The VLDB Journal (2013) 22:229–252
DOI 10.1007/s00778-012-0284-8

REGULAR PAPER

Lindex: a lattice-based index for graph databases

Dayu Yuan · Prasenjit Mitra

Received: 18 October 2011 / Revised: 31 May 2012 / Accepted: 13 June 2012 / Published online: 15 September 2012

Abstract Subgraph querying has wide applications in various fields such as cheminformatics and bioinformatics. Given a query graph, q, a subgraph-querying algorithm retrieves all graphs, $D(q)$, which have q as a subgraph, from a graph database, D. Subgraph querying is costly because it uses subgraph isomorphism tests, which are NP-complete. Graph indices are commonly used to improve the performance of subgraph querying in graph databases. Subgraph-querying algorithms first construct a candidate answer set by filtering out a set of false answers and then verify each candidate graph using subgraph isomorphism tests. To build graph indices, various kinds of substructure (subgraph, subtree, or path) features have been proposed with the goal of maximizing the filtering rate. Each of them works with a specifically designed index structure, for example, discriminative and frequent subgraph features work with gIndex, δ-TCFG features work with FG-index, etc. We propose Lindex, a graph index, which indexes subgraphs contained in database graphs. Nodes in Lindex represent key-value pairs where the key is a subgraph in a database and the value is a list of database graphs containing the key. We propose two heuristics that are used in the construction of Lindex that allows us to determine answers to subgraph queries conducting less subgraph isomorphism tests. Consequently, Lindex improves subgraph-querying efficiency. In addition, Lindex is compatible with any choice of features. Empirically, we demonstrate that Lindex used in conjunction with subgraph indexing features proposed in previous works outperforms other specifically designed index structures. As a novel index structure, Lindex (1) is effective in filtering false graphs, (2) provides fast index lookups, (3) is fast with respect to index construction and maintenance, and (4) can be constructed using any set of substructure index features. These four properties result in a fast and scalable subgraph-querying infrastructure. We substantiate the benefits of Lindex and its disk-resident variation Lindex+ theoretically and empirically.

D. Yuan (✉)
Department of Computer Science and Engineering,
The Pennsylvania State University, University Park, PA, USA
e-mail: duy113@psu.edu

P. Mitra
College of Information Sciences and Technology,
The Pennsylvania State University, University Park, PA, USA
e-mail: pmitra@ist.psu.edu

1 Introduction

Graphs are widely used to model structures and relationships of objects in various scientific and commercial fields. For instance, chemical molecules are modeled as graphs [16], and three-dimensional mechanical parts are stored and manipulated as attributed graphs in a CAD-mechanical-components database [8]. Graphs are also used in pattern recognition and have broad applications in computer vision and image processing [6].

A popular method of retrieving graphs from graph databases is by performing a *subgraph query*. Given a graph database, D, and a query graph, q, a subgraph-querying algorithm retrieves all graphs $g \in D$ containing q as a subgraph. Deciding whether one graph is a subgraph of another is referred to as the subgraph isomorphism problem; the problem has been shown to be NP-complete [7]. Consequently, for large databases, an index is necessary to enable efficient query processing. Typical graph indices are sets of key-value pairs. The key is a subgraph of a graph in the database, and the value consists of a list of database graphs that contain the subgraph. When the query graph is a key in the index, the value can be returned directly as the answer. Otherwise, the

index is used to return a *candidate set* $C(q)$ of graphs that may potentially contain the query. $C(q)$ is typically larger than the answer set $D(q)$, $|C(q)| \geq |D(q)|$. A subgraph isomorphism test is performed to check that q is contained in each candidate graph in $C(q)$. Thus, $|C(q)|$ subgraph isomorphism tests must be performed to eliminate graphs that are not in $D(q)$($D(q)$ will also be referred to as the *supporting set* of q in the rest of the paper). Existing methods for subgraph indexing and querying use such a *filtering + verification* paradigm [5,13–15,19,21,23]. Previous works have mainly focused on mining "good" substructure features for indexing.[1] A good feature set improves the filtering power by reducing the number of candidate graphs, which leads to a reduction in the number of subgraph isomorphism tests in the verification step [5,15,19]. Subtree features are also mined for indexing, and they are less time-consuming to be mined in comparison with subgraph features [21,23].

In related works [5,13,19,23], different graph index structures have been used for different kinds of features; no index structure is general enough to support all kinds of substructure features. For example, gIndex [19] cannot index the δ-TCFG features (which is designed specifically for FG-index [5]) because it does not support the search for the closest δ-TCFG supergraph of the query as needed by the FG-index method. At the same time, FG-index cannot index the discriminative and frequent subgraph features of gIndex efficiently, since no apriori pruning can be made during the index lookup [19]. For the same reason, gIndex cannot support MimR features [15] efficiently. We show in Sect. 7 that using gIndex to index MimR features results in a significant increase in the index-lookup cost, which dominates the overall query-processing time when the query is a relatively large graph. The index structure of TreePi [21] only supports subtree features. The iGraph framework benchmarked existing indexing methods and concluded that "there is no single winner for all experiments" [9]. This observation motivates the need for a graph index that can be implemented in a graph database management system and is independent of the features being indexed. The DBMS can have an extensible library of feature-selection strategies that are application-dependent and any of them can be chosen to use with the index structure.

We propose and evaluate the lattice-structure index, Lindex.[2] In Lindex, each node is associated with a key-value pair. A key is a (substructure) feature, say sg, and its value set V, as in an inverted index, is the set of database graphs (IDs) that contain sg. In Lindex, an edge between two index nodes indicates that the key in the parent node is a subgraph of the key in the child node. In Fig. 1, we show a simple Lindex. The root node in the Lindex has a key sg_0, the empty graph (a graph with no nodes or edges). The node sg_0 has two children nodes with keys sg_1 ($sg_0 \subset sg_1$) and sg_3 ($sg_0 \subset sg_3$).

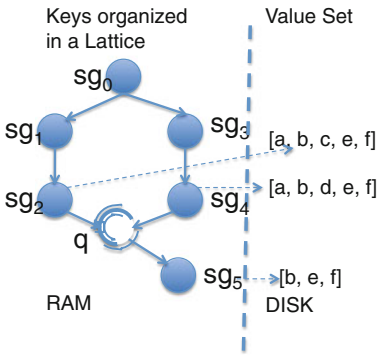


Fig. 1 Example of Lindex

We show in detail how Lindex is designed and implemented independent of the choice of index features. Lindex provides the following: (1) high filtering power (Sect. 4), (2) fast index-lookup strategies (Sect. 5), (3) compact memory consumption (Sect. 3), and (4) fast index construction and maintenance (Sect. 6.3).

High filtering power: In previous methods [5,13–15,19,21,23], when a query graph was not a key in the index, the number of subgraph isomorphism tests needed to answer a query q was at least $|D(q)|$. *Can we design a method that may require fewer subgraph isomorphism tests than the size of the answer set* $|D(q)|$*?* We can do so using Lindex. Lindex utilizes the fact that *database graphs that contain a supergraph of a query q are guaranteed to be in the answer set for* q; those graphs do not need to be checked for subgraph isomorphisms. Consider the example in Fig. 1. Let q be the query. Given the Lindex, $maxSub(q)$ contains the maximal-subgraph features of q, which are also direct parents of q, namely, sg_2 and sg_4. The intersection of their value sets $[a, b, c, e, f]$ and $[a, b, d, e, f]$ is $[a, b, e, f]$, which is the candidate set of answers. Our algorithm also finds the minimal-supergraph features of q, $minSup(q)$. In our example, the minimum supergraph of q (direct child of q) is sg_5 whose value set is $[b, e, f]$. From the Lindex, by construction, we know that b, e, f contain sg_5. Therefore, the database graphs b, e, f are guaranteed to contain q. Hence, we can directly put $[b, e, f]$ in the answer set resulting in saving three costly subgraph isomorphism tests. In our method, only the graph a has to be checked, while all previous works have to check all of $[a, b, e, f]$. No indices in previous works support the minimal-supergraph-feature lookup, except for the FG-index. However, FG-index returns only one supergraph of q (closed δ-TCFG supergraph) [5], while our proposed Lindex returns all minimal-supergraph features of queries.

[1] Substructure features will be simply referred to as features in the rest of the paper.

[2] A preliminary version of Lindex was reported in an online-only workshop proceeding [20].

As we show in Sect. 5.2, Lindex supports fast lookup for minimal-supergraph features of queries.

A second property of Lindex allows us to partition the value sets such that subgraph isomorphism tests need to be performed only on database graphs appearing in one partition resulting in further reduction in the candidate set. For details of this property, please see Sect. 4.2.

Fast index lookup: Finding the maximal-subgraph features of a query graph, say, q in Fig. 1, requires checking for subgraph isomorphisms from the indexed substructure, say, sg_2 to q. Looking up maximal-subgraph features using Lindex is efficient because instead of identifying a full (subgraph) mapping from sg_2 to q from scratch, by walking down the lattice, our system can *incrementally* grow a mapping from a parent graph, sg_1 (sg_3) to q to generate a mapping from a child graph sg_2 (sg_4) to q. In order to make this traversal even faster, we propose a heuristic spanning-tree-based traversal of the lattice that works well in practice (see Sect. 5.1).

Compact memory consumption: Lindex is compact with respect to memory consumption. In Lindex (see Fig. 1), there is an edge from sg_1 to sg_2 if sg_1 is a subgraph of sg_2. Since $sg_1 \subset sg_2$, the label of sg_2 can be stored as an extension of the label of sg_1, thereby saving space required to record each feature. Say, sg_1 has an edge labeled $\mathcal{A}$ and sg_2 has two edges with labels $\mathcal{A}$ and $\mathcal{B}$. Instead of storing sg_2 as a graph with two edges, we store sg_2 as sg_1 + edge $\mathcal{B}$.

Easy to construct: Lindex is also easy to construct given the features. The construction requires no other information except for the containment relationships between features. All existing feature-mining algorithms [5,13,15,19,23] mine the containment relationships of features and then perform feature selection. For example, given a feature f, gIndex finds all features $f_i \subset f$ to decide whether f is a discriminative feature; FG-index finds all features $f_j \supset f$ to decide whether f is a δ-TCFG. The construction of Lindex takes no extra subgraph isomorphism tests if the containment relations are recorded by the feature miner. In addition, due to the explicit recording of the lattice structure and fast maximal-subgraph lookup, as we will show in Sect. 7, Lindex is fast on constructing value sets in comparison with other indices.

Note that the benefits of Lindex due to its lattice structure can be reaped in conjunction with the benefits due to other feature-selection techniques. For example, as we show later, using Lindex with the features used by FG-index outperforms FG-index. Lindex is also flexible enough that it can be partially stored in main memory and partially on disk (see Sect. 3.3). Lindex also allows us to use different strategies for selecting the in-memory features and the disk-resident features. In one of our experiments, we use MimR [15] to select the in-memory features but the index nodes stored on disk have keys corresponding to frequent subgraphs in the database as in FG-index. As future research identifies better feature-selection algorithms for a set of workload queries whose answers should be pre-computed and stored on disk, for example, Frequently Asked non-FG-Queries [4], we can seamlessly plug those algorithms into the Lindex framework.

In summary, our contributions are listed as follows:

1. We propose Lindex, an index structure independent of index features.
2. Lindex filters more false answers than other indices given the same index features, reduces the number of subgraph isomorphism tests over false positive candidates, and results in lower verification costs.
3. Lindex is fast with respect to index lookup, using both maximal-subgraph-feature search and minimal-supergraph-feature search for queries.
4. Lindex is concisely structured and requires no extra data structure, for example, feature-inverted-index [4].
5. Lindex is easy to construct and maintain.

We demonstrate the superiority of Lindex by comparing it with FG-index [5], gIndex [19], Tree+δ [23], SwfitIndex [13], and MimR [15] both theoretically and empirically (Sect. 7). We observed that when the query is small, Lindex can filter out 1/3 more false graphs than gIndex (Fig. 10a, b). Lindex's index-lookup time is 1/10 of that of FG-index for large queries (Fig. 10c). In addition, the memory consumption of Lindex is only half of that of other indices (Table 3). Also, the index-structure construction cost of Lindex is only 1/6 of that of FG-index's construction time (Table 4).

Organization: We organize the paper as follows: Sect. 2 introduces preliminaries and related works. Section 3 introduces the structure of Lindex. In Sect. 4, we present the strategies adopted by Lindex to generate a smaller candidate set to save on the verification cost. In sect. 5, we show how to efficiently lookup Lindex. The overall query-processing framework using Lindex and its time complexity is discussed in Sect. 6. We compare Lindex against existing methods and show our empirical results in Sect. 7. Finally, we conclude in Sect. 8.

2 Preliminaries and related work

2.1 Preliminaries

A graph $g = (V, E, L)$ is defined on a set of vertices V and a set of edges E such that $E \subseteq V \times V$. Each node v or edge e is associated with one label $L(u)$ or $L(e) \in L$. In this paper, we consider databases of *undirected, labeled, and connected* graphs, but our methods can be applied to other types of graphs with minor modifications. Existing works have used frequent subgraphs (trees) as index features [5,13,19,21,23].

Table 1 Symbols and terms summarization

Symbol	Description
D	Graph database; a collection of graphs
q	Query graph
$D(g)$	Supporting set of g in database D
$C(q, F)$	Candidate set of query q with features F
$g_1 \subset g_1$	Graph g_1 is a subgraph of g_2
$\mathcal{DL}(\mathcal{D})$	Dense graph lattice based on D
$\mathcal{SL}(\mathcal{F})$	Sparse graph lattice consisting features F
$maxSub(g, F)$	Maximal subgraphs of g in lattice $\mathcal{SL}(F)$
$minSup(g, F)$	Minimal supergraphs of g in lattice $\mathcal{SL}(F)$
$V_d(sg), V_i(sg)$	Direct/Indirect value set of node with key sg
$M(g_1, g_2)$	All mappings from graph g_1 to g_2

Definition 1 (*Frequent subgraphs*) In a graph database D, the *support* of a subgraph sg is the number of graphs $g \in D$ that contain sg. The graphs containing sg in D comprise the *supporting set* of sg, $D(sg)$. The *frequency* of sg, $freq(sg)$, is the ratio between sg's supporting set and $|D|$, $|D(sg)|/|D|$. The subgraph sg is a *frequent subgraph* if and only if its support is greater than $\sigma|D|$, where σ is named as the minimum support.

Subgraph querying is defined as follows: *given a query graph q, find q's supporting set in the graph database D.* In order to answer subgraph queries, typically, one has to perform (sub)graph isomorphism tests.

Definition 2 (*Subgraph isomorphism*) We say two graphs $g_1 = (V_1, E_1, L)$ and $g_2 = (V_2, E_2, L)$ are *isomorphic* to each other if there is a bijection between V_1 and V_2, that is, a mapping $M(g_1, g_2)$ that maps any adjacent vertices u_1, v_1 in g_1 to a pair of adjacent vertices $u_2 = M(u_1)$, $v_2 = M(v_1)$ in g_2, where $L(u_1) = L(u_2)$, $L(v_1) = L(v_2)$, and $L(E(u_1, v_1)) = L(E(u_2, v_2))$, and vice-versa. *Subgraph isomorphism* between g_1 and g_2 is an isomorphism between g_1 and a graph g_2' that is a subgraph embedded in g_2.

In Table 1, we summarize several symbols and terms that appear frequently in this paper.

2.2 Subgraph querying

Because subgraph isomorphism is NP-complete, scanning a large graph database and checking whether each database graph contains a query is computationally infeasible. Inclusive-logic-based filtering algorithms use the following: if a database graph g does not contain a feature f that is contained in the query q, then g cannot be a supergraph of q, and can be filtered out [5,13,15,19,21,23]. Usually, an inverted index is built to implement the above inclusive-logic

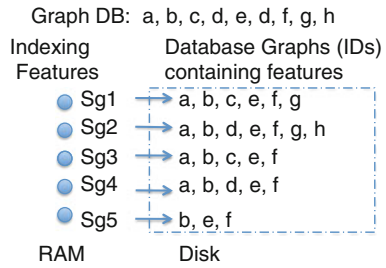


Fig. 2 Graph inverted index for subgraph querying

filtering, as shown in Fig. 2. Each feature is a substructure stored in memory, and there is a value set (supporting set) $D(sg)$ associated with each feature sg. The value sets (IDs of graphs containing the feature sg) are stored on disk. Given a query q, we first look up the index for a feature isomorphic to q. If there exists such a feature $sg_i = q$, $D(sg_i)$ is returned directly as the answer. Otherwise, we look for features contained in the query, say, sg_3 and sg_4 in Fig. 2. The value sets of sg_3 and sg_4 are intersected to obtain a candidate set, $C(q) = D(sg_3) \cap D(sg_4) = \{a, b, e, f\}$. Implicitly, graphs $\{c, d, g, h\}$ are pruned out. Finally, each candidate graph in $C(q)$ is checked via a subgraph isomorphism test to see whether it contains q. This procedure is similar to that used to process simple conjunctive queries using the Boolean Retrieval Model for documents. However, it is more challenging than using the Boolean retrieval model for documents, because of the following factors:

1. An exponential number of subgraph features could be indexed, which cannot fit the memory. So, it may be preferable to index a small set of features $F \subset \mathcal{F}$.
2. Finding the features contained in the query (maximal-subgraph-feature search), for example, $sg_3 \subset q$ and $sg_4 \subset q$ in Fig. 2, involves subgraph isomorphism tests or graph canonical labeling, both of which are NP-hard problems.
3. Verifying each candidate graph with subgraph isomorphism test is expensive in comparison with string matching.

In order to address the above difficulties, previous works mine a set of features (challenge 1) to minimize the number of isomorphism tests (challenge 3) [5,13,15,19,21,23]. Correspondingly, in order to reduce the time taken to lookup the index (challenge 2), different index structures have been built for different types of features. For example, gIndex uses a hash table to store all the canonically labeled subgraph features [19]; FG-index uses a specially designed edge-inverted index to organize all features [5]. However, there is no index structure that can efficiently support all different kinds of features.

In this paper, we propose Lindex and a novel filtering model described in Sect. 4 to decrease the size of candidates $C(q, F)$ from that obtained by previous methods (challenge 3), for any choice of features F. This modified filtering model needs the underlying graph index to support the fast search of the maximal-subgraph features and minimal-supergraph features of a given query (challenge 2). We show in Sect. 7 that the index lookup in Lindex (maximal-subgraph-feature search and minimal-supergraph-feature search) is much faster than that in traditional indexes, such as gIndex and FG-index. Furthermore, Lindex is concise and easier to construct than other indexes, for example, FG-index, which has several components.

2.3 Related work

In this section, we survey the related work in brief. First, we discuss existing work on feature-based inverted indexes, and then, we discuss other alternatives to indexing graph databases.

2.3.1 Feature-based inverted indexes

All feature-based inverted indexes adopt the inclusive-logic filtering introduced in Sect. 2.2. Han et al. have shown that feature-based inverted indexes perform the best in comparison with other indexes [9], which proves the power of the inclusive-logic filtering.

The first inverted index approach, GraphGrep, indexes all paths up to length 10 [14]. Yan et al. [19] proposed gIndex, which indexes frequent and discriminative subgraphs. gIndex has a higher filtering power than GraphGrep, since subgraphs can record more structural information than paths.

In gIndex, discriminative features f are indexed as black nodes and indiscriminative intermediate features $f' \subset f$ are indexed as white nodes. Both black and white nodes are stored in memory. The index in gIndex is a hash table where the keys are the DFS codes of the features and their values are the corresponding supporting sets, as in Fig. 3. For white nodes (indiscriminative features), there is no value set. When a query graph q comes in, the search algorithm first enumerates all frequent subgraphs of q, canonically labels them, and searches the label in the hash table. A frequency-based apriori rule can be used to prune the search space during index lookup to find maximal-subgraph features for queries: if an enumerated subgraph f is not frequent (not in the hash table), then any of f's supergraphs cannot be frequent or reside in the hash table either; hence, it is safe to skip enumerating f's supergraphs. In order to use this apriori rule, gIndex stores lots of frequent features although they are not discriminative (i.e., the white node in the index). Hence, it is not space efficient. In addition, this frequency-based apriori rule cannot be used on other features, for example, MimR features (introduced later). If we organize the MimR features with gIndex (as proposed by Sun et al. [15]), the maximal-subgraph-feature search is time-consuming because of the large unpruned search space (see Sect. 7 for details).

features	type	Link to Value Set
Sg1	White	
Sg2	Black	[a, b, c, e, f]
Sg3	White	
Sg4	Black	[a, b, d, e, f]
Sg5	White	[d,e,f]

Fig. 3 Example of gIndex structure

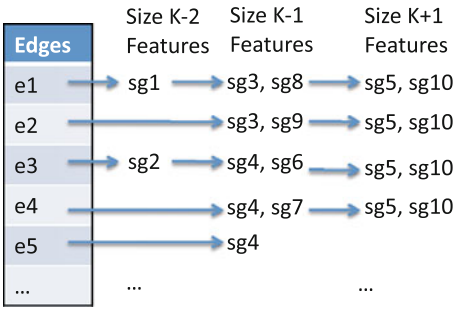


Fig. 4 Example of FG-index

Sun et al. [15] have proposed an approach for mining *informative* subgraphs called MimR. The MimR subgraphs are learnt based on a set of possible queries. The subgraphs are selected using a forward feature-selection algorithm based on information theory to maximize the average filtering power using those training queries. In order to maximize the filtering power, the algorithm uses an approximate method to select a set of subgraph features that have the minimum redundancy among them while trying to maximize their information content. We found that MimR constructs a candidate set $C(q)$ that is smaller than that created using other features when the same number of features are indexed.

Cheng et al. [5] index *frequent-subGraph queries* or *FG queries*, which they define as queries with large supporting sets $D(q)$s. FG-index pre-computes and stores the answer sets for all possible FG queries on disk, so that FG queries can be answered directly without isomorphism tests. FG-index further mines δ-TCFG features to index FG queries and stores them in memory. A frequent subgraph is δ-TCFG if and only if $\nexists$ a frequent subgraph g' s.t. $g' \supset g$ and $freq(g') \geq (1-\delta) freq(g)$. FG-index indexes the δ-TCFG features using an edge-inverted index, in which the key is a distinct edge e contained in δ-TCFG features and the value set is the set of δ-TCFG features containing e, as shown in Fig. 4.

The δ-TCFG features (IDs) in the value sets are assigned to different buckets according to their edge count.[3]

A join operation of the value sets is conducted while searching for minimal-supergraph features of q, and a union operation is conducted while searching for maximal-subgraph features of q. For example, given a size-k query q having distinct edges $\{e_1, e_2, e_3, e_4\}$, the minimal-supergraph candidates of q, $\{sg_5, sg_{10}\}$, can be found by joining value sets of e_1, e_2, e_3, and e_4 (the candidates must contain more than k edges). The set of candidates for the minimal supergraphs of q is still large due to the loss of the structural information by using the edge index (as reported by Chen et al. [4]). Hence, in FG*-index (an advanced FG-index), an additional "inverted-feature-index" is added [4]. The keys of the "inverted-feature-index" are subgraphs of the δ-TCFG features, and the value sets are the IDs of δ-TCFG features. Those subgraphs are selected to have edge count between $l = 2$ and $u = 4$. The inverted-feature-index improves the time taken to lookup the index for the closest-δ-TCFG supergraph feature, but it needs additional memory to store the keys and the value sets [4]. The situation is worse for maximal subgraphs; the subgraph features of q can contain any of the four edges; and they can be $C = \{sg_3, sg_4, sg_6, sg_7, sg_8, sg_9\}$. Each of the candidate feature $sg_i \in C$ need to be checked with subgraph isomorphism tests. In order to save the index-lookup time for the maximal-subgraph-feature search in FG-index, FG-index does not return the complete set of maximal-subgraph features [5], which inevitably decreases the filtering power of the index. In addition, FG-index is optimized for queries that are frequent subgraphs (FG queries) in the database. Unfortunately, for queries that are not frequent subgraphs in the database, non-FG queries, FG-index falls back to the filter+verify strategy. Thus, for non-FG queries, the filtering power of FG-index is lower than that of MimR because the δ-TCFG features used by FG-index to index the graph database are mined without trying to maximize the filtering power. As we show in Sect. 7, FG-index does not optimize the average performance when the workload contains both FG and non-FG queries.

Identifying which subgraphs to index is slow because the mining process involves a significant amount of testing for subgraph isomorphisms [18]. To reduce the time required to mine features, three *subtree*-based approaches were proposed [13,21,23]. TreePi [21] and SwiftIndex [13] use frequent and discriminative subtrees for indexing. SwiftIndex organizes features as QI-sequences (string labels) in a prefix tree. When a query q comes in, SwfitIndex traverses the prefix tree and checks whether the feature corresponding to an index node is a subgraph of the query using subgraph isomorphism, which is similar to what Lindex checks. However, SwiftIndex only supports the lookup for maximal-subgraph features. Tree+δ, as the name denoted, has two separate indices, one gIndex hash table for subtree features and a δ index for subgraph features; the latter can largely improve the filtering power [23]. The frequency-based *a prioi* rule can be used on the gIndex hash table because Tree+δ indexes discriminative and frequent subtree features. However, there is no pruning for the δ index. The filtering power of subtrees is lower than that of subgraphs. When offline index construction is affordable, we still prefer a subgraph-based index because of its higher filtering power.

We summarize the various existing feature-based inverted index structures and their properties in Table 2. The "Index-Structure" column shows the core data structure of the index. The "Filtering Effectiveness" column shows the filtering power of the selected features and filtering models. We rate the filtering power of each index using "fair," "good," and "best." The "Index-Lookup" column shows the methods used by different algorithms to find maximal-subgraph or minimal-supergraph features, and their corresponding time costs. "The Extra-Memory" column describes what additional memory the index needs besides that for storing the features.

2.3.2 Other graph indexes

Besides feature-based inverted indexes, researchers have also proposed some "tree-like" indexes [10,17]. In CTree, graphs in a database and their subgraphs or closure graphs are organized using an index tree. The answer set is retrieved by traversing this index tree. However, in CTree, subgraph isomorphism or approximate subgraph isomorphism has to be tested on each step of the tree traversal, which leads to longer response times in comparison with the inverted-index-based approaches mentioned above. Williams, Huang, and Wang proposed a Graph Decomposition Index (GDIndex) [17]. GDIndex contains all graph decompositions of graphs in the database, where the graph decomposition of a graph g refers to the enumeration of all connected, induced subgraphs of g. GDIndex is very effective in processing graphs with limited size (less than 20 nodes). The index is usually large and does not fit in main memory for large graphs.

In the past few decades, researchers in cheminformatics developed an approach called *Fingerprint* to filter out false answers [1]. Each chemical structure's fingerprint is a feature vector in a high-dimensional space. A feature may be defined as follows: an atom, a pattern representing an atom and its nearest neighbors, a pattern representing a group of atoms, a path up to n bonds in length, etc [12]. The feature space includes all these patterns and consequently has a high dimension. To conquer the high dimensionality, a hash code

[3] For simplicity, we did not show the bucket assignment by edge frequency.

Table 2 Related works on subgraph querying

Name	Index structure	Filtering effectiveness	Index lookup	Extra memory
gIndex	A hash table of features DFS codes	Inclusive-logic filtering (fair)	Support maximal-subgraph-feature search only. Use Apriori rule to prune the search space (fast)	Extra memory for indiscriminative features (white node)
MimR	A hash table of features DFS codes	Inclusive-logic filtering (good)	Support maximal-subgraph-feature search only. No pruning available (slow)	
FG-index	An edge-inverted index. Features are distributed both in memory and on disk	Direct answer retrieval for FG queries (good). Inclusive-logic filtering for Non-FG queries (fair)	Support maximal-subgraph-feature search. A union operation is conducted to find the candidate max-sub features (slow). Also support closest δ-TCFG supergraph-feature search (slow)	Extra memory for edge-inverted index
FG*-index	Additional feature-inverted index and a FAQ Index	Same as FG-index.	Improved closest δ-TCFG supergraph-feature search (fast)	Extra memory for feature-inverted index
QuickSI	Swift Index: a prefix tree of subtree features' QI-sequences	Inclusive-logic filtering (fair)	Support max-subtree-feature search only (fast)	
Tree+δ	gIndex for subtree features and a separate hash table δ for subgraph features	Inclusive-logic filtering (fair)	Support maximal-subgraph-feature search only. Apriori rule used for gIndex (fast), but no pruning for δ index (slow)	
Lindex	Organize features in a lattice	Modified inclusive-logic filtering and direct inclusion of true answers (best)	Support maximal-subgraph-feature and minimal-supergraph-feature search (fast)	
Lindex+	Features are distributed in memory and on disk	Same as Lindex. Direct answer retrieval for indexed queries (best)	same as Lindex	

of the feature vector is used instead for filtering; however, hash coding of the vector reduces the filtering power.

GCode [24] encodes database graphs into spectrum codes based on spectral graph theory and uses interlacing theorem to prune the false candidate graphs. Other works, such as GString [11], primarily target two-dimensional chemical structure graphs. Lindex can be used to index more generic graphs.

In this paper, we extend our preliminary version of Lindex [20]. Detailed algorithms for index-structure construction (Sect. 3.2.2) and value-set construction (Sect. 4.4) are introduced. We have optimized the maximum-subgraph-search and minimum-supergraph-search algorithms. We construct a minimum spanning tree based on the frequency of the index features, as introduced in Sect. 5.1.3. In this paper, we show how this minimum spanning tree can help to reduce the isomorphism tests in maximal-subgraph search. The maximal-subgraph features obtained using the algorithm in the preliminary version contains false positives. In this paper, we design algorithm to prune out those false positives efficiently as introduced in Sect. 5.2.2. We compare and contrast the index lookup of Lindex and related works in Sects. 5.1.1 and 5.2.1. We show how we can extend Lindex to a disk-based index, Lindex+, and present its query processing framework and discuss its time complexity (Sect. 6). Additionally, in Sect. 6.3, we discuss how to update Lindex briefly.

This paper also extends the empirical evaluation over the preliminary version. Apart from the query processing times, we report the index construction and memory consumption costs. Apart from comparisons with gIndex, FG-index, and MimR, in this paper, we also provide comparisons with QuickSI and Tree+δ. We test the scalability of Lindex with additional indexing features. As reported in Sect. 7.3, we control the number of indexing features using minimum support. We also study the performance Lindex with real and synthetic graphs with various edge count and density parameters, as reported in Sect. 7.4. Apart from empirical evidence, in this paper, we also prove the correctness of the value-set-partition heuristic (Sect. 4.3).

3 A lattice-based graph index

In this section, we propose Lindex, a graph-lattice-based index. First, we introduce a graph lattice, and then, we show how a compact index, Lindex, is constructed based on the graph lattice.

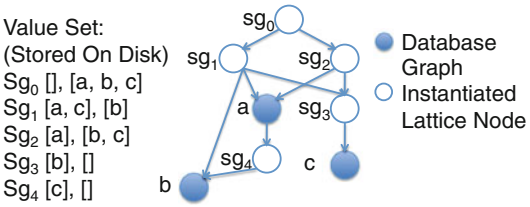


Fig. 5 Example of Lindex: a sparse lattice. The database *graphs* are not part of the index, it is drawn for demonstration purpose

3.1 Graph lattice

In a graph lattice, each node represents a graph. Any pair of graphs $\{g_A, g_B\}$ in the lattice have a least upper bound $g_A \cup g_B$ and a greatest lower bound $g_A \cap g_B$. A complete graph lattice has a *greatest element* that is a graph containing all graphs in the lattice and a *least element* that is an empty graph $\emptyset$.

Definition 3 (*Graph lattice*) A graph lattice $(GL, \subseteq)$ is a lattice defined using the subgraph isomorphism relation, $\subseteq$, and satisfies reflexive, antisymmetric, and transitive properties as follows. Given three graphs g_A, g_B, and g_C in a graph lattice,

$$\begin{aligned} &(1)\ g_A \subseteq g_A \\ &(2)\ \text{if } g_A \subseteq g_B \text{ and } g_B \subseteq g_A, \text{ then } g_A = g_B \\ &(3)\ \text{if } g_A \subseteq g_B \text{ and } g_B \subseteq g_C, \text{ then } g_A \subseteq g_C \end{aligned} \quad (1)$$

As shown in a Hasse diagram of a graph lattice in Fig. 5, the lattice is composed of a set of graphs. A directed edge is drawn from sg_2 to sg_3 if and only if $sg_2 \subset sg_3$. Here, we omit edges that can be obtained using the transitivity of the subgraph isomorphism relation, that is, there is no edge $E(sg_0, sg_3)$ if $sg_0 \subset sg_2 \subset sg_3$. If a lattice constructed from a database contains all the subgraphs of all the graphs in the database, we refer to it as a *Dense Lattice*, $\mathcal{DL}(\mathcal{D})$. A lattice is a *Sparse Lattice*, $\mathcal{SL}(\mathcal{F})$, if it only consists a subset, F, of subgraphs in the dense lattice. A dense lattice is too large to generate and manipulate for most databases in practice. Figure 5 is a sparse lattice, that is, not all subgraphs of database graphs are listed. Note that the solid nodes in Fig. 5 represent database graphs, and they are not part of the sparse lattice but visualized only for demonstration purposes.

In a sparse graph lattice, $\mathcal{SL}(\mathcal{F})$, for each graph g, the maximal subgraphs and minimal supergraphs of g are defined as follows:

Definition 4 (*maxSub, minSup*)

$$maxSub(g, F) = \{sg_i \in F | sg_i \subset g, \nexists x \in F \text{ s.t. } sg_i \subset x \subset g\} \quad (2)$$

$$minSup(g, F) = \{sg_i \in F | g \subset sg_i, \nexists x \in F \text{ s.t. } g \subset x \subset sg_i\} \quad (3)$$

In Fig. 5, graphs sg_1 and sg_2 are maximal subgraphs and the graph sg_4 is a minimal supergraph of graph a. For brevity, when the Lindex being referred to is clear from the context, we simplify the notation by dropping the second argument from $maxSub$ and $minSup$ and write them as $maxSub(g)$ and $minSup(g)$.

3.2 Structure of Lindex

Lindex, $L(D, F)$, is an inverted index built on a sparse graph lattice $\mathcal{SL}(\mathcal{F})$ and a graph database D. In Lindex, a *node* is associated with a *<key,value>* pair. The key is a subgraph (feature) in the graph lattice $\mathcal{SL}(\mathcal{F})$, and the value is a list of IDs of graphs in the database that are supergraphs of the key. In this paper, we denote t as a Lindex node, sg_t as the corresponding graph key of t, and $V(sg_t)$ as the value of t. Node $t_1 \prec t_2$ if and only if $sg_{t_1} \subset sg_{t_2}$. Given a graph g and a Lindex $L(D, F)$, its maximal-subgraph nodes are the nodes corresponding to $maxSub(g, F)$ in the sparse lattice $\mathcal{SL}(F)$, and its minimal-supergraph nodes are the nodes corresponding to $minSup(g, F)$. The *root* of a Lindex is a node r whose key is an empty graph $\emptyset$. In this paper, we abuse the notation slightly when the meaning is clear from the context. A node may refer to either a node in Lindex or a graph in a graph lattice, based on the context.

Lindex versus GDIndex: Lindex is different from the lattice structure in GDIndex [17]. GDIndex is a dense lattice, in which all subgraphs of the database graphs are instantiated. A hash table is also built whose keys are the canonical labels of subgraphs, and values are the links to those subgraphs on the dense lattice. In order to process the query q, GDIndex first converts q to its canonical form and looks up the hash table to locate a subgraph $sg = q$. Then, GDIndex searches for database graphs containing sg by walking through the dense lattice. This query-processing model is not based on the inclusive-logic filter+verification paradigm, and no isomorphism tests are needed for verification. However, the dense lattice is too big to fit into memory. Hence, GDIndex is not scalable to large graph datasets. Lindex, on the other hand, is built upon a sparse lattice, which only stored subgraph features selected for indexing by miners studied in previous works [5,15,19,23]. Hence, Lindex is scalable to large databases and large graphs. Lindex does not use a hash table. For each feature in the sparse lattice, Lindex stores an associated list of graph IDs on disk as value sets. Furthermore, Lindex uses the inclusive logic for filtering, and that is different from the query-processing model of GDIndex.

3.2.1 Serializing the index features

Graph canonical labels, such as "DFSCode" [18] of subgraphs, are commonly used to store subgraph features in a feature-based index [15,19]. Here, the label is a string

representation of the entire graph, which is different from the label for nodes and edges. The canonical label (DFSCode) is constructed as follows. For each edge in a graph, a tuple of the form <ID(u), ID(v), L (u), L (edge(u, v)), L (v)> is created for an edge between u and v where the functions ID($\cdot$) and L($\cdot$) return the id and the label of a vertex and/or edge, respectively. The label for the graph is a sequence of tuples constructed from its edges. Different assignments of vertex-ids to vertices in a graph result in different labels. The different labels of a graph are sorted lexicographically; the first label is chosen as the canonical label.

Efficient memory usage enables us to index additional graph features and thus increases the filtering power of a graph index. In order to minimize memory usage, Lindex stores the label of a graph key of a node as an extension of the label of its (chosen) parent node. (We discuss how a parent is chosen and the rationale behind choosing a parent in Sect. 5.1.3 in more detail). For example, the label of the graph sg_3 is <1,2,6,1,7>,<1,3,6,2,6> and the label of its chosen parent sg_2 is <1,2,6,1,7>. The label of sg_3 is stored as just <1,3,6,2,6>. The existence of the first edge (belonging to sg_2) is understood to be in sg_3 and is not explicitly stored to save memory. Besides reducing memory consumption while saving key graphs, extension labeling also integrates an implicit *identity mapping* from all the vertices in the parent graph to vertices in the children with the same vertex-ids as those in the parent. These implicit mappings are useful in mapping expansion while traversing the lattice (for more details, see Sect. 5.1).

Algorithm 1 Construct Lindex lattice

Input: Selected Index Features F
Output: Sparse lattice $\mathcal{SL}(F)$
Procedure:
1: Mine the containment relationships between features F, and store their mappings
2: Sort F in descend order of their edge count.
3: Build a Lattice $\mathcal{SL}(F)$ with disconnected nodes F
4: **for** Each feature (node) sg_i **do**
5: **for** Each feature sg_j with $j < i$ **do**
6: **if** $sg_j \supset sg_i$ and sg_j sg_i is not connected by a path **then**
7: connect sg_i to sg_j by a direct edge
8: **end if**
9: **end for**
10: **end for**
11: Construct a Spanning Tree T out of $\mathcal{SL}(F)$
12: Depth First Search, label each node sg_i as an extension of its parent in T

3.2.2 Lindex lattice construction

Algorithm 1 describes the process of constructing a Lindex. To construct the sparse lattice of Lindex, we need to first find the containment relationships of features, as shown in line 1 of Algorithm 1, which are by-products of the feature miner in the feature-selection phase [5,13,15,19,23]. Since Lindex is independent of features, any substructure features can be used as input to the algorithm. Lines 2 to 10 construct the sparse lattice $\mathcal{SL}(F)$. In line 11, a spanning tree of the lattice is built. The spanning tree is built for fast index lookup, and its construction will be introduced in more detail in Sect. 5.1.3. Then, the index subgraph features are labeled as an extension of their parents in the spanning tree. The extension labeling can be easily computed given the recorded mappings between index features.

The above procedure does not include the construction of the value sets. For each feature sg, if its supporting set $D(sg)$ can be returned directly from the feature mining algorithm, then the value sets can be populated directly. Usually, as in gIndex [19] and cIndex [2], a sampled graph dataset $D' \subset D$ is used for feature mining. Thus, $D'(sg)$ instead of $D(sg)$ will be returned by the feature mining algorithm. In order to find the complete set of $D(sg)$, we need to populate the value sets (details in Sect. 4.4). We show empirically that Lindex is also faster with respect to constructing value sets than other indexes, because of the explicit recording of the lattice structure and its fast maximal-subgraph lookup.

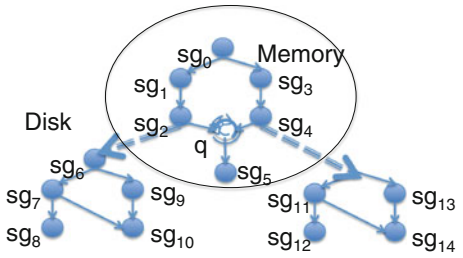


Fig. 6 Example of Lindex+

3.3 Lindex+: disk-resident Lindex

Although compact, Lindex may still be too big to fit in main memory when there are too many indexed subgraph features. As in FG-index [5], in such a case, we propose a disk-resident extension of Lindex, Lindex+, as shown in Fig. 6. Constructing Lindex+ involves two steps: (a) Choose a set of nodes N_1 (sg_0–sg_5 in Fig. 6) from the corresponding lattice to instantiate in memory and construct a Lindex using it; and (b) From the rest of the nodes N_2 from the lattice that are not chosen to be instantiated in memory construct a set of on-disk Lindexes. We refer to the first-level in-memory index as the *first-level* Lindex. The N_2 nodes on disk are linked from the N_1 nodes that are instantiated in main memory.

For example, the on-disk nodes group, for example sg_{6-10}, is linked from sg_2 because sg_2 is the **maximum subgraph** of features sg_{6-10}. A *maximum subgraph* of graph is

a maximal subgraph of the graph with the minimum frequency. If two maximal subgraphs have the same minimum frequency, we choose the first maximal subgraph as the **maximum subgraph** according to their labels lexicographically. Each of the in-memory nodes is associated with a closure of on-disk features, whose maximum subgraph is the in-memory node. For each closure of on-disk features, we build one on-disk Lindex, which is linked by an in-memory node. For example, the on-disk Lindex built on the closure sg_{6-10} is linked by the in-memory node sg_2, in Fig. 6. Since the in-memory and on-disk Lindex share the same structure, we generally name them as Lindex and treat them the same in this paper.

The sparse lattice of the first-level Lindex resides in memory. We store all value sets and all on-disk Lindexes (including both sparse lattices and their value sets) on disk. We have experimented using MimR [15] features and δ-TCFG [5] for the nodes stored in main memory and stored all frequent subgraphs (FG) on disk. Thus, FG queries can be answered directly as in FG-index. Note that this mix-and-match feature of Lindex+ is a result of the generality of the Lindex data structure and allows us to take advantage of the best of multiple worlds.

4 Filtering power of Lindex

In this section, we introduce two strategies that Lindex uses to decrease the size of the candidate sets. Both strategies are independent of underlying features, thus, leading to the generality of Lindex.

4.1 Direct inclusion of true answers

The first strategy used to prune candidate sets is as follows. Any graph g indexed by a supergraph node of a query q can be pruned from $C(q)$ (the set of candidate graphs that are checked with subgraph isomorphism tests) and directly included in the answer set because g contains q. This strategy is formalized in Property 1.

Property 1 (Minimal supergraph pruning) *Given a query q, and Lindex $L(D, F)$, the candidate set on which an algorithm should check for subgraph isomorphism is $C(q) = \cap_i D(f_i) - \cup_j D(h_j)$, $\forall f_i \in maxSub(q)$, and $\forall h_j \in minSup(q)$.*

The key challenge to implementing the above-mentioned direct-inclusion strategy is how to find minimal supergraphs of the query in a time-efficient manner. To the best of our knowledge, no other feature-based indexes can support the minimal-supergraph lookup. FG-index supports the search for the closest δ-TCFG supergraph of the query, but the closest δ-TCFG supergraph is only one of the minimal supergraphs. Cheng et al., [3] have used a similar direct-inclusion strategy for supergraph querying (returns the database graphs contained in the query). In supergraph querying, if a feature $sg \subset q$, then all database graphs contained in sg, $\{g \in D | g \subset sg\}$ will be included in the answer directly. Unlike subgraph querying, the direct inclusion in supergraph querying does not use minimal-supergraph features.

4.2 Pruning using the value set partitions

In this subsection, we show how the need for subgraph isomorphism tests is further reduced by partitioning the value sets in Lindex. We introduce the key idea using an example.

Example 1 In Fig. 7, we show a part of a dense lattice $\mathcal{DL}(D)$. sg_1, sg_2, sg_3, and sg_4 are graph keys of nodes in $\mathcal{DL}(D)$ out of which sg_1, sg_2, and sg_3 are instantiated in the sparse lattice $\mathcal{SL}(F)$ constructed from $\mathcal{DL}(D)$. Let a and b be two graphs in the database. Which database graphs should be in the value set of sg_1 to enable us to answer a query q? In our example, clearly, if $q = sg_1$, then we need both a and b. However, if q is not equal to sg_1, we argue it suffices to keep only b in the value set of sg_1 and not a. We identify the different cases that may occur and show that we do not need to include a in the value set of sg_1 in any of these cases. (1) q exists in the path between sg_1 and sg_2. In this case, sg_2 is a supergraph of q, and thus, a containing sg_2 can be included in the answer set of q without verification and, thus, does not need to be in the value set of sg_1. (2) q exists in the path between sg_2 and a. Because sg_2 is a maximal subgraph of q and not sg_1, the algorithm will not use the value set of sg_1 in this case. (3) For all the other placements of q in the other paths, there is no path from q to a implying that a does not contain q; thus, we do not need a in the value set of sg_1. Thus, the node a that is not need to be included in sg_1's value set is because $\exists sg_2$ such that $a \supset sg_2 \supset sg_1$. Notice, however, graph b needs to be included in sg_1's value set even if $\exists sg_3$ such that $b \supset sg_3 \supset sg_1$ essentially because $\exists sg_4$ that $sg_1 \subset sg_4 \subset b$ and sg_4 is not instantiated. The query q can be placed along the path from sg_1 to sg_4 and q has no containment relationship with sg_3. In this case, graph b will be missed from the answer set if b is not included in the value set of sg_1 (note that sg_4 is not instantiated in the sparse lattice and will not be returned as a minimal supergraph of q).

In existing methods [5,13,15,19,21,23], the value set for a node t consists of $D(sg_t)$. We propose to partition the value set for each node into two parts: the direct value set $V_d(sg_t)$ and the indirect value set $V_i(sg_t)$. Both value sets are used when the query subgraph exists as a key in the lattice; otherwise, an intersection of the direct value sets of the maximal subgraphs of the query gives us the candidate set on which subgraph isomorphism should be performed.

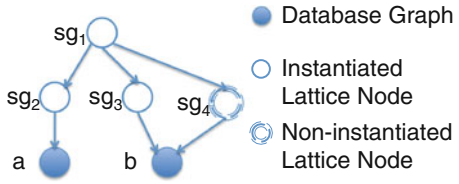


Fig. 7 Example of value set partition

Based on this intuition, we propose the following proposition: If all paths in the dense graph lattice $\mathcal{DL}(D)$, from a graph sg_t to a graph g, where $g \in D$, go through at least one node sg'_t such that $sg'_t \in F$, where F is the set of graphs instantiated in the sparse lattice, then the database graph g should not be included in the direct value set of node t with graph key sg_t in Lindex $L(D, F)$. Hence, the following definition:

Definition 5 (*Direct and indirect value set*) For a Lindex L(D, F), a graph g is in the direct value set $V_d(sg_t)$ iff $g \in D(sg_t)$ and there exists a path in the dense graph lattice, $\mathcal{DL}(\mathcal{D})$, from sg_t to g that does not pass through any node in the instantiated sparse graph lattice $\mathcal{SL}(\mathcal{F})$ That is,

$$V_d(t) = \{g \in D(sg_t) | \exists \text{path } P(sg_t, g) \in \mathcal{DL}(D) \wedge \exists \text{path } P(sg_t, x_1, \ldots, x_n, g) \in \mathcal{DL}(D), \forall x_i \notin F, i \in \{1, n\}\} \quad (4)$$

The *indirect value set* $V_i(sg_t)$ contains all database graphs that are supergraphs of sg_t but not contained in $V_d(sg_t)$.

Our algorithm also uses the following property to reduce the number of subgraph isomorphism tests:

Property 2 (Direct value set pruning) *Given a query q and Lindex $L(D, F)$, the candidate graphs need for verification is* $C(q) = \cap_i V_d(f_i) - \cup_j D(h_j)$, $(\forall f_i \in maxSub(q)$ *and* $\forall h_j \in minSup(q))$.

Instead of using $\cap_i D(f_i)$ as in Property 1, Property 2 uses $\cap_i V_d(f_i)$. The size of $\cap_i V_d(f_i)$ is smaller than that of $\cap_i D(f_i)$, reducing the number of required subgraph isomorphism tests further.

We now provide a simple example to show how our algorithm works and highlight the main advantages of Lindex over existing algorithms.

Example 2 In Fig. 5 we show part of an instantiated sparse lattice with the nodes sg_0, sg_1, sg_2, sg_3, and sg_4. Graphs a, b, c are in the database and are not instantiated in the lattice but they are shown in the graph for illustration purposes only. Let us say that the query is isomorphic to graph a. The lattice would be traversed, and the maximal subgraphs of a are determined to be sg_1 and sg_2. The minimal supergraph of a is sg_4. The algorithm takes the intersection of the direct value sets of sg_1 and sg_2. The lists shown in the figure beside each node depict its value set: the first list is the direct value set, and the second list is the indirect value set. The algorithm takes the intersection of the direct value sets of the maximal subgraphs and obtains the candidate set $\{a, c\} \cap a = a$ for verification. The union of the value sets (both direct and indirect) of the minimal supergraph sg_4 is $\{c\}$ and that is directly included in the answer and deducted from the candidate set. Finally, the set $\{a\}$ is taken, and its element a is verified to contain the query ($a \supseteq q$) and is added to the answer set. The answer set $\{a, c\}$ is output.

The example shown above illustrates the efficacy of Lindex over other indexing schemes like gIndex, FG-index, or MimR. gIndex, FG-index (assuming that a is not an indexed feature), and MimR would find all the maximal subgraphs sg_1 and sg_2. Their value- sets would be $\{a, b, c\}$, and their intersection would be verified. Thus, to generate the answer, they would need three subgraph isomorphism tests. If the query q is a frequent subgraph, both FG-index and Lindex+ would have a node in the index (could be disk-resident) and the value set of a would be retrieved directly.

4.3 Proof of value set partition

We prove the correctness of Property 2 below.

Theorem 1 *If there exists any graph g in a database D that contains a query graph q, the graph g is in the answer set using Property 2.*

Proof **Case 1:** If q appears in the sparse lattice ($q \in F$), then all database graphs in the value set of q are returned. By construction, the graph g will appear in a value set (either direct or indirect) of q since g contains q and thus g is in the answer set.

Case 2a: If q is not indexed ($q \notin F$), let there exist a feature graph sg' in the sparse lattice $\mathcal{SL}(F)$ such that sg' is a minimal supergraph of q and the database graph g is in the value set of sg' ($q \subset sg' \subset g$). In this case, the algorithm finds the complete set of minimal supergraphs of the query q in the sparse lattice, and g will be added into the answer set directly without any subgraph isomorphism test.

Case 2b: If neither q nor any supergraph of q is in F, or $q \notin F$ and none of q's (minimal) supergraph nodes contain g in the value set, then the algorithm finds $maxSub(q)$, the maximal subgraphs of q in $\mathcal{SL}(F)$. (1) The graph g must appear in the intersection of $V_d(f)$, $\forall f \in maxSub(q)$. To show this, we need to show that g appears in the direct value set of all maximal-subgraph nodes of q in $L(F, D)$. The graph g belongs to $D(f)$ because f is a maximal subgraph of q and g contains q, and thus, g also contains q's maximal subgraph f. By Definition 4, a database graph g that contains f will be in $V_d(f)$ unless all paths from f to g in the dense

lattice $\mathcal{DL}(D)$ pass through at least one node from F. There must exist a path from f to g in the dense lattice $\mathcal{DL}(D)$ because g appears in the database and f is a subgraph of g. For the path $P = (f, \ldots, q, \ldots, g)$, its first half $(f, \ldots, q)$ does not pass through any graphs in F (otherwise, f is not a *maximal* subgraph of q). Recall, in this case, neither q nor any supergraph of q that is a subgraph of g is in $\mathcal{SL}(F)$. Thus, the second half $(q, \ldots, g)$ does not pass any graph in F either. Therefore, there exists a path P connecting f and g without passing any graphs in F. Hence, g does not appear in the indirect value set and is in each $V_d(f)$ and thereby in the candidate set C. (2) Because g contains q, g will be added to the answer set.

4.4 Partitioned value-set construction

Value-set construction: Due to the fact that a sampled database $D' \subset D$ is used during the feature mining, the supporting set $D'(f)$, instead of $D(f)$, is returned by the feature miner for each selected feature $f \in F$. Our system needs to populate the value sets in Lindex. As in document indexing, it processes database graphs one after another and adds them to the graph inverted index. For a database graph g, we first find all indexing features contained in g, $sub(g) = \{f \subseteq g | f \in F\}$, and then we append the ID for graph g to the end of f's value set for each $f \in sub(g)$.

In this subsection, we show how our algorithm partitions the value sets for Lindex during value-set population. Let a graph g be in the direct value set of a node with key sg_t, that is, $g \in V_d(sg_t)$. From Definition 4, we know that there exists a path in the dense lattice $\mathcal{DL}(D)$, $p(sg_t, g)$, from t's graph key sg_t to g that does not pass through any node $n \in F$ instantiated in the sparse lattice $\mathcal{SL}(F)$. The direct value set $V_d(sg_t)$ is easy to be obtained if we have the dense lattice $\mathcal{DL}(D)$. However, computing and searching on $\mathcal{DL}(D)$ are infeasible because of its large size. Now, we introduce our algorithm that partition the value sets of nodes in the Lindex without pre-computing and utilizing $\mathcal{DL}(D)$ (see Algorithm 2).

For each database graph g, our algorithm first finds the maximal-subgraph nodes of g and adds g into the direct value set of each maximal-subgraph node (lines 4–7). For index nodes t whose keys are subgraphs of g but not maximal subgraphs, g can be either in their direct value sets or indirect value sets. We try to extend the embeddings of sg_t on g to a certain size $maxSize$ and check whether there is an embedding of sg_t on g that can be extended without passing any graphs corresponding to a lattice node. Here, $maxSize$ is the size (number of edges) of the largest subgraph key in Lindex (line 1). An embedding of sg_t on g is an instance of g's subgraph that is isomorphic to sg_t. We extend the embedding by adding one edge l in each step (lines 12–13), where $l \in g$, $l \notin e$ but l is connected to e in g.

Algorithm 2 Value-Set Construction

Input: Graph Database D, Sparse lattice $\mathcal{SL}(F)$
Output: Lindex L
Procedure:
1: maxSize ← edge count of the largest subgraph in F
2: HashSet labels ← canonical label of each sugbraph in F
3: **for** each graph $g \in D$ **do**
4: $maxSub(g)$ ← maxSub-Search($\mathcal{SL}(F)$, g);
5: **for** each graph $f \in maxSub(g)$ **do**
6: $V_d(f) \leftarrow V_d(f) \cup g$;
7: **end for**
8: **for** each node t that is a subgraph node but not a maximal-subgraph node of g **do**
9: Calculate the set of embeddings $E_{sg_t,g}$
10: **for** each embedding $e = e_{sg_t,g}$ **do**
11: **while** e is extendable **do**
12: Find an unvisited edge, $l = (u, v) \in g$ s.t. $u \in e$ $v \notin e$
13: $e \leftarrow e \cup l$
14: **if** size(e) > maxSize **then**
15: break
16: **else if** The set $labels$ contains canonical-label(e) **then**
17: $e \leftarrow e - l$
18: **end if**
19: **end while**
20: **if** size(e) > maxSize **then**
21: break
22: **end if**
23: **end for**
24: **if** $\exists e$, where e's size $\geq$ maxSize **then**
25: $V_d(sg_t) \leftarrow V_d(sg_t) \cup g$
26: **else**
27: $V_i(sg_t) \leftarrow V_i(sg_t) \cup g$
28: **end if**
29: **end for**
30: **end for**

Thus, if an embedding $e_{sg_t,g}$ can be extended to a embedding of size larger than $maxSize$, without passing through any feature graph in F, then there is a path in the dense lattice $\mathcal{DL}(D)$ from sg_t to g that does not pass through any graph in the sparse lattice. Thus, g should be in the direct value set of node t (lines 24–25). Otherwise, g is in the indirect value set of the node t (line 27). Following the above steps, we populate the value set for each index node in Lindex (the <key, value> pair) and finalize the construction of Lindex.

Partitioning the value set into direct and indirect value sets takes additional computation in comparison with finding the unpartitioned value set. However, we show in our experiments that partitioning the value set by extending the embedding is affordable. In our empirical study, we find that populating the value set of Lindex takes even less time than populating the value set of gIndex and FG-index. This is mainly because Lindex can find the (maximal) subgraph features of the inserted database graph g efficiently as shown in Sect. 5.1.

Because the feature mining and index building are precomputed offline, it will not increase query processing times. Also, to alleviate the computation bottleneck, we can set the

$maxSize$ for each subgraph feature f as the size $f+\delta$, where δ is a manually set parameter.

5 Fast index lookup

Recall that in Sect. 2.2, we introduce the three challenges of solving the subgraph-querying problem. One of the challenges is how to search for the maximal-subgraph features of q efficiently. Although the verification time dominates the overall response time for the subgraph-querying problem most of the time, the index-lookup time can sometimes be a major part of the response time. As we show in Sect. 7, for some large queries, the index-lookup time of gIndex using MimR features exceeds the verification cost. To maintain a low index-lookup cost, FG-index only finds a *subset* of maximal subgraphs of the query, which inevitably decreases the filtering power of the $\delta-$TCFG features. In addition, as introduced in Sect. 4.1, the direct inclusion of true answers needs the support of fast minimal-supergraph-feature search over the index structure. In this section, we introduce efficient algorithms that Lindex uses to search for $maxSub(q, F)$ and $minSup(q, F)$.

5.1 Maximal-subgraph search

5.1.1 Maximal-subgraph search in related work

To find the maximal-subgraph features of a query, q, previous methods like gIndex [19] check whether all possible subgraphs of the query q are indexed. gIndex first enumerates all subgraphs of q, canonically labels them to strings, and then looks them up in a hash table as shown in Fig. 3. The total number of subgraphs of q is exponential in the size of q. To decrease the running time, an apriori-based pruning rule is used on discriminative and frequent features (DF) [19]. Because all frequent subgraphs are indexed, if one subgraph sg of the query q is not indexed then it is not frequent and none of its super graphs can be frequent; its super graphs need not be enumerated. However, for features like MimR, δ-TCFG, such a pruning strategy cannot be applied because not all frequent subgraphs are indexed.

Another method, adopted in FG-index [5] and SwiftIndex [13], checks for containment between each indexed subgraph feature sg and the query q using a subgraph isomorphism test. For FG-index, the maximal subgraphs are obtained as follows. Given a query q containing the distinct edges, say e_1, e_2, e_3, and e_4 as in Fig. 4. All subgraphs containing any of the above four edges can be candidates for the maximal-subgraph features of q. FG-index unions the features containing e_1, e_2, e_3, and e_4 to get a set of candidate maximal-subgraph features and tests each candidate feature starting from the largest feature first to the smallest with subgraph isomorphism tests. There are two drawbacks of this maximal-subgraph-feature search algorithm: (1) too many candidate features are generated due to the union operation, and (2) each isomorphism test is evoked from scratch. Hence, using FG-index searching for maximal-subgraph features is generally slow. FG-index does not return the complete set of maximal subgraphs of the q, leading to the decrease in filtering power of the inclusive-logic filtering.

There are also more advanced indexes supporting fast search of maximal-subgraph features, such as cIndex [2] and GPTree [22]. They were built to address "supergraph querying" but can also be used to find subgraph features of queries. However, in order to support subgraph queries, both need to index additional small features H, which are mined from subgraph-search indexing features F. This hierarchical index, unlike Lindex, requires additional memory to store H and their corresponding value sets.

5.1.2 Lindex maximal-subgraph search

The advantage of maintaining a graph lattice is that instead of constructing canonical labels for each subgraph of q or running an isomorphism test for each index feature, while traversing a graph lattice, mappings constructed to check that a feature graph sg_t is contained in q can be extended to check whether a supergraph of sg_t, sg'_t in the lattice is contained in q by incrementally expanding the mappings from sg_t to q. Also, the isomorphism test from sg'_t to query q can be saved if sg_t is not subgraph isomorphic to q. This computational saving does not require any extra indexes or memory.

Before describing the algorithm to search for maximal-subgraph features, $maxSub(q, F)$, we introduce a spanning tree that is used to facilitate the $maxSub(q, F)$ search. A node sg_t has multiple parents in the sparse lattice $\mathcal{SL}(F)$ (each parent being a subgraph of sg_t). In choosing one parent from whose label the label of a node is derived, we create a spanning tree of the lattice. We use this spanning tree to visit each node in the sparse lattice while identifying $maxSub(q, F)$. The spanning tree is constructed during the construction of Lindex (see Sect. 5.1.3 for details).

In Algorithm 3, $maxSub-search$, the graph lattice is traversed as follows. For the nodes (graphs) in the first level (children of the root node of the lattice), there exists no mapping to the query from nodes in the level above, that is, the root node that corresponds to an empty graph. In this case, the algorithm checks whether each graph sg_c is contained in the query graph q (line 3). If it is, all mappings of sg_c on q are stored and used later when the mappings are expanded to check whether any child, say sg'_c, of c is also contained in q. Let us say that $M(sg_c, q)$ is the set of mappings from sg_c to q. Since sg'_c, as a child of sg_c, is labeled as an extension of sg_c, the vertex $v_i(sg_c)$ in sg_c maps to the vertex $v_i(sg'_c)$ and vice-versa (the identical mapping can be obtained

Algorithm 3 maxSub-search

Input: *A sparse lattice* $\mathcal{SL}(F)$, Query Graph q
Output: $maxSub(q)$ Maximal Subgraphs of q
Procedure: main
1: $maxSub(q) \leftarrow \emptyset$
2: **for** each child sg_c of $\mathcal{L}$'s root **do**
3: $M(sg_c, q) \leftarrow$ all mappings from sg_c to q
4: **if** $M(sg_c, q) \neq \emptyset$ **then**
5: maxSub-Search(sg_c, q, $\mathcal{SL}(F)$, $M(sg_c, q)$, $maxSub(q)$)
6: **end if**
7: **end for**
8: return $maxSub(q)$
Procedure: maxSub-Search
Input: Lattice node sg_c, Query q, Sparse lattice $\mathcal{SL}(F)$, Mappings $\{M(sg_c, q)\}$, $maxSub(q)$
Output: $maxSub(q)$
1: extendable$\leftarrow$ False
2: **for** each child sg'_c of graph sg_c **do**
3: $M(sg'_c, q) \leftarrow \emptyset$ {Mappings from sg'_c to q}
4: **for** each mapping $m \in M(sg_c, q)$ **do**
5: **if** m can be extended to a mapping $m(sg'_c, q)$ **then**
6: $M(sg'_c, q) \leftarrow M(sg'_c, q) + m(sg'_c, q)$
7: **end if**
8: **end for**
9: **if** $M(sg'_c, q)$ is not empty **then**
10: maxSubSearch(sg'_c, q, $\mathcal{SL}(\mathcal{F})$, $M(sg'_c, q)$, $maxSub(q)$)
11: extendable $\leftarrow$ True
12: **end if**
13: **end for**
14: **if** extendable = False **then**
15: $maxSub(q) \leftarrow maxSub(q) \cup sg_c$
16: **end if**

from the extension label of sg'_c as introduced in Sect. 3.2.1). In $M(sg_c, q)$, vertex $v_i(sg_c)$ maps to $v_j(q)$. Given that vertex $v_i(sg'_c)$ maps to $v_i(sg_c)$ in sg_c and $v_i(sg_c)$ maps to $v_j(q)$ in the query graph, the vertex $v_i(sg'_c)$ maps to $v_j(q)$.

The partial mappings $pM(sg'_c, q)$ can be constructed at query-time. For each $pM(sg'_c, q)$, the algorithm, $maxSub-Search$, tries to expand it to a complete mapping $M(sg'_c, q)$ (line 5–7). If the partial mapping can be expanded, the algorithm continues to search for mappings from sg'_c's children to q. If a lattice node sg'_c is not contained in q, then none of sg'_c's descendants can be subgraphs of q and are thus not checked. Expanding existing partial mappings $pM(sg'_c, q)$ utilizing prior information available in $M(sg_c, q)$ is significantly cheaper than checking for the isomorphism between sg'_c and q afresh. Thus, our method requires significantly less time than prior methods.

Subgraph isomorphism test versus embedding searching: A potential concern with our maximal-subgraph-search algorithm may be the difference between the time complexity of detecting subgraph isomorphism of two graphs and that of finding all mappings between the two graphs. In the traditional method, such as in FG-index, it is sufficient to test whether there is one mapping between the sg_t and q if $sg_t \subset q$. But in the Lindex search, we need to find all the mappings between sg_t and the query. However, using Lindex is always faster than using other indexes, for example, FG-index because they still need to find all the mappings. Given two features sg_t and $sg_{t'}$, $sg_t \subset sg_{t'}$, $sg_t \subset q$, and $sg_{t'} \not\subset q$. Assume that there are n mappings in $M(sg_t, q)$; there are n corresponding partial mappings $pM(sg_{t'}, q)$ where $sg_{t'} \supset sg_t$. In order to find the complete set of maximal subgraphs of the query q, FG-index will eventually verify whether $sg'_t \subseteq q$ holds. FG-index exhaustively searches each partial mapping between $sg_{t'}$ and q to show that none of them can be extended to a full mapping. Therefore, although in the isomorphism test between sg_t and q, the traditional method, such as FG-index, does not enumerate all the mappings between sg_t and q, but in the isomorphism test between $sg_{t'}$ and q, it enumerates all those n mappings and extends them.

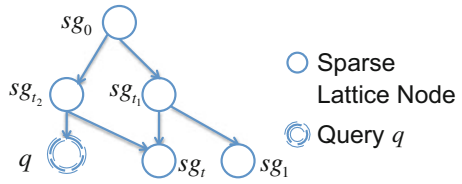


Fig. 8 Example of spanning-tree construction

Memory consumption: In the algorithm described above, we adopt a breadth-first approach and store all the mappings from one feature sg_t to the query q, $M(sg_t, q)$. However, in cases where memory is limited, the traversal can be easily changed to depth-first in which mappings are enumerated and extended one after another.

As with gIndex, maximal subgraphs obtained by the above algorithm are a super-set of $maxSub(q)$ for a given query q because there could be a scenario where sg_1 and sg_2 are identified as the maximal subgraphs of a query q even though there is a path from sg_1 to sg_2 in the sparse lattice $\mathcal{SL}(F)$ because the path was not included in the spanning tree T. The output of this lattice traversal using the spanning tree related to the lattice is a set $CMB(q)$ of Candidate Maximal suBgraphs of a query graph q. In the Sect. 5.2, we show how we can generate the set of minimal supergraphs of q and also prune $CMB(q)$ to obtain $maxSub(q)$.

5.1.3 Spanning-tree construction

Constructing the spanning tree (introduced above) involves choosing which parent's label to extend while labeling each index feature with an eye toward minimizing the search time for finding the maximal-subgraph features of queries. Consider a query q. Assume there is a node sg_t in the sparse lattice in Fig. 8, $sg_t \not\subset q$, which has two parents sg_{t_1} and sg_{t_2} where $sg_{t_1} \not\subset q$ and $sg_{t_2} \subset q$.

If we choose sg_{t_2} as the parent of sg_t in the spanning tree, then $sg_t \not\subset q$ will not be detected until we expand the

mappings $M(sg_{t_2}, q)$ to try to find $M(sg_t, q)$; whereas if we choose sg_{t_1}, we can know that $sg_t \not\subset q$ without expanding $M(sg_{t_1}, q)$ to $M(sg_t, q)$. In order to save search time, we have to identify a *mapping extension failure* as soon as possible, but that depends on the query graph. Intuitively, a reasonable heuristic choice is to choose the parent with the least chance of appearance in a query workload. The least popular parent will provide early pruning for most queries. However, because we do not have a real-world query workload, we use the database as a surrogate. Our algorithm uses the following heuristic:

If $freq(sg_{t_1}) < freq(sg_{t_2})$*, then Probability* $(sg_{t_1} \not\subset q) >$ *Probability* $(sg_{t_2} \not\subset q)$.

We follow this heuristic to construct the spanning tree: for each node, from its list of parents, we choose its extension labeling parent sg_t, as the parent with minimum frequency in the database.

5.2 Minimal-supergraph search

5.2.1 Previous work comparison

To the best of our knowledge, the only graph index that supports supergraph-feature search is FG-index. In FG-index, an edge index is built to facilitate the index lookup, as shown in Fig. 4. Given a size-k query containing distinct edges e_1 and e_3, the supergraph-feature candidates, $\{g_5, g_{10}\}$, are obtained by intersecting the features (IDs) containing e_1 and e_3. Then, each candidate feature, from small to large, is verified with subgraph isomorphism tests, until the closest δ-TCFG supergraph is found [5]. However, as indicated in FG*-index, the candidate set obtained by the edge index is generally large due to the fact that distinct edges cannot fully record the structural information of graphs [4]. To further reduce the index-lookup time, FG*-index uses an additional inverted index to index the δ-TCFG features. The keys of this inverted index are all subgraphs of δ-TCFG features with edge counts between two and four [4]. Although effective, extra memory is required to store this additional feature-inverted index (both keys and value sets). However, as we show below, Lindex can support the minimal-supergraph-feature search without any additional index structure. Also, candidate sets for supergraph features are generally much smaller than that of FG-index because substructure features (instead of distinct edges) lead to tighter candidate sets.

5.2.2 Minimal-supergraph search

We introduce the algorithm to find the minimal supergraphs $minSup(q, F)$ and the exact set of maximal subgraphs $maxSub(q, F)$ of the query q in this subsection. The algorithm is based on the following observation:

Property 3 *The set of minimal supergraphs of a query q is a subset of the intersection of the set of descendants of each maximal-subgraph feature of q in the sparse lattice, $\mathcal{SL}(F)$.* $minSup(q, F) \subseteq \cap_{sg \in maxSub(q,F)} Descendant(sg)$.

This property evolves from the following simple observation: If a graph in the sparse lattice is a supergraph of the query q, it must contain each of q's subgraphs in the lattice as subgraphs of its own. Our algorithm starts from $CMB(q)$ generated as described in Sect. 5.1. For each feature h in $CMB(q)$, the algorithm finds the descendants of h in the sparse lattice. Then, the algorithm finds the Candidate Minimal-suPergraph set, $CMP(q)$, for $minSup(q)$ as the intersection of these descendant sets. The set $CMP(q)$ is then sorted based on the sizes of its element graphs. For each graph h in $CMP(q)$, a subgraph isomorphism test is performed to determine whether it is a supergraph of the query q. Once h is a supergraph of q, all descendants of h can be removed from $CMP(q)$ because here we are only interested in the set of *minimal* supergraphs of q. The removal of h's descendants from $CMP(q)$ further reduces the search space.

Algorithm 4 minSup-search

Input: Sparse lattice $\mathcal{SL}(F)$, Query Graph q,
Candidate maximal-subgraph Set $CMB(q)$
Output: $minSup(q)$

1: sort $CMB(q)$ according to their graph keys' size in descending order
2: candidate Set $C \leftarrow \emptyset$
3: $Des(f_i) \leftarrow$ find descendants of f_i in $\mathcal{SL}$, $f_i \in CMB(q)$
4: **if** any of f_i's descendants $= f_j$, where $j < i$ **then**
5: $CMB(q) \leftarrow CMB(q) - f_i$
6: **end if**
7: $CMP(q) \leftarrow \cap_{f_i \in CMB(q)} Des(f_i)$
8: sort candidate $CMP(q)$ with regard to their graph keys' size in ascending order
9: **for** each graph $h \in CMP(q)$ **do**
10: **if** $h \supset q$ **then**
11: $minSup(q) \leftarrow minSup(q) + h$
12: remove all h's descendants from $CMP(q)$
13: **end if**
14: **end for**

Our algorithm to find minimal supergraphs of a query q also helps us prune $CMB(q)$ (as generated by the subroutine described in Sect. 5.1) to obtain $maxSub(q)$. As remarked above, if an edge between subgraph nodes sg_1 and sg_2 appears in lattice $\mathcal{SL}(F)$, but not its spanning tree, our algorithm in Sect. 5.1 could potentially choose both sg_1 and sg_2 in $CMB(q)$. While determining $minSup(q)$, the algorithm would prune sg_1 out. As indicated above, our algorithm constructs the set of proper descendants $Des(sg_i)$ for each element sg_i of $CMB(q)$ while constructing $minSup(q)$. We can check whether a $sg_j \in CMB(q)$ is included in $Des(sg_i)$ for any $i \neq j$. If $\exists i$ such that $sg_j \in Des(sg_i)$, then sg_i should be removed from the $CMB(q)$, because sg_i is not a *maximal* subgraph of q (for it is contained in sg_j).

6 Lindex framework

6.1 Answer subgraph querying with Lindex+

Based on the properties introduced in previous sections, Lindex takes the following steps to process subgraph queries, as specified in Algorithm 5. First, the algorithm searches for the set of maximal-subgraph features of q in Lindex. If q is a key in Lindex L, all graphs that appear in the value set (both direct and indirect value sets) of the feature $f = q$ in L are returned. Otherwise, we obtain the *maximum* subgraph feature of q, $F_k(q)$ (recall the difference between maximal and maximum). Then, load the on-disk Lindex+, L', rooted from $F_k(q)$ into memory. We continue searching for q on L'. If q is a key in L', then the value set of q is returned as the answer directly.

Algorithm 5 Graph Querying Using Lindex+

Input: Lindex $L(D, F)$, Graph Database D, Query q
Output: Answer set An
Procedure:
1: CandidateSet $C \leftarrow \emptyset$ (Graphs need to be verified)
2: AnswerSet $An \leftarrow \emptyset$
3: Graphs $F \leftarrow$maxSub-search$(\mathcal{SL}(F), q)$
4: **if** $\exists f \in F and f = q$ **then**
5: return $An \leftarrow V(f)$
6: **else**
7: choose maximum subgraph $f_k \in F$
8: load outer memory Lindex rooted from f_k, $L'(D, F')$
9: Graphs $F' \leftarrow$maxSub-search$(\mathcal{SL}(F'), q)$
10: **if** $\exists f' \in F' and f' = q$ **then**
11: return $An \leftarrow V(f')$
12: **end if**
13: **end if**
14: $C \leftarrow \cap_{f \in F} V_d(f)$
15: Graphs $H \leftarrow$minSup-search$(\mathcal{SL}(F), q, F)$;
16: $C \leftarrow C - \cup_{h \in H} V(h)$
17: $An \leftarrow \cup_{h \in H} V(h)$
18: Add all subgraphs $g \in C$ that contain the query q to An
19: return An

Otherwise, the algorithm falls back to "filter+verify" and constructs a set of candidate graphs $C(q)$ that are in the direct value set of all the maximal-subgraph features of q in L. Then, the algorithm finds the set of minimal-supergraph features of q. The database graphs that are indexed by any minimal-supergraph features are removed from $C(q)$ and are added to the answer set $Tr = \cup_{h \in minSup(q)} V(h)$. Next, the algorithm checks that each candidate graph in $C(q)$ contains q using a subgraph isomorphism test; false positives that are not supergraphs of the query q are pruned. Let $C(q)'$ be the set of graphs that passed the subgraph isomorphism test. The final answer set is the union of $C(q)'$ and Tr.

6.2 Time complexity

The complexity of the subgraph isomorphism tests typically dominates the time complexity for the rest of the query processing including traversing the lattice and the rest of the algorithm. There are three places in our algorithms where we perform subgraph isomorphism tests: (1) In maximal-subgraph search, partial mappings are expanded as the lattice is traversed. These are "partial" isomorphism tests, and the exact number of expansions of the mappings needed is difficult to quantify because it is dependent upon the data. Their cost has to be estimated empirically. (2) In minimal-supergraph search, Algorithm 4, line 10. Here, the number of tests done is equal to $|CMP(q)|$. (3) In the main algorithm of subgraph-querying processing, Algorithm 5, line 18. Here, the number of tests done is equal to $|C|$. The majority of the time savings accrues from following: *We save on the number of subgraph isomorphisms when the additional isomorphism tests performed by our algorithm in Algorithm 4 and Algorithm 3 substantially cut down the number of isomorphisms performed in Algorithm 5*.

Time complexity for maxSub-search: The time complexity of the *maxSub-search* algorithm contains the time taken for the following steps: (1) find all mappings from the children of the root to the query. Since for features mined by most feature-selection algorithms, the root's children are distinct edges, the complexity of this step equals to that of counting distinct edges in the query; (2) time taken to expand the mappings for each descendant (of the root's children) until the maximal subgraphs are obtained. This is proportional to the number of nodes traversed in the lattice.

Time complexity for minSup-search: Let us consider the time required for the minSup-search algorithm. The minSup-search algorithm has the following operations: (1) sort $CMB(q)$ and $CMP(q)$. These operations are $O(n\log(n))$ where n is the size of the lists. (2) Find the set of descendants of the maximal subgraphs of q in the Lindex (line 3). This operation is linear in the number of descendants of the maximal subgraphs in the Lindex. (3) Prune the descendants of minimal-supergraph features in this list (line 12). These operations can be done at most in quadratic time with respect to the number of descendants. (4) Check for subgraph isomorphism (line 10). The number of isomorphism tests run depends on the size of $CMP(q)$. We denote the time taken by the first three items as $T_{traverse}$ (the time taken to traverse and process the partial lattice), and the cost of each isomorphism test is T_{iso}. The time taken by minSup-search can be estimated as: $T_{minSup-Search} = |\text{maxSub(q)}| \times T_{traverse} + |CMP(q)| \times T_{iso}$.

Overall query processing complexity: Thus, the response time of the overall query processing can be expressed as:

$$
\begin{aligned}
T_{response} &= T_{filtering} + T_{verification} \\
&= T_{search} + T_{intersection} + T_{verification} \\
&= (T_{maxSub} + T_{\cap}) + (T_{minSup} + T_{\cup}) + |C(q)| \times T_{iso},
\end{aligned} \tag{5}
$$

where $|C(q)|$ is the number of candidate graphs need to be verified (line 18, Algorithm 5), $T_{\cap}$ ($T_{\cup}$) is the cost of intersection (union) operations on value sets and depends on the size of answer sets of *maxSub-search* (*minSup-search*). $T_{\cap}(T_{\cup})$ also includes time spending on loading value set from disk to primary memory.

$$T_{\cap\cup} \propto |maxSub(q)| + |minSup(q)|.$$

Compared with previous work [5,19], $T_{minSup(q)} + T_{\cup}$ are the newly added costs. The time saved on the $T_{verification}$ largely outweighs the additional time taken to compute the minimal supergraphs of the query especially when query q is a non-FG query. In addition, our *maxSub-search* algorithm outperforms traditional algorithms. Furthermore, $T_{\cap}$ is less than that in other algorithms, because we can find the precise set of maximal subgraphs of the query and consequently save time because our algorithm loads less values sets from disk (especially useful for large databases).

6.3 Updating Lindex

In this subsection, we briefly discuss how Lindex can be updated. When a graph g is deleted from the database, we walk down the lattice as we do with a query g and remove g from all the value sets of all its subgraphs. When a graph g is added to the database, we do a similar walk except that we need to determine whether g needs to be in the direct value set or the indirect value set. The costs incurred for deletion are similar to that in gIndex and FG-index; insertions in our index have an extra cost of checking direct/indirect, but that is offset because maxSub-search in Lindex is much faster due to its strategy of expanding mappings of parent nodes to obtain mappings to child nodes.

When the optimal set of features that should be indexed changes, the feature-selection algorithms, for example, gIndex, FG-index, MimR, must be re-run. Incremental feature-selection/update algorithms need to be designed in the future. In practice, the set of optimal features changes very slowly because the semantics of the data or user access patterns do not change overnight, say in chemical databases; however, in scenarios where the features change and incremental feature update algorithms are designed, we can design Lindex to use those features and feature update algorithms. Note that the feature-selection part of the update algorithm dominates the index-construction time for gIndex, FG-index, MimR, and Lindex and is orders of magnitude greater than the time taken to create the index once the features are obtained for all of these methods.

Index\Feature	DFG	ΔTCFG	MimR	Tree+Δ	DFT
Gindex	*		*	*	*
FG-index		*			
Lindex	*		*	*	*
Lindex+		*	*		
SwiftIndex					*

Fig. 9 Index structures and features studied

7 Experimental evaluation

In this section, we compare Lindex with gIndex [19], FG-index [5], Tree+δ [23], SwiftIndex [13], and MimR [15]. Other indexing methods, not based on feature-based indices, such as CTree [10], GDIndex [17], and GCode [24], do not perform well in the general case [9]. The comparison between feature-based indices and non-feature-based indices is out of the scope of this paper. Interested readers can refer to iGraph [9].

For Lindex+, the on-disk subgraph features are only used for answering the query directly if the query is indexed. Fig. 9 lists all the indices and features we compare in this paper.

The asterisks show that certain index structures (rows) are evaluated with certain features (columns). For example, Lindex+ is evaluated jointly with δ-TCFG features, and we name it as Lindex+(δ-TCFG). DFG/T refers to discriminative and frequent subgraph/subtree features [13,19,23].

We set the default parameters suggested by the original works unless we specified otherwise. For DFG and DFT, the default max-min support is $0.1|D|$ (D is the graph dataset), $maxL = 10$, discriminative ratio $\gamma = 2$ [13,19]. For Tree+δ, the default parameters are set as follows: the minimum discriminative ratio $\epsilon_0 = 0.1$, max-min support is $0.1|D|$, and $\sigma*$ is 0.8 for discriminative graph feature selection during query processing. Since the DFG/T and subtree features in Tree+δ adopt a size-increasing min-support strategy, the actual min-support for a size-L feature is $\sqrt{L/maxL}$ $\times$ (max-min support). For example, for a 4-edge feature, its actual min-support is $\sqrt{4/10} \times 0.1 = 0.063$. For δ-TCFG, we use minimum support = $0.03|D|$ and $\delta = 0.1$. To make the FG-index indexing comparable number of in-memory features as DFG/T and Tree+δ, the minimum support is set to be $0.03|D|$ instead of $0.1|D|$ or $0.01|D|$ (suggested by the original work). This slight modification of the setting is fair, since (1) the number of features for DFG/T, MimR, Tree+δ, and δ-TCFG are comparable. (2) Lindex (or Lindex+) is compared with other index structures using the same features. For example, no matter how many δ-TCFG features

are selected for indexing, Lindex (or Lindex+) and FG-index use the same δ-TCFG feature set. We also compare the performance of Lindex and other indices with varying number of features mined with varying minimum support.

Storing value sets in memory is not scalable for large datasets. For Lindex(+) and FG-index, to make the memory consumption comparable, we store the value sets of gIndex on disk. As in related works [5,15,19,21,23,13], we do not have access to a real-world query workload. Thus, we used a randomly generated query set to train the MimR algorithm as suggested by Sun et al. [15]. To test the effectiveness of different indices, first we generate a series of query sets Q_4 to Q_{24}, where the subscript denotes the number of edges in each query in the query set. Each query set contains 200 queries generated from the graph database. The queries are generated as follows. First, we choose 1,000 graphs out of the database at random and find all of their subgraphs without eliminating duplicates. Then, we sample subgraphs from this lot according to a uniform distribution or a normal distribution. Thus, frequent subgraphs have a higher chance to be selected. Where not explicitly specified, experimental evaluations are on queries sampled using a normal distribution.

Datasets: We evaluate the performance of our index on the AIDS Antiviral Screen dataset consisting of 43,905 chemical structures [5,13,15,19,23]. We also use a second dataset obtained from emolecules.com, which we will refer to as the eMolecules dataset to test the scalability of our algorithm. To evaluate the performance of Lindex over graphs with varying density, we use synthesized graph data generated with GraphGen [5]. The density of a graph g with nodes V and edges E is defined as the ratio $\frac{|E|}{|V|\times|V|}$. We generate five graph sets (each containing 10,000 graphs) with average density 0.1, 0.2, to 0.5.

We implement our algorithms with Java and use the Lucene library[4] to build the value sets. We implemented gIndex, FG-index, SwiftIndex, Tree+δ indices based on the same framework. The feature mining tasks were run on machines with 32G RAM, and the indexing algorithms were tested with a maximum heap size of 2G.

7.1 Experiment on the AIDS dataset

7.1.1 Memory consumption

Table 3 shows the number of selected features from the AIDS dataset and their memory cost. The second line lists the count of various features mined from the graph dataset. Since gIndex(DFG/T) stores both black nodes (for filtering) and white nodes in memory [19], the count of DFG/T features has two numbers, the first one is the total number of features in memory and the second number is the black nodes count. Lindex only stores the black nodes. For the δ-TCFG features, the first number is the total number of features (both in-memory and on-disk) and the second one is the count of δ-TCFG features. For Tree+δ features, the first number is the count of subtree features and the second number is the count of δ features. Lines 3–7 show the memory consumption of gIndex, FG-index, Lindex, and SwiftIndex on various features. It shows that Lindex is the most compact index with respect to memory consumption.

Table 3 Index memory cost: AIDS dataset (KB)

Mem	DFG	δ-TCFG	MimR	Tree+δ	DFT
Feature count	7599 6328	9873 5712	5000	6172 38	7500 6172
gIndex	1,359		1,534	1,348	1,339
FG-index		1,826			
Lindex	677		772	676	671
Lindex+		841	776		
SwiftIndex					860

7.1.2 Query processing time

Figures 10 and 11 show the real query processing times of the different indices. The filtering time includes index-lookup time (finding $maxSub(q, F)$), value-set fetching time (from disk), and value-set join or union operation time. The verification time is the time spent on subgraph isomorphism tests in the verification step. Lindex(+) using the same set of features has less candidate verification time due to its high filtering power. Also, Lindex uses less filtering time in comparison with other indices since the *maximal-subgraph and minimal-supergraph* search over Lindex is fast.

Figure 10a shows that Lindex outperforms gIndex with the same set of discriminative and frequent features (DFG). When the query is small (4–7 edges), Lindex saves verification time. When the query is large, Lindex saves filtering time. Similar results are observed in Fig. 10b too. The SwiftIndex outperforms gIndex on filtering time but is still not as good as Lindex. Also, SwiftIndex does not improve the filtering power as Lindex does on small queries. The filtering time of gIndex (DFG/T) takes a small portion of the overall response time because a frequency-based apriori rule can be used to prune the maximal-subgraph search space early [19]. But for the δ features in Tree+δ and MimR features, this apriori rule cannot be applied and the maximal-subgraph search time exceeds the verification time by several times when the query is large, (see Fig. 11a, b). By using Lindex on MimR features (and δ features in Tree+δ), the filtering time decreases significantly, thus reducing the overall response time. Figure 10c shows that using the same δ-TCFG features, Lindex outperforms FG-index significantly on processing

[4] http://lucene.apache.org/.

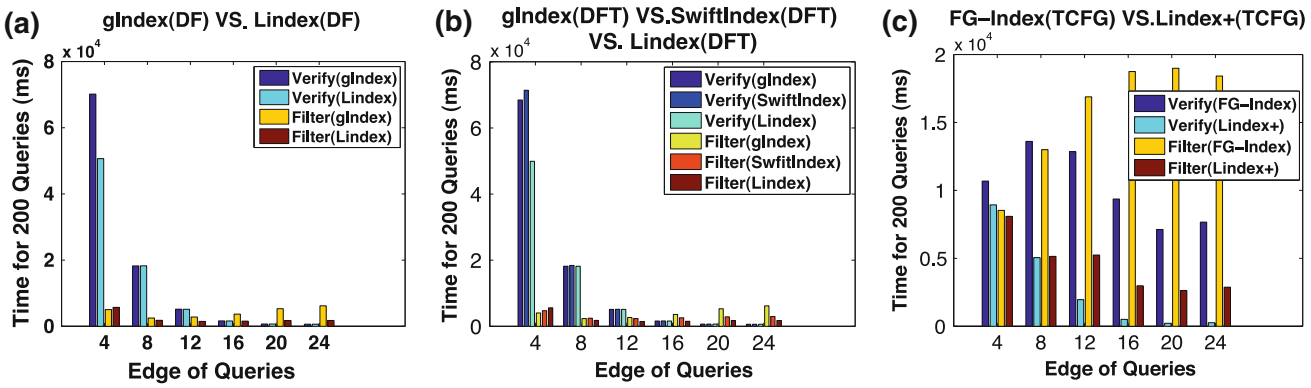
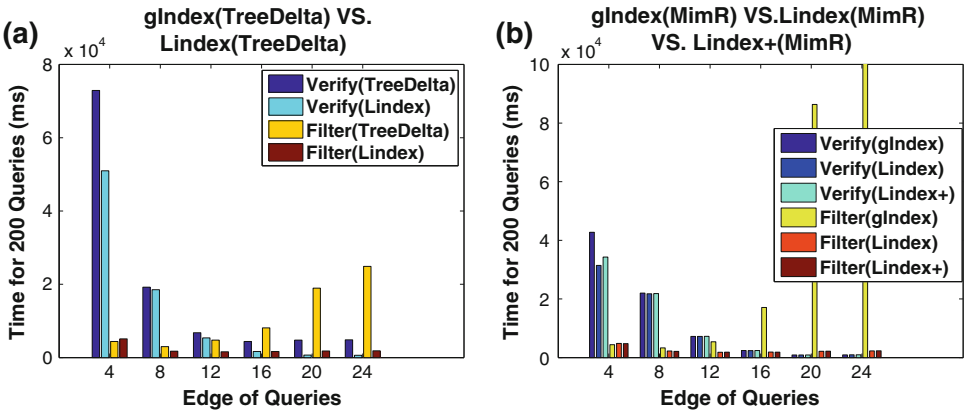


Fig. 10 Experiment on AIDS dataset: 40,000 graphs (1)

Fig. 11 Experiment on AIDS dataset: 40,000 graphs (2)



large queries. In FG-index, in order to control the filtering time, an incomplete $maxSub(q, F)$ is returned [5]. As we can see, this incomplete *maxSub* set decreases the overall filtering power, thus enlarging the cost of isomorphism tests. Also, when the query is large, the time of the incomplete maximal-subgraph search still takes six times as that in Lindex+.

From the above experiment, we observe that MimR features and δ-TCFG features can filter more false graphs in comparison with discriminative and frequent subgraphs (note that the different kinds of features have comparable counts); however, their filtering time is very high. By using Lindex, the filtering time reduces significantly and this reduction allows us to index δ-TCFG and MimR features. Additionally, Lindex also helps improve the filtering power by adopting two advanced query processing strategies (see Sect. 4), especially for small queries.

7.1.3 Index-construction time

In this subsection, we report the construction time for Lindex. As discussed in Sects. 3.2.2 and 4.4, the construction of a graph index consists of two sub routines that construct the index and populate the value sets. The index structure, for example, sparse lattice $\mathcal{SL}(\mathcal{F})$ for Lindex, hash table for gIndex, edge-inverted index for FG-index, is first constructed given the features mined by the feature miner. As shown in Table 4, the index-structure construction time is low for most indices, except for FG-index, in which an edge-inverted index of features is built. The FG-index (δ-TCFG) row shows the time taken to build an in-memory FG-index (IGI), and the FG-index (FG) row shows the time taken to build and save the on-disk part of FG-index. Construction of the gIndex is fast because it is just a hash table storing all canonical labels of the features. Lindex lattice construction runs faster than gIndex because Lindex only stores the black nodes (gIndex stores both black and white nodes). Lindex construction is slower than SwfitIndex construction due to the spanning-tree construction (Sect. 5.1.3) and extension labeling (Sect. 3.2.1). Our observations of the index-structure construction times are consistent with that of previous works [5,13,19].

After the index structure is created, the value sets are constructed. Due to the large size of value sets for large databases, the value sets are commonly stored on disk. The original gIndex stores value sets in memory [19], which makes gIndex not scalable to large datasets as reported by Cheng et al. [5]. We store value sets on disk for all graph indexing methods. We use the Lucene library to implement the value sets. The results of our experiments are Lucene-dependent, but it is fair since all of the studied indices are implemented with the same framework.

Table 4 Index-structure construction time (ms)

Index name (features)	Original index	Lindex
gIndex (DFG)	518	202
gIndex (DFT)	151	104
SwiftIndex (DFT)	84	104
FG-index (δ-TCFG)	4,169	241
FG-index (FG)	867,926	124,224
gIndex (MimR)	381	338

Table 5 Value sets construction time (s)

Index name (features)	Original index	Lindex
gIndex (DFG)	1,516	454
gIndex (DFT)	1,510	450
SwiftIndex (DFT)	198	450
FG-index (δ-TCFG)	5,784	2,333
FG-index (FG)	3,875	199

Implementing different indices with the same framework is important in performance evaluation and comparison, as shown in iGraph [9]. In addition, the coupling (dependence) between the index structure (or query processing model) and the Lucene-implemented value sets is low, so that other value-set implementations, for example, iGraph [9], can also be adopted. By using the Lucene library, we build the value sets by inserting database graphs. For each database graph g, we first lookup up the index structure to find features f contained in g and then append g's ID at the end of f's value sets. As we argued, this value-set construction strategy is suitable for large datasets when the graph miner cannot return $D(f)$ for each feature f. Table 5 shows the time taken to construct the value sets for gIndex, FG-index, SwfitIndex, and Lindex. Lindex, despite spending time partitioning value sets, outperforms gIndex and FG-index due to its fast subgraph-feature lookup (Sect. 5.1). SwiftIndex runs faster than Lindex on constructing value sets since its index lookup is comparable to Lindex and it does not partition the value sets. As can be seen from Table 5, the value-set construction time is high for all indices and it is comparable to the time taken to mine subgraph features. However, unlike frequent subgraph mining algorithms, value-set construction can run on distributed systems. For example, the graph database can be segmented and stored on distributed file systems. Multiple machines can build the value sets for one or several segments of the graph database. Then, the value sets can be merged. Lucene library does support the merging of value sets. Detailed discussions for distributed and time-efficient value-set construction are beyond the scope of this paper, and we would like to study this problem in our future work.

Index maintenance: From the above experimental results, we can also observe the advantage of Lindex on index maintenance over a dynamically changing graph database. If after updating, the graphs in the updated database and the original database graphs have similar characteristics, the features do not need to be changed [19]. The index structure is stable but the value sets need to be updated. Therefore, inserting (deleting) a graph g makes no other changes except for appending (deleting) g's ID to (from) f's value set, $\forall f \subset g$. As in Table 5, Lindex is fast with respect to inserting graphs in the database, which implies that Lindex can support fast updates. When the updated graphs and original graphs do not have similar characteristics, features need to be re-mined and both the index structure and value set should be re-built, in which case, Lindex is still faster to construct in comparison with other indices.

7.2 Indexing large datasets

In this subsection, we test Lindex on large-scale datasets.

Dataset and setting: We construct five datasets randomly sampled from the eMolecules dataset. They contain 65,536 (2^{16}), …, 1,048,576 (2^{20}) graphs. They are denoted as $DB(16)$ to $DB(20)$. The largest dataset we use, $DB(20)$, is 10 times larger than the largest dataset used in previous works (FG-index was tested on 100,000 graphs [4]). We observe linear growth with respect to query processing times and

Table 6 Index memory cost: eMolecules dataset (KB)

Data	Count		Mem			
	DFG	TCFG	gIndex	Lindex (DFG)	FG-index	Lindex+ (TCFG)
DB (16)	3,512	4,887 2,038	678	399	740	332
DB (17)	4,465	4,898 2,036	833	506	839	340
DB (18)	5,495	4,922 2,046	1,016	622	1,002	349
DB (19)	6,906	4,925 2,046	1,299	780	1,263	358
DB (20)	8,618	4,931 2,051	1,601	972	1,704	374

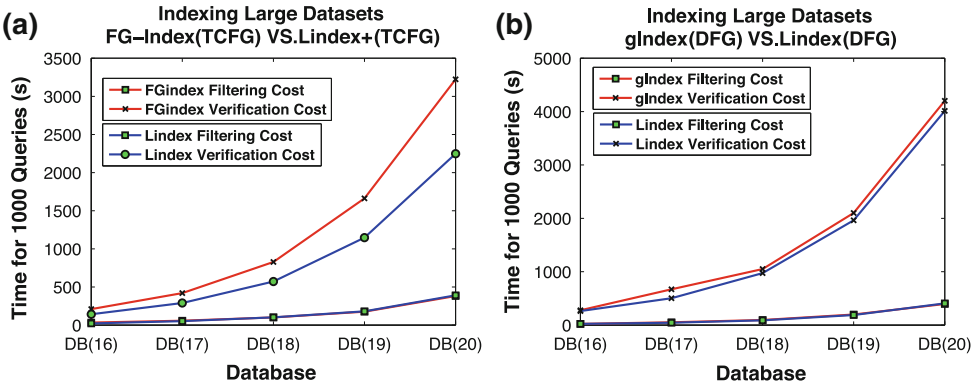


Fig. 12 Experiment on eMolecules: large-scale datasets

index-construction times of Lindex as we show in this subsection. We adopt the same parameter settings for the AIDS dataset. Since the MimR feature miner is not scalable to large datasets, we do not compare it in this section. Also, as can be observed on the AIDS dataset, the improvement of Lindex over DFT features is similar to that on DFG features, so we skip the study of DFT features.

Memory consumption: The experimental results on memory consumption are given in Table 6. The two columns on the left show the number of DFG and δ-TCFG features mined from the five graph datasets. As before, the first number of the TCFG column is the total number of frequent subgraphs and the second number is the δ-TCFG count. The table shows that the δ-TCFG features are more stable compared to the DFG features. As we further investigate the features, we observe that most of the new DFG features added (with the growth of the database) are not frequent subgraphs but features with edges < 4 (gIndex select all less-than-4-edge subgraphs). The frequent subgraphs mined by both DFG and δ-TCFG algorithms do not change much when the database grows. This further shows that the five sets of database graphs have similar characteristics.

The four columns on the right show the memory consumption of gIndex, FG-index, and Lindex(+). This result is consistent with the result observed on the AIDS dataset. Lindex(+) takes less memory than gIndex and FG-index while indexing the same features.

Query Processing Time: Figure 12a shows the query processing time of Lindex and FG-index over 1,000 queries (uniformly sampled). Lindex+(δ-TCFG) is faster than FG-index on all graph datasets. In addition, the filtering time and verification time are both linear to the growth of the graph database. We further observe that the increase in the filtering time is mainly due to the value-set fetching and join operations, of which the time complexity is $O(|D|)$, where D is the graph database. The index-lookup time does not change much. For this experimental setting, the value-set fetching time dominates the overall filtering time. Thus, the lines for the filtering time of Lindex and FG-index overlap in the figure. As we will see in the next subsection, when more features are indexed, the overall filtering time is dominated by the index-lookup time. Lindex outperforms FG-index significantly with respect to the filtering time in that case.

Table 7 Index-structure construction time (ms)

Data	FG-index	Lindex+(TCFG)
DB(16)	3,299	1,720
DB(17)	5,765	1,684
DB(18)	10,322	2,039
DB(19)	19,370	2,651
DB(20)	38,230	2,611

Similar results can be observed on DFG features, as shown in Fig. 12b. The results shown above demonstrate the scalability of Lindex: the query processing time of Lindex is linear to $|D|$, and the improvement of Lindex over gIndex (FG-index) is also linear to $|D|$ (In Fig. 12a, b, the database size $|DB(i+1)| = 2 * |DB(i)|$; hence, exponential curves are shown in the Figure).

Construction time: As in Table 4, the time taken to construct gIndex and Lindex is low. Table 7 shows the index-construction time for FG-index and Lindex+. Although the δ-TCFG and FG features are stable, the index-construction time for FG-index (in-memory + on-disk) grows linearly with the size of the graph database. This increase is because an in-memory edge-inverted index is built for infrequent edges [5]. For each infrequent edge e, the value set of e contains all database graphs having edge e. To construct this infrequent edge index, we have to scan the whole graph database. Hence, the construction time increases when the graph database doubles its size.

For dataset $DB(16)$, the time taken to construct the *value sets* are: gIndex = 442 s, Lindex(DFG) = 141 s, FG-index = 5,388 s, and Lindex+(TCFG) = 272 s. In addition, the value-set construction time increases linearly with the growth of the database. Also, as before, the value-set construction time is comparable to the frequent subgraph feature mining time. For example, for the graph database $DB(20)$, the frequent feature mining time for DFG features is 2,011 s, and the value-set

Table 8 Index memory cost: DB(17), (KB)

Min–max support	0.1	0.06	0.03	0.02
Feature count	4,465	4,559	4,847	5,111
gIndex	833	877	1,011	1,135
Lindex (DFG)	506	526	586	639
Min support	0.03	0.02	0.01	
Feature count	4,898 2,036	8,948 3,808	24,171 10,744	
FG-index	839	1,284	3,133	
Lindex+ (TCFG)	340	646	1,817	

Table 9 Index-construction time: DB(17), (ms)

Min–max support	0.1	0.06	0.03	0.02
gIndex	267	272	290	304
Lindex (DFG)	124	150	169	182
Min support	0.03	0.02	0.01	
FG-index	5,765	27,625	77,227	
Lindex+(TCFG)	1,684	7,609	21,591	

construction time of Lindex(DFG) is 2,451 s. The value-set construction time, unlike the frequent feature mining time, also includes the time for writing the value sets to disk.

7.3 Indexing large feature sets

In this subsection, we study how Lindex performs with a large number of indexing features.

Dataset and setting: We use a middle-sized graph dataset, $DB(17)$, containing 131,072 graphs. This dataset is larger than the 100,000-graph dataset—the largest dataset used in previous work [5]. We run the DFG feature miner with max-min support 0.1, 0.06, 0.03, and 0.02 and construct gIndex and Lindex with those features. We also run the δ-TCFG feature miner with minimum support 0.03, 0.02, and 0.01 and construct FG-index and Lindex+ with selected features. (The definition of max-min support for DFG is different from the minimum support for δ-TCFG as explained before).

Memory consumption: In Table 8, rows 2 and 5 show the feature count of the DFG and δ-TCFG features. Decreasing the minimum support increases the number of selected DFG features slightly. However, the number of δ-TCFG features grows significantly. This growth is different from that observed in the last subsection, where the DFG features grow but the δ-TCFG features are stable when the graph database doubles its size. Rows 3, 4, 6, and 7 show the memory consumption of gIndex, Lindex, and FG-index correspondingly. It shows, as expected, that Lindex is more efficient in memory consumption than gIndex and FG-index.

Query processing times: Figure 13a shows the query processing times of 1,000 uniformly sampled queries with Lindex+(δ-TCFG) and FG-index. Decreasing the minimum supports increases the number of features indexed. Correspondingly, the index-lookup time for both Lindex and FG-index increases. However, with more features indexed, the verification time decreases due to the tighter candidate sets (more false answers are filtered). At some point, the filtering time (mostly index-lookup time) exceeds the verification time. The merit of Lindex is that the intersection point of filtering time and the verification time is lower than that of other indices. Also, as can be seen from Fig. 12a, the query processing time of Lindex is always lower than that of FG-index. Similar results can be seen on gIndex(DFG) and Lindex(DFG), (see Fig. 13b).

Construction time: Table 9 shows the index-structure construction time for gIndex, FG-index, and Lindex(+). Since the number of selected features for gIndex and Lindex(DFG) does not grow much when the min-max support decreases, the index-construction time does not increase much. However, for the δ-TCFG features, the index-structure construction time for both FG-index and Lindex+ grows significantly, due to the explosion of the number of features. In addition, as the table shows, the time taken to construct the index is linear with respect to the number of features indexed. Hence, Lindex can be scaled to handle large feature sets.

The value-set construction time for DFG features does not change much for both gIndex and Lindex (DFG) when the max-min support decreases. For example, the time taken to construct value sets for gIndex grows from 851 to 1,211 s

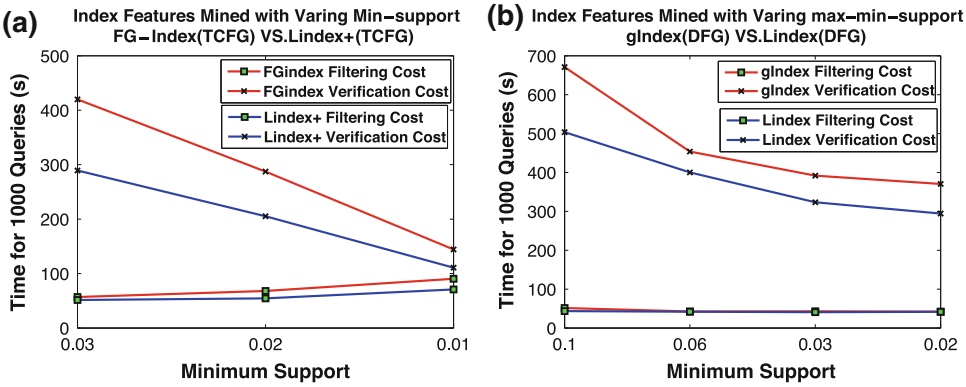


Fig. 13 Experiment on eMolecules: varying parameters

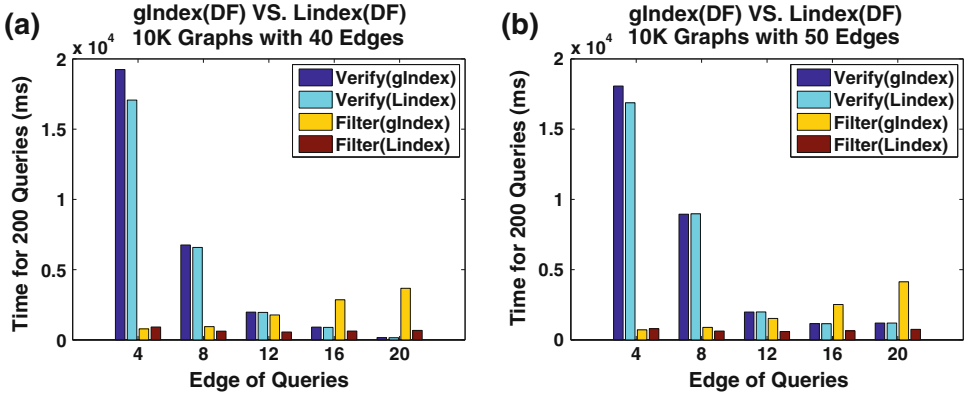


Fig. 14 Experiment on eMolecules: Lindex and gIndex on varying edge dataset

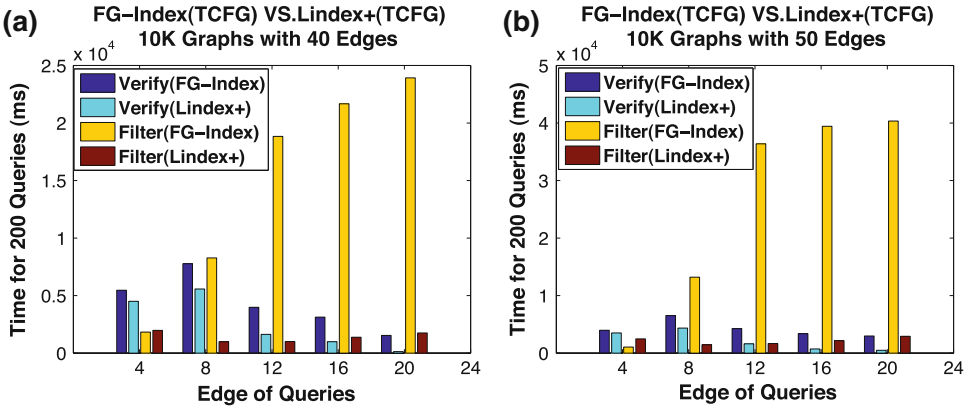


Fig. 15 Experiment on eMolecules: Lindex and FG-index on varying edge dataset

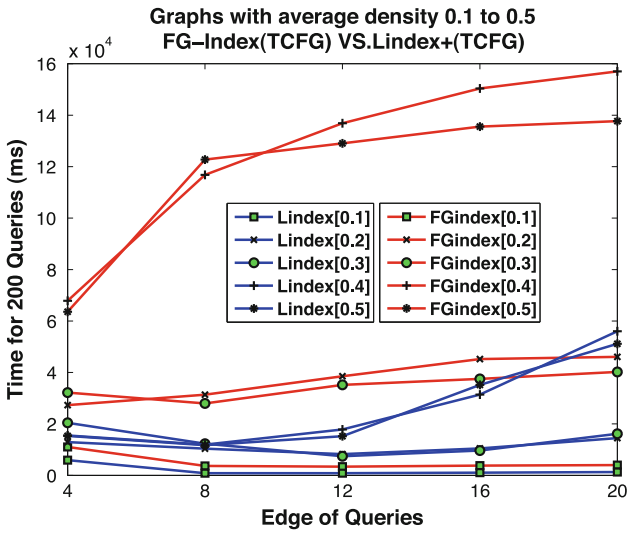


Fig. 16 Experiment on synthesized graphs: varying density

when the max-min support drops from 0.06 to 0.03; the time for Lindex (DFG) grows from 285 to 331 s. However, the time taken to construct value sets for δ-TCFG features grows dramatically. For FG-index, the time taken to construct value sets is 25,711 s when min-support= 0.02 and 65,599 s when min-support= 0.01. For Lindex+, the value-set construction time is 1,050 s when min-support = 0.02 and 1,993 s when min-support= 0.01. This increase is because the number of selected DFG feature changes slightly when max-min support decreases, but the selected δ-TCFG features and FG features (stored on-disk) grow significantly when the minimum support decreases. In addition, we also observe that the value-set construction time for Lindex is linear with respect to the number of features indexed.

7.4 Various graphs

Impact of graph size: In all the experiments above, the average edge count of graphs is 30. We also study the effectiveness of Lindex over graphs with 40 and 50 edges. We randomly selected 10,000 graphs from eMolecules dataset with average edge counts of 40 and 50, and ran the experiment. As shown in Figs. 14 and 15, the improvement of Lindex is consistent with previous experimental results.

Graph density: We also realized that the density (edge count to node count ratio) of the graphs we studied is generally low. In order to better study the performance of Lindex, we generate five synthesized graph datasets with

average density {0.1, 0.2, 0.3, 0.4, 0.5}. The above datasets were generated by the tool provided by Cheng, et al. [5].

We first mine DFG features for gIndex, and we observe all frequent subgraphs have less than 4 edges. Since all features with edge count less than 4 are selected as discriminative and frequent features as in gIndex [19]), in this dataset, all frequent subgraphs are selected. Therefore, the two strategies Lindex used to reduce the verification time lose their power. In this case, Lindex outperforms gIndex only on filtering time and the improvement is similar to that in Fig.10a. We further mine δ-TCFG features with $\delta = 0.8$. As in Figure 16, Lindex outperforms FG-index by a factor of 10 and this improvement is consistent when the graph density varies.

8 Conclusion

We proposed Lindex to process subgraph queries efficiently. Our query answering algorithm identifies a set of maximal subgraphs in the index and obtains a candidate set of answers by intersecting the direct value set of these subgraphs. In our algorithm, we prune this candidate set by identifying supergraphs of the query and eliminating graphs in the database that contain these supergraphs from the candidate set; our lattice-based index also makes efficient finding of supergraphs possible. Pruning the candidate set reduces the number of subgraph isomorphism tests required to answer graph queries. Consequently, we show that Lindex outperforms other existing methods, including gIndex, FG-index, MimR, Tree+δ, and QuickSI, on a query workload over an existing benchmark dataset.

References

1. Barnard, J.: Substructure searching methods: old and new. J. CIM **33**, 532–538 (1993)
2. Chen, C., Yan, X., Yu, P.S., Han, J., Zhang, D.Q., Gu, X.: Towards graph containment search and indexing. In: VLDB. VLDB Endowment (2007)
3. Cheng, J., Ke, Y., Fu, A.W.C., Yu, J.X.: Fast graph query processing with a low-cost index. VLDB J. **20**, 521–539 (2011)
4. Cheng, J., Ke, Y., Ng, W.: Efficient query processing on graph databases. ACM Trans. Database Syst. **34**, 2:1–2:48 (2009)
5. Cheng, J., Ke, Y., Ng, W., Lu, A.: Fg-index: Towards verification-free query processing on graph databases. In: SIGMOD (2007)
6. Conte, D., Foggia, P., Sansone, C., Vento, M.: Graph matching applications in pattern recognition and image processing. In: ICIP (2003)
7. Cook, S.A.: The complexity of theorem-proving procedures. In: STOC, pp. 151–158 (1971)
8. El-Mehalawi, M., Miller, R.A.: A database system of mechanical components based on geometric and topological similarity. J. CAD **35**(1), 83–94 (2003)
9. Han, W.S., Lee, J., Pham, M.D., Yu, J.X.: igraph: a framework for comparisons of disk-based graph indexing techniques. Proceedings of the VLDB Endowment **3**, 449–459 (2010)
10. He, H., Singh, A.K.: Closure-tree: An index structure for graph queries. In: ICDE, p. 38 (2006)
11. Jiang, H., Wang, H., Yu, P., Zhou, S.: Gstring: A novel approach for efficient search in graph databases. In: ICDE, pp. 566–575 (2007)
12. Leach, A.R., Gillet, V.J.: Substructure searching. In: An introduction to chemoinformatics, pp. 10–12
13. Shang, H., Zhang, Y., Lin, X., Yu, J.X.: Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. Proceedings of the VLDB Endowment (2008)
14. Shasha, D., Wang, J.T.L., Giugno, R.: Algorithmics and applications of tree and graph searching. In: PODS (2002)
15. Sun, B., Mitra, P., Giles, C.L.: Irredundant informative subgraph mining for graph search on the web. In: CIKM (2009)
16. Trinajstic, N.: Chemical Graph Theory. Vol. 1, 2, 2nd edn. CRC Press, Boca Raton (1992)
17. Williams, D.W., Huan, J., Wang, W.: Graph database indexing using structured graph decomposition. In: ICDE (2007)
18. Yan, X., Han, J.: gspan: Graph-based substructure pattern mining. In: ICDM, p. 721 (2002)
19. Yan, X., Yu, P.S., Han, J.: Graph indexing: a frequent structure-based approach. In: SIGMOD (2004)
20. Yuan, D., Mitra, P.: Lattice-based graph index for subgraph search. In: WebDB (2011)
21. Zhang, S., Hu, M., Yang, J.: Treepi: A novel graph indexing method. ICDE pp. 966–975 (2007)
22. Zhang, S., Li, J., Gao, H., Zou, Z.: A novel approach for efficient supergraph query processing on graph databases. In: EDBT, pp. 204–215 (2009)
23. Zhao, P., Yu, J.X., Yu, P.S.: Graph indexing: tree + delta $\leq$ graph. In: VLDB, pp. 938–949 (2007)
24. Zou, L., Chen, L., Yu, J.X., Lu, Y.: A novel spectral coding in a large graph database. In: EDBT, pp. 181–192 (2008)