In-Memory Subgraph Matching: An In-depth Study

Shixuan Sun
Hong Kong University of Science and Technology
Hong Kong SAR, China
ssunah@cse.ust.hk

Qiong Luo
Hong Kong University of Science and Technology
Hong Kong SAR, China
luo@cse.ust.hk

ABSTRACT

We study the performance of eight representative in-memory subgraph matching algorithms. Specifically, we put QuickSI, GraphQL, CFL, CECI, DP-iso, RI and VF2++ in a common framework to compare them on the following four aspects: (1) method of filtering candidate vertices in the data graph; (2) method of ordering query vertices; (3) method of enumerating partial results; and (4) other optimization techniques. Then, we compare the overall performance of these algorithms with Glasgow, an algorithm based on the constraint programming. Through experiments, we find that (1) the filtering method of GraphQL is competitive to that of the latest algorithms CFL, CECI and DP-iso in terms of pruning power; (2) the ordering methods in GraphQL and RI are usually the most effective; (3) the set intersection based local candidate computation in CECI and DP-iso performs the best in the enumeration; and (4) the failing sets pruning in DP-iso can significantly improve the performance when queries become large. Our source code is publicly available at https://github.com/RapidsAtHKUST/SubgraphMatching.

CCS CONCEPTS

• **Information systems** → **Information retrieval query processing**.

KEYWORDS

graph, subgraph matching, comparison and analysis

1 INTRODUCTION

Subgraph matching finds all embeddings in a data graph G that are isomorphic to a query graph q where both G and q are labeled graphs. For example, given the graphs in Figure 1, $\{(u_0, v_0), (u_1, v_4), (u_2, v_5), (u_3, v_{12})\}$ is a match from q to G. As one of the fundamental graph query operations, subgraph matching is widely used in both academia and industry [44].
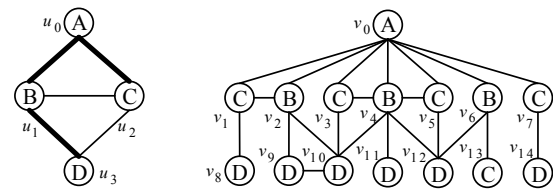
(a) Query graph q.   (b) Data graph G.

Figure 1: Example graphs.

Due to the importance of subgraph matching, a variety of algorithms have been proposed. In Table 1, we categorize representative subgraph matching algorithms by their computation models, main methods, and execution styles. Most of the algorithms in the database community follow an *exploration-based* method, which recursively extends intermediate results by mapping query vertices to data vertices along an order of query vertices [53]. In the artificial intelligence and the bioinformatics communities, many algorithms formulate subgraph matching within a *state space representation* where each state represents an intermediate result [10]. The feasible state in the state space is a match. Another approach is based on the *constraint programming* in which vertices and edges in q correspond to variables and constraints respectively [48]. The domain of the variables is the data vertices. This approach can evaluate subgraph matching by finding assignments to variables that satisfy the constraints. Despite the different models, these three approaches all adopt the backtracking search, which recursively extends partial results by mapping query vertices to data vertices to find all solutions. An alternative approach is to convert the query q to a multi-way join in which attributes and relations correspond to vertices and edges in q respectively, and evaluate the multi-way join to find all results [15]. Additionally, researchers accelerate subgraph matching by utilizing the parallel computation capability provided by hardware (e.g., GPUs [54]) and distributed environments.

Table 1: A summary of representative subgraph matching methods.

Community	Model	Methodology	Algorithms/Systems	
			Sequential	**Parallel**
Database	Exploration	Backtracking Search	QuickSI [45], GADDI [58], SPath [59], GraphQL [21], TurboIso [19], BoostIso [42], CFL [7], SGMatch [43], CECI [6], DP-iso [17]	PGX [41], PSM [51], STwig [53]
	Multi-Way Join	Pair-Wise Join	PostgreSQL, MonetDB, Neo4j	GpSM [54]
		Worst-Case Optimal Join	LogicBlox [5]	EmptyHeaded [1], GraphFlow [24]
Artificial Intelligence	State Space Representation	Backtracking Search	Ullmann [55], VF2 [13], VF2++ [23], VF3 [10]	VF3P [9]
	Constraint Programming	Backtracking Search	LAD [47], Glasgow [4]	pGlasgow [4, 35]
Bioinformatics	State Space Representation	Backtracking Search	RI [8], VF2+ [11]	pRI [28], Grapes [14]

Researchers have also put a lot of effort on some other graph query operations closely related to subgraph matching: (1) *subgraph enumeration*, which focuses on unlabeled graphs; (2) *subgraph containment*, which finds data graphs containing the given query graph from a collection of data graphs; and (3) *visual subgraph query*, which aims to provide user-friendly graphic user interfaces to make it easy for non-expert users to explore graph databases.

In this paper, we study eight representative in-memory subgraph matching algorithms by their individual techniques. We categorize the algorithms under study into three kinds. The first kind of algorithms (e.g., QuickSI [45]) follow the *direct-enumeration* framework, which directly explores G to enumerate all results. Most algorithms based on the state space representation model (e.g., RI [8] and VF2++ [23]) adopt this framework as well. The second category of algorithms (e.g., GADDI [58], SPath [59] and SGMatch [43]) utilize the *indexing-enumeration* framework, which constructs indices on G and answers all queries with the assistance of the indices. The third group of algorithms (e.g., GraphQL [21], TurboIso [19], CFL [7], CECI [6] and DP-iso [17]) adopt the *preprocessing-enumeration* framework, which is widely used in the recent algorithms in the database community. Specifically, these algorithms first generate a *candidate vertex set* for each query vertex, and build auxiliary data structures maintaining edges between candidate vertex sets. Then, they generate a *matching order* based on auxiliary data structures. Finally, they enumerate all results with the assistance of the auxiliary data structures along the matching order. Additionally, some algorithms design other optimization strategies to further reduce the search space during the enumeration. Because these steps are closely related, they all affect the subgraph matching performance. Unfortunately, all previous studies [26, 33] regarded each algorithm as a whole to compare their performance, and none of them evaluated the effectiveness of individual techniques such as the filtering method and the ordering method.

Our Work. We propose to study the performance of in-memory subgraph matching algorithms on four aspects: (1) method of filtering candidate vertices; (2) method of ordering query vertices; (3) method of enumerating partial results; and (4) other optimization techniques. We are not limited to giving a simple comparison of absolute performance between algorithms on the elapsed time of answering a query; rather, we investigate individual techniques in existing algorithms, and pinpoint the techniques in an algorithm that lead it to the performance differences.

We focus on five representative algorithms, which are QuickSI, GraphQL, CFL, CECI and DP-iso. Specifically, QuickSI and GraphQL performed well in a previous performance study [33]. CFL, CECI and DP-iso are the state-of-the-art algorithms that were not involved in any previous performance studies. We do not study any indexing-enumeration algorithms, because a previous performance study [33] has reported the issues incurred by the indices. For the selected algorithms, we first compare and analyze the filtering methods, the ordering methods, the enumeration methods and the optimization methods correspondingly. Then, we re-implement these algorithms within a common framework and optimize them with our best effort for the comparison.

Moreover, we study three algorithms proposed in other communities, which are RI [8], VF2++ [23] and Glasgow [4]. RI won the International Contest on Pattern Search in Biological Databases [56]. VF2++ significantly outperforms the widely used VF2 [13] algorithm. Glasgow is optimized for deciding whether G contains q, and supports finding all matches from q to G. As it is a very efficient algorithm in AI community [36], we study it in our experiments as well. We re-implement RI and VF2++ within our framework. As Glasgow is based on the constraint programming, it cannot be integrated into the framework. Therefore, we only compare the overall performance of Glasgow with other algorithms.

We conduct experiments on both real-world and synthetic datasets to study the performance of competing methods, and provide an in-depth analysis. In summary, we make the following contributions in this paper.

- We study the performance of eight subgraph matching algorithms from three research communities.
- We compare and analyze filtering methods, ordering methods, enumeration methods and optimization methods in seven selected algorithms respectively.
- We conduct experiments with both real-world and synthetic datasets to examine the effectiveness of each kind of methods respectively.
- We report our new findings through our experiments and analysis.

Table 2: Notations.

Notations	Descriptions
g, q and G	graph, query graph and data graph
V, E and Σ	vertex set, edge set and label set
$d(u)$, $L(u)$ and $N(u)$	degree, label and neighbors of u
$e(u, v)$	an edge between u and v
$g[V]$	vertex-induced subgraph of g on V
$E(q_t)$ and $\overline{E(q_t)}$	tree edges and non-tree edges of q_t
q_t, $u.c$ and $u.p$	spanning tree of q, the child and the parent of u in q_t
C and LC	candidate vertex set and local candidate vertex set
f and φ	subgraph isomorphism and matching order
$N_+^{\varphi}(u)$ $(N_-^{\varphi}(u))$	backward (forward) neighbors of u given φ
$\mathcal{A}$	auxiliary data structure
$\mathcal{A}_{u'}^u(v)$	neighbors of v in $C(u')$ where $v \in C(u)$

2 BACKGROUND AND RELATED WORK

2.1 Preliminaries

In this paper, we focus on the undirected labeled graph $g = (V, E)$ where V is a set of vertices and E is a set of edges. Given $u \in V$, $N(u)$ denotes the neighbors u' of u, i.e., $\{u' \in V | e(u, u') \in E\}$. Given $V' \subseteq V$, $g[V']$ is the vertex-induced subgraph of g on V'. Graph g is vertex-labeled, i.e., there is a label function L that associates a vertex u in V to a label l in a label set Σ. The query graph and the data graph are denoted by q and G respectively. We call the vertices in $V(q)$ *query vertices*, and the vertices in $V(G)$ *data vertices*. q and G share the same label function L. Given q, *2-core* of q is a maximal connected subgraph q' of q that satisfies $\forall u \in V(q'), d(u) \geqslant 2$, where $d(u)$ is the degree of u in q'. We call query vertices in 2-core of q the *core vertices*. We summarize the notations frequently used in Table 2. Next, we give a formal definition of subgraph matching and related preliminaries.

Definition 2.1. Subgraph Isomorphism: Given $q = (V, E)$ and $G = (V', E')$, a subgraph isomorphism is an **injective function** f from V to V' such that (1) $\forall u \in V, L(u) = L(f(u))$; and (2) $\forall e(u, u') \in E, e(f(u), f(u')) \in E'$.

Problem Definition. Given q and G, subgraph matching is to find all subgraph isomorphisms from q to G.

For brevity, we call a subgraph isomorphism a *match*. We assume that q is connected and $|V(q)| \geqslant 3$, because it is trivial to find all matches of a single vertex or a single edge.

Common Framework. Lee et al. [33] proposed a common framework to generalize the backtracking search of subgraph matching algorithms. Based on their work, we abstract and define some important concepts used in the latest algorithms. Algorithm 1 presents the generic subgraph matching framework, which takes q and G as input and outputs all matches from q to G. For each vertex $u \in V(q)$, line 1 first generates a *complete candidate vertex set* $C(u)$ defined in Definition 2.2, and builds an auxiliary data structure $\mathcal{A}$ that maintains edges between candidate vertex sets. Given $e(u, u') \in E(q)$ and $v \in C(u)$, $\mathcal{A}_{u'}^u(v) = N(v) \bigcap C(u')$, i.e., the neighbors of v in $C(u')$.

Algorithm 1: Generic Subgraph Matching

Input: a query graph q and a data graph G;
Output: all matches from q to G;
/* The filtering method. */
1   $C, \mathcal{A} \leftarrow$ generate candidate vertex sets and build auxiliary data structure;
/* The ordering method. */
2   $\varphi \leftarrow$ generate a matching order;
/* The enumeration method. */
3   Enumerate($q, G, C, \mathcal{A}, \varphi, \{\}, 1$);
4   **Procedure** Enumerate($q, G, C, \mathcal{A}, \varphi, M, i$)
5   **if** $i = |\varphi| + 1$ **then** Output M, **return**;
6     $u \leftarrow$ select an extendable vertex given φ and M;
/* The local candidate computation method. */
7     $LC(u, M) \leftarrow$ ComputeLC($q, G, C, \mathcal{A}, \varphi, M, u, i$);
8   **foreach** $v \in LC(u, M)$ **do**
9   **if** $v \notin M$ **then**
10   Add (u, v) to M;
11   Enumerate($q, G, C, \mathcal{A}, \varphi, M, i + 1$);
12   Remove (u, v) from M;

Definition 2.2. Complete Candidate Vertex Set: Given q and G, a complete candidate vertex set $C(u)$ of $u \in V(q)$ is a set of data vertices such that for each $v \in V(G)$, if (u, v) exists in a match from q to G, then v belongs to $C(u)$.

With candidate vertex sets and auxiliary data structures, line 2 generates a *matching order* defined in Definition 2.3. We further define the *backward (forward) neighbors*.

Definition 2.3. Matching Order: A matching order φ is a permutation of $V(q)$. $\varphi[i]$ is the ith vertex in φ, and $\varphi[i : j]$ is the set of vertices from index i to j ($1 \leqslant i \leqslant j \leqslant |\varphi|$).

Definition 2.4. Backward (Forward) Neighbors: Given an order π of query vertices, the backward (forward) neighbors $N_+^{\pi}(u)$ ($N_-^{\pi}(u)$) of $u \in \pi$ are the neighbors of u positioned before (after) u in π.

Line 3 recursively enumerates all results. M records mappings from query vertices to data vertices. If all query vertices have been mapped, then line 5 outputs M. Otherwise, line 6 selects an *extendable vertex* defined as follows.

Definition 2.5. Extendable Vertices: Given φ and M, extendable vertices $\Gamma(\varphi, M)$ are query vertices u such that each $u' \in N_+^{\varphi}(u)$ has been mapped in M but u has not.

Line 7 computes the *local candidate vertex set* $LC(u, M) = \{v \in C(u) | \forall u' \in N_+^{\varphi}(u), e(v, M[u']) \in E(G)\}$ where $M[u']$ is the data vertex mapped to u'. Lines 8-12 loop over $LC(u, M)$ to extend M, and recursively invoke the *Enumerate* procedure. Suppose that the first i vertices in φ have been mapped in M. Then, M is a match from $q[\varphi[1 : i]]$ to G.

2.2 Related Work

Subgraph Matching. Table 1 lists a variety of subgraph matching algorithms. The representative backtracking based algorithms will be introduced in Section 3. An alternative approach of the backtracking search is to convert subgraph

matching to a multi-way join. The worst-case optimal join is a kind of join algorithms whose running time matches the maximum output size of a query [38]. Previous experiment results showed that the database system LogicBlox [5] based on WCOJ outperformed the database systems (e.g., PostgreSQL, MonetDB and Neo4j) based on the pair-wise join (PJ) on subgraph matching queries [39]. EmptyHeaded [1] and Graphflow [24] developed by the academia adopt WCOJ to evaluate in-memory subgraph matching. As the join plan has an important impact on the performance, EmptyHeaded and Graphflow proposed methods to optimize it. Specifically, EmptyHeaded picks the *generalized hypertree decomposition* (GHD) [15] of q with the *minimum* width as the join plan among all GHDs. EmptyHeaded evaluates the sub-query (i.e., a GHD node) by WCOJ with an arbitrary ordering of query vertices in the sub-query. After that, EmptyHeaded executes a sequence of PJ on intermediate results of sub-queries to obtain final results. Graphflow designs an adaptive join plan generator based on a cost model considering the statistics of q and G as well as the cost of primitive operators, e.g., hash joins [37]. Among all possible plans, Graphflow picks the minimum cost one. Moreover, Graphflow not only consider to extend one vertex at a time but also conduct a PJ on two sub-queries when enumerating join plans.

Although both the algorithms in our study and the WCOJ-based algorithms can evaluate subgraph matching, the two kinds of algorithms have several differences. Specifically, the algorithms in our study target at queries with tens of vertices, and are evaluated on datasets with thousands to millions of vertices, whereas the WCOJ-based methods focus on smaller queries (generally less than 10 vertices) with datasets containing millions to hundred of millions of vertices. The WCOJ-based methods pick an "optimal" query vertex order in a number of possible orders based on their cost models. In contrast, the ordering methods in our study use the greedy approach to generate a matching order based on some heuristic rules or cost estimation, because it is very expensive to enumerate all possible orders of tens of vertices. The algorithms in our study design a variety of filtering methods to reduce the candidates for each query vertex, whereas Empty-Headed and Graphflow generally filter vertices based on the label information. The WCOJ-based methods can perform pair-wise joins on the intermediate results of two sub-queries. In contrast, the algorithms in our study recursively extend intermediate results by mapping query vertices to data vertices. Additionally, the WCOJ-based methods by default find *subgraph homomorphisms*, which allow that a result contains the same data vertices.

Subgraph matching has been widely studied on multi-core CPUs, such as PGX [41], PSM [51], VF3P [9], pRI [28],

and Grapes [14]. Graphflow [24] can execute with multiple threads on CPUs. EmptyHeaded [1] exploits both multi-threading and SIMD (single-instruction multiple data). GpSM [54] works on GPUs and is based on pair-wise joins. Except the sequential version, CECI [6] and Glasgow [4, 35] can run in parallel on both a single machine and multiple machines. STwig [53] works in a distributed environment.

Subgraph Enumeration. Due to the lack of labels, the search space of subgraph enumeration is much larger. Most of the recent work utilizes distributed environments to parallelize the search. Afrati et al. [2] proposed a multiway join based approach executing in one map-reduce round. Shao et.al [46] presented an approach based on Giraph. Lai et al. presented TwinTwig [30] and SEED [31] on MapReduce. To reduce the output/shuffle cost, Qiao et al. proposed CRYSTAL [40] to compress the intermediate results. Wang et al. [57] designed a distributed algorithm with a recursive-backtracking framework. BiGJoin [3] is a distributed algorithm based on WCOJ. Lai et al. [32] conducted a comprehensive comparison of the distributed algorithms. There are also parallel algorithms working on a single machine, such as LIGHT [50] and DualSim [27]. Moreover, both EmptyHeaded [1] and Graphflow [24] can handle unlabeled graphs as well.

Subgraph Containment. Subgraph containment finds all data graphs containing the query graph from a collection of data graphs with tens to thousands of vertices. Most of the algorithms follow the *indexing-filtering-verification* (IFV) paradigm, such as CT-Index [29]. Previous research focused on designing effective indices to eliminate unsatisfiable instances without subgraph isomorphism test. However, the indexing structures have severe scalability issues [25]. Sun et al. [52] proposed to utilize the preprocessing-enumeration subgraph matching algorithm to perform subgraph containment, which does not require any indices. Moreover, some researchers argue that the filtering technique cannot be beneficial when paired with reasonable subgraph isomorphism test algorithms [36].

Visual Subgraph Query. Because queries are visually offered by human, the visual subgraph query systems such as QUBLE [22] and BOOMER [49] utilize the latency introduced by the human interaction to process queries during query formulation, which has several potential benefits such as improving the system response time.

Graph Database Systems. A variety of graph databases are developed to manage graphs efficiently, which supports a collection of graph update and search operations. Recently, researchers conduct a comprehensive study of existing graph databases, give a guideline for system selection and provide an open-source suite for studying graph database systems [34]. In contrast, our paper focuses on in-memory algorithms designed for a single graph query operation.

3 COMPARISON AND ANALYSIS OF COMPETING ALGORITHMS

In this section, we compare and analyze the competing methods under our study.

3.1 Filtering Methods

3.1.1 Overview

Basic Methods. The *label and degree filtering* (LDF) generates $C(u)$ based on $L(u)$ and $d(u)$: $C(u) = \{v \in V(G)|L(v) = L(u) \wedge d(v) \geqslant d(u)\}$. All existing algorithms adopt LDF. The *neighbor label frequency filtering* (NLF) utilizes $N(u)$ to prune $C(u)$ as follows: given $v \in C(u)$, if there exists $l \in L(N(u))$ such that $|N(u,l)| > |N(v,l)|$ where $L(N(u)) = \{L(u')|u' \in N(u)\}$ and $N(u,l) = \{u' \in N(u)|L(u') = l\}$, then remove v from $C(u)$. CFL, CECI and DP-iso utilize NLF.

QuickSI, RI and VF2++. As direct-enumeration algorithms, they do not generate candidate vertex sets before the enumeration. Instead, they use some filtering rules to prune invalid vertices during the enumeration (see Section 3.3).

GraphQL. GraphQL generates candidate vertex sets in two steps, the local pruning and the global refinement. The local pruning generates $C(u)$ based on the *profile* of the neighborhood subgraph of u, which is the lexicographic order of labels of u and neighbors within distance r (hops of neighbors) from u, for example, the profile of u_1 in Figure 1(a) within distance 1 is $ABCD$. If the profile of $u \in V(q)$ is a sub-sequence of that of $v \in V(G)$, then add v to $C(u)$. The global refinement prunes candidate vertex sets generated by the local pruning with a pseudo subgraph isomorphism algorithm [20] as follows: given $v \in C(u)$, (1) build a bipartite graph B_v^u between $N(u)$ and $N(v)$ by adding $e(u',v')$ to B_v^u where $u' \in N(u)$ and $v' \in N(v)$ if $v' \in C(u')$; (2) check whether there is a semi-perfect matching in B_v^u, i.e., all vertices in $N(u)$ are matched; and (3) If not, remove v from $C(u)$. The pseudo subgraph isomorphism algorithm repeats the above procedure k times where k is specified by users. When $k = 1$ and $r = 1$, the time complexity is $O(|V(q)| \times |E(G)| + \sum_{u \in V(q)} \sum_{v \in V(G)}(d(u) \times d(v) + \Theta(d(u), d(v))))$ where $O(|V(q)| \times |E(G)|)$ is the time on the local pruning, $d(u) \times d(v)$ is the time on constructing B_v^u, and $\Theta(.)$ is the time complexity of testing the existence of a semi-perfect matching.

(a) $B_{v_1}^{u_2}$.   (b) $B_{v_3}^{u_2}$.

Figure 2: Running example of filtering in GraphQL.

EXAMPLE 3.1. *Given graphs in Figure 1, the local pruning generates $C(u_0) = \{v_0\}$, $C(u_1) = \{v_2, v_4, v_6\}$, $C(u_2) = \{v_1, v_3, v_5\}$ and $C(u_3) = \{v_{10}, v_{12}\}$ given $r = 1$. Given $v_1 \in$*
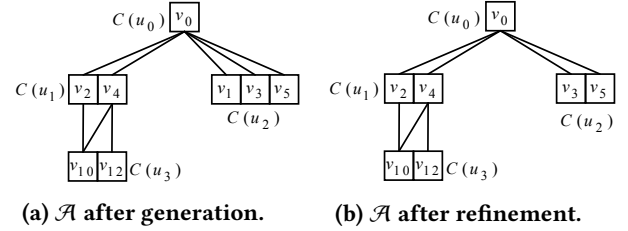

(a) $\mathcal{A}$ after generation.   (b) $\mathcal{A}$ after refinement.

Figure 3: Running example of filtering in CFL.

$C(u_2)$, the bipartite graph between $N(u_2)$ and $N(v_1)$ is shown in Figure 2(a). As $v_0 \in C(u_0)$ and $v_2 \in C(u_1)$, we add $e(u_0,v_0)$ and $e(u_1,v_2)$ to $B_{v_1}^{u_2}$. We remove v_1 from $C(u_2)$, since there is no semi-perfect matching. In contrast, v_3 is a valid candidate of u_2 in Figure 2(b).

CFL. Besides candidate vertex sets, CFL designs a tree-structured auxiliary data structure $\mathcal{A}$, called the *compressed path index*. The space complexity of $\mathcal{A}$ in CFL is $O(|V(q)| \times |E(G)|)$, and the time complexity of the filtering method is $O(|E(q)| \times |E(G)|)$. Specifically, CFL generates and prunes candidate vertex sets based on the following observation.

OBSERVATION 3.1. *Suppose that for each $u \in V(q)$, $C(u)$ is complete. $S_{u'}$ denotes $N(v) \bigcap C(u')$ where $v \in C(u)$ and $u' \in N(u)$. If the mapping (u,v) exists in a match from q to G, then v must satisfy for each $u' \in N(u)$, $S_{u'} \neq \emptyset$.*

Moreover, CFL derives the following two rules.

GENERATION RULE 3.1. *Given $X \subseteq N(u)$ where $u \in V(q)$, $C(u)$ can be generated by $\bigcap_{u' \in X} N(C(u'))$, i.e., intersecting the neighbors of candidates of $u' \in X$.*

FILTERING RULE 3.1. *Given $X \subseteq N(u)$ and $v \in C(u)$ where $u \in V(q)$, if there exists $u' \in X$ such that $C(u') \bigcap N(v) = \emptyset$, then v can be safely removed from $C(u)$ without breaking its completeness.*

CFL first obtains a BFS tree q_t of q, and then constructs $\mathcal{A}$ in two phases: (1) generate $C(u)$ along q_t level-by-level from top to bottom based on Generation Rule 3.1, and also perform backward pruning at each level based on Filtering Rule 3.1; and (2) refine $C(u)$ along q_t in a bottom-up order based on Filtering Rule 3.1. Given u and its parent $u.p$ in q_t, $\mathcal{A}$ in CFL maintains edges between candidates in $C(u.p)$ and those in $C(u)$. Example 3.2 illustrates the filtering in CFL.

EXAMPLE 3.2. *Given graphs in Figure 1, the BFS tree q_t of q is depicted by thick lines. In the generation phase, CFL first generates $C(u_0) = \{v_0\}$ with NLF. Based on $C(u_0)$, CFL obtains $C(u_1) = \{v_2, v_4, v_6\}$ with the generation rule. Note that when adding v to $C(u)$, v must pass the check of LDF and NLF. Similarly, CFL gets $C(u_2) = \{v_1, v_3, v_5\}$ based on $C(u_0)$ and $C(u_1)$. Next, it prunes $C(u_1)$ based on $C(u_2)$ by utilizing the non-tree edge $e(u_1,u_2)$ (the filtering rule). As a result, CFL removes v_6 from $C(u_1)$. $C(u_3)$ is generated based on $C(u_1)$ and*
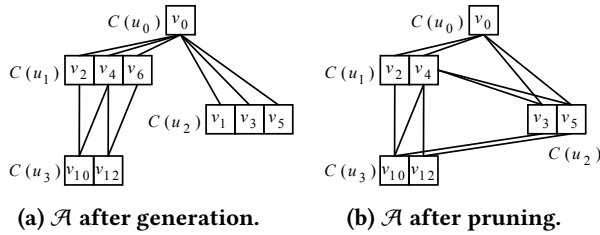
(a) $\mathcal{A}$ after generation. (b) $\mathcal{A}$ after pruning.
Figure 4: Running example of filtering in CECI.

(a) The 1st refinement of $\mathcal{A}$. (b) Final result of $\mathcal{A}$.
Figure 5: Running example of filtering in DP-iso.

$C(u_2)$. Figure 3(a) illustrates $\mathcal{A}$ after the generation. Next, the bottom-up refinement first prunes $C(u_1)$ and $C(u_2)$ based on $C(u_3)$, and then refines $C(u_0)$ based on $C(u_1)$ and $C(u_2)$. Take $C(u_2)$ as an example: CFL removes v_1 from $C(u_2)$, since v_1 has no neighbor in $C(u_3)$. $\mathcal{A}$ after the refinement is shown in Figure 3(b). $\mathcal{A}$ maintains edges between candidate vertices during both generation and refinement phases. Given $v_4 \in C(u_1)$, CFL can directly retrieve that $\mathcal{A}_{u_3}^{u_1}(v_4) = \{v_{10}, v_{12}\}$.

CECI. CECI designs an auxiliary data structure $\mathcal{A}$, called the *compact embedding cluster index*. CECI generates and refines candidate vertex sets based on the same observation as CFL, but maintains candidate edges for both tree edges and non-tree edges in q_t. The time complexity and the space complexity are both $O(|E(q)| \times |E(G)|)$.

CECI first generates a BFS tree q_t of q. To differentiate from the matching order, the BFS traversal order is denoted by δ. $\mathcal{A}$ in CECI is built in two phases: (1) the construction and filtering along the order of δ; and (2) the refinement along the reverse order of δ. In the first phase, CECI generates $C(u)$ based on $C(u_p)$ along δ with Generation Rule 3.1, and then prunes $C(u)$ based on $C(u_n)$ along δ with Filtering Rule 3.1 where $u_n \in N_+^\delta(u) - \{u_p\}$ (i.e., the non-tree edge $e(u_n, u)$ such that u_n is before u in δ). Moreover, when constructing and pruning $C(u)$ based on $C(u_p)$ (or $C(u_n)$) at each step, CECI rules out v from $C(u_p)$ (or $C(u_n)$) if v has no neighbors in $C(u)$. In the second phase, CECI refines $C(u)$ based on $C(u_c)$ with Filtering Rule 3.1 along the reverse order of δ where u_c is the child of u. $\mathcal{A}$ in CECI maintains edges between candidates in $C(u)$ and $C(u')$ where $u' \in N_+^\delta(u)$ (i.e., both tree and non-tree edges in q_t). Example 3.3 shows a running example of CECI.

EXAMPLE 3.3. *Given graphs in Figure 1, q_t is depicted by thick lines, and $\delta = (u_0, u_1, u_2, u_3)$. Based on NLF, $C(u_0) = \{v_0\}$. CECI generates $C(u)$ based on $C(u_p)$. The result is shown in Figure 4(a). After that, CECI uses non-tree edges to prune $C(u)$ along δ. Take $e(u_1, u_2)$ as an example: CECI loops over $C(u_2)$ to remove candidates having no neighbors in $C(u_1)$, and rules out candidates in $C(u_1)$ that have no neighbor in $C(u_2)$ as well. As a result, v_6 is removed from $C(u_1)$. Similarly, v_1 is ruled out from $C(u_2)$ when the non-tree edge $e(u_2, u_3)$ is considered. The result after pruning with non-tree edges is shown in Figure 4(b). In the second phase, CECI refines $C(u)$ based on $C(u_c)$ along the reverse order of δ. In this example, the refinement*
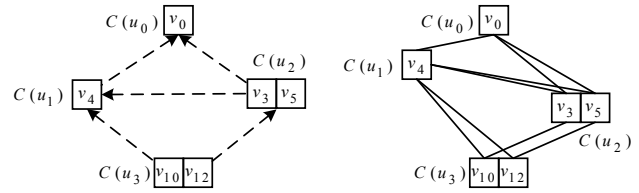
does not update the candidates, and the result is the same as in Figure 4(b). Different from CFL, $\mathcal{A}$ in CECI maintains edges between candidates for each edge in $E(q)$.

DP-iso. DP-iso designs an auxiliary data structure $\mathcal{A}$, called *candidate space*, which maintains edges between candidates in $C(u)$ and those in $C(u')$ if $e(u, u') \in E(q)$. The time complexity and the space complexity are both $O(|E(q)| \times |E(G)|)$. DP-iso first performs a BFS on q from the selected vertex, and the BFS traversal order is denoted by δ. For each $u \in V(q)$, DP-iso generates $C(u)$ with LDF. After that, DP-iso refines $C(u)$ based on Filtering Rule 3.1. In the first phase, DP-iso refines $C(u)$ based on $C(u')$ where $u' \in N_-^\delta(u)$ along the reverse order of δ. In this phase, DP-iso uses NLF to prune invalid candidates as well. In the second phase, DP-iso refines $C(u)$ based on $C(u')$ where $u' \in N_+^\delta(u)$ along the order of δ. DP-iso repeats the k refinement phases by alternating the reverse order of δ and the order δ where k is specified by the user. The original paper sets k to 3. DP-iso records edges between candidates after the refinement. Example 3.4 shows a running example of DP-iso.

EXAMPLE 3.4. *Given graphs in Figure 1, DP-iso obtains $\delta = (u_0, u_1, u_2, u_3)$. The 1st refinement phase prunes candidates along the reverse order of δ. Based on NLF, $C(u_3) = \{v_{10}, v_{12}\}$. Next, DP-iso refines $C(u_2)$ based on $C(u_3)$. The result after the 1st refinement is shown in Figure 5(a) where the dashed line illustrates $C(u)$ is refined based on which candidate vertex sets. In this example, k is set to 1 for simplicity. After the refinement, $\mathcal{A}$ in DP-iso records edges between candidates. The final result is shown in Figure 5(b).*

3.1.2 Analysis

All the filtering methods prune $C(u)$ by deriving some constraints on candidates based on Definition 2.1. The pseudo subgraph isomorphism algorithm in the global refinement of GraphQL can be simplified as the following observation.

OBSERVATION 3.2. *Suppose that for each $u \in V(q)$, $C(u)$ is complete. $S_{u'}$ denotes $N(v) \cap C(u')$ where $v \in C(u)$ and $u' \in N(u)$. If the mapping (u, v) exists in a match from q to G, then v must satisfy (1) for each $u' \in N(u)$, $S_{u'} \neq \emptyset$; and (2) there exists a bag constructed by selecting a vertex from each $S_{u'}$ where $u' \in N(u)$ such that the bag is a set, i.e., contains no duplicate elements.*

Note that Observation 3.2 introduces condition (2) which is not in Observation 3.1. If for any $u, u' \in V(q)$, $C(u) \bigcap C(u') = \emptyset$ (e.g., each query vertex has a distinct label), then Observation 3.2 is equivalent to Observation 3.1, because condition (2) must be satisfied if condition (1) is met.

GraphQL refines $C(u)$ along an order of vertices based on Observation 3.2, whereas CFL, CECI and DP-iso based on Observation 3.1. Ideally, we can repeat refining $C(u)$ to reach a *steady state*, in which for each $v \in C(u)$ and $u \in V(q)$, v satisfies the constraint in Observation 3.1 or 3.2, but this process can be time consuming. Therefore, existing algorithms perform a limited number of refinement iterations to save the time cost. As the methods under study differ on number of refinement iterations and prune $C(u)$ based on different subsets of $N(u)$ at each step (e.g., CECI prunes $C(u)$ based on $C(u_c)$, whereas DP-iso filters $C(u)$ based on $C(u')$ where $u' \in N_-^\delta(u)$ in the first refinement), we conduct experiments to compare their practical performance.

3.2 Ordering Methods

QuickSI. QuickSI proposes an *infrequent-edge first ordering method*. It first converts q into a weighted graph q^w, in which $w(u) = |\{v \in V(G)|L(v) = L(u)\}|$ and $w(e(u,u')) = |\{e(v,v') \in E(G)|L(v) = L(u) \wedge L(v') = L(u')\}|$ where $w(.)$ is the weight of a vertex or an edge. Next, QuickSI selects the edge $e^*(u,u') = arg\min_{e(u,u') \in E(q)} w(e(u,u'))$, and adds u and u' into φ along the ascending order of their weights. Without loss of generality, suppose that u is selected as the first vertex and u' is the second. We set $u'.p$ as u. After that, QuickSI iteratively picks the edge $e^*(u,u') = arg\min_{e(u,u') \in E(q) \wedge u \in \varphi \wedge u' \notin \varphi} w(e(u,u'))$, sets $u'.p$ to u, and adds u' into φ until φ contains all query vertices.

GraphQL. GraphQL proposes a *left-deep join based method*, which models the query as a left-deep join tree in which the leaf nodes are candidate vertex sets. GraphQL first selects $u^* = arg\min_{u \in V(q)} |C(u)|$ as the start vertex of φ. After that, GraphQL iteratively selects $u^* = arg\min_{u \in N(\varphi) - \varphi} |C(u)|$ as the next vertex in φ.

CFL. CFL proposes a *path-based ordering method*, and puts core vertices at the beginning of the matching order. It first picks three core vertices u with min $\frac{|\{v \in V(G)|L(u) = L(v)\}|}{d(u)}$, then selects the root vertex u_r from the three query vertices with minimum $|C(u)|$ where $C(u)$ is generated by NLF, and finally generates a BFS tree q_t of q rooted at u_r. Let $\mathcal{P}$ denote all root-to-leaf paths P in q_t. CFL designs a dynamic programming method to build a weight array W in polynomial time. This method estimates the number of paths in $\mathcal{A}$ that are isomorphic to each path $P \in \mathcal{P}$. P^u is the suffix of P from u where $u \in P$, and $c(P)$ denote the estimated number of paths in $\mathcal{A}$ that are isomorphic to P. CFL first

selects $P^* = arg\min_{P \in \mathcal{P}} \frac{c(P)}{|NT(P)|}$ where $NT(P)$ is the non-tree edges adjacent to vertices in P, adds vertices in P^* to φ along their order in P^*, and removes P^* from $\mathcal{P}$. After that, CFL iteratively picks $P^* = arg\min_{P \in \mathcal{P}} \frac{c(P^u)}{|C(u)|}$ where u is the connection vertex of P to φ, adds vertices in $P^* - \varphi$ to φ, and removes P^* from $\mathcal{P}$.

CECI. CECI uses the BFS traversal order of q as the matching order. CECI first selects $u_r = arg\min_{u \in V(q)} \frac{|C(u)|}{d(u)}$ where $C(u)$ is generated by NLF, and then performs a BFS on q started from u_r to obtain the matching order.

DP-iso. DP-iso proposes an adaptive ordering method that dynamically selects the next query vertex during the enumeration. DP-iso decomposes the query vertices into the set of degree-one vertices and the set V' of the remaining vertices, and prioritizes the vertices in V'. For the simplicity of presentation, we focus on the vertices in V'. DP-iso first generates a BFS traversal order δ of q starting from $u_r = arg\min_{u \in V(q)} \frac{|C(u)|}{d(u)}$ where $C(u)$ is generated by LDF. Next, DP-iso generates a collection of *tree-like* paths P in q according to δ. Specifically, a path P starting from $u \in V(q)$ is called *tree-like* if all vertices u' in P except u satisfy that $N_+^\delta(u') = 1$. A tree-like path P is *maximal* if there is no other tree-like path that has P as a prefix. DP-iso builds a weight array to estimate the number of embeddings in $\mathcal{A}$ that are isomorphic to the maximal tree-like paths. When $u \in V(q)$ becomes extendable given δ and M during the enumeration, DP-iso immediately computes $LC(u, M)$, and selects the next query vertex to be mapped among all extendable vertices based on the weight array and the local candidates.

RI. RI generates φ only based on the structure of q. RI first selects $u^* = arg\max_{u \in V(q)} d(u)$ as the start vertex of φ. After that, RI iteratively selects $u^* = arg\max_{u \in N(\varphi) - \varphi} |N(u) \bigcap \varphi|$ as the next vertex, i.e., RI prefers the vertex with more neighbors in φ. When there are ties in the *max* function, RI breaks ties by considering the following properties of u in order of: (1) the maximum value of $|\{u' \in \varphi | \exists u'' \in V(q) - \varphi, e(u', u'') \in E(q) \wedge e(u, u'') \in E(q)\}|$, i.e., the number of vertices in φ that have a neighbor outside of φ and is adjacent with u; and (2) the maximum value of $|\{u' \in N(u) - \varphi | \forall u'' \in \varphi, e(u', u'') \notin E(q)\}|$, i.e., the number of neighbors of u that are not in φ, and not even adjacent with vertices in φ.

VF2++. VF2++ first picks the vertex $u \in V(q)$ the label of which is least frequently appeared in G but with the largest degree as the root vertex u_r. Next, VF2++ generates a BFS tree q_t of q rooted at u_r. Let $V_i(q_t)$ denote the vertices at depth i in q_t. VF2++ adds query vertices to φ depth-by-depth from depth 0. Specifically, for the vertices in $V_i(q_t)$, VF2++ iteratively selects $u^* = arg\max_{u \in V_i(q_t) - \varphi} |N(u) \bigcap \varphi|$ as the next vertex in φ. VF2++ breaks ties by the properties of u in order of: (1) the largest degree value; and (2) the minimum value of $|\{v \in V(G)|L(u) = L(v)\}|$.

3.3 Enumeration Methods

These algorithms all adopt the recursive enumeration procedure in Algorithm 1 to find all matches, but use different local candidate computation methods.

3.3.1 Overview

QuickSI, RI and VF2++. Algorithm 2 presents the *ComputeLC* function of QuickSI and RI. If the depth $i = 1$, line 1 returns $C(u)$ generated by LDF. Otherwise, lines 3-8 loop over the neighbors of the data vertex that is mapped to $u.p$ to obtain local candidates. Given $v \in N(M[u.p])$, VF2++ designs extra filtering rules to determine whether v can be a candidate of u in addition to the check at lines 4-8 in Algorithm 2. Specifically, let $L(N_-^\varphi(u))$ denote $\{L(u')|u' \in N_-^\varphi(u)\}$, $N_-^\varphi(u, l)$ represents $\{u' \in N_-^\varphi(u)|L(u') = l\}$, and $X(v, l)$ is $\{v' \in N(v)|L(v') = l \land v' \notin M\}$. VF2++ requires v to satisfy that: $\forall l \in L(N_-^\varphi(u)), |N_-^\varphi(u, l)| \leqslant |X(v, l)|$. However, the effectiveness of the additional filtering rules is obtained at the cost of extra computational and storage overhead.

Algorithm 2: ComputeLC of QuickSI and RI

1   **if** $i = 1$ **then return** $\{v \in V(G)|LDF(v, u)\ is\ true\}$;
2   $\Phi \leftarrow \emptyset$;
3   **foreach** $v \in N(M[u.p])$ **do**
4   **if** $LDF(v, u)$ *is true* **then**
5          $flag \leftarrow true$;
6   **foreach** $u' \in N_+^\varphi(u)$ *and* $u' \neq u.p$ **do**
7   **if** $e(v, M[u']) \notin E(G)$ **then** $flag \leftarrow false$, **break**;
8   **if** *flag is true* **then** $\Phi \leftarrow \Phi \bigcup \{v\}$;
9   **return** Φ;

GraphQL. Algorithm 3 describes the *ComputeLC* function of GraphQL. Because GraphQL does not maintain edges between candidates, it has to loop over the entire $C(u)$ to compute local candidates.

Algorithm 3: ComputeLC of GraphQL

1   **if** $i = 1$ **then return** $C(u)$;
2   $\Phi \leftarrow \emptyset$;
3   **foreach** $v \in C(u)$ **do**
4      $flag \leftarrow true$;
5   **foreach** $u' \in N_+^\varphi(u)$ **do**
6   **if** $e(v, M[u']) \notin E(G)$ **then** $flag \leftarrow false$, **break**;
7   **if** *flag is true* **then** $\Phi \leftarrow \Phi \bigcup \{v\}$;
8   **return** Φ;

CFL. Given $u \in V(q)$ and $v \in C(u.p)$, the neighbors of v in $C(u)$ can be directly retrieved from $\mathcal{A}$. Algorithm 4 presents the *ComputeLC* function of CFL.

Algorithm 4: ComputeLC of CFL

1   **if** $i = 1$ **then return** $C(u)$;
2   **if** $|N_+^\varphi(u)| = 1$ **then return** $\mathcal{A}_u^{u.p}(M[u.p])$;
3   $\Phi \leftarrow \emptyset$;
4   **foreach** $v \in \mathcal{A}_u^{u.p}(M[u.p])$ **do**
5   Same with Lines 5-8 in Algorithm 2;
6   **return** Φ;

Algorithm 5: ComputeLC of CECI and DP-iso

1   **if** $i = 1$ **then return** $C(u)$;
2   **if** $|N_+^\varphi(u)| = 1$ **then return** $\mathcal{A}_u^{u.p}(M[u.p])$;
3   **return** $\bigcap_{u' \in N_+^\varphi(u)} \mathcal{A}_u^{u'}(M[u'])$;

CECI and DP-iso. CECI and DP-iso maintain the edges between candidates for all edges in $E(q)$. Algorithm 5 illustrates the *ComputeLC* function of CECI and DP-iso. With the assistance of $\mathcal{A}$, Algorithm 5 performs set intersections to compute the local candidates.

3.3.2 Analysis

Let α denote $|N_+^\varphi(u)|$, β denote the cost of verifying whether $e(v, v')$ belongs to $E(G)$ and d_G denote the average degree of G. The cost of the LDF check is omitted, as the label and degree of a vertex can be directly obtained. We store G as the *compressed sparse row* (CSR) where the neighbor set of a vertex is a sorted array, and adopt the *binary search* to check the existence of an edge. Hence, β can be estimated by $O(\log d_G)$. We implement a hybrid set intersection method: if the cardinalities of two sets are similar, we use the merge-based method; otherwise, we adopt the Galloping algorithm [1]. The cost of the set intersection is proportional to the cardinality of the smallest set. Algorithm 2 loops over $N(M[u.p])$ where $|N(M[u.p])|$ can be estimated by d_G. Then, the cost of Algorithm 2 is $O(d_G)$ when $\alpha = 1$, while $O(d_G \times (\alpha - 1) \times \beta)$. The cost of Algorithm 3 is $O(|C(u)| \times \alpha \times \beta)$, since GraphQL does not construct $\mathcal{A}$ to maintain edges between candidates. When $\alpha = 1$, the cost of Algorithms 4 and 5 is $O(1)$, because they can directly return results based on the auxiliary data structures. When $\alpha > 1$, the cost of Algorithm 4 is $O(|N(M[u.p]) \bigcap C(u)| \times (\alpha - 1) \times \beta)$. Algorithm 5 computes the local candidates based on set intersections, the cost of which can be estimated by $O(\min_{u' \in N_+^\varphi(u)} |N(M[u']) \bigcap C(u)| \times (\alpha - 1))$.

Through the analysis, we can see that constructing auxiliary data structures to maintain edges between candidates can significantly improve the efficiency of the local candidate computation, especially when there is only one backward neighbor. Overall, Algorithm 5 is the most efficient.

3.4 Optimization Methods

Graph Compression. TurboIso [19] compressed the query graph, and BoostIso [42] compressed the data graph. The authors of CFL studied the effects of the two compression techniques in detail, and found that (1) the data graph compression technique worked well only when the data graph was very dense; (2) only a small number of query vertices could be compressed by the query graph compression method; and (3) CFL significantly outperformed the algorithms that adopt the compression techniques [7]. Therefore, we do not study the compression techniques in this paper.

(a) Query graph q.   (b) Search tree.

Figure 6: Running example of failing sets pruning.

Failing Sets Pruning. DP-iso proposed the *failing sets pruning* method, which utilizes the information obtained from the exploration of the search subtree rooted at a partial result M to rule out some invalid partial results, especially the siblings of M in the search tree. We find that this optimization method cannot only work with DP-iso but also other algorithms. Therefore, we further evaluate the effectiveness of the failing sets pruning technique on different algorithms in our experiments. The following is a running example of the failing sets pruning technique.

EXAMPLE 3.5. *Given q in Figure 6(a) and G in Figure 1(b), suppose the matching order φ is $(u_0, u_1, u_2, u_3, u_4)$. Figure 6(b) visualizes a part of the search tree in which circles and edges denote intermediate results and mappings between query vertices to data vertices respectively. M is extended from M' by mapping u_2 to v_{10}. The search tree is explored in a depth-first search order. After the exploration of the subtree rooted at M, we know there is no match in the subtree, because v_0 is already in M. Moreover, the failure is not relevant to u_2, because it does not involve in the conflict. Consequently, extending M' by mapping u_2 to other candidates cannot lead to matches as well. Then, we can skip the sibling of M (i.e., the nodes in the rectangle) in the search tree to accelerate the exploration.*

3.5 Glasgow Algorithm

Glasgow [4] models subgraph matching as a *constraint programming* problem in which vertices and edges in q correspond to variables and constraints respectively. It recursively maps query vertices to data vertices to find all results. Glasgow first obtains a candidate vertex set (i.e., *domain*) for $u \in V(q)$ based on the degrees of $u' \in N(u)$, but does not maintain edges between candidates. It does not generate φ in advance of the enumeration, but determines which query vertex will be mapped next during the search. Specifically, at a recursive search call, it picks the query vertex u not mapped but with the minimum number of candidates as the next. As Glasgow is optimized for deciding whether G contains q, it prioritizes the candidate v with a large degree to map to u first. When mapping u to v, Glasgow conducts inference based on the new mapping to eliminate invalid candidates. However, Glasgow maintain a number of status during the search, which consumes a large amount of memory.

Table 3: Properties of real-world datasets.

Category	Dataset	Name	$|V|$	$|E|$	$|\Sigma|$	d
Biology	Yeast	*ye*	3,112	12,519	71	8.0
	Human	*hu*	4,674	86,282	44	36.9
	HPRD	*hp*	9,460	34,998	307	7.4
Lexical	WordNet	*wn*	76,853	120,399	5	3.1
Citation	US Patents	*up*	3,774,768	16,518,947	20	8.8
Social	Youtube	*yt*	1,134,890	2,987,624	25	5.3
	DBLP	*db*	317,080	1,049,866	15	6.6
Web	eu2005	*eu*	862,664	16,138,468	40	37.4

4 EXPERIMENTAL SETUP

Techniques under study. We study QuickSI (QSI), GraphQL (GQL), CFL, CECI, DP-iso (DP), RI, VF2++ (2PP) and Glasgow (GLW) in our experiments.

Implementation. We obtained the source code of QuickSI and GraphQL from the authors of BoostIso [42] and the source code of CFL [7], RI [8], VF2++ [23] and Glasgow [4] from their original authors respectively. We carefully examined the obtained source code and re-implemented CFL, RI and VF2++ within the framework in Algorithm 1. Currently, the source code of CECI and DP-iso is not publicly available. We contacted the authors of CECI [6] through email and re-implemented a sequential version of CECI under their detailed instructions. We re-implemented DP-iso based on the original paper [17] and the author's thesis [16]. All algorithms were implemented in C++.

Experiment Environment. The code was complied by g++ 4.9.2. We conducted experiments on a Linux machine with two Intel Xeon E5-2670 v3 CPUs and 128GB RAM.

Data graphs. We used both real-world and synthetic datasets to evaluate competing algorithms.

Real-world datasets. We selected eight real-world datasets from five categories, including datasets used in previous work [6, 17, 19, 21, 26, 33, 42, 45, 53, 59] as well as new types of graphs such as the web network. Table 3 lists the properties of the real-world datasets. Yeast, Human, HPRD and WordNet contain labels. For the unlabeled datasets, we followed the method used in previous work [7, 17], which randomly chooses a label from a label set Σ and assigns the label to the vertex. We tried different sizes of label sets and picked those with which a reasonable number of queries completed within time limit.

Synthetic datasets. We generated synthetic datasets with the RMAT [12] model, which generates power-law graphs. We set the four parameters of RMAT as $a = 0.45, b = 0.22, c = 0.22$ and $d = 0.11$. We randomly assigned distinct labels to vertices. We varied $|\Sigma|$ from 1 to 64, $|V|$ from $0.1M$ (million) to $64M$ and d from 4 to 32 respectively to find the "sane default" such that we can demonstrate the capabilities of the competing algorithms without breaking most of them. Based on our experiments, the default configuration is $|V| = 1M$, $d = 16$ and $|\Sigma| = 16$. We varied $|V|$, $|E|$ and $|\Sigma|$ respectively to examine the scalability of competing techniques.

- Vary $|V|$: We generate 4 data graphs with the number of vertices as $1M$, $4M$, $16M$ and $64M$ respectively.
- Vary d: We generate 4 data graphs with the degree as 8, 12, 16 and 20 respectively.
- Vary $|\Sigma|$: We generate 4 data graphs with the number of distinct labels as 8, 12, 16 and 20 respectively.

Query graphs. We generated query graphs for each data graph G by randomly extracting subgraphs from G to keep consistent with previous research [4, 6, 7, 17, 19, 26, 42, 53]. Specifically, for each dataset, we generated nine query sets each of which contains 200 **connected** query graphs with the same number of vertices. We vary $|V(q)|$ from 4 to 20 for Human and WordNet, but from 4 to 32 for other datasets, because Human is very dense and most vertices in WordNet have the same label, which makes them two very challenging. Except query graphs with $|V(q)| = 4$, we generated four dense query sets $(d(q) \geqslant 3)$ and four sparse query sets $(d(q) < 3)$. Q_{iD} and Q_{iS} denote the dense and sparse query sets containing query graphs with i vertices respectively. To generate q with specified configuration (e.g., $|V(q)| = 8$ and $d(q) \geqslant 3$), we perform a random walk on G until getting the specified number of vertices and extract the induced subgraph to check whether the density satisfies the requirement. If so, we add it to the query set. Otherwise, we conduct a new random walk. Table 4 lists the query sets for each data graph. Due to space limit, we present the experiment results of Human and WordNet on Q_{20D} and Q_{20S}, and other datasets on Q_{32D} and Q_{32S} as representatives by default.

Table 4: Datasets and query sets.

Dataset	Query Set	Default
Yeast, HPRD, US Patents, Youtube, DBLP, eu2005	Q_4, Q_{8D}, Q_{16D}, Q_{24D}, Q_{32D}, Q_{8S}, Q_{16S}, Q_{24S}, Q_{32S}	Q_{32D}, Q_{32S}
Human, WordNet	Q_4, Q_{8D}, Q_{12D}, Q_{16D}, Q_{20D}, Q_{8S}, Q_{12S}, Q_{16S}, Q_{20S}	Q_{20D}, Q_{20S}

Metrics. We measured the time in milliseconds (ms) to process individual query in a query set, which consists of the *preprocessing time* (i.e., the time spent on filtering vertices, building auxiliary data structures and generating matching orders) and the *enumeration time* (i.e., the time spent on enumerating results). Furthermore, in order to compare the filtering methods, we examined the *number of candidate vertices*, which is $\frac{1}{|V(q)|} \sum_{u \in V(q)} |C(u)|$, and the *memory cost* on the candidate vertices as well as the auxiliary data structures. Given q and G, there can be a large number of matches. Following CFL and DP-iso, we terminated the query after finding 10^5 matches to cover as much search space as time allowed. We killed a query if it cannot complete within five minutes (3×10^5 ms), so that our experiments can finish in reasonable time. For the purpose of comparison, we recorded the enumeration time of killed queries as five minutes. We call a killed query an *unsolved query*. To evaluate an algorithm on a query set, we report the average value for the
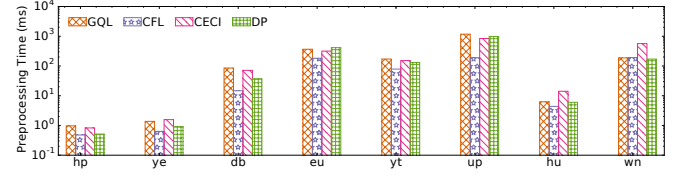
(a) Vary datasets on dense queries.
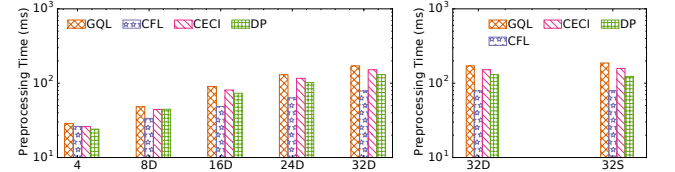
(b) Vary $|V(q)|$ on *yt*.　　**(c) Q_{32D} and Q_{32S} on *yt*.**

Figure 7: Efficiency of filtering methods.

metrics such as the preprocessing time, the enumeration time and the number of candidate vertices. For the enumeration time, we report the standard deviation as well.

5 EXPERIMENT RESULTS

In this section, we evaluate the performance of competing algorithms.

5.1 Evaluating Filtering Methods

In this subsection, we evaluate the filtering methods of GQL, CFL, CECI and DP.

Preprocessing Time. Figure 7(a) presents the preprocessing time on different real-world datasets. All algorithms spend more time on large datasets such as *up*. Because GQL has a higher time complexity, it generally runs slower than CFL. Although CECI and DP have the same time complexity as CFL, they spend more time than CFL, since (1) CECI has to synchronize different copies of $C(u)$; (2) DP performs three iterations of refinement, whereas CFL only performs a bottom-up refinement; and (3) CECI and DP preserve edges between candidates for non-tree edges in q_t. As shown in Figure 7(b), the preprocessing time grows when $|V(q)|$ increases. In Figure 7(c), the difference between dense queries and sparse queries on preprocessing time is small. Overall, the absolute value of the preprocessing time is small.

Number of Candidate Vertices. Figure 8 compares the pruning power of filtering methods. Besides competing algorithms, we illustrate the results of two baseline methods: (1) $C(u)$ generated by LDF; and (2) $C(u)$ in the steady state (denoted by STEADY) based on Filtering Rule 3.1. The performance of competing methods is close to LDF on *wn*, because most vertices (more than 80%) in *wn* have the same label. GQL outperforms other algorithms on *wn* because of its stronger filtering rule. However, GQL does not dominate the other algorithms on the other datasets, because (1) it refines $C(u)$ along a random order; and (2) these datasets have a number of distinct labels. CECI performs worse than CFL
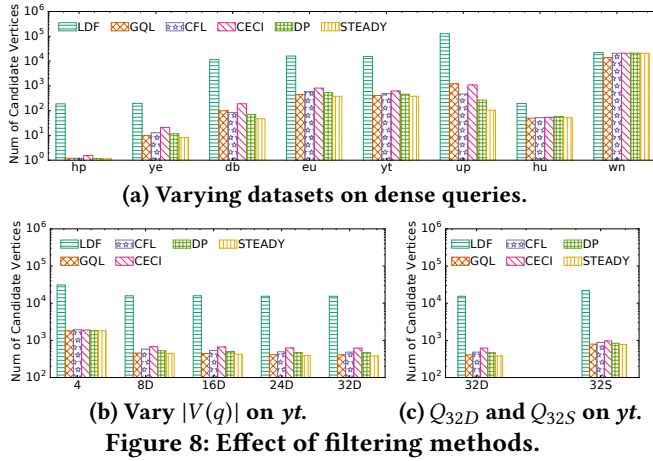
(a) Varying datasets on dense queries.

(b) Vary $|V(q)|$ on *yt*.

(c) Q_{32D} and Q_{32S} on *yt*.

Figure 8: Effect of filtering methods.

and DP, because the refinement of CECI prunes $C(u)$ based on u_c in q_t, whereas CFL and DP take more neighbors of u into consideration. DP is slightly better than CFL, since it conducts more refinement. Although CFL and DP only perform a small number of refinements, their performance is generally close to STEADY. In Figure 8(b), the number of candidate vertices on Q_4 is more than other query sets, because query graphs in Q_4 are sparse. $|C(u)|$ varies slightly with $|V(q)|$ varied from 8 to 32. There are more candidates on sparse queries than dense queries in Figure 8(c), because query vertices in dense queries have more neighbors. Except that GQL outperforms other methods when most of data vertices have the same label, GQL, CFL and DP are competitive with each other on other datasets.

5.2 Evaluating Enumeration Methods

We optimize the local candidate computation of QSI, GQL, CFL and 2PP as follows: (1) maintain edges between candidates for all edges in $E(q)$; (2) adopt Algorithm 5 to compute local candidates; and (3) remove the extra filtering rules in 2PP. After that, we examine the average speedup of the enumeration time achieved with the optimization for each algorithm. We omit RI, because RI has the same local candidate computation with QSI. We use $C(u)$ generated by LDF as the candidate vertex set of QSI and 2PP. Figures 9 shows the speedup achieved with the optimization. The speedup on *hp* is limited, because the enumeration time on *hp* is very short. Although CFL has maintained edges between candidates for tree-edges in q_t, the optimization still achieves 1.3-4.8X average speedup. GQL and 2PP achieve several orders of magnitude speedup with the optimization. The experiment results demonstrate the superiority of Algorithm 5, which indicates the necessity of maintaining edges between candidates for all edges in $E(q)$. The speedup of 2PP shows that the overhead of extra filtering rules exceeds the benefit.

We further examine whether we can improve the enumeration by adopting more recent set intersection method.
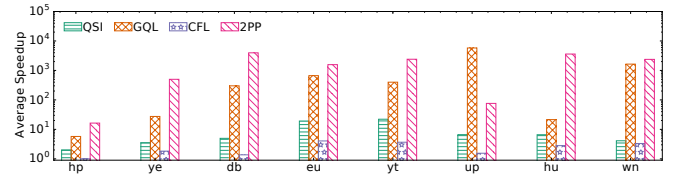

Figure 9: Effect of the set intersection based local candidate computation on the enumeration time.
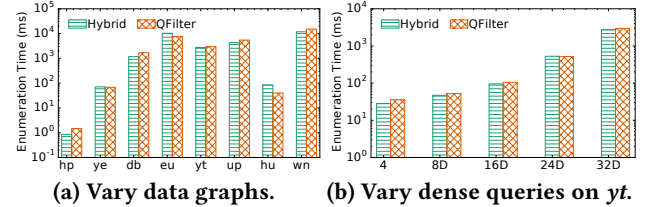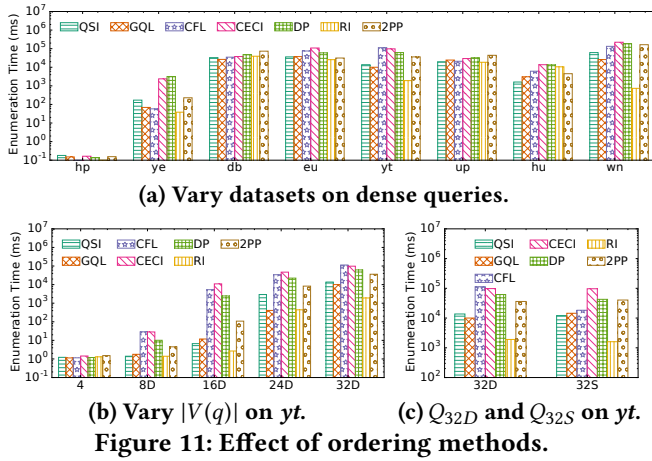

(a) Vary data graphs.   **(b) Vary dense queries on *yt*.**

Figure 10: Comparison of set intersection methods.

Specifically, we compare the hybrid method, denoted by *Hybrid*, on sorted integer arrays provided by EmptyHeaded [1] and *QFilter* [18], which encodes neighbor sets in a compact layout. Figure 10 presents the enumeration time of the optimized GQL algorithm with *Hybrid* and *QFilter* respectively. *QFilter* outperforms *Hybrid* on the dense graphs *eu* and *hu*, because its compact layout allows each instruction to manipulate more elements than *Hybrid*. However, the overhead of the filter step as well as the compact layout makes *QFilter* underperform *Hybrid* on the sparse graphs.
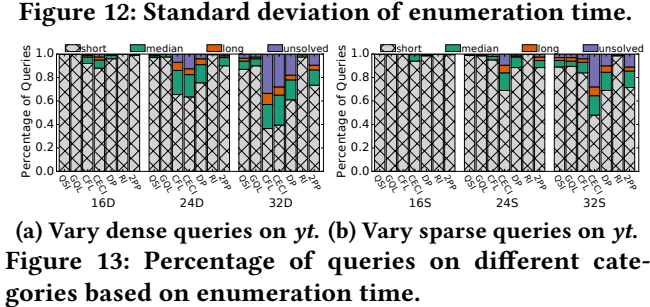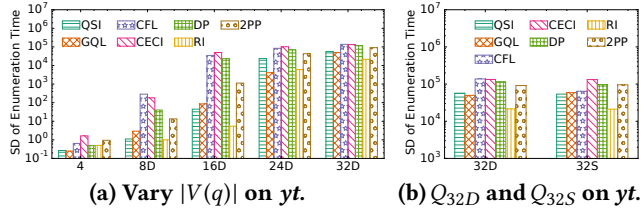
5.3 Evaluating Ordering Methods

In this subsection, we compare the ordering methods of QSI, GQL, CFL, CECI, DP, RI and 2PP. In order to eliminate the impact of different local candidate computation methods on the performance, our experiments use the optimized version of QSI, GQL, CFL, RI and 2PP (see Section 5.2). Because $C(u)$ generated by LDF have much more candidates than that obtained by advanced filtering methods, we use $C(u)$ generated by the filtering method of GQL as the candidate vertex set of QSI, RI and 2PP to make a fair comparison of the ordering methods. Additionally, we disable the failing sets pruning technique in DP-iso.

Enumeration Time. Figure 11(a) shows the enumeration time on different datasets. The enumeration time on *hp* is very short, because of the small number of candidates. GQL and RI outperform the latest algorithms. GQL runs faster than RI on *hu*, but slower on *yt* and *wn*. Figure 11(b) presents the enumeration time on *yt* with $|V(q)|$ varied. The enumeration time of all methods increases with $|V(q)|$. CFL performs much better on sparse queries than dense queries in Figure 11(c). Figure 12 presents the standard deviation (*SD*) of the enumeration time on *yt*. The large *SD* value indicates that the enumeration time on different queries in a query set can vary greatly. In order to gain more insight into the performance of competing algorithms on individual

(a) Vary datasets on dense queries.

(b) Vary $|V(q)|$ on yt.　　**(c) Q_{32D} and Q_{32S} on yt.**

Figure 11: Effect of ordering methods.

queries, we further report the percentage of short ($t < 1$ second), median ($1 \leqslant t < 60$ seconds), long ($60 \leqslant t < 300$ seconds) and unsolved queries respectively in Figure 13. The competing algorithms answer each query in Q_4, Q_{8D} and Q_{8S} within 1 second. We omit the result. There are more median/long/unsolved queries with the increase of $|V(q)|$. RI completes more than 95% queries within 1 second on Q_{32D} and Q_{32S}, which significantly outperforms other algorithms.

(a) Vary $|V(q)|$ on yt.　　**(b) Q_{32D} and Q_{32S} on yt.**

Figure 12: Standard deviation of enumeration time.

(a) Vary dense queries on yt. (b) Vary sparse queries on yt.

Figure 13: Percentage of queries on different categories based on enumeration time.

Number of Unsolved Queries. Table 5 lists the total number of unsolved queries of competing methods on yt, up, hu and wn. Each dataset has 1800 queries. If a query cannot be completed within the time limit by any competing algorithms, it is called as a *fail-all* query. wo/fs and w/fs represent the algorithms without/with the failing sets pruning technique. The experiment setting in this subsection is wo/fs. RI has fewer unsolved queries than the other algorithms on yt, up and wn, which are sparse datasets, but has more unsolved queries on hu. The gap between the number of fail-all queries and that of each algorithm shows that each

algorithm can generate ineffective ordering on some queries that can be solved by other algorithms.

Table 5: Number of unsolved queries.

Algorithm	yt		up		hu		wn	
	wo/fs	w/fs	wo/fs	w/fs	wo/fs	w/fs	wo/fs	w/fs
QSI	14	0	26	9	12	6	69	20
GQL	11	0	23	8	10	2	17	3
CFL	95	6	24	12	16	8	191	139
CECI	161	5	39	7	40	9	547	351
DP	70	6	40	13	30	20	307	221
RI	2	0	18	8	23	9	0	0
2PP	49	3	49	17	12	7	270	220
Fail-All	**0**	**0**	**7**	**3**	**2**	**0**	**0**	**0**

Spectrum Analysis. We conduct the *spectrum analysis* to study whether the matching orders generated by competing algorithms can be further improved. Specifically, we permutate $V(q)$ to generate 1000 matching orders φ to evaluate the query. We pick a dense query and a sparse query from the given dataset, which are denoted by q_{iD} and q_{iS} respectively where i is the number of vertices. We set the time limit for executing q on G with one generated order as 1 minute, and omit its result if it cannot complete within the time limit. Figure 14 presents the spectrum analysis on the selected queries, and the enumeration time of GQL and RI. The blue point is the enumeration time with our generated orders. As shown in the figure, we can find matching orders that can significantly reduce the enumeration time.
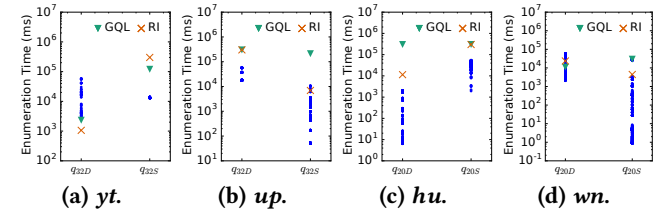

(a) yt.　　**(b) up.**　　**(c) hu.**　　**(d) wn.**

Figure 14: Spectrum analysis of ordering methods.

Table 6: Speedup with generated matching orders.

Algorithm	Q_{32D} on yt				Q_{32S} on yt			
	mean	std	max	> 10	mean	std	max	> 10
GQL	4613	25948	214285	48	12536	51078	300000	42
RI	1067	13272	187500	21	1441	19255	272727	3

With the same method, we analyze the performance of GQL and RI for each query in Q_{32D} and Q_{32S} on yt. For comparison, we also measure the performance of the other algorithms under study as well as the performance of the 1000 randomly sampled matching orders. Then for each query, we compute the speedup of the best performance of all these matching orders over GQL and RI respectively. Table 6 presents the maximum, the standard deviation, and the mean of the speedups of the 200 queries. It also lists the number of queries with a speedup of more than 10 times, denoted as ">10". As shown in the table, both GQL and RI can generate ineffective matching orders, with more than 40 queries on GQL and 3-21 queries on RI performing ten times worst than the best matching order.

Discussion. Given q and φ, q_t is constructed as follows: $\forall u \in \varphi[2 : |\varphi|], u.p \in N_+^\varphi(u)$. The *non-tree edges* are denoted by $\overline{E(q_t)}$. Given $u' \in N_+^\varphi(u)$ where $u' \neq u.p$, $e(u', u)$ must belong to $\overline{E(q_t)}$. We find that the non-tree edges tend to appear at the front of φ generated by GQL and RI, which can terminate the invalid search path at an early stage based on Algorithm 5. Prioritizing non-tree edges is more effective when datasets are sparse, because given two vertices, they have fewer common neighbors. Therefore, RI performs very well on sparse data graphs, as the ordering method of RI picks the query vertices with more backward neighbors at each step. However, the benefit of the strategy becomes smaller when the data graph is very dense. Consequently, RI performs poorly on dense graphs such as *hu*, because the ordering method of RI does not consider any statistics of the data graph. The ordering method of GQL is based on the size of candidate vertex sets, which still works on very dense graphs. The path-based ordering methods in CFL and DP result in a number of unsolved queries, because they put low priorities on the edges between paths when estimating cost. Overall, GQL and RI outperform other methods. RI performs well on sparse data graphs, but worse on dense ones. Nevertheless, all the ordering methods can generate ineffective matching orders.

5.4 Evaluating Optimization Methods

We first evaluate the effectiveness of the failing sets pruning method on DP, which proposed the optimization, and then examine its effect on other algorithms. The algorithms under study have the same settings with that in Section 5.3. Figure 15 presents the effect of the failing sets pruning method on the enumeration time, in which wo/fs and w/fs represent the algorithm without/with the optimization respectively. w/fs performs worse than wo/fs on Q_4 and Q_{8D} in Figure 15(a). In contrast, w/fs can speed up DP by up to one order of magnitude on large queries. This is because there are more invalid search paths for large queries that can be terminated earlier by the optimization. In Figure 15(b), the optimization can speed up each algorithm by orders of magnitude on *yt*. As shown in Table 5, the optimization significantly reduces the number of unsolved queries for each algorithm

5.5 Overall Performance

In this subsection, GQL and RI have the same settings with that in Section 5.3. GQLfs and RIfs enable the failing set pruning. We compare them with the original implementation of CECI, DP-iso, RI, VF2++ and Glasgow (GLW). CECI and DP-iso are the latest algorithms in the database community, and the other three are efficient algorithms from other communities. We denote them O-CECI, O-DP, O-RI and O-2PP respectively to differentiate with the optimized
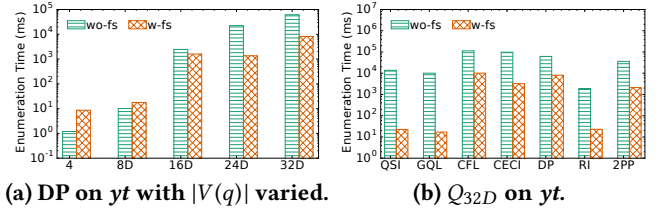
(a) DP on *yt* with $|V(q)|$ varied.   (b) Q_{32D} on *yt*.

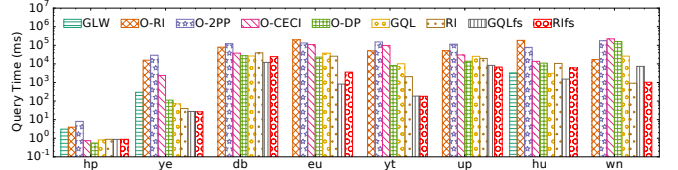**Figure 15: Effect of failing sets pruning.**

Figure 16: Overall performance.

methods used in previous sections. Figure 16 presents the overall performance of the competing algorithms. GLW only works on *hp*, *ye* and *hu*, but runs out of memory on other datasets. O-DP significantly outperforms O-RI, O-2PP and O-CECI, while runs slower than GQLfs and RIfs because of its ineffective ordering methods. GQLfs performs better than RIfs on dense datasets (e.g., *eu* and *hu*), but worse on very sparse datasets (e.g, *yt* and *wn*). The algorithms after our optimization significantly outperform the state-of-the-art algorithms from different communities.

5.6 Scalability Evaluation

In this subsection, we evaluate the scalability of GQLfs and RIfs. We require them to find all results and examine the effect of different properties of datasets on the search space size. Figures 17 present the experiment results of Q_{16D} on synthetic datasets with $d(G)$, $|\Sigma|$ and $|V(G)|$ varied respectively. We estimated the number of results by computing the average number of results in the solved queries. If there are more than 50% unsolved queries, we discard the results of the query time and the number of results. When the data graph is very sparse or has many labels, queries take short time, because there are fewer results. In contrast, there are a lot of unsolved queries when the data graph is dense or $|\Sigma|$ is small. Compared with $|V(G)|$, the algorithms are much more sensitive to $|\Sigma|$ and $d(G)$. The memory cost of the auxiliary data structure is less than 500MB.

Figure 18 shows the experiment results of Q_{16D} on the *friendster* dataset with 124 million vertices and 1.8 billion edges. We randomly assign 64 distinct labels to vertices and vary the density by randomly selecting 40%, 60% and 80% edges. We vary $|\Sigma|$ from 64, 96, 128 to 160. As shown in the figure, the query time is short when the dataset is sparse and there are a number of labels, since the number of results significantly decreases with the increase of $|\Sigma|$ or the decrease of density. We omit the results on the number of unsolved queries and the number of results due to space limit.
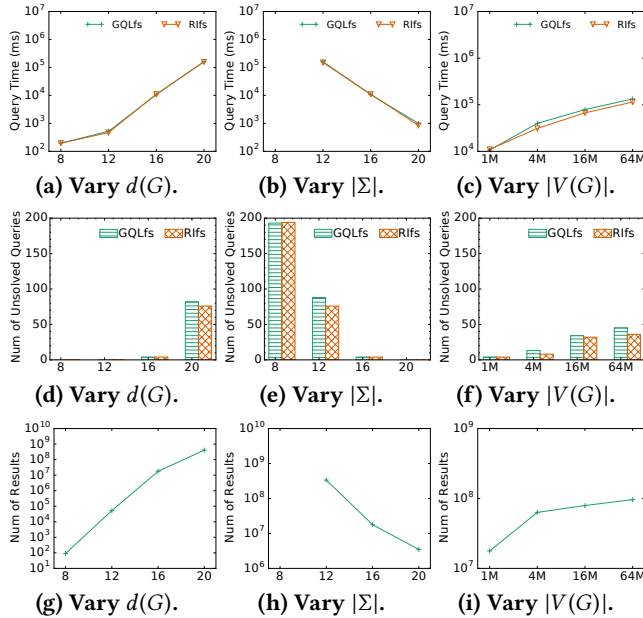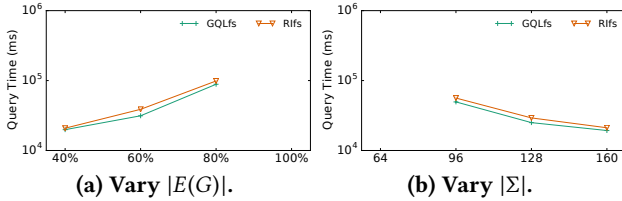
(a) Vary $d(G)$.   **(b) Vary $|\Sigma|$.**   **(c) Vary $|V(G)|$.**

(d) Vary $d(G)$.   **(e) Vary $|\Sigma|$.**   **(f) Vary $|V(G)|$.**

(g) Vary $d(G)$.   **(h) Vary $|\Sigma|$.**   **(i) Vary $|V(G)|$.**

Figure 17: Scalability evaluation on synthetic datasets.

(a) Vary $|E(G)|$.   **(b) Vary $|\Sigma|$.**

Figure 18: Scalability evaluation on *friendster* dataset.

6   CONCLUSIONS

In this paper, we study the performance of eight subgraph matching algorithms. We compare and analyze each individual technique in selected algorithms, and conduct experiments on both real-world and synthetic datasets to evaluate their effectiveness. We summarize our findings as follows.

Comparison with results in previous research. 1. Our experiment results confirm the observation in previous studies that (1) there is no algorithm that dominates other algorithms on all queries, and the ordering methods can generate some ineffective matching orders [33]; and (2) the performance of different algorithms can vary greatly on a query, and the run time of different queries in a query set can vary greatly as well [26]. 2. The latest algorithms such as CFL, CECI, and DP-iso state that they achieved great overall performance improvement due to effective filtering methods and ordering methods. However, our experiment results show that the filtering method of GraphQL is competitive with that of latest algorithms, and that the ordering methods of the latest algorithms perform worse than the earlier algorithms such as QuickSI and GraphQL. In contrast to previous results, we find that the latest algorithms outperform GraphQL in

the overall time, because they use auxiliary data structures to maintain edges between candidates so that they significantly improve the efficiency of the local candidate computation. 3. we confirm that the preprocessing-enumeration algorithms outperform the direct-enumeration algorithms such as RI and VF2++. We attribute the performance improvement to the following two reasons: (1) the candidate vertex sets generated by the filtering methods can provide more accurate information for ordering methods; and (2) the auxiliary data structure significantly improves the efficiency of the local candidate computation.

Effectiveness of techniques in each category. 1. The filtering methods of GraphQL, CFL and DP-iso perform better than CECI in terms of the pruning power. As the filtering method of GraphQL has a higher time complexity than CFL and DP-iso, it runs slower. The pruning power of all filtering methods is sensitive to the size of the label set. When most of vertices have the same label, GraphQL performs better than DP-iso and CFL because of its filtering rule. Overall, the three methods are generally competitive with each other. 2. The ordering methods of GraphQL and RI are usually the most effective among the competing ordering methods, because they tends to put the non-tree edges at the front of the matching order. RI performs very well on sparse datasets, but poorly on very dense ones, because it does not consider the statistics of the data graph. The path-based ordering methods in CFL and DP-iso can result in a large number of unsolved queries, and the adaptive ordering does not dominate the static ordering in our experiments. 3. The local candidate computation method affects the enumeration performance greatly, and the set intersection based method performs the best among competing methods. Therefore, it is necessary to build auxiliary data structures to maintain edges between candidates. 4. The failing sets pruning technique can slow down the performance on small queries, but can significantly improve the performance on large queries and reduce the number of unsolved queries.

Recommendation. 1. Use the candidate vertex computation method of GraphQL as default. If the preprocessing time often dominates the query time, then switch to the method of CFL or DP-iso. 2. Adopt the ordering methods of GraphQL and RI on dense and sparse data graphs respectively. 3. Use CECI/DP-iso-style auxiliary data structures to maintain edges between candidates, and adopt the set intersection based local candidates computation. If the data graphs are very dense, then use QFilter as the set intersection method. 4. Enable the failing sets pruning on large queries, but disable it on small ones. By integrating the recommended techniques in each category into Algorithm 1, we can obtain an optimized method that outperforms the state-of-the-art algorithms such as DP-iso.

REFERENCES

[1] Christopher R Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. Emptyheaded: A relational engine for graph processing. In *TODS*.

[2] Foto N Afrati, Dimitris Fotakis, and Jeffrey D Ullman. 2013. Enumerating subgraph instances using map-reduce. In *ICDE*.

[3] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. 2018. Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows. In *PVLDB*.

[4] Blair Archibald, Fraser Dunlop, Ruth Hoffmann, Ciaran McCreesh, Patrick Prosser, and James Trimble. 2019. Sequential and parallel solution-biased search for subgraph algorithms. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*.

[5] Molham Aref, Balder ten Cate, Todd J Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L Veldhuizen, and Geoffrey Washburn. 2015. Design and implementation of the LogicBlox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*.

[6] Bibek Bhattarai, Hang Liu, and H Howie Huang. 2019. Ceci: Compact embedding cluster index for scalable subgraph matching. In *SIGMOD*.

[7] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient subgraph matching by postponing cartesian products. In *SIGMOD*.

[8] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis Shasha, and Alfredo Ferro. 2013. A subgraph isomorphism algorithm and its application to biochemical data. In *BMC bioinformatics*.

[9] Vincenzo Carletti, Pasquale Foggia, Pierluigi Ritrovato, Mario Vento, and Vincenzo Vigilante. 2019. A Parallel Algorithm for Subgraph Isomorphism. In *International Workshop on Graph-Based Representations in Pattern Recognition*.

[10] Vincenzo Carletti, Pasquale Foggia, Alessia Saggese, and Mario Vento. 2017. Challenging the time complexity of exact subgraph isomorphism for huge and dense graphs with VF3. In *IEEE transactions on pattern analysis and machine intelligence*.

[11] Vincenzo Carletti, Pasquale Foggia, and Mario Vento. 2015. VF2 Plus: An improved version of VF2 for biological graphs. In *International Workshop on Graph-Based Representations in Pattern Recognition*.

[12] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A Recursive Model for Graph Mining.. In *SDM*.

[13] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (sub) graph isomorphism algorithm for matching large graphs. In *TPAMI*.

[14] Rosalba Giugno, Vincenzo Bonnici, Nicola Bombieri, Alfredo Pulvirenti, Alfredo Ferro, and Dennis Shasha. 2013. Grapes: A software for parallel searching on biological graphs targeting multi-core architectures.

[15] Georg Gottlob, Martin Grohe, Nysret Musliu, Marko Samer, and Francesco Scarcello. 2005. Hypertree decompositions: Structure, algorithms, and applications. In *International Workshop on Graph-Theoretic Concepts in Computer Science*.

[16] Myoungji Han. 2018. An Efficient Algorithm for Subgraph Isomorphism using Dynamic Programming on Directed Acyclic Graphs. In *Thesis*.

[17] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. 2019. Efficient Subgraph Matching: Harmonizing Dynamic Programming, Adaptive Matching Order, and Failing Set Together. In *SIGMOD*.

[18] Shuo Han, Lei Zou, and Jeffrey Xu Yu. 2018. Speeding Up Set Intersections in Graph Algorithms using SIMD Instructions. In *SIGMOD*.

[19] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. Turbo iso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *SIGMOD*.

[20] Huahai He and Ambuj K Singh. 2006. Closure-tree: An index structure for graph queries. In *ICDE*.

[21] Huahai He and Ambuj K Singh. 2008. Graphs-at-a-time: query language and access methods for graph databases. In *SIGMOD*.

[22] Ho Hoang Hung, Sourav S Bhowmick, Ba Quan Truong, Byron Choi, and Shuigeng Zhou. 2014. QUBLE: towards blending interactive visual subgraph search queries on large networks. In *VLDBJ*.

[23] Alpár Jüttner and Péter Madarasi. 2018. VF2++: An improved subgraph isomorphism algorithm. In *Discrete Applied Mathematics*.

[24] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An active graph database. In *SIGMOD*.

[25] Foteini Katsarou, Nikos Ntarmos, and Peter Triantafillou. 2015. Performance and scalability of indexed subgraph query processing methods. In *PVLDB*.

[26] Foteini Katsarou, Nikos Ntarmos, and Peter Triantafillou. 2017. Subgraph querying with parallel use of query rewritings and alternative algorithms. In *EDBT*.

[27] Hyeonji Kim, Juneyoung Lee, Sourav S Bhowmick, Wook-Shin Han, JeongHoon Lee, Seongyun Ko, and Moath HA Jarrah. 2016. DUAL-SIM: Parallel Subgraph Enumeration in a Massive Graph on a Single Machine. In *SIGMOD*.

[28] Raphael Kimmig, Henning Meyerhenke, and Darren Strash. 2017. Shared Memory Parallel Subgraph Enumeration. In *IPDPSW*.

[29] Karsten Klein, Nils Kriege, and Petra Mutzel. 2011. CT-index: Fingerprint-based graph indexing combining cycles and trees. In *ICDE*.

[30] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. 2015. Scalable subgraph enumeration in MapReduce. In *PVLDB*.

[31] Longbin Lai, Lu Qin, Xuemin Lin, Ying Zhang, Lijun Chang, and Shiyu Yang. 2016. Scalable distributed subgraph enumeration. In *PVLDB*.

[32] Longbin Lai, Zhu Qing, Zhengyi Yang, Xin Jin, Zhengmin Lai, Ran Wang, Kongzhang Hao, Xuemin Lin, Lu Qin, Wenjie Zhang, et al. 2019. Distributed subgraph matching on timely dataflow. In *PVLDB*.

[33] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. 2012. An in-depth comparison of subgraph isomorphism algorithms in graph databases. In *PVLDB*.

[34] Matteo Lissandrini, Martin Brugnara, and Yannis Velegrakis. 2018. Beyond macrobenchmarks: Microbenchmark-based graph database evaluation. In *PVLDB*.

[35] Ciaran McCreesh and Patrick Prosser. 2015. A parallel, backjumping subgraph isomorphism algorithm using supplemental graphs. In *International conference on principles and practice of constraint programming*.

[36] Ciaran McCreesh, Patrick Prosser, Christine Solnon, and James Trimble. 2018. When subgraph isomorphism is really hard, and why this matters for graph databases. In *Journal of Artificial Intelligence Research*.

[37] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing subgraph queries by combining binary and worst-case optimal joins. In *arXiv preprint arXiv:1903.02076*.

[38] Hung Q Ngo. 2018. Worst-Case Optimal Join Algorithms: Techniques, Results, and Open Problems. In *PODS*.

[39] Dung Nguyen, Molham Aref, Martin Bravenboer, George Kollias, Hung Q Ngo, Christopher Ré, and Atri Rudra. 2015. Join processing for graph patterns: An old dog with new tricks. In *Proceedings of the GRADES'15*.

[40] Miao Qiao, Hao Zhang, and Hong Cheng. 2017. Subgraph Matching: on Compression and Computation. In *PVLDB*.

[41] Raghavan Raman, Oskar van Rest, Sungpack Hong, Zhe Wu, Hassan Chafi, and Jay Banerjee. 2014. Pgx. iso: parallel and efficient in-memory engine for subgraph isomorphism. In *Proceedings of Workshop on GRAph Data management Experiences and Systems*.

[42] Xuguang Ren and Junhu Wang. 2015. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. In *PVLDB*.

[43] Carlos R Rivero and Hasan M Jamil. 2017. Efficient and scalable labeled subgraph matching using SGMatch. In *Knowledge and Information Systems*.

[44] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. 2017. The ubiquity of large graphs and surprising challenges of graph processing. In *PVLDB*.

[45] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. In *PVLDB*.

[46] Yingxia Shao, Bin Cui, Lei Chen, Lin Ma, Junjie Yao, and Ning Xu. 2014. Parallel subgraph listing in a large-scale graph. In *SIGMOD*.

[47] Christine Solnon. 2010. All different-based filtering for subgraph isomorphism. In *Artificial Intelligence*.

[48] Christine Solnon. 2019. Experimental Evaluation of Subgraph Isomorphism Solvers. In *International Workshop on Graph-Based Representations in Pattern Recognition*.

[49] Yinglong Song, Huey Eng Chua, Sourav S Bhowmick, Byron Choi, and Shuigeng Zhou. 2018. BOOMER: Blending visual formulation and processing of p-homomorphic queries on large networks. In *SIGMOD*.

[50] Shixuan Sun, Yulin Che, Lipeng Wang, and Qiong Luo. 2019. Efficient Parallel Subgraph Enumeration on a Single Machine. In *ICDE*.

[51] Shixuan Sun and Qiong Luo. 2018. Parallelizing Recursive Backtracking Based Subgraph Matching on a Single Machine. In *ICPADS*.

[52] Shixuan Sun and Qiong Luo. 2019. Scaling Up Subgraph Query Processing with Efficient Subgraph Matching. In *ICDE*.

[53] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. 2012. Efficient subgraph matching on billion node graphs. In *PVLDB*.

[54] Ha-Nguyen Tran, Jung-jae Kim, and Bingsheng He. 2015. Fast subgraph matching on large graphs using graphics processors. In *DASFAA*.

[55] Julian R Ullmann. 1976. An algorithm for subgraph isomorphism. In *JACM*.

[56] Mario Vento, Xiaoyi Jiang, and Pasquale Foggia. 2015. International contest on pattern search in biological databases.

[57] Zhaokang Wang, Rong Gu, Weiwei Hu, Chunfeng Yuan, and Yihua Huang. 2019. BENU: Distributed Subgraph Enumeration with Backtracking-Based Framework. In *ICDE*.

[58] Shijie Zhang, Shirong Li, and Jiong Yang. 2009. GADDI: distance index based subgraph matching in biological networks. In *EDBT*.

[59] Peixiang Zhao and Jiawei Han. 2010. On graph query optimization in large networks. In *PVLDB*.