Accelerating Subgraph Matching through Fine-grained and Powerful Equivalences

Yujie Lu
School of Data Science
Fudan University
yjlu23@m.fudan.edu.cn

Zhijie Zhang
School of Data Science
Fudan University
zhangzj22@m.fudan.edu.cn

Weiguo Zheng*
School of Data Science
Fudan University
zhengweiguo@fudan.edu.cn

Lei Zou
Wangxuan Institute of Computer Technology
Peking University
zoulei@pku.edu.cn

ABSTRACT

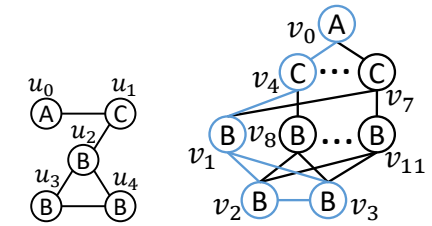
Subgraph matching, a cornerstone of graph analytics, critically suffers from redundant computations during the search process. Existing methods primarily target identical computations redundant operations that are localized to individual query vertices but fail to address similar redundancies that recur across multiple query vertices. In this paper, we present a novel algorithm, called FiPE, that accelerates subgraph matching through Fine-grained and Powerful Equivalences. FiPE redefines redundancy elimination by shifting the optimization granularity from isolated vertices to vertex pairs and multiple vertex patterns. It introduces *vertex-pair equivalence* to cluster candidate pairs with isomorphic neighbor structures, even if their individual vertices differ, enabling pruning of similar computations between these vertex pairs. FiPE proposes *group equivalence* to defer equivalence checks to later search depths, capturing potential redundancies incrementally. To fully exploit the advantages of the equivalence, we introduce two optimization techniques: a matching order generation method to reduce the overall search space and an efficient conflict resolution mechanism to avoid two query vertices being mapped to the same data vertex. Experiments on real-world graphs highlight the superiority of FiPE. FiPE achieves a speedup of 2 to 3 orders of magnitude on various graphs under the EPS (embeddings per second) metric.

PVLDB Reference Format:
Yujie Lu, Zhijie Zhang, Weiguo Zheng, and Lei Zou. Accelerating Subgraph Matching through Fine-grained and Powerful Equivalences. PVLDB, 18(11): 3896 - 3909, 2025.
doi:10.14778/3749646.3749662

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

*Corresponding author

Proceedings of the VLDB Endowment, Vol. 18, No. 11 ISSN 2150-8097.
doi:10.14778/3749646.3749662



(a) Query graph Q (b) Data graph G

Figure 1: A running example of subgraph matching.

1 INTRODUCTION

Subgraph matching has broad applications across domains such as social network analysis [9, 26, 27, 47], biological data mining [8, 36], and fraud detection [10, 33]. It enables the discovery of complex patterns within large-scale graph data by identifying isomorphic mappings subgraphs to a specified query graph Q. Let us consider the query graph Q and the data graph G in Figure 1. The subgraph induced by the blue vertices in G is isomorphic to Q.

1.1 Redundant Computation in the Search Tree

Due to its NP-hard nature [14], subgraph matching poses significant computational challenges, especially when the query graph Q and data graph G are large. Therefore, numerous algorithms [2–6, 11, 14, 16, 18, 19, 21, 30, 31, 35, 37, 40] have been proposed for subgraph matching. These algorithms typically follow a filtering-ordering-enumerating paradigm: (1) The filtering step identifies initial candidate vertices in the data graph for each query vertex; (2) The matching order, which specifies the sequence in which query vertices or edges are compared to the data graph, is established; (3) The embeddings of the query graph are enumerated by exploring all possible vertex assignments in a depth-first manner.

The enumeration phase can be represented as a depth-first search (DFS) tree. Figure 2 illustrates the DFS tree corresponding to the query graph Q and data graph G depicted in Figure 1, using the matching order $u_0 \prec u_1 \prec u_2 \prec u_3 \prec u_4$. A path from root to leaf in the DFS tree materializes the vertex pairs $\{u_0, v_0\}$, $\{u_1, v_4\}$, $\{u_2, v_1\}$, $\{u_3, v_2\}$, and $\{u_4, v_3\}$, representing an isomorphic embedding of Q in G. After finding this embedding, the algorithm backtracks

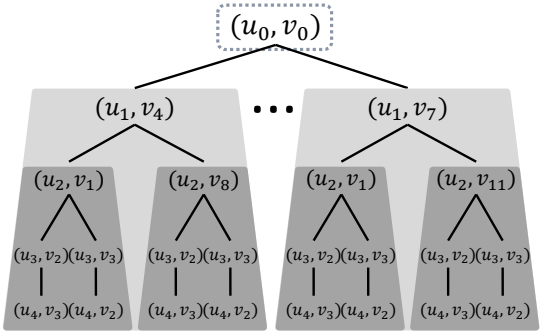


Figure 2: DFS backtracking tree.

to vertex u_2 and then attempts to match u_3 with v_3. This process continues recursively until the entire search tree is traversed.

Identical Computation between Subtrees. Identical computations may occur frequently during the matching process. This happens when a query vertex u matches different data vertices, but the candidates for u's neighboring vertices remain *unchanged*. This may arise due to the absence of edge connections or because the candidates share the same connections to the data vertices. As a result, the subsequent search process becomes identical, leading to redundant computations. Given two subtrees whose roots are siblings in the DFS tree, we say these two subtrees are "identical subtrees" if they share the same structure except the roots.

Example 1.1. The search tree depicted in Figure 2 illustrates an example in which duplicate computations occur for the query vertices u_3 and u_4. As shown by the dark shaded areas in Figure 2, the subtrees rooted at (u_2, v_1) and (u_2, v_8) within the subtree rooted at (u_1, v_4) exhibit identical search structures. Similarly, the subtrees rooted at (u_2, v_1) and (u_2, v_{11}) within the subtree rooted at (u_1, v_7) also have identical structures. Such identical computations are considered common in the backtracking process. In particular, the edge (v_2, v_3) is repeatedly visited unnecessarily. ■

We quantify redundant computations during the search tree using the concept of the identical ratio at depth k. The identical ratio, shorted by IR, is defined as:

$$\text{IR@}k = 1 - \frac{\#\text{different subtrees@}k}{\#\text{subtrees@}k}, \tag{1}$$

where "#different subtrees@k" and "#subtrees@k" represent the number of different subtrees and the number of total subtrees at depth k, respectively. For example, in Figure 2 there are 16 subtrees at depth 4, of which only two are distinctrooted at (u_4, v_3) and (u_4, v_2). Therefore, IR@4 $= 1 - \frac{2}{16} = 87.5\%$, indicating a high level of identical computation at this level of the search.

Table 1 (first row) presents the average identical ratios for 100 query graphs, each with of 8 vertices, on the YeastS graph [12]. The results reveal that identical search subtrees constitute a significant portion of the search space. Thus, minimizing redundant computations can lead to substantial query speedups [44].

Similar Computation in the Search Tree. Beyond identical computations within subtrees, the subgraph matching process also involves numerous *similar* computations that exhibit substantial overlap in their corresponding search subtrees.

Table 1: The average overlap ratios on Yeast at different depths in the backtracking process (%).

depth(k)	1	2	3	4	5	6	7
IR@k	98.47	31.92	34.53	38.79	31.66	34.52	41.17
SR@k	98.47	97.63	94.19	90.18	82.35	83.36	84.76

Example 1.2. As shown by the shaded gray areas in Figure 2, the subtrees rooted at (u_1, v_4) and (u_1, v_7) are not identical due to differing internal vertices (u_2, v_8) and (u_2, v_{11}). However, they share a significant portion of their structure, resulting in backtracking processes that are highly similar. Identifying and leveraging the redundant computations within these similar subtrees could greatly improve the efficiency of subgraph matching. ■

To measure the similarity between subtrees, we use the Jaccard similarity of their search spaces. Subtrees with a similarity score exceeding 0.9 are grouped into the same *bucket*. For the depth k of the search tree, we count the number of buckets and compute the similarity ratio (shorted by SR) as:

$$\text{SR@}k = 1 - \frac{\#\text{buckets@}k}{\#\text{subtrees@}k}, \tag{2}$$

where "#buckets@k" and "#subtrees@k" represent the number of buckets and the number of subtrees at depth k, respectively.

The second row of Table 1 reports the average similarity ratio for the same set of query graphs on the YeastS graph. The overlap ratios consistently exceed 80%, compared to the 30-40% observed for identical ratios. These findings reveal substantial redundancy within the search space, highlighting significant opportunities for optimization through the reduction of duplicate computations.

Limitations of Existing Methods. Most existing subgraph matching algorithms primarily focus on addressing redundant computations of identical search subtrees. These methods can be broadly categorized into three approaches: (1) *Pruning based on repeated failures* [2, 6, 13, 21]: This approach identifies and records the query vertices that cause failures and skips those that are not responsible for failures. (2) *Pruning based on repeated successes* [21]: This method detects equivalent candidates for each query vertex. Once one candidate is matched, the results for other equivalent candidates can be inferred directly. (3) *Pruning based on postponed extension* [4, 18, 40]: This technique analyzes the query graph to identify vertices leading to identical subtrees and moves these vertices to the end of the matching order.

While these approaches can reduce some redundant computations, they are limited to pruning identical computations for individual query vertices or their candidates. As shown in Figure 2, beyond identical subtrees, we observe that the subtrees rooted at (u_1, v_4), …, (u_1, v_7) share substantial similarity as they have identical partial structures. However, existing methods are not able to capture such redundant computations between these similar subtrees. To overcome this limitation and further reduce computational overhead, it is crucial to identify and leverage redundancies among similar subtrees throughout the search tree.

1.2 Our Approach and Contributions

As mentioned earlier, overlaps between subtrees frequently occur during the backtracking process. Effectively detecting and reducing this redundancy can greatly enhance the performance of subgraph matching. However, identifying these overlaps is highly challenging, as the structure of the DFS search tree is not known in advance, and overlaps may arise at any depth within the tree.

In this paper, we propose a novel backtracking search method FiPE, which leverages Fine-grained and Powerful Equivalences. FiPE could address the challenge of efficiently handling the numerous similar regions of the search space, rather than focusing solely on the relatively infrequent identical parts. FiPE utilizes *vertex-pair equivalence* and *group equivalence* to effectively capture overlaps between similar subtrees, enabling a finer-grained detection of redundancies. Compared with existing methods, FiPE can identify and exploit not only identical computations but also structural similarities across subtrees, significantly reducing redundant computations and improving the overall efficiency. Furthermore, we introduce an optimized ordering and conflict resolution mechanism to enhance the efficiency and scalability of the FiPE search process.

To reduce the duplicate computations in both identical and similar subtrees, we adopt a finer-grained equivalence, *vertex-pair equivalence*, by using a pair of vertices as the basic unit to identify overlaps. Unlike vertex-based equivalence, vertex-pair equivalence extends the equivalence scope to pairs of vertices. Vertex-pair equivalence focuses on the relationships between vertices and their neighbors, allowing it to capture redundancies caused by overlapping neighbor sets that vertex-based equivalence would miss. Thus, the vertex-pair equivalence is theoretically proved to dominate the existing vertex-based equivalence (Theorem 4.1).

Vertex equivalence focuses only on the currently matched vertex, limiting its ability to detect redundancies to sibling vertices. To address this limitation, we propose *group equivalence*, which considers all previously matched query vertices as a whole. This broader perspective allows for the detection of overlaps between more distant parts of the search tree. With *group equivalence*, we can identify overlaps between different paths in the DFS search tree. As long as these distinct paths result in similar search subtrees, we can further explore shared portions of the search subtrees and eliminate redundant computations. By capturing these overlaps, *group equivalence* addresses redundancies that vertex-based equivalence fails to detect, significantly enhancing the efficiency.

To fully exploit the advantages of the methods proposed above, we introduce two optimization techniques. The first is a matching order generation method designed to minimize the number of candidates and effectively reduce the overall search space. The second is an efficient conflict resolution mechanism, which addresses a critical challenge in subgraph isomorphism matching by ensuring that no two query vertices are mapped to the same data vertex.

In summary, we make the following contributions:

(1) We propose a finer-grained *vertex-pair equivalence* by using a pair of vertices as the basic unit, effectively reducing overlaps caused by repeated traversals in DFS search trees.
(2) We introduce the concept of *group equivalence*, which generalizes vertex equivalence to a set of vertices, facilitating the identification of overlaps between matching paths. *Group equivalence* enhances the ability to detect overlapping subtrees and reduces redundant computations more effectively.
(3) We develop a matching order generation method to reduce the search space and an efficient conflict resolution mechanism to ensure valid vertex mappings, enhancing the overall efficiency and robustness of the approach.
(4) We conduct extensive experiments to demonstrate the effectiveness and efficiency of the proposed methods.

2 PRELIMINARY

2.1 Problem Definition

In this paper, we focus on undirected and vertex-labeled graphs. For ease of representation, let $G = (V_G, E_G, \Sigma, L_G)$ represent the data graph, where V_G is a set of vertices and E_G is a set of edges. The set Σ represents all possible vertex labels, with label mappings defined by $L_G : V_G \rightarrow \Sigma$. Similarly, a query graph is denoted as $Q = (V_Q, E_Q, \Sigma, L_Q)$. We denote a vertex in the query graph Q as u and a vertex in the data graph G as v. An edge in the query graph is represented as $e(u_1, u_2) \in E_Q$, and an edge in the data graph is denoted as $e(v_1, v_2) \in E_G$. Given $u \in V_Q$, $N(u)$ denotes the neighbors of u, i.e., $N(u) = \{u' | e(u, u') \in E_Q\}$.

Definition 2.1 (Subgraph Isomorphism). Given a query graph $Q = (V_Q, E_Q, \Sigma, L_Q)$ and a data graph $G = (V_G, E_G, \Sigma, L_G)$, Q is subgraph isomorphic to G if there exists an injective function $f: V_Q \rightarrow V_G$ that satisfies the following two conditions:

(1) $\forall\, u \in V_Q$, we have $L_Q(u) = L_G(f(u))$ where $f(u) \in V_G$;
(2) $\forall\, e(u_1, u_2) \in E_Q$, we have $e(f(u_1), f(u_2)) \in E_G$.

The injective function means that the vertices in V_Q must match different vertices in V_G, and f is also called an *isomorphic embedding* of Q in G. Each embedding can be represented by a set of vertex pairs $\{(u, f(u)) | u \in V_Q, f(u) \in V_G\}$. A partial embedding $M : I \rightarrow V_G$ is an embedding of a subgraph of Q that is induced by I, where I is a subset of V_Q. An extension of partial embedding M is to add a vertex pair (u, v) into M, denoted by $M \cup (u, v)$.

Definition 2.2 (Subgraph Matching). Given a query graph Q and a data graph G, subgraph matching finds all embeddings of Q in G.

The filtering-ordering-enumerating framework outlined in Algorithm 1 is widely adopted in existing subgraph matching algorithms [34, 44]. First, a filtering process is conducted to identify and retain the *initial candidate set* $C(u)$ for each query vertex u (line 1). Subsequently, a matching order φ is determined to guide the enumeration process (line 2). Given the candidate set C and the matching order φ, the embeddings of the query graph are then enumerated through a general backtracking process (lines 4-14). At each backtracking step, the algorithm selects the next query vertex u to be mapped (line 7). The valid candidates for u are then computed based on the previously mapped vertices (line 8). For each data vertex v in the *valid candidate set* $C_M(u)$ and not used in M (lines 10-11), the algorithm extends the partial mapping and recursively invokes the next step (lines 12-14).

2.2 Related Work

Subgraph matching encompasses a wide range of problem settings and algorithms. Regarding label configurations, some approaches

Algorithm 1: General Backtracking Method

Input: query graph Q, data graph G
Output: all the embeddings of Q in G

```
1  C ← build candidate sets;
2  φ ← generate a matching order;
3  Enumerate(C, φ, ∅, 0);
4  Procedure Enumerate(C, φ, M, i)
5      if i = |φ| then
6          output M, return;
7      u ← Select-Next-Vertex(C, φ, M);
8      C_M(u) ← Compute-Candidate-Set(C, M, u);
9      foreach v ∈ C_M(u) do
10         if v is used in M then
11             continue;
12         M ← M ∪ (u, v);
13         Enumerate(C, φ, M, i + 1);
14         Remove (u, v) from M;
```

focus on edge-labeled graphs [17, 23, 42], while others target unlabeled graphs [22, 24, 25]. The majority, however, are designed for vertex-labeled graphs [2–5, 13, 16, 19, 21], with some methods claiming to support various label configurations. Several subgraph enumeration techniques are tailored for multicore architectures or distributed systems [1, 7, 20, 22, 24, 25, 29, 32, 38, 39, 41, 43]. In contrast, optimization techniques for single-threaded subgraph matching algorithms [2–5, 13, 16, 19, 21] often rely on processes.

To reduce the search space, various techniques leverage local features to generate initial candidate sets for each query vertex u [21, 35, 37, 40]. NLF [46] refines the candidate selection by choosing data vertices whose neighbor label frequency is no less than that of u. CFL [4] constructs a Breadth-First Search (BFS) tree for the query graph and filters candidates by examining edges in the tree. DPiso [13] uses a similar propagation technique by constructing a Directed Acyclic Graph (DAG) for the query graph.

Most exploration-based subgraph matching algorithms can be broadly categorized into three approaches: (1) Pruning based on repeated failures [2, 6, 13, 21], (2) Pruning based on repeated successes [21], (3) Pruning based on postponed extension [4, 18, 40]:

Repeated failures. This approach identifies data vertices responsible for query failures, bypassing vertices that cause these failures. DPiso [13] introduces a pruning method based on failing set technique which records the failure reasons along the matching process. BICE [6] integrates failing sets and introduces bipartite matching to further prune unnecessary backtracking steps. GuP [2] proposes guard-based pruning, which records failures encountered during the backtracking process as "guards". It prunes redundant backtracking efforts when the same failures are encountered again.

Repeated successes. The method identifies equivalent candidates for each query vertex. Once a candidate is matched, the results of other equivalent candidates can be directly inferred. VEQ [21] introduces equivalence sets, capturing both success and failure cases, where matching one vertex in a set enables direct inference for the remaining vertices in the set.

Postponed extension. This technique examines the query graph to locate vertices that lead to identical subtrees, postponing these vertices to the end of the matching order for optimization. First, vertices with degree 1 can be placed at the end for matching [4, 13, 21]. Second, when all neighbors of a query vertex have been added to the partial matching, it can be placed at the end [18, 40].

Although these approaches effectively reduce redundant computations, their optimization is limited to eliminating redundancies that are strictly *identical*. Repeated failures occur when subtrees consistently return empty result sets, while repeated successes depend on isomorphic subtrees to infer the same successful results. Postponed extension addresses redundancies that naturally arise from identical subtree structures. However, these methods fail to address redundancies among *similar* but non-identical search subtrees, presenting opportunities for further optimization.

3 OVERVIEW OF OUR APPROACH

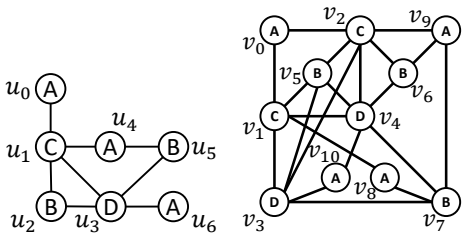
Different from the existing algorithms that treat individual vertices as the unit of matching, we propose a more flexible and powerful subgraph matching approach, called FiPE. FiPE can aggregate similar matching subtrees and leverage their overlapping regions to minimize redundant computations. To achieve this, we first establish vertex-pair equivalence between consecutive vertex pairs in the matching order, enabling the detection of redundant computations across adjacent layers. Building on vertex-pair equivalence, we construct group equivalence to identify redundant computations throughout the entire matching path. Finally, we design a matching order generator and an efficient enumeration technique to seamlessly integrate with the equivalence computation process.

Algorithm 2 outlines the processing flow of FiPE. In a backtracking step, we first get the current vertex pair p to be matched (line 3). Then we compute the sub-trees for current p based on the candidate sets of the vertex pair (line 4). If none valid search tree can be built, just return empty results (lines 5-6). We recognize the overlapped search space and put them into different equivalent groups (line 7). For each group, we add it to partial matching M and then

Algorithm 2: FiPE-Backtrack(Q, G, CS, O, M)

Input: query graph Q, data graph G, candidate sets CS, matching order O, and partial matching M
Output: the embeddings of Q in G

```
1  if |M| = (|O| − 1) then
2      Enumerate all embeddings;
3  p =← next pair from O;
4  T(p) ← Compute-Subtrees(CS, u);
5  if T(p) = ∅ then
6      return ∅;
7  E(p) ← Compute-Equivalence(p, T(p));
8  foreach e in E(p) do
9      M′ ← M ∪ {e};
10     Set-Space(M');
11     FiPE-Backtrack(Q, G, CS, O, M′);
12     Clear-Space(M');
```



(a) Query graph Q_2 (b) Data graph G_2

Figure 3: An example for FiPE illustration.

the next backtracking step is evoked after updating the candidates of unmatched query pairs (lines 9-11). From line 8 of Algorithm 2, we observe that the primary distinction between FiPE and other subgraph matching algorithms lies in its basic matching unit.

Vertex-pair Equivalence. We propose a *vertex-pair equivalence* technique to address redundant computations between adjacent layers of the search tree. This method captures equivalence by focusing on vertex pairs along the matching order rather than individual vertices. Specifically, we define the neighbors of a query vertex pair as the union of the neighbors of both vertices within the pair. The candidate pairs with the same neighbors are then grouped into vertex-pair equivalence sets. By leveraging these equivalence sets, we can eliminate redundant computations within each set. Vertex-pair equivalence relaxes the constraint of vertex equivalence and identifies similar computations within the search tree.

Group Equivalence. Vertex-pair equivalence identifies and eliminates redundant computations in local search spaces formed between consecutive vertices in the matching order. To extend this ability to the entire search space, we propose the *group equivalence* technique, which considers all vertices in the matching path. We gradually build group equivalence by relaxing the conditions for constructing vertex-pair equivalence. We allow candidate vertex pairs to be grouped into the same equivalent set even if their neighbors are not the same.

Optimizations. We propose an ordering technique to optimize the search tree by reducing both its width and height. To minimize width, vertices with the smallest candidate sets are prioritized, reducing the overall branching factor. To minimize height, *independent vertices*whose neighbors are already matchedbypass edge constraint verification and are embedded separately, eliminating the need for backtracking. We introduce an efficient enumeration technique to list embeddings while avoiding vertex reuse. Our approach postpones the vertex reuse check until the enumeration process for independent vertices. Since this process eliminates the need for edge constraint validations and candidate set updates for any vertex, our enumeration technique achieves remarkable efficiency.

4 VERTEX-PAIR EQUIVALENCE

The existing vertex equivalence requires all neighbors of two vertices to be identical, which is often too restrictive. To effectively identify and exploit overlaps in the search subtrees during the backtracking process, we adopt a pair of vertices (instead of one vertex) as the basic unit for constructing equivalence, referred to as *vertex-pair equivalence*. This finer granularity enables a more precise characterization of overlaps, reducing redundant computations caused by repeated visits of vertex pairs.

Definition 4.1 (Vertex-pair Equivalence). Let $\{u_h, u_t\}$ be a pair of query vertices in Q following the matching order φ, and let M denote the current partial matching that includes all vertices preceding u_h in φ. Let $v_h^0, v_h^1 \in V_G$ be candidate vertices for u_h, and $v_t^0, v_t^1 \in V_G$ be candidate vertices for u_t. We say that the candidate combinations (v_h^0, v_t^0) and (v_h^1, v_t^1) are *vertex-pair equivalent* under M if extending M with either mapping $\{(u_h, v_h^0), (u_t, v_t^0)\}$ or $\{(u_h, v_h^1), (u_t, v_t^1)\}$ results in the same candidate sets for all remaining unmatched query vertices.

Vertex-pair equivalence implies that the two candidate combinations yield identical search subtrees in the remaining search space. It allows us to extract overlaps between neighbors of different vertices, avoiding redundant computations caused by these overlaps.

Lemma 4.2. *If (v_h^0, v_t^0) and (v_h^1, v_t^1) are vertex-pair equivalent, then the subtrees in the DFS search tree rooted at these combinations are isomorphic. Consequently, any subtree rooted at (v_h^1, v_t^1) is redundant and can be safely pruned without affecting the correctness of the search process.*

Proof. By Definition 4.1, (v_h^0, v_t^0) and (v_h^1, v_t^1) are vertex-pair equivalent if, after adding $\{(u_h, v_h^0), (u_t, v_t^0)\}$ or $\{(u_h, v_h^1), (u_t, v_t^1)\}$ to the partial matching M, the candidate sets for all unmatched vertices remain identical. Since the candidate sets determine the remaining search space, the subtrees rooted at these two vertex pairs in the DFS search tree are isomorphic. Thus, exploring the subtree rooted at (v_h^1, v_t^1) is redundant and can be safely pruned without affecting the correctness of the search process. □

To determine vertex-pair equivalence, we analyze how adding a candidate pair $(u_h, v_h), (u_t, v_t)$ to the current partial matching M affects the candidate sets of other unmatched vertices. This process applies to both edge-constrained vertex pairs (where $e(u_h, u_t) \in E_Q$) and unconstrained ($e(u_h, u_t) \notin E_Q$) vertex pairs.

We restrict the update to the candidate sets of the neighbors of u_h, u_t. This localized update is sufficient because if the candidate sets of all neighboring vertices remain unchanged, the candidate sets of all other unmatched vertices will remain unchanged as well.

Neighbor Classification. We classify the neighbors of the vertex pair $\{u_h, u_t\}$ into three categories based on their connection patterns in the query graph Q:

- h_{nbr}: Neighbors connected only to the head vertex u_h. For edge-constrained pairs ($e(u_h, u_t) \in E_Q$), h_{nbr} omits u_t; otherwise, it includes all of u_hs neighbors.
- t_{nbr}: Neighbors connected only to the tail vertex u_t. For edge-constrained pairs, t_{nbr} excludes u_h; otherwise, t_{nbr} includes all neighbors of u_t.
- s_{nbr}: Shared neighbors connected to both u_h and u_t.

Candidate Set Updates. To update the candidate sets of these neighbors after adding the candidate pair $\{(u_h, v_h), (u_t, v_t)\}$ to the partial matching, we first define the function $N_M(v, u')$, which computes the valid neighbors of a candidate vertex v for a query

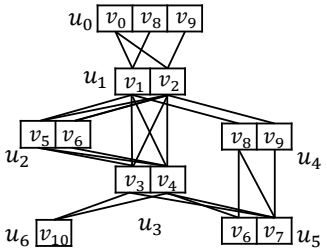
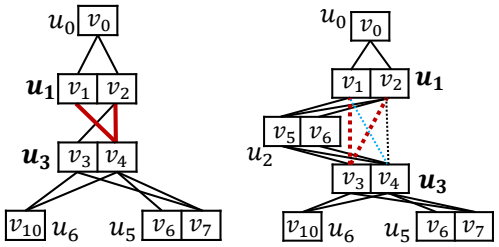


Figure 4: Initial candidate set structure $\mathcal{A}$.



(a) Edge-constrained $\mathcal{A}_e$ (b) Non-edge-constrained $\mathcal{A}_n$

Figure 5: Vertex-pair equivalence.

vertex u' under the current partial matching M, as follows:

$$N_M(v, u') = C_M(u') \cap N_G(v),$$

where $C_M(u')$ denotes the candidate set of u' under M, and $N_G(v)$ represents the neighbors of v in the data graph G.

Based on this definition, the candidate sets for the three categories of neighbors are updated as follows:

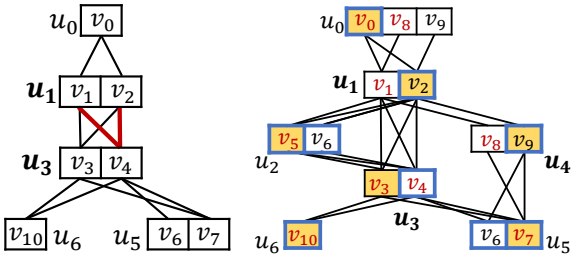
$$C_{M'}(h_{nbr}) = N_M(f(u_h), h_{nbr}) \cap C_M(h_{nbr}),$$
$$C_{M'}(t_{nbr}) = N_M(f(u_t), t_{nbr}) \cap C_M(t_{nbr}),$$
$$C_{M'}(s_{nbr}) = N_M(f(u_h), s_{nbr}) \cap N_M(f(u_t), s_{nbr}),$$

where M' denotes the updated partial matching after adding the candidate pair, and $f(u)$ represents the mapping of a query vertex u to a data vertex in M. For pairs without edge connection, the updates to h_{nbr} and t_{nbr} do not exclude u_t and u_h, respectively, as there is no edge constraint between u_h and u_t.

Example 4.3. We construct the auxiliary data structure $\mathcal{A}$ based on the initial candidate sets. In Figure 4, we list the candidates of each query vertex and extract edges from the data graph G_2 in Figure 3(b). Let us illustrate vertexpair equivalence for the query vertices (u_1, u_3) in both the edgeconstrained and unconstrained cases, using the auxiliary structures from Figure 5.

Edgeconstrained. As shown in Figure 5(a), since $e(u_1, u_3) \in E_Q$, we consider the candidate edges $\{e(v_1, v_4),\ e(v_2, v_3),\ e(v_2, v_4)\}$. The neighbors are classified into $h_{nbr} = \{u_0\}, t_{nbr} = \{u_5, u_6\}$. Selecting either $e(v_2, v_3)$ or $e(v_2, v_4)$ yields identical candidate sets for u_0, u_5, u_6, so these two edges are vertexpair equivalent. By contrast, $e(v_1, v_4)$ leads to a different neighbor candidate profile.

Unconstrained. As shown in Figure 5(b), when $e(u_1, u_3) \notin E_Q$, we form the Cartesian product $\{v_1, v_2\} \times \{v_3, v_4\}$. All combinations share the same candidates for u_0 and u_6, and in addition (v_1, v_3) and



(a) Vertex equivalence $\mathcal{A}_v$ (b) Group equivalence

Figure 6: Vertex equivalence and group equivalence.

(v_2, v_3) produce identical sets for u_5. Thus these two pairs are equivalent and only one is explored, whereas (v_1, v_4) and (v_2, v_4) yield different neighbor candidates and must be examined separately. ■

Using vertex-pair equivalence, we group duplicate search spaces of vertex pairs into distinct equivalence sets. For the vertex pairs within each set, we perform the backtracking step only once.

Complexity Analysis. To separate the candidate combinations of a vertex pair (u_h, u_t) into different vertex-pair equivalent sets, we can optimize the process by first separately traversing the candidate sets of u_h (the head vertices) and u_t (the tail vertices) instead of evaluating every combination directly. This approach reduces the computational overhead by avoiding a full pairwise traversal of candidate combinations.

For a given vertex pair (u_h, u_t), let $C(u_h)$ and $C(u_t)$ represent the candidate sets of u_h and u_t, respectively. The process involves the following three steps:

(1) **Head Traversal**: Traverse the candidate set $C_M(u_h)$ of u_h to partition it based on equivalence. For each candidate vertex $v_h \in C_M(u_h)$, we check the neighbors on h_{nbr}. This step has a time complexity of $O(|C_M(u_h)| \cdot \delta_Q)$, where δ_Q is the maximum degree of any query vertex in the query graph V_Q.

(2) **Tail Traversal**: Traverse the candidate set $C_M(u_t)$ of u_t to partition it based on equivalence. For each candidate vertex $v_t \in C_M(u_t)$, we check the neighbors on t_{nbr}. This step has a time complexity of $O(|C_M(u_t)| \cdot \delta_Q)$.

(3) **Combining Head and Tail**: After partitioning $C_M(u_h)$ and $C_M(u_t)$, we combine the partitions and then compare the neighbors on s_{nbr} to form vertex-pair equivalent sets. The time complexity of this step is $O(|C_M(u_h)| \cdot |C_M(u_t)| \cdot \delta_Q)$.

The time complexity of handling a single vertex pair (u_h, u_t) is:

$$O\left(|C_M(u_h)| \cdot \delta_Q + |C_M(u_t)| \cdot \delta_Q + |C_M(u_h)| \cdot |C_M(u_t)| \cdot \delta_Q\right),$$

where $|C_M(u_h)|$ and $|C_M(u_t)|$ represent the sizes of the valid candidate sets for u_h and u_t, respectively. Since $|C_M(u_h)|$ and $|C_M(u_t)|$ are bounded by $\max_{u_i \in V_Q} |C(u_i)|$, this simplifies to:

$$O\left(\max_{u_i \in V_Q} |C(u_i)|^2 \cdot \delta_Q\right).$$

In practice, the sizes of the valid candidate sets $|C_M(u_h)|$ and $|C_M(u_t)|$ are usually small, making the overhead of computing vertex pairs negligible.

Compare with Vertex Equivalence. The vertex equivalence technique proposed in VEQ [21] groups the candidates of each vertex that share the same neighbors in $\mathcal{A}$ into an equivalence set. By reusing the search results within the set, it avoids redundant computations at each depth of the search tree. Because vertex-pair equivalence provides a finer-grained equivalence, vertex equivalence is a special instance of vertex-pair equivalence. In other words, any redundancy pruned by vertex equivalence would also be pruned by vertex-pair equivalence.

Theorem 4.1. *For an adjacent vertex pair (u_h, u_t), if u_h belongs to a vertex equivalence set, then there always exists a corresponding vertex-pair equivalence set for (u_h, u_t). In this vertex-pair equivalence set, the head vertices of the pairs share identical neighbors in the search space, and the tail vertices map to the same vertex.*

Proof. Consider an adjacent vertex pair (u_h, u_t), and assume that u_h belongs to a vertex equivalence set. Let the vertex equivalence set of u_h be denoted as $\{v_h^0, v_h^1, \ldots, v_h^k\}$, where k is the number of candidates in the set. By the definition of vertex equivalence, all candidates in this set share the same neighbors in the search space. Now, consider u_t, the vertex adjacent to u_h, and let $\{v_t^0, v_t^1, \ldots, v_t^m\}$ denote the candidates for u_t. We now construct vertex-pair equivalence sets for (u_h, u_t) by defining, for each vertex v_t^i in $v_t^0, v_t^1, \ldots, v_t^m$, a vertex-pair equivalence set as $\{(v_h^0, v_t^i), (v_h^1, v_t^i), \ldots, (v_h^n, v_t^i)\}$. In this construction, the head vertices v_h^j ($j \in \{0, 1, \ldots, n\}$) in each pair share identical neighbors in the search space, as guaranteed by the vertex equivalence of u_h, while the tail vertices are fixed to the same vertex v_t^i.

Thus, vertex equivalence is inherently subsumed by vertex-pair equivalence, as the latter captures all redundancies identified by the former while providing a finer-grained equivalence structure. □

This indicates that we can always construct structurally equivalent sets from the vertex equivalence sets for the vertex pair without edge constraints, thereby pruning at least as many redundant computations in the search space as vertex equivalence.

Example 4.4. Considering the auxiliary data structure $\mathcal{A}_v$ shown in Figure 6(a). the candidates v_1, v_2 of u_1 are *vertex equivalent* because they have the same neighbors on u_0, u_3. The candidate vertex pairs $\{(v_1, v_3), (v_2, v_3)\}$ and $\{(v_1, v_4), (v_2, v_4)\}$ are edge equivalent sets. For the auxiliary structure $\mathcal{A}_e$ in Figure 5(a), none vertex equivalent set can be constructed on neither u_1 nor u_3. But we can recognize the vertex-pair equivalent set $\{(v_1, v_4), (v_2, v_4)\}$ as we depicted in Example 4.3. ■

5 GROUP EQUIVALENCE

While vertex-pair equivalence improves the detection of overlap in the search space, it remains limited to vertex pairs. To enable finer-grained and more comprehensive overlap detection, we propose *group equivalence*, which generalizes equivalence to a set of vertices.

Definition 5.1 (Group equivalence). Given two partial matchings M_1 and M_2 with all the matched vertices following the matching order φ of a query graph Q, the partial matchings M_1 and M_2 are *group equivalent* if the valid candidate sets for all unmatched vertices (i.e., independent vertices) are identical.

Lemma 5.2. *If two partial matchings M_1 and M_2 are group equivalent, then the subtrees in the DFS search tree rooted at these partial matchings are isomorphic. Consequently, any subtree rooted at M_2 is redundant and can be safely pruned without affecting the correctness of the search process.*

Proof. By the definition of group equivalence, M_1 and M_2 have identical valid candidate sets for all unmatched vertices. Since the candidate sets determine the remaining search space, the subtrees rooted at M_1 and M_2 in the DFS tree are isomorphic. Thus, pruning the subtree rooted at M_2 eliminates redundancy without affecting the correctness of the search process. □

Group equivalence treats all matched query vertices as a single group, enabling the detection of duplicate computations across the entire partial matching process progressively during backtracking. However, directly computing group equivalence is computationally expensive, as the number of partial matchings for a query graph Q grows exponentially with the depth of the matching process. To overcome this challenge, we propose constructing group equivalence incrementally based on vertex-pair equivalence computations.

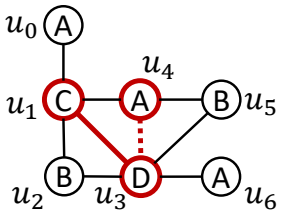
Delayed Neighbors. During vertexpair equivalence computation, there may be cases where the candidate sets of a vertex pair are not equivalent at the current depth. However, as additional vertices are matched, the introduction of new constraints on the candidate sets may cause the group (i.e., a set of vertex pairs combined with the newly added vertex) to evolve into equivalence. This process of equivalence evolution may continue iteratively. To detect group equivalence in a timely manner, we propose delayed neighbors, denoted as d_{nbr}.

Delayed neighbors defer part of the equivalence computation to deeper levels of the search tree, allowing us to identify redundant computations that emerge as the search progresses. We denote u_{nxt} as the next vertex after u_t in the matching order φ and $u^{\succ}$ as the set of vertices matched after u_t along φ. The computation of d_{nbr} is: if the vertex pair has an edge constraint, d_{nbr} is assigned as u_{nxt}. Otherwise, d_{nbr} is computed as $N(n_{nbr}) \cap N(u_t) \cap u_t^{\succ}$.

Example 5.3. Considering the example in Figure 6(b). The matching order is $u_1 \prec u_3 \prec u_4 \prec u_0 \prec u_2 \prec u_5 \prec u_6$. (u_1, u_3) is the first vertex pair to be matched. There are four valid combinations $(v_1, v_3), (v_2, v_3), (v_1, v_4), (v_2, v_4)$ and the next vertex in φ is u_4. The four categories of neighbors are $h_{nbr} = \{u_0\}, t_{nbr} = \{u_6\}, s_{nbr} = \{u_2\}$ and $d_{nbr} = \{u_5\}$. The combinations (v_1, v_3) and (v_2, v_3) have the same neighbors on u_0, u_2, and u_6, but different neighbors on u_5. Therefore, they are not *vertex-pair equivalent*. However, since the difference is on a delayed neighbor, we consider them as *potentially group equivalent* and add this equivalent set to the partial matching M, updating the candidates of the neighbors u_0, u_2, u_5, and u_6. Then we continue to compute the vertex-pair equivalence for the next vertex pair (u_3, u_4). The combinations are (v_3, v_8) and (v_4, v_8) and they have the same neighbors on u_5. And then the partial matching $M_0 = \{(u_1, v_1), (u_3, v_3), (u_4, v_8)\}$ and $M_1 = \{(u_1, v_1), (u_3, v_4), (u_4, v_8)\}$ highlighted in red forms a *group equivalent set* because the same neighbors on the unmatched vertices. The partial matching $\{(u_1, v_2), (u_3, v_3), (u_4, v_8)\}$ highlighted in yellow shadow and $\{(u_1, v_1), (u_3, v_3), (u_4, v_8)\}$ highlighted in blue thick border will be explored separately. ■

Query Vertex	Candidates
u_0	$\{v_0, v_8, v_9\}$
u_1	$\{v_1, v_2\}$
u_2	$\{v_5, v_6\}$
u_3	$\{v_3, v_4\}$
u_4	$\{v_8, v_9\}$
u_5	$\{v_6, v_7\}$
u_6	$\{v_{10}\}$

(a) Candidate sets



(b) Matching order

Figure 7: Matching order and candidate sets of query graph Q_2 in Figure 3(a).

Complexity Analysis. We compute the group equivalent sets through a progressive process. At each step, the complexity is equivalent to that of computing vertex-pair equivalence. We denote the time complexity of vertex-pair equivalence as $\mathcal{T}_{vp}$. Thus, the overall time complexity of complete equivalence is: $O\left(|\varphi| \cdot \mathcal{T}_{vp}\right)$.

6 OPTIMIZATIONS

6.1 Matching Order

In this section, we propose an improved matching order for query vertices to minimize the number of subtrees built during the backtracking process. We prioritize selecting vertices with the fewest candidates. In case of ties, we choose the vertex with the highest degree to other unmatched vertices.

It is important to note that once all neighbors of a vertex u have been added to the partial matching, the selection of candidates for u no longer affects the valid candidates of other unmatched vertices, as there are no edge constraints between them. Additionally, the valid candidates for u are equivalent, as they all share the same neighbors. We refer to such vertices as *independent vertices* and place these vertices at the end of the matching order.

We begin by applying the concept of *leaf decomposition* [4] to separate degree-1 vertices (leaf vertices). For degree-1 vertices, once their sole neighbor is matched, they no longer introduce edge constraints with other unmatched vertices, and thus, are considered independent vertices. And then, after selecting a query vertex, we check if any remaining unmatched vertices do not introduce edge constraints with other unmatched vertices. These vertices are then classified as independent vertices.

Example 6.1. Consider the example in Figure 3. The initial candidate sets for each vertex in query graph Q_2 are shown in Figure 7(a). First, we put the degree-1 vertices u_0 and u_6 into the independent set. For the remaining vertices, since they all have the same number of candidates, we select them arbitrarily. Assume we select u_1 and u_3 as the first two vertices. Then, u_2 does not introduce any edge constraints with the unmatched vertices, so we place it in the independent set. If we select u_4 next, then u_5 can be added to the independent set. In summary, the matching order for Q_2 is $u_1 \prec u_3 \prec u_4$, which is highlighted with a bold red border in Figure 7(b). The independent set is $\{u_0, u_2, u_5, u_6\}$. ■

Complexity Analysis. After selecting a query vertex, we need to check whether its neighbors become independent vertices, which has a time complexity of $O\left(\delta_Q\right)$, where δ_Q represents the maximum vertex degree of query graph Q. Selecting the next query vertex from the remaining unmatched vertices incurs a time complexity of $O\left(n\right)$. Therefore, the time complexity for generating the complete matching order is $O\left(|E_Q|^2 \delta_Q\right)$.

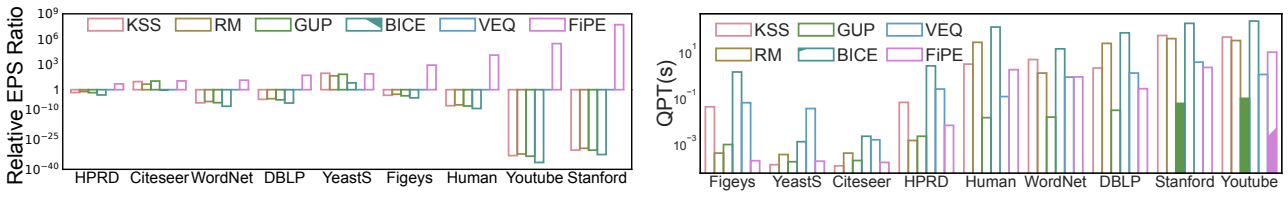
6.2 Embedding Enumeration

In this subsection, we propose an efficient enumeration technique for obtaining the embeddings for the complete equivalent set and handling the latent vertex conflicts, which occur when multiple query vertices map to the same data vertex in an embedding. After completing the matching of all non-independent vertices, it becomes crucial to quickly compute the embeddings based on the constructed complete equivalent sets and the candidate sets of the independent vertices. Since there are no edge constraints among the candidates of the independent set, we only need to ensure that multiple query vertices do not map to the same data vertex in the embedding, i.e., ensuring the embedding is injective.

For subgraph matching algorithms that accelerate computation by reusing duplicate results, conflict handling is a significant challenge. To reuse overlapping subtrees in the search tree, we must ensure that the candidate sets of unmatched vertices do not conflict with the already computed overlapping parts. While some algorithms propose complex conflict resolution methods that record extensive intermediate information during the search [21], it is often unclear whether the overhead of maintaining this information justifies the pruning efficiency gained.

We propose a relatively simple and efficient conflict-handling approach. FiPE defers conflict checking until the end of the enumeration (after matching all non-independent vertices). By postponing conflict resolution, this approach avoids discarding an entire subtree due to a single vertex conflict, ensuring maximum utilization of overlaps. Following the matching order, we sequentially select vertex pairs from the equivalent set of adjacent vertex pairs. We simply enumerate all possible partial matchings and check for vertex conflicts. Subsequently, we enumerate the embeddings of independent vertices for each partial matching. For the partial matchings that have no duplicate candidates with all the candidates of independent vertices, we only need to count the valid embeddings of the independent vertices once. The whole embeddings can then be obtained by applying Cartesian product on them.

Example 6.2. Let us consider the group equivalent set (highlighted in red) in Figure 6(b): $M_0 = \{(u_1, v_1), (u_3, v_3), (u_4, v_8)\}$, $M_1 = \{(u_1, v_1), (u_3, v_4), (u_4, v_8)\}$. The candidate sets for independent vertices are $(u_0, \{v_0, v_8\}), (u_2, \{v_5\}), (u_6, \{v_{10}\}), (u_5, \{v_7\})$. The candidate set of M_0 is $\{v_1, v_3, v_8\}$ which has conflicts with the candidate set of u_0 and so does M_1. We need to enumerate the embeddings for them separately. And the whole embeddings highlighted in red are $\{(u_1, v_1), (u_3, v_3), (u_4, v_8), (u_0, v_0), (u_2, v_5), (u_6, v_{10}), (u_5, v_7)\}$ and $\{(u_1, v_1), (u_3, v_4), (u_4, v_8), (u_0, v_0), (u_2, v_5), (u_6, v_{10}), (u_5, v_7)\}$. The whole embeddings for partial matching $M_2 = \{(u_1, v_2), (u_3, v_3), (u_4, v_9)\}$ are highlighted in yellow shadow and the ones for $M_3 = \{(u_1, v_2), (u_3, v_4), (u_4, v_9)\}$ are highlighted in blue thick border. ■



(a) Relative EPS ratio (VEQ = 1).

(b) Query Processing Time.

Figure 8: Overall Performance, where query graph size = (8, 10, 12, 16, 20) and label size = (15, 30, 45, 60).

Table 2: Data graphs with vertex labels.

Dataset	$\|V\|$	$\|E\|$	d_G	Type
Figeys	2,239	6,432	5.7	Protein
YeastS	2,361	7,182	6.1	Protein
Citeseer	3,279	4,552	2.8	Citation
Human	4,674	86,282	36.9	Protein
HPRD	9,303	34,998	7.5	Protein
WordNet	146,005	656,999	9.0	Lexical
Standford	281,903	1,992,636	14.1	Web
DBLP	317,080	1,049,866	6.6	Collab.
Youtube	1,134,890	2,987,624	5.3	Social

Complexity Analysis. Let $C_M(u_i)$ denote the equivalent sets of query vertex u_i in partial embedding M. The overall complexity of embedding enumeration is: $O\left(\max_{u_i \in V_Q} |C_M(u_i)|^{|V_Q|}\right)$.

7 EXPERIMENTS

7.1 Experimental Setting

Comparison Methods. We select five representative subgraph matching algorithms for comparison, including VEQ [21], BICE [6], RM [35], KSS [40], and GuP [2]. The source codes of the comparison methods are obtained from the authors. FiPE supports various filter techniques for candidate set construction. For evaluation, we adopt CFL [4] as the default filter method.

Data Graphs. We conduct experiments using both real-world and synthetic datasets. For real-world datasets, we choose a diverse set of 9 graphs from different domains, including protein interactions, academic collaboration, lexical networks, and social networks. A detailed description of these graphs is provided in Table 2. Experiments are performed on all real-world datasets using label sizes of 15, 30, 45, and 60 by default. Vertex labels are randomly assigned following the methodology of the prior works [2, 34, 44].

For synthetic datasets, we employ the EvoGraph [28] method to scale small graphs. This allows us to simulate real-world data and evaluate the scalability of various methods. The vertex labels are assigned by following the same procedure as for real-world graphs.

Query Graphs. We utilize a sampling method consistent with prior research [2, 13, 34] to generate most of the query graphs. We perform a Metropolis-Hastings random walk [15] on the data graphs and select induced subgraphs as queries. We generate a diverse set of query graphs by varying two key parameters:

- Query graph size: 8, 10, 12, 16, and 20 vertices.
- Label size: 15, 30, 45, and 60 distinct labels.

We use these label and query sizes as the default setting for all experiments. For each combination of query size and label size, we collect 1,000 random query graphs for each real-world data graph, leading to a total of 180,000 query graphs.

We also include the universal query graphs used in Circinus [18] and apply them to all data graphs. Note that some universal queries may not have any embeddings in certain data graphs.

Metrics. We employ three key metrics to evaluate the effectiveness and efficiency of the methods under comparison:

- *Query Processing Time (QPT).* Query Processing Time (QPT) measures the time required to return a specified number of embeddings and is commonly used in traditional subgraph matching methods [2, 13, 21, 34]. Consistent with prior research, we evaluate each algorithms performance based on the QPT to return 10^5 embeddings within a 300second time limit.
- *Embeddings per Second (EPS).* EPS is defined as the average number of embeddings returned per second. The recent survey [44] defines EPS as a unified throughput metric that combines the number of reported embeddings with the time taken to produce them. Unlike the QPT metric, which measures only execution time under fixed limits and can be skewed by threshold settings, EPS yields a more stable comparison of subgraphmatching algorithms [44]. Following the setup in [44], we set the time limit for the EPS metric to one second.
- *Relative EPS Ratio.* The Relative EPS Ratio is the ratio of the EPS of a given method to the EPS of the baseline method. By definition, the Relative EPS Ratio for the baseline method is 1. This metric facilitates a clear comparison between different methods.

Experimental Environment. All experiments were executed on a server running Ubuntu 22.04, equipped with an Intel(R) E5-2596v4 CPU @ 2.2 GHz and 128 GB of RAM.

7.2 Overall Performance

7.2.1 The EPS metric. Figure 8(a) presents the relative EPS ratio on 9 real-world graphs under the default query label sizes, with VEQ as the baseline method. It is evident that FiPE outperforms other methods on the majority of the graphs, with substantial improvements of 6 to 8 orders of magnitude on large-scale datasets such as Stanford and Youtube compared to the other methods. It is worth noting that the EPS on large-scale graphs is exceptionally high. This is attributed to the Cartesian product approach used to enumerate candidates in the independent set, which allows the

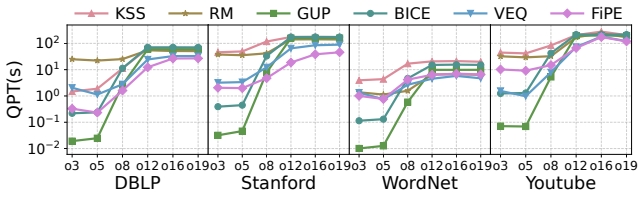
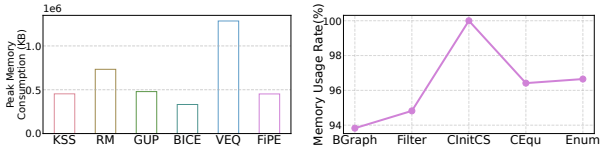


Figure 9: QPT performance under varying output limits.



(a) Peak memory consumption (b) Memory usage of each stage

Figure 10: Memory consumption.

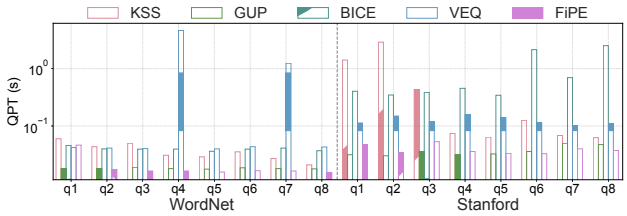


Figure 11: QPT performance of universal queries.

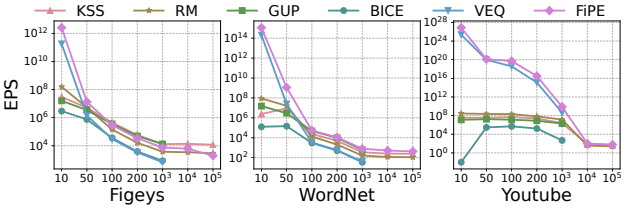


Figure 12: Effect of label size of data graphs.

rapid computation of a large number of embeddings in a short time. Overall, FiPE demonstrates a significant advantage on large-scale graphs, where duplicate computations are more frequent, and our method effectively exploits these redundancies. For small-scale graphs, existing subgraph matching algorithms are already highly efficient, typically processing queries within milliseconds, so the performance gap between existing methods is relatively small.

7.2.2 The QPT metric. We conducted experiments with an output limit of 10^5, evaluating them using the QPT metric. Figure 8(b) presents the QPT results across six methods, averaged on the default label and query sizes. It shows that FiPE achieves strong performance on most graphs. For smaller data graphs, GuP, RM, and FiPE all exhibit fast processing, with average execution times below 1 ms. While GuP shows superior QPT, its efficiency declines when enumerating a large number of embeddings. However, VEQ performs relatively poorly under QPT compared to EPS, as its advantages become apparent when generating substantial results.

Based on these experiments, we explore FiPEs QPT performance under output limits of 10^3, 10^5, 10^8, 10^{12}, 10^{16}, and 10^{19}, denoted by o3, o5, o8, o12, o16, and o19, respectively. The results in Figure 9 reveal that the QPT performance of most methods is significantly sensitive to the output limit. With low limits, GuP achieves submillisecond runtimes but deteriorates rapidly as the cap increases. In contrast, the performance of FiPE improves with higher limits, a direct result of its finegrained equivalence sharing. This confirms FiPEs suitability for largeoutput or fullenumeration scenarios.

7.2.3 Peak Memory Consumption. Figure 10(a) reports the peak memory usage of different methods, during the EPS evaluation in Section 7.2.1, measured by the Maximum Resident Set Size on Ubuntus `time` tool. FiPE exhibits outstanding memory efficiency, requiring less memory than both VEQ and BICE, and slightly less than GuP and KSS.

To analyze FiPEs internal memory profile, we decompose its execution into five stages: Build Graph (BGraph), Filter, Compute Initial Candidate sets (CInitCS), Compute Equivalences (CEqu), and Enumerate (Enum). We take the query with the largest memory usage as a case study, which is the query number 195 on the Youtube with label size = 60, query size = 20. Figure 10(b) shows the percentage of memory usage at each stage compared to the maximum memory usage. The BGraph stage alone accounts for 94% of the maximum footprint, with CInitCS reaching the overall peak. The CEqu phase adds no significant extra overhead. FiPEs simple yet fast vertexconflict detection mechanism avoids storing extraneous data structures while enabling extensive reuse of duplicate computations.

7.2.4 Universal Queries. The queries extracted from the data graph will generate at least one result. To evaluate the performance on zero result cases, we conduct experiments with some universal queries from Circinus [18]. Because the result might be zero, we use QPT as the comparison metric. RM is excluded from this section of experiments, as its implementation does not correctly handle queries with zero results. Figure 11 reports the QPT of universal queries from Circinus [18] on the WordNet and Standford datasets with four default label sizes. Queries with zero embeddings are marked with an asterisk (*) in the figure. Overall, GuP and FiPE demonstrate superior performance across both datasets.

7.3 Scalability Evaluation

We analyze the scalability of FiPE by examining its performance under the EPS metric with respect to three factors: the label size, the size of the data graph, and the size of the query graph.

7.3.1 Impact of the Label Size of Data Graph. We begin by analyzing seven label sizes10, 50, 100, 200, 10^3, 10^4, and 10^5 under the default query size settings. Figure 12 shows the EPS results for the Figeys, WordNet, and Youtube graphs. Due to memory overflow issues faced by BICE, VEQ, and GuP when handling large label sets, these methods could not process data graphs with 10^4 and 10^5 labels. In most cases, FiPE outperforms other methods. From the trends observed in the figure, the performance differences between

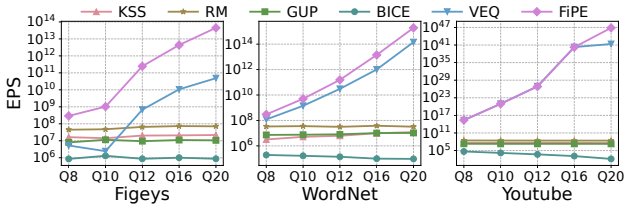


Figure 13: Effect of query graph size.

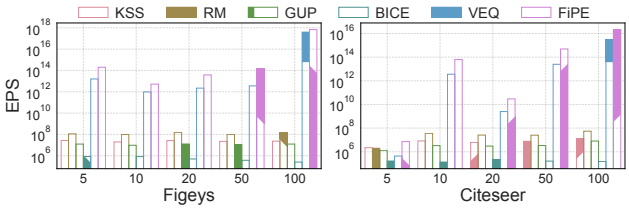


Figure 14: Effect of data graph size.

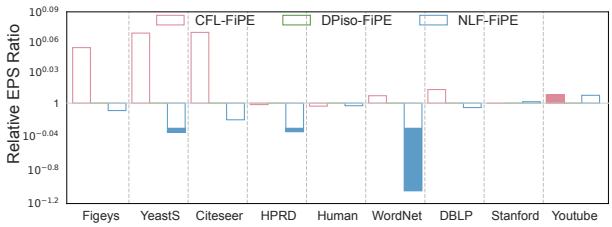
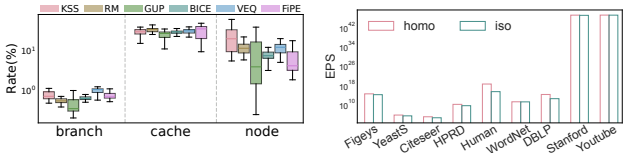


Figure 15: Relative EPS ratio (DPiso-FiPE = 1).



(a) Locality evaluation (b) EPS under iso. and homo. semantics

Figure 16: Locality evaluation and two semantics.

the various methods gradually diminish as the label size increases. For very large label sizes, the EPS results of the methods converge and show negligible differences. As the label size increases, the candidate set for each query vertex becomes smaller. In extreme cases, each query vertex u has only one candidate. At this point, the backtracking process becomes simpler, and no redundant computations can be exploited, which leads to minimal differences between the methods.

7.3.2 Impact of Query Graph Size. In this section, we analyze the performance of comparison methods and FiPE under different query graph sizes: 8, 10, 12, 16, and 20 (with the default label size settings). Figure 13 illustrates the EPS results across these query sizes for three datasets. In the majority of scenarios, FiPE demonstrates superior performance compared to other algorithms. Notably, the efficiency of FiPE increases with the size of the query graph.

7.3.3 Impact of Data Graph Size. To assess the influence of data graph size, synthetic graphs are created by scaling the smaller Figeys and Citeseer graphs using EvoGraph [28] with scaling factors of 5, 10, 20, 50, and 100. We apply the default four kinds of label sizes for each data graph and then generate 100 query graphs for each default query sizes. This results in a total of 2,000 queries for each scaled data graph. Figure 14 illustrates the EPS performance of all methods on these synthetic graphs. FiPE consistently achieves the best performance across all scenarios, showcasing its notable scalability advantages. Similar to the observations in Section 7.2, FiPE demonstrates a more pronounced advantage over other methods as the size of the data graph increases.

7.4 Locality Evaluation

Figure 16(a) reports the branch-, cache-, and node-miss rates of all methods on four large graphs (Youtube, Stanford, DBLP, WordNet), aggregated over 8,000 queries spanning five query sizes (8, 10, 12, 16, 20) and four label sizes (15, 30, 45, 60). Using `perf` tool, we measure branch-miss rate (instructionfetch locality), cache-miss rate (on-chip data locality), and node-miss rate (memorypage locality) to evaluate each algorithms runtime behavior. FiPE exhibits excellent locality: its node-miss rate is among the lowest of all methods, branch-miss hovers around 1%, and cache-miss remains at a moderate level. These stable, low miss rates reflect FiPEs predictable control flow and efficient data access patterns derived from fine-grained equivalence sharing. By contrast, GuP shows large fluctuations in both branch- and node-miss rates across queries.

7.5 Evaluation on Iso. and Homo. Semantics

By default, FiPE operates under subgraphisomorphism semantics, guaranteeing a onetoone mapping between query and data vertices. We further extended FiPE to handle homomorphism queries, where multiple query vertices may map to the same data vertex, by replacing the vertexconflict enumeration with a simple Cartesian product over each equivalence sets candidates. Figure 16(b) plots EPS, averaged on the default label and query sizes, under both isomorphism and homomorphism semantics. FiPE delivers excellent EPS performance under both semantics. Across all nine graphs, homomorphism consistently yields higher throughput than isomorphism, with the largest gaps observed on Human and DBLP.

7.6 Impact of Filtering Technique

In this subsection, we analyze the impact of employing various filtering techniques. Although CFL [4] is our default, FiPE can seamlessly integrate with a wide range of filtering methods. We compare NLF [46], CFL, and DPiso [13] across all datasets.

NLF offers a less restrictive filtering process, allowing for the rapid generation of candidate sets but with lower accuracy, where accuracy refers to the proportion of candidates in the set that ultimately contribute to valid embeddings. In contrast, DPiso applies more stringent checks, leading to higher accuracy at the cost of increased computation time. CFL strikes a balance between the two, delivering moderate accuracy and execution time. Detailed explanations of these filtering methods can be found in Section 2.2.

Table 3: The overlap statistic at different depths in the backtracking process.

	depth	7	6	5	4	3	2	1
Citeseer	# trees	51.91	18.79	9.31	9.21	44.95	10.64	6.72
General	# buckets	1.00	1.00	1.00	1.00	1.02	1.04	1.05
	ratio(%)	97.78	93.50	73.12	44.89	33.37	32.30	29.61
Citeseer	# trees	5.60	2.66	2.12	1.67	1.67	1.67	1.67
FiPE	# buckets	2.03	1.55	1.94	1.54	1.55	1.55	1.55
	ratio(%)	12.19	8.13	4.41	3.06	2.89	2.89	2.89
Figeys	# trees	56.45	51.63	183.20	364.96	493.82	393.59	1149.44
General	# buckets	1.00	1.01	1.09	1.19	1.96	2.22	2.98
	ratio(%)	98.15	97.81	96.09	95.52	93.66	94.75	94.50
Figeys	# trees	37.83	47.65	49.53	49.51	49.51	49.51	49.51
FiPE	# buckets	10.30	17.63	19.26	19.27	19.22	19.22	19.22
	ratio(%)	46.88	39.53	30.26	27.72	28.11	28.11	28.11
YeastS	# trees	69.70	45.61	39.99	39.13	43.86	54.50	85.05
General	# buckets	1.00	1.00	1.00	1.07	1.84	2.39	2.56
	ratio(%)	98.47	97.63	94.19	90.18	82.35	83.36	84.76
YeastS	# trees	63.99	56.43	14.28	14.33	14.35	14.35	14.35
FiPE	# buckets	6.35	34.04	11.86	11.69	11.73	11.70	11.70
	ratio(%)	53.58	27.07	8.07	5.76	5.36	5.84	5.84

Figure 15 depicts the Relative EPS ratio of the three filters compared to DPiso-FiPE. We apply the default label sizes for each data graph and then generate 100 query graphs for each default query size. NLF-FiPE exhibits relatively poor performance in most scenarios, indicating that FiPE is better suited to filtering methods capable of producing candidate sets with relatively higher accuracy. Overall, CFL-FiPE consistently achieves the best results across the majority of datasets, establishing CFL as the recommended default filtering technique for our approach.

7.7 The Performance on Reducing Overlapping

We assess the effectiveness of FiPE in reducing redundant computations by analyzing the similarity between subtrees in its backtracking process. Throughout the process, we record the candidates of unmatched vertices at each depth, approximate their Jaccard similarity using MinHashLSH [45], and group those with a similarity exceeding 0.9 into a single bucket. Table 3 summarizes the average number of subtrees, the number of buckets, and the overlap ratio per dimension for 100 queries of size 8 across the Figeys, Citeseer, and YeastS datasets. The overlap ratio is calculated as the average of individual overlap ratios from each query. The results demonstrate that FiPE effectively reduces the number of constructed search spaces and significantly lowers the overlap ratio. Remarkably, FiPE achieves a consistent overlap rate at lower depths since independent vertices are matched at these depths, and their candidate sets remain unchanged after matching other vertices.

7.8 Discussion

FiPE Strengths and Limitations: The experimental results demonstrate that FiPE achieves dramatic speedups when exhaustively enumerating subgraph matches on large-scale graphs. It is agnostic to the global shape of the query graphwhether cyclic, acyclic, or tree-likebecause it relies only on local structural equivalences. However, this generality comes with overhead: if a query or data graph lacks repeated structure, few equivalences exist, and the cost of maintaining them may outweigh the benefit.

Directed Graphs and DAGs: FiPE readily extends to directed graphs by splitting each vertexs neighbors into incoming and outgoing partitions. This in-out neighbor partitioning simplifies the equivalence detection, since the candidates are considered equivalent only if they match both sets of in- and out-neighbors. In directed acyclic graphs, the absence of cycles imposes a strict ancestordescendant structure, resulting in smaller, finergrained equivalence classes. Thus, equivalence construction incurs lower overhead, which leads to faster enumeration and improved performance in scenarios requiring only a small number of embeddings.

Discussion on How to Integrate into Graph Database Systems. FiPEs techniques can be seamlessly integrated into the pattern matching pipeline of graph database systems, which typically include query plan construction, optimization, graph traversal and matching, and result return. During optimization, our ordering technique can serve as an optimization rule to construct a query order that avoids redundant backtracking on "independent" vertices. In the traversal stage, each new vertex match builds equivalence sets with already matched vertices; by aggregating candidate branches for the next step, FiPE uncovers shared computations, reducing overhead and accelerating matching without altering the pipeline. For isomorphism queries, the vertexconflict technique enables rapid enumeration of embeddings within these equivalence sets.

Comparison to GuP: The guardbased pruning technique of GuP records failure patterns encountered during backtracking and uses them to avoid repeating the same failed branches. In contrast, FiPE constructs fine-grained equivalent sets, which not only prevent repeated failures within each equivalent set but also enable the reuse of successful search results by sharing embeddings among equivalent candidates. This broader form of pruning makes FiPE more powerful, but it also incurs additional computational overhead. Thus, when only a small number of embeddings is required and the total enumeration time is short, the lighterweight GuP outperforms FiPE (as shown in Figure 8(b)); however, in scenarios demanding all or enormous embeddings (where enumeration time dominates overall cost), FiPEs aggressive redundancy elimination yields superior performance (as shown in Figure 8(a) and Figure 9).

8 CONCLUSION

In this paper, we propose a novel algorithm, FiPE, that accelerates subgraph matching through fine-grained and powerful equivalences. FiPE optimizes subgraph matching by identifying and reusing similar search subtrees. By introducing vertex-pair equivalence and group equivalence, FiPE enables finer-grained detection of redundant computations. Vertex-pair equivalence identifies redundant computations between pairs of vertices, while group equivalence generalizes this concept to detect redundancies across the entire matching process. Extensive experiments demonstrate that FiPE reduces query processing time and improves scalability, outperforming state-of-the-art methods.

ACKNOWLEDGMENTS

This work was substantially supported Key Projects of the National NSF of China (U23A20496). Lei Zou was supported by The National Key Research and Development Program of China under grant 2023YFB4502303.

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 Trans. Database Syst.* 42, 4, Article 20 (oct 2017), 44 pages. doi:10.1145/3129246

[2] Junya Arai, Yasuhiro Fujiwara, and Makoto Onizuka. 2023. GuP: Fast Subgraph Matching by Guard-Based Pruning. *Proc. ACM Manag. Data* 1, 2, Article 167 (jun 2023), 26 pages. doi:10.1145/3589312

[3] Bibek Bhattarai, Hang Liu, and H. Howie Huang. 2019. CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 14471462.

[4] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient Subgraph Matching by Postponing Cartesian Products. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 11991214. doi:10.1145/2882903.2915236

[5] Vincenzo Carletti, Pasquale Foggia, Alessia Saggese, and Mario Vento. 2018. Challenging the Time Complexity of Exact Subgraph Isomorphism for Huge and Dense Graphs with VF3. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 40, 4 (2018), 804–818. doi:10.1109/TPAMI.2017.2696940

[6] 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 (may 2023), 21862198. doi:10.14778/3598581.3598591

[7] Vibhor Dodeja, Mohammad Almasri, Rakesh Nagi, Jinjun Xiong, and Wen-mei Hwu. 2022. PARSEC: Parallel subgraph enumeration in CUDA. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 168–178.

[8] Kayhan Erciyes. 2023. Graph-theoretical analysis of biological networks: a survey. *Computation* 11, 10 (2023), 188.

[9] Wenfei Fan. 2012. Graph pattern matching revised for social network analysis. In *Proceedings of the 15th international conference on database theory*. 8–21.

[10] Binbin Fang, Hanzhu Chen, Weiwei Wang, and Yansong Wang. 2024. GraphFA: Graph Enhanced Fraud Detectors with Camouflage Detection for Financial Anti-Fraud. In *2024 9th International Conference on Intelligent Computing and Signal Processing (ICSP)*. IEEE, 323–327.

[11] Yunjun Gao, Tianming Zhang, Linshan Qiu, Qingyuan Linghu, and Gang Chen. 2021. Time-Respecting Flow Graph Pattern Matching on Temporal Graphs. *IEEE Transactions on Knowledge & Data Engineering* 33, 10 (2021), 3453–3467.

[12] Nabil Guelzim, Samuele Bottani, Paul Bourgine, and François Képès. 2002. Topological and causal structure of the yeast transcriptional regulatory network. *Nature genetics* 31, 1 (2002), 60–63.

[13] 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 *SIGMOD 2019 - Proceedings of the 2019 International Conference on Management of Data*. 1429–1446.

[14] 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 the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 337348. doi:10.1145/2463676.2465300

[15] W. K. Hastings. 1970. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika* 57, 1 (04 1970), 97–109. doi:10.1093/biomet/57.1.97 arXiv:https://academic.oup.com/biomet/article-pdf/57/1/97/23940249/57-1-97.pdf

[16] Huahai He and Ambuj K. Singh. 2008. Graphs-at-a-Time: Query Language and Access Methods for Graph Databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 405418. doi:10.1145/1376616.1376660

[17] Jiewen Huang, Daniel J Abadi, and Kun Ren. 2011. Scalable SPARQL querying of large RDF graphs. *Proceedings of the VLDB Endowment* 4, 11 (2011), 1123–1134.

[18] Tatiana Jin, Boyang Li, Yichao Li, Qihui Zhou, Qianli Ma, Yunjian Zhao, Hongzhi Chen, and James Cheng. 2023. Circinus: Fast Redundancy-Reduced Subgraph Matching. *Proc. ACM Manag. Data* 1, 1, Article 12 (may 2023), 26 pages. doi:10.1145/3588692

[19] Alpár Jüttner and Péter Madarasi. 2018. VF2++An improved subgraph isomorphism algorithm. *Discrete Applied Mathematics* 242 (2018), 69–81. doi:10.1016/j.dam.2018.02.018 Computational Advances in Combinatorial Optimization.

[20] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An Active Graph Database. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 16951698. doi:10.1145/3035918.3056445

[21] 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 the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 925937. doi:10.1145/3448016.3457265

[22] Hyeonji Kim, Juneyoung Lee, Sourav S. Bhowmick, Wook-Shin Han, JeongHoon Lee, Seongyun Ko, and Moath H.A. Jarrah. 2016. DUALSIM: Parallel Subgraph Enumeration in a Massive Graph on a Single Machine. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 12311245. doi:10.1145/2882903.2915209

[23] Jinha Kim, Hyungyu Shin, Wook-Shin Han, Sungpack Hong, and Hassan Chafi. 2015. Taming subgraph isomorphism for RDF query processing. *arXiv preprint arXiv:1506.01973* (2015).

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

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

[26] Qing Liu, Minjun Zhao, Xin Huang, Jianliang Xu, and Yunjun Gao. 2020. Truss-based Community Search over Large Directed Graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2183–2197.

[27] Tinghuai Ma, Siyang Yu, Jie Cao, Yuan Tian, Abdullah Al-Dhelaan, and Mznah Al-Rodhaan. 2018. A comparative study of subgraph matching isomorphic methods in social networks. *IEEE Access* 6 (2018), 66621–66631.

[28] Himchan Park and Min-Soo Kim. 2018. EvoGraph: An Effective and Efficient Graph Upscaling Method for Preserving Graph Properties. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (London, United Kingdom) *(KDD '18)*. Association for Computing Machinery, New York, NY, USA, 20512059. doi:10.1145/3219819.3220123

[29] Miao Qiao, Hao Zhang, and Hong Cheng. 2017. Subgraph matching: on compression and computation. *Proceedings of the VLDB Endowment* 11, 2 (2017), 176–188.

[30] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2017. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing. *Proc. VLDB Endow.* 11, 4 (dec 2017), 420431.

[31] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2018. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing. *Proc. VLDB Endow.* 11, 4 (oct 2018), 420431. doi:10.1145/3164135.3164139

[32] Yingxia Shao, Bin Cui, Lei Chen, Lin Ma, Junjie Yao, and Ning Xu. 2014. Parallel subgraph listing in a large-scale graph. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of Data*. 625–636.

[33] Junshuai Song, Xiaoru Qu, Zehong Hu, Zhao Li, Jun Gao, and Ji Zhang. 2021. A subgraph-based knowledge reasoning method for collective fraud detection in E-commerce. *Neurocomputing* 461 (2021), 587–597.

[34] Shixuan Sun and Qiong Luo. 2020. In-Memory Subgraph Matching: An In-Depth Study. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 10831098. doi:10.1145/3318464.3380581

[35] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. 2020. RapidMatch: A Holistic Approach to Subgraph Query Processing. *Proc. VLDB Endow.* 14, 2 (oct 2020), 176188. doi:10.14778/3425879.3425888

[36] Yuanyuan Tian, Richard C Mceachin, Carlos Santos, David J States, and Jignesh M Patel. 2007. SAGA: a subgraph matching tool for biological graphs. *Bioinformatics* 23, 2 (2007), 232–239.

[37] J. R. Ullmann. 1976. An Algorithm for Subgraph Isomorphism. *J. ACM* 23, 1 (jan 1976), 3142. doi:10.1145/321921.321925

[38] Zhaokang Wang, Weiwei Hu, Guowang Chen, Chunfeng Yuan, Rong Gu, and Yihua Huang. 2021. Towards Efficient Distributed Subgraph Enumeration Via Backtracking-Based Framework. *IEEE Transactions on Parallel and Distributed Systems* 32, 12 (2021), 2953–2969. doi:10.1109/TPDS.2021.3076246

[39] Da Yan, Yingyi Bu, Yuanyuan Tian, and Amol Deshpande. 2017. Big graph analytics platforms. *Foundations and Trends in Databases* 7, 1-2 (2017), 1–195.

[40] Rongjian Yang, Zhijie Zhang, Weiguo Zheng, and Jeffrey Xu Yu. 2023. Fast Continuous Subgraph Matching over Streaming Graphs via Backtracking Reduction. *Proc. ACM Manag. Data* 1, 1, Article 15 (may 2023), 26 pages. doi:10.1145/3588695

[41] Zhengyi Yang, Longbin Lai, Xuemin Lin, Kongzhang Hao, and Wenjie Zhang. 2021. HUGE: An Efficient and Scalable Subgraph Enumeration System. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 20492062. doi:10.1145/3448016.3457237

[42] Pingpeng Yuan, Pu Liu, Buwen Wu, Hai Jin, Wenya Zhang, and Ling Liu. 2013. TripleBit: a fast and compact system for large scale RDF data. *Proceedings of the VLDB Endowment* 6, 7 (2013), 517–528.

[43] Yuejia Zhang, Weiguo Zheng, Zhijie Zhang, Peng Peng, and Xuecang Zhang. 2022. Hybrid Subgraph Matching Framework Powered by Sketch Tree for Distributed Systems. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 1031–1043. doi:10.1109/ICDE53745.2022.00082

[44] Zhijie Zhang, Yujie Lu, Weiguo Zheng, and Xuemin Lin. 2024. A Comprehensive Survey and Experimental Study of Subgraph Matching: Trends, Unbiasedness, and Interaction. *Proc. ACM Manag. Data* 2, 1, Article 60 (mar 2024), 29 pages. doi:10.1145/3639315

[45] Eric Zhu, Vadim Markovtsev, Aleksey Astafiev, Arham Khan, Chris Ha, Wojciech ukasiewicz, Adam Foster, Sinusoidal36, Spandan Thakur, Stefano Ortolani, Titusz, Vojtech Letal, Zac Bentley, fpug, hguhlich, long2ice, oisincar, Ron Assa, Senad Ibraimoski, Rupesh Kumar, Qin TianHuan, Michael Joseph Rosenthal, Keyur Joshi, Kevin Mann, JonR, Joe Halliwell, and Andrii Oriekhov. 2024. *ekzhu/datasketch: v1.6.5.* doi:10.5281/zenodo.11462182

[46] Gaoping Zhu, Xuemin Lin, Ke Zhu, Wenjie Zhang, and Jeffrey Xu Yu. 2012. TreeSpan: efficiently computing similarity all-matching. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 529540. doi:10.1145/2213836.2213896

[47] Xiangjian Zuo, Lixiang Li, Haipeng Peng, Shoushan Luo, and Yixian Yang. 2020. Privacy-preserving subgraph matching scheme with authentication in social networks. *IEEE Transactions on Cloud Computing* 10, 3 (2020), 2038–2049.