Efficient Subgraph Search with Presorting and Indexing on Label Frequency

Haichuan Shang Masaru Kitsuregawa
Institute of Industrial Science
University of Tokyo
{shang,kitsure}@tkl.iis.u-tokyo.ac.jp

ABSTRACT

Graphs are widely used to model complicated data semantics in many applications. In this paper, we aim to develop efficient techniques to retrieve graphs, containing a given query graph, from a large set of graphs. Considering the problem of testing subgraph isomorphism is generally NP-hard, most of the existing techniques are based on the framework of filtering-and-verification to reduce the precise computation costs; consequently various novel feature-based indexes have been developed. While the existing techniques work well for small query graphs, the verification phase becomes a bottleneck when the query graph size increases. Motivated by this, in the paper we firstly propose a novel and efficient algorithm for testing subgraph isomorphism. Secondly, we develop a new feature-based index technique to accommodate the proposed algorithm in the filtering phase. Our extensive experiments on real and synthetic data demonstrate the efficiency and scalability of the proposed techniques, which significantly improve the existing techniques.

1. INTRODUCTION

Many recent real applications strongly demand efficiently and effectively managing graph structured data such as paths, trees, and general graphs. These applications include Bioinformatics, Chemistry, Social networks, Software and Data Engineering, World Wide Web, etc. In such applications, graphs are used to model complex structures and relationships. For instance, graphs may represent chemical compounds in Chemistry. Graphs are also used in UML and ER diagrams.

The subgraph containment query problem can be described as follows. Given a graph database $D = \{g_1, g_2, ..., g_n\}$ and a query graph q, retrieve all graph $g_i \in D$ such that q is a subgraph of g_i. For example, if we use the graph in Figure 1 as the query q, then among the 3 graphs ($D = \{g_a, g_b, g_c\}$) in Figure 2, only graph g_b contains q. The subgraph containment (or subgraph isomorphism) problem has been shown NP-complete.

In recent years, a number of techniques for processing subgraph containment queries have been proposed [6, 8, 2]. The main paradigm follows the framework of *filtering*-and-*verification* which is based on feature-based indexes. In the filtering phase, a feature-based index is used to prune the captured negative results and generate a candidate set. In the verification phase, a precise computation is conducted to generate the final results based on subgraph isomorphism (NP-complete). The existing techniques include gIndex [6], TreePi [7] and TreeDelta [8].

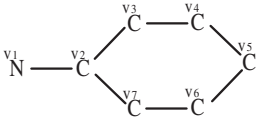


Figure 1: Simple Query Graph

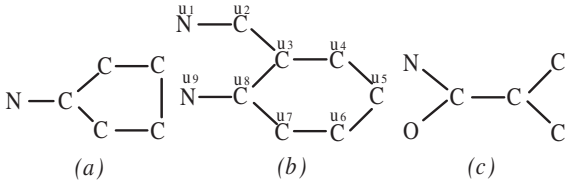


Figure 2: Simple Graph Database

However, the existing verification techniques are not efficient especially when the query graph size becomes large. Note that the larger graphs the higher cost for subgraph isomorphism testing. Moreover, due to intrinsic limits of feature-based indexes, the accuracy of filtering may be getting worse while graph sizes are increasing; that is, the ratio of the generated candidate set size over the actual result set size is getting larger. This leads to a dramatic performance degrade with an increment of query graph sizes. In [2], Cheng et al. propose a new paradigm, FG-Index, with the aim to use index only to process a subgraph containment query; that is, verification free. Nevertheless, when query graph sizes increase, many graphs still remain for a verification.

Motivated by these, in this paper, our primary focus is on developing efficient verification techniques. We propose an efficient subgraph isomorphism testing algorithm QuickSI (Quick Subgraph Isomorphism) to conduct a verification to generate final results. Comparing to the well adopted Ullman's algorithm [5], QuickSI achieves up to 1-4 orders of magnitude speed-up. In addition, our verification techniques can also be used in the filtering phase to efficiently generate candidates.

Our main contributions are summarized as follows.

- To significantly reduce the verification costs, we develop an efficient subgraph isomorphism testing algorithm QuickSI. Several new techniques are proposed. Firstly, we propose QI-Sequence, for a given query graph, to bound the search space in the subgraph isomorphism testing. Secondly, we determine the QI-Sequence order based on the frequencies of features that appear in the underneath graph database. The QI-Sequence order further reduces the search space. With the two techniques, our new algorithm QuickSI significantly improves the existing verification techniques by up to 4 orders of magnitudes speed-up.

- In addition, we develop a novel index called Swift-Index where the mined tree features are represented as QI-Sequences and all QI-Sequences in the index are organized as a prefix tree. The prefix tree index makes it possible to significantly reduce the cost in the filtering phase by sharing the cost of subgraph isomorphism testing. Note that in order to check whether or not a graph contains a query graph, in the filtering phase, all the existing algorithms need to check if the graph contains all the indexed features that are contained in the query graph (subgraph isomorphism). Sharing reduces the cost of checking the common parts of several features. Our Swift-Index significantly outperforms the filtering techniques used in $gIndex$.

Experimental results show that our new techniques significantly outperform the most recent, efficient technique, FG-index [2] towards both index construction and query processing when query graph size is not very small. Against real data set, our query processing techniques can achieve up to an order of magnitude speed-up over FG-Index while the index size is 20% of FG-Index. In addition, the results also show that our techniques have high scalability on the database size, the graph size and the number of distinct labels.

The rest of the paper is organized as follows. Section 2 presents the problem statement and the framework. Section 3 introduces a new verification approach and a new subgraph isomorphism testing algorithm called QuickSI. Section 4 proposes a new filtering approach with a new prefix-tree index called Swift-Index. Experimental studies are reported in Section 5. The conclusion is given in Section 6, respectively.

2. THE PROBLEM STATEMENT AND THE FRAMEWORK

We firstly give our problem statement on subgraph containment queries (or subgraph isomorphism queries). Then, we outline the framework of $filtering$-and-$verification$ followed by an overview towards the most related work - Ullman's Algorithm for verification. For presentation simplicity, graphs to be studied in the paper are "simple" undirected graphs; nevertheless, our results can be immediately extended to cover directed and/or multigraphs.

2.1 Problem Statement

A graph is *simple* if it has no loops nor multiple edges. Given two sets of labels, Σ_V and Σ_E, a *labeled* graph g is defined as a triple (V, E, l) where V is the set of vertices, $E \subseteq V \times V$ is the set of undirected edges, and l is a mapping function: $V \rightarrow \Sigma_V$ and $E \rightarrow \Sigma_E$. We denote the vertex set and the edge set of a graph g as $V(g)$ and $E(g)$, respectively. Given an edge $(u,\ v) \in E(g)$ and the mapping function l of g, $l(u)$, $l(v)$ are the labels of u and v in g and $l(u, v)$ is the label of the edge (u, v) in g. We use $|V(g)|$ and $|E(g)|$ to represent the number of vertices and the number of edges, respectively.

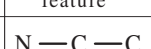
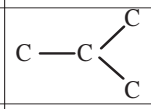
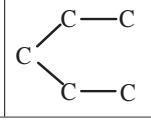
	feature	graph ID-list
f_1		{a, b, c}
f_2		{c}
f_3		{a, b}

Figure 3: A Sample of Feature-based Index

Definition 1. (SUBGRAPH ISOMORPHISM) Given two graphs $g' = (V', E', l')$ and $g = (V,\ E,\ l)$, g' is *subgraph-isomorphic* to g, denoted as $g' \subseteq g$, if there is an injective function $f:\ g' \rightarrow g$ such that

1. $\forall v \in V'$, $f(v) \in V(g)$ such that $l'(v) = l(f(v))$.
2. $\forall (u, v) \in E'$, $(f(u), f(v)) \in E$ such that $l'(u, v) = l(f(u), f(v))$.

A graph g' is a *subgraph* of g if g' is subgraph-isomorphic to g where g is also called a supergraph of g', denoted by $g' \subseteq g$. We may also simply say that g contains g'. A subgraph $IndG(V', g)$ of g is *induced* if it is the maximum subgraph for a given subset V' of $V(g)$; that is, $IndG(V', g)$ consists of all edges in g with the vertices in V'.

Definition 2. (SUBGRAPH CONTAINMENT QUERY) Given a graph database $D = \{g_1, g_2, ..., g_n\}$ and a query graph q, the problem of subgraph containment query (or subgraph isomorphism query) is to find a set of graphs which contain q from D, such as $D_q = \{g | g \in D \wedge q \subseteq g\}$.

Problem Statement. In this paper, we will develop efficient algorithms to process subgraph containment queries. In the rest of the paper, we assume edges are not labeled; nevertheless our techniques can be immediately extended to cover *edge-labeled* graphs.

2.2 Filtering and Verification Framework

The framework of *filtering*-and-*verification* is presented in Algorithm 1, where a feature-based index plays the key role in the framework.

FEATURE-BASED INDEX. A feature based index $I = \{(f_i, f_i.list)\}$ is a set of indexed items, $(f_i, f_i.list)$. Here, f_i is a fragment (or subgraph) of a graph, which can be a path, a tree, or a graph. And $f_i.list$ is a list of graph identifiers for the graphs that contain the subgraph, such as $f_i.list = \{g_i.ID | f_i \subseteq g_i \wedge g_i \in D\}$. (Note that we use $g_i.ID$ to denote the graph identifier of graph g_i.) Below, we call f_i a feature and $f_i, list$ its graph ID-list (or simply ID-list).

Algorithm 1: **QueryProecssing**(q, I, D)

Input : q is a query graph;
I is a graph index;
D is a graph database;
Output: R is a set of matched graphs;

```
1 F := {f_i | f_i ⊆ q ∧ f_i ∈ I};
2 C := ⋂_{f_i ∈ F} f_i.list;
3 R := ∅;
4 for each g ∈ C do
5     if q ⊆ g then
6         R := R ∪ {g};
7 return R
```

EXAMPLE 1. *The feature f_1 in Figure 3 is contained by all three graphs in Figure 2, therefore its ID-list $f_1.list = \{a, b, c\}$. As the feature f_2 is only contained by graph (b) in Figure 2, its ID-list $f_2.list = \{b\}$.*

As shown in Algorithm 1, the filtering phase and the verification phase are specified in line 1-2 and line 4-6, respectively. Line 1 retrieves the features, which are contained in the query graph q, from the feature-based index I. Line 2 gets all graph identifiers for the graphs that contain all the features appearing in the query graph, which is known as the candidate set C. Line 4-6 process subgraph isomorphism testing for each graph g whose graph identifier is in C. If there is a subgraph isomorphism mapping from q to g, $q \subseteq g$, g is added to the result set R. Obviously, $q \not\subseteq g$ if $|V(g)| < |V(q)|$. Line 7 finally returns the matched result set.

In the next subsection, we introduce the Ullman's algorithm which is widely used for subgraph isomorphism testing.

3. A NEW VERIFICATION APPROACH

Ullman algorithm is designed based on the branch and bound paradigm [4]. In such a paradigm, one of the critical issues is how to choose an effective search order so that it can cut as many branches as possible in searching. It is important to know that the search order in the Ullman algorithm is random, and a random order can possibly result in a search order that seriously slows the algorithm. An example is shown to explain.

EXAMPLE 2. *Suppose that in Ullman algorithm, it determines if a given query graph q (Figure 1) is sub-isomorphic to the graph g_b (Figure 2(b)) by visiting the vertices in the query graph q according to the following visiting order: v_1, v_3, v_2, v_4, v_5, v_6, and v_7. Assume that v_1 and v_3 have been visited. There are 14 pairs of vertices with labels N and C in g_b that need to be considered (2 N-labeled vertices, and 7 C-labeled vertices). In fact, there are only three pairs of vertices in g, namely, $\langle u_1, u_3\rangle$, $\langle u_9, u_3\rangle$ and $\langle u_9, u_7\rangle$ need to be considered.*

In order to reduce the search space, in this paper, we propose QI-Sequence to encode a graph for efficient subgraph isomorphism testing. In brief, we encode a search order and topological information in QI-Sequence for a query graph q, and we determine the effective search order using the frequencies of features that appear in the underneath graph database D. Following the search order and other topological information specified in the QI-Sequence for q, we identify the mapping between q and g. Such encoding and ordering can significantly reduce the unnecessary branch and bound, and is shown effective in our extensive experimental studies.

The rest of this section is organized as follows. Section 3.1 introduces QI-Sequence to encode a query graph. Section 3.2 presents an efficient algorithm QuickSI to test if the query graph q is sub-isomorphic to a data graph, based on the QI-Sequence of q. In Section 3.3, we discuss how to determine an effective QI-Sequence, as a search order, by effectively utilizing feature frequencies in the graph database.

3.1 QI-Sequence

Given a query graph q of size β in terms of the number of vertices in q, a QI-Sequence is a sequence that represents a rooted spanning tree for q. It consists of a list of *spanning entries*, T_i, for $1 \le i \le \beta$, where each T_i keeps the basic information of the spanning tree of q. In QI-Sequence, a T_i may be followed by a list of *extra entries*, R_{ij}, which keeps the extra topology information related to the corresponding spanning entry.

Formally, a QI-Sequence of q is represented as a *regular expression* $SEQ_q = [[T_i R_{ij}^*]^\beta]$. Here, T_i contains several information. Firstly, $T_i.v$ records a vertex v_k in a query graph q, for example, $T_i.v := v_k$. Secondly, T_i keeps a pair, $[T_i.p, T_i.l]$, where $T_i.p$ stores the parent vertex of $T_i.v$ in the spanning tree and $T_i.l$ stores the label of $T_i.v$. It is important to note that the subindex i of T_i specifies the search order. As for R_{ij}, there are two kinds of extra entries in R_{ij}, namely, degree constraint and extra edge. The *degree constraint* is in the form of $[deg : d]$, where d is the degree of v_i.[1] The *extra edge* (i.e., edge that does not appear in the spanning tree) is in the form of $[edge : j]$, where j indicates a vertex indicated by $T_j.v$ in SEQ_q. We only record such an extra edge, $[edge : j]$, in R_{ij} after T_i in SEQ_q if the extra edge is from $T_i.v$ to $T_j.v$ for $j < i$.

Table 1 illustrates two different QI-Sequences of the query graph, q in Figure 1, based on two different spanning trees. Note that an entry T_i in a QI-Sequence does not necessarily correspond to the vertex v_i; for instance, T_1 in the QI-Sequence (b) in Table 1 correspond to v_4. The two QI-Sequences are different. The QI-Sequence (Table 1(a)) is label selective as the possible mapping of N is less than C in graph database. On the other hand, the QI-Sequence (Table 1(b)) is random. It is clear that the two QI-Sequences will have different search spaces when processing subgraph isomorphism testings. We will discuss how to choose an effective QI-Sequence in details in Section 3.3.

Let $SEQ_{g'}$ and SEQ_g be two QI-Sequences for two graphs, g' and g. In the following Theorem 1, we show that if the two QI-Sequences are identical then the two graphs are identical. Our QI-Sequence based subgraph isomorphism testing algorithm is designed based on Theorem 1.

THEOREM 1. *Given two graphs g' and g. Let $SEQ_{g'}$ and SEQ_g be the two corresponding QI-Sequences. If the two QI-Sequences are identical, then the corresponding graphs, g' and g, must be identical.*

[1]To avoid a redundant computation, we do not record $[deg : d]$ when $d \le 2$.

Table 1: Two SEQ_qs for query graph q in Figure 1

Type	$[T_i.p, T_i.l]$	$T_i.v$
T_1	$[0,\ N]$	v_1
T_2	$[1,\ C]$	v_2
R_{21}	$[deg:3]$	
T_3	$[2,\ C]$	v_3
T_4	$[3,\ C]$	v_4
T_5	$[4,\ C]$	v_5
T_6	$[5,\ C]$	v_6
T_7	$[6,\ C]$	v_7
R_{71}	$[edge:2]$	

(a)

Type	$[T_i.p, T_i.l]$	$T_i.v$
T_1	$[0,\ C]$	v_4
T_2	$[1,\ C]$	v_5
T_3	$[1,\ C]$	v_3
T_4	$[2,\ C]$	v_6
T_5	$[4,\ C]$	v_7
T_6	$[5,\ C]$	v_2
R_{61}	$[deg:\ 3]$	
R_{62}	$[edge:\ 3]$	
T_7	$[6,\ N]$	v_1

(b)

Proof-Sketch: Theorem 1 is immediate based on the following result. A QI-Sequence $SEQ_{\mathbf{g}}$, for a graph $\mathbf{g}$, can be uniquely converted to a graph $\mathbf{g}'$ which is identical to $\mathbf{g}$.

3.2 QuickSI Algorithm

In this section, we discuss our new algorithm for subgraph isomorphism testing. Let q and g be a query graph and a graph in the candidate set after filtering phase, and let SEQ_q be the QI-Sequence for q. Our QuickSI algorithm is designed to check if there exists a QI-Sequence for a subgraph, g' of g, denoted as $SEQ_{g'}$, which is identical to SEQ_q.

The QuickSI algorithm is presented in Algorithm 2. There are five inputs. (1) SEQ_q is the QI-Sequence of a query graph q of size β ($= |V(q)|$). (2) $\mathscr{F}$ and $\mathscr{H}$ are two vectors as used in Ullman's algorithm. (3) g is a graph of size α ($= |V(g)|$), and (4) d is the current search position for $1 \leq d \leq \beta$. The algorithm adopts *depth-first-search* order following the order explicitly specified in SEQ_q.

We explain the two vectors below. Firstly, $\mathscr{H} = \{H_1, ..., H_i, ..., H_\beta\}$ is used to store mapping from the QI-Sequence SEQ_q to a graph g. $H_i := u_j$ indicates that the vertex $T_i.v$ of q has been mapped to the vertex $u_j \in g$. Given a successful mapping $H_1, H_2, ..., H_i$, the degree constraint, $[deg : x]$, specified in R_{ij}, implies that the vertex $H_i \in g$ must have the degree, $deg(H_i, g)$, not smaller than x; that is, $deg(H_i, g) \geq x$. Moreover, each edge constraint $[edge : x]$, specified in R_{ij}, implies that there must be an edge between H_i and H_x in graph g where $x < i$. Secondly, $\mathscr{F} = \{F_1, ..., F_i, ..., F_\alpha\}$ is used to indicate whether or not the ith vertex in g is used at an intermediate state of the computation.

In Algorithm 2, α and β are the numbers of vertices in g and q, respectively. We first test whether computation has reached the end of SEQ_q by checking depth d. If $d > \beta$, it implies that we have already found a QI-Sequence, $SEQ_{g'}$, for $g' \subseteq g$, that equals SEQ_q. We can conclude that q is a subgraph of g, because q is identical to g' and $g' \subseteq g$. Otherwise, we get the d-th vertex entry T_d and try to find a mapping vertex in g. If there is a vertex $u \in g$ with same label that satisfies all constraints in the extra entries R_{dj}, it can be a valid mapping, and the searching will continue recursively, until the algorithm ends up with a successful mapping or fails in all possible trials at a certain label.

EXAMPLE 3. *Consider SEQ_q (Table 1(a)) for the query graph, q, in Figure 1, and the graph g_b in Figure 2(b). The QuickSI algorithm first finds that u_1 in g_b can be mapped to T_1. It stores the mapping $H_1 := u_1$ and sets the vector element $F_1 := 1$. For the vertex set $V := \{u_2\}$ which is connected to u_1, it finds $l(u_2) = C$ which is same as $T_2.l$. When it tests the degree restriction, $[deg : 3]$, specified in R_{21}, it finds the degree of u_2 is 2, which is less than 3. The tree search algorithm returns to T_1, releases u_1 by setting $F_1 := 0$ and matches T_1 to a different vertex u_9. Finally, it finds a successful mapping $\mathscr{H} = \{u_9, u_8, u_7, u_6, u_5, u_4, u_3\}$ or $\mathscr{H} = \{u_9, u_8, u_3, u_4, u_5, u_6, u_7\}$.*

Algorithm 2: ***QuickSI***$(SEQ_q, g, \mathscr{H}, \mathscr{F}, d)$

Input : SEQ_q: QI-Sequence of query graph q;
g: a graph;
$\mathscr{H}$: a vector with length β, initialized by all 0;
$\mathscr{F}$: a vector with length α, initialized by all 0;
d: depth, initialized by 1;
Output: Boolean: SEQ_q is a subgraph of g;

```
1  if d > β then
2      return True;
3  T := T_d ∈ SEQ_q;
4  V := ∅;
5  if d = 1 then
6      V := {v | v ∈ V(g), l(v) = T.l and F_v = 0} ;
7  else
8      V := {v | v ∈ V(g), (v, H_{T.p}) ∈ E(g), l(v) = T.l and
       F_v = 0} ;
9  for each vertex v ∈ V do
10     for each restriction R_dj ∈ SEQ_q do
11         goto line-9 if R_dj is not satisfied;
12     H_d := v;
13     F_v := 1;
14     if QuickSI(SEQ, g, H, F, d + 1) then
15         return True;
16     F_v := 0;
17 return False;
```

Correctness. It can be immediately verified that if there is a QI-Sequence, $SEQ_{g'}$ for a subgraph of g, $g' \subseteq g$, that equals SEQ_q, then Algorithm 2 must be able to find it. According to Theorem 1, the correctness of the algorithm is immediate.

Cost Analysis. Note that the above subgraph isomorphism testing follows depth-first search strategy. As the search depth is fixed, the computation cost depends on the fan-out at each depth. We define *search breadth* at depth i below, denoted by B_i. Search breadth represents the number of possible isomorphism mappings from the prefix sequence $SEQ_q^i = [[T_i R_{ij}^*]^i]$; that is, $SEQ_q^i = [[T_i R_{ij}^*]^i]$ contains the first i entries in SEQ_q.

Definition 3. (SEARCH BREADTH) Given SEQ_q for a query graph q and a graph g, the search breadth $B_i = |\{\mathscr{H}^i | \mathscr{H}^i : SEQ_q^i \rightarrow g\}|$ $(1 \leq i \leq \beta)$ where $SEQ_q^i = [[T_i R_{ij}^*]^i]$ is a prefix of SEQ_q. Also, $\mathscr{H}^i$ is a distinct mapping vector from SEQ_q^i to g. The length of a $\mathscr{H}^i$ is i since $\mathscr{H}^i$ maps SEQ_q^i to g.

Given a QI-Sequence SEQ_q and a graph g, the isomorphism testing cost is computed as follows. We use T_{iso} to denote the total number of comparisons performed in the algorithm QuickQI. As we can pre-compute the degree for

data graphs, it takes $O(1)$ time to check both kinds of extra entries (degree constraint and extra edge) if an adjacent matrix is used. It takes $O(deg)$ to find a forwarding edge in a data graph g to go one depth further regarding SEQ_q, where deg is the degree of the vertex mapped to $H_{T_i.p}$ in g. (Note that $T_i.p$ points to the parent vertex of $T_i.v$.)

$$\begin{aligned} T_{iso} &= \alpha + B_1 \cdot r_1 + \Sigma_{i=1}^{\beta-1}\Sigma_{j=1}^{B_i} deg_{<i,j>} \cdot r_{i+1} \quad (1) \\ &\leq \alpha + B_1 \cdot r_1 + \Sigma_{i=1}^{\beta-1} B_i \cdot deg_{max} \cdot r_{i+1} \end{aligned}$$

Here, $deg_{<i,j>}$ is the degree of the vertex $H_{T_i.p}$ in g at j-th mapping, $r_i = 1 + |\{R_{ij}|R_{ij} \in SEQ_q\}|$ which is the number of extra entries at depth i, and deg_{max} is the maximum vertex degree of g. We have the following Theorem.

THEOREM 2. *Let $SEQ_q = [[T_i R^*_{ij}]^\beta]$ be a QI-Sequence and $deg_{max}(g)$ be the maximum vertex degree in g.*

$$T_{iso} \leq \alpha + B_1 \times deg_{max}(g)^\beta \times r_{max},$$

where r_{max} is the maximum number of extra entries for any R_{ij}.

PROOF. Because we keep connectivity during the isomorphism testing, it is immediate that if $\forall\ i \geq 2$, then

$$\begin{aligned} B_i &\leq \Sigma_{j=1}^{B_{i-1}} deg_{<i-1,j>} \\ &\leq B_{i-1} deg_{max} \end{aligned}$$

The theorem immediate follows from Eq. (1). □

The space requirement is $O(|SEQ_q| + |g|)$ where $|g|$ denotes the space required to store a graph g.

As an example, consider T_{iso} for testing whether the query graph q is sub-isomorphic to graph g_a (Figure 2(a)), using the two QI-Sequences in Table 1. With the random QI-Sequence in Table 1(b), $T_{iso} \leq 161$, whereas with the QI-Sequence in Table 1(a), $T_{iso} \leq 37$.

3.3 Effective QI-Sequence

In this section, we discuss how to determine an effective QI-Sequence, SEQ_q, for fast subgraph isomorphism testing. Reconsider Eq. (1), search breadths play an important role in subgraph isomorphism testing. Minimizing B_i can reduce cost of subgraph isomorphism testing. However, it is too costly to find the optimal QI-Sequence, in order to minimize the total breadths and therefore significantly reduce the subgraph isomorphism testing cost for any data graph in the graph database D. Instead, we develop efficient heuristics to construct an effective QI-Sequence, SEQ_q, for a query graph to reduce the total breadths and the subgraph isomorphism testing cost for any data graph in the graph database D. Our approach is based on the inner support defined below.

Definition 4. (INNER SUPPORT) Given a query graph, q, and a data graph, g, the inner support $\phi(q, g)$ is the number of isomorphic mappings from q to g.

It is immediate that the search breadth B_i is $\phi(SEQ^i_q, g)$ for a data graph.

Counting Inner Supports for Vertices and Edges. Suppose that we index all 1-vertex and 1-edge features, we can count the average inner support $\phi_{avg}(v)$ for each distinct vertex v and $\phi_{avg}(e)$ for each distinct edge e in the graph database D as follows.

$$\phi_{avg}(v) = \frac{|\{f|f(v) \in V(g) \wedge g \in D\}|}{|\{g|f(v) \in V(g) \wedge g \in D\}|} \quad (2)$$

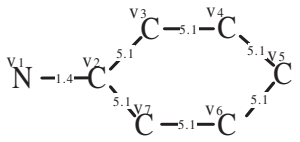


Figure 4: The Weight Graph

Table 2: Average Inner Support

vertices	$\phi(v)$
N	1.5
C	6.1

edges	$\phi(e)$
(N, C)	1.4
(C, C)	5.1

$$\phi_{avg}(e) = \frac{|\{f|f(e) \in E(g) \wedge g \in D\}|}{|\{g|f(e) \in E(g) \wedge g \in D\}|} \quad (3)$$

The average inner support $\phi_{avg}(e)$ ($\phi_{avg}(v)$) of an edge e (of a vertex v) is the average number of its possible mappings in the graphs which contain this edge (vertex). In Eq. (3), we omit the graphs which do not contain any mapping of the given edge, because these graphs will be pruned in the filtering step. Therefore, only the graphs that contain at least one mapping of the given edge need to be tested. Eq. (3) counts the average number of distinct edges in the graphs which have high probability not to be pruned in the filtering step. It is reasonable because only the graphs which pass the filtering step need to be tested in the verification step. The statistic for the graphs in the candidate set after filtering is more valuable than the statistic for all graphs in the database.

Finding Minimum Spanning Tree. Once the average inner supports of each distinct vertex and edge are counted, we add those supports to the vertices and edges of q and convert q to a weighted graph q^w, where each edge e in q^w has a weight $w(e) = \phi_{avg}(e)$ and $w(v) = \phi_{avg}(v)$. Then, we find the minimum spanning tree in q^w based on edge weights. The minimum spanning tree will be used to generate a QI-Sequence of q and we will use the vertex weights to determine the order of the first two entries in such a QI-Sequence.

We extend *Prim's algorithm* [1] to compute the minimum spanning tree for q^w and construct the QI-Sequence for q. Our extension contains the technique to choose a "better" minimum spanning tree when more than one minimum spanning tree are involved. The main idea is presented in Algorithm 3.

In Algorithm 3, V_T and E_T store the the set of vertices and edges in intermediate steps. P is the set of current possible edges which will be chosen to the spanning tree. SEQ_q will be refined as follows to fix the order of the first two vertices to generate a QI-Sequence of q. Suppose that (u, v) is the first edge in SEQ_q. If $\phi(u) \neq \phi(v)$, we pick one of them with lower average inner support as the first vertex. Otherwise, we choose one with higher degree. If the degrees are also equal, we randomly select one.

SelectFirstEdge (Algorithm 4) and *SelectSpanningEdge* (Algorithm 5) in Algorithm 3 deal with cases when there are several candidate edges in P with the same weight. Our algorithm will choose the edge which make the induce subgraph of the current vertex set V_T as dense as possible.

EXAMPLE 4. *Suppose we have a graph q as shown in Figure 1(a) and the average frequency is shown in Table 2, the*

Algorithm 3: ***SpanningTree***(q^w)

Input : q^w: weighted query graph;
Output: T: a minimum spanning tree of q;
SEQ_q: a QI-Sequence;

1 $V_T := \emptyset$;
2 $E_T := \emptyset$;
3 $SEQ_q := \emptyset$;
4 $P := \{e | e \in E(q^w) \wedge \forall(e' \in E)\exists(q^w)\exists w(e) \leq w(e')\}$;
5 $e := SelectFirstEdge(P,\ q^w)$;
6 $E_T := \{e\}$,$SEQ_q \leftarrow e$, $V_T := \{e.u, e.v\}$;
7 Remove e from q^w;
8 **while** $V_T \neq V(q^w)$ **do**
9 $P := \{e | e \in E(q^w) \wedge e.u \in V_T \wedge e.v \notin V_T\}$;
10 $e := SelectSpanningEdge(P,\ q^w,\ V_T)$;
11 $E_T := E_T \cup \{e\}$, $SEQ_g \leftarrow e$, $V_T := V_T \cup \{e.v\}$;
12 Remove e from q^w;
13 **for each** $e \in q^w$ **satisfying** $e.u \in V_T \wedge e.v \in V_T$ **do**
14 Sort them by the increasing order of $w(e)$;
15 $SEQ_q \leftarrow e$ and remove e from q^w;
16 $T := (V_T, E_T)$;
17 **return** T, SEQ_q;

Algorithm 4: ***SelectFirstEdge***$(P,\ q^w)$

Input : P: a set of edges;
q^w: a weight graph;
Output: e: an edge in P;

1 **if** $|P| > 1$ **then**
2 $P := \{e | e \in P \wedge \forall(e' \in P)\exists(deg(e.u, q^w) + deg(e.v, q^w) \leq deg(e'.u, q^w) + deg(e'.v, q^w))\}$;
3 Randomly select an edge $e \in P$;
4 **return** e;

weight graph q^w is calculated as Figure 4. In the weight graph q^w, there are only 1 edge (v_1, v_2) which has the minimum weight 1.4. Therefore, we will select it as the first edge of the minimum spanning tree. Afterwards, as the edges (v_2, v_3) and (v_2, v_7) have the same weight, both of them are selected to the set P. In the function SelectSpanningEdge, we find that they have same induced subgraph and the degree of v_3 and v_7 are also same. Thus we randomly choose one of them, for example, v_3. Assume the edges (v_3, v_4),(v_4, v_5),(v_5, v_6) and (v_5, v_6) subsequently become the spanning edges. We add (v_2, v_7) to the SEQ_q as an extra entry.

4. FILTERING-AND-VERIFICATION

Our filtering-and-verification algorithm is shown in Algorithm 6, called QI-Framework, based on the QuickSI algorithm shown in Algorithm 2. Given a query graph q, it first obtains the candidate set, C, by calling a Filtering procedure (line 1) which we will discuss in the next section. Next, it iteratively checks every graph g_i in the candidate set C and inserts g_i into final result if q is contained by g_i by calling QuickSI (line 3-7). It is worth noting that it only needs to convert q to a QI-Sequence once (line 2). Finally, it returns the result R (line 8).

4.1 A New Filtering Approach

We observe that the subgraph isomorphism testing cost

Algorithm 5: ***SelectSpanningEdge***$(P,\ q^w)$

Input : P: a set of edges;
q^w: a weight graph;
V_T: a set of vertices;
Output: e: an edge in P;

1 $P := \{e | e \in P \wedge (\forall e' \in P)\exists(w(e) \leq w(e'))\}$;
2 **if** $|P| > 1$ **then**
3 $P := \{e | e \in P \wedge \forall(e' \in P)\exists(|IndG(V_T \cup \{e.v\}, q^w)| \geq |IndG(V_T \cup \{e'.v\}, q^w)|)\}$;
4 **if** $|P| > 1$ **then**
5 $P := \{e | e \in P \wedge \forall(e' \in P)\exists(deg(e.v, q^w) \leq deg(e'.v, q^w))\}$;
6 Randomly select an edge $e \in P$;
7 **return** e;

Algorithm 6: ***QI-Framework***$(q,\ I,\ D)$

Input : q is a query graph;
I is the index;
D is a graph database;
Output: R: a set of graphs in which q is a subgraph;

1 $C :=$ Filtering(q, I);
2 Convert q to a QI-Sequence SEQ_q;
3 **for each** $g_i \in C$ **do**
4 $\mathscr{H} := \{0, ..., 0\}$;
5 $\mathscr{F} := \{0, ..., 0\}$;
6 **if** QuickSI(SEQ_q, g_i,$\mathscr{H}$,$\mathscr{F}$,1) **then**
7 $R = R \cup \{g_i\}$;
8 **return** R;

can be further reduced if two indexed features f_j and f_k in the database share a common subgraph. We explain our main idea below. Suppose that there are two indexed features, f_j and f_k in the database which share a common subgraph. Let f_i be a feature in a query graph q. We need to test whether $f_i \subseteq f_j$ and further test whether $f_i \subseteq f_k$, in the existing $filtering$-and-$verification$ framework. In our approach, instead, we pre-compute QI-Sequences for f_j and f_k, denoted as SEQ_{f_j} and SEQ_{f_k}, and maintain SEQ_{f_j} and SEQ_{f_k} in a prefix-tree index called Swift-Index. Given a query graph q, we do not decompose the query graph, q, into a set of features f_i. Instead, we search from the prefix-tree index in a top-down fashion, and test if a QI-Sequence, say SEQ_{f_j}, appear in Q using our QuickSI algorithm. The prefix-tree structure allows us to reduce the computational cost for subgraph isomorphism testing, because if a prefix of QI-Sequences does not appear in the query graph q, the whole QI-Sequences cannot appear in q.

Taking the advantage of the paradigm in QuickSI, we develop efficient filtering techniques to generate a candidate set. Our techniques are based on a new effective prefix-tree index called Swift-Index which indexes tree features that appear in the graph database D. Our QuickSI paradigm not only can be used to speed up the verification but also can be used to speed up the filtering computation.

Tree features in Swift-Index are organized by a *prefix* tree [3]. To construct such an index, we first convert each tree feature f to a QI-Sequence SEQ_f. Then we organize all QI-Sequences into a prefix tree. Note that in a QI-Sequence of a feature, there are no extra edge constraints since a feature

is a tree. In the prefix tree, each node represents an entry T_i of a SEQ_f for a tree feature f such that all entries in SEQ_f^i are recorded along the path from the root to the node. A dummy node is created to represent the root in the prefix tree. Consequently, each node of a prefix tree accumulatively carry a prefix of a QI-Sequence, SEQ_f^i.

5. PERFORMANCE EVALUATION

In this section, we report extensive empirical results to evaluate the effectiveness and efficiency of our new techniques. We compare our verification algorithm QuickSI described in Section 3 against the widely applied subgraph isomorphism testing algorithm Ullman [5]. To analyze the benefit of our verification algorithm and index technique on overall query processing performance, we implement two algorithms GSI and SSI. GSI combines gIndex [6] with our verification algorithm QuickSI by feeding the output of gIndex to QuickSI to produce final results. We show that our verification algorithm can bring immediate benefit to the performance of current *filtering*-and-*verification* based algorithms [2]. SSI is a combination of our new index technique Swift-Index proposed in Section 4 with QuickSI for verification. We compare FG-Index [2] and (Tree+Δ) [8] with GSI and SSI. All algorithms proposed in this paper are implemented in standard C++ and compiled with GNU GCC. Experiments are run on a PC with Intel Xeon 2.40GHz dual CPU and 4G memory running Debian Linux.

In our experiments, we use default parameters or suggested values unless specified otherwise. Particularly, default values $\sigma = 0.1$ and $\delta = 0.1$ are used in FG-Index [2] algorithm. In algorithm (Tree+Δ) , the support threshold is set to 0.01 and the maximal tree size is by default 10. For GSI algorithm, we adopt the default parameters in [6] with support threshold 0.1 and maximum fragment size $maxL$=10. The values of θ and γ are set to 0.1 and L is set to 10 in algorithm SSI.

Real dataset. We use the AIDS Antiviral Screen dataset, which consists of 43,905 classified chemical molecules. The dataset is publicly available on the website of Development Therapeutics Program.

5.1 Performance on Real Dataset

The AIDS Antiviral dataset is a popular benchmark in recent related works[6, 7, 8, 2]. There are totally 62 distinct vertex labels in the data set but the majority of the vertex labels are C, O and N. We derive different subsets from the full collection for comparison purpose. *Default real dataset* is a subset containing 10K graphs, which is firstly used in [6] and can be downloaded from `http://www.xifengyan.net/software.htm`. On average, each graph has 25.4 vertices and 27.3 edges. Other subsets with 1K, 5K, 20K and 40K graphs are derived in a similar way in order to study the scalability of the algorithms against different database size. We also create a *large real dataset* in order to evaluate the performance of our techniques on large graphs. This set consists of the largest 10K graphs taken from the original AIDS Antiviral collection. In the large real dataset, each graph has 40.4 vertices and 44 edges on average. We adopt the query set from [6] to test the effectiveness of our technique in terms of query response time. There are six query sets $Q4$, $Q8$, $Q12$, $Q16$, $Q20$ and $Q24$. Each query set Qi consists of 1000 query graphs with i edges. Default query set is $Q16$ in the following experiments.

[2]We do not use the index techniques in [7] and [8] as those indexes are closely interfered with the verification procedure.

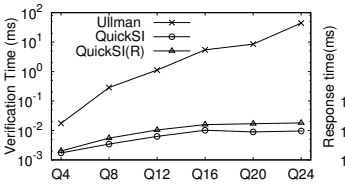


Figure 5: Verification Time

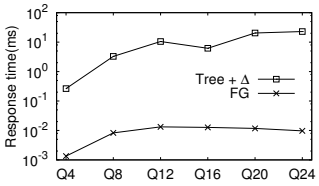


Figure 6: Response time

In the first experiment, we demonstrate the efficiency of our subgraph isomorphism testing algorithm QuickSI against Ullman algorithm. We first run filtering algorithm proposed in Section 4 against the default real dataset to create candidate sets for each query set. The candidate sets are then verified for subgraph isomorphism using QuickSI and Ullman respectively. We use QuickSI and QuickSI(R) to denote QuickSI algorithm with an effective QI-Sequence and a random QI-Sequence, respectively. Average verification time for each query set is recorded and demonstrated in Figure 5, which shows that both QuickSI and QuickSI(R) algorithm significantly outperform Ullman algorithm. Both QuickSI and QuickSI(R) achieve even more performance gain with increasing query graph size. For query set $Q24$, the average runtime of QuickSI is 5,535 times less than that of Ullman. Moreover, compared with Ullman algorithm, performance of both QuickSI and QuickSI(R) are less sensitive to query graph size. Meanwhile, QuickSI is twice as fast as QuickSI(R). It confirms our heuristic QI-Sequence construction algorithm plays an important role in reducing the verification cost.

Figure 6 illustrates the average query response time of two previous algorithms FG-Index and (Tree+Δ) against different query sets. It turns out that FG-Index is much more competitive than (Tree+Δ) in terms of query response time, which is our primary performance measure.[3] Thus we exclude (Tree+Δ) in the following experiments.

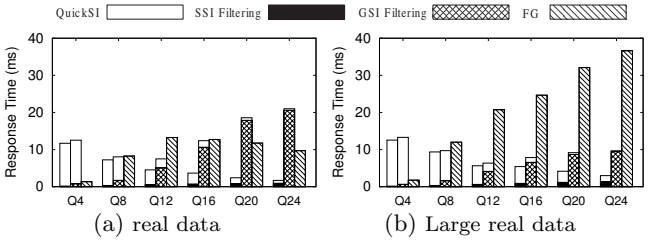


Figure 7: Response time

Figure 7 reports the average query response time per query comparing SSI, GSI and FG-Index algorithms against the default real dataset and the large real dataset. Filtering time and verification time (QuickSI) are recorded separately for SSI and GSI.

[3]The main goal of (Tree+Δ) is to reduce mining cost while achieving high efficiency in processing subgraph containment queries.

The experiments demonstrate that our new SSI Algorithm is the clear winner in the three algorithms on both datasets regarding query processing time.

FG-Index is attractive for very small queries. This is a reasonable result, as for small queries, large amount of graphs in the candidate set can be verified without subgraph isomorphism testing using FG-Index, whereas for larger queries, the *verification free* technique can not take effect on most candidates.

Comparing GSI and FG-Index, we can see that FG-Index beats GSI with a factor of up to 2 on median-sized data graphs, while on the large real dataset, GSI Algorithm outperforms FG-Index by a large margin. Remember that in [2], gIndex is outperformed by FG-Index Algorithm with at least one order, while with the help of our efficient verification algorithm, GSI Algorithm is comparable, in some cases much better than FG-Index. The difference shows that our verification algorithm can bring immediate improvement to the overall query performance of current *filtering*-and-*verification* based algorithms.

GSI Algorithm always spends less verification time compared with SSI Algorithms, since its graph-based index has better pruning ability, but the overall performance of GSI dramatically decreases when query graph size increases, because the filtering time grows and becomes the dominant cost.

We also record the index construction time, number of features in the index and the size of index for both default real dataset and large real dataset. Results are listed in Table 3 and Table 4. It is clear that the SSI technique has the smallest feature number, construction time and number of features. Note that both SSI and GSI indexes are counted in ASCII mode, while FG-Index is counted in binary mode.

	Construction Time (s)	#Features	Index Size
FG-Index	167.08 (166.5 + 0.58)	1641	$12.5M$
GSI	146.6	3276	$13M$
SSI	26.6	462	$5.5M$

Table 3: Statistic for Real data

	Construction Time (s)	#Features	Index Size
FG-Index	2133 (2102 + 31)	7100	$53.8M$
GSI	306.2	4394	$13M$
SSI	170.7	922	$11.8M$

Table 4: Statistic for Large Real data

In order to study the scalability of the algorithms against the graph database size, Figure 8 demonstrates the overall performance of three techniques on different subsets of the AIDS Antiviral collection. Because the binary code of gIndex from [6] fails to build index when the number of graphs reaches $20K$, there is no experimental result of GSI for the $20K$ and $40K$ datasets. In Figure 8(a), the query set with medium size $Q16$ is used as default query set to evaluate the response time. We can see the SSI wins on all four metrics in Figure 8, showing that the scalability of SSI also outperforms FG-Index.

6. CONCLUSION

In this paper, we study the problem of efficiently processing subgraph containment queries. An efficient subgraph-isomorphic verification algorithm, QuickSI, is proposed. In addition, combining QuickSI with a novel prefix-tree index, Swift-Index, our new techniques significantly improve the existing techniques for subgraph containment queries, in particular for graphs with median and/or large sizes. Our new techniques achieve high scalability regarding graph sizes and graph database sizes. Possible directions for future studies include an investigation of whether or not our current techniques can be effectively used to support the existing techniques for subgraph containment queries.

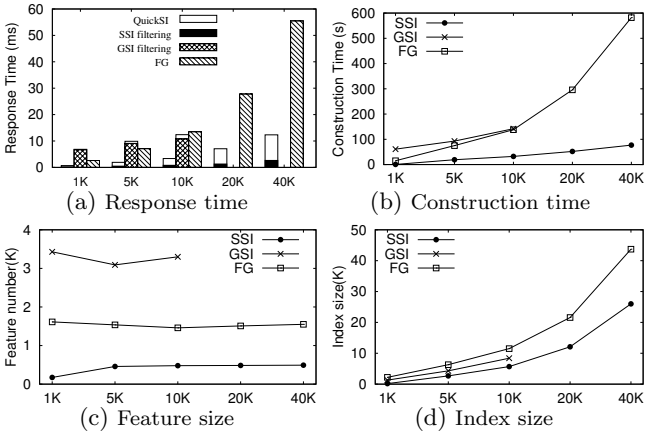


Figure 8: Scalability to #Graphs

7. REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. Ullman. *Data Structures and Algorithms*. 1983.

[2] J. Cheng, Y. Ke, W. Ng, and A. Lu. Fg-index: towards verification-free query processing on graph databases. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pages 857–872, 2007.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. 2001.

[4] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520.

[5] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.

[6] X. Yan, P. S. Yu, and J. Han. Graph indexing: a frequent structure-based approach. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pages 335–346, 2004.

[7] S. Zhang, M. Hu, and J. Yang. Treepi: A novel graph indexing method. In *Proceedings of the International Conference on Data Engineering*, pages 966–975, 2007.

[8] P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: tree + delta <= graph. In *Proceedings of the International Conference on Very Large Data Bases*, pages 938–949, 2007.