A Comprehensive Survey of Subgraph Matching: [Experiments & Analysis]

HAOLIN JIANG, Rutgers, The State University of New Jersey, USA
SANTOSH PANDEY*, University of South Florida, USA
HANG LIU, Rutgers, The State University of New Jersey, USA

Subgraph matching is a fundamental problem in graph analysis with a wide range of real-world applications. As subgraph matching techniques evolve, the existing mainstream *filter–order–enumeration* framework falls short in two aspects: (i) this *filter–order–enumeration* perspective overlooks an emerging line of compiler-based approaches with caching and validation-based orderings. (ii) The recent rise of complex pruning techniques has shifted the focus of core optimizations beyond filtering and enumeration. This paper advocates the need for a comprehensive survey that not only thoroughly discusses the compiler-based approaches (i.e., cache-based methods and their ordering techniques), but also reframes algorithm-level optimizations such that the role of pruning is adequately addressed.

This survey revisits 17 representative exploration-based subgraph matching methods—including both algorithm-level techniques and compiler-based ones—and establishes two optimization pillars, i.e., redundancy reduction and order generation, that can inherently summarize all these efforts. This newly established perspective permits us to systematically organize various optimization techniques and analyze how they interact with each other in the same implementation framework. Our contributions are: (i) Cache-, filter-, and prune-based strategies can remove both overlapping and different redundancies, sending our performance up to 1.81× faster than existing state-of-the-art (SOTA) settings, and (ii) heuristic and validation-based orderings, though grounded in fundamentally different design principles, often converge to similar behavior, leading to comparable performance in practice. Finally, (iii) we provide empirical guidance on when and how different strategies are most effective across diverse graph scenarios.

CCS Concepts: • **Mathematics of computing → Graph algorithms**; • **Information systems → Data mining**.

Additional Key Words and Phrases: Subgraph Matching; Graph Algorithms; Data Mining

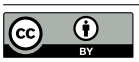
ACM Reference Format:
Haolin Jiang, Santosh Pandey, and Hang Liu. 2025. A Comprehensive Survey of Subgraph Matching: [Experiments & Analysis]. *Proc. ACM Manag. Data* 3, 6 (SIGMOD), Article 380 (December 2025), 30 pages. https://doi.org/10.1145/3771791

1 Introduction

Subgraph matching is a profound problem gaining long-standing attention in graph analysis [6, 7, 15, 26, 29, 41, 43, 52, 61, 64, 69, 73, 74, 91, 97, 98, 105]. It aims to find all subgraphs in a data graph that exactly match a given query graph. Two well-known application scenarios highlight its irreplaceable role. (i) In anti-money laundering (AML), a Europol report [28] revealed that nearly 70% of criminal

*Work done while the author was at Rutgers, The State University of New Jersey.

Authors' Contact Information: Haolin Jiang, Department of Electrical and Computer Engineering, Rutgers, The State University of New Jersey, USA, haolin.jiang@rutgers.edu; Santosh Pandey, Bellini College of Artificial Intelligence, Cybersecurity and Computing, University of South Florida, USA, spandey1@usf.edu; Hang Liu, Department of Electrical and Computer Engineering, Rutgers, The State University of New Jersey, USA, hl1097@scarletmail.rutgers.edu.



ACM 2836-6573/2025/12-ART380
https://doi.org/10.1145/3771791

networks engaged in laundering, yet only about 1.1% of illicit proceeds are recovered annually in the EU. A key challenge in AML is identifying specific structural patterns [2, 11, 12]—such as circular fund flows or layered transaction chains—in large-scale transaction graphs. Subgraph matching enables precise detection of known laundering typologies, making it a critical tool for uncovering covert financial crime in the real world. (ii) In neuroscience, a recent Nature study [59] applied subgraph matching to the complete brain network of adult Drosophila (~130,000 neurons, millions of synapses), revealing that simple chain-like patterns were underrepresented, whereas structures with mutual or recurrent links were enriched. The observed motif distribution deviates significantly from randomness, reflecting underlying interaction patterns and highlighting subgraph matching as a key tool for interpreting large-scale brain networks. Besides, subgraph matching also plays an indispensable role in security events tracking [6, 7, 48], drug discovery [58, 101, 108], chemical compound search [53, 85], and functional module identification [15, 73].

These impactful use cases have motivated a proliferation of research on subgraph matching. One of the pioneers and most influential approaches, proposed by Ullmann in 1976 [95], laid the foundation for subgraph matching by incrementally extending partial matches until all valid results are found. Since then, our database community has significantly expanded on this foundation, giving rise to two major paradigms of subgraph matching algorithms [57, 87, 88, 90, 113]: *join-based* and *exploration-based*. *Join-based* methods [1, 3, 51, 55, 56, 68, 72, 74, 78, 90, 92, 96, 104, 106, 109, 110, 112] decompose the query into smaller substructures (e.g., edges or paths) and incrementally combine their matches through join operations. In contrast, *exploration-based* methods [8, 9, 13, 16, 22, 31–33, 37, 38, 40, 44, 45, 47, 49, 50, 54, 60, 63, 77, 79, 80, 87, 88, 90, 98, 111, 113, 114] perform recursive search by extending partial matches one vertex at a time and backtracking to enumerate all results. Over the past decade, *exploration-based* methods have become dominant in subgraph matching [88, 113], driven by key optimization trends such as candidate filtering [8, 9, 36–38, 49, 107], dynamic pruning [4, 22, 36, 49, 63], and exploration order generation [14, 16, 36, 38, 47, 49, 80, 89, 99].

Two recent in-depth surveys [88, 113] have provided insightful comparisons of all major filter- and prune-based methods, categorizing their design choices under a proposed *filter–order–enumeration* perspective—a framework that has also been adopted by many open-source libraries [23, 71, 76].

However, this *filter–order–enumeration* perspective experiences two serious limitations: (i) This perspective overlooks a separate line of recent work, which builds on the same *exploration-based* paradigm but shifts the focus from algorithm-level optimizations (i.e., filter-based and prune-based methods) to compiler-level optimizations (i.e., cache-based methods) [17, 20, 27, 30, 39, 65–67, 82, 83, 94, 100, 102, 103]. Initiated by AutoMine [67] in 2019, this line of work sided with compiler-based optimizations, which generate different subgraph matching source code according to the query graph structures and patterns. This avenue of effort has also gained ground in the popular system optimization field, e.g., GPUs [34, 35, 42, 103], FPGAs/ASICs [19, 21, 25, 46, 75, 86, 93], and distributed computings [10, 18, 81, 102]. Notably, such strategies do not remove unpromising candidates before enumeration—the core idea of filtering—but instead rely on caching to avoid intersection recomputation. As a result, this cache-based optimization does not align with any steps in the *filter–order–enumeration* perspective. (ii) In addition, as highlighted by the most recent survey [113], subgraph matching algorithms have advanced from simple candidate prefiltering and enumeration to more sophisticated forms of search pruning aimed at minimizing unnecessary exploration. However, pruning is still regarded merely as part of the *enumeration* phase, without explicitly recognizing it as a distinct step within the *filter–order–enumeration*. As a result, the role of pruning is obscured, hindering a nuanced understanding of this important optimization strategy.

To address the aforementioned limitations, we establish twin pillars of optimization directions that can inherently summarize all the exploration-based methods, i.e., redundancy reduction and order generation. The former encompasses three main strategies—cache-, filter-, and prune-based

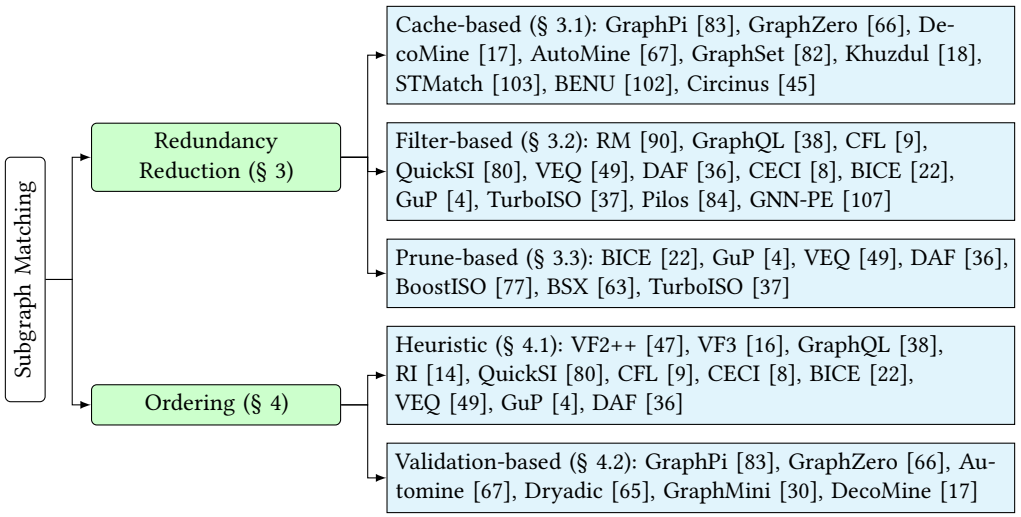


Fig. 1. Outline of major subgraph matching efforts under our proposed twin optimization pillars.

methods—each targeting redundancy in distinct ways. The latter involves two types of ordering: heuristic methods, which rely on structural heuristics, and validation-based methods, which select orders based on cost models.

Figure 1 exhibits our newly proposed survey structure for various representative subgraph matching efforts. This new structure not only thoroughly discusses the compiler-based approaches (i.e., cache-based methods and their ordering techniques), but also reframes algorithm-level optimizations such that the role of pruning is adequately addressed. In this survey, we comprehensively study a total of 17 recent exploration-based subgraph matching methods. These include representative algorithmic techniques, e.g. GraphQL [38], RI [14], VF3 [16], VF2++ [47], QuickSI [80], CFL [9], CECI [8], DAF [36], VEQ [49], Pilos [84], BICE [22], and GuP [4]—as well as compiler-based methods, such as GraphPi [83], GraphZero [66], AutoMine [67], GraphSet [82], and Decomine [17].

Building on this perspective, we make the following contributions in this paper:

- We present detailed case studies comparing representative cache-, filter-, and prune-based methods, and comprehensively analyze their combinations, overlaps, and performance implications (Section 3).
- We compare two dominant ordering strategies—heuristic and validation-based—through SOTA representatives, revealing their underlying similarities and sources of performance divergence (Section 4).
- Through a unified reimplementation and novel experimental design, we first show that (i) combining multiple redundancy reduction techniques yields up to a 1.81× speedup, (ii) heuristic and validation-based orderings perform comparably, and (iii) different strategies exhibit distinct strengths and overlaps across graph contexts, providing empirical guidance for selection (Section 5).

This survey differs from prior surveys [88, 113] in two aspects: (i) We extend beyond previous studies by incorporating the latest techniques (§3.1, §3.2, §4.2) and explicitly reframing prune-based methods as a distinct optimization category (§3.3). This leads to a more intuitive and systematic understanding under our proposed *redundancy–order* framework. We further uncover two novel insights: how cache-, filter-, and prune-based methods can overlap or complement each other (§3.4), and how validation-based ordering may perform similarly to heuristic ordering due to practical cardinality estimation (§4.3). (ii) In the experiment, our proposed subgraph matching

method outperforms the SOTA across a variety of datasets (see §5.2). And we are the first to demonstrate that heuristic and validation-based orderings achieve similar performance (see §5.6). We also introduce new profiling metrics (e.g., intersection counts and distributions) to examine how graph characteristics (e.g., density and size) influence optimization performance (§5.3) and how different reduction methods overlap with each other (§5.5). These empirical findings lead to critical insights for the subgraph matching community (§5.7).

2 Background & Motivations

2.1 Preliminaries

This paper focuses on simple graphs $g = (V, E)$, where V and E are the sets of vertices and edges. Given a vertex $u \in V$, we use $N(u) = \{u' \in V \mid (u, u') \in E\}$ to denote the neighbors of u. The degree of a vertex u is given by $d(u) = |N(u)|$.

Problem definition. We define subgraph matching (or isomorphism) as follows: Given a query graph $q = (V_q, E_q)$ and a data graph $g = (V_g, E_g)$, a subgraph isomorphism is an injective function $f : V_q \to V_g$ such that for every edge $(u, u') \in E_q$, it holds that $(f(u), f(u')) \in E_g$. Given q and g, subgraph matching aims to find all subgraphs in g that are isomorphic to q, i.e., all occurrences of q within g. For clarity, we use u to denote a vertex in the query graph and v to denote a vertex in the data graph. Of note, we use subgraph matching and isomorphism interchangeably.

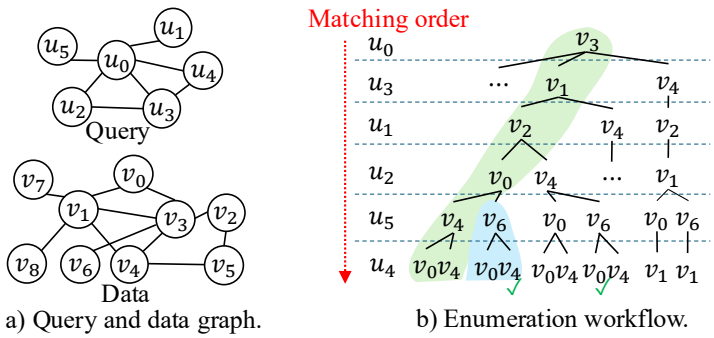


Fig. 2. An example of a naive subgraph matching process.

Scope of our work. This paper focuses on: matching a query on a static data graph (i.e., one that does not change over time) using a single-threaded, in-memory CPU implementation. This setup allows us to clearly identify and evaluate core optimizations, forming a solid foundation for tackling more specialized scenarios, such as streaming graph updates or parallel execution.

Subgraph matching workflow. Now we illustrate the subgraph matching process in its naive form (i.e., without any optimizations). The process recursively enumerates query vertices based on a given order. As shown in Figure 2, given the query and data graphs in Figure 2a and the search order $\varphi = \{u_0, u_3, u_1, u_2, u_5, u_4\}$, Figure 2b demonstrates the step-by-step enumeration workflow. For brevity, we use (u, v) to denote the mapping of u to v. We follow the branch highlighted in the green area, which begins by matching u_0 to v_3, i.e., (u_0, v_3). The neighbors of v_3, i.e., $\{v_0, v_1, v_2, v_4, v_6\}$, generate five branches, among which only v_1 and v_4 branches are shown for clarity. After mapping u_3 to v_1, the process continues with u_1, which is then mapped to v_2. When proceeding to u_2, since u_2 is adjacent to both u_0 and u_3 in the query graph, we require u_2's candidate connecting to the candidates of both u_0 and u_3, that is, v_3 and v_1. Thereby, u_2's candidate set is computed as $N(v_3) \cap N(v_1) = \{v_0, v_4\}$. After mapping v_0 for u_2, the enumeration proceeds to u_5 and explores the mapping (u_5, v_4). Once u_5 is matched, the process continues with u_4. Similar to u_2, u_4 is also adjacent

to both u_0 and u_3, resulting in the same candidate set: $N(v_3) \cap N(v_1) = \{v_0, v_4\}$. However, both v_0 and v_4 have already been matched to other query vertices, rendering both potential mappings for u_4 invalid.

Enumeration then backtracks to u_5 and explores (u_5, v_6) (blue area). The process proceeds to u_4 and considers both (u_4, v_0) and (u_4, v_4). The mapping (u_4, v_0) fails since v_0 has already been assigned to u_2, whereas (u_4, v_4) succeeds. This leads to the discovery of a valid match: $\{v_3, v_1, v_2, v_0, v_6, v_4\}$. The remaining enumeration proceeds in a similar manner until all candidates are explored.

2.2 Existing Work & Motivations

Recent surveys [88, 113] have analyzed extensive algorithmic optimizations [8, 9, 14, 16, 22, 36, 38, 47, 49, 77, 90] through the perspective of the *filter-order-enumeration*, which divides the process into three stages: (1) *filtering*, where a coarse candidate set $C(u)$ is generated using degree, label, or structural constraints; (2) *ordering*, which defines the matching sequence of query vertices φ; and (3) *enumeration*, where each candidate $v \in C(u)$ is recursively explored to find all valid embeddings. Recent works [4, 22, 36, 37, 49, 63] often optimize this last stage, i.e. *enumeration*, with pruning techniques to avoid unpromising search branches.

However, the *filter–order–enumeration* perspective fails to fully capture the core motivations behind recent advances: it neither effectively accommodates the entire avenue of compiler-based methods [18, 30, 45, 65–67, 82, 83, 102, 103, 110], nor highlights the distinct role of pruning [4, 22, 36, 49, 63]. To address this issue, we propose a new angle to compare and analyze all the related papers (see Figure 1). Particularly, we identify two optimization directions in the subgraph matching community: redundancy reduction and order generation, which are discussed below:

Direction #1: Redundancy reduction. A central aspect of subgraph matching optimization is the elimination of redundant computations. Yet, existing studies lack a complete analysis of all related papers on this objective. First, the existing survey treats filtering and pruning as separate phases—filtering as a preprocessing step and pruning as part of enumeration—despite their shared goal of reducing unnecessary computation. Second, recent compiler-based systems adopt a fundamentally different strategy that goes beyond the *filter-order-enumeration* perspective: caching, which reuses intermediate results to avoid repeated computations. Despite its proven effectiveness across single-machine [30, 45, 65–67, 82, 83], GPU-based [103, 110], and distributed environments [18, 102], caching remains largely overlooked in existing surveys. Taken together, these strategies—cache-based, filter-based, and prune-based—all aim to reduce redundancy, yet their relationships, trade-offs, and combined effects remain underexplored. This leads to two central research questions: **RQ1**: Are these strategies overlapping or complementary in practice, and can their integration lead to better performance? **RQ2**: How do different graph characteristics affect the applicability and effectiveness of these strategies?

Direction #2: Order generation. Another critical direction is order generation. Prior studies have shown that different ordering strategies can lead to performance differences of up to two orders of magnitude [83, 88, 113]. Among them [8, 9, 36–38, 47, 49, 80], RI [14]—a simple heuristic strategy that prioritizes query vertices with the highest connectivity— has been consistently recommended as the default choice in recent surveys, as it demonstrates the best overall performance across diverse datasets [88, 113]. However, the reasons behind RI's effectiveness remain unclear. In parallel, compiler-based systems have explored fundamentally different approaches to order generation. For example, validation-based ordering, i.e., the SOTA ordering strategy in the compiler-based category—proposed by GraphPi [83] and widely adopted in recent compiler-based systems [17, 30, 66, 67, 82]—exhaustively enumerates all candidate orders and selects the one with the lowest estimated cost. Intuitively, such validation-based strategies should be more accurate and are expected to outperform heuristic methods like RI. However, our experiments show that

GraphPi often performs comparably to RI (see Section 5), contradicting this intuitive expectation. This observation motivates the following question: **RQ3**: Is validation-based ordering inherently superior to heuristic-based ordering? If so, what causes its performance to degrade in practice? Conversely, can insights from GraphPi shed light on RI's effectiveness and inspire the design of better ordering strategies?

Overview. Figure 1 illustrates the landscape of optimization strategies in subgraph matching, along with representative works. Guided by the three research questions introduced earlier, our study focuses on two key directions: redundancy reduction (Section 3) and order generation (Section 4). For redundancy reduction, we first review representative techniques from the three major strategies—caching, filtering, and pruning—and then conduct a detailed analysis to address **RQ1** and **RQ2**. For order generation, we examine representative strategies through selected examples, followed by a systematic comparison to answer **RQ3**. Finally, in Section 5, we provide experimental validation to support our insights.

3 Redundancy Reduction

3.1 Cache-based Reduction

We begin by presenting the important cache-based methods—an entire class of techniques not covered in previous surveys. In a nutshell, cache-based methods reuse intersection results, significantly lowering computation overhead.

Figure 3 illustrates cache-based optimizations for the toy graphs in Figure 2. Given a predefined search order, this approach first performs (i) loop generation (Figure 3a). Subsequently, it applies (ii) control-flow optimizations like Common Subexpression Elimination (CSE) and Loop-Invariant Code Motion (LICM) to efficiently reuse intermediate results (Figure 3b). Once the optimized code is generated and executed, it naturally carries out the (iii) enumeration workflow (see Figure 3c).

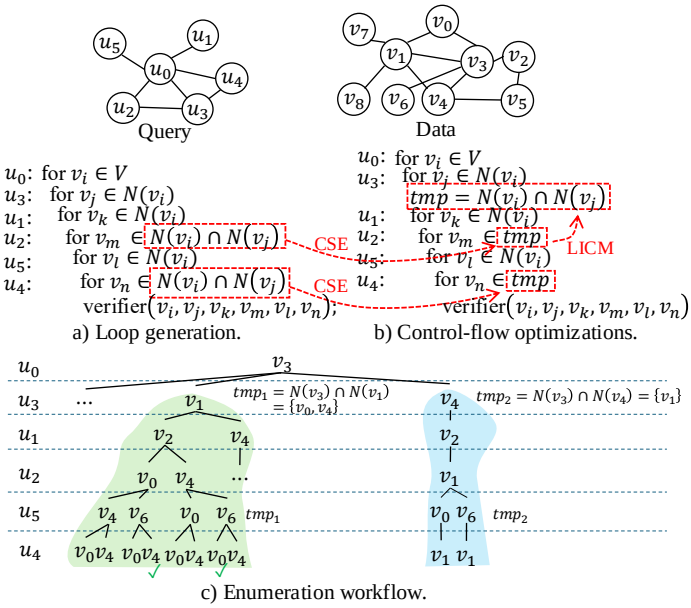


Fig. 3. Cache-based computation reduction.

(i) Loop generation constructs a nested for-loop based on the search order $\varphi = \{u_0, u_3, u_1, u_2, u_5, u_4\}$ and query graph. The generated code is shown in Figure 3a, where $N(v)$ represents the neighbors

of a data vertex v. At each level of the nested loop, the enumeration process iterates over candidate data vertices derived from the neighbors of previously matched vertices, in accordance with the query graph's connectivity constraints. When a query vertex is adjacent to multiple previously matched vertices, its candidate set is refined by intersecting their neighbor sets. For example, in Figure 3a, since u_0 and u_3 are connected to u_2 (in the query graph), and v_i and v_j are then assigned to u_0 and u_3, respectively, the candidate of u_2, i.e., v_m must be derived by $N(v_i) \cap N(v_j)$.

(ii) Control-flow optimizations enhance enumeration efficiency by reusing previously computed intersections (CSE) and shifting invariant computations from deeper to shallower levels (LICM). Automine [67] and GraphZero [66] apply only CSE to eliminate redundant intersections, while Dryadic [65], GraphPi [83], GraphMini [62], GraphSet [82], and Decomine [17] incorporate both CSE and LICM. As shown in Figure 3b, CSE detects that the intersection $N(v_i) \cap N(v_j)$ is used for both u_2 and u_4. To enable reuse instead of recomputation, it caches the result as tmp during iteration over u_1. LICM further optimizes this by recognizing that tmp remains unchanged regardless of v_k, allowing its computation to be hoisted outside the loop iterating over u_1.

(iii) Enumeration workflow is shown in Figure 3c. Assuming u_0 maps to v_3, the two neighbors of v_3, i.e., v_1 and v_4, generate two branches (shadow areas). After mapping u_3 to v_1, the cache-based method first computes $tmp_1 = N(v_3) \cap N(v_1) = \{v_0, v_4\}$, as LICM moves intersection from u_1's level to u_3's level (see Figure 3b). It then iterates over u_1. Once u_1 is mapped to v_2, the enumeration proceeds to u_2. Originally, this step triggered the computation $N(v_i) \cap N(v_j)$ (see Figure 3a) for u_2's candidates. However, under the LICM optimization, this intersection is precomputed by tmp_1, thereby directly reusing the result. After mapping u_2 to v_0 and u_5 to v_4, we are supposed to compute $N(v_i) \cap N(v_j)$ for u_4's candidates. However, since this computation has already been optimized by CSE (Figure 3b), the result of tmp_1 is directly reused here. This saves four redundant computations at u_5's level. Upon backtracking to u_1 and extending (u_1, v_4), the computation for u_2 is also omitted by tmp_1. Simply put, tmp_1 is reused five times at u_1 and u_4 levels under (u_3, v_1) (green area). Similarly, in the branch where u_3 maps to v_4, tmp_2 is computed at u_3 level and reused twice at u_5 level.

This optimization also applies to labeled graphs with minor adjustments to the CSE process, while LICM remains unchanged. For instance, if u_2 and u_4 in Figure 3 are constrained to labels A and B, respectively, then during control-flow optimizations (Figure 3b), CSE can process the shared candidate set $tmp = N(v_i) \cap N(v_j)$ in a label-aware manner: $tmp^A = N^A(v_i) \cap N^A(v_j)$ and $tmp^B = N^B(v_i) \cap N^B(v_j)$, where $N^X(v_*)$ denotes the neighbors of v_* with label X. In other words, the intersection tmp is precomputed separately for each label constraint.

3.2 Filter-based Reduction

As most filter-based techniques have already been covered by previous surveys, we use GraphQL [38] as a representative to illustrate their core ideas, followed by recent advances in this category (which are not covered by existing surveys [88, 113]).

Figure 4 illustrates filter-based optimization for the toy graphs in Figure 2. First, (i) candidate filtering is performed sequentially for each query vertex to generate candidates (Figure 4a). Next, a (ii) candidate structure is constructed to preserve connectivity, serving as an auxiliary representation of the data graph (Figure 4b). Finally, the (iii) enumeration workflow performs directly on this structure, bypassing the original data graph (Figure 4c).

(i) Candidate filtering [8, 9, 22, 36, 38, 49] generally consists of two stages. First, candidates $C(u)$ are filtered by degree and label. In Figure 4a, v_1 is a candidate of u_0 because v_1's degree is no less than u_0. Second, GraphQL's connectivity rule is applied: *if v is u's candidate, there must exist k different $v' \in N(v)$ such that each v' can be mapped with each $u' \in N(u)$, where $k = |N(u)|$*. For example, v_4 in Figure 4a is not a candidate for u_3 because we cannot find three different data vertices

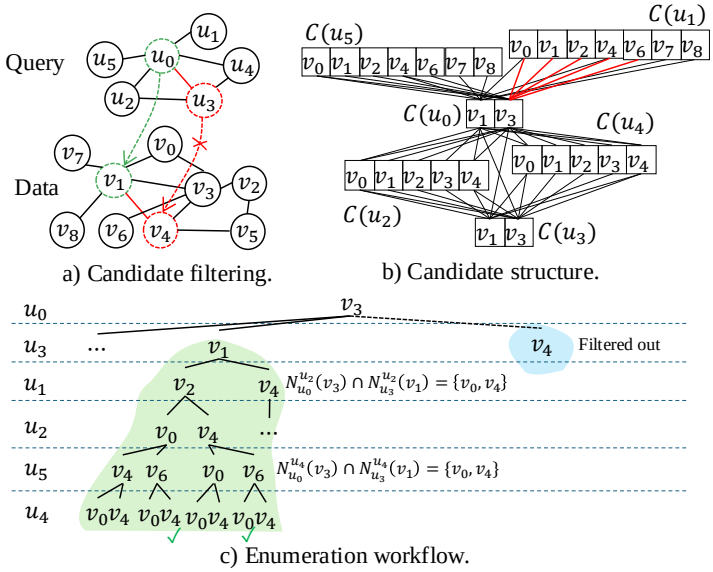


Fig. 4. Filter-based computation reduction.

from v_4's neighborhood, i.e., $N(v_4) = \{v_1, v_3, v_5\}$ that can be matched with $N(u_3) = \{u_0, u_2, u_4\}$, as v_5 cannot be matched with either u_2 or u_4. Moreover, recent advances such as Pilos [84] further enhance filtering by incorporating higher-order neighborhood information. Intuitively, if a query vertex u can be mapped to a data vertex v, the 2-hop neighborhood of u should also be a subgraph of that of v. Rather than explicitly comparing the two neighborhoods, Pilos uses the graph Laplacian spectrum and the interlacing theorem to turn this structural check into an eigenvalue comparison, efficiently removing unpromising candidates. Similarly, GNN-PE [107] shares the same motivation but achieves it through Graph Neural Networks (GNNs).

(ii) Candidate structure is constructed to store the results from the candidate filter step. For clarity, assuming we have the candidate v_j for u_j, and are expanding to u_j's neighbor u_i, we let $N^{u_i}_{u_j}(v_j)$ denote v_j's neighbors that can match u_i, that is, $N^{u_i}_{u_j}(v_j) = N(v_j) \cap C(u_i), u_i \in N(u_j)$. For example, in Figure 4b, we have $N^{u_1}_{u_0}(v_3) = N(v_3) \cap C(u_1) = \{v_0, v_1, v_2, v_4, v_6\}$, i.e., red lines in Figure 4b. In general, the filter-based methods focus on both tree and non-tree edges in the query graph. The maintenance of non-tree edges enables the set intersection for candidate computations, a key technique for efficient enumeration [88]. Of note, certain efforts only maintain the candidates for tree edges [9].

(iii) Enumeration workflow is illustrated in Figure 4c, following the order $(u_0, u_3, u_1, u_2, u_5, u_4)$. Assuming u_0 maps to v_3, the enumeration proceeds to u_3. The neighbors of v_3 that are valid candidates for u_3, i.e. $N^{u_3}_{u_0}(v_3) = \{v_1\}$, lead to the branch (u_3, v_1) (green area), and the branch (u_3, v_4) (blue area) is filtered out according to step (i). After mapping u_3 to v_1 and u_1 to v_2, $N^{u_2}_{u_0}(v_3) \cap N^{u_2}_{u_3}(v_1) = \{v_0, v_4\}$ is computed for u_2's candidates. As the enumeration goes deeper, after mapping (u_2, v_0) and (u_5, v_4), we determine u_4's candidates by $N^{u_4}_{u_0}(v_3) \cap N^{u_4}_{u_3}(v_1) = \{v_0, v_4\}$. In backtracking, when the candidates of u_5 and u_2 change, filter-based methods will repeatedly perform $N^{u_4}_{u_0}(v_3) \cap N^{u_4}_{u_3}(v_1)$ whenever it reach the last level, i.e., u_4. Further backtracking to u_1 and extending (u_1, v_4), the $N^{u_2}_{u_0}(v_3) \cap N^{u_2}_{u_3}(v_1)$ is recomputed for u_2's candidates. The remaining enumeration follows the same process. In summary, the filter-based method avoids the computation for unpromising mappings (u_3, v_4) but experiences some redundancies, i.e., $N^{u_2}_{u_0}(v_3) \cap N^{u_2}_{u_3}(v_1)$ and $N^{u_4}_{u_0}(v_3) \cap N^{u_4}_{u_3}(v_1)$ are

recomputed multiple times, highlighting the potential for further improvement through cache-based optimizations.

3.3 Prune-based Reduction

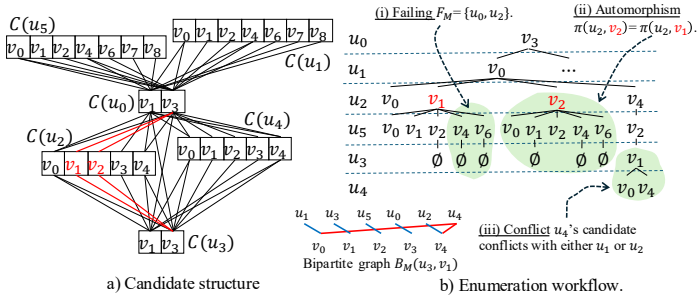


Fig. 5. Running example.

Unlike previous surveys [88, 113] that simply treat pruning as part of enumeration, we reframe it as a separate optimization technique. Further, we demystify pruning through a simple running example and systematically organize all existing techniques under BICE [22], providing an intuitive and systematic understanding.

Figure 5 illustrates prune-based reduction for the same toy graph, following the order $\varphi = \{u_0, u_1, u_2, u_5, u_3, u_4\}$. Unlike filter- and cache-based optimizations, which identify improvements before enumeration, prune-based optimization operates dynamically during the enumeration process. Pruning techniques are categorized as (i) failing pruning, which eliminates unpromising exploration based on past failures; (ii) automorphism pruning, which detects symmetries among candidates to merge similar searches; and (iii) conflict detection, which identifies structural conflicts to terminate exploration early. BICE [22] combines all three strategies for greater effectiveness. Therefore, we first analyze the pruning techniques adopted by BICE and then briefly introduce their variants [4, 36, 49]. While described in the unlabeled setting for clarity, these techniques can naturally extend to labeled graphs.

(i) Failing pruning. As shown in Figure 5b, after mapping u_5 to v_2 under the enumeration branch of (u_1, v_0) and (u_2, v_1), the enumeration proceeds to u_3 and computes its candidates as $\emptyset$, leading to a failure at u_3. Since u_3 is adjacent to u_0 and u_2 (Figure 5a), the $\emptyset$ of u_3 is attributed to the candidates of u_0 and u_2, resulting in $F_M = \{u_0, u_2\}$. When the enumeration backtracks to u_5, since u_5 has no impact on the candidates of u_3, remapping u_5 to any other vertex cannot resolve the failure at u_3. Consequently, all remaining candidates of u_5 (green area v_4 and v_6) are pruned, avoiding redundant explorations. This is achieved by checking $u_5 \notin F_M$.

BICE adopts the failing set [36], represented by an array F_M, to record query vertices contributing to failures. F_M is initially empty and dynamically updated during enumeration. To further enhance failure tracking, GuP introduces a more fine-grained variant of the failing set, called the nogood guard. Due to space constraints, we refer readers to the original work [4] for details.

(ii) Automorphism pruning. The intuition behind automorphism pruning is that data vertices with the same neighborhoods generate similar matches, leading to redundant searches. For example, as shown in Figure 5a (red lines), v_1 and v_2 in $C(u_2)$ form an equivalence class $\pi(u_2, v_1) = \pi(u_2, v_2)$, which indicates v_1 and v_2 are the same when they are the candidates for u_2, as both v_1 and v_2 connect to the same candidates v_3 in $C(u_0)$ and $C(u_3)$. These equivalence classes $\pi(u, v)$ are precomputed for all candidate pairs (u, v) before enumeration. As shown in Figure 5b, after exploring all branches

under (u_2, v_1), the process attempts to explore (u_2, v_2). Since v_1 and v_2 are equivalent under u_2, exploring v_2 would yield identical results. Therefore, further exploration under mapping (u_2, v_2) can be pruned (highlighted in green). BICE explores cell-wide verification for this pruning, while VEQ [49] uses a dynamically maintained equivalence during enumeration to enable this pruning.

(iii) Conflict detection is essentially based on the pigeonhole principle: each data vertex can only be mapped to one query vertex, and all query vertices should have their mapped data vertices. As shown at the bottom right in Figure 5b, when extending the mapping (u_3, v_1) under the (u_2, v_4) branch, BICE first constructs a bipartite graph $B_M(u_3, v_1)$ between query and data vertices (see BICE [22]). It then checks for a maximum perfect matching to ensure a one-to-one mapping (bottom left of Figure 5b). Since no such matching exists, as u_4 conflicts with either u_1 or u_2, the bottom right branch in Figure 5b is pruned.

3.4 Insight #1: Cache-, filter-, and prune-based methods remove both overlapping and different redundancies.

Considering cache-, filter-, and prune-based methods are optimizations targeting different redundancies, this section combines them to check for more effective reductions. We are the first to bring these optimizations together, offering insights beyond prior surveys.

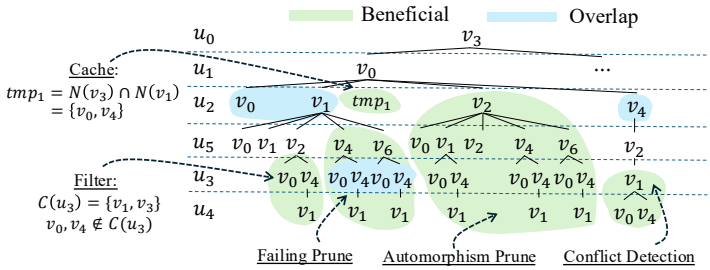


Fig. 6. Impacts of cache-, filter-, and prune-based methods.

Figure 6 illustrates the enumeration workflow of cache-based methods under search order φ with the integrations of filter-based and prune-based methods. Before the enumeration begins, this combined optimization first applies filtering to determine the valid candidates $C(u)$, shown in Figure 4b. The failing set F_M and the bipartite graph B_M are initialized as empty and dynamically maintained during the enumeration, as detailed in Figure 5b.

After initialization, the enumeration starts by mapping u_0 to v_3 and u_1 to v_0 (Figure 6). The four neighbors of v_3—v_0, v_1, v_2, and v_4—create four branches for u_2.

Cache-based optimization is applied after the enumeration branch of mapping u_2 to v_1. We first precompute the candidates for u_3 as $tmp_1 = N(v_3) \cap N(v_1) = \{v_0, v_4\}$. After mapping u_5 to v_2, the enumeration advances to u_3. $tmp_1 = \{v_0, v_4\}$ is reused for all the u_3 under this branch.

Filter-based optimization is triggered next. Still using u_3 as an example, after caching-based optimization, maps u_3 to candidates in tmp_1. Filter-based optimization permits u_4's candidates as $C(u_3) = \{v_1, v_3\}$. Since there is no intersection between tmp_1 and $C(u_3)$, a failure occurs at u_3's level, resulting in the skip of its further explorations, i.e., the green area under (u_5, v_2).

Failing pruning. The aforementioned failure is caused by u_0 and u_2, we update the failing set to $F_M = \{u_0, u_2\}$. The enumeration then backtracks to u_5. Since u_5 is irrelevant to u_3's failure ($u_5 \notin F_M = \{u_0, u_2\}$), The failing pruning eliminates the remaining exploration of u_5, i.e., green area at (u_5, v_4) and (u_5, v_6). Backtracking continues to u_2, where the remaining candidates, v_2 and v_4, are explored.

Automorphism pruning finds that v_2 is equivalent to v_1, i.e., $\pi(u_2, v_2) = \pi(u_2, v_1)$, when exploring (u_2, v_2). Since (u_2, v_1) has been explored and (u_2, v_2) would yield similar results (detailed in Section 3.3), the exploration under (u_2, v_2) is pruned, i.e., green area under (u_2, v_2). The enumeration then proceeds to (u_2, v_4).

Conflict detection identifies that no maximum perfect matching exists for the bipartite graph when extending (u_3, v_1) under the (u_2, v_4) branch (as detailed in Section 3.3). This indicates that u_4 inevitably conflicts with either u_1 or u_2. Consequently, further exploration under (u_3, v_1) is pruned, i.e., green area under (u_3, v_1).

In summary, cache-, filter-, and prune-based methods can reduce different redundancies. To implement this workflow, we integrate proactive filter and prune checks at each level of the cache-based methods, as shown in Figure 7a. For the prune check, each mapping pair (u, v) is examined and pruned if it satisfies any pruning condition, as shown in Figure 7b. Of note, such a combination also applies to labeled graphs.

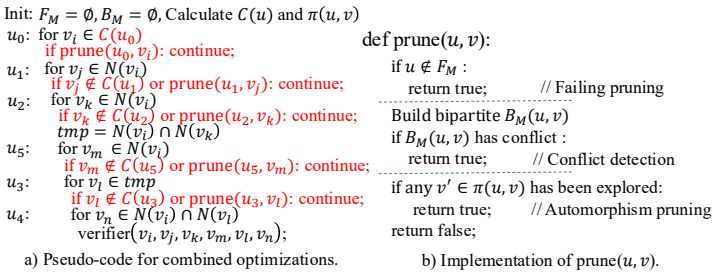


Fig. 7. Pseudo-code for combined optimization.

However, directly combining these techniques does not always improve turnaround time due to (i) checking overhead, (ii) effectiveness overlapping, and (iii) structural sensitivity.

(i) Checking overhead of filter and prune can be considerable due to nested-loop structures. The `prune()` function itself can be computationally expensive due to complex pruning strategies. When combined with a nested-loop structure, the overhead of if-condition checks is further amplified across multiple levels (Figure 6a). As the query size increases, deeper loops exacerbate this effect, causing the cumulative overhead to grow exponentially.

(ii) Effectiveness overlapping exists among different reduction optimizations. As shown in the blue area in Figure 6, at u_2's level, all candidates of u_2—v_0, v_1, v_2, v_4—are already valid members of $C(u_2) = \{v_0, v_1, v_2, v_3, v_4\}$, so no invalid candidates are eliminated, making filter checks redundant with cache-based optmizations. Similarly, in the blue area at u_3's level, the prune-based methods eliminate branches (u_5, v_4) and (u_5, v_6), rendering cache-based methods ineffective as those precomputed candidates tmp_1 are never explored. Thus, cache-, filter-, and prune-based methods may duplicate efforts, reducing their individual benefits.

(iii) Structural sensitivity influences the effectiveness of different techniques. Redundancy varies across graph structures. Intuitively, filter-based methods tend to be more effective in sparse graphs by significantly reducing the candidate space, while cache-based methods are more beneficial in dense graphs by minimizing repeated computations. Misapplying these techniques—such as using filter-based strategies on dense graphs or relying on cache-based optimizations in sparse graphs—often results in minimal gains, or even added overhead without performance improvement.

Consequently, without careful consideration, this combination can inadvertently degrade performance. We illustrate this tradeoff further in Section 5.

4 Order Generation

4.1 Heuristic Methods

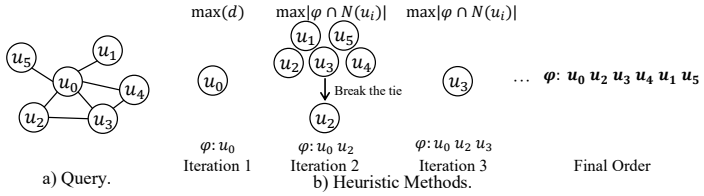


Fig. 8. Heuristic methods.

Heuristic methods [8, 9, 14, 36, 38, 47, 49, 80] follow an iterative approach, selecting vertices or edges based on predefined criteria while ensuring connectivity. As most heuristic methods have been thoroughly studied by previous surveys, we only use RI, a SOTA heuristic ordering technique [88, 113], to illustrate their ideas.

Figure 8 illustrates how RI determines the order φ using the same query graph from Figure 2. As shown in Figure 8b, RI first selects the query vertex with the highest degree, $\max d(u)$, choosing u_0 in iteration 1. It then iteratively selects the vertex most connected to the already chosen part of the query graph, i.e., $\arg\max_{u_i} |N(u_i) \cap \varphi|$. Since all unselected query vertices satisfy this condition, RI resolves the tie by selecting u_2 in iteration 2. This process continues until all query vertices are selected, resulting in the final order $\varphi = \{u_0, u_2, u_3, u_4, u_1, u_5\}$.

Methods such as VF2++[47], RI[14], QSI [80], CFL [9], GQL [38], and CECI [8] determine the order before enumeration, while more recent approaches like DAF [36] and VEQ [49] employ dynamic ordering, adapting vertex selection during enumeration. Among these, RI has been confirmed as the most SOTA method, and as other techniques have already been extensively analyzed in prior studies [88, 113], we focus solely on RI while omitting discussions on alternative heuristic methods.

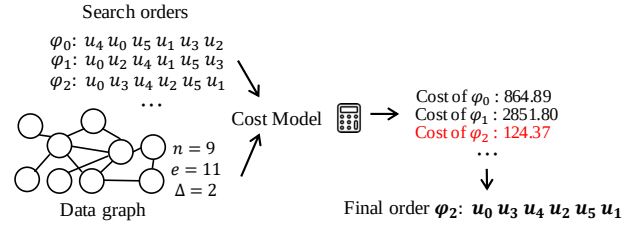
4.2 Validation-based Methods

This section discusses the validation-based ordering, which was not discussed by prior surveys [88, 113]. Specifically, validation-based methods [17, 62, 66, 67, 82, 83] explore the entire order space by generating all possible query vertex permutations and selecting the one with the lowest estimated cost. Figure 9a illustrates this process using the same query and data graph as in Figure 2. Since $cost(\varphi_2)$ is the smallest, we choose φ_2 as the order. There are various cost models; below, we explain the SOTA method from GraphPi [83].

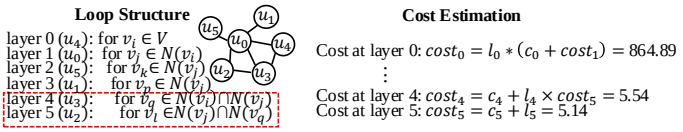
Theoretical cost modeling, i.e., $cost(\varphi)$. GraphPi derives $cost(\varphi)$ by recursively estimating per-layer costs. As shown in Figure 9b, we start from layer 5 (the deepest layer), whose cost is $cost_5(\varphi_0) = c_5 + l_5$, where c_5 accounts for computing $N(v_j) \cap N(v_q)$, estimated as $|N(v_j)| + |N(v_q)|$, and l_5 denotes the number of iterations for enumerating all candidates $v_l \in N(v_j) \cap N(v_q)$, estimated as $|N(v_j) \cap N(v_q)|$. At layer 4, the total cost $cost_4(\varphi_0)$ includes two parts: (1) computing u_3's candidates $N(v_i) \cap N(v_j)$ with cost $c_4 = |N(v_i)| + |N(v_j)|$; (2) recursively exploring layer 5 for each v_q, with per-iteration cost $cost_5(\varphi_0)$ and iteration count $l_4 = |N(v_i) \cap N(v_j)|$. Thus, $cost_4(\varphi_0) = c_4 + l_4 \times cost_5(\varphi_0)$.

Without loss of generality, we let c_i denote the intersection cost at layer i and l_i as the iteration count at layer i. We arrive at:

$$cost_i(\varphi) = \begin{cases} c_i + l_i \times cost_{i+1}(\varphi) & \text{if } 0 \le i < n_q - 1, \\ c_i + l_i & \text{if } i = n_q - 1, \end{cases} \tag{1}$$



a) Validation-based methods workflow.



b) Estimation of φ_0: $u_4\,u_0\,u_5\,u_1\,u_3\,u_2$.

Fig. 9. Validation-based methods.

where n_q is the number of query vertices, the total cost is given by the first layer cost, i.e. $cost(\varphi_0) = cost_0(\varphi_0)$.

Practical cost estimation (c_i, l_i). Since deriving the precise $cost(\varphi)$ requires knowing the # of matching elements, which is impractical, GraphPi estimates the cost as follows:

We let n and e be the numbers of vertices and edges in the data graph (e.g., $n = 7$, $e = 9$). Thereby, estimating c_5 and l_5 reduces to estimating $|N(v_j)| + |N(v_q)|$ and $|N(v_j) \cap N(v_q)|$, respectively. The cardinality of $|N(v_j)|$ and $|N(v_q)|$ can be estimated as the average degree, i.e., $\frac{e}{n}$. Therefore, $c_5 = |N(v_j)| + |N(v_q)| = \frac{2e}{n} = 5.04$. For $|N(v_j) \cap N(v_q)|$, since (v_j,v_q,v_l) form a triangle (as v_j and v_q share a common neighbour v_l and are connected), this value can be estimated by the # of triangles containing edge (v_j, v_q), i.e. $\frac{\Delta}{e}$, where Δ is the # of triangles in the data graph (e.g. $\Delta = 2$ in Figure 9a). Therefore, $l_5 = |N(v_j) \cap N(v_q)| = \frac{\Delta}{e} = 0.10$. Based on this, we compute $cost_5(\varphi_0) = c_5 + l_5 = 5.14$, shown in Figure 9b. Other layers are calculated recursively in the same way, as detailed earlier. The overall process is shown in Figure 9b. To extend this approach to more general patterns beyond triangles, see [83].

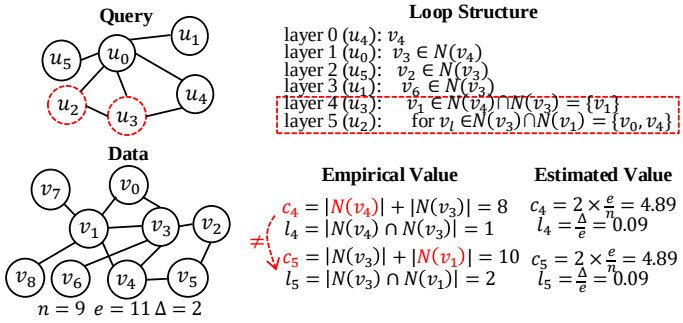
4.3 Insight #2: Validation-based methods are analogous to heuristics-based designs.

From a theoretical perspective, GraphPi and RI are fundamentally different. Particularly, RI relies solely on the query graph and fixed rules to derive the matching order. In contrast, GraphPi incorporates both the query and data graphs and exhaustively evaluates all candidate orders to select the one with the lowest estimated cost.

However, the practical estimation of GraphPi renders the validation-based method identical to RI. Of note, this manuscript is the first to observe this important phenomenon. Although RI's order ($\{u_0, u_2, u_3, u_4, u_1, u_5\}$) and GraphPi's order ($u_0, u_3, u_4, u_2, u_5, u_1\}$) yield costs as 176 (RI) and 168 (GraphPi) if we adopt the cost model in Equation 1, GraphPi's cost estimation would render them to the same cost as 124.37 (red text in Figure 9a).

The reason behind this is GraphPi's estimation design. Specifically, for an intersection involving k sets at layer i, i.e., $\bigcap_{m=1}^{k} N(v_m)$, the cost c_i is estimated as $c_i = \sum_{m=1}^{k} |N(v_m)| = k\frac{e}{n}$, which depends solely on k. Of note, $\frac{e}{n}$ is the average degree of the data graph. Further, l_i also depends only on k [83]. In short, GraphPi determines the per-layer cost solely based on the # of sets, ignoring cardinality differences, which leads to its convergence to RI.

We further illustrate this degeneration with an empirical profiling of our toy example shown in Figure 10, where the query vertices are mapped sequentially following the loop structure. After



a) Query and data graph. b) Running Example of φ_0: $u_4\, u_0\, u_5\, u_1\, u_3\, u_2$.

Fig. 10. Empirical profiling of GraphPi's cost estimation.

completing the exploration of layers 0 to 4, with u_3 mapped to v_1 via $N(v_4) \cap N(v_3)$, we focus on layer 5 to determine the candidates for u_2 by computing the intersection $N(v_3) \cap N(v_1)$. As highlighted in Figure 10b (layers 4 and 5), the empirical costs c_4 and c_5 for $N(v_4) \cap N(v_3)$ and $N(v_3) \cap N(v_1)$ differ — $c_4 = |N(v_4)| + |N(v_3)| = 8$ and $c_5 = |N(v_3)| + |N(v_1)| = 10$. However, both are assigned the same estimated cost (4.89), as they are uniformly treated as two-set intersections of the form $N(v_3) \cap N(v_*)$, ignoring the difference between $N(v_4)$ and $N(v_1)$. A similar issue is observed for l_4 and l_5, whose empirical values are 1 and 2, yet both are estimated as 0.09.

As a result of overlooking neighborhood-specific differences, GraphPi simply favors orders with more intersections at shallower layers, resembling RI's "most connected vertex" strategy. Therefore, despite theoretical differences, GraphPi and RI converge to similar orderings and exhibit comparable performance. Of note, such near-equivalence also persists in labeled settings, as their orderings are label-agnostic. We further validate this observation through experiments in Section 5. This insight also suggests that refining GraphPi's per-layer cost estimation to capture fine-grained differences between intersections could further improve its performance.

5 Evaluations

This section addresses the following questions that are not covered by previous surveys [88, 113]:

- **RQ1:** Are these optimizations overlapping or complementary? Can their integration lead to better performance?
- **RQ2:** How do different graph characteristics affect the effectiveness of cache-, filter-, and prune-based methods?
- **RQ3:** Is validation-based ordering inherently superior to heuristic-based ones? If so, what factors cause its performance to degrade in practice? Conversely, can insights from GraphPi help explain RI's effectiveness and guide the design of better ordering strategies?

To answer **RQ1**, we first compare our proposed combination against 13 existing SOTA methods, demonstrating its superior performance with up to a 1.81× speedup over the best-performing baseline (Section 5.2). Then, we report the combined effectiveness and profile the overlap between cache-based, filter-based, and prune-based methods, and assess their combined effectiveness and interaction overhead (Section 5.5). For **RQ2**, we study each type of redundancy reduction method individually (Section 5.3) and evaluate the impact of different graph characteristics on their effectiveness. For **RQ3**, we conduct a comprehensive comparison of ten ordering strategies (Section 5.6), providing experimental evidence that supports and extends our earlier analysis.

5.1 Experimental Setup

Studied work. We compare redundancy reduction and order generation strategies across 14 representative algorithms, including Pilos [84], RI [14], VF3 [16], VF2++ [47], QuickSI [80], GraphQL [38], CFL [9], CECI [8], DAF [36], VEQ [49], BICE [22], GuP [4], GraphPi [83], and GraphZero [66]. Previous surveys [88, 113] provide a unified evaluation framework covering the first nine algorithms, and we adopt their implementations for our comparison. Based on their implementation, we additionally include Pilos, BICE, GuP (reimplemented in C++ from its Rust version), GraphPi, and GraphZero, using source code obtained from the original authors for fair evaluations. Of note, Pilos is evaluated with its default parameters [84].
Environment. All algorithms are compared in C++ and compiled with -O3 optimization. All experiments are evaluated on a Linux server with NVIDIA Grace CPU Superchip and 237 GB of memory. The source code will be available at https://github.com/Orrinn/SubgraphMatching.
Datasets. Table 1 describes the 10 graph datasets we used for evaluation, which span six domains, aligning with prior studies [9, 38, 47, 66, 67, 83, 87, 88, 113]. Following established methodologies [5, 24, 70], we assess dataset degree distributions using Maximum Likelihood Estimation (MLE) and the Kolmogorov–Smirnov (K–S) test to estimate the power-law exponent (γ) and its statistical significance (p). The final classification of each dataset is in the "Distribution" column of Table 1.

Table 1. Graph datasets specifications.

Dataset	Abbr.	|V|	|E|	d_G	Distribution
Figeys	fg	2,239	6,432	5.7	Exponential
HPRD	hp	9,460	34,998	7.4	Power-law
WordNet	wd	109,108	134,076	2.5	
Twitch	tw	168,144	6,797,557	80.9	
Youtube	yt	1,134,890	2,987,624	5.3	
US Patient	up	3,774,768	16,518,947	8.8	
LiveJournal	lj	4,846,609	42,851,237	17.7	
Citeseer	cs	3,279	4,552	2.8	Uniform
Wikivote	wk	7,117	100,763	28.3	
DBLP	db	317,080	1,049,866	6.6	

Query graphs. Following prior work [8, 22, 36, 49, 87, 88, 113], we generate connected query graphs through random walks. Specifically, we start from a randomly selected node in the base graph and perform a random walk until a specified number of unique vertices have been visited. The induced subgraph over these vertices is then extracted as the query graph. We vary the query size from 8 to 256, represented by Q_8, Q_{12}, Q_{16}, Q_{32}, Q_{64}, Q_{128}, and Q_{256}. For each size, we sample 200 query graphs. To avoid sampling structurally identical queries, we apply canonical labeling during generation. In addition, we include a set of real-world workload queries, as shown in Figure 11. This workload is built by (i) extracting non-tree-structured queries (excluding chains and stars, which can be solved in polynomial time) from the *LSQB* and *LC-QuAD* SPARQL datasets, and (ii) complementing them with representative queries commonly used in practical graph mining systems [17, 65, 67, 83, 103] and join-centric frameworks [1, 68, 90]. Notably, to ensure fair comparison across different data graphs, we use the same set of query graphs for all data graphs. Unless otherwise specified, we default to Q_{16}.
Evaluation metrics. We use two primary metrics in our evaluation. *Overall runtime* serves as the main indicator of enumeration performance, while *intersection count* is used as a profiling metric

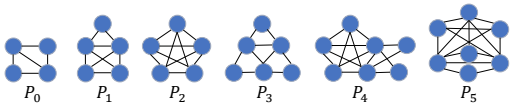


Fig. 11. Real-world query workload.

to reveal the internal computational cost. Rather than relying solely on average values, we report the average, median, minimum, and maximum values to highlight overall trends under different scenarios. To ensure timely completion of experiments, we follow [9, 22, 36, 49, 88], which either terminate each query after retrieving 10^5 embeddings or abort those that exceed 5 minutes. As [113] notes that the relative performance of different methods may vary under different termination thresholds, we further verify their turnaround time up to 10^{10} embeddings.

5.2 Overall Performance

Figure 12 compares the overall performance of 14 original methods (RI, DAF, VF3, GuP, BICE, GQL, VF2++, GraphPi, GraphZero, CECI, CFL, QSI, VEQ, Pilos) and our newly proposed combinations (using DAF's failing set, GQL's filter, and cache-based optimization together) on the Q_{16} query for output threshold 10^5. As suggested in [88, 113], we adopt RI ordering for DAF. Results for VEQ and Pilos are omitted on the *twitch*, *youtube*, *livejournal*, and *uspatent* because both run out of memory, with `Pilos` failing specifically as it relies on VEQ's pruning method as a core component [84].

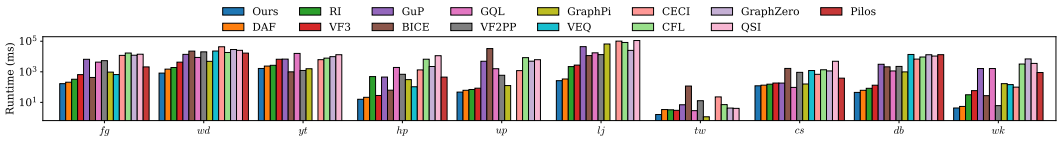


Fig. 12. Overall performance.

Figure 12 shows that our proposed combination method delivers the best overall performance on 8 out of 10 data graphs. Notably, on the *wordnet* dataset, it achieves an average runtime of 824.3ms, representing a 1.81× speedup over the fastest existing method (DAF, 1496.9ms). We also observe a few exceptions. On the *youtube* dataset, BICE outperforms our method because its bipartite matching prunes more effectively than DAF's strategy within our combination; and on the *twitch* dataset, the purely cache-based GraphPi achieves the best performance, likely because caching is already highly effective on dense graph structures, leaving little room for pruning to improve while still incurring additional overhead.

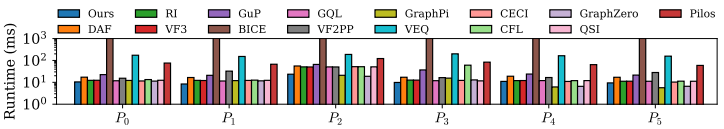


Fig. 13. Overall performance for real-world query workload.

Figure 13 shows a similar trend on real-world workloads, with our method achieving SOTA performance on P_0–P_3. For more complex patterns (i.e., P_4 and P_5), GraphPi tends to dominate, in part, because it achieves more benefits in caching intermediate intersection results in complex query graphs. Pilos exhibits performance similar to VEQ because it adopts VEQ's pruning method.

5.3 Cache- vs Filter- vs Prune-based Reduction

In this section, we evaluate representative techniques for the three key redundancy reduction strategies adopted across the 14 algorithms, aiming to derive strategy-level insights that have

not been explored in prior studies. To ensure a fair comparison, we adopt RI ordering throughout this section and report overall performance using a fixed output threshold of 10^5. For cache-based methods, we apply both LICM and CSE optimizations, denoted as `Cache`. For filter-based methods, we compare five filters: `fPilos`, `fGQL`, `fCFL`, `fCECI`, and `fDAF`. For prune-based methods, we include two failing pruning methods (`FP:DAF`, `FP:GuP`), one conflict detection method (`CD:BICE`), and two automorphism pruning methods (`AP:BICE`, `AP:VEQ`). All methods are compared against a non-optimized baseline. We run Q_{16} on all datasets (Figures 14 and 15), and Q_8 to Q_{256} on *dblp* to assess the impact of query size (Figures 16 and 17). `AP:VEQ` runs out of memory on *youtube*, *twitch*, *uspatent*, and *livejournal*, so we omit their results in these cases.

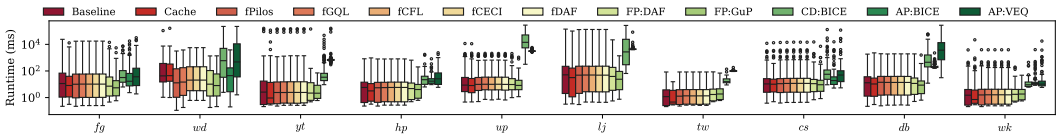


Fig. 14. Overall performance of different reduction optimizations on Q_{16}.

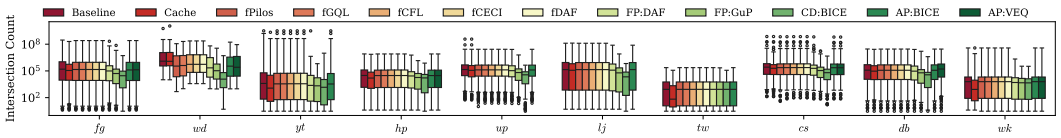


Fig. 15. Profiling of different reduction optimizations on Q_{16}.

Strategy-Level Comparison. We begin by comparing the representative technique from each category (`Cache`, `fGQL`, `fPilos`, and `FP:GuP`) to derive high-level insights into cache-, filter-, and prune-based strategies. We then zoom into each strategy in detail to extract the intra-strategy insights. Among prune-based methods, `FP:GuP` is selected because it delivers the strongest performance on Q_{16} across all datasets. For filter-based methods, we report both `fPilos` and `fGQL`: `fPilos` can yield better performance on sparse graphs, but typically incurs higher preprocessing overhead, whereas `fGQL` achieves comparable effectiveness on the other datasets with lower overhead. We include both for a clear view of this trade-off.

Table 2. Strategy ranking across graph density levels on Q_{16}.

Strategy	Sparse ($d_G \leq 3$)	Medium	Dense ($d_G \geq 20$)
Cache-based	3rd	2nd	1st
Filter-based	2nd	3rd	3rd
Prune-based	1st	1st	2nd

(i) Impact of data graph density. Table 2 shows that for Q_{16}, prune-based performs best on sparse and medium graphs, while cache-based dominates on dense graphs.

As shown in Figure 14, on dense datasets such as *twitch* (avg. degree ~80.9) and *wikivote* (~28.3), `Cache` consistently outperforms the other two strategies, reducing median runtime by 2.54× and 2.12× respectively. In contrast, `fGQL`, and `FP:GuP` yield only marginal improvements (~1.1×) over the baseline. As graph density decreases—e.g., on *dblp* (avg. degree ~6.6), *hprd* (~7.4), *uspatent* (~8.8) and *livejournal* (~17.7) —cache-based methods become less effective, while the prune-based method gains ground. On these datasets, `FP:GuP` achieves slightly better speedups than `Cache`: 1.51× vs. 1.22× on *dblp*, 1.81× vs. 1.42× on *hprd*, 1.28× vs. 1.23× on *uspatent*, and 2.56× vs. 1.42×

on *livejournal*. Filter-based method, in contrast, retains limited benefits on both datasets, with speedups around 1.05×. This trend flips over on extremely sparse graphs such as *wordnet* (~2.5), where filtering and pruning become highly effective and caching nearly loses impact. `fGQL` and `FP:GuP` achieve substantial speedups of 2.91× and 5.33×, respectively, while `Cache` offers limited benefit (1.03×), with its runtime distribution falling back close to the baseline. `fPilos` follows a similar trend, showing the largest speedup on the extremely sparse dataset *wordnet* (3.32× over baseline) and performance comparable to `fGQL` on the other datasets.

Table 3. Strategy ranking across query sizes on *dblp*.

Strategy	Small ($\|V_G\| \leq 12$)	Large ($\|V_G\| > 12$)
Cache-based	1st	2nd
Filter-based	2nd	3rd
Prune-based	3rd	1st

Our intersection profiling results (Figure 15) shed light on the underlying cause of this trend. `Cache` reduces intersections by 12.64× and 7.41× on *twitch* and *wikivote* respectively, but drops to 1.28×, 1.96×, 1.19×, and 1.48× on *dblp*, *hprd*, *uspatent*, and *livejournal*; and becomes negligible on *wordnet* with over 99% intersections remaining. Meanwhile, `FP:GuP` becomes increasingly effective, reducing intersections by 1.01× and 1.33× on *twitch* and *wikivote*; 1.98×, 1.71×, 2.21×, and 4.18× on *dblp*, *hprd*, *uspatent*, and *livejournal*; and 11.78× on *wordnet*. `fGQL`, by contrast, remains mostly ineffective on non-sparse datasets, with reductions near 1.05×, but becomes highly effective on *wordnet*, achieving a 3.06× reduction. Compared with `fGQL`, `fPilos` shows clear improvements on *wordnet* (1.16× over `fGQL`, 3.55× over Baseline), while achieving comparable intersection counts on the other datasets.

(ii) Impact of query graph size. Table 3 shows that on *dblp*, caching is most effective for small queries, while pruning gain impact as query size increases.

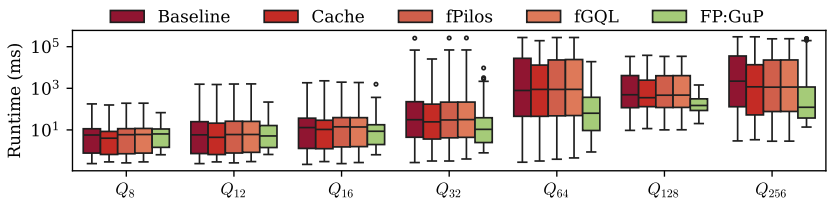


Fig. 16. Overall performance of different query sizes *dblp*.

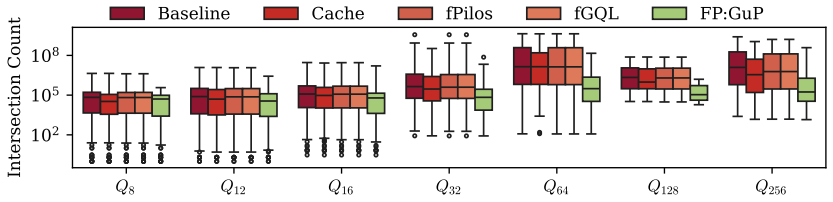


Fig. 17. Profiling of different query sizes *dblp*.

Figures 16 and 17 show that for small queries (Q_8 and Q_{12}), `Cache` provides the largest boost, achieving up to 1.4× speedups and 2.1× intersection reductions for Q_8. In contrast, `fGQL` yields only marginal improvements (1.01×), and `FP:GuP` even degrades performance on Q_8 by 1.1× due to its

overhead, despite a 1.3× reduction in intersections. On larger queries (e.g., Q_{16} and Q_{32}), however, FP:GuP delivers a substantial 1.51× speedup for Q_{16}, as unpromising search begins to dominate the sources of redundancy. By contrast, both fGQL and Cache provide lower speedups—around 1.05× and 1.3×, respectively. For extremely large queries (Q_{64}–Q_{256}), we observe the same trend already evident at Q_{16} but with more pronounced differences: FP:GuP achieves substantial speedups (up to 18.4× on Q_{256}) and dominates, while Cache provides only moderate benefits (1.9× for Q_{256}) and lags behind. For completeness, we also report fPilos's results. Across Q_8–Q_{256}, its runtime and total intersection counts are broadly comparable to those of fGQL.

Impact of different filter-based methods. As shown in Figure 14, fPilos, fGQL, fCFL, fCECI, and fDAF exhibit similar performance on non-sparse datasets such as *twitch*, *youtube*, *uspatent*, and *livejournal*. On sparse graphs like *wordnet*, however, fPilos and fGQL achieve significantly greater intersection reductions (from 1.23M to 0.35M and 0.41M, whereas the other filters reduce intersections to around 0.55M (Figure 15)).

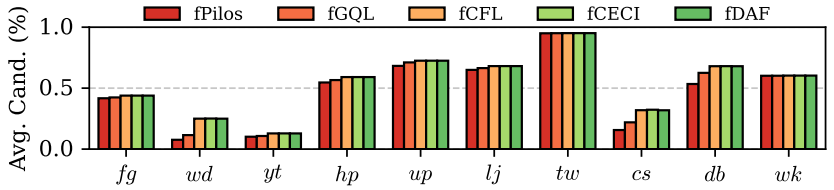


Fig. 18. Average candidates ratio (%) on Q_{16}.

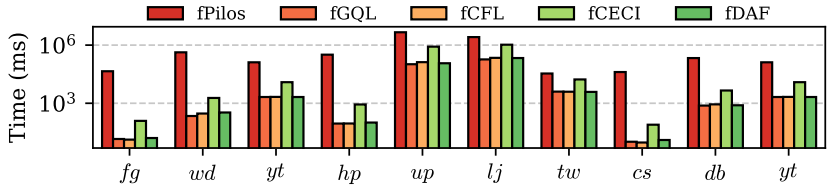


Fig. 19. Filtering overhead on Q_{16}.

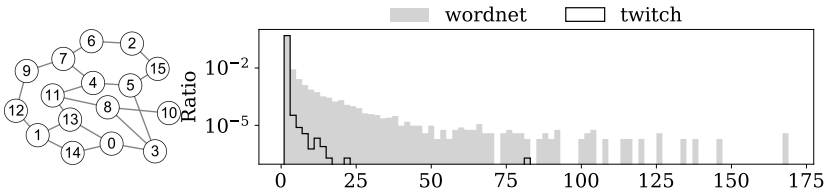
To unveil the performance gap among filters, we follow [49, 88, 113] and profile their filtering effectiveness using the average number of candidates per query vertex, calculated as Avg. Candidates $= \frac{\sum_{u \in V_q} |C(u)|}{|V_q|}$. For each dataset, we compute the mean of this value across all queries and normalize it by the size of the data graph. We also report the average filtering overhead to better illustrate the trade-off between filtering effectiveness and time budget. As shown in Figure 18, fCFL, fCECI, and fDAF produce similarly sized candidate sets across all datasets, which explains their comparable filtering effectiveness (Figure 14 and 15). In contrast, fGQL significantly outperforms the other methods on extremely sparse datasets such as *wordnet* (avg. degree ~2.5) and *citeseer* (~2.8), achieving much lower average candidate ratios—11% on *wordnet* and 21% on *citeseer*, compared to ~25% and ~32% for the other filters. fPilos yields better candidate reductions on *wordnet* (7%), *citeseer* (15%), and *dblp* (53%), while on the other datasets its performance is comparable to fGQL. These gains stem from fPilos's rigorous filtering—multiple rules applied iteratively until no further candidates remain [84]—in contrast to fGQL's single one-pass rule [88]. This exhaustive refinement increases preprocessing time. As shown in Figure 19, on *uspatent* and *livejournal* the average per-query time is 4599s (~1.3 hours) and 2592s (~43 minutes) for fPilos, versus 103s and 182s for fGQL (about 14× higher). Overall, this reflects a trade-off: on sparse graphs—when

overhead is acceptable (e.g., small queries and data graphs)—fPilos can provide stronger filtering; otherwise, fPilos and fGQL are broadly comparable.

In addition, the profiling further substantiates the observed dependence of filtering effectiveness on graph sparsity. This trend is evident on dense graphs such as *twitch* (~80.9), where even the best-performing filter, fGQL, retains 95% of the vertices. Separately, on *youtube*, although fGQL achieves a relatively low candidate ratio (~10%), the sheer size of the graph (~1.13M) still results in a large candidate set and consequently a long runtime.

Impact of different prune-based methods. Unlike cache- and filter-based optimizations, prune-based techniques can sometimes hurt performance. As shown in Figure 14, on *dblp*, FP:DAF and FP:GuP reduce runtime by 1.21× and 1.51×, respectively. In contrast, CD:BICE, AP:BICE, and AP:VEQ significantly increase it—up to 15.06×. However, these prune-based methods still achieve strong pruning effectiveness in terms of intersection count. As shown in Figure 15, FP:DAF, FP:GuP, and CD:BICE reduce intersections by 1.29×, 1.98×, and 3.31×, respectively.

In contrast, automorphism-based methods such as AP:BICE and AP:VEQ yield negligible reductions (~3%). Thus, while CD:BICE achieves the largest pruning effect, it fails to improve performance — highlighting a key trade-off of prune-based reductions: intersection reduction alone does not guarantee overall speedup.



a) Q_{16} Query. b) Size of equivalent vertex classes.

Fig. 20. Automorphism pruning profiling.

Now we unveil the overhead of prune-based methods: For CD:BICE, the cost of maintaining and matching bipartite graphs dominates runtime. Especially on datasets like *uspatent*, *livejournal*, *youtube*, and *dblp* (shown in Figure 15), where dramatic intersection count differences (10^2 – 10^6) do not lead to a similar degree of variations in runtime (e.g., ~3.1s on *uspatent* and ~4.2s on *livejournal*), indicating that intersection count is no longer the performance indicator.

Even worse, AP:BICE and AP:VEQ incur high overhead from index maintenance but offer little benefit. To explain this ineffectiveness, we profile a Q_{16} query (Figure 20a) and analyze the distribution of symmetric vertices on the *wordnet* and *twitch* datasets. As shown in Figure 20, over 99% of vertices belong to singleton automorphism classes (i.e., no symmetric counterparts), leaving limited redundancy to remove, unveiling the limited benefit of automorphism pruning in AP:BICE and AP:VEQ.

5.4 Evaluations on Labeled Graphs

Following prior surveys [88, 113], we generate labeled versions of *wordnet*, *twitch*, *citeseer*, and *dblp*, as well as query graphs, by randomly assigning one of five labels ($|L| = 5$) to each vertex. Specifically, we evaluate three representative methods—Cache, fGQL, and FP:GuP—by running Q_{16} across different data graphs, as well as varying query sizes on the *dblp* dataset.

As shown in Figure 21, the relative performance of different techniques is broadly consistent with the unlabeled settings (Table 2), though some minor shifts are observed. In dense graphs such as *twitch*, Cache remains the most effective (1.6× speedup), whereas fGQL continues to perform best on sparse graphs such as *wordnet* (8.2×). A similar pattern holds for most query sizes: FP:GuP

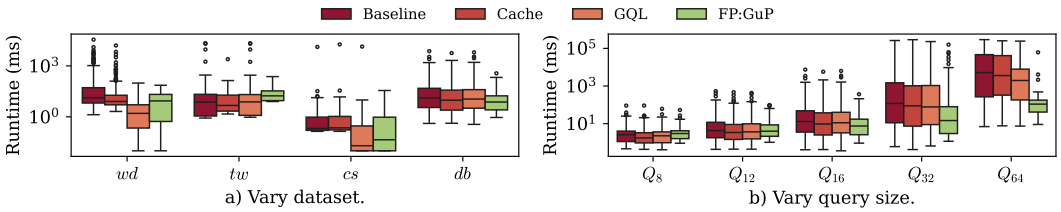


Fig. 21. Performance on labeled settings.

dominates medium-to-large queries (Q_{16}–Q_{64}) but introduces overhead on small queries (Q_8–Q_{12}), where Cache is more effective. However, as query sizes grow larger, fGQL becomes stronger (e.g., Q_{64}, 3.1× speedup) as label constraints help eliminate invalid candidates more effectively. Conversely, Cache shows slightly reduced gain (1.4×), likely due to the added complexity of CSE mechanisms under label-aware execution (see Section 3.1).

5.5 Filter-based and Prune-based Methods Over Cache-based Methods

This section uniquely evaluates the overlap and combined effectiveness of cache-, filter-, and prune-based methods. We show that these techniques exhibit varying degrees of overlap: cache-based methods overlap more with filter-based ones than with prune-based ones. Furthermore, combining all three generally improves performance. However, in certain cases, such as extremely dense graphs, due to extraordinarily high overlapped redundancy reduction and nontrivial extra overheads, the combined effort could experience performance degradation compared to an individual optimization.

Adding filter-based methods atop cache-based methods. To quantify the overlap between cache-based and filter-based methods, we analyze how many iterations—executed under cache-based optimization—could have already been eliminated by the GQL filter for the same Q_{16} query in Figure 20a on the *wordnet* dataset. The search order, generated by RI, is {u_0, u_3, u_5, u_4, u_{11}, u_8, u_{13}, u_1, u_{14}, u_7, u_9, u_{12}, u_6, u_2, u_{15}, u_{10}}, and is visualized in Figure 22a using directed edges on the query graph, with the starting vertex highlighted in yellow. The profiling results are shown in Figure 22b.

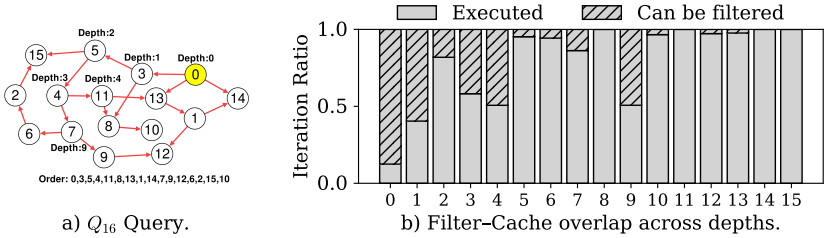


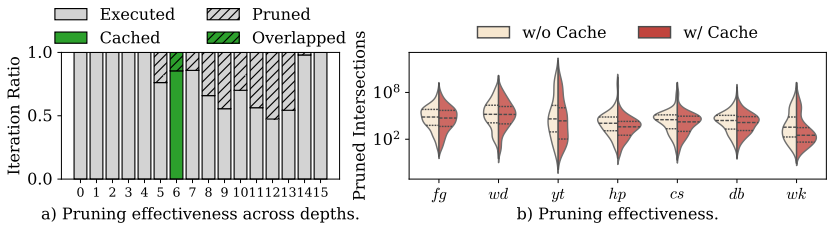
Fig. 22. Overlap between cache-based and filter-based methods.

The filter-based methods can further help remove redundancy in cache-based methods for two cases: (i) when the query vertex appears at a shallow depth in the search order, or (ii) when a vertex connects to only a single previously matched vertex. Otherwise, its additional refinement over cache-based methods is limited.

As shown in Figure 22a, the query vertices at depths 0–4 (i.e., u_0, u_3, u_5, u_4, and u_{11}) satisfy the conditions (i) and the vertex at depth 9 (i.e., u_7) follows conditions (ii). As a result, fGQL eliminates 18%–87% of additional invalid iterations over the cache-based methods at those depths. In contrast, at depths 5 and 12, where u_8 (adjacent to both u_3 and u_{11}) is highly connected and u_6 appears deep (depth 12) in the search order, filter-based methods remove only 3%–4% of additional invalid iterations. Moreover, at depths 14 and 15, where iteration counts reach 202M and 1.678B, respectively, fGQL provides no additional benefit (0%), suggesting that caching alone is sufficient.

These profiling results collectively suggest that adding filter-based methods on top of cache-based optimization does not always lead to performance gains, as the benefits vary across cases and the added overhead can be nontrivial.

Adding prune-based methods atop cache-based methods. To illustrate the additional benefits and overlap brought by prune-based methods beyond cache-based ones, we break down the executed iterations at each depth under cache-based optimization, using the same query graph and order as in Figure 22a. Figure 23a shows the percentage of iterations that can be precomputed (Cached), can be pruned by FP:DAF (Pruned), and their overlap. FP:DAF is most effective at the middle depths (5–13), eliminating 14%–52% of intersections. In this example, the cache-based method only precomputes the intersection at depth 6, 14% of which turn out to be unnecessary due to subsequent pruning. Nevertheless, caching and pruning tend to target different forms of redundancy, resulting in relatively limited overlap between them (Figure 23a).



a) Pruning effectiveness across depths. b) Pruning effectiveness.

Fig. 23. Overlap between prune- and cache-based methods.

We further evaluate such overlap by measuring the number of intersections pruned by FP:DAF with and without cache-based optimization across multiple datasets (Figure 23b) for all Q_{16} queries. As shown, the overlap between prune-based and cache-based methods is generally limited on non-dense datasets. For instance, on *wordnet* (avg. degree $\sim$ 2.5), caching reduces the number of intersections pruned by only 4%. However, the overlap increases with graph density. A notable example is *wikivote* ($\sim$ 28.3), where the median number of intersections pruned by FP:DAF drops by 91% when caching is applied. This is likely because denser graphs lead to more precomputation (as discussed in Section 5.3), increasing the chance of overlap with pruning. We further evaluate different combinations of the three redundancy-reduction methods in terms of intersection counts, complementing the runtime results reported in Section 5.2.

5.6 Ordering

In this section, we evaluate ordering strategies by first presenting an overall comparison of 10 methods under representative redundancy optimization settings (Cache, fGQL, and FP:DAF), highlighting consistent performance trends across configurations. We then focus on two representative methods, RI and GraphPi, to examine their runtime distributions and behaviors under varying output thresholds—an aspect not explored in previous studies. We confirm that (i) state-of-the-art heuristics, i.e., RI and VF3, and validation-based ordering strategies, i.e., GraphPi, generally deliver comparable performance in practice despite their fundamentally different designs. (ii) GraphPi can suffer from poor worst-case performance, as its tie-breaking rule may be misled by inaccurate early-stage heuristics.

General trends of convergent performance. Figure 24 shows that RI, VF3, and GraphPi consistently rank among the top ordering strategies across datasets and optimization settings. For instance, on the *figeys* dataset, the median runtimes of RI, VF3, and GraphPi are comparable across all three optimizations: 7.93ms, 8.57ms, and 7.75ms under cache-based (Figure 24a); 10.11ms, 10.51ms, and 10.31ms under filter-based (Figure 24b); and 7.33ms, 9.39ms, and 7.36ms under prune-based

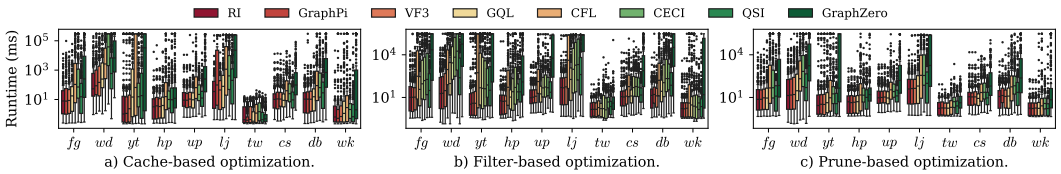


Fig. 24. Comparision of ordering.

optimization (Figure 24c). In addition to our prior claim, RI and GraphPi perform similarly as the best options, VF3 joins the best-performing group because VF3 adopts RI's max-connectivity-first ordering with a minor difference in tie-breaking heuristics. This pattern generally holds across datasets, except on *livejournal* and *twitch* under cache-based methods, where GraphPi shows greater variation.

To further examine RI and GraphPi, we visualize their runtime distributions across 200 queries (Figure 25), which perform similarly on most datasets (e.g., *uspatent*, *hprd*), reinforcing their empirical similarity. This alignment also holds under varying output thresholds (Figure 26), with runtime trends remaining closely matched as the answer size grows from 10^4 to 10^{10}—with the exception of *wordnet*, and of *figeys* and *livejournal* when the output exceeds 10^9.

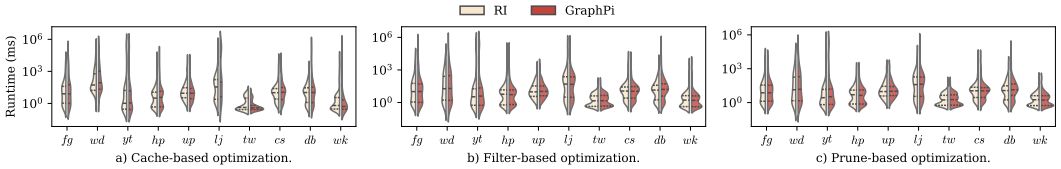


Fig. 25. Detailed comparison of RI and GraphPi.

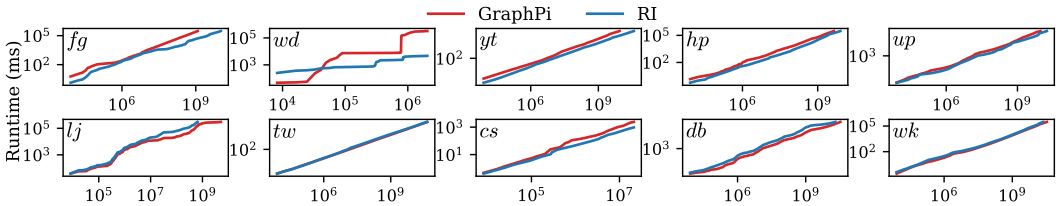


Fig. 26. Varying the output limits.

Divergent cases in specific scenarios. While GraphPi performs comparably to RI in most cases, it occasionally suffers from much worse worst-case runtimes. For example, on the *wordnet* dataset (Figure 26), its runtime increases sharply after answer size exceeds 10^5. Similar long-tail behavior appears in the runtime distributions across datasets (Figure 25), especially under cache-based optimization on *dblp* (299,195.4ms vs. 2,262.9ms) and *livejournal* (242,747.8ms vs. 96,870.1ms), with consistent patterns observed under other optimizations on *dblp* and *wordnet*.

Such poor worst-case performance stems from GraphPi's core-first tie-breaking strategy, which—similar to CFL [9]—prioritizes matching the query's core region first. As noted in [36], such strategies can be misleading by overlooking the benefits of matching non-core vertices early. This weakness is also reflected in CFL's consistently high worst-case runtimes (Figure 24), highlighting the inherent instability of core-first designs and the importance of careful tie-breaking in order generation.

5.7 Emperical Suggestions

Based on our experimental results, we present the empirical recommendations summarized in Figure 27: (i) Redundancy reduction: We recommend combining cache-, filter-, and prune-based

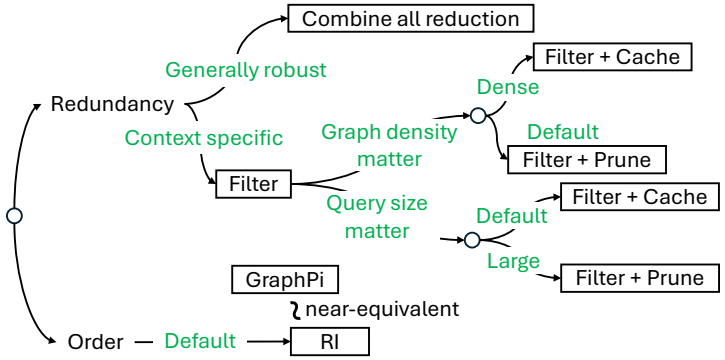


Fig. 27. Empirical suggestions for subgraph matching.

methods by default to ensure robust performance across diverse scenarios. If the characteristics of the data and/or query graphs are known, one can pursue the following options: Filter-based methods are the essential building block for eliminating invalid candidates and should generally be applied first. On top of this, cache-based methods are most effective for dense data graphs, while prune-based methods are preferable in default cases. For query graphs, cache-based methods work best on small queries, whereas prune-based methods are more suitable for large ones. (ii) Order generation: We recommend using RI by default. Although GraphPi may appear more promising, our theoretical and empirical analyses show that the two are comparable in effectiveness. Given GraphPi's higher overhead, RI remains the more practical choice.

6 Conclusion

This paper presents a comprehensive study of recent exploration-based subgraph matching methods, encompassing both algorithms and emerging compiler-based systems. We identify two fundamental optimization dimensions—redundancy reduction and order generation—and conduct in-depth case analyses to compare representative techniques. Our unified implementation enables fair evaluation across methods, revealing new insights into their effectiveness, overlap, and composability. Notably, we show that combining multiple redundancy reduction techniques can yield up to 1.81× speedup, and that heuristic- and validation-based ordering strategies, despite their differences, often achieve comparable performance. Our findings provide empirical guidance on design choices across diverse graph scenarios, enabling more informed and effective subgraph matching system development.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback and valuable suggestions. This work was supported in part by the NSF CAREER Award No. 2326141, and NSF Awards 2411294 and 2417750.

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] Erik Altman, Jovan Blanuša, Luc Von Niederhäusern, Beni Egressy, Andreea Anghel, and Kubilay Atasu. 2023. Realistic Synthetic Financial Transactions for Anti-Money Laundering Models. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*. https://openreview.net/forum?id=XZf2bnMBag

[3] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. 2018. Distributed Evaluation of Subgraph Queries Using Worst-Case Optimal Low-Memory Dataflows. *Proceedings of the VLDB Endowment* 11, 6 (Feb. 2018), 691–704. doi:10.14778/3184470.3184473

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

[5] Albert-László Barabási and Réka Albert. 1999. Emergence of Scaling in Random Networks. *Science* 286, 5439 (1999), 509–512. arXiv:https://www.science.org/doi/pdf/10.1126/science.286.5439.509 doi:10.1126/science.286.5439.509

[6] Bibek Bhattarai and Howie Huang. 2022. SteinerLog: Prize collecting the audit logs for threat hunting on enterprise network. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*. 97–108.

[7] Bibek Bhattarai and H. Howie Huang. 2024. Prov2vec: Learning Provenance Graph Representation for Anomaly Detection in Computer Systems. In *Proceedings of the 19th International Conference on Availability, Reliability and Security* (Vienna, Austria) *(ARES '24)*. Association for Computing Machinery, New York, NY, USA, Article 15, 14 pages. doi:10.1145/3664476.3664494

[8] 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*. ACM, Amsterdam Netherlands, 1447–1462. doi:10.1145/3299869.3300086

[9] 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*. ACM, San Francisco California USA, 1199–1214. doi:10.1145/2882903.2915236

[10] Laurent Bindschaedler, Jasmina Malicevic, Baptiste Lepers, Ashvin Goel, and Willy Zwaenepoel. 2021. Tesseract: Distributed, General Graph Pattern Mining on Evolving Graphs. In *Proceedings of the Sixteenth European Conference on Computer Systems*. ACM, Online Event United Kingdom, 458–473. doi:10.1145/3447786.3456253

[11] Jovan Blanuša, Maximo Cravero Baraja, Andreea Anghel, Luc von Niederhäusern, Erik Altman, Haris Pozidis, and Kubilay Atasu. 2024. Graph Feature Preprocessor: Real-time Subgraph-based Feature Extraction for Financial Crime Detection. In *Proceedings of the 5th ACM International Conference on AI in Finance* (Brooklyn, NY, USA) *(ICAIF '24)*. Association for Computing Machinery, New York, NY, USA, 222–230. doi:10.1145/3677052.3698674

[12] Anthony Bonato, Juan Sebastian Chavez Palan, and Adam Szava. 2024. Enhancing Anti-Money Laundering Efforts with Network-Based Algorithms. arXiv:2409.00823 [cs.SI] https://arxiv.org/abs/2409.00823

[13] Vincenzo Bonnici, Alfredo Ferro, Rosalba Giugno, Alfredo Pulvirenti, and Dennis Shasha. 2010. Enhancing Graph Database Indexing by Suffix Tree Structure. In *Pattern Recognition in Bioinformatics, PRIB 2010 (Lecture Notes in Computer Science, Vol. 6282)*. Springer, 195–203. doi:10.1007/978-3-642-16001-1_20

[14] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Alfredo Ferro, and Dennis Shasha. 2013. A subgraph isomorphism algorithm and its application to biochemical data. *BMC Bioinformatics* 14, Suppl 7 (2013), S13. doi:10.1186/1471-2105-14-S7-S13

[15] Mario Cannataro and Pietro H Guzzi. 2012. *Data management of protein interaction networks*. Vol. 17. John Wiley & Sons.

[16] 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

[17] Jingji Chen and Xuehai Qian. 2022. DecoMine: A Compilation-Based Graph Pattern Mining System with Pattern Decomposition. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. ACM, Vancouver BC Canada, 47–61. doi:10.1145/3567955.3567956

[18] Jingji Chen and Xuehai Qian. 2023. Khuzdul: Efficient and Scalable Distributed Graph Pattern Mining Engine. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. ACM, Vancouver BC Canada, 413–426. doi:10.1145/3575693.3575743

[19] Qihang Chen, Boyu Tian, and Mingyu Gao. 2022. FINGERS: Exploiting Fine-Grained Parallelism in Graph Mining Accelerators. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Lausanne Switzerland, 43–55. doi:10.1145/3503222.3507730

[20] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, Loc Hoang, and Keshav Pingali. 2021. Sandslash: a two-level framework for efficient graph pattern mining. In *Proceedings of the 35th ACM International Conference on Supercomputing* (Virtual Event, USA) *(ICS '21)*. Association for Computing Machinery, New York, NY, USA, 378–391. doi:10.1145/3447818.3460359

[21] Xuhao Chen, Tianhao Huang, Shuotao Xu, Thomas Bourgeat, Chanwoo Chung, and Arvind Arvind. 2021. FlexMiner: A Pattern-Aware Accelerator for Graph Pattern Mining. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Valencia, Spain, 581–594. doi:10.1109/ISCA52012.2021.00052

[22] Yunyoung Choi, Kunsoo Park, and Hyunjoon Kim. 2023. BICE: Exploring Compact Search Space by Using Bipartite Matching and Cell-Wide Verification. *Proceedings of the VLDB Endowment* 16, 9 (May 2023), 2186–2198. doi:10.14778/3598581.3598591

[23] Jack Chueng. 2025. SubgraphMatchingSurvey: A Survey and Benchmarking of Subgraph Matching Algorithms. https://github.com/JackChuengQAQ/SubgraphMatchingSurvey/. Accessed: July 2025.

[24] Aaron Clauset, Cosma Rohilla Shalizi, and M. E. J. Newman. 2009. Power-Law Distributions in Empirical Data. *SIAM Rev.* 51, 4 (2009), 661–703. arXiv:https://doi.org/10.1137/070710111 doi:10.1137/070710111

[25] Guohao Dai, Zhenhua Zhu, Tianyu Fu, Chiyue Wei, Bangyan Wang, Xiangyu Li, Yuan Xie, Huazhong Yang, and Yu Wang. 2022. DIMMining: Pruning-Efficient and Parallel Graph Mining on near-Memory-Computing. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. ACM, New York New York, 130–145. doi:10.1145/3470496.3527388

[26] Amol Deshpande. 2018. In situ graph querying and analytics with graphgen: extended abstract. In *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)* (Houston, Texas) *(GRADES-NDA '18)*. Association for Computing Machinery, New York, NY, USA, Article 2, 2 pages. doi:10.1145/3210259.3210261

[27] Vinicius Dias, Carlos H. C. Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. 2019. Fractal: A General-Purpose Graph Pattern Mining System. In *Proceedings of the 2019 International Conference on Management of Data*. ACM, Amsterdam Netherlands, 1357–1374. doi:10.1145/3299869.3319875

[28] Europol. 2021. *Serious and Organised Crime Threat Assessment (SOCTA) 2021.* Technical Report. European Union Agency for Law Enforcement Cooperation (Europol), The Hague, Netherlands. https://www.europol.europa.eu/cms/sites/default/files/documents/socta2021_1.pdf Accessed: 2025-03-31.

[29] Wenfei Fan. 2012. Graph pattern matching revised for social network analysis. In *Proceedings of the 15th International Conference on Database Theory* (Berlin, Germany) *(ICDT '12)*. Association for Computing Machinery, New York, NY, USA, 8–21. doi:10.1145/2274576.2274578

[30] Michael Frasca, Kamesh Madduri, and Padma Raghavan. 2012. NUMA-aware Graph Mining Techniques for Performance and Energy Efficiency. In *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Salt Lake City, UT, 1–11. doi:10.1109/SC.2012.81

[31] Rosalba Giugno, Vincenzo Bonnici, Nicole Bombieri, Alfredo Pulvirenti, Alfredo Ferro, and Dennis Shasha. 2013. GRAPES: A Software for Parallel Searching on Biological Graphs Targeting Multi-Core Architectures. *PloS one* 8, 10 (2013), e76850. doi:10.1371/journal.pone.0076850

[32] Chuangyi Gui, Xiaofei Liao, Long Zheng, and Hai Jin. [n. d.]. Cyclosa: Redundancy-Free Graph Pattern Mining via Set Dataflow. ([n. d.]).

[33] Chuangyi Gui, Xiaofei Liao, Long Zheng, Pengcheng Yao, Qinggang Wang, and Hai Jin. 2021. SumPA: Efficient Pattern-Centric Graph Mining with Pattern Abstraction. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, Atlanta, GA, USA, 318–330. doi:10.1109/PACT52795.2021.00030

[34] Wei Guo, Yuchen Li, Miaomiao Sha, Bingsheng He, Xiaokui Xiao, and Kian-Lee Tan. 2020. GPU-Accelerated Subgraph Enumeration on Partitioned Graphs. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Azza Abouzied, and Hung Q. Ngo (Eds.). ACM, 1067–1082. doi:10.1145/3318464.3389699

[35] Wentian Guo, Yuchen Li, and Kian-Lee Tan. 2022. Exploiting Reuse for GPU Subgraph Enumeration. *IEEE Transactions on Knowledge and Data Engineering* 34, 9 (Sept. 2022), 4231–4244. doi:10.1109/TKDE.2020.3035564

[36] 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 the 2019 International Conference on Management of Data*. ACM, Amsterdam Netherlands, 1429–1446. doi:10.1145/3299869.3319880

[37] 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, 337–348. doi:10.1145/2463676.2465300

[38] 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, 405–418. doi:10.1145/1376616.1376660

[39] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. 2020. Peregrine: A Pattern-Aware Graph Mining System. In *Proceedings of the Fifteenth European Conference on Computer Systems*. ACM, Heraklion Greece, 1–16. doi:10.1145/3342195.3387548

[40] Kasra Jamshidi, Harry Xu, and Keval Vora. 2023. Accelerating Graph Mining Systems with Subgraph Morphing. In *Proceedings of the Eighteenth European Conference on Computer Systems*. ACM, Rome Italy, 162–181. doi:10.1145/3552326.3567489

[41] Xun Jian, Zhiyuan Li, and Lei Chen. 2023. SUFF: Accelerating Subgraph Matching with Historical Data. *Proceedings of the VLDB Endowment* 16, 7 (March 2023), 1699–1711. doi:10.14778/3587136.3587144

[42] Guanxian Jiang, Qihui Zhou, Tatiana Jin, Boyang Li, Yunjian Zhao, Yichao Li, and James Cheng. 2022. VSGM: View-Based GPU-Accelerated Subgraph Matching on Large Graphs. In *SC22: International Conference for High Performance*

Computing, Networking, Storage and Analysis. IEEE, Dallas, TX, USA, 1–15. doi:10.1109/SC41404.2022.00057

[43] Jiaxin Jiang, Byron Choi, Xin Huang, Jianliang Xu, and Sourav S Bhowmick. 2024. DKWS: A Distributed System for Keyword Search on Massive Graphs. *IEEE Transactions on Knowledge and Data Engineering* 36, 5 (2024), 1935–1950. doi:10.1109/TKDE.2023.3313726

[44] Zite Jiang, Shuai Zhang, Xingzhong Hou, Mengting Yuan, and Haihang You. 2024. IVE: Accelerating Enumeration-Based Subgraph Matching via Exploring Isolated Vertices. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, Utrecht, Netherlands, 4208–4221. doi:10.1109/ICDE60146.2024.00321

[45] 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

[46] Xin Jin, Zhengyi Yang, Xuemin Lin, Shiyu Yang, Lu Qin, and You Peng. 2021. FAST: FPGA-based Subgraph Matching on Massive Graphs. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, Chania, Greece, 1452–1463. doi:10.1109/ICDE51399.2021.00129

[47] Alpár Jüttner and Péter Madarasi. 2018. VF2++—An improved subgraph isomorphism algorithm. *Discrete Applied Mathematics* 242 (03 2018). doi:10.1016/j.dam.2018.02.018

[48] Arijit Khan, Xiangyu Ke, and Yinghui Wu. 2025. Graph Data Management and Graph Machine Learning: Synergies and Opportunities. arXiv:2502.00529 [cs.DB] https://arxiv.org/abs/2502.00529

[49] 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*. ACM, Virtual Event China, 925–937. doi:10.1145/3448016.3457265

[50] Hyunjoon Kim, Yunyoung Choi, Kunsoo Park, Xuemin Lin, Seok-Hee Hong, and Wook-Shin Han. 2022. Fast subgraph query processing and subgraph matching via static and dynamic equivalences. *The VLDB Journal* 32, 2 (June 2022), 343–368. doi:10.1007/s00778-022-00749-x

[51] 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, 1231–1245. doi:10.1145/2882903.2915209

[52] Kyoungmin Kim, In Seo, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. 2018. TurboFlux: A Fast Continuous Subgraph Matching System for Streaming Graph Data. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 411–426. doi:10.1145/3183713.3196917

[53] Sunghwan Kim, Jie Chen, Ting Cheng, Asta Gindulyte, Jiyao He, Shuai He, Qifang Li, Benjamin A. Shoemaker, Paul A. Thiessen, Bo Yu, Lily Zaslavsky, Jian Zhang, and Evan E. Bolton. 2025. PubChem 2025 update. *Nucleic Acids Research* 53, D1 (2025), D1516–D1525. doi:10.1093/nar/gkae1059

[54] K. Klein, N. Kriege, and P. Mutzel. 2011. CT-Index: Fingerprint-Based Graph Indexing Combining Cycles and Trees. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering, ICDE 2011, Hannover, Germany, April 11-16, 2011*. IEEE, 1115–1126.

[55] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. [n. d.]. Scalable Subgraph Enumeration in MapReduce. ([n. d.]).

[56] Longbin Lai, Lu Qin, Xuemin Lin, Ying Zhang, Lijun Chang, and Shiyu Yang. 2016. Scalable distributed subgraph enumeration. *Proc. VLDB Endow.* 10, 3 (Nov. 2016), 217–228. doi:10.14778/3021924.3021937

[57] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. 2012. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *Proceedings of the VLDB Endowment* 6, 2 (2012), 133–144.

[58] Seok Lim, Youngho Kim, Junmo Gu, Seungmin Lee, Wooje Shin, and Sun Kim. 2022. Supervised chemical graph mining improves drug-induced liver injury prediction. *iScience* 26, 1 (2022), 105677. doi:10.1016/j.isci.2022.105677

[59] A. Lin, R. Yang, S. Dorkenwald, A. Matsliah, A. R. Sterling, P. Schlegel, S. C. Yu, C. E. McKellar, M. Costa, K. Eichler, A. S. Bates, N. Eckstein, J. Funke, G. S. X. E. Jefferis, and M. Murthy. 2024. Network statistics of the whole-brain connectome of Drosophila. *Nature* 634, 8032 (Oct 2024), 153–165. doi:10.1038/s41586-024-07968-y Epub 2024 Oct 2.

[60] Zhiheng Lin, Ke Meng, Chaoyang Shui, Kewei Zhang, Junmin Xiao, and Guangming Tan. 2024. Exploiting Fine-Grained Redundancy in Set-Centric Graph Pattern Mining. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. ACM, Edinburgh United Kingdom, 175–187. doi:10.1145/3627535.3638507

[61] Guanfeng Liu, Kai Zheng, Yan Wang, Mehmet A Orgun, An Liu, Lei Zhao, and Xiaofang Zhou. 2015. Multi-constrained graph pattern matching in large-scale contextual social graphs. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 351–362.

[62] Juelin Liu, Sandeep Polisetty, Hui Guan, and Marco Serafini. 2023. GraphMini: Accelerating Graph Pattern Matching Using Auxiliary Graphs. In *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, Vienna, Austria, 211–224. doi:10.1109/PACT58117.2023.00026

[63] Yujie Lu, Zhijie Zhang, and Weiguo Zheng. 2025. BSX: Subgraph Matching with Batch Backtracking Search. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–27.

[64] 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. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, Utrecht, Netherlands, 3230–3243. doi:10.1109/ICDE60146.2024.00250

[65] Daniel Mawhirter, Samuel Reinehr, Wei Han, Noah Fields, Miles Claver, Connor Holmes, Jedidiah McClurg, Tongping Liu, and Bo Wu. 2021. Dryadic: Flexible and Fast Graph Pattern Matching at Scale. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, Atlanta, GA, USA, 289–303. doi:10.1109/PACT52795.2021.00028

[66] Daniel Mawhirter, Sam Reinehr, Connor Holmes, Tongping Liu, and Bo Wu. 2021. GraphZero: A High-Performance Subgraph Matching System. *SIGOPS Oper. Syst. Rev.* 55, 1 (jun 2021), 21–37. doi:10.1145/3469379.3469383

[67] Daniel Mawhirter and Bo Wu. 2019. AutoMine: Harmonizing High-Level Abstraction and High Performance for Graph Mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. ACM, Huntsville Ontario Canada, 509–523. doi:10.1145/3341301.3359633

[68] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. arXiv:1903.02076 [cs.DB] https://arxiv.org/abs/1903.02076

[69] Seunghwan Min, Sung Gwan Park, Kunsoo Park, Dora Giammarresi, Giuseppe F. Italiano, and Wook-Shin Han. 2021. Symmetric Continuous Subgraph Matching with Bidirectional Dynamic Programming. *Proceedings of the VLDB Endowment* 14, 8 (April 2021), 1298–1310. doi:10.14778/3457390.3457395

[70] MEJ Newman. 2005. Power laws, Pareto distributions and Zipf's law. *Contemporary Physics* 46, 5 (2005), 323–351. arXiv:https://doi.org/10.1080/00107510500052444 doi:10.1080/00107510500052444

[71] openGraphMatching Developers. 2025. openGraphMatching: A Python Package for Graph Matching. https://pypi.org/project/openGraphMatching/. Accessed: July 2025.

[72] Todd Plantenga. 2013. Inexact subgraph isomorphism in MapReduce. *J. Parallel Distrib. Comput.* 73, 2 (feb 2013), 164–175. doi:10.1016/j.jpdc.2012.10.005

[73] 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.

[74] Miao Qiao, Hao Zhang, and Hong Cheng. 2017. Subgraph matching: on compression and computation. *Proc. VLDB Endow.* 11, 2 (Oct. 2017), 176–188. doi:10.14778/3149193.3149198

[75] Linshan Qiu, Lu Chen, Hailiang Jie, Xiangyu Ke, Yunjun Gao, Yang Liu, and Zetao Zhang. 2024. GPU-Accelerated Batch-Dynamic Subgraph Matching. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, Utrecht, Netherlands, 3204–3216. doi:10.1109/ICDE60146.2024.00248

[76] RapidsAtHKUST. 2025. SubgraphMatching: A Research-Oriented Subgraph Matching Framework. https://github.com/RapidsAtHKUST/SubgraphMatching/. Accessed: July 2025.

[77] Xuguang Ren and Junhu Wang. 2015. Exploiting Vertex Relationships in Speeding up Subgraph Isomorphism over Large Graphs. *Proceedings of the VLDB Endowment* 8, 5 (Jan. 2015), 617–628. doi:10.14778/2735479.2735493

[78] Xuguang Ren, Junhu Wang, Wook-Shin Han, and Jeffrey Xu Yu. 2019. Fast and Robust Distributed Subgraph Enumeration. *Proceedings of the VLDB Endowment* 12, 11 (July 2019), 1344–1356. doi:10.14778/3342263.3342272

[79] Carlos R. Rivero and Hasan M. Jamil. 2017. Efficient and scalable labeled subgraph matching using SGMatch. *Knowl. Inf. Syst.* 51, 1 (apr 2017), 61–87. doi:10.1007/s10115-016-0968-2

[80] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming Verification Hardness: An Efficient Algorithm for Testing Subgraph Isomorphism. *Proceedings of the VLDB Endowment* 1, 1 (Aug. 2008), 364–375. doi:10.14778/1453856.1453899

[81] 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*. ACM, Snowbird Utah USA, 625–636. doi:10.1145/2588555.2588557

[82] Tianhui Shi, Jidong Zhai, Haojie Wang, Qiqian Chen, Mingshu Zhai, Zixu Hao, Haoyu Yang, and Wenguang Chen. 2023. GraphSet: High Performance Graph Mining through Equivalent Set Transformations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, Denver CO USA, 1–14. doi:10.1145/3581784.3613213

[83] Tianhui Shi, Mingshu Zhai, Yi Xu, and Jidong Zhai. 2020. GraphPi: High Performance Graph Pattern Matching through Effective Redundancy Elimination. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Atlanta, GA, USA, 1–14. doi:10.1109/SC41405.2020.00104

[84] Konstantinos Skitsas, Davide Mottin, and Panagiotis Karras. 2025. PILOS: Scalable Large-Subgraph Matching by Online Spectral Filtering. In *Proceedings of the 2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1180–1193. doi:10.1109/ICDE65448.2025.00093

[85] Maria Sorokina, Philipp Merseburger, Karthikeyan Rajan, and et al. 2021. COCONUT online: Collection of Open Natural Products database. *Journal of Cheminformatics* 13, 1 (2021), 2. doi:10.1186/s13321-020-00478-9

[86] Xunbin Su, Yinnian Lin, and Lei Zou. 2023. FASI: FPGA-friendly Subgraph Isomorphism on Massive Graphs. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, Anaheim, CA, USA, 2099–2112. doi:10.1109/ICDE55515.2023.00163

[87] Shixuan Sun and Qiong Luo. 2019. Scaling Up Subgraph Query Processing with Efficient Subgraph Matching. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 220–231. doi:10.1109/ICDE.2019.00028

[88] 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*. ACM, Portland OR USA, 1083–1098. doi:10.1145/3318464.3380581

[89] Shixuan Sun and Qiong Luo. 2022. Subgraph Matching With Effective Matching Order and Indexing. *IEEE Transactions on Knowledge and Data Engineering* 34, 1 (2022), 491–505. doi:10.1109/TKDE.2020.2980257

[90] 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), 176–188. doi:10.14778/3425879.3425888

[91] Shixuan Sun, Xibo Sun, Bingsheng He, and Qiong Luo. 2022. RapidFlow: An Efficient Approach to Continuous Subgraph Matching. *Proceedings of the VLDB Endowment* 15, 11 (July 2022), 2415–2427. doi:10.14778/3551793.3551803

[92] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. 2012. Efficient subgraph matching on billion node graphs. *arXiv preprint arXiv:1205.6691* (2012).

[93] Nishil Talati, Haojie Ye, Yichen Yang, Leul Belayneh, Kuan-Yu Chen, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2022. NDMiner: Accelerating Graph Pattern Mining Using near Data Processing. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. ACM, New York New York, 146–159. doi:10.1145/3470496.3527437

[94] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. 2015. Arabesque: A System for Distributed Graph Mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, Monterey California, 425–440. doi:10.1145/2815400.2815410

[95] J. R. Ullmann. 1976. An Algorithm for Subgraph Isomorphism. *J. ACM* 23, 1 (Jan. 1976), 31–42. doi:10.1145/321921.321925

[96] Todd L Veldhuizen. 2014. Leapfrog triejoin: A simple, worst-case optimal join algorithm. In *Proc. International Conference on Database Theory*.

[97] Hanchen Wang, Rong Hu, Ying Zhang, Lu Qin, Wei Wang, and Wenjie Zhang. 2022. Neural Subgraph Counting with Wasserstein Estimator. In *Proceedings of the 2022 International Conference on Management of Data*. ACM, Philadelphia PA USA, 160–175. doi:10.1145/3514221.3526163

[98] Huiju Wang, Zhengkui Wang, Kian-Lee Tan, Chee-Yong Chan, Qi Fan, and Xiao Yue. 2017. VCExplorer: A Interactive Graph Exploration Framework Based on Hub Vertices with Graph Consolidation. arXiv:1709.06745 [cs.DB] https://arxiv.org/abs/1709.06745

[99] 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 *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, Kuala Lumpur, Malaysia, 245–258. doi:10.1109/ICDE53745.2022.00023

[100] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. [n. d.]. RStream: Marrying Relational Algebra with Streaming for Efficient Graph Mining on A Single Machine. ([n. d.]).

[101] Yuxuan Wang, Yizhou Xia, Jun Yan, Yifan Yuan, Hong-Bin Shen, and Xiaoyong Pan. 2023. ZeroBind: a protein-specific zero-shot predictor with subgraph matching for drug-target interactions. *Nature Communications* 14, 1 (2023), 7861. doi:10.1038/s41467-023-43597-1

[102] Zhaokang Wang, Rong Gu, Weiwei Hu, Chunfeng Yuan, and Yihua Huang. 2019. BENU: Distributed Subgraph Enumeration with Backtracking-Based Framework. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, Macao, Macao, 136–147. doi:10.1109/ICDE.2019.00021

[103] Yihua Wei and Peng Jiang. 2022. STMatch: Accelerating Graph Pattern Matching on GPU with Stack-Based Loop Optimizations. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Dallas, TX, USA, 1–13. doi:10.1109/SC41404.2022.00058

[104] Lichen Xiang, Arif Khan, Elena Serra, Mahantesh Halappanavar, and Anand Sukumaran-Rajam. 2021. cuTS: Scaling Subgraph Isomorphism on Distributed Multi-GPU Systems Using Trie Based Data Structure. In *SC '21: The International Conference for High Performance Computing, Networking, Storage and Analysis, St. Louis, Missouri, USA, November 14 - 19, 2021*, Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin (Eds.). ACM, 69:1–69:14. doi:10.1145/3458817.3476214

[105] 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

[106] 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, 2049–2062. doi:10.1145/3448016.3457237

[107] Yutong Ye, Xiang Lian, and Mingsong Chen. 2024. Efficient Exact Subgraph Matching via GNN-Based Path Dominance Embedding. *Proc. VLDB Endow.* 17, 7 (March 2024), 1628–1641. doi:10.14778/3654621.3654630

[108] Kaiqiang Yu, Kaixin Wang, Cheng Long, Laks Lakshmanan, and Reynold Cheng. 2025. Fast Maximum Common Subgraph Search: A Redundancy-Reduced Backtracking Approach. arXiv:2502.11557 [cs.DB] https://arxiv.org/abs/2502.11557

[109] Ye Yuan, Delong Ma, Zhenyu Wen, Zhiwei Zhang, and Guoren Wang. 2021. Subgraph matching over graph federation. *Proc. VLDB Endow.* 15, 3 (Nov. 2021), 437–450. doi:10.14778/3494124.3494129

[110] L. Zeng, L. Zou, M. Tamer Özsu, L. Hu, and F. Zhang. 2020. GSI: GPU-friendly Subgraph Isomorphism. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 1249–1260. doi:10.1109/ICDE48307.2020.00112

[111] Shijie Zhang, Shirong Li, and Jiong Yang. 2009. GADDI: Distance Index Based Subgraph Matching in Biological Networks. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*. ACM, Saint Petersburg Russia, 192–203. doi:10.1145/1516360.1516384

[112] 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

[113] 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 (March 2024), 29 pages. doi:10.1145/3639315

[114] Peixiang Zhao and Jiawei Han. 2010. On Graph Query Optimization in Large Networks. *Proceedings of the VLDB Endowment* 3, 1-2 (Sept. 2010), 340–351. doi:10.14778/1920841.1920887

Received April 2025; revised July 2025; accepted August 2025