Scaling Up Subgraph Query Processing with Efficient Subgraph Matching

Shixuan Sun Qiong Luo
Department of Computer Science and Engineering, Hong Kong University of Science and Technology
{ssunah, luo}@cse.ust.hk

***Abstract*—A subgraph query finds all data graphs in a graph database each of which contains the given query graph. Existing work takes the indexing-filtering-verification (IFV) approach to first index all data graphs, then filter out some of them based on the index, and finally test subgraph isomorphism on each of the remaining data graphs. This final test of subgraph isomorphism is a sub-problem of subgraph matching, which finds all subgraph isomorphisms from a query graph to a data graph. As such, in this paper, we study whether, and if so, how to utilize efficient subgraph matching to improve subgraph query processing. Specifically, we modify leading subgraph matching algorithms and integrate them with top-performing subgraph querying algorithms. Our results show that (1) the slow verification method in existing IFV algorithms can lead us to over-estimate the gain of filtering; and (2) our modified subgraph querying algorithms with efficient subgraph matching are competitive in time performance and can scale to hundreds of thousands of data graphs and graphs of thousands of vertices.**

I. Introduction

Given a graph database containing a collection of data graphs $\mathcal{D} = \{G_1, ..., G_n\}$ and a query graph q, a subgraph query retrieves all data graphs in $\mathcal{D}$ that contain q. Subgraph queries are widely present in real-world applications such as computer aided design, protein interaction relationship retrieval, social network analysis and RDF (Resource Description Framework) queries. Deciding whether a query graph q is a subgraph of a data graph G, i.e., the subgraph isomorphism problem, is proven to be NP-complete [5]. Thus, it is computationally expensive to conduct this subgraph isomorphism test against each data graph in $\mathcal{D}$.

To perform subgraph isomorphism tests on as few data graphs as possible, dozens of algorithms have been proposed (listed in Table II in Section II), and all of them follow an *indexing-filtering-verification* paradigm (IFV). Algorithm 1 illustrates the general procedure of IFV subgraph query processing methods. Given a graph database $\mathcal{D}$, the first step is to build an index $\mathcal{I}$ on features such as paths, trees and graphs (Lines 1-3). Typically, $\mathcal{I}$ can be abstracted as a key-value store where the keys are the features f and the corresponding values are the data graphs containing f. IFV methods process a query q against $\mathcal{D}$ in two steps - filtering and verification. The filtering step (Lines 5-6) decomposes q into a collection of features $\mathcal{F}(q)$ and obtains a set of data graphs $\mathcal{C}(q)$ each of which contains all the features in $\mathcal{F}(q)$ based on $\mathcal{I}$. $\mathcal{C}(q)$, called a *candidate set*, contains all the data graphs in $\mathcal{D}$ that subsume q, because if a data graph G contains q, G must contain all the features of q. Therefore, in the verification step, IFV-based

Algorithm 1: The IFV Procedure

```
1  I: the index of a graph database D;
2  Procedure BuildIndex(D)
3      I ← Build a graph index of D;
4  Function Query(D, I, q)
       /* The filtering step                                   */
5      F(q) ← Decompose q into a collection of features;
6      C(q) ← ⋂_{f∈F(q)} Lookup(I, f);
       /* The verification step                                */
7      A(q) ← ∅;
8      foreach G ∈ C(q) do
9          if Verify(q, G) is true then A(q) ← A(q) ∪ {G};
10     return A(q);
```

methods verify the data graphs in $\mathcal{C}(q)$ only, instead of $\mathcal{D}$, with a subgraph isomorphism test algorithm to get the final *answer set* $\mathcal{A}(q)$ (Lines 7-9). With such an IFV paradigm, the number of subgraph isomorphism tests is reduced from $|\mathcal{D}|$ to $|\mathcal{C}(q)|$.

Although IFV algorithms have led to significant performance improvement, these index-based methods have considerable drawbacks. Firstly, a performance study [15] finds that these algorithms fail to build indices on large graph databases in terms of number of distinct labels, number of vertices in data graphs, density of data graphs and number of data graphs in $\mathcal{D}$ due to their poor time and space efficiency of index construction. Secondly, as in any index-based methods, whenever $\mathcal{D}$ is modified, $\mathcal{I}$ must be updated correspondingly to guarantee that $\mathcal{C}(q)$ obtained in the filtering step contains $\mathcal{A}(q)$. The cost incurred by the index update is high [39]. As such, IFV algorithms are hardly applicable to graphs that change frequently, such as networks of purchasing records in online stores and trading records of shareholders in financial firms. Finally, the verification step in most of existing algorithms still adopts the VF2 algorithm [6], which has been outperformed by the latest subgraph matching algorithms by several orders of magnitude [1], [23].

Subgraph matching is a topic closely related to subgraph query processing. It finds all subgraph isomorphisms from a query graph to a single large data graph. Most subgraph query algorithms conduct the subgraph isomorphism test by modifying subgraph matching algorithms to return after finding the first subgraph isomorphism. Recent research efforts on subgraph matching have led to significant improvement on the execution time. Moreover, the methodology of subgraph matching also makes great advancement. Specifically, previous subgraph matching algorithms such as VF2 adopt the direct-enumeration method that simply expands partial results recursively along a matching order of query vertices by mapping

query vertices to data vertices to find subgraph isomorphisms. In contrast, latest algorithms such as GraphQL [14], TurboIso [11] and CFL [1] adopt the preprocessing-enumeration method that constructs an auxiliary data structure (e.g., the candidate vertex set for each query vertex) in the preprocessing phase before the recursive enumeration to reduce the number of candidate vertices and optimize the matching order. These auxiliary data structures in subgraph matching are different from the index in IFV algorithms: They are light-weight and dynamically constructed for each query, whereas the index in IFV is built based on the data graphs to answer all queries.

Unfortunately, the advancement in subgraph matching has not been utilized by current research on subgraph query processing. In this paper, we conduct an empirical study on this problem. Specifically, we experiment on three categories of subgraph query processing algorithms. The first category includes three top-performing IFV algorithms CT-Index [20], Grapes [10] and GGSX [2]. The second category consists of the *vertex connectivity based filtering-verification* (vcFV) subgraph query processing algorithms, which are obtained by modifying the leading preprocessing-enumeration subgraph matching algorithms. In particular, we modify CFL [1] and GraphQL [14] to answer subgraph queries by (1) utilizing their preprocessing techniques as the filtering method instead of the index-based filtering in IFV algorithms; and (2) returning immediately after finding the first subgraph isomorphism in the enumeration. In the third category, we integrate the IFV algorithms with the vcFV algorithms to get the *index and vertex connectivity based filtering-verification* (IvcFV) algorithms.

In summary, we make the following contributions.

- We conduct experiments on real-world datasets with both sparse and dense query graphs to study the performance of three categories of subgraph query processing algorithms.
- We evaluate the scalability of the competing algorithms on a variety of synthetic datasets.
- We reveal the impact of subgraph matching on subgraph query processing (see Section IV-D).
- We make an initial step towards indexing-free subgraph query processing by utilizing latest techniques in subgraph matching.

Paper Organization. Section II presents preliminaries and related work. Section III introduces the competing algorithms. We evaluate the performance of the competing algorithms in Section IV and conclude in Section V.

II. Background

In this section, we first introduce the preliminaries used in this paper. Then, we present the related work.

A. Preliminaries

In this paper, we focus on the vertex-labeled undirected graph $g = (V, E, L)$, where V is a set of vertices, E is a set of edges, and L is a function that associates a vertex u with a label $L(u) \in \Sigma$ (Σ is the set of labels). A graph database is a collection of graphs, denoted as $\mathcal{D} = \{G_1, G_2, ..., G_n\}$.

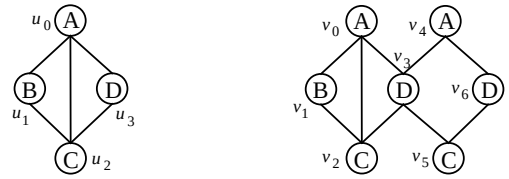
TABLE I: Notations.

Notations	Descriptions
φ	subgraph isomorphism
$\mathcal{D}$	graph database
Σ	label set
g, q and G	graph, query graph and data graph
$\mathcal{A}(q)$ and $\mathcal{C}(q)$	answer set and candidate set of q
f and $\mathcal{F}(q)$	feature and feature set of q
$V(g)$ and $E(g)$	vertex set and edge set of g
$d(u)$, $L(u)$ and $N(u)$	degree, label and neighbors of u
$e(u, v)$	edge connecting vertices u and v
Φ and $\Phi(u)$	candidate vertex set and candidate vertex set of u

We call a graph $G \in \mathcal{D}$ a data graph. The query graph q is connected. In the following, we give a formal definition of subgraph query and related preliminaries used in this paper, and list the frequently used notations in Table I.

Definition II.1. *Subgraph Isomorphism: Given graphs $g = (V, E, L)$ and $g' = (V', E', L')$, a subgraph isomorphism from g to g' is an injective function $\varphi : V \rightarrow V'$ that satisfies: (1) $\forall u \in V, L(u) = L'(\varphi(u))$; and (2) $\forall e(u, u') \in E, e(\varphi(u), \varphi(u')) \in E'$.*

If there exists a subgraph isomorphism from g to g', then g is subgraph-isomorphic to g', denoted as $g \subseteq g'$.



(a) Query graph q. **(b)** Data graph G.

Fig. 1: Example graphs.

Example II.1. *Given q and G in Figure 1, $\varphi = \{(u_0, v_0), (u_1, v_1), (u_2, v_2), (u_3, v_3)\}$ is a subgraph isomorphism from q to G. Then, q is subgraph-isomorphic to G.*

Definition II.2. *Subgraph Query: Given a graph database $\mathcal{D} = \{G_1, G_2, ..., G_n\}$ and a query graph q, a subgraph query finds all data graphs G in $\mathcal{D}$ such that $q \subseteq G$.*

Definition II.3. *Subgraph Matching: Given a data graph G and a query graph q, subgraph matching finds all subgraphs in G that are isomorphic to q.*

B. Related Work

In this section, we introduce the work related to this paper.

1) Subgraph Query: Table II lists 15 existing subgraph query processing algorithms. All of them follow the IFV paradigm with differences on the structure of index features (e.g., path, tree, cycle or graph) and the method of feature extraction. In the following, we briefly introduce the properties of mining-based and enumeration-based algorithms.

Mining-based Approaches. Mining-based methods extract "frequent" features to construct indices based on graph mining. A feature is regarded as frequent if the *support ratio*, which is the percentage of data graphs in the graph database containing the feature, is above a threshold. Except for the support ratio, a *discriminative ratio*, which is an algorithm-specific metric to

TABLE II: A list of IFV algorithms.

Algorithm	Feature Extraction	Feature Structure	Index Storage
GraphGrep [30]	Enumeration	Path	Memory
GraphGrepSX [2]	Enumeration	Path	Memory
Grapes [10]	Enumeration	Path	Memory
SING [7]	Enumeration	Path	Memory
CT-Index [20]	Enumeration	Tree/Cycle	Memory
GDIndex [35]	Enumeration	Graph	Memory
GCode [43]	Enumeration	Graph	Memory
SwiftIndex [28]	Mining	Tree	Memory
TreePi [40]	Mining	Tree	Memory
Tree+Delta [42]	Mining	Tree/Graph	Memory
CP-Index [36]	Mining	Graph	Memory
gIndex [37]	Mining	Graph	Memory
FG-Index [4]	Mining	Graph	Memory/Disk
FG*-Index [3]	Mining	Graph	Memory/Disk
Lindex+ [38]	Mining	Graph	Memory/Disk

measure the filtering power of a feature, is another threshold to determine whether this feature should be included in the index. These parameters control the trade-off between the complexity of feature extraction (i.e., both time and space) and the filtering power of indices. However, several performance studies show that mining-based methods consume too much time to build indices due to the expensive mining operations, and the parameters are hard to specify [10], [12], [15], [20].

Enumeration-based Approaches. The enumeration-based approaches exhaustively enumerate all specified features and store them into indices. The index construction of these methods is faster than that of mining-based, but the resulting indices are of a large size and consume a great amount of memory to construct. Therefore, this kind of algorithms generally adopt simpler feature structures such as paths and cycles than the mining-based. Also, they require users to specify the sizes of features, e.g., the number of vertices or the number of edges, to balance the filtering power and the index size.

Additionally, because the sizes of indices can be exponential to the sizes of data graphs, some algorithms [3], [4], [38] design indices that leverage both the memory and disk. As the graph database itself consumes a small amount of memory space compared with the indices (see Tables VII and IX in Section IV), we assume that it fits into memory.

Other Approaches. Wang et al. [33], [34] proposed a graph cache system to improve the performance of subgraph query processing. A recent work [16] proposed to integrate the indexing-filtering strategy in IFV methods with subgraph matching algorithms to boost the performance of subgraph matching on a collection of data graphs. In particular, this approach first obtains the candidate graphs with the indexing-filtering method and then performs subgraph matching on the candidate graphs instead of all data graphs. In contrast, we propose the vcFV framework to utilize the preprocessing strategy in the state-of-the-art subgraph matching algorithms as a filtering method instead of the index-based filtering to address the problems caused by the index in IFV methods.

2) Subgraph Matching: We categorize the subgraph matching algorithms into direct-enumeration and preprocessing-enumeration based on their execution phases.

Direct-enumeration Approaches. The direct-enumeration approaches such as Ullmann [32], VF2 [6], QuickSI [28] and SPath [41] do not construct any auxiliary data structure given a query graph before enumeration, but obtain the candidate data vertices of each query vertex individually based on filters such as the neighborhood signature. Due to the lack of accurate information to estimate cost, the matching order can be ineffective and there may be many false positive candidate data vertices. A detailed performance study [23] shows that these algorithms have problems in their matching order selection, and signature-based filters are only effective for some datasets.

Preprocessing-enumeration Approaches. To address the problems in the direct-enumeration algorithms, researchers proposed to divide subgraph matching into two phases, which first construct an auxiliary data structure given a query graph and then conduct enumeration based on the auxiliary data structure. The auxiliary data structure not only reduces the candidates but also provides accurate cost estimation to generate an effective matching order. GraphQL [14] proposed the pseudo subgraph isomorphism test based method to generate the candidate vertex sets for each query vertex, and designed the join-based ordering strategy. TurboIso [11] designed the tree-structured auxiliary data structure, candidate region, and generated the matching order by the path-based ordering strategy. CFL [1], the state-of-the-art algorithm, accelerated subgraph matching by postponing cartesian products with a matching order that prioritizes the query vertices in the core structure, and proposed a new tree-structured auxiliary data structure CPI. These approaches achieved impressive speedups over the direct-enumeration algorithms such as VF2.

Other Approaches. Some researchers improved subgraph matching by exploiting the vertex relationship in the data graphs [24], [26], utilizing the matching results among multiple queries [27] and parallelizing the algorithms in a single machine as well as the distributed environment [19], [31]. Also, researchers considered the confidentiality of the subgraph queries in a graph database [9].

Finally, subgraph matching can also be done on unlabeled graphs. Due to the lack of the label information, the search space of unlabeled subgraph matching is much larger than that of labeled. Moreover, unlabeled subgraph matching is generally used as an offline analysis task, whereas subgraph query and labeled subgraph matching are integrated as a query operation in a graph database. To handle the large search space, most research focuses on distributed environments [21], [22], [25], [29], and other work deals with large graphs that exceed the main memory [18].

III. Competing Algorithms

In this paper, we study eight subgraph query processing algorithms in three categories based on their execution phases.

A. IFV Algorithms

Previous performance studies [12], [15] compared the performance of nine algorithms to cover as much of the design space of subgraph query processing algorithms (e.g., enumeration-based or mining-based, the feature structure and the index structure) as possible. The nine algorithms are

Grapes, GraphGrepSX, CT-Index, gIndex, GCode, Tree+Delta, SwiftIndex, TreePi and FG-Index. Among the nine algorithms, Grapes, GGSX and CT-Index perform better than the other algorithms on a variety of metrics such as indexing time, query time, filtering precision and scalability [15]. Following previous work, we select these three algorithms to compare in our experiments.

Grapes [10]. Grapes is a parallel algorithm working in multi-core machines with the index constructed by an enumeration-based strategy. It adopts the path as the indexing feature and exhaustively enumerates paths of up to a maximum length. The index is stored as a trie. Grapes adopts VF2 to verify whether candidate graphs contain the query graph.

GGSX [2]. GraphGrepSX has an index constructed by an enumeration-based strategy, whose basic feature is the path. The index is stored as a suffix tree. The verification phase of GGSX also uses VF2.

CT-Index [20]. The index structure is constructed by an enumeration-based strategy, whose basic features contain trees and cycles. The index is stored as a hash set of *fingerprints*. The verification of CT-Index is a modified VF2 that adopts additional heuristics to optimize the matching order.

B. vcFV Algorithms

A naive method using subgraph matching algorithms to answer subgraph queries is as follows: (1) Modify subgraph matching algorithms to immediately return after finding the first subgraph isomorphism; and (2) Execute the modified algorithm against each data graph in $\mathcal{D}$ to find the data graphs containing the given q. As the subgraph isomorphism problem is NP-complete, conducting subgraph isomorphism test on each data graph could be time-consuming. However, we find the recently proposed preprocessing-enumeration subgraph matching algorithms provides a different strategy.

Specifically, the preprocessing-enumeration subgraph matching algorithms such as GraphQL [14], TurboIso [11] and CFL [1] introduce a preprocessing phase before the enumeration in order to reduce the size of candidates for each query vertex and obtain accurate statistics to optimize the matching order. In particular, the preprocessing phase is to generate a *complete candidate vertex set* (Definition III.1) for each query vertex.

Definition III.1. *Complete Candidate Vertex Set: Given q and G, a candidate vertex set $\Phi(u)$ for $u \in V(q)$ is complete if $\Phi(u)$ satisfies: If a mapping (u, v) exists in a subgraph isomorphism from q to G where $v \in V(G)$, then $v \in \Phi(u)$. If $\forall u \in V(q)$, $\Phi(u)$ is complete, then we say Φ is complete.*

Example III.1. *Given q and G in Figure 1, suppose that $\Phi(u_0) = \{v_0, v_4\}$, $\Phi(u_1) = \{v_1\}$, $\Phi(u_2) = \{v_2\}$ and $\Phi(u_3) = \{v_3\}$. Φ is complete, because for each $u \in V(q)$, $\Phi(u)$ satisfies that if a mapping (u, v) exists in a subgraph isomorphism from q to G, then $v \in \Phi(u)$.*

Based on Definition III.1, the following proposition holds.

Algorithm 2: The vcFV Procedure

```
   Input: a query graph q and a graph database D
   Output: an answer set A(q) keeping all data graphs in D that contain q
1  begin
2      A(q) ← ∅;
3      foreach G ∈ D do
4          Φ ← Filter(q, G);
5          if ∀u ∈ V(q), Φ(u) ≠ ∅ then
6              if Verify(q, G, Φ) is true then
7                  A(q) ← A(q) ∪ {G};
8      return A(q);
```

Proposition III.1. *Given q and G, suppose that Φ is complete. If there exists a query vertex $u \in V(q)$ such that $\Phi(u)$ is empty, then there is no subgraph isomorphism from q to G.*

Proof. Given q and G, Φ is a complete candidate vertex set. Suppose there exists a subgraph isomorphism φ from q to G and there exists $u \in V(q)$ such that $\Phi(u)$ is empty. By Definition III.1, $\Phi(u)$ contains $\varphi(u)$, which contradicts that $\Phi(u)$ is empty. Thus, the proposition is proved by contradiction. □

Given q and $G \in \mathcal{D}$, Φ is complete. If $\exists u \in V(q), \Phi(u) = \emptyset$, then G must not contain q and we do not need to conduct a subgraph isomorphism test against G. Therefore, when using the preprocessing-enumeration subgraph matching algorithms to answer subgraph queries, we can follow the filtering-verification paradigm whose filtering is based on the vertex connectivity, i.e., the candidate vertex sets generated based on the structure of q and G, instead of the index. In order to differentiate with the IFV algorithms as well as the direct-enumeration subgraph matching algorithms, we say that the preprocessing-enumeration algorithms follow the vertex connectivity based filtering-verification (vcFV) framework and call them the vcFV algorithms in this paper.

Because the preprocessing-enumeration subgraph matching algorithms construct the candidate vertex sets anyway before the enumeration, one approach would be to simply regard these algorithms as a black box and directly run them on each data graph. In contrast, we apply this strategy but separate the process into two phases. This way we can easily examine the bottleneck of subgraph query processing, the filtering precision or the performance of the verification.

Algorithm 2 outlines the vcFV framework in which the *Filter* function is the preprocessing phase of the integrated subgraph matching algorithms and the *Verify* function is the enumeration phase that returns after finding the first subgraph isomorphism. We regard data graphs passing the check of line 5 as the candidate set $\mathcal{C}(q)$ of vcFV algorithms.

According to Algorithm 2, if the *Filter* function can obtain a *minimum complete candidate vertex set* (Definition III.2), then we can avoid the verification. However, reducing the construction of a minimum complete candidate vertex set to the subgraph isomorphism problem, we have Proposition III.2. This is also why the preprocessing-enumeration algorithms construct a complete candidate vertex set based on some heuristics instead of building a minimum one.

Definition III.2. *Minimum Complete Candidate Vertex Set: Given q and G, a complete candidate vertex set Φ is minimum*

if Φ satisfies that for every query vertex $u \in V(q)$, if $v \in \Phi(u)$, then the mapping (u, v) belongs to a subgraph isomorphism from q to G.

Proposition III.2. *Given q and G, finding a minimum complete candidate vertex set is NP-hard.*

In this paper, we select GraphQL [14] and CFL [1] as representatives to study the vcFV category of subgraph query processing algorithms, because previous work [17], [23] shows that GraphQL works well on small data graphs and CFL is the state-of-the-art subgraph matching algorithm. Moreover, we integrate the preprocessing phase in CFL with the enumeration phase of GraphQL to obtain a new algorithm called CFQL, taking advantage of both CFL and GraphQL.

GraphQL [14]. The *Filter* function (i.e., the preprocessing phase in GraphQL) obtains the complete candidate vertex sets in two steps. The first step is to generate a candidate vertex set for each query vertex based on the *neighborhood profiles* (i.e., the label information of neighbor vertices). The second step is to prune the candidate vertex set based on an approximate subgraph isomorphism algorithm [13] that works as follows: (1) Given $v \in \Phi(u)$, construct a bigraph with $N(u)$ and $N(v)$ as bipartitions where an edge is added between $u' \in N(u)$ and $v' \in N(v)$ if $v' \in \Phi(u')$, denoted as B; (2) Perform maximum bigraph matching to check whether there exists a semi-perfect matching in B, i.e., every vertex in $N(u)$ is matched; and (3) If not, remove v from $\Phi(u)$. The space complexity of the *Filter* function is $O(|V(q)| \times |V(G)|)$. The time complexity of the *Filter* function is $O(|V(q)| \times |V(G)| \times \Theta(d_q, d_G))$ where d_q and d_G are the average degrees of q and G respectively. Θ is the time complexity of the maximum bigraph matching. In our implementation, we use a breadth-first search based maximum bigraph matching algorithm whose time complexity is $O(|V(B)| \times |E(B)|)$, because a detailed performance study of maximum bigraph matching algorithms shows that this algorithm has a reasonable performance and it is easy to implement [8]. Because GraphQL does not specify the order of generating/pruning candidate vertex sets, our implementation conducts the generating/pruning along the ascending order of the query vertex *id*s.

The *Verify* function (i.e., the enumeration phase in GraphQL) adopts the *join-based ordering* strategy. This ordering strategy generates a matching order by picking a query vertex with the minimum number of candidates from the neighbors of the selected query vertices at each step.

CFL [1]. The *Filter* function (i.e., the preprocessing phase in CFL) constructs complete candidate vertex sets based on the following observation: Given q and G, suppose that Φ is a complete candidate vertex set. If a mapping (u, v) exists in a subgraph isomorphism from q to G where $v \in \Phi(u)$, then $\forall u' \in N(u), N(v) \cap \Phi(u') \neq \emptyset$. According to this observation, given $v \in \Phi(u)$, v can be safely removed if there exists a query vertex $u' \in N(u)$ such that $N(v) \cap \Phi(u') = \emptyset$. Equivalently, given a query vertex u, $\Phi(u)$ can be obtained by intersecting the sets of neighbors, with label $L(u)$, of vertices in $\Phi(u')$ for all $u' \in N(u)$. CFL first generates a BFS (Breadth-first search) tree q_t from q. Then, it obtains Φ in two steps based on the observation: (1) Generate a complete candidate vertex set for each query vertex along q_t level-by-level in a top-down manner, in which CFL also conducts backward pruning at each level based on the non-tree edges; and (2) Refine Φ along q_t in a bottom-up order. The time complexity of the *Filter* function is $O(|E(q)| \times |E(G)|)$ and the space complexity is $O(|V(q)| \times |E(G)|)$.

The *Verify* function (i.e., the enumeration phase in CFL) adopts the *path-based ordering* strategy. This strategy generates the matching order by sorting the paths in q_t based on the information of candidate vertex sets. Moreover, the matching order of CFL prioritizes the vertices in the *core structure* (i.e., 2-core) of q.

CFQL. Given q and G, the performance of the *Verify* functions can differ greatly between algorithms because of their different ordering strategies. If a *Verify* function performs well on a variety of data graphs and query graphs, we say the *Verify* function is robust. We integrate the *Filter* function of CFL with the *Verify* function of GraphQL to obtain a new vcFV algorithm called CFQL, because our experimental results show that the *Filter* function of CFL runs faster than that of GraphQL, and the *Verify* function of GraphQL is more robust than that of CFL.

TABLE III: A summary of competing algorithms.

Category	Algorithm	Indexing	Filtering	Verification
IFV	CT-Index	Hashset	Index	VF2
	Grapes	Trie	Index	VF2
	GGSX	Suffix tree	Index	VF2
vcFV	CFL	N/A	Preprocesing of CFL	Enumeration of CFL
	GraphQL	N/A	Preprocesing of GraphQL	Enumeration of GraphQL
	CFQL	N/A	Preprocesing of CFL	Enumeration of GraphQL
IvcFV	vcGrapes	Trie	Index and preprocesing of CFL	Enumeration of GraphQL
	vcGGSX	Suffix tree	Index and preprocesing of CFL	Enumeration of GraphQL

C. IvcFV Algorithms

A general idea to improve the performance of existing IFV algorithms is to replace VF2 in the verification with the state-of-the-art subgraph matching algorithm. Our experimental results show that CFQL that integrates the preprocessing of CFL with the enumeration of GraphQL can perform well. Therefore, we integrate CFQL into Grapes and GGSX. Because Grapes and GGSX have the index-based filtering whereas CFQL has the vertex connectivity based filtering, the integrated algorithms have two levels of filtering with one level index-based and the other vertex connectivity based. We call the integrated methods the IvcFV methods and denote them as vcGrapes and vcGGSX. We do not integrate with CT-Index in this category, as its indexing phase often fails on large datasets, which results in failing to answer queries on those datasets.

D. Summary of Competing Algorithms

Table III summarizes the eight competing algorithms in this paper. The three IFV algorithms conduct the filtering based on the indices and verify the existence of the query graph with

VF2. The vcFV algorithms do not build any indices, but filter based on the preprocessing method of the integrated subgraph matching algorithm. We modify the enumeration phase of the corresponding subgraph matching algorithm to immediately return after finding the first subgraph isomorphism as the verification function. The IvcFV algorithms first filter based on the index and then the preprocessing phase of the integrated subgraph matching algorithm.

IV. Experiments

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

A. Experimental Setup

Algorithm Configuration. The parameter values of Grapes, GGSX and CT-Index are configured the same as the previous work [15], since we use the same real-world dataset and synthetic dataset generator. As the indexing methods of vcGrapes and vcGGSX are the same as Grapes and GGSX respectively, they adopt the same configuration as their original version. The vcFV algorithms do not require any input configuration. The following is algorithm-specific parameter setting.

- Grapes, vcGrapes: We use 6 threads and enumerate paths of up to a length of 4.
- GGSX, vcGGSX: We enumerate paths of up to a length of 4.
- CT-Index: We create 4096-bit fingerprints by enumerating trees and cycles of up to a length of 4.

Experiment Environment. We obtain the source code of Grapes and GGSX from their authors respectively, and obtain an executable binary of CT-Index from its author. Grapes and GGSX are implemented in C++, and the binary of CT-Index is based on JAVA. We implement the other algorithms in C++. We compile the source code of algorithms implemented in C++ with g++ 4.9.3 with the -O3 flag. We perform all experiments on a 64-bit Linux machine equipped with two Intel Xeon E5-2650 V3 processors and 64GB RAM.

Datasets. We conduct experiments on both real-world and synthetic datasets.

Real-world Datasets: We obtain all real-world datasets used in the performance study [15] from its author, which are AIDS, PDBS, PCM and PPI. AIDS contains topological structures of molecules. PDBS is a set of graphs representing DNA, RNA and proteins. PCM is a collection of graphs representing protein interaction maps. PPI is also a dataset representing protein interaction networks, but the networks are much larger than those in PCM. The statistics of these real-world datasets is listed in Table IV.

Synthetic Datasets: To be consistent with experiments in previous research [12], [15], we use GraphGen [4], a tool that can generate a collection of data graphs with parameters such as #graphs, #labels, and density, to generate a variety of synthetic datasets. Specifically, we vary the number of graphs $|\mathcal{D}|$, the number of vertices in a data graph $|V(G)|$, the number of distinct labels $|\Sigma|$ and the degree of data graphs $d(G)$ respectively to examine the scalability. A challenging problem is to set the default values for these parameters such that we can demonstrate the capabilities of the algorithms without breaking most of them. Katsarou et al. [15] carefully examined the scalability of the index-based algorithms with GraphGen, and computed a set of "sane defaults" as follows: $|\mathcal{D}| = 1000$, $|V(G)| = 200$, $|\Sigma| = 20$ and density=0.025 (i.e., $\frac{2\times|E(G)|}{|V(G)|\times(|V(G)|-1)}$). In our experiment, we use degree (i.e., $\frac{2\times|E(G)|}{|V(G)|}$) instead of density, because given a fixed value of density, a linear increase of $|V(G)|$ results in a quadratic increase of $|E(G)|$. Thus, we set $|\mathcal{D}| = 1000$, $|\Sigma| = 20$, $|V(G)| = 200$ and $d(G) = 8$ as default (different from the "sane defaults", we increase $d(G)$ from 5 to 8 to stress test scalability), and vary the parameters as follows:

- Vary $|\mathcal{D}|$: We generate 5 datasets with $|\mathcal{D}|$ as 10^2, 10^3, 10^4, 10^5 and 10^6 respectively.
- Vary $|\Sigma|$: We generate 5 datasets with $|\Sigma|$ as 1, 10, 20, 40 and 80 respectively.
- Vary $|V(G)|$: We generate 5 datasets with $|V(G)|$ as 50, 200, 800, 3200 and 12800 respectively.
- Vary $d(G)$: We generate 5 datasets with $d(G)$ as 4, 8, 16, 32 and 64 respectively.

TABLE IV: Statistics of the real-world datasets.

	AIDS	PDBS	PCM	PPI
#graphs	40,000	600	200	20
#labels	62	10	21	46
#vertices per graph	45	2,939	377	4,942
#edges per graph	46.95	3,064	4,340	26,667
degree per graph	2.09	2.06	23.01	10.87
#labels per graph	4.4	6.4	18.9	28.5

Query Sets. To examine the algorithms comprehensively, we generate query graphs with two methods used in previous work: random walk [12], [15], [17] and breadth-first search [33], [34]. The random walk method works as follows:

1) Select a data graph from $\mathcal{D}$ at random;
2) Select a vertex from the selected graph at random;
3) Starting from the selected vertex, perform a random walk and add the edges and vertices visited to the graph;
4) When the desired number of edges is reached, terminate and return the graph.

The breadth-first search method is the same as the random walk except Step 3: Starting from the selected vertex, perform a breadth-first search and whenever a new vertex is visited, add both this vertex and all its edges to visited vertices into the graph.

Given a dataset, we generate 8 query sets, including 4 with random walk and 4 with breadth-first search. Each query set contains 100 query graphs with the same number of edges. Because the query graphs generated by random walk are sparser than those obtained by breadth-first search, we use Q_{iS} (Sparse) to represent a query set generated by random walk with $i-$edges and Q_{iD} (Dense) to denote that generated by breadth-first search. The statistics of query sets on real-world datasets is shown in Table V, where *% of trees* denotes the percentage of query graphs with a tree structure (i.e., query graphs that have no cycles) in a query set. As the statistics of query sets for synthetic datasets has the same trend as that on real-world datasets, we omit the statistics for brevity.

TABLE V: Statistics of query sets on the real-world datasets.

	AIDS								PDBS							
	Q_{4S}	Q_{8S}	Q_{16S}	Q_{32S}	Q_{4D}	Q_{8D}	Q_{16D}	Q_{32D}	Q_{4S}	Q_{8S}	Q_{16S}	Q_{32S}	Q_{4D}	Q_{8D}	Q_{16D}	Q_{32D}
$\|V\|$ **per** q	5.00	8.92	16.29	31.30	4.99	6.00	9.66	18.16	4.95	8.80	16.41	32.13	4.93	5.99	9.45	18.90
$\|\Sigma\|$ **per** q	2.41	2.88	3.58	4.15	2.37	2.11	1.99	2.76	2.43	2.96	3.45	3.73	2.45	2.00	2.56	3.23
d **per** q	1.60	1.80	1.97	2.05	1.60	2.67	3.32	3.62	1.62	1.83	1.96	2.00	1.63	2.67	3.41	3.44
% of trees	1.00	0.92	0.38	0.14	0.99	0.00	0.00	0.00	0.95	0.85	0.72	0.58	0.93	0.00	0.00	0.00
	PCM								**PPI**							
	Q_{4S}	Q_{8S}	Q_{16S}	Q_{32S}	Q_{4D}	Q_{8D}	Q_{16D}	Q_{32D}	Q_{4S}	Q_{8S}	Q_{16S}	Q_{32S}	Q_{4D}	Q_{8D}	Q_{16D}	Q_{32D}
$\|V\|$ **per** q	4.96	8.48	15.49	29.47	4.00	5.20	7.20	10.11	4.98	8.75	16.13	30.86	4.51	5.60	9.02	16.44
$\|\Sigma\|$ **per** q	4.34	6.64	10.18	14.06	3.65	4.43	5.82	7.77	3.28	4.71	6.63	8.43	3.20	3.86	4.85	6.57
d **per** q	1.62	1.92	2.09	2.19	2.00	3.09	4.46	6.39	1.61	1.83	1.99	2.09	1.80	2.88	3.59	4.06
% of trees	0.96	0.66	0.22	0.04	0.00	0.00	0.00	0.00	0.98	0.77	0.50	0.22	0.51	0.00	0.00	0.00

Metrics. Given a dataset, the time limit for the index construction is 24 hours. The time limit for processing a query is 10 minutes (i.e., 6×10^5 ms). If an algorithm cannot complete the query within the time limit, we record it as 10 minutes for comparison purpose. Moreover, if an algorithm fails to complete more than 40% queries in a query set, then we omit its experiment results on the query set. All algorithms maintain both the index, if present, and the data graphs in memory, and we exclude the time of loading the data from the disk. The used metrics are listed as follows.

- Indexing Time: Time spent on index construction.
- Query Time: Time spent on processing a query graph, consisting of both filtering time and verification time.
- Filtering Time: Time spent on the filtering step. For vcFV and IvcFV algorithms, we take the time on extracting complete candidate vertex sets as part of the filtering time, because extracting complete candidate vertex sets is part of the filtering step of these algorithms.
- Verification Time: Time spent on the verification step.
- Filtering Precision: Given a query set Q on a dataset, the filtering precision on Q is computed as follows:

$$Filtering\ Precision = \frac{1}{|Q|} \sum_{q \in Q} \frac{|\mathcal{A}(q)|}{|\mathcal{C}(q)|}. \quad (1)$$

- Memory Cost[1]: The space cost of the auxiliary data structures: the candidate vertex sets for vcFV algorithms, and the indices for Grapes, GGSX and CT-Index.

Additionally, the verification time of a query q on a dataset $\mathcal{D}$ is computed as follows:

$$T_{verification}(\mathcal{D}, q) = \sum_{G \in \mathcal{C}(q)} T_{verify}(G, q). \quad (2)$$

The verification time depends on both the number of candidate graphs and the efficiency of the subgraph isomorphim (shortened as SI in the following) algorithm. In order to examine the effects of these two factors, we further evaluate the following two metrics.

- $|\mathcal{C}(q)|$: The average number of candidate graphs obtained by a query graph in the query set.

[1]The memory cost of CT-Index is examined by JProfiler. The other algorithms are examined by checking the statistics in the /proc/pid file during the run time.

TABLE VI: Indexing time on real-world datasets(seconds).

	AIDS	PDBS	PCM	PPI
CT-Index	225	1,714	OOT	OOT
GGSX	8	5	433	2,209
Grapes	6	1	66	223

- Per SI Test Time: The average time spent on the subgraph isomorphism test on a candidate graph in the candidate set, which is computed as follows:

$$Per\ SI\ Test\ Time = \frac{1}{|Q|} \sum_{q \in Q} \frac{T_{verification}(\mathcal{D}, q)}{|\mathcal{C}(q)|}. \quad (3)$$

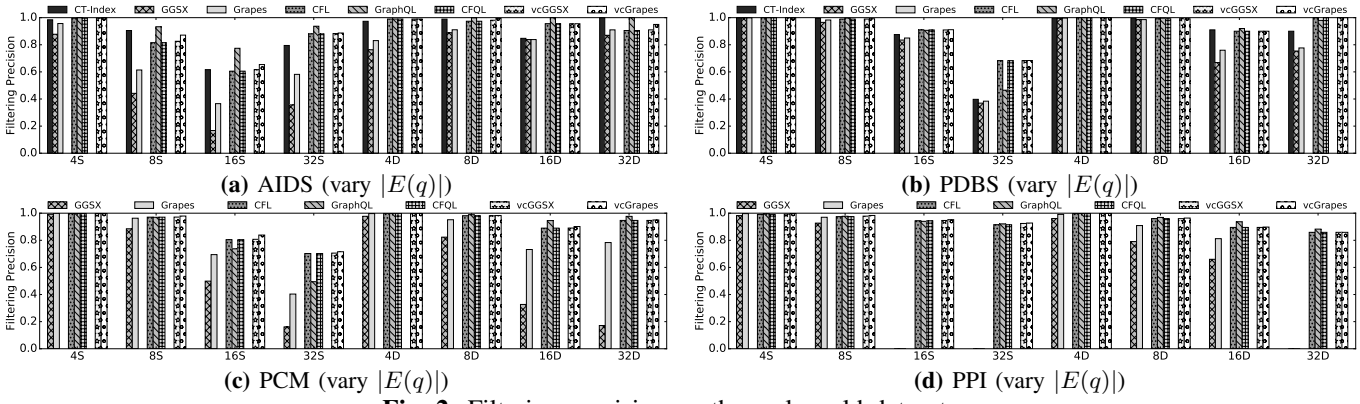
B. Results on Real-world Datasets

In this subsection, we study the performance of competing algorithms on the real-world datasets.

1) Evaluation of index construction: Table VI lists the indexing time on the real-world datasets. As the vcFV algorithms have no indexing step, there is no results of the vcFV algorithms in the table. Because the indexing phases of vcGrapes and vcGGSX are the same as their original versions, we omit the experimental results of their index construction, for example the indexing time and the memory cost, in this study. CT-Index takes much more time to construct indices than the other two algorithms, because it takes trees and cycles as features, which are more complex than paths used in both Grapes and GGSX. CT-Index fails to build indices on PCM or PPI within 24 hours so the indexing time is marked as out-of-time (OOT).

2) Evaluation of the filtering step: The main objective of the filtering-verification paradigm including IFV, vcFV and IvcFV is to reduce the number of candidate graphs. Figure 2 shows the filtering precision on the real-world datasets. The first three bars are IFV algorithms, the next three are vcFV algorithms and the last two are IvcFV algorithms. There are some missing bars, as GGSX and Grapes fail to complete large queries on PPI because of the slow verification, and CT-Index cannot support the queries on PCM and PPI due to the lack of indices.

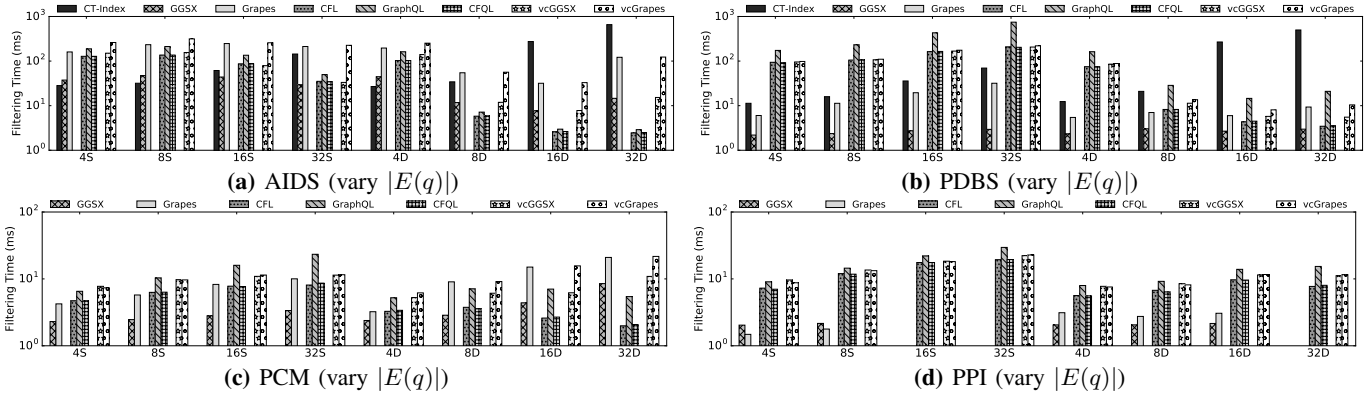
There is no single winner for all cases. Generally, the filtering precision of the competing algorithms is higher on the dense query graphs than that of sparse queries, because dense query graphs provide more features for the index-based filtering, and the vertex connectivity based filtering performs better with the query vertex having more neighbors. Among IFV algorithms, CT-Index outperforms Grapes and GGSX, because its complex indexing feature has stronger filtering power. The filtering precision of vcFV algorithms is



(a) AIDS (vary $|E(q)|$) **(b)** PDBS (vary $|E(q)|$)

(c) PCM (vary $|E(q)|$) **(d)** PPI (vary $|E(q)|$)

Fig. 2: Filtering precision on the real-world datasets.



(a) AIDS (vary $|E(q)|$) **(b)** PDBS (vary $|E(q)|$)

(c) PCM (vary $|E(q)|$) **(d)** PPI (vary $|E(q)|$)

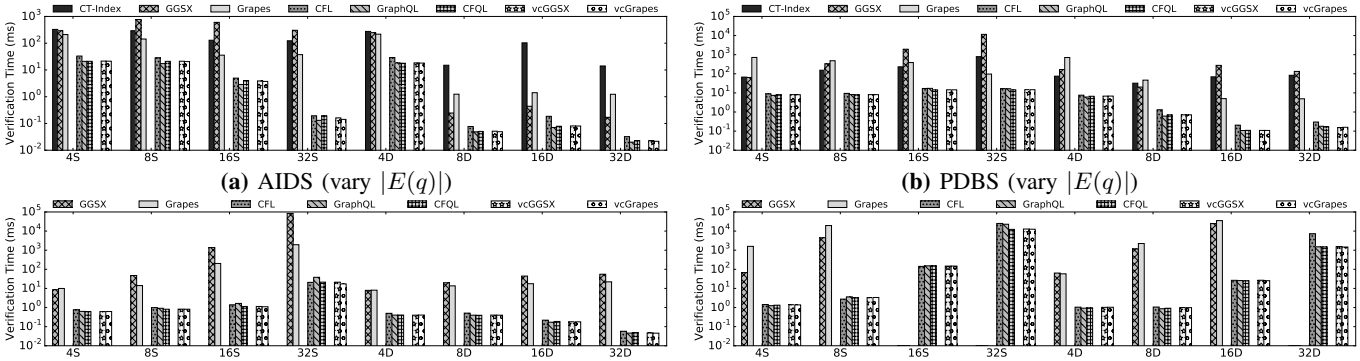
Fig. 3: Filtering time on the real-world datasets.

competitive with that of IFV algorithms, which shows that the vertex connectivity based filtering can also achieve reasonable filtering power. As the filtering, i.e., the preprocessing phase, of CFL and GraphQL is based on heuristics, e.g., different preprocessing orders of query vertices, there is no guarantee on which one is better. Because the filtering precision of CFQL is higher than that of both Grapes and GGSX, integrating with CFQL makes both vcGrapes and vcGGSX achieve a significantly higher filtering precision than Grapes and GGSX respectively, and slightly higher than CFQL.

Figure 3 presents the filtering time on the real-world datasets. The filtering time of IFV algorithms increases with the query size, because there are more features in large queries. In contrast, vcFV algorithms generally take less time on dense queries with the query size increase. This result is because the vertex connectivity based filtering method in vcFV has stronger filtering power when query vertices have more neighbors so that an invalid data graph can be ruled out at an early stage. Among vcFV algorithms, CFL outperforms GraphQL, because the time complexity of the filtering of CFL is better than that of GraphQL.

The filtering time of IvcFV algorithms is expected to be less than that of CFQL, because CFQL conducts the filtering on all data graphs, whereas the vertex connectivity based filtering in IvcFV algorithms processes $|\mathcal{C}'(q)|$ data graphs that pass the index-based filtering. However, the experimental results show that the filtering time of IvcFV algorithms is longer than that of CFQL. There are two reasons for this phenomenon. The first one is that the filtering time of CFQL can be shorter than that of the index based filtering (e.g., dense queries on AIDS) because the time complexity of the filtering in CFQL is good. The second one is that given q and $\mathcal{D}$, CFQL spends most of its filtering time on the candidate data graphs, because for the non-candidate data graph, the filtering of CFQL returns immediately after finding that the candidate vertex set of a query vertex is empty instead of constructing candidate vertex sets for each query vertex. Also, the filtering precision of CFQL is higher than both Grapes and GGSX. Overall, because the time complexity of the filtering method of the competing algorithms is polynomial to the input graph size, the filtering time on all cases is less than 1 second (most less than 0.1 second). Therefore, the absolute value of the filtering time is small.

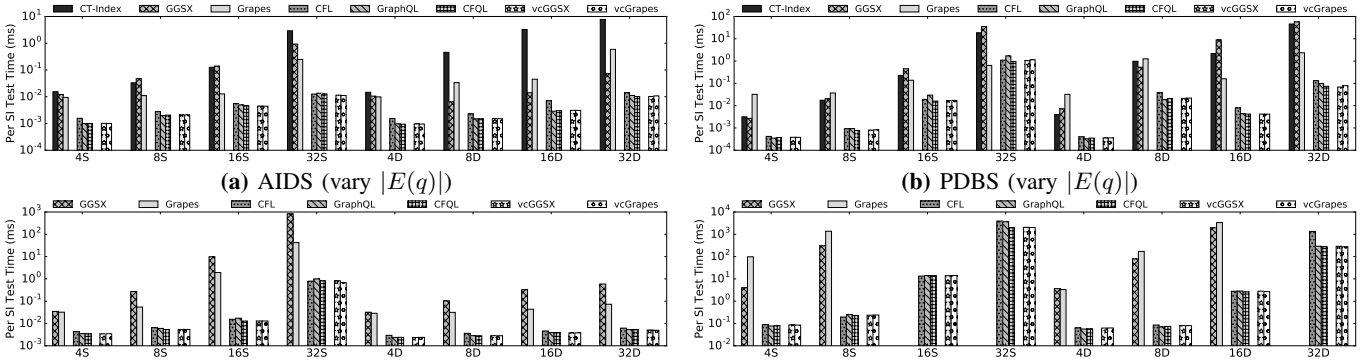
3) Evaluation of the verification step: Figure 4 presents the verification time on the real-world datasets. Both vcFV and IvcFV algorithms that adopt the state-of-the-art subgraph matching algorithms consistently outperform IFV algorithms that use VF2. As both the number of candidate graphs and the efficiency of verification methods can affect the verification time, we show the results on these two factors in Figure 5 and 6 respectively. Benefiting from the integrated efficient subgraph matching algorithm, the vcFV and IvcFV algorithms outperform existing algorithms by up to four orders of magnitude in terms of per SI test time. Because the number of candidate graphs of vcFV and IvcFV algorithms is close to that of IFV algorithms on most cases as shown in Figure 6,



(a) AIDS (vary $|E(q)|$) **(b)** PDBS (vary $|E(q)|$)

(c) PCM (vary $|E(q)|$) **(d)** PPI (vary $|E(q)|$)

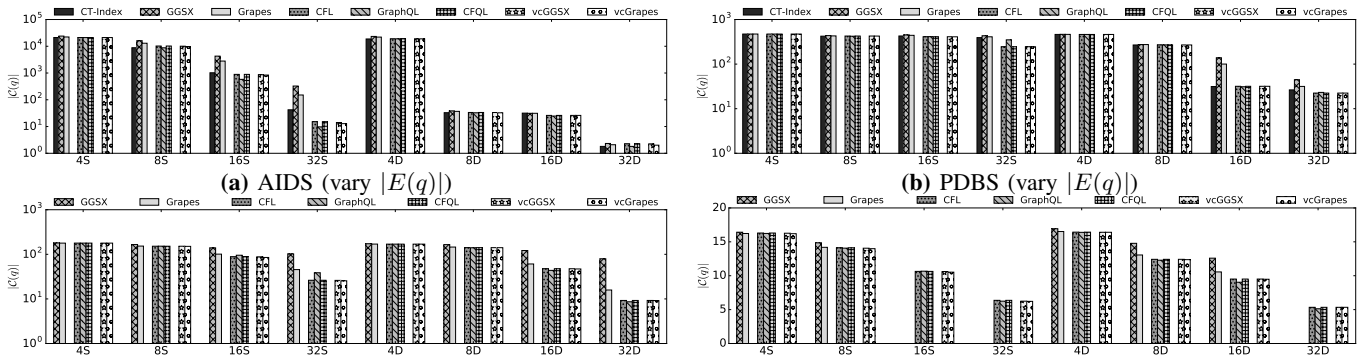
Fig. 4: Verification time on the real-world datasets.



(a) AIDS (vary $|E(q)|$) **(b)** PDBS (vary $|E(q)|$)

(c) PCM (vary $|E(q)|$) **(d)** PPI (vary $|E(q)|$)

Fig. 5: Per SI test time on the real-world datasets.



(a) AIDS (vary $|E(q)|$) **(b)** PDBS (vary $|E(q)|$)

(c) PCM (vary $|E(q)|$) **(d)** PPI (vary $|E(q)|$)

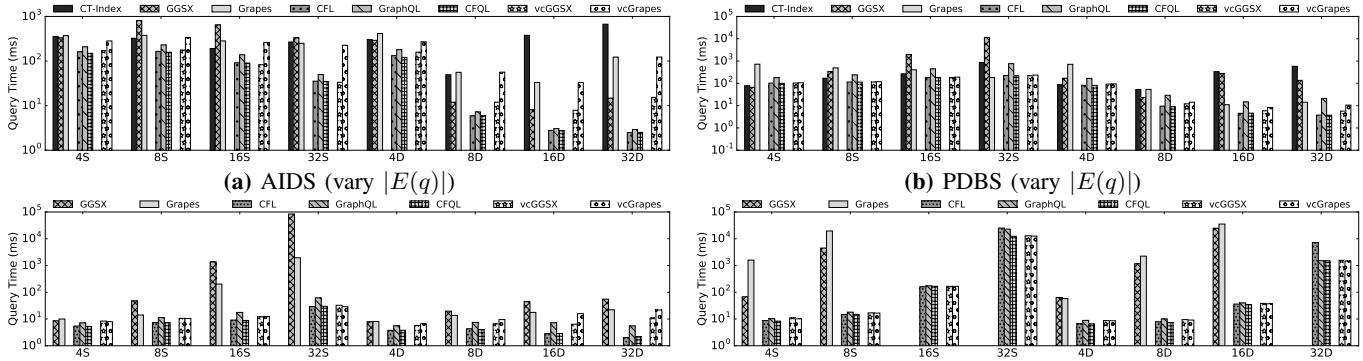
Fig. 6: The number of candidate graphs on the real-world datasets.

the performance improvement of the verification step is due in large part to the subgraph matching algorithm.

The per SI test time of CFQL is less than that of CFL, which shows that the join-based ordering in GraphQL tends to be better than the path-based ordering of CFL. Moreover, CFL fails to complete 26 queries of the total 3200 queries within the time limit, whereas CFQL fails to finish 15 queries. This result shows that the join-based ordering is more robust than the path-based ordering. CFQL takes very little time to conduct SI test on a data graph in AIDS, PDBS and PCM, which either has a small degree value or has a small number of vertices. Comparing the filtering time in Figure 3 with the verification time in Figure 4, we find that although the subgraph isomorphism problem is NP-complete, the filtering time is more than the verification time on AIDS, PDBS and PCM. However, on some challenging datasets, the verification still consumes a lot of time, for example, the SI test of a query graph in Q_{32S} against a data graph in PPI spends more than 2000 ms.

4) Evaluation on the query time: Figure 7 presents the query time on the real-world datasets. Because CFQL is the fastest among vcFV algorithms, we use it as the representative of vcFV algorithms. CFQL outperforms IFV algorithms, benefiting from the state-of-the-art subgraph matching algorithm in its verification phase. In the following, we focus on comparing CFQL with vcGrapes and vcGGSX, which have the same verification method as CFQL.

As discussed in Section IV-B3, the filtering time of CFQL, vcGrapes and vcGGSX is longer than the verification time on AIDS, PDBS and PCM, i.e., the filtering time dominates the



(a) AIDS (vary $|E(q)|$) **(b)** PDBS (vary $|E(q)|$)

(c) PCM (vary $|E(q)|$) **(d)** PPI (vary $|E(q)|$)

Fig. 7: Query time on the real-world datasets.

TABLE VII: Memory cost on the real-world datasets (MB).

	AIDS	PDBS	PCM	PPI
Datasets	28.1	27.5	7.2	4.2
CFQL	0.055	3.627	0.150	2.576
CT-Index	338	317	N/A	N/A
GGSX	109	4	1,138	146
Grapes	254	72	3,302	831

query time on the three datasets. Because the filtering time of CFQL is less than that of vcGrapes and vcGGSX, CFQL outperforms vcGrapes and vcGGSX on the three datasets. On PPI, the verification time is much more than the filtering time. As CFQL, vcGrapes and vcGGSX have a similar filtering precision, their performance is very close. Overall, CFQL, a subgraph query algorithm obtained by simply modifying the subgraph matching algorithm, is competitive with vcGrapes and vcGGSX, the top-performing IFV algorithms integrated with the state-of-the-art verification method.

5) Evaluation of the memory cost: Table VII presents the memory cost on the real-world datasets. For brevity, we only present the memory cost of CFQL as the representative of vcFV algorithms. We also omit the memory cost of vcGrapes and vcGGSX, because they are obtained by integrating Grapes and GGSX with CFQL respectively whose memory cost is at most the sum of CFQL and their original versions. The memory cost of the *datasets* in the table is the memory consumed by the data graphs that are stored in the CSR format (i.e., a label array, an offset array and an edge array). In the IFV-based algorithms, the memory cost can be exponential to the graph size (e.g, PCM and PPI). In contrast, CFQL consumes less than 5MB memory on all data graphs to store its auxiliary data structure, because the space complexity of CFQL is $O(|V(q)| \times |E(G)|)$.

C. Results on Synthetic Datasets

In this subsection, we run the competing algorithms on the synthetic datasets to compare their scalability with respect to several properties of datasets. We use CFQL as the representative of vcFV algorithms, as it performs the best among vcFV algorithms. In order to obtain results of Grapes and GGSX whose verification algorithms are very slow, we conduct the experiments on the synthetic datasets with Q_{8S} and Q_{8D}. The efficiency of the SI test depends on the integrated subgraph matching algorithm. When the competing algorithms adopt the same subgraph matching method, the key factor affecting the query performance is the filtering step. Therefore, we mainly examine the filtering step in terms of the filtering precision and the filtering time. Additionally, we evaluate the memory cost of the competing algorithms.

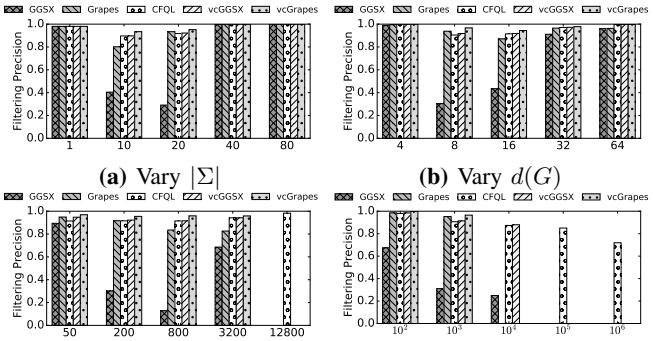
TABLE VIII: Indexing time on synthetic datasets(seconds).

$\|\Sigma\|$	**1**	**10**	**20**	**40**	**80**
CT-Index	OOT	OOT	OOT	OOT	OOT
GGSX	28	105	194	184	167
Grapes	5	23	76	105	140
$\boldsymbol{d(G)}$	**4**	**8**	**16**	**32**	**64**
CT-Index	9,653	OOT	OOT	OOT	OOT
GGSX	26	216	1,131	4,220	18,541
Grapes	13	79	275	706	2,807
$\boldsymbol{\|V(G)\|}$	**50**	**200**	**800**	**3200**	**12800**
CT-Index	OOT	OOT	OOT	OOT	OOT
GGSX	35	198	1,078	1,830	8,630
Grapes	18	74	267	464	OOM
$\boldsymbol{\|\mathcal{D}\|}$	$\mathbf{10^2}$	$\mathbf{10^3}$	$\mathbf{10^4}$	$\mathbf{10^5}$	$\mathbf{10^6}$
CT-Index	62,178	OOT	OOT	OOT	OOT
GGSX	11	174	3,256	OOM	OOM
Grapes	7	73	OOM	OOM	OOM

1) Evaluation of index construction: Table VIII presents the indexing time on the synthetic datasets with $|\Sigma|$, $d(G)$, $|V(G)|$ and $|\mathcal{D}|$ varied respectively. If indexing cannot be completed within the time limit, we mark them as out-of-time (OOT). Indexing that runs out of memory is marked as out-of-memory (OOM). Due to the poor time efficiency, CT-Index fails to build indices on most of the cases. So, we omit it in the subsequent comparison. Although Grapes and GGSX run much faster than CT-Index, they consume a large amount of memory, and fail on some large datasets due to the OOM error. Both Grapes and GGSX also take a lot of time on the index construction, for example, when $d(G) = 64$ or $|V(G)| = 12800$. Thus, the index construction of the IFV algorithms severely restricts their scalability.

2) Evaluation of the filtering step: Because the experimental results on the sparse queries (Q_{8S}) and the dense queries (Q_{8D}) have the same trends with the properties of the datasets varied, we only present the experimental results on Q_{8S}.

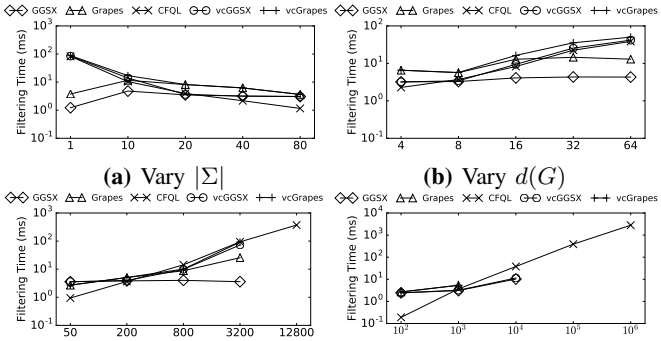
Evaluation on the filtering precision. The filtering precision results on the synthetic datasets are shown in Figure 8. Both CFQL and Grapes significantly outperform GGSX. vcGrapes slightly outperforms CFQL and Grapes, because the filtering precision of CFQL and Grapes is reasonable, which is greater than 0.75 on all cases.



(a) Vary $|\Sigma|$ **(b)** Vary $d(G)$

(c) Vary $|V(G)|$ **(d)** Vary $|\mathcal{D}|$

Fig. 8: Filtering precision on the synthetic datasets.



(a) Vary $|\Sigma|$ **(b)** Vary $d(G)$

(c) Vary $|V(G)|$ **(d)** Vary $|\mathcal{D}|$

Fig. 9: Filtering time on the synthetic datasets.

The filtering precision of these algorithms increases with the increase of $|\Sigma|$ from 10 to 80, because more distinct labels result in fewer occurrences of features for IFV algorithms and fewer candidate data vertices for CFQL. In particular, we find that the algorithms return all data graphs as candidate graphs when there is only one label, which is equivalent to having no label information, because without labels, data graphs generally contain simple features (e.g., a path whose length is 4) of the query graph for IFV algorithms, and data vertices cannot be ruled out by the pruning strategy of CFQL. In other words, the filtering strategies of these algorithms, including both IFV and vcFV algorithms, cannot work well in the unlabeled context. However, the algorithms achieve high filtering precision in this case, because most data graphs contain the query graphs.

The growth of $d(G)$ first leads to a decrease of the filtering precision and followed by an increase. This phenomenon is because when the data graph is very sparse, it contains fewer features/candidate data vertices. With the increase of $d(G)$, the data graph contains more features or more vertices passing the filtering. When the data graph becomes more dense, it is more likely to contain the given query, which results in the increase of the filtering precision.

Evaluation on the filtering time. The filtering time on the synthetic datasets is presented in Figure 9. The IFV algorithms conduct the filtering based on their indices, i.e., trie structure of Grapes and suffix tree structure of GGSX. Therefore, the filtering time is mainly affected by the depth of the tree, which is 4 in our experiments. Both Grapes and GGSX take a longer time with the increase of $|V(G)|$ and $|\mathcal{D}|$, because more data graphs contain the features in the query graph. The filtering time of CFQL decreases with the increase of $|\Sigma|$, because most of the invalid candidate data vertices can be ruled out by the label filter. Because the time complexity of the filtering method in CFQL is $O(|E(q)| \times |E(G)|)$, the filtering time of CFQL is roughly linear to $d(G)$, $|V(G)|$ and $|\mathcal{D}|$. Therefore, CFQL has a good scalability and completes the filtering step on all test cases within 3 seconds.

3) Evaluation of the memory cost: Table IX presents the memory cost on the synthetic datasets. Because the space complexity of CFQL is $O(|V(q)| \times |E(G)|)$, CFQL consumes less than 1.5MB memory on all the test cases. In contrast, both Grapes and GGSX consume a large amount of memory to keep their indices.

TABLE IX: Memory cost on the synthetic datasets (MB).

Vary $\|\Sigma\|$	**1**	**10**	**20**	**40**	**80**
Datasets	7.8	7.8	7.8	7.8	7.8
CFQL	0.0320	0.0258	0.0237	0.0214	0.0348
GGSX	0.3242	649	3,650	5,937	10,502
Grapes	0.5586	1,171	6,443	10,267	15,483
Vary $d(G)$	**4**	**8**	**16**	**32**	**64**
Datasets	4.8	7.8	13.7	25.9	50.2
CFQL	0.0222	0.0214	0.0167	0.0160	0.0180
GGSX	779	3,668	8,532	9,842	9,957
Grapes	1,360	6,475	15,181	17,756	18,152
Vary $\|V(G)\|$	**50**	**200**	**800**	**3200**	**12800**
Datasets	1.9	7.7	35.8	122.1	491.6
CFQL	0.0044	0.0205	0.1679	0.1857	1.0286
GGSX	1,116	3,679	7,723	2,607	2,608
Grapes	1,940	6,492	19,265	19,658	N/A
Vary $\|\mathcal{D}\|$	$\mathbf{10^2}$	$\mathbf{10^3}$	$\mathbf{10^4}$	$\mathbf{10^5}$	$\mathbf{10^6}$
Datasets	0.8	8	77	778	7,787
CFQL	0.0193	0.0227	0.0387	0.0424	0.0437
GGSX	381	3,647	36,317	N/A	N/A
Grapes	661	6,437	N/A	N/A	N/A

D. Discussion

Through the experiments, we reveal the impact of the advancement in subgraph matching on subgraph query processing on two aspects

Impact of the performance improvement in subgraph matching. Because latest subgraph matching algorithms can significantly accelerate the verification phase, the slow verification method in existing IFV algorithms can lead us to overestimate the gain of filtering. Take Q_{8S} on PPI as an example (see Figure 5d): Using VF2 in verification saves around 1000 ms on average by reducing one candidate data graph. However, using CFQL, we only save 0.2 ms, because the verification of CFQL is fast. Moreover, with the performance improvement of the verification, the filtering time can be much longer than the verification time (e.g., queries on AIDS, PDBS and PCM), although the subgraph isomorphism problem is NP-complete. However, improving the filtering precision is still important especially when the query graph and the data graph become large. For example, per SI test of Q_{32S} on PPI consumes more than 2000 ms on average even with CFQL (see Figure 5d). Therefore, improving the filtering precision is justified only when the verification phase is the bottleneck of the query.

Impact of the methodology advancement in subgraph matching. We find that the latest preprocessing-enumeration

subgraph matching algorithms can be easily modified to subgraph query processing algorithms that also follow the filtering-verification paradigm. The filtering is based on the vertex connectivity (i.e., the auxiliary data structure constructed dynamically based on the structure of q and G) instead of the index in IFV algorithms. Our experimental results show that the filtering precision of vcFV algorithms is competitive with that of the top-performing IFV algorithms, although the preprocessing methods are based on heuristics. Moreover, the filtering process of vcFV algorithms is very fast because of the good time complexity of the preprocessing method in state-of-the-art subgraph matching algorithms. Additionally, the memory consumption of vcFV algorithms is small. As a result, vcFV algorithms can scale up subgraph queries to hundreds of thousands of data graphs and graphs of tens of thousands of vertices, without the index in IFV algorithms.

V. Conclusions

In this paper, we study eight subgraph query processing algorithms in three categories: (1) the traditional IFV algorithms; (2) our vcFV algorithms taking the preprocessing-enumeration subgraph matching paradigm for subgraph query processing; and (3) the integration of the first two categories of algorithms. Our results show that working without an index, vcFV algorithms eliminate the problems of index scalability and index update cost, and outperform the IFV algorithms on both query time and scalability. Also, vcFV algorithms can work on the graphs that are frequently updated. Our source code is publicly available at https://github.com/RapidsAtHKUST/SubgraphContainment.

VI. Acknowledgments

This work was partly supported by grants 16206414 from the Hong Kong Research Grants Council and MRA11EG01 from Microsoft.

References

[1] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang. Efficient subgraph matching by postponing cartesian products. In *SIGMOD*, 2016.
[2] V. Bonnici, A. Ferro, R. Giugno, A. Pulvirenti, and D. Shasha. Enhancing graph database indexing by suffix tree structure. In *Pattern Recognition in Bioinformatics*, 2010.
[3] J. Cheng, Y. Ke, and W. Ng. Efficient query processing on graph databases. In *TODS*, 2009.
[4] J. Cheng, Y. Ke, W. Ng, and A. Lu. Fg-index: towards verification-free query processing on graph databases. In *SIGMOD*, 2007.
[5] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, 1971.
[6] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2004.
[7] R. Di Natale, A. Ferro, R. Giugno, M. Mongiovì, A. Pulvirenti, and D. Shasha. Sing: Subgraph search in non-homogeneous graphs. In *BMC bioinformatics*, 2010.
[8] I. S. Duff, K. Kaya, and B. Uçcar. Design, implementation, and analysis of maximum transversal algorithms. In *TOMS*, 2011.
[9] Z. Fan, B. Choi, J. Xu, and S. S. Bhowmick. Asymmetric structure-preserving subgraph queries for large graphs. In *ICDE*, 2015.
[10] R. Giugno, V. Bonnici, N. Bombieri, A. Pulvirenti, A. Ferro, and D. Shasha. Grapes: A software for parallel searching on biological graphs targeting multi-core architectures. In *PloS one*, 2013.
[11] W.-S. Han, J. Lee, and J.-H. Lee. Turbo iso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *SIGMOD*, 2013.
[12] W.-S. Han, J. Lee, M.-D. Pham, and J. X. Yu. igraph: a framework for comparisons of disk-based graph indexing techniques. In *PVLDB*, 2010.
[13] H. He and A. K. Singh. Closure-tree: An index structure for graph queries. In *ICDE*, 2006.
[14] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *SIGMOD*, 2008.
[15] F. Katsarou, N. Ntarmos, and P. Triantafillou. Performance and scalability of indexed subgraph query processing methods. In *PVLDB*, 2015.
[16] F. Katsarou, N. Ntarmos, and P. Triantafillou. Hybrid algorithms for subgraph pattern queries in graph databases. In *IEEE International Conference on Big Data*, 2017.
[17] F. Katsarou, N. Ntarmos, and P. Triantafillou. Subgraph querying with parallel use of query rewritings and alternative algorithms. In *EDBT*, 2017.
[18] H. Kim, J. Lee, S. S. Bhowmick, W.-S. Han, J. Lee, S. Ko, and M. H. Jarrah. Dualsim: Parallel subgraph enumeration in a massive graph on a single machine. In *SIGMOD*, 2016.
[19] R. Kimmig, H. Meyerhenke, and D. Strash. Shared memory parallel subgraph enumeration. In *IPDPSW*, 2017.
[20] K. Klein, N. Kriege, and P. Mutzel. Ct-index: Fingerprint-based graph indexing combining cycles and trees. In *ICDE*, 2011.
[21] L. Lai, L. Qin, X. Lin, and L. Chang. Scalable subgraph enumeration in mapreduce. In *PVLDB*, 2015.
[22] L. Lai, L. Qin, X. Lin, Y. Zhang, L. Chang, and S. Yang. Scalable distributed subgraph enumeration. In *PVLDB*, 2017.
[23] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. In *PVLDB*, 2012.
[24] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo. Strong simulation: Capturing topology in graph pattern matching. In *TODS*, 2014.
[25] M. Qiao, H. Zhang, and H. Cheng. Subgraph matching: on compression and computation. In *PVLDB*, 2017.
[26] X. Ren and J. Wang. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. In *PVLDB*, 2015.
[27] X. Ren and J. Wang. Multi-query optimization for subgraph isomorphism search. In *PVLDB*, 2016.
[28] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. In *PVLDB*, 2008.
[29] Y. Shao, B. Cui, L. Chen, L. Ma, J. Yao, and N. Xu. Parallel subgraph listing in a large-scale graph. In *SIGMOD*, 2014.
[30] D. Shasha, J. T. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2002.
[31] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. In *PVLDB*, 2012.
[32] J. R. Ullmann. An algorithm for subgraph isomorphism. In *JACM*, 1976.
[33] J. Wang, N. Ntarmos, and P. Triantafillou. Indexing query graphs to speed up graph query processing. In *EDBT*, 2016.
[34] J. Wang, N. Ntarmos, and P. Triantafillou. Graphcache: a caching system for graph queries. In *EDBT*, 2017.
[35] D. W. Williams, J. Huan, and W. Wang. Graph database indexing using structured graph decomposition. In *ICDE*, 2007.
[36] Y. Xie and P. S. Yu. Cp-index: on the efficient indexing of large graphs. In *CIKM*, 2011.
[37] X. Yan, P. S. Yu, and J. Han. Graph indexing: a frequent structure-based approach. In *SIGMOD*, 2004.
[38] D. Yuan and P. Mitra. Lindex: a lattice-based index for graph databases. In *VLDBJ*, 2013.
[39] D. Yuan, P. Mitra, H. Yu, and C. L. Giles. Updating graph indices with a one-pass algorithm. In *SIGMOD*, 2015.
[40] S. Zhang, M. Hu, and J. Yang. Treepi: A novel graph indexing method. In *ICDE*, 2007.
[41] P. Zhao and J. Han. On graph query optimization in large networks. In *PVLDB*, 2010.
[42] P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: tree+ delta¡= graph. In *PVLDB*, 2007.
[43] L. Zou, L. Chen, J. X. Yu, and Y. Lu. A novel spectral coding in a large graph database. In *EDBT*, 2008.