1 (Hierarchical Graph). *Given a query graph Q, a hierarchical graph H_Q of Q is simply a re-arrangement of Q by selecting a vertex u in Q as the root (i.e., level 0) and organizing the rest vertices into levels based on the distance to u.*

For example, in Figure 4(a), H_Q is a hierarchical graph of Q in Figure 1(a) rooted at u_0. Based on the structure of H_Q, we also construct the corresponding hierarchical graph of CS, denoted as H_C. Specifically, let U_l be the set of vertices in level l of H_Q. Then, the vertices in level l of H_C are the union of candidates of all query vertices in U_l. After that, for any two vertices v and v' in H_C, we created an edge between them if (v, v') exists in the CS. In general, the hierarchical graph of CS can be considered a "slim" version of CS by removing the duplicate candidates at the same level. Figure 4(b) shows a hierarchical graph of the CS in Figure 1(f).

Flow Feature Learning and Prediction. Based on the hierarchical graphs, we develop a novel GNN architecture, which adopts a one-pass bottom-up message-passing scheme to conform to the hierarchical graphs (see subsection 5.3 for details). More specifically, the information flows from the bottom of the hierarchical graphs to the top for only one pass. This can ideally simulate the bottom-up dynamic programming computation paradigm. Moreover, to deal with the complex matching conditions, we utilize a cross-graph attention to optimize the message aggregation in the GNN (see subsection 5.4 for details). The information is propagated to the root vertices of H_Q and H_C, which are eventually fed to an MLP to generate the final result.

5.3 One-pass Bottom-up Message-passing

To conform to the hierarchical structures, our message-passing process should obey three principles: (1) each vertex only receives messages from neighboring vertices in the same or lower level;

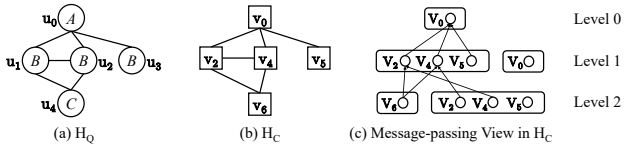


Figure 4: Hierarchical graphs and the message-passing view.

(2) each vertex only sends messages to neighboring vertices in the same or upper level; and (3) once a vertex has updated and passed its feature, it halts on feature updating. Here, we use *level* to distinguish from *layer* in a standard GNN, where the root level is numbered 0. The overall message-passing process can be achieved by irregular adjacency matrices:

$$\mathcal{H}^{(k)} = \text{AGG}(\mathcal{A}_C^{(k)}(\mathcal{H}^{(k+1)} \| \mathcal{H}^{(k)})\mathcal{W}_C^{(k)}) \tag{2}$$

Here, $\mathcal{A}_C^{(k)}$ is an irregular adjacency matrix to propagate the vertex features between level k and $k+1$ in H_C, $\mathcal{H}^k$ is the feature of vertices in level k and $\mathcal{W}_C^{(k)}$ is the weight matrix. Specifically, the rows of $\mathcal{A}_C^{(k)}$ correspond to the vertices at level k, and the columns correspond to the vertices at both level $k+1$ and k. Thus, $\mathcal{A}_C^{(k)}$ a matrix of size $n^{(k)} \times (n^{(k+1)} + n^{(k)})$ where n^k is the number of vertices in level k, and $\mathcal{A}_C^{(k)}(i,j) = 1$ indicates that there is an edge between vertex i at level k and vertex j at level k or $k+1$ in H_C.

According to Equation 2, each pair of adjacent levels has its own irregular adjacency matrix, and the feature of vertices in level k is updated by that of their neighboring vertices in level k and $k+1$. The learning process starts from the bottom of the hierarchical graph and propagates the messages level-by-level simulating a bottom-up information flow. Apparently, this one-pass bottom-up message-passing method can control the direction, range, and iterations of information propagation, addressing *Issue 1* and *Issue 2*.

Example 5.2. *Figure 4 illustrates an example of message-passing process for the query graph and data graph shown in Figure 1. In specific, Figure 4(a) and (b) show the hierarchical graphs H_Q and H_C, respectively. Figure 4(c) presents a computational view of the message-passing, where vertices sending/receiving messages in the same round are placed at the same level. This information flows from level 2 where v_2, v_4, and v_6 send their input features upwards to v_2 and v_4 along the edges between them. Note that, as v_2, v_4, and v_5 are vertices at the same level in H_C, they send and receive messages simultaneously, and hence appear in both level 2 and level 1. Then, in level 1, the feature of v_2 (Resp. v_4) is updated by aggregating its initial feature with the messages from v_4 (Resp. v_2) and v_6. After that, the features of v_2, v_4 and v_5 are aggregated to v_0. Last, the feature of v_0 is used to estimate the final count.*

5.4 Customized Message Aggregation

In the message-passing process, each vertex receives messages from neighboring vertices at the same or lower level. According to Equation 1, messages from different candidate sets should be aggregated differently. More specifically, if the vertices match the same query vertex, summation should be used. If they match different query vertices, multiplication should be used. Besides, the matching conditions defined by the non-tree edges influence the message aggregation as well (i.e., *Issue 3*). To customize the message aggregation, we employ two optimization techniques.

Nonlinear Aggregation. Message-passing, implemented by matrix multiplication, inherently performs the summation aggregation. On the basis of that, we introduce nonlinear computations via a multi-layer perceptron (MLP) to simulate cumulative multiplication in Equation 1.

$$\mathcal{H}^{(k)} = \mathcal{W}_2\sigma(\mathcal{W}_1\mathcal{H}^{(k)} + \mathbf{b}_1) + \mathbf{b}_2 \tag{3}$$

where $\mathcal{W}_1$ and $\mathcal{W}_2$ are the weight matrices for the first and second linear transformations, $\mathbf{b}_1$ and $\mathbf{b}_2$ are their corresponding bias vectors, and σ is the activation function to introduce nonlinearity. By combining the two linear layers, we introduce approximate nonlinear aggregation.

However, there are matching condition checks defined by the edges in the same level of H_Q. Non-tree connections of a vertex in H_Q must align with those of its candidate vertex in H_C. If these connections match, the accumulated weight for each candidate vertex is valid. Motivated by this observation, we incorporate the structural information of the query graph in the aggregation process to customize the aggregation for each candidate vertex under the guidance of the corresponding query vertex.

Query-guided Message Aggregation. In order to incorporate the information of the query graph into the candidate space, we utilize the cross-graph attention [58], which has recently demonstrated great performance on dealing with inter-graph GNN learning. In general, this graph learning architecture enables us to capture the relationships between the query and candidate vertices at the same level of H_Q and H_C. In each level of the hierarchical graphs, the cross-graph attention-reshaped messages are combined with the input feature to generate the new feature of vertices in the current level of H_C. This processing is conducted iteratively from bottom to top of the hierarchical graphs.

To realize cross-graph attention, we follow a query-key-value architecture to establish query-key pairs and update values by combining information from both the queries and keys at each level. The detailed computation steps are as follows.

- The hierarchical graph H_Q performs the bottom-up message-passing to establish a structured input of messages at each level.
- The queries and keys are constructed by using linearly transformed query vertex messages and candidate vertex messages at the same level, respectively.
- The initial values are derived from linearly transformed query vertex messages, and these values are updated through interactions between the queries and keys to form query-to-candidate messages.
- The new feature of each candidate vertex is obtained by integrating the candidate messages with query-to-candidate messages.

At the k-th level, we use $n_Q^{(k)}$ and $n_C^{(k)}$ to denote the number of query and candidate vertices, respectively. The input messages received by these vertices are represented by the vectors $\mathcal{H}_Q^{(k)'}$ and $\mathcal{H}_C^{(k)'}$, respectively. Besides, by $\mathcal{Q}^{(k)}$, $\mathcal{K}^{(k)}$, and $\mathcal{V}^{(k)}$, we denote the query, key, and value matrices, respectively. Specifically, $\mathcal{K}^{(k)}$ is obtained by transforming $\mathcal{H}_C^{(k)'}$ with a weight matrix $\mathcal{W}_\mathcal{K}$, while $\mathcal{Q}^{(k)}$ and $\mathcal{V}^{(k)}$ are obtained by transforming $\mathcal{H}_Q^{(k)'}$ with weight

matrices $\mathcal{W}_\mathcal{Q}$ and $\mathcal{W}_\mathcal{V}$, respectively. The following equation summarizes the details.

$$\mathcal{Q}^{(k)} = \mathcal{H}_Q^{(k)'}\mathcal{W}_\mathcal{Q}, \mathcal{K}^{(k)} = \mathcal{H}_C^{(k)'}\mathcal{W}_\mathcal{K}, \mathcal{V}^{(k)} = \mathcal{H}_Q^{(k)'}\mathcal{W}_\mathcal{V} \quad (4)$$

The correlation between each query-key pair guides the update of the value vectors. We conduct dot product between query and key matrices to measure their alignment, resulting in attention matrix $\mathcal{X}_a^{(k)} \in \mathbb{R}^{n_C^{(k)} \times n_Q^{(k)}}$, where $\mathcal{X}_a^{(k)}(i, j)$ indicates the correlation between candidate vertex i and query vertex j. This attention matrix $\mathcal{X}_a^{(k)}$ is then scaled by the square root of d_k and normalized through a softmax function, where d_k is the number of vector dimensions. Finally, by performing a dot product between the attention scores $\mathcal{X}_a^{(k)}$ and the value vectors $\mathcal{V}^{(k)}$, we obtain new query-to-candidate messages. By further combining this query-to-candidate messages with the candidate messages, we obtain the updated feature $\mathcal{H}_C^{(k)''}$ for the candidate vertices at level k. The following equation summarizes the details.

$$\mathcal{X}_a^{(k)} = \text{softmax}(\frac{\mathcal{K}^{(k)}\mathcal{Q}^{(k)T}}{\sqrt{d_k}}), \mathcal{H}_C^{(k)''} = \mathcal{X}_a^{(k)}\mathcal{V}^{(k)} + \mathcal{H}_C^{(k)'} \quad (5)$$

Cross-graph attention reshapes the input messages for each candidate level based on the corresponding query level. By combining nonlinear and summation aggregation, we customize the message aggregation for updating the feature of each candidate vertex. Algorithm 2 illustrates the details of our method.

5.5 Prediction and Training

Prediction. Through bottom-up message-passing, we collect the root-level features $\mathcal{H}_Q^{root}$ and $\mathcal{H}_C^{root}$. We use a regression-based approach for count prediction. More specifically, we construct a regression neural network by an MLP. The concatenation of $\mathcal{H}_Q^{root}$ and $\mathcal{H}_C^{root}$ is then fed into this neural network to make the final prediction $\hat{c}$. The overall prediction is summarized as:

$$\hat{c} = \text{MLP}(\mathcal{H}_Q^{root} \| \mathcal{H}_C^{root}). \quad (6)$$

Following the previous works, we use *q-error* as the evaluation metric for prediction accuracy, defined as $\max(\frac{\max(1,c)}{\max(1,\hat{c})}, \frac{\max(1,\hat{c})}{\max(1,c)})$, where c is the ground truth counts.

Training Strategy. To enhance parameter initialization, improve the generalization, and stabilize performance across different query graphs, we first use 20% of the query graphs from each dataset to pretrain the model.

In the pretraining phase, in addition to *q-error*, we also incorporate Mean Squared Logarithmic Error (MSLE) into the loss function. MSLE is a widely used regression loss metric for smoothing the errors, as defined below.

$$\text{MSLE} = \frac{1}{N}\sum_{i=1}^{N}\text{SLE}(c_i, \hat{c_i}) = \frac{1}{N}\sum_{i=1}^{N}(\log(\epsilon + c_i) - \log(\epsilon + \hat{c_i}))^2 \quad (7)$$

where N is the batch size, and ϵ is a small constant introduced to prevent logarithmic calculations of 0. The pretraining purpose is to obtain generally well-optimized initial parameters for subsequent fine-tuning on individual datasets rather than to achieve optimal performance on any single query. Therefore, the model is updated based on the average *q-error* of a batch of queries, together with

Algorithm 2: Flow-learning

Input: Hierarchical query graph H_Q, hierarchical CS graph H_C, initial feature matrices $\mathcal{H}_Q$ and $\mathcal{H}_C$
Output: Root level features of H_Q and H_C

```
// Build adjacency matrices from top to bottom
```
for $k = 0, ..., |H_Q| - 2$ **do**
  Construct $\mathcal{A}_Q^{(k)}$ and $\mathcal{A}_C^{(k)}$ with rows representing the vertices at level k in H_Q and H_C, respectively, and columns formed by concatenating the vertex sets from levels $k+1$ and k in H_Q and H_C
  Set $\mathcal{H}_Q^{(k)}$ and $\mathcal{H}_C^{(k)}$ as the subsets of $\mathcal{H}_Q$ and $\mathcal{H}_C$ corresponding to vertices in level k in H_Q and H_C
Set $\mathcal{H}_Q^{(|H_Q|-1)}$ and $\mathcal{H}_C^{(|H_Q|-1)}$

```
// Message passing from bottom to top
```
$\mathcal{H}_Q^{lower\_msg} \leftarrow \mathcal{H}_Q^{(|H_Q|-1)}, \mathcal{H}_C^{lower\_msg} \leftarrow \mathcal{H}_C^{(|H_Q|-1)}$
for $k = |H_Q| - 2, ..., 0$ **do**
  $\mathcal{H}_Q^{(k)'} = \mathcal{A}_Q^{(k)}(\mathcal{H}_Q^{lower\_msg} \| \mathcal{H}_Q^{(k)})\mathcal{W}_Q^{(k)}$
  $\mathcal{H}_C^{(k)'} = \mathcal{A}_C^{(k)}(\mathcal{H}_C^{lower\_msg} \| \mathcal{H}_C^{k})\mathcal{W}_C^{(k)}$
  $\mathcal{H}_Q^{(k)'}, \mathcal{H}_C^{(k)'} = \text{MLP}(\mathcal{H}_Q^{(k)'}), \text{MLP}(\mathcal{H}_C^{(k)'})$
  $\mathcal{Q}^{(k)}, \mathcal{K}^{(k)}, \mathcal{V}^{(k)} = \mathcal{H}_Q^{(k)'}\mathcal{W}_Q, \mathcal{H}_C^{(\mathcal{K})'}\mathcal{W}_k, \mathcal{H}_Q^{(k)'}\mathcal{W}_\mathcal{V}$
  $\mathcal{H}_C^{(k)''} = \text{softmax}(\frac{\mathcal{K}^{(k)}\mathcal{Q}^{(k)T}}{\sqrt{d_k}})\mathcal{V}^{(k)} + \mathcal{H}_C^{(k)'}$
  $\mathcal{H}_Q^{lower\_msg}, \mathcal{H}_C^{lower\_msg} = \mathcal{H}_Q^{(k)'}, \mathcal{H}_C^{(k)''}$
$\mathcal{H}_Q^{root} \leftarrow \mathcal{H}_Q^{lower\_msg}, \mathcal{H}_C^{root} \leftarrow \mathcal{H}_C^{lower\_msg}$
return $\mathcal{H}_Q^{root}, \mathcal{H}_C^{root}$

MSLE that is applied to smooth out loss variations within the batch. We define the pretraining loss function as:

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}\alpha \cdot \text{q-error}(c_i, \hat{c_i}) + \beta \cdot \text{SLE}(c_i, \hat{c_i}) \quad (8)$$

where α and β are weight coefficients. The model is pretrained on a mixed set of query graphs of different sizes from all datasets, using the hybrid loss function with weight decay. The optimal parameters are saved to initialize the fine-tuning phase.

In the fine-tuning phase, we further split the remaining 80% of the query graphs into training and testing sets. Note that query sets for pretraining, training (fine-tuning) and testing do not overlap, and we only use *q-error* as the loss function in training phase.

5.6 Analysis and Discussion

Time Complexity. We construct the hierarchical graphs using a BFS on the query graph and then assign each vertex and edge in the CS to a specific level. The time cost is $O(|V_Q| + |E_Q| + |V_C| + |E_C|)$. The bottom-up flow learning updates features upward along the edges of the H_Q and H_C using sparse matrix multiplication in a single pass. The time cost of learning in H_Q and H_C is $O(|E_Q|+|E_C|)$. Each level of query vertices interacts with the candidate vertices at the same level, resulting in a complexity of $O(\frac{|V_Q|\times|V_C|}{|H_Q|})$, where the impact of hyperparameters is omitted.

Table 1: Statistics of Datasets. $\#Q_x$ is the number of query graphs of size x used in one dataset.

Dataset	$\|V\|$	$\|E\|$	$\|\Sigma\|$	d	$\#Q_4$	$\#Q_8$	$\#Q_{12}$	$\#Q_{16}$	$\#Q_{20}$	$\#Q_{24}$	$\#Q_{32}$
Yeast	3,112	12,519	71	9.0	198	399	-	369	-	319	234
HPRD	9,460	34,998	307	7.4	200	399	-	379	-	397	391
Human	4,674	86,282	44	36.9	176	164	46	21	-	-	-
WordNet	76,853	120,399	5	3.1	200	398	276	183	136	-	-
DBLP	317,080	1,049,866	15	6.6	203	398	-	-	-	-	-
Youtube	1,134,890	2,987,624	25	5.3	199	399	-	164	-	64	27
Eu2005	862,664	16,138,468	40	37.4	200	295	-	-	-	-	-
Patents	3,774,768	16,518,947	20	8.8	200	400	-	376	-	215	161
Twitter	41,652,230	1,202,513,344	100	57.7	214	-	-	-	-	-	-
Friendster	65,608,366	1,806,067,135	100	55.1	400	280	-	-	-	-	-

Parameter Size. In the FlowSC learning framework, each level has weight matrices $\mathcal{W}_Q^{(k)}$ and $\mathcal{W}_C^{(k)}$, along with two weight matrices in each MLP. Additionally, there are 3 linear transformation matrices $\mathcal{W}_\mathcal{Q}$, $\mathcal{W}_\mathcal{K}$ and $\mathcal{W}_\mathcal{V}$, which are shared across all levels. Therefore, the total parameter size is $O(|H_Q|(|V_Q| + |V_C|)d_{hid})$, where d_{hid} is the dimensions of features.

Root Selection. When constructing the hierarchical graphs, we need to choose a vertex in the query graph as the root. Our root selection is based on the intuition to concentrate as many candidate vertices as possible at the root level, thereby simplifying the structural complexity of the lower levels in H_C. Consequently, we select the query vertex with the largest number of candidate vertices as the root in Q. The exploration of more advanced strategies for root selection could be considered in future work.

Extension to Dynamic Graphs. To deal with dynamic graphs, we need to update the candidate space and learning model. First, when new vertices/edges are inserted/deleted, we conduct BipartitePlus only in the neighborhood of these vertices/edges to update the candidate space, following the well-studied continuous subgraph matching problem [39, 62]. Second, our learning model can directly handle dynamic graphs although its accuracy performance may decline. A simple way is to retrain the model periodically or when the distribution of the graph has changed over a threshold.

6 EXPERIMENT

We empirically evaluate the performance of our proposals in this section. Experiments are conducted on an Ubuntu 22.04.1 LTS system, equipped with an Intel Xeon Silver 4314 CPU @ 2.40GHz with 64 cores and NVIDIA RTX A5000 GPUs with 24GB VRAM each.

6.1 Experiment Setup

Datasets. We conduct experiments on 10 real-world datasets, where the query graphs of the first 8 datasets are publicly available [50] and widely used in previous works [47, 55, 72]. The Twitter [32] and Friendster [63] datasets do not have publicly available query sets. We generate random query sets for the two datasets following [50]. To obtain the exact counts, we use the recommended algorithm in [50]. Since obtaining an exact count is extremely time-consuming, we follow the existing work [55] to only select queries for which the exact count can be computed within 30 minutes for the first 8 datasets, and 2 hours for Twitter and Friendster. The statistics of data graphs and query graphs for each dataset are summarized in Table 1. In Human_20, only 7 queries, and in DBLP_16, only 11 queries have exact counts, which are insufficient for training. Thus, we exclude them from evaluation.

Compared Algorithms. In the experiments, we compare FlowSC with the following algorithms: (1) **FaSTest [47]**, the state-of-the-art method, which significantly outperforms earlier leading sampling-based methods (e.g., Alley [29]); (2) **LearnSC [26]**, the lastest learning-based method; (3) **NeurSC [55]**, and (4) **LSS [72]**.

The source code of FaSTest, LSS and NeurSC are publicly available, and LearnSC was obtained from the authors. All parameter settings follow their default values. Besides, we employ the active learning version ALSS [71] for LSS. FaSTest can be optimized with triangle and four-cycle indices by precomputing and keeping them in memory for continuous querying. In real-world applications, subgraph matching/counting typically involves independent data-query pairs. In this paper, we follow the common setting [50, 55, 70] of loading a data-query graph pair into memory each time. Therefore, we primarily use the FaSTest with EdgeBipartite, but conduct an extra experiment to compare all FaSTest variants.

Evaluation Metrics. In the experiments, we evaluate the performance of the compared methods regarding the estimation accuracy and efficiency. Following existing works [26, 47, 55, 72], we measure the accuracy using *q-error*, defined as $\max(\frac{\max(1,c)}{\max(1,\hat{c})}, \frac{\max(1,\hat{c})}{max(1,c)})$, where $\hat{c}$ and c are the estimated and ground truth counts, respectively. The efficiency is measured by the average elapsed time per data-query pair.

Experiment Settings. In the experiment, 20% query graphs are exclusively used for pretraining of our model. The remaining 80% are used for evaluation. In particular, the sampling-based methods are tested on all remaining queries. For learning-based methods, the remaining queries are further divided into 5 disjoint folds. Each fold iteratively serves as the test set once, and the remaining 4 folds are used for training (or fine-tuning), with model parameters reinitialized in each iteration. For FlowSC, the vertex feature is represented with a 128-dimensional vector, initialized by transforming the vertex's label through a linear layer. The final prediction is obtained through a 3-layer MLP. The constant ϵ in MSLE is set to 1, and the weight coefficients in pretraining loss $\mathcal{L}$ are tuned within the range $(0, 1)$ with $\alpha + \beta = 1$. The optimizers used in pretraining and fine-tuning are AdamW and Adam, respectively. For pretraining, we collect the reserved 20% query graphs of all datasets, where the batch size and epochs are set to 10 and 20, respectively. The model is then fine-tuned by the query graphs of each dataset independently, and the batch size is set to 10 by default.

6.2 Accuracy Performance

Figure 5 reports the accuracy performance of the competing algorithms on each query graph set. Following the convention, we use box-plots to present the distribution of *q-error*. In particular, the upper bound (Resp. lower bound) of a box is 75% (Resp. 25%) percentile of the *q-error*, the whiskers contain all *q-error* from minimum to maximum, and the line in the box is the median. Overall, FlowSC consistently outperforms other learning-based methods, reducing the mean q-error across all datasets by more than 2 orders of magnitude. It also surpasses the state-of-the-art FaSTest on large datasets and complex query graphs. The result is omitted if an algorithm fails due to training time or memory constraints.

Comparing with Learning-based Methods. Among the three recent learning-based methods, LSS is relatively stable, whereas

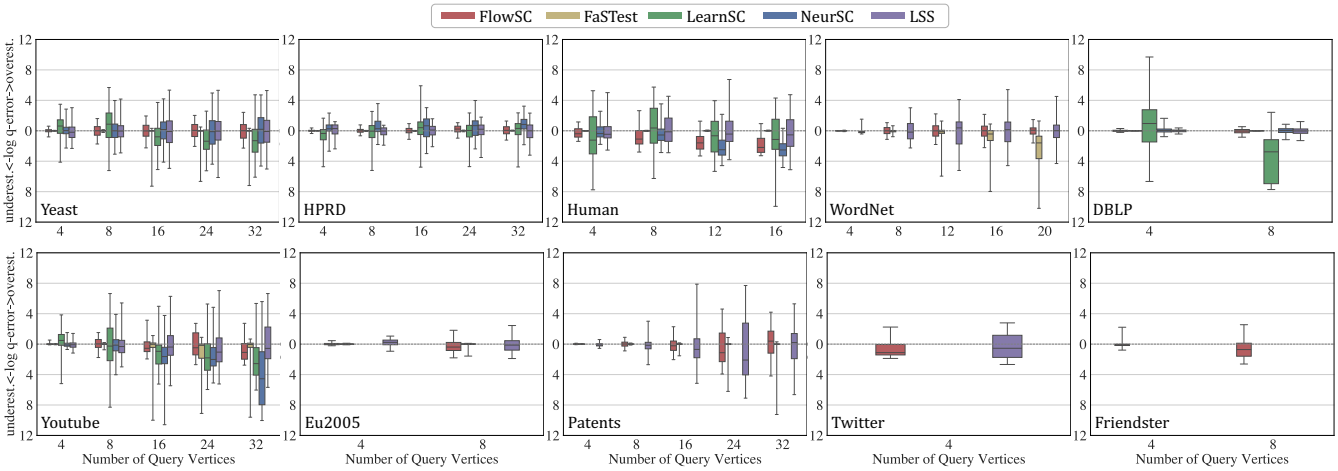


Figure 5: Evaluating accuracy with the x-axis representing the number of query graph vertices.

LearnSC and NeurSC exhibit great fluctuation. NeurSC generally outperforms LSS on the Yeast dataset due to its ability to capture the relationship between the query and data graphs. However, its complex query-candidate interactions lead to instability during training, especially for large query graphs. Similarly, another complex-designed model, LearnSC, suffers from poor model initialization and training difficulties, which lead to low accuracy. In addition, limited query graphs make the learning-based models difficult to learn the relationship between input structures and counts. FlowSC is also affected by this issue. For example, the precision of FlowSC declines on Human when the query graph size is 12 and 16 with only 46 and 21 samples, respectively.

We have an interesting observation on WordNet that the accuracy of FlowSC and LSS improves when the query size increases from 12 to 20. This contradicts the intuition that smaller queries are easier to handle. However, we find that the subgraph count variance actually decreases as query size increases from 12 to 20, which facilitates the regression learning.

FlowSC shows a better performance over the learning-based competitors under all settings. Even on Yeast, where all three counterparts are well-initialized, FlowSC obtains an average *q-error* of 25.86 when the query size is 32, while that of NeurSC, LSS, and LearnSC is 822.72, 2172.25, and 16108.52, respectively. We also observe that FlowSC can handle large data graphs very well. For instance, it returns an average *q-error* of 126.38 on Patents when the query size is 32, while the *q-error* of LSS is 241351.67.

Moreover, FlowSC can handle two billion-scale datasets Twitter and Friendster, demonstrating good scalability of FlowSC.

Comparing with Sampling-based Method. We observe from Figure 5 that FaSTest performs much better than the learning-based method, and even outperforms FlowSC on relatively easy datasets. For instance, on HPRD, FaSTest achieves results with *q-error* close to 1, while the *q-error* of FlowSC averages at 4.49. This is attributed to FaSTest's effective filtering algorithm, which constructs a relatively tight candidate space on HPRD, combined with its advanced sampling algorithms. However, the advantage of FaSTest diminishes for larger data-query pairs, where the candidate space inevitably grows despite strong filtering, resulting in an expansive sample space and higher failure rates. For example, on WordNet, a dataset having the most average candidate vertices, it is observed that the average *q-error* of FaSTest exceeds 10^9 when query size is 20 due to frequent sampling failure. In contrast, FlowSC has a much lower average *q-error* of 13.23. Additionally, FaSTest cannot process Twitter and Friendster due to excessive memory usage.

To further evaluate FaSTest, we test two index-aided versions, namely Fastest+3C and Fastest+4C, which denote FaSTest with triangle, and both triangle and four-cycle indices, respectively. Figure 6 reports the experiment results. On HPRD, where the candidate space is small, all three FaSTest variants perform much better than FlowSC. However, their performance drops sharply on WordNet due to frequent sampling failures caused by a much larger sample space, especially for large query graphs where the matching conditions are complex. This experiment demonstrates that even equipped with more advanced indices, FaSTest cannot consistently deliver reliable performance.

Relationship Between Subgraph Counts and Precision. Figure 7 evaluates the relationship between subgraph counts and precision of all methods on Yeast. Overall, FlowSC, NeurSC and LearnSC tend to overestimate small counts while underestimate large ones, which is a common regression-to-the-mean effect in regression problems [47]. However, LSS avoids this problem via an active learning strategy that samples more in high-uncertainty regions to obtain a more balanced accuracy distribution across different count ranges. The accuracy of FaSTest is mainly influenced by the sample space and query graph size, rather than showing a specific trend across different count ranges.

6.3 Efficiency Performance

Query Processing Time. Figure 8 records the end-to-end elapsed time for conducting a query against a data graph. A cross indicates failure due to training time or memory limits.

Learning-based methods are generally faster than sampling-based methods on large datasets. For example, FaSTest struggles on Youtube, Eu2005, and Patents, due to its inefficient filter and the

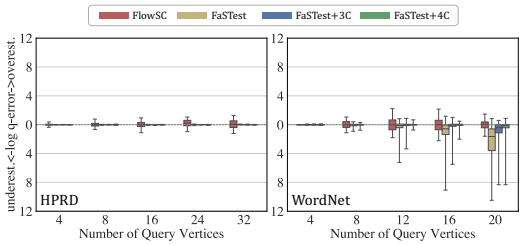


Figure 6: Evaluating all FaSTest variants.

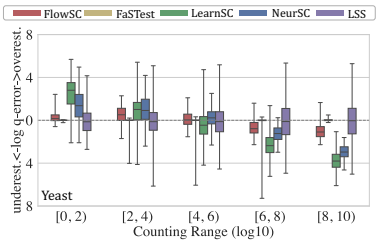


Figure 7: Effect of subgraph counts.

Table 2: Training efficiency.

Dataset	FlowSC	LearnSC	NeurSC	LSS
Yeast	10.83	7.87	24.18	12.18
HPRD	13.98	9.95	26.47	15.80
Human	3.27	4.78	21.04	2.42
WordNet	240.73	-	-	5.76
DBLP	32.35	334.51	1078.51	4.23
Youtube	58.91	65.51	1013.10	5.43
Eu2005	251.89	-	-	6.63
Patents	617.38	-	-	11.35
Twitter	21,407.63	-	-	13.66
Friendster	88,184.39	-	-	-
Epochs	15	50	50	50

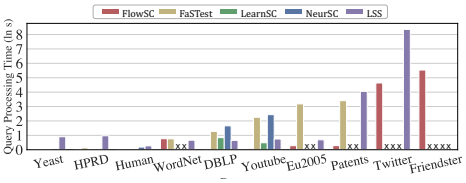


Figure 8: Efficiency performance (ln scaled).

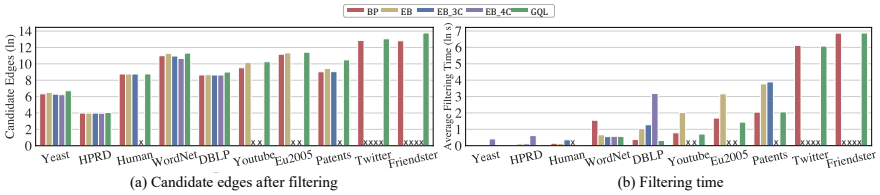


Figure 9: Filtering performance (ln scaled).

large sample space generated by large graphs. Among the learning-based methods, LearnSC and NeurSC showcase relatively poor efficiency. For example, they are 6.4 and 107.7 times slower than FlowSC on Youtube. This is because they adopt a query-candidate interaction network architecture, where edges are created between query vertices and their corresponding candidate vertices. This augmented graph not only incurs significant computational overhead for construction but also high inference costs for GNNs in the graph learning phase. LSS, instead, adopts a lightweight architecture that avoids graph learning on the data graph, resulting in relatively fast overall inference. However, its inference time grows quadratically [55] with the growth of the query graph size. As a result, even on datasets like Yeast, where other methods perform well, LSS takes a long average query time because there are more large query graphs in Yeast. Overall, FlowSC beats the competitors on all datasets but WordNet, where the filtering phase takes lots of processing time. The outstanding efficiency performance of FlowSC attributes to (1) the compact candidate space brought by our powerful filtering algorithm; and (2) the one-pass bottom-up hierarchical flow-learning architecture, where only a small number of vertices are involved in each computation step and the complex interactions occur only among vertices at the same level.

Training Time. Table 2 reports the average training time per epoch and the number of epochs for learning-based methods. In general, LSS is more efficient in training speed as it avoids complex learning on large data graphs. More importantly, unlike other learning-based methods that digest a pair of graphs each time, LSS loads the data graph into memory at once and pre-processes all query graphs in a single pass. This results in a long preprocessing time but a short training time. For example, on the Twitter dataset, LSS takes 4434.12 seconds for the initial preprocessing of a single data graph but only 13.66 seconds per epoch for training. FlowSC demonstrates comparable performance to LSS on small datasets such as Yeast, HPRD and Human, but worse on larger datasets like WordNet. LearnSC and NeurSC rely on computationally intensive networks to capture query-candidate interactions, leading to slower training that becomes practically infeasible on large datasets. FlowSC strikes a good balance between training speed and accuracy.

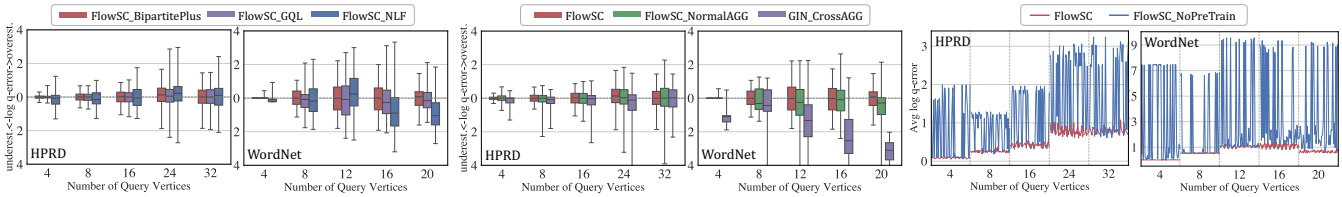
Due to the pretraining strategy, FlowSC achieves good accuracy with relatively few epochs. In our experiments, it is sufficient to fine-tune FlowSC with only 15 epochs on all datasets. The pretraining of FlowSC can be done in an average of 752.23 seconds per epoch. After 20 epochs of pretraining, the model parameters are well initialized across all datasets. In contrast, without pretraining, other methods require more epochs for training. Considering that pretrained parameters can be used as initialization for all datasets, this strategy is highly efficient and effective.

6.4 Evaluating Individual Techniques

Evaluating CS Filtering Techniques. We start by evaluating the candidate space filtering methods. We compare BipartitePlus (BP) to EdgeBipartite [47] (EB), EdgeBipartite with triangle safety (EB_3C), EdgeBipartite with triangle and four-cycle safety (EB_4C), as well as GQL [23]. We investigate the number of candidate edges after refinement and the average processing time of filtering.

It is observed from Figure 9(a) that BipartitePlus is more effective than others on refining the candidate space, especially for large datasets such as Youtube, Eu2005, Patents, Twitter and Friendster. For example, on Patents, the average number of survived edges is 8378, 12587, and 35793 for BipartitePlus, EdgeBipartite, and GQL, respectively. Even compared to the two precomputed methods EB_3C and EB_4C, BipartitePlus can still achieve similar performance on Yeast, HPRD, Human, DBLP and Patents. The experiment results verify the strong filtering capability of BipartitePlus.

As for the filtering efficiency, Figure 9(b) shows that BipartitePlus and GQL can achieve similarly strong performance on most datasets, while EdgeBipartite and its variants fail to respond on large datasets such as Twitter and Friendster. On Friendster, BipartitePlus completes filtering in an average of 963.15s, slightly faster than GQL (968.80s), and removes over 60% more edges. The



(a) Ablation study on filtering module

(b) Ablation study on flow-learning module

(c) Ablation study on training strategy

Figure 10: Ablation studies.

only exception occurs on WordNet, a dataset with densely distributed labels, where lots of vertices share the same label. This increases the number of triangle matching checks and broadens the search space, leading to extra overhead for BipartitePlus. For such datasets, we may adjust the parameters τ_1 and τ_2 to further delay the activation of BipartitePlus refinement.

Sensitivity to Candidate Space. In this experiment, we evaluate the sensitivity of our flow-learning framework to the candidate space. By FlowSC_BipartitePlus, FlowSC_GQL, and FlowSC_NLF, we denote FlowSC using BipartitePlus, GQL, and NLF, respectively. The experiment results are reported in Figure 10(a). It is observed that FlowSC_BipartitePlus outperforms the others under all settings, while FlowSC_NLF performs the worst. The reason is that a compact candidate space can reduce the noise, and therefore our flow-learning model can make more accurate predictions.

Effectiveness of Flow-learning Model. We evaluate the core components of our flow learning model, namely one-pass bottom-up message passing mechanism and customized aggregation. By FlowSC_NormalAGG, we replace the customized aggregation with standard GCN sum aggregation, while by GIN_CrossAGG, we replace the one-pass bottom-up message passing with graph isomorphism network (GIN) and still incorporate query guidance through cross-graph attention. Figure 10(b) illustrates the experiment results. FlowSC outperforms its two variants on all settings. FlowSC_NormalAGG achieves similar accuracy to FlowSC on small datasets. However, the performance gap becomes significant on larger data graphs (e.g., WordNet). This is because the model struggles to learn the matching relationships without query guidance. Although GIN can effectively capture neighborhood features of vertices, it fails to explicitly capture the relationship between structure and count. As a result, even equipped with cross-graph attention, its performance is still unsatisfactory.

Effect of Training Strategy. In the last experiment, we evaluate the effect of pertaining. We compare FlowSC, which is initialized with pretrained parameters, and FlowSC_NoPreTrain, which is initialized with random parameters for each test run. We conduct 50 tests on query graph sets of different sizes from HPRD and WordNet, and reinitialize the model before each run. This setup allows us to assess how much the pretrained initialization helps compared to training from scratch. We record the average *q-error* for each test. As reported in Figure 10 (c), FlowSC consistently achieves high accuracy across all query settings. FlowSC_NoPreTrain can only achieve comparable accuracy when it finds the correct optimization direction. This experiment demonstrates that the pretraining strategy can significantly improve the stability without losing accuracy.

7 RELATED WORK

Subgraph Matching. Recently, the filtering-ordering-enumeration framework has dominated research trend on the subgraph matching problem due to its outstanding performance [50, 70]. Powerful filtering algorithms [6, 7, 20, 21, 23], good matching orders [7, 8, 15, 46], and enumeration acceleration [3, 20, 28, 51, 64] have significantly improved subgraph matching efficiency. Another category of subgraph matching approaches is the join-based methods [4, 33, 34, 43, 65], which transform the query graph into a multi-way join, progressively match smaller substructures, and combine them to form a complete matching. Besides, machine learning based algorithms are also investigated [38, 56, 57, 66].

Subgraph Counting. Existing subgraph counting methods can be categorized into three major classes: sampling-based [9, 13, 29, 35, 47], summarization-based [24, 25, 40, 42, 49], and learning-based [26, 36, 55, 72]. Sampling-based methods often struggle with sampling failures in the exponentially large sample space.Summarization-based methods rely heavily on the independence assumption of query substructure counts, which often leads to poor performance on real-world datasets. While learning-based methods offer faster inference, the accuracy is often unsatisfactory. In addition, there are some learning-based algorithms specifically designed for counting small motifs [14, 27, 37, 52, 61, 68].

Graph Neural Networks. GNNs have been widely adopted in various graph-related tasks due to their ability to effectively capture and utilize graph-structured information. Prominent GNN architectures include GCN [30], GAT [5], GraphSAGE [19], GIN [59] and Graph Transformers [16, 31, 67]. In addition, various GNN variants like [1, 12, 60, 69] have been proposed to improve expressiveness, scalability, or generalization in different application scenarios.

8 CONCLUSION

In this paper, we study the problem of subgraph counting. We present FlowSC, an efficient and accurate approach to subgraph counting, which combines a novel candidate space filtering algorithm with a bottom-up flow-learning model. Extensive experiments demonstrate the outstanding performance of our proposals.

ACKNOWLEDGEMENTS

This work was supported by China Scholarship Council (20230844 0228), the Major Key Project of PCL (PCL2024A05), Guangdong Basic and Applied Basic Research Foundation (2025A1515011716), National Key R&D Program of China (2022YFB3104100), Australian Research Council (DP230101445, FT210100303 and DE250100226) and NSFC (U2241211 and U20B2046).

REFERENCES

[1] Sami Abu-El-Haija, Bryan Perozzi, Amol Kapoor, Nazanin Alipourfard, Kristina Lerman, Hrayr Harutyunyan, Greg Ver Steeg, and Aram Galstyan. 2019. Mixhop: Higher-order graph convolutional architectures via sparsified neighborhood mixing. In *International Conference on machine learning*. PMLR, 21–29.

[2] 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.

[3] Junya Arai, Yasuhiro Fujiwara, and Makoto Onizuka. 2023. Gup: Fast subgraph matching by guard-based pruning. *Proceedings of SIGMOD* 1, 2 (2023), 1–26.

[4] Molham Aref, Balder Ten Cate, Todd J Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L Veldhuizen, and Geoffrey Washburn. 2015. Design and implementation of the LogicBlox system. In *Proceedings of SIGMOD*. 1371–1382.

[5] Graph attention networks. 2018. Veličković, Petar and Cucurull, Guillem and Casanova, Arantxa and Romero, Adriana and Lio, Pietro and Bengio, Yoshua. In *International Conference on Learning Representations*.

[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] 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).

[9] Marco Bressan, Stefano Leucci, and Alessandro Panconesi. 2019. Motivo: Fast Motif Counting via Succinct Color Coding and Adaptive Sampling. In *Proceedings of VLDB*. 1651–1663.

[10] Walter Cai, Magdalena Balazinska, and Dan Suciu. 2019. Pessimistic Cardinality Estimation: Tighter Upper Bounds for Intermediate Join Cardinalities. In *Proceedings of SIGMOD*. 18–35.

[11] Mario Cannataro and Pietro H Guzzi. 2012. *Data management of protein interaction networks*. Vol. 17.

[12] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. Fastgcn: fast learning with graph convolutional networks via importance sampling. *arXiv preprint arXiv:1801.10247* (2018).

[13] Xiaowei Chen and John C. S. Lui. 2016. Mining Graphlet Counts in Online Social Networks. In *Proceedings of ICDM*. 71–80.

[14] Zhengdao Chen, Lei Chen, Soledad Villar, and Joan Bruna. 2020. Can graph neural networks count substructures? *Advances in neural information processing systems* 33 (2020), 10383–10395.

[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] Vijay Prakash Dwivedi and Xavier Bresson. 2021. Generalization and representational limits of graph neural networks. In *International Conference on Machine Learning*.

[17] Wenfei Fan. 2012. Graph Pattern Matching Revised for Social Network Analysis. In *Proceedings of ICDT*. 8–21.

[18] L. R. Ford and D. R. Fulkerson. 1956. Maximal Flow Through a Network. *Canadian Journal of Mathematics* 8 (1956), 399–404.

[19] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

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

[26] 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.

[27] Charilaos Kanatsoulis and Alejandro Ribeiro. 2024. Counting Graph Substructures with Graph Neural Networks. In *International Conference on Learning Representations*.

[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] 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.

[30] Thomas N Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*.

[31] Dexiong Kreuzer, Dominique Beaini, Will Hamilton, Genevieve Letarte, and Prudencio Tossou. 2023. Structure-Aware Transformer for Graph Representation Learning. In *International Conference on Learning Representations*.

[32] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *Proceedings of the International Conference on World Wide Web*. 591–600.

[33] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. 2015. Scalable subgraph enumeration in mapreduce. *Proceedings of VLDB* 8, 10 (2015), 974–985.

[34] Longbin Lai, Lu Qin, Xuemin Lin, Ying Zhang, Lijun Chang, and Shiyu Yang. 2016. Scalable distributed subgraph enumeration. *Proceedings of VLDB* 10, 3 (2016), 217–228.

[35] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2016. Wander Join: Online Aggregation via Random Walks. In *Proceedings of SIGMOD*. 615–629.

[36] Xin Liu, Haojie Pan, Mutian He, Yangqiu Song, Xin Jiang, and Lifeng Shang. 2020. Neural subgraph isomorphism counting. In *Proceedings of SIGKDD*. 1959–1969.

[37] Xin Liu, Weiqi Wang, Jiaxin Bai, and Yangqiu Song. 2024. Towards Subgraph Isomorphism Counting with Graph Kernels. *arXiv preprint arXiv:2405.07497* (2024).

[38] Xuanzhou Liu, Lin Zhang, Jiaqi Sun, Yujiu Yang, and Haiqin Yang. 2023. D2Match: leveraging deep learning and degeneracy for subgraph matching. In *International Conference on Machine Learning*. PMLR, 22454–22472.

[39] Ziyi Ma, Jianye Yang, Xu Zhou, Guoqing Xiao, Jianhua Wang, Liang Yang, Kenli Li, and Xuemin Lin. 2024. Efficient Multi-Query Oriented Continuous Subgraph Matching. *Proceedings of ICDE* (2024), 3230–3243.

[40] Dror Marcus and Yuval Shavitt. 2010. Efficient counting of network motifs. In *International Conference on Distributed Computing Systems Workshops*. IEEE, 92–98.

[41] Dániel Marx and Michal Pilipczuk. 2014. Everything you always wanted to know about the parameterized complexity of Subgraph Isomorphism (but were afraid to ask). In *31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014), STACS 2014, March 5-8, 2014, Lyon, France (LIPIcs)*, Vol. 25. 542–553.

[42] Thomas Neumann and Guido Moerkotte. 2011. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *Proceedings of ICDE*. 984–994.

[43] Hung Q Ngo. 2018. Worst-case optimal join algorithms: Techniques, results, and open problems. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 111–124.

[44] 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.

[45] 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.

[46] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. In *Proceedings of VLDB*, Vol. 1. 364–375.

[47] Wonseok Shin, Siwoo Song, Kunsoo Park, and Wook-Shin Han. 2024. Cardinality Estimation of Subgraph Matching: A Filtering-Sampling Approach. In *Proceedings of VLDB*. 1697–1709.

[48] Tom AB Snijders, Philippa E Pattison, Garry L Robins, and Mark S Handcock. 2006. New specifications for exponential random graph models. *Sociological methodology* 36, 1 (2006), 99–153.

[49] 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.

[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 *Proceedings of VLDB*, Vol. 14. 176–188.

[52] Behrooz Tahmasebi, Derek Lim, and Stefanie Jegelka. 2020. Counting substructures with higher-order graph neural networks: Possibility and impossibility results. *arXiv preprint arXiv:2012.03174* (2020).

[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] Hanchen Wang, Jianke Yu, Xiaoyang Wang, Chen Chen, Wenjie Zhang, and Xuemin Lin. 2023. Neural similarity search on supergraph containment. *Transactions on Knowledge and Data Engineering* 36, 1 (2023), 281–295.

[57] Hanchen Wang, Ying Zhang, Lu Qin, Wei Wang, Wenjie Zhang, and Xuemin Lin. 2022. Reinforcement learning based query vertex ordering model for subgraph matching. In *Proceedings of ICDE*. IEEE, 245–258.

[58] Jianwei Wang, Kai Wang, Xuemin Lin, Wenjie Zhang, and Ying Zhang. 2024. Neural Attributed Community Search at Billion Scale. In *Proceedings of SIGMOD*, Vol. 1. 1–25.

[59] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *International Conference on Learning Representations*.

[60] Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. 2018. Representation learning on graphs with jumping knowledge networks. In *International conference on machine learning*. PMLR, 5453–5462.

[61] Zuoyu Yan, Junru Zhou, Liangcai Gao, Zhi Tang, and Muhan Zhang. 2023. Efficiently counting substructures by subgraph gnns without running gnn on subgraphs. *arXiv preprint arXiv:2303.10576* (2023).

[62] Jianye Yang, Sheng Fang, Zhaoquan Gu, Ziyi Ma, Xuemin Lin, and Zhihong Tian. 2024. TC-Match: Fast Time-constrained Continuous Subgraph Matching. *Proceedings of VLDB* 17 (2024), 2791–2804.

[63] Jaewon Yang and Jure Leskovec. 2012. Defining and evaluating network communities based on ground-truth. In *Proceedings of the SIGKDD workshop on mining data semantics*. 1–8.

[64] 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.

[65] Zhengyi Yang, Longbin Lai, Xuemin Lin, Kongzhang Hao, and Wenjie Zhang. 2021. Huge: An efficient and scalable subgraph enumeration system. In *Proceedings of SIGMOD*. 2049–2062.

[66] Yutong Ye, Xiang Lian, and Mingsong Chen. 2024. Efficient Exact Subgraph Matching via GNN-Based Path Dominance Embedding. *Proceedings of VLDB* 17, 7 (2024), 1628–1641.

[67] Chengxuan Ying, Tianle Cai, Shengjie Luo, Shuxin Zheng, Guolin Ke, Di He, Yanming Shen, and Tie-Yan Liu. 2021. Do Transformers Really Perform Bad for Graph Representation?. In *Advances in Neural Information Processing Systems*.

[68] Xingtong Yu, Zemin Liu, Yuan Fang, and Xinming Zhang. 2023. Learning to count isomorphisms with graph neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 4845–4853.

[69] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2019. Graphsaint: Graph sampling based inductive learning method. *arXiv preprint arXiv:1907.04931* (2019).

[70] 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. 1–29.

[71] Kangfei Zhao, Jeffrey Xu Yu, Qiyan Li, Hao Zhang, and Yu Rong. 2023. Learned sketch for subgraph counting: a holistic approach. *The VLDB Journal* 32, 5 (2023), 937–962.

[72] Kangfei Zhao, Jeffrey Xu Yu, Hao Zhang, Qiyan Li, and Yu Rong. 2021. A Learned Sketch for Subgraph Counting. In *Proceedings of SIGMOD*. 2142–2155.

[73] Zhuoyue Zhao, Robert Christensen, Feifei Li, Xiao Hu, and Ke Yi. 2018. Random Sampling over Joins Revisited. In *Proceedings of SIGMOD*. 1525–1539.