FG-Index: Towards Verification-Free Query Processing on Graph Databases

James Cheng, Yiping Ke, Wilfred Ng, and An Lu
Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
Hong Kong, China
{csjames,keyiping,wilfred,anlu}@cse.ust.hk

ABSTRACT

Graphs are prevalently used to model the relationships between objects in various domains. With the increasing usage of graph databases, it has become more and more demanding to efficiently process graph queries. Querying graph databases is costly since it involves *subgraph isomorphism* testing, which is an *NP-complete* problem. In recent years, some effective graph indexes have been proposed to first obtain a candidate answer set by filtering part of the false results and then perform verification on each candidate by checking subgraph isomorphism. Query performance is improved since the number of subgraph isomorphism tests is reduced. However, candidate verification is still inevitable, which can be expensive when the size of the candidate answer set is large.

In this paper, we propose a novel indexing technique that constructs a *nested inverted-index*, called *FG-index*, based on the set of *Frequent subGraphs* (*FGs*). Given a graph query that is an FG in the database, FG-index returns the exact set of query answers without performing candidate verification. When the query is an infrequent graph, FG-index produces a candidate answer set which is close to the exact answer set. Since an infrequent graph means the graph occurs in only a small number of graphs in the database, the number of subgraph isomorphism tests is small. To ensure that the index fits into the main memory, we propose a new notion of *δ-Tolerance Closed Frequent Graphs* (*δ-TCFGs*), which allows us to flexibly tune the size of the index in a parameterized way. Our extensive experiments verify that query processing using FG-index is orders of magnitude more efficient than using the state-of-the-art graph index.

Categories and Subject Descriptors: H.2.4 [Database Management]: Systems - Query processing

General Terms: Algorithms, Experimentation, Performance

Keywords: Graph Databases, Graph Indexing, Graph Querying, Frequent Subgraphs

1. INTRODUCTION

Graphs play an important role in representing and understanding objects and their relationships in various domains. In the scientific domain, graphs are used to model the molecular structures of chemical compounds, the food chains between various organisms within a specific ecotope, the social networks between human beings, and so on. Particularly in the field of computer science, graphs are popularly used, such as the E-R diagrams in database design, the automaton in the theory of computing, the graphical models in artificial intelligence, the UML diagrams in software engineering, and so on. In addition, rapidly increasing Web sites and XML documents can also be modelled as graphs. Therefore, it is evident that graph databases will become more and more prevalently used in the near future. As a result, efficient query processing on graph databases has become increasingly demanding.

Let $\mathcal{D} = \{g_1, g_2, \ldots, g_N\}$ be a graph database that contains N graphs. A typical *graph query* can be described as follows: *given a graph q, retrieve all $g_i \in \mathcal{D}$ such that g_i is a supergraph of q.* Due to the diversity of graphs, graph queries are in general complex, since any part (subgraph) of a query is a predicate that needs to be satisfied in the query evaluation. For example, many XML path queries with complex predicates can be expressed as graphs. Processing the graph query by a sequential scan on $\mathcal{D}$ to check whether q is a subgraph of each g_i is infeasible, since *subgraph isomorphism* testing is known as an *NP-complete* problem [7].

In recent years, some effective indexes on graph databases [17, 24] are proposed. Query processing using these indexes is performed in two fundamental steps: *filtering* and *candidate verification.* First, the filtering step uses the index to eliminate part of the false results and produces a candidate answer set. Then, the candidate verification step verifies whether the query is indeed a subgraph of each candidate. Since the candidate answer set is in general much smaller than the entire graph database, query processing using the indexes is significantly more efficient than the sequential scan approach. However, due to the high complexity of subgraph isomorphism testing (either by a conventional approach [19, 24] or by checking path embeddings [17]), candidate verification is still very expensive since the size of the candidate answer set is at least that of the exact answer set.

"Can we avoid the expensive candidate verification in querying graph databases?" In this paper, we provide an encouraging answer to this question. We propose a novel indexing technique that constructs a *nested inverted-index*, called

FG-index, based on *Frequent subGraphs* (*FGs*) [13, 15, 22] proposed by the data mining community. An FG is a graph that is a subgraph of at least $(\sigma \cdot |\mathcal{D}|)$ number of graphs in $\mathcal{D}$, where σ $(0 \leq \sigma \leq 1)$ is a user-defined threshold.

Query processing using FG-index has the following benefits. If a query is an FG in the database, FG-index returns the exact set of query answers without performing any candidate verification. Otherwise, that is, q is an infrequent subgraph, FG-index produces a candidate answer set which is close to the exact answer set. Intuitively, an infrequent subgraph means it is the subgraph of only a small number of graphs in the database; thus, the number of subgraph isomorphism tests is small.

Our work achieves a tremendous improvement over the existing work [17, 24]. First, for the existing work, processing a query with a large answer set implies performing candidate verification on an even larger candidate answer set. However, having a large answer set actually implies that the query has a high probability of being an FG, which can be very efficiently processed using the FG-index without any candidate verification. Second, in the case when the query is not an FG, the candidate answer set obtained from the FG-index is as small as the exact answer set and hence the number of candidate verifications is minimal.

Apart from candidate verification, the size of the index is also important for efficient query processing. Thus, a challenge in our approach is that the set of FGs can be large for a small σ and hence the index built on the FGs can be too large to fit into the main memory. In this case, the query performance will be degraded, since the processing needs to access the disk frequently. Inspired by the work on compressing the set of frequent patterns [21, 4], we propose a new notion of *δ-Tolerance Closed Frequent subGraphs* (*δ-TCFGs*) for compressing the set of FGs. The notion of δ-TCFGs allows us to flexibly tune the size of FG-index in a parameterized way. Each δ-TCFG can be regarded as a representative supergraph of a cluster of FGs. In this way, FG-index builds an outer inverted-index on the set of δ-TCFGs, which is resident in the main memory. Then, an inner inverted-index is built on the cluster of FGs of each δ-TCFG, which is resident in the disk.

A comprehensive set of experiments verifies that FG-index is an effective index for processing queries that are both FGs and non-FGs. FG-index significantly outperforms the state-of-the-art graph index, gIndex [24], on both index construction and query processing. In particular, the query processing using FG-index is orders of magnitude faster than that using gIndex for a wide range of queries. The results also verify that the concept of δ-TCFGs is effective in controlling the size of the memory-resident portion of the index. In addition, FG-index is also shown to have a good scalability on the database size, the graph size and the graph density.

Organization. This paper is organized as follows. Section 2 defines the preliminary concepts. Section 3 gives the background on mining FGs and defines the notion of δ-TCFGs. Section 4 presents the framework of FG-index and Section 5 discusses in detail the construction of FG-index and query processing. Section 6 reports the experimental results. Finally, Section 7 discusses the related work and Section 8 concludes the paper.

2. PRELIMINARIES

In this paper, we restrict our discussion on *undirected, labelled connected graphs*, while our method can be easily extended to process directed and unlabelled graphs. Hereafter, we simply call an undirected labelled connected graph a graph.

A graph g is defined as a 4-tuple (V, E, L, l), where V is the set of vertices, E is the set of edges, L is the set of labels and l is a labelling function that maps each vertex or edge to a label in L. We define the *size* of a graph g as $size(g) = |E(g)|$.

Given a set of graphs G, a *distinct edge* in G is defined as a 3-tuple, (l_u, l_e, l_v), where l_e is the label of an edge (u, v) in a graph $g \in G$, and l_u and l_v are the labels of u and v in g. Given a distinct edge e and a graph g in G, we define the *count* of e in g, denoted as $count(e, g)$, as the number of occurrences of e in g.

We define subgraph isomorphism as follows.

Definition 1. (Subgraph Isomorphism) *Given two graphs, $g = (V, E, L, l)$ and $g' = (V', E', L', l')$, a* subgraph isomorphism *from g to g' is an injective function f: $V \rightarrow V'$, such that $\forall(u, v) \in E$, $(f(u), f(v)) \in E'$, $l(u) = l'(f(u))$, $l(v) = l'(f(v))$, and $l(u, v) = l'(f(u), f(v))$.*

A graph g is called a *subgraph* of another graph g' (or g' is a *supergraph* of g), denoted as $g \subseteq g'$ (or $g' \supseteq g$), if there exists a subgraph isomorphism from g to g'. The graph g is called a *proper subgraph* of g', denoted as $g \subset g'$, if $g \subseteq g'$ and $g \not\supseteq g'$.

Let $\mathcal{D} = \{g_1, g_2, \ldots, g_N\}$ be a *graph database* that contains a set of N graphs. A *query graph* is a graph that has at least one edge. Processing a query graph with a single vertex is trivial and thus not discussed. Given a query graph q, the *graph query processing* studied in this paper is *to find the set of all graphs in $\mathcal{D}$ that are supergraphs of q*. We denote the *answer set* of q as $\mathcal{D}_q = \{g: g \in \mathcal{D}, q \subseteq g\}$. In the subsequent discussions, a query graph is simply called a *query*.

3. δ-TOLERANCE CLOSED FREQUENT SUBGRAPHS

In this section, we first introduce several concepts related to frequent subgraph mining and explain how the data mining technique can be applied in graph indexing. Then, we define δ-tolerance closed frequent subgraphs.

3.1 Frequent Subgraph Mining

Given a graph database $\mathcal{D}$ and a graph g, the *frequency* of g in $\mathcal{D}$, denoted as $freq(g)$, is defined as $|\{g' : g' \in \mathcal{D}, g' \supseteq g\}|$. A graph g is called a *Frequent subGraph* (*FG*) [13, 15, 22] if $freq(g) \geq (\sigma \cdot |\mathcal{D}|)$, where σ $(0 \leq \sigma \leq 1)$ is a user-specified *minimum frequency threshold*.

Let $\mathcal{F}$ be the set of all FGs that are mined from $\mathcal{D}$. A graph g is called a *Maximal Frequent subGraph* (*MFG*) [12] if $g \in \mathcal{F}$ and $\nexists g' \in \mathcal{F}$ such that $g' \supset g$. A graph g is called a *Closed Frequent subGraph* (*CFG*) [23] if $g \in \mathcal{F}$ and $\nexists g' \in \mathcal{F}$ such that $g' \supset g$ and $freq(g') = freq(g)$.

Example *1.* Figure 1 shows 13 FGs, $f_1, \cdots, f_{13}$, mined from a graph database, where a, b, c represent three distinct edges. Figure 2 organizes the FGs according to their size and represents each FG as a node, where the number following ":" is the frequency of the FG. (These two figures are used throughout the paper. Thus, the number on each edge in Figure 2 will be introduced in other examples.)

Among the FGs, f_8, f_9 and f_{13} are MFGs since they have no proper supergraphs. All the FGs, except f_{12}, are CFGs. The FG f_{12} is not a CFG since $f_{13} \supset f_{12}$ and $\mathit{freq}(f_{13}) = \mathit{freq}(f_{12})$. ■

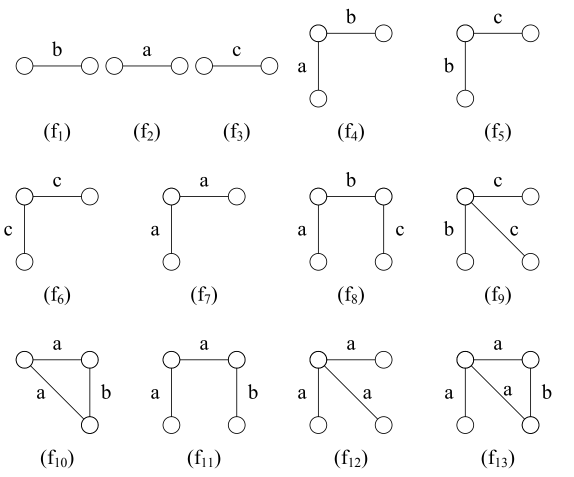


Figure 1: Frequent Subgraphs

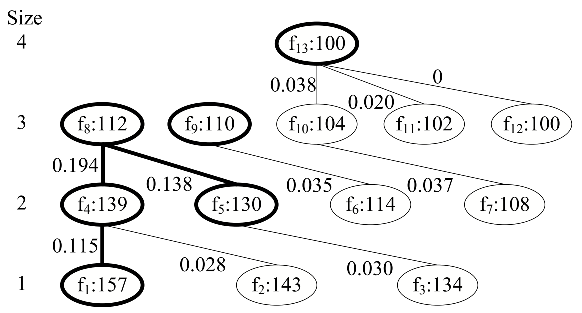


Figure 2: Frequent Subgraphs and Their Frequency

3.2 An Analysis on FG-Based Graph Indexing

Assume that $\mathcal{D}_g$ is available for every FG $g \in \mathcal{F}$. Given a query q, if $q \in \mathcal{F}$, we can search $\mathcal{F}$ to obtain the query answer set $\mathcal{D}_q$ without candidate verification. However, it is non-trivial to index all FGs since the number of FGs can be very large when σ is small.

Yan et al. [24] propose an index, called *gIndex*, on a set of *discriminative* FGs, denoted as $\mathcal{F}_d$. $\mathcal{F}_d$ is a subset of $\mathcal{F}$ and is substantially smaller than $\mathcal{F}$. A query q is processed by first obtaining a candidate answer set $C_q = (\bigcap_{f \in \mathcal{F}_d \wedge f \subseteq q} \mathcal{D}_f)$. $\mathcal{D}_q$ is then obtained by verifying whether each $g \in C_q$ is a supergraph of q. However, this means that at least $|\mathcal{D}_q|$ subgraph isomorphism tests are needed in the verification process.

The cost of query processing using gIndex can be described by the cost model defined in Equation (1), where $T_{response}$ is the response time of processing a query q, T_{search} is the time taken to compute C_q using the index, $T_{I/O}$ is the I/O time taken for fetching a candidate graph from the disk, and T_{isSubG} is the time taken for testing subgraph isomorphism between q and each candidate graph.

$$T_{response} = (T_{search} + |C_q| \times T_{I/O} + |C_q| \times T_{isSubG}) \ . \quad (1)$$

Since subgraph isomorphism testing is an expensive operation, $(|C_q| \times T_{isSubG})$ occupies a large portion of $T_{response}$ of gIndex. In fact, a moderate sized C_q can result in a long query response time, as evidenced by our experimental results.

In this paper, we investigate the possibility of designing an index on $\mathcal{F}$ so that a query $q \in \mathcal{F}$ can be answered without candidate verification. This is a tremendous reduction on the query response time since if $q \in \mathcal{F}$, then $|\mathcal{D}_q|$ is large and hence $|C_q|$ is also large. The challenge is, however, how to index $\mathcal{F}$ when $\mathcal{F}$ is large.

To address this problem, we propose a new notion of δ-*Tolerance CFGs* (δ-*TCFGs*). The set of δ-TCFGs, denoted as $\mathcal{T}$, is a concise representation of $\mathcal{F}$. Based on $\mathcal{T}$, $\mathcal{F}$ can be classified into $|\mathcal{T}|$ disjoint partitions. Each $g \in \mathcal{T}$ is associated with a partition, which contains a set of FGs that are subgraphs of g. The size of $\mathcal{T}$ is substantially smaller than that of $\mathcal{F}$. Thus, we can build an index on $\mathcal{T}$, which is to be resident in the main memory. Then, we build a nested index on the partition of FGs associated with each $g \in \mathcal{T}$. These nested indexes consist of the majority of FGs and are resident in the disk.

Query processing using our index is done in a reverse direction as gIndex; that is, given a query q, we search in the index for q's supergraph, whose partition contains q. Thus, the search space narrows down quickly as each partition is local.

In order to answer queries that are not in $\mathcal{F}$, our index also incorporates a set of infrequent distinct edges. Together with the index built on $\mathcal{T}$, we are able to obtain C_q as small as $\mathcal{D}_q$, for a query $q \notin \mathcal{F}$.

In the rest of this section, we define the notion of δ-TCFGs, which is used to build our index in Section 5.

3.3 The Notion of δ-TCFGs

We define the notion of δ-TCFGs as follows.

Definition 2. (δ-Tolerance Closed Frequent subGraph) *A graph g is a δ-*Tolerance Closed Frequent subGraph *(δ-*TCFG*) if and only if $g \in \mathcal{F}$ and $\nexists g' \in \mathcal{F}$ such that $g' \supset g$ and $\mathit{freq}(g') \geq ((1-\delta) \cdot \mathit{freq}(g))$, where δ $(0 \leq \delta \leq 1)$ is a user-specified* frequency tolerance factor.

We can define CFGs and MFGs by δ-TCFGs as follows.

Lemma 1. *A graph g is a CFG if and only if g is a 0-TCFG.*

Proof. *IF:* Let g be a 0-TCFG. By Definition 2, $g \in \mathcal{F}$ and $\nexists g' \in \mathcal{F}$ such that $g' \supset g$ and $\mathit{freq}(g') \geq ((1-0) \cdot \mathit{freq}(g))$. Thus, we have $\mathit{freq}(g') \geq \mathit{freq}(g)$. But $g' \supset g$ implies $\mathit{freq}(g') \leq \mathit{freq}(g)$. It follows that $\mathit{freq}(g') = \mathit{freq}(g)$. We thus conclude that g is a CFG, since $\nexists g' \in \mathcal{F}$ such that $g' \supset g$ and $\mathit{freq}(g') = \mathit{freq}(g)$.

ONLY IF: Let g be a CFG. It follows that $g \in \mathcal{F}$ and $\nexists g' \in \mathcal{F}$ such that $g' \supset g$ and $\mathit{freq}(g') = \mathit{freq}(g)$. By Definition 2, g is also a 0-TCFG. □

LEMMA 2. *A graph g is an MFG if and only if g is a 1-TCFG.*

PROOF. *IF:* Let g be a 1-TCFG. By Definition 2, $g \in \mathcal{F}$ and $\nexists g' \in \mathcal{F}$ such that $g' \supset g$ and $freq(g') \geq ((1-1) \cdot freq(g)) = 0$, which simply means that $g \in \mathcal{F}$ and $\nexists g' \in \mathcal{F}$ such that $g' \supset g$. Thus, g is an MFG.

ONLY IF: If g is an MFG, it is trivial that g is also a 1-TCFG according to Definition 2. □

COROLLARY 1. *Let $\mathcal{T}_\delta$ be the set of δ-TCFGs, $\mathcal{C}$ be the set of CFGs, and $\mathcal{M}$ be the set of MFGs. Then, $\mathcal{M} \subseteq \mathcal{T}_\delta \subseteq \mathcal{C}$.*

PROOF. It follows directly from Lemmas 1 and 2. □

Corollary 1 gives the upper bound and the lower bound on the size of $\mathcal{T}_\delta$. The following example illustrates the concept of δ-TCFGs.

EXAMPLE *2.* Consider the 13 FGs in Figure 2. The number on each edge in Figure 2 is computed as $d_e = (1 - freq(f_i)/freq(f_j))$, where f_i is the smallest proper supergraph of f_j that has the greatest frequency. If $d_e \leq \delta$, according to Definition 2, f_j is not a δ-TCFG; otherwise, f_j is a δ-TCFG. Let $\delta = 0.04$, then the set of 0.04-TCFGs is $\{f_1, f_4, f_5, f_8, f_9, f_{13}\}$, i.e., the set of bold nodes in Figure 2. For example, f_1 is a 0.04-TCFG since f_1 does not have a proper supergraph that has frequency greater than $((1-0.04) \times 157) \approx 150$. The FG f_6 is not a 0.04-TCFG since we have $freq(f_9) > ((1-0.04) \times freq(f_6))$.

The set of 0.15-TCFGs is $\{f_4, f_8, f_9, f_{13}\}$. The set of 1-TCFGs, i.e., the set of MFGs, is $\{f_8, f_9, f_{13}\}$; while the set of 0-TCFGs (i.e., CFGs) contains all FGs except f_{12}. ■

For simplicity, in the rest of the paper, we use the lighter notation $\mathcal{T}$ to represent $\mathcal{T}_\delta$ when δ is clear in the context.

Since a query q can be in $(\mathcal{F} - \mathcal{T})$, in the following we define the connection between the graphs in $\mathcal{T}$ and those in $(\mathcal{F} - \mathcal{T})$.

Let G be a set of graphs and $\mathbb{N} = \{1, 2, \dots\}$ be the set of natural numbers. Let $h : G \rightarrow \mathbb{N}$ be an injective function that assigns a unique ID $n \in \mathbb{N}$ to each graph $g \in G$. We define a total order on G as follows.

DEFINITION 3. (GRAPH SET ORDER) *Given a set of graphs G, a* graph set order *$\preceq$ on G is a total order defined as follows. Let $g_1, g_2 \in G$, $g_1 \preceq g_2$ if one of the following three statements is true.*

1. *$size(g_1) < size(g_2)$.*
2. *$size(g_1) = size(g_2)$ and $freq(g_1) > freq(g_2)$.*
3. *$size(g_1) = size(g_2)$, $freq(g_1) = freq(g_2)$, and $h(g_1) \leq h(g_2)$.*

We further define $g_1 \prec g_2$ if $g_1 \preceq g_2$ and $g_1 \neq g_2$.

DEFINITION 4. (CLOSEST δ-TCFG SUPERGRAPH) *Given $g_t \in \mathcal{T}$ and $g \in (\mathcal{F} - \mathcal{T})$, g_t is called the* closest δ-TCFG supergraph *of g if $g_t \supset g$ and $\nexists g'_t \in \mathcal{T}$ such that $g'_t \supset g$ and $g'_t \prec g_t$.*

DEFINITION 5. (CLOSURE OF A δ-TCFG) *Given $g_t \in \mathcal{T}$, the* closure *of g_t, denoted as $CLOS(g_t)$, is defined as $CLOS(g_t) = \{g : g_t$ is the closest δ-TCFG supergraph of $g\}$.*

LEMMA 3. *For each $g \in (\mathcal{F} - \mathcal{T})$, the closest δ-TCFG supergraph of g is unique.*

PROOF. It follows directly from Definitions 3 and 4. □

Lemma 3 ensures that, based on the graph set order defined by $\preceq$, a query $q \in (\mathcal{F} - \mathcal{T})$ must have a unique closest δ-TCFG supergraph, g, and q can be located in the closure of g. We illustrate the concept of closure by the following example.

EXAMPLE *3.* Referring to Figures 1 and 2, the set of FGs is ordered according to the graph set order $\preceq$, where the ID of each f_i is assigned as i. We have $f_1 \prec f_4$ since $size(f_1) < size(f_4)$; while for f_1 and f_2 which are of the same size, $f_1 \prec f_2$ since $freq(f_1) > freq(f_2)$.

When $\delta = 0.04$, f_{13} is the closest δ-TCFG supergraph of f_7, f_{10}, f_{11} and f_{12}; in other words, $CLOS(f_{13}) = \{f_7, f_{10}, f_{11}, f_{12}\}$. Similarly, $CLOS(f_4) = \{f_2\}$, $CLOS(f_5) = \{f_3\}$, and $CLOS(f_9) = \{f_6\}$. ■

4. FRAMEWORK OF GRAPH QUERY PROCESSING USING FG-INDEX

We give the framework of the FG-index-based graph query processing as follows.

1. *Index Construction.*

   FG-index consists of the following two parts: the *core FG-index* and *Edge-index*.

   First, we construct the *memory-resident* inverted-index on the set $\mathcal{T}$. Then, a *disk-resident* inverted-index is built on the FGs in the closure of each δ-TCFG. In the same way, if the closure is too large, a local set of δ-TCFGs can be computed from the set of FGs in the closure and a further nested inverted-index can be constructed. The *core FG-index* consists of the memory-resident and all the disk-resident inverted-indexes.

   To ensure that any query can be answered, we include in FG-index another index, called *Edge-index*, which is built on the set of infrequent distinct edges[1] in $\mathcal{D}$. For each infrequent distinct edge e in Edge-index, we associate $\mathcal{D}_e$ with e.

2. *Query Processing.*

   Given a query q, we first search it in the core FG-index. If q is a δ-TCFG, we can directly retrieve q and $\mathcal{D}_q$ from the memory-resident core FG-index. Otherwise, we first find q's closest δ-TCFG supergraph, g. Then, g's disk-resident index is loaded to locate q and retrieve $\mathcal{D}_q$.

   If q cannot be found in the core FG-index, then $q \notin \mathcal{F}$. In this case, we use the core FG-index to find a set of q's subgraphs, S_{core}, and then retrieve $\mathcal{D}_g$ for each $g \in S_{core}$. If q consists of a set of infrequent distinct edges, S_{edge}, then for each $e \in S_{edge}$, $\mathcal{D}_e$ is also retrieved from Edge-index. Then, we compute C_q as the intersection of all $\mathcal{D}_g$ for $g \in S_{core}$ and all $\mathcal{D}_e$ for $e \in S_{edge}$. Finally, we obtain $\mathcal{D}_q$ by verifying whether each $g \in C_q$ is a supergraph of q.

[1]A distinct edge can be regarded as a graph with only one edge. Thus, an infrequent distinct edge is simply an infrequent subgraph.

In the next section, we discuss the construction and query processing of FG-index. Referring to Equation (1), our subsequent discussions reveal the following benefits of using FG-index.

1. The size of FG-index can be tuned in a parameterized way to reduce the search space and hence T_{search}.
2. If $q \in \mathcal{F}$, then T_{isSubG} can be completely eliminated and $T_{I/O}$ is minimized since $C_q = \mathcal{D}_q$.
3. If $q \notin \mathcal{F}$, then $|C_q| \approx |\mathcal{D}_q|$. Note that $|\mathcal{D}_q| < (\sigma \cdot |\mathcal{D}|)$. When σ is small, $|\mathcal{D}_q|$ is small and hence $|C_q|$ is also small. In other words, $T_{response}$ can be controlled by σ.

5. FG-INDEX

In this section, we discuss in detail the development of FG-index including the index construction, the query processing, the maintenance of the index, as well as the limitation and opportunity of FG-index.

5.1 Structure of FG-Index

We first define the structure of the inverted-index in FG-index.

Definition 6. (Inverted-Graph-Index) *Given a set of graphs G, an* Inverted-Graph-Index (IGI) *constructed on G consists of the following components:*

- *An array, called the* Graph Array (GA)*, stores G. Let* GA[i] *be the i-th entry in the GA. The graph stored in GA[i] is assigned an* ID i. *If G is a set of δ-TCFGs, then each GA[i] also keeps the location of the* nested IGI *that is built on $CLOS(g)$, where g is the δ-TCFG stored in GA[i].*
- *An array, called the* Edge Array (EA)*, stores the set of distinct edges in G.*
- *Each distinct edge e in the EA is associated with a list of* ID-entries. *Each ID-entry consists of a list of arrays called* ID-arrays.

  Each ID-array consists of a set of IDs, which are the IDs of a set of graphs, $G' \subseteq G$, such that $\forall g_1, g_2 \in G'$, $size(g_1) = size(g_2)$ and $count(e, g_1) = count(e, g_2)$.

  Let $n = size(g)$ and $m = count(e, g)$ for a graph $g \in G$. The ID-array that consists of the ID of g is called an m-edge ID-array*, and the corresponding ID-entry that consists of the ID-array is called a* size-n ID-entry. *For simplicity, we denote $IDA(m, n, e)$ as* the m-edge ID-array in the size-n ID-entry of e.

The first level of the core FG-index is an IGI constructed on $\mathcal{T}$. For each $g \in \mathcal{T}$, if $CLOS(g) \neq \emptyset$, then a *nested IGI* is constructed on $CLOS(g)$. If the size of $CLOS(g)$ is still large, a *local* set of δ-TCFGs, $\mathcal{T}_{local}$, can be computed from $CLOS(g)$. Then, for each $g' \in \mathcal{T}_{local}$, a *nested IGI* is constructed on $CLOS(g')$. An example of an IGI is shown as follows.

Example 4. Referring to the FGs in Figures 1 and 2, let $\delta = 0.04$, then $\mathcal{T} = \{f_1, f_4, f_5, f_8, f_9, f_{13}\}$. Figure 3 shows the corresponding IGI constructed on $\mathcal{T}$. For example, the size-3 ID-entry of the distinct edge c has two ID-arrays: the 1-edge ID-array, denoted as $IDA(1, 3, c)$, containing one ID 4, and the 2-edge ID-array, denoted as $IDA(2, 3, c)$, containing one ID 5. The two IDs correspond to f_8 and f_9 in GA[4] and GA[5], respectively.

As shown in Figure 1, f_9 is of size 3, $count(b, f_9) = 1$ and $count(c, f_9) = 2$. In the IGI shown in Figure 3, f_9 is stored in GA[5]. Thus, we have ID 5 in $IDA(1, 3, b)$ and $IDA(2, 3, c)$. ■

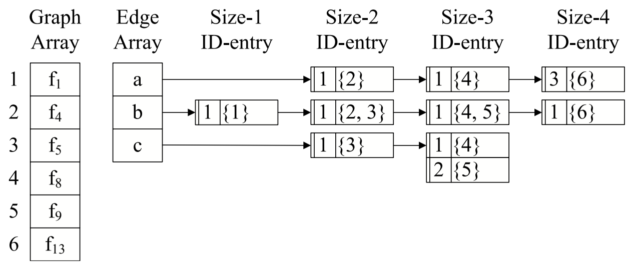


Figure 3: Inverted-Graph-Index of Example 4

5.2 Index Construction

The algorithm for our index construction, *BuildIndex*, is shown in Algorithm 1, which is divided into three parts: the computation of $\mathcal{T}$ (Lines 1-9), the construction of the core FG-index (Lines 10-18), and the creation of Edge-index (Lines 19-20).

Algorithm 1 BuildIndex

Input: Graph Database $\mathcal{D}$, the set of FGs $\mathcal{F}$, and the frequency tolerance factor δ.
Output: The core FG-index and Edge-index.

```
1.  Sort $\mathcal{F}$ s.t. $\forall g_1, g_2 \in \mathcal{F}$, $g_1$ is ordered before $g_2$ if $g_1 \prec g_2$;
2.  Let $\mathcal{T} = \mathcal{F}$ and $\mathcal{T}_i$ be the set of FGs that consist of $i$ edges;
3.  for each $i = 1, 2, \ldots$ do
4.     for each $g \in \mathcal{T}_i$ do
5.        for each $g' \in \mathcal{T}_{i+1}$ do
6.           if ($g \subset g'$)
7.              if ($freq(g') \geq (1 - \delta) \cdot freq(g)$)
8.                 $\mathcal{T} \leftarrow \mathcal{T} - \{g\}$;
9.              break; /* go to Line 10 */
10.       if ($g \in \mathcal{T}$)
11.          Store $g$ in the first free entry in the GA;
12.          for each distinct edge $e$ in $g$ do
13.             if ($e \notin$ EA)
14.                Add $e$ to the EA;
15.             Add the ID of $g$ to $IDA(count(e, g), size(g), e)$;
16. for each $g \in (\mathcal{F} - \mathcal{T})$ do
17.    Find $g$'s closest $\delta$-TCFG supergraph, $g'$;
18.    Add $g$ to the nested IGI of $g'$;
19. for each infrequent distinct edge $e$ in $\mathcal{D}$ do
20.    Add $e$ and $\mathcal{D}_e$ to Edge-index;
```

BuildIndex first sorts $\mathcal{F}$ (note that the sorting does not involve any expensive graph operation). Based on the order defined by $\prec$, the first $g' \in \mathcal{F}$, where $g' \supset g$, has the greatest frequency among all other $g'' \supset g$. Thus, if $freq(g') < ((1 - \delta) \cdot freq(g))$, then $\forall g'' \supset g$, $freq(g'') \leq freq(g') < ((1 - \delta) \cdot freq(g))$. This implies that, in order to check whether g is a

δ-TCFG, we only need to find the first supergraph of g that has one more edge than g (Lines 5-9). If the first supergraph of g, g', is found (Line 6), and $freq(g') \geq ((1-\delta) \cdot freq(g))$ (Line 7), then g is not a δ-TCFG by Definition 2. Thus, g is removed from $\mathcal{T}$ (Line 8). Otherwise, if $freq(g') < ((1-\delta) \cdot freq(g))$ or g has no supergraph, then g is a δ-TCFG. In implementation we initially assume all $g \in \mathcal{F}$ are δ-TCFGs by setting a flag for each g, and then unset the flag when g is found not to be in $\mathcal{T}$.

Lines 10-15 of Algorithm 1 construct an IGI on $\mathcal{T}$. For each $g \in \mathcal{T}$, the ID of g is assigned as the position in the GA where g is stored. For each distinct edge e in g, if e is not in the EA, we add e to the EA; then, the ID of g is added to the end of the array $IDA(count(e, g), size(g), e)$. Thus, the IDs in each ID-array are automatically sorted. We use a hashtable to access each distinct edge in the EA.

After the first level of the core FG-index is constructed, BuildIndex builds a nested IGI on the closure of each δ-TCFG (Lines 16-18). Each nested IGI is constructed in a similar way as described in Lines 11-15. In Line 17, BuildIndex finds g's closest δ-TCFG supergraph, g'. Since now we have already constructed the IGI on $\mathcal{T}$, we can use the IGI to efficiently find g', which will be discussed in Algorithm 2 when we process a query using the IGI.

The core FG-index only covers the queries that are FGs. To ensure that all queries can be answered using FG-index, we also create an Edge-index as part of the FG-index to include the set of infrequent distinct edges in $\mathcal{D}$ (Lines 19-20). These edges are also accessed via the same hashtable used for the EA, where a flag is used to indicate whether an edge is frequent or not.

Efficiency of Index Construction. The construction of an IGI for the core FG-index is straightforward and involves no expensive operation (Lines 10-15 in Algorithm 1). Edge-index can be generated by scanning the database once (it can actually be obtained almost free from the FG mining process). The computation of $\mathcal{T}$ (Lines 1-9 in Algorithm 1) is more expensive since it needs to find the first supergraph of a graph g in $\mathcal{F}$. Fortunately, the order imposed by $\prec$ allows us to narrow down the search to only those graphs with one more edge than g. Moreover, before performing the subgraph isomorphism test between g and its possible supergraph g', some effective checking can be first performed on g and g', such as the checking on their frequency, the number of their distinct edges, the number of vertices and the degree of the vertices.

5.3 Query Processing

The processing of a query q using FG-index can be classified into two cases: $q \in \mathcal{F}$ and $q \notin \mathcal{F}$.

5.3.1 Query is an FG

When $q \in \mathcal{F}$, Algorithm 2 outlines how $\mathcal{D}_q$ is obtained from the core FG-index. Let E be the set of distinct edges in q. The algorithm, *FG-Query*, processes only those graphs that contain all edges in E. It starts with the graphs that have the same size as q (Line 2) until a supergraph of q is found (Lines 7-11).

Let i denote the size of the graphs that FG-Query is currently processing. For each $e \in E$, FG-Query first obtains $C(e)$, which is the set of IDs of graphs that are of size i and have at least $count(e, q)$ occurrences of e, as shown in Line 4. The IDs in each $C(e)$ are then sorted in ascending order (Line 5). Then in Line 6, $C(e)$ are intersected for all e in order to find a supergraph of q. According to the order in which each graph is added to the GA (i.e., the way that the ID of each indexed graph is assigned), the first supergraph of q, whose ID is obtained by the intersection, must be either q or the closest δ-TCFG supergraph of q. Thus, we either output $\mathcal{D}_q$ (the graphs are retrieved from the disk), or continue to find q by recursively invoking FG-Query to process on the nested IGI of q's closest δ-TCFG supergraph (Lines 9 and 11). If the intersection of $C(e)$ does not obtain any ID or no supergraph of q is found for the current i, FG-Query increments i by 1 (Line 2) and continues a new round of iteration to search a supergraph of q.

Algorithm 2 FG-Query

Input: The IGI in the core FG-index and a query q.
Output: $\mathcal{D}_q$.

```
1.  Let $E$ be the set of distinct edges in $q$;
2.  for each $i = size(q), size(q)+1, \ldots$ do
3.      for each $e \in E$ do
4.          $C(e) \leftarrow (\bigcup_{j \geq count(e,q)} IDA(j, i, e))$;
5.          Sort $C(e)$ in ascending order;
6.      Intersect $C(e)$, $\forall e \in E$, until an $ID$ is obtained;
7.      if ($g$ in GA[$ID$] is a supergraph of $q$)
8.          if ($g = q$)
9.              Return $\mathcal{D}_g$;
10.         else
11.             Return FG-Query with $g$'s nested IGI and $q$ as input;
12.     else
13.         Go to Line 6 and continue the intersection;
```

EXAMPLE 5. Referring to the IGI in Example 4, let $q = f_{11}$. We demonstrate how $\mathcal{D}_q$ is obtained by FG-Query. Since $size(f_{11}) = 3$, we start search IGI from Size-3 ID-arrays, that is, $i = 3$. Since $count(a, f_{11}) = 2$, we have $C(a) = \bigcup_{j \geq 2} IDA(j, 3, a) = \emptyset$. Similarly, since $count(b, f_{11}) = 1$, $C(b) = \bigcup_{j \geq 1} IDA(j, 3, b) = IDA(1, 3, b) = \{4, 5\}$. The intersection terminates immediately since $C(a) = \emptyset$. Therefore, FG-Query proceeds to $i = 4$ in Line 2. Now $C(a) = IDA(3, 4, a) = \{6\}$ and $C(b) = IDA(1, 4, b) = \{6\}$. Therefore, intersecting $C(a)$ and $C(b)$ obtains an ID "6". Since GA[6] $= f_{13}$ is a supergraph of $q = f_{11}$, i.e., f_{13} is the closest δ-TCFG supergraph of f_{11}, Line 11 invokes FG-Query to process on the nested IGI of f_{13}. The recursive call of FG-Query finally returns $\mathcal{D}_{f_{11}}$ (details omitted). ■

Response Time Analysis for FG-Query. The efficiency of the intersection of $C(e)$ for all $e \in E$ depends on the size of $C(e)$. The IDs in each $C(e)$ belong to a local set of δ-TCFGs that are of a specific size and contain at least $count(e, q)$ occurrences of e. Thus, the size of $C(e)$ is small, because the size of the whole set of δ-TCFGs is small as controlled by δ. In most cases (in almost all cases in our experiments), the first ID returned by the intersection is the ID of q or the ID of q's closest δ-TCFG supergraph and hence the intersection terminates early. In fact, the time taken for the intersection is negligible compared to the I/O time if the nested IGI is resident in the disk. However, the nested IGI is also small and we only need to load the portion of the IGI related to distinct edges in q into the main memory.

Therefore, T_{search} for processing $q \in \mathcal{F}$ is in general much smaller than $T_{I/O}$ used in retrieving $\mathcal{D}_q$ from the disk, which is inevitable unless the main memory is large enough to keep the whole database. The advantage of using FG-index is that we minimize $T_{I/O}$ since $C_q = \mathcal{D}_q$. More importantly, we completely eliminate the large T_{isSubG} since candidate verification is not required. The following equation shows the response time of processing a query by FG-Query.

$$T_{response} = (T_{search} + |\mathcal{D}_q| \times T_{I/O}) \approx (|\mathcal{D}_q| \times T_{I/O}) \ . \quad (2)$$

5.3.2 Query is not an FG

When FG-Query returns no result, then $q \notin \mathcal{F}$. In this case, *IFG-Query*, as shown in Algorithm 3, is called to obtain $\mathcal{D}_q$. IFG-Query consists of two parts: Lines 1-10 process on the set of frequent distinct edges E (if any) in q, while Lines 11-12 handle the set of infrequent distinct edges (if any).

First in Lines 1-10, IFG-Query uses the core FG-index to find a small set of subgraphs of q that are indexed. Then for each subgraph g found, $\mathcal{D}_g$ is retrieved and added to S_{core}. Then in Lines 11-12, IFG-Query retrieves $\mathcal{D}_e$ for each infrequent distinct edge e of q from Edge-index and includes all $\mathcal{D}_e$ in S_{edge}. Finally in Lines 13-15, IFG-Query generates C_q by intersecting all ID sets (i.e., $\mathcal{D}_g$ or $\mathcal{D}_e$) that are in S_{core} and S_{edge}, and produces the answer set $\mathcal{D}_q$ by verifying if $g \supseteq q$ for each $g \in C_q$.

Algorithm 3 IFG-Query

Input: The core FG-index, Edge index, and a query q.
Output: $\mathcal{D}_q$.

```
1.  Let $E$ be the set of distinct edges in $q$,
    except the infrequent distinct edges;
2.  for each $i = size(q) - 1, size(q) - 2, \dots, 1$ do
3.      $C \leftarrow \left(\bigcup_{1 \le j \le count(e,q)} IDA(j, i, e)\right), \forall e \in E$;
4.      Sort $C$ in descending order;
5.      for each $ID$ in $C$ do
6.          if ($g$ in GA[$ID$] has edges in $E$ and $g \subset q$)
7.              Retrieve $\mathcal{D}_g$, and add $\mathcal{D}_g$ to $S_{core}$;
8.              Remove all distinct edges in $g$ from $E$;
9.              if ($E = \emptyset$)
10.                 go to Line 11;
11. for each infrequent distinct edge $e$ in $q$ do
12.     Retrieve $\mathcal{D}_e$ from Edge-index, and add $\mathcal{D}_e$ to $S_{edge}$;
13. $C_q \leftarrow \left(\bigcap_{S \in (S_{core} \cup S_{edge})} S\right)$;
14. for each $g \in C_q$ do
15.     Include $g$ in $\mathcal{D}_q$ if $g \supseteq q$;
```

We now explain how to search for the subgraphs of q that are indexed in the core FG-index. Unlike the search for the supergraph of q in FG-Query, the search for subgraphs moves in the reverse direction starting with those indexed graphs that have one fewer edge than q (Line 2). Then, the IDs of the graphs (stored in C) are sorted in descending order (Line 4), since for graphs of the same size, a larger ID implies a smaller frequency of g and hence a smaller $\mathcal{D}_g$.

Line 6 performs a subgraph isomorphism test between g and q to ensure g is a subgraph of q before using $\mathcal{D}_g$ to produce $\mathcal{D}_q$. Since the ID set C is the union instead of the intersection of all $IDA(j, i, e)$, processing all IDs in C may be costly (Line 5). To save the number of subgraph isomorphism tests in this step, we obtain only a smaller number of *maximal subgraphs* of q (Line 8 removes all distinct edges in a found subgraph from E, while Line 6 checks whether a graph contains edges in E that do not appear in any found subgraphs). Here, if g is a maximal subgraph of q, then $\nexists g' \supset g$ such that $g' \subset q$. Using maximal subgraphs of q is effective to reduce the size of C_q because $\forall g' \subset g$, $\mathcal{D}_{g'} \supseteq \mathcal{D}_g$. Note that we do not obtain all maximal subgraphs of q in the index but stop the search when all edges in E are covered (Line 9), since obtaining all those missing maximal subgraphs does not further reduce the size of C_q substantially.

Response Time Analysis for IFG-Query. T_{search} mainly consists of the time taken to perform the subgraph isomorphism testing when searching for the maximal subgraphs of q (Line 6) and the I/O time to retrieve $\mathcal{D}_g$ (the graph IDs) from the disk (Line 7). Since we only require a small number of maximal subgraphs of q, T_{search} is small compared with the rest part of $T_{response}$.

In mining frequent patterns, including FGs, it is well-known that the frequency of a pattern is closest to that of its largest sub-pattern or smallest super-pattern. Thus, the $\mathcal{D}_g$ of a maximal subgraph g of q is close to $\mathcal{D}_q$. In addition, the maximal subgraphs of q obtained by IFG-Query have the smallest frequency among other indexed subgraphs of q. Therefore, we can deduce $|C_q| \approx |\mathcal{D}_q|$ in most cases. Thus, we obtain the response time of processing a query by IFG-Query as follows.

$$\begin{aligned} T_{response} &= T_{search} + |C_q| \times (T_{I/O} + T_{isSubG}) \\ &\approx T_{search} + |\mathcal{D}_q| \times (T_{I/O} + T_{isSubG}) \ . \quad (3) \end{aligned}$$

Since q is an infrequent subgraph, $|\mathcal{D}_q|$ is bounded by $(\sigma \cdot |\mathcal{D}|)$. In fact, when q contains any infrequent edge e, we have $|C_q| \le |\mathcal{D}_e| \le (\sigma \cdot |\mathcal{D}|)$. Thus, we can control $T_{response}$ using σ. Although the number of FGs can be large when σ is small, the FGs are indexed into the nested IGIs whose sizes are controlled by δ. In practice, we do not need to use a very small σ that returns a set of FGs needed to be indexed for more than two levels of IGIs.

5.4 Memory and Disk Residence

The IGI at the first level of the core FG-index, the δ-TCFGs, and Edge-index are resident in the main memory. All the other parts of FG-index are resident in the disk.

The $\mathcal{D}_g$ for each indexed graph g is stored in the disk. Given two graphs g and g', if $g \supset g'$, then $(\mathcal{D}_g \cap \mathcal{D}_{g'}) = \mathcal{D}_g$ and hence a large number of graph IDs are duplicates. We keep the exact set of $\mathcal{D}_g$ for each $g \in \mathcal{T}$. Then, we construct a tree on $CLOS(g)$ as follows. The root of the tree is g. For each $g' \in CLOS(g)$, if $\nexists g'' \in CLOS(g)$ such that $g'' \supset g'$, g' is connected as a child of the root. For each of the rest $g' \in CLOS(g)$, and for a graph g_p in the tree, if $g_p \supset g'$ and $\nexists g'' \in CLOS(g)$ such that $g'' \supset g'$ and $g'' \prec g_p$, then g' is added as a child of g_p. Then, given g_p and its child g', $\mathcal{D}_{g'} = (\mathcal{D}_{g'} - \mathcal{D}_{g_p})$. Thus, only $\mathcal{D}_g$ is exact, while the duplicate IDs in $\mathcal{D}_{g'}$ are removed and can be recovered by traversing from g' up to the root g.

5.5 Insert and Delete Maintenance

We discuss briefly the update maintenance of FG-index.

Let g be a new graph to be inserted into $\mathcal{D}$. For each $g' \subseteq g$, we locate g' in the core FG-index and add the ID of g to $\mathcal{D}_{g'}$. If an infrequent distinct edge e or a new distinct edge e' is found in g, we add the ID of g to $\mathcal{D}_e$ or add e' to Edge-index.

Let $g \in \mathcal{D}$ be deleted from $\mathcal{D}$. For each $g' \subseteq g$, we locate g' in the core FG-index and delete the ID of g from $\mathcal{D}_{g'}$. For any infrequent distinct edge e in g, we delete the ID of g from $\mathcal{D}_e$ in Edge-index.

If $size(g)$ is large, g may contain a large number of subgraphs. However, we do not actually search each subgraph of g one by one. When we find the closest δ-TCFG supergraph, g_t, of some $g' \subseteq g$, if $g_t \subseteq g$, then we only need to add/remove the ID of g to/from $\mathcal{D}_{g_t}$ and skip searching all g'', where $g' \subseteq g'' \subset g_t$. The update for all such g'' can be skipped because the duplicate graph IDs are removed for each graph in the closure of g_t, as discussed in Section 5.4. So we only need to update $\mathcal{D}_{g_t}$, while $\mathcal{D}_{g''}$ can be recovered from $\mathcal{D}_{g_t}$. Therefore, the number of searches and updates is approximately the number of δ-TCFGs that are subgraphs of g, which is small since the total number of δ-TCFGs is small.

5.6 Discussions

A limitation of FG-index is that the index construction cost depends largely on the cost of mining FGs and the efficiency of our algorithm for computing $\mathcal{T}$ mainly depends on the size of $\mathcal{F}$. Although many efficient FG mining algorithms [22, 20, 15] have been proposed in recent years, a more promising way of improving the performance of the construction of FG-index is to mine the set of δ-TCFGs directly from the database.

In Algorithm 1, we compute the set of exact δ-TCFGs. However, for the purpose of indexing, we do not require an accurate set of δ-TCFGs. Instead, an approximate set of δ-TCFGs is sufficient, as far as the number of δ-TCFGs is small enough for the main memory to hold the memory-resident portion of FG-index. The following lemma states that FG-index constructed on an approximate set of δ-TCFGs can be used to answer any query $q \in \mathcal{F}$ without candidate verification.

LEMMA 4. *Let $\mathcal{T}$ be a set of FGs randomly selected from $\mathcal{F}$ such that $\mathcal{T} \supseteq \mathcal{M}$. For each $g \in (\mathcal{F} - \mathcal{T})$, the closest δ-TCFG supergraph of g exists and is unique.*

PROOF. Since $\mathcal{T} \supseteq \mathcal{M}$, by the definition of MFGs and Definition 4, the closest δ-TCFG supergraph of $g \in (\mathcal{F} - \mathcal{T})$ must exist. While the uniqueness follows directly from Definitions 3 and 4. $\square$

Lemma 4 ensures the correctness of query processing using FG-index even if $\mathcal{T}$ is a randomly selected set of FGs that violates the definition of δ-TCFGs. In Lemma 4, the set of MFGs, $\mathcal{M}$, is included in $\mathcal{T}$ to make sure all $q \in \mathcal{F}$ can be answered without candidate verification. Thus, going without this constraint only means that some queries in $\mathcal{F}$ may need to go through candidate verification. If we further relax this constraint in computing the δ-TCFGs, it is highly possible that a more efficient algorithm can be designed to mine an approximate set of δ-TCFGs. In mining condensed frequent patterns, a very efficient approximate algorithm is proposed in [4]. It is interesting to study whether the concept of their work can be applied to mine δ-TCFGs, which is considered as future work.

6. PERFORMANCE EVALUATION

In this section, we verify the efficiency of index construction and query processing of FG-index by comparing it with the state-of-the-art graph index, *gIndex* [24]. The set of FGs, which is the input to the construction of FG-index, is mined using *gSpan* [22]. We run all experiments on an AMD Opteron 248 with 8GB RAM, running Linux 64-bit.

We use the real dataset that is used in the evaluation of gIndex [24]. It is an AIDS antiviral screen dataset containing 43K graphs. We denote this dataset as *AIDS* in our experiment.

To test the scalability of FG-index on database size, graph size and graph density, we design a synthetic graph dataset generator based on the IBM synthetic transaction generator [1], which allows us to specify the number of graphs in the dataset, the average density and size of each graph, the number of distinct node/edge labels, etc. (see details in graphGen[2]).

We set $\delta = 0.1$ for FG-index in all experiments except in Section 6.1.2 when we assess the effect of δ on the performance of FG-index. The settings of gIndex are the same as suggested in [24], that is, the maximum size of a discriminative FG is 10 and the minimum frequency threshold for a discriminative FG of size 10 is $(0.1 \times |\mathcal{D}|)$.

6.1 Performance on Real Graph Dataset

We assess the performance of FG-index on the *AIDS* dataset for different choices of σ and δ.

6.1.1 Effect of σ

Table 1 reports the index construction time and memory consumption for *AIDS*, where the time of FG-index is represented as (*FG mining time + FG-index construction time*). In the table, $|\mathcal{T}|$ is the number of δ-TCFGs indexed in FG-index and $|\mathcal{F}_d|$ is the number of discriminative FGs indexed in gIndex.

Table 1: Index Construction Performance for *AIDS*

	FG-index ($\sigma = 0.1$)	FG-index ($\sigma = 0.01$)	gIndex
Time (sec)	10.03 (10+0.03)	1085 (627+458)	217
Memory (MB)	2	95	107
$|\mathcal{T}|$ or $|\mathcal{F}_d|$	195	21893	3276

The time of constructing FG-index for *AIDS* at $\sigma = 0.1$ is 20 times smaller than that of gIndex, but that at $\sigma = 0.01$ is 5 times larger. The tremendous variation in the construction performance of FG-index is because the number of FGs at $\sigma = 0.1$ is significantly smaller than that at $\sigma = 0.01$ (455 vs. 59120). As a result, the number of indexed graphs and the memory consumption at $\sigma = 0.1$ are also significantly smaller than those at $\sigma = 0.01$. The result also shows that the time for mining FGs occupies a large portion of the total index construction time for FG-index.

Although constructing FG-index is costly at smaller σ, the construction of FG-index at $\sigma = 0.1$ performs better than that of gIndex. More importantly, we next show that query performance of FG-index at $\sigma = 0.1$ is as efficient as that at $\sigma = 0.01$.

To evaluate the query performance of FG-index, we use the six query sets tested in gIndex [24], denoted as Q_4, Q_8,

[2]http://www.cse.ust.hk/graphgen/

Q_{12}, Q_{16}, Q_{20}, Q_{24}, where each Q_i contains 1000 queries and each query in Q_i consists of i edges.

Figure 4 reports the average response time of processing a query in each Q_i. The result shows that query processing using FG-index is over two orders of magnitude faster than that using gIndex for Q_4, and about an order of magnitude faster for other sets of queries. We find that, for both $\sigma = 0.1$ and $\sigma = 0.01$, most of the large-size queries (almost all queries with 12 or more edges) are not FGs and hence candidate verification is also needed for FG-index. Therefore, the improvement of FG-index over gIndex is smaller for queries of larger size. However, the result reveals an advantage of using FG-index, that is, even for processing queries that are not FGs, the performance of FG-index is also significantly better than that of gIndex.

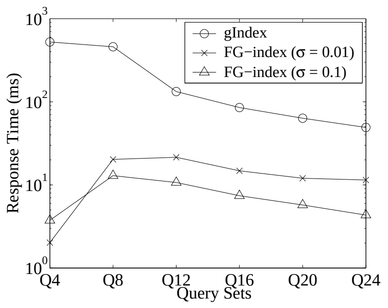


Figure 4: Response Time on Varying Query Sizes

This experiment also shows that FG-index at $\sigma = 0.01$ is only better than that at $\sigma = 0.1$ for processing Q_4. This is because smaller σ results in more queries in Q_4 to become FGs. More specifically, setting $\sigma = 0.01$ allows 30% more queries in Q_4 to be answered without candidate verification than setting $\sigma = 0.1$. However, when the size of the queries increases, the number of FGs in other Q_i is almost the same for both $\sigma = 0.1$ and $\sigma = 0.01$. In this case, the time spent on candidate verification is almost the same and T_{search} becomes the major factor for query performance. As a result, FG-index at $\sigma = 0.01$ is worse because it has a larger search space than FG-index at $\sigma = 0.1$.

The peak memory consumption of query processing using FG-index is about 1.8 MB and 13 MB at $\sigma = 0.1$ and $\sigma = 0.01$ for all Q_i, while that of gIndex is on average 30 MB.

Analysis on the Cost of Candidate Verification. The high cost of candidate verification can be inferred from several aspects of the experimental results. First, FG-index performs the best for Q_4 since more queries are FGs and fewer candidate verifications are performed than for other Q_i. On the contrary, gIndex performs the worst for Q_4 since each query $q \in Q_4$ has a larger $\mathcal{D}_q$, which results in a greater number of candidate verifications. Second, for queries with size greater than 4, the performance of both FG-index and gIndex improves when the size of the queries increases. The performance improvement comes from the smaller number of candidate verifications required for queries of larger size, even though in general subgraph isomorphism testing is more expensive for larger graphs.

More Tests on Query Performance. Since the size of the queries in each Q_i is very regular, we test the query performance on another two sets of queries. We first mine the set of FGs on the *AIDS* dataset at $\sigma = 0.005$ and then randomly select 1000 FGs as queries. In this way, the selected query set contains both frequent and infrequent graphs with different sizes. Among the 1000 queries, 85% of them are not FGs at $\sigma = 0.01$ and almost all of them (only two exceptions) are not FGs at $\sigma = 0.1$.

Table 2: Query Performance for *AIDS*

	FG-index ($\sigma = 0.1$)	FG-index ($\sigma = 0.01$)	gIndex
Response Time (sec)	0.031	0.040	0.57
Memory (MB)	2	14	47

Table 2 reports the average response time and the peak memory consumption for processing a query. Although most of the queries are not FGs, processing a query using FG-index for both $\sigma = 0.1$ and $\sigma = 0.01$ is over an order of magnitude faster than that using gIndex. The memory consumption of FG-index for both $\sigma = 0.1$ and $\sigma = 0.01$ is also significantly smaller than that of gIndex, even though the number of δ-TCFGs in FG-index at $\sigma = 0.01$ is much larger than the number of discriminative FGs in gIndex, as reported in Table 1.

6.1.2 Effect of δ

We assess the effect of δ on the performance of FG-index at $\sigma = 0.1$ using *AIDS* dataset. We choose δ at values of 0, 0.05, 0.1, 0.2 and 1. We skip δ between 0.2 and 1 to show a clear effect of increasing δ on the query performance.

Table 3: Index Construction Performance on δ

	δ=0	δ=0.05	δ=0.1	δ=0.2	δ=1
Time (sec)	10.03	10.03	10.03	10.03	10.03
Memory (MB)	2	2	2	2	2
$\lvert\mathcal{T}\rvert$	455	255	195	156	92

Table 3 shows that the index construction performance is very stable for all values of δ. The time recorded is 10 seconds for mining FGs and 0.03 second for constructing FG-index in all cases. The number of 0.05-TCFGs drops to half that of 0-TCFGs (i.e., CFGs), while the number of 1-TCFGs (i.e., MFGs) is only 20% that of 0-TCFGs. The memory consumption is consistently 2 MB, which does not decrease with the decrease in the number of δ-TCFGs because the input to the index construction is the set of FGs.

We use the query sets Q_4, Q_{12} and Q_{24}, as well as a set of 1000 queries randomly selected from the set of FGs mined at $\sigma = 0.05$, denoted as Q_{rs} in Figure 5.

Figure 5 shows that the response time increases steadily with the increase in δ. The increase in response time is because the number of δ-TCFGs that are indexed and resident in memory is smaller for a larger δ and hence more I/O time is needed to access the disk-resident index. However, the increase in response time is small since the search time is only a small portion of the total response time. We observe that the increase in the response time for δ from 0.2 to 1 is much greater for Q_{12} and Q_{24}. The reason is analyzed as follows. If a query q is not an FG, then C_q is computed from the intersection of the $\mathcal{D}_g$ of a set of maximal subgraphs of q, where each $\mathcal{D}_g$ is retrieved from the disk. Thus, a query with a larger size is likely to have more maximal subgraphs

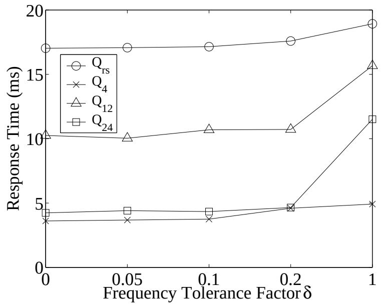


Figure 5: Response Time on Varying δ

that are indexed and hence more disk accesses to retrieve their $\mathcal{D}_g$. As a result, the increase in the response time of processing Q_{12} and Q_{24} is greater than that of Q_4. Since Q_{rs} contains queries of both small and large sizes, the increase in the response time is in-between that of Q_4 and Q_{12}.

The memory consumption decreases with increasing δ for all queries, which matches with the decrease in the number of δ-TCFGs, since the memory-resident index is built on the δ-TCFGs. We omit detailed figures since the decrease is small (from 1.9 MB at $\delta = 0$ to 1.6 MB at $\delta = 1$).

6.2 Performance on Synthetic Graph Dataset

In the following set of experiments, we use the synthetic datasets to test the scalability of FG-index on different database sizes, graph sizes and graph density.

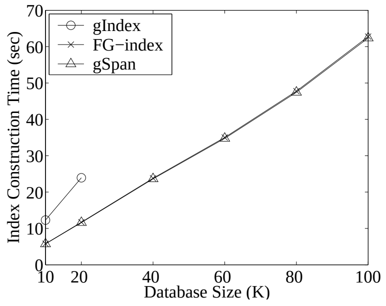
6.2.1 Effect of Database Size

We generate six synthetic datasets to assess the performance of FG-index on different database sizes from 10K to 100K graphs. The average number of graph edges in each dataset is 30, the number of distinct labels is 30, and the average graph density is 0.15. We build FG-index on the set of FGs at $\sigma = 0.01$.

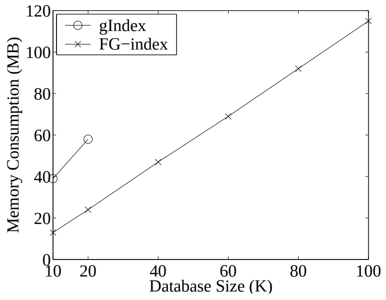
Figure 6(a) reports the index construction time. We also show the time of mining the set of FGs at $\sigma = 0.01$, denoted as "gSpan" in Figure 6(a). The line for gSpan almost coincides with the line for FG-index, which means that the time of mining FGs dominates the total construction time of FG-index. We are not able to construct gIndex for the datasets with 40K or more graphs. For the datasets that consist of 10K and 20K graphs, Figure 6(a) shows that the index construction time of gIndex is twice larger than that of FG-index. The memory consumption of constructing FG-index, which includes the memory used to mine FGs, is also lower than that of constructing gIndex, as reported in Figure 6(b).

We randomly select 1000 queries from the set of FGs mined from each dataset at $\sigma = 0.005$. About 40% of the selected queries are not FGs at $\sigma = 0.01$. Figure 7(a) shows that processing a query using FG-index is almost three orders of magnitude faster than using gIndex.

Figure 7(b) shows that the memory consumption of querying using FG-index is also significantly smaller than that using gIndex, which implies that the memory-resident portion of FG-index is smaller than that of gIndex. We also find that the memory usage of FG-index is very stable and is not affected by the increase in the database size. This is because

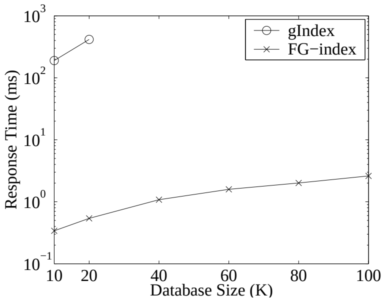


(a) Construction Time

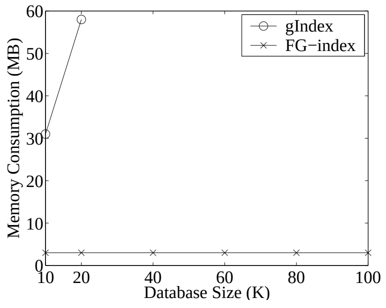


(b) Memory Consumption

Figure 6: Index Construction Performance on Varying Database Sizes



(a) Response Time



(b) Memory Consumption

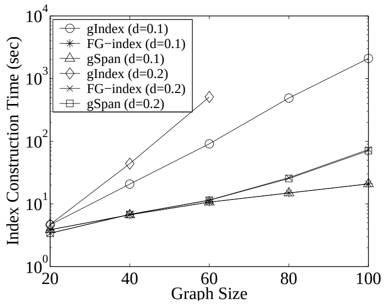
Figure 7: Query Performance on Varying Database Sizes

the number of δ-TCFGs is very stable for different database sizes, which verifies that the concept of δ-TCFGs is effective in controlling the size of the memory-resident portion of FG-index.

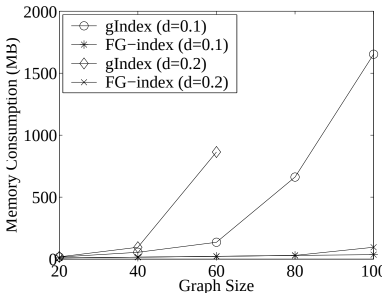
The results of this experiment thus demonstrate that FG-index is more scalable on the database size than gIndex.

6.2.2 Effect of Graph Size and Density

For this experiment, we generate 10 datasets, which are classified into two sets of five datasets. We vary the average size of the graphs in each of the five datasets from 20 to 100. The graphs in the first set of datasets have an average density of 0.1 while those in the second set have an average density of 0.2. Other settings of the synthetic datasets are the same as those in Section 6.2.1, except that the database size is fixed at 10K. We construct FG-index from the set of FGs mined at $\sigma = 0.05$.



(a) Construction Time

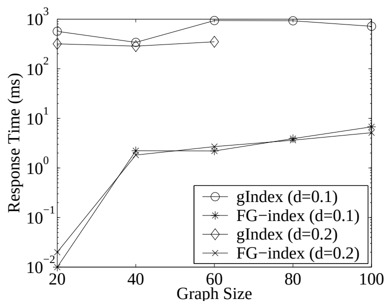


(b) Memory Consumption

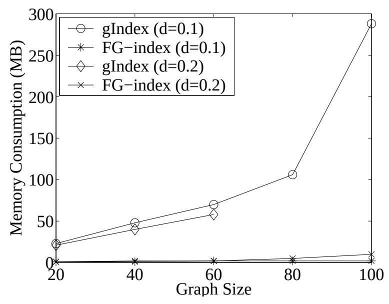
Figure 8: Index Construction Performance on Varying Graph Sizes and Graph Density

Figure 8(a) reports the index construction time, which again shows that the time of mining FGs dominates the total construction time of FG-index. Compared with gIndex, the construction time of FG-index is significantly less in all cases and up to two orders of magnitude less when the size of the graph becomes larger. The memory consumption of constructing FG-index is also significantly less as reported in Figure 8(b). For the same graph size, a larger density results in a larger number of FGs in the database. Therefore, the construction cost of both FG-index and gIndex increases when the graph density d increases from 0.1 to 0.2. However, the increase rate of FG-index is much smaller than that of gIndex because the effect of graph density is smaller on the number of δ-TCFGs than that on the number of FGs. When $d = 0.2$, we are not able to construct gIndex on graphs with 80 or more edges. The effect of graph size is also smaller on the construction cost of FG-index than that of gIndex, as indicated by the slope of the lines.

We prepare 10 sets of queries for testing each dataset. Each query set contains 1000 queries randomly selected from the set of FGs mined from the corresponding dataset at $\sigma = 0.025$. About 80% of the queries in each query set are not FGs at $\sigma = 0.05$, except that almost all the queries in the two query sets derived at graph size 20 are FGs.



(a) Response Time



(b) Memory Consumption

Figure 9: Query Performance on Varying Graph Sizes and Graph Density

As shown in Figure 9(a), processing a query using FG-index is over two orders of magnitude faster than that using gIndex in all cases. Figure 9(a) also shows that, for graph sizes over 40, the query performance of FG-index increases, but only slightly, with the increase in the graph size, while that of gIndex is recorded a larger variation over different graph sizes. The query processing of FG-index at graph size 20 is extremely fast because all the queries derived at graph size 20 are FGs, which can be processed by FG-index without candidate verification. Figure 9(b) shows that the memory consumption of using FG-index is small and stable, and is significantly less than that of using gIndex.

In addition to the remarkable and stable performance over different graph sizes, the results of this experiment also show that the performance of FG-index is more stable on the graph density than gIndex.

7. RELATED WORK

There have been a number of studies on developing indexing models on graph databases. Due to the diversity of graph models as well as the complexity of graph processing, most of the existing graph indexes are designed for specific applications. For example, Daylight [2] and AnMol [18] are indexes on molecular structures. DataGuides [10], T-index [16], Index Fabric [8], APEX [5], F&B-index [14], and D(k)-index [3] are for semi-structured data and XML. Most of these indexes are based on path or subtree structures.

For the indexing techniques that are developed for more general graph models, GraphGrep [17] is a path-based approach to index graph databases. However, the set of paths in a graph database is huge and hence may affect the performance of the index. To address the weakness of the path-based approach, gIndex [24] is proposed as a graph-based indexing approach and it is reported in [24] that gIndex significantly outperforms GraphGrep. We have discussed and compared with gIndex in Sections 3.2 and 6. Apart from the query processing on a database that consists of a collection of graphs, searching subgraphs in a single large graph is studied in GraphDB [9] and SUBDUE [6, 11].

The concept of δ-TCFGs is inspired by that of the compressed frequent patterns [21, 4]. The goal of their work is to reduce the number of frequent patterns and δ also defines the accuracy of the estimated frequency of a recovered pattern. We apply the concept of compression to index graph databases, which is a challenging problem in an entirely different context, and our use of δ does not involve the frequency estimation of a graph.

8. CONCLUSIONS

In this paper, we propose FG-index, which is an effective index for supporting efficient query processing on graph databases. A distinguished feature of FG-index is that queries that are FGs are answered by FG-index without performing candidate verification, while queries that are not FGs are answered with minimal number of candidate verifications. Thus, our work achieves a significant improvement over existing work on graph indexing [17, 24] that requires candidate verification for all queries. Although the set of FGs, upon which FG-index is built, can be large when the minimum frequency threshold σ is small, we propose a new compression technique which effectively condenses a large set of FGs into a small set of representative FGs, called δ-TCFGs. Our experiments verify the effectiveness of δ-TCFGs in compressing the set of FGs as well as the efficiency of FG-index in query processing. For a wide range of graph datasets, our experimental results show that FG-index significantly outperforms the state-of-the-art graph index, gIndex [24], on both index construction and query processing. In particular, we show that not only for queries that are FGs, but also for queries that are not FGs, the query processing using FG-index is up to orders of magnitude faster than gIndex.

Acknowledgements. This work is partially supported by RGC CERG under grant number HKUST6185/03E. We would like to thank Dr. Xifeng Yan and Prof. Jiawei Han for providing us gIndex and gSpan. We also thank Mr. Yi Wang and the anonymous reviewers for their valuable comments.

9. REFERENCES

[1] IBM Synthetic Data Generation Code for Associations and Sequential Patterns. `http://www.almaden.ibm.com/cs/projects/iis/hdb/Projects/data_mining/mining.shtml`, 1996.

[2] Daylight theory manual - daylight version 4.9. *Daylight Chemical Information Systems, Inc. www.daylight.com*, 2006.

[3] Q. Chen, A. Lim, and K. W. Ong. D(k)-index: an adaptive structural summary for graph-structured data. In *Proc. of SIGMOD*, pages 134–144, 2003.

[4] J. Cheng, Y. Ke, and W. Ng. δ-tolerance closed frequent itemsets. In *Proc. of ICDM*, pages 139–148, 2006.

[5] C.-W. Chung, J.-K. Min, and K. Shim. Apex: an adaptive path index for xml data. In *Proc. of SIGMOD*, pages 121–132, 2002.

[6] D. J. Cook and L. B. Holder. Substructure discovery using minimum description length and background knowledge. *Journal of Artificial Intelligence Research*, 1:231–255, 1994.

[7] S. A. Cook. The complexity of theorem-proving procedures. In *Proc. of STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971.

[8] B. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon. A fast index for semistructured data. In *Proc. of VLDB*, pages 341–350, 2001.

[9] R. H. Güting. GraphDB: Modeling and querying graphs in databases. In *Proc. of VLDB*, pages 297–308, 1994.

[10] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proc. of VLDB*, pages 436–445, 1997.

[11] L. Holder, D. Cook, and S. Djoko. Substructure discovery in the subdue system. In *Proc. of KDD*, pages 169–180, 1994.

[12] J. Huan, W. Wang, J. Prins, and J. Yang. Spin: mining maximal frequent subgraphs from graph databases. In *Proc. of KDD*, pages 581–586, 2004.

[13] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Proc. of PKDD*, pages 13–23, 2000.

[14] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering indexes for branching path queries. In *Proc. of SIGMOD*, pages 133–144, 2002.

[15] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proc. of ICDM*, pages 313–320, 2001.

[16] T. Milo and D. Suciu. Index structures for path expressions. In *Proc. of ICDT*, pages 277–295, 1999.

[17] D. Shasha, J. T. L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *Proc. of PODS*, pages 39–52, 2002.

[18] S. Srinivasa and S. Kumar. A platform based on the multi-dimensional data model for analysis of bio-molecular structures. In *Proc. of VLDB*, pages 975–986, 2003.

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

[20] C. Wang, W. Wang, J. Pei, Y. Zhu, and B. Shi. Scalable mining of large disk-based graph databases. In *Proc. of KDD*, pages 316-325, 2004.

[21] D. Xin, J. Han, X. Yan, and H. Cheng. Mining compressed frequent-pattern sets. In *Proc. of VLDB*, pages 709–720, 2005.

[22] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *Proc. of ICDM*, pages 721–724, 2002.

[23] X. Yan and J. Han. Closegraph: mining closed frequent graph patterns. In *Proc. of KDD*, pages 286–295, 2003.

[24] X. Yan, P. S. Yu, and J. Han. Graph indexing based on discriminative frequent structure analysis. *ACM Trans. Database Syst.*, 30(4):960–993, 2005.