A Lattice-based Graph Index for Subgraph Search

Dayu Yuan
Dept. of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802, USA
duy113@cse.psu.edu

Prasenjit Mitra
College of Information Sciences and Technology
The Pennsylvania State University
University Park, PA 16802, USA
pmitra@ist.psu.edu

ABSTRACT

Given a query graph q, a "subgraph-search" algorithm retrieves from a graph database D all graphs that have q as a subgraph, $D(q)$. Subgraph search is costly because of its involvement of a subgraph-isomorphism test, which is a NP-complete problem. Graph indexes are used to improve the algorithm efficiency by first filtering out a set of false answers and then verifying each graph that has passed the filtration with subgraph isomorphism tests. Many substructure features have been proposed to build the index aiming at improving the filtering power of the index. In this paper we improve the filtering power and query processing time by design of the index structure. We propose a lattice like index, Lindex, which is generally applicable on all graph features. Lindex achieves a high filtering rate by organizing index subgraphs in a graph lattice and adopting a specific design of value sets. Besides finding the candidate set $C(q)$ after filtering, Lindex can also find a set of true answers $Tr(q)$ without involving subgraph isomorphism tests. Accordingly, only candidate graphs in $C(q) - Tr(q)$ need to be verified. Our experiments show that Lindex outperforms other cutting-edge indexes on both frequent subgraph and infrequent subgraph queries.

1. INTRODUCTION

In recent times, there has been a growing need for efficient querying of graph data. Chemical information on the web and other types of graphical information, i.e., graphs representing CAD/CAM models [3], from the web can be extracted and stored in databases. One of the most popular methods of retrieving graphs from graph databases is by using *subgraph search*. In a graph database D, given a query graph q, a subgraph search algorithm retrieves all graphs $g \in D$ containing q as a subgraph. Deciding if one graph is a subgraph of another is referred to as the subgraph isomorphism problem; the problem has been shown to be NP-complete [2]. For large databases, an index based on the "filtering+verification" paradigm is necessary to enable efficient query processing [11, 5] . When the query graph is not a key in the index, the index lookup returns a candidates set $C(q)$ of graphs that may potentially contain the query, filtering out all graphs $D - C(q)$. However, $C(q)$ is typically larger than the answer set $D(q)$. In the verification step, a subgraph isomorphism test is performed to check that q is contained in each graph in $C(q)$. Thus, $|C(q)|$ subgraph isomorphism tests must be performed to eliminate subgraphs 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). Based on this paradigm, two categories of indexing algorithms have been proposed. Feature-based indexing methods first mine substructure features and then build an inverted index to facilitate the filtration [1, 6, 7, 8, 11, 12, 13]. Non-feature-based algorithms include CTree [5], GDI [9] and gCode [14]. Because non-feature-based approaches usually perform worse than the feature based approaches, as indicated in an experimental survey [4], we focus on feature-based indexing in this paper.

Feature-based algorithms mainly focus on maximizing the filtering power by reducing the size of the candidate set [8, 11, 12, 13]. The filtering power depends on the set of indexed substructures (subgraphs or subtrees). Gindex indexes frequent and discriminative subgraphs [11]. TreePi [12] and SwiftIndex [6] index frequent and discriminative subtrees (DFT). Tree+δ [13], indexes mostly DFTs but also indexes some subgraphs as well; the latter can largely improve the filtering power. Sun, et al., proposed a forward feature-selection algorithm based on information theory, MimR, that indexes a set of subgraphs with the approximate maximum average filtering power on a given set of training queries [8]. Cheng, et al., propose to index *frequent-subGraph queries* or *FG queries*, which are defined as queries with large supporting sets $D(q)$s [1]. FGindex pre-computes and stores the answer sets for all possible FG queries on disk. For non FG-queries, FGindex falls back to a filter+verify strategy. Unfortunately, for non-FG queries, the filtering power of FGindex is lower than that of MimR and Gindex because the subgraphs used by FGindex to index the database are mined without trying to maximize the filtering power. Thus, as we show in Section 4, FGindex does not optimize the average performance when the workload contains both FG and non-FG queries.

In previous methods [1, 6, 7, 8, 11, 12, 13], when a query graph was not a key in the index, the number of subgraph isomorphisms needed to answer a query q was at least $|D(q)|$. *Can we design a method that requires fewer subgraph isomorphisms than the size of the answer set $|D(q)|$?* In this work,

WebDB '11, Athens, Greece

we provide an affirmative answer. We propose *Lindex*, a lattice-based inverted index for graph databases. Each node in the index has a key-value pair associated with it. A key is a subgraph, say sg, and its value set V, is the set of database graphs that contain sg. In Lindex, an edge between two index nodes indicates that the subgraph corresponding to the key in the parent node is a subgraph of that in the child node. As in an inverted index, associated with each subgraph key sg in a node is a value-set that lists all database graphs g that contain sg. Lindex pre-computes and maintains the containment relationships among subgraphs that are keys in the index.

In Lindex, we have identified two properties that enable us to prune the candidate sets and thereby reduce the subgraph isomorphism tests. The first property utilizes the fact that database graphs which 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. We have also identified a second property that allows us to partition the value-sets such that subgraph isomorphisms need to be performed only on database graphs appearing in that partition resulting in further verification savings.

Besides improving the algorithm's efficiency by reducing the verification cost, Lindex also saves filtering cost. In the filtering step, the "the maximal subgraphs" of the query q have to be identified. Finding the "maximal subgraphs" also requires checking for subgraph isomorphisms from q to the indexed subgraphs, say, sg. Lindex makes this step efficient because instead of identifying mappings (isomorphisms) from q to an indexed subgraph, sg, from scratch, by walking down the lattice, one can grow a mapping from a query graph to a parent subgraph, sg' to incrementally generate a mapping to a child graph sg.

Lindex complements previous work that focused on identifying features to index. The benefits of Lindex, reduced verification cost and filtering cost, are independent of underlying substructure features. The benefits due to its lattice structure can be reaped in conjunction with the benefits due to other feature selection techniques. As future research identifies better feature selection algorithms and a better set of queries whose answers should be pre-computed and stored on disk, we can seamlessly plug those algorithms into the Lindex framework.

Experiment results show that Lindex has a faster response time than FGindex and Gindex, which are two representative graph indexes that have been shown to be very efficient in related work.

2. LINDEX FOR SUBGRAPH SEARCH

First we introduce a graph lattice, and then we show how Lindex is constructed based on a graph lattice. Then, we show how Lindex is designed to save verification cost.

2.1 Graph Lattice

A Graph Lattice $(GL, \subseteq)$ is a lattice defined using the subgraph isomorphism relation, $\subseteq$, satisfying reflexive, antisymmetric and transitive properties as follows: Given three graphs g_A, g_B and g_C in a graph lattice, (1) $g_A \subseteq g_A$(2) if $g_A \subseteq g_B$ and $g_B \subseteq g_A$, $g_A = g_B$(3) if $g_A \subseteq g_B$ and $g_B \subseteq g_C$, $g_A \subseteq g_C$. In a lattice, any pair of graphs $\{g_A, g_B\}$ 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 $\emptyset$. As shown in a Hasse diagram of a graph lattice in Fig. 1, 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, i.e., 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 a database, we refer to it as the *Complete Lattice*, $\mathcal{CL}(D)$. A lattice is a *Partial Lattice*, $\mathcal{PL}(S)$, if it only consists a subset, S, of subgraphs in the Complete Lattice. A complete lattice is too large to generate and manipulate, hence, in practice, Lindex always instantiates a partial lattice. Fig 1 is an instantiated partial lattice since not all subgraphs of database graphs are listed. Note that the solid nodes in Fig 1 represents database graphs, and they are not part of the instantiated partial lattice but visualized for demonstration purpose.

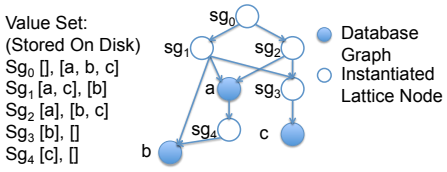


Figure 1: An example of Lindex

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

DEFINITION 1 (MAXSUB, MINSUPER).

$$maxSub(g,S) = \{sg_i \in S | sg_i \subset g, \nexists x \in S s.t. sg_i \subset x \subset g\}$$
$$minSup(g,S) = \{sg_i \in S | g \subset sg_i, \nexists x \in S s.t. g \subset x \subset sg_i\}$$

For brevity, because the discussion in the paper pertains to any database and feature selection algorithm, we abuse the notation to drop the second argument from $maxSub$ and $minSup$ and write them as $maxSub(g)$ and $minSup(g)$.

2.2 Structure of Lindex

Lindex, $L(D, S)$, is an inverted index built on a partial graph lattice $\mathcal{PL}(S)$ and a graph database D. In Lindex, a *node* t is associated with a <*key,value*> pair. The key, sg_t, is a subgraph in the graph lattice $\mathcal{PL}(S)$ and the value $V(sg_t)$ is a list of IDs of graphs in the database that are supergraphs of the key. Node $t_1 \prec t_2$ if and only if $sg_{t_1} \subset sg_{t_2}$. Given a graph g and a Lindex $L(D, S)$, its maximal-subgraph nodes are the nodes corresponding to $maxSub(g, S)$ in the partial lattice $\mathcal{PL}(S)$, and its minimal-supergraph nodes are the nodes corresponding to $minSup(g, S)$. The *root* of Lindex is a node r whose key is an empty graph $\emptyset$. In this paper we loosely use the term "a node" to refer to either a node in Lindex or a subgraph in a graph lattice based on context.

Each subgraph that is a key in Lindex is labeled using a string. Graphs that are keys in nodes whose only parent is the least element $\emptyset$ are labeled canonically using a depth-first-search (DFS) code as in [10]. The canonical label is constructed as follows. For each edge in a graph, a tuple of the form <ID(u), ID(v), Label(u), Label(edge(u, v)), Label(v)> is created for an edge between u and v where the functions ID() and Label() 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 referred to as the *canonical label*.

In order to minimize memory usage, which enables us to index additional graph features, Lindex stores the label of a graph key of a node as an extension of the label of its (chosen) parent node. For example, in Fig. 1, 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 $< 1,2,6,1,7 >$, then 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 details, see Section 3). Most feature mining algorithms find containment relationships among features. Hence, the construction of the Lindex needs no additional cost.

2.3 Strategies to Reduce Isomorphism Tests

Any graph g indexed by a supergraph node of q can be pruned from $C(q)$ (the set of graphs that are checked using subgraph isomorphisms) and directly included in the answer set because g contains q. This strategy is formalized as follows:

PROPERTY 1 (MINIMAL SUPERGRAPH PRUNING). *Given a query q, and Lindex $L(D,S)$, 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 $\forall h_j \in minSup(q)$.*

2.3.1 Value Set Partition

In this subsection, we show how the need for subgraph isomorphism tests is further reduced by partitioning the value-sets corresponding to a node in Lindex. In Fig. 2, we show a

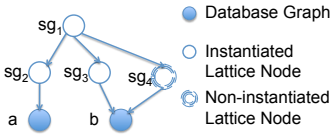


Figure 2: An example of value set partition

part of a complete lattice $\mathcal{L}$. sg_1, sg_2, sg_3, and sg_4 are graph keys of nodes in $\mathcal{L}$ out of which sg_1, sg_2 and sg_3 are instantiated in the partial lattice $\mathcal{S}$ constructed from $\mathcal{L}$. Let a and b be two graphs in the graph database. Which database graphs should be in the value-set of sg_1 to enable us to answer a query q? Recall that answers to graph queries are computed by intersecting the value-sets of the maximal subgraphs of a query and then verifying via subgraph isomorphism tests that the graphs in the intersection contain the query. 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 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 super-graph 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 a path from q to b but no path from q to a implying that q does not contain a; thus we do not need a in the value-set of sg_1. Note that because q can be placed in the direct path from sg_1 to b (by passing sg_4 which is not indexed), b must be in the value-set of g_1.

In existing methods [1, 12, 6, 8, 11, 13], 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 a node 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.

Based on this intuition, we propose the following proposition: If all paths in the complete graph lattice $\mathcal{CL}(D)$, from a graph sg_t to a graph g, where $g \in D$, go through at least one node $sg_{t'}$ such that $sg_{t'} \in S$, where S is the set of graphs instantiated in the partial lattice, then the database graph g should not be included in the direct value set of node t in Lindex $L(D,S)$.

This consequently leads to the following definition:

DEFINITION 2 (DIRECT & INDIRECT VALUE SET). *For a Lindex L(D, S), a graph g is in $V_d(sg_t)$ iff $g \in D_{sg_t}$ and there exists a path in the complete graph lattice, $\mathcal{CL}(D)$, from sg_t to g that does not pass through any node in the instantiated graph lattice $\mathcal{PL}(S)$ that has a key which is a supergraph of sg_t. That is,*

$$V_d(t) = \{g \in D(sg_t) | \exists path\ P(sg_t, g) \in \mathcal{CL}(D) \vee \exists path\ P(sg_t, x_1, ..., x_n, g) \in \mathcal{CL}(D) | \forall x_i \notin S, i \in \{1, n\}\}$$

The indirect 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 uses the following property to reduce the number of subgraph isomorphism tests:

PROPERTY 2 (DIRECT VALUE SET PRUNING). *Given a query q and Lindex $L(D,S)$, the candidate set $C(q) = \cap_i V_d(f_i) - \cup_j D(h_j), \forall f_i \in maxSub(q)$, and $\forall h_j \in minSup(q)$ contains $D(q) - \cup_j D(h_j)$.*

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. The proof of the soundness and completeness of the Lindex algorithm depends upon this property and is shown in our full version. Also for the construction of the partition, please refer to our full version [1].

We now provide a simple example to show how our algorithm works and the core benefits of Lindex over existing algorithms. In Fig. 1, let us say that the query is the graph a. The maximal subgraphs of a are determined to be sg_1 and sg_2 and the minimal supergraph of a is sg_4. The algorithm takes the intersection of the direct value sets of sg_1

[1] http://www.cse.psu.edu/research/publications/tech-reports/2011

and sg_2. The lists shown in the figure beside each node depicts 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 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 subgraph of the query and is added to the answer set. The answer set $\{a, c\}$ is output.

2.4 Lindex+: Disk-resident Lindex

To further reduce the candidate set $C(q)$, FGindex proposed to index FG-queries on disk so that they can be answered directly without subgraph isomorphism tests. Here, we propose Lindex+ to support indexing FG-queries on disk. Lindex+ contains two parts: an in-memory Lindex and a set of on-disk Lindexes. We also name the in-memory index as *first level* Lindex. We first introduce the concept of the maximum subgraph: it is one of the maximal subgraphs with the minimum frequency and the maximum depth in lattice. If two maximal subgraphs have the same minimum frequency and maximum depth, we choose the first subgraph according to their labels lexicographically. Each node of the first level Lindex is associated with a closure of on-disk features, whose maximum subgraph is the in-memory node. We build one on-disk Lindex for the closure of each first level index node. The instantiated lattice of the first-level Lindex resides in memory. We store value sets, instantiated lattices of the on-disk index on disk. To find an indexed FG-query graph q, we first traverse the in-memory Lindex to find its maximum subgraph sg, and then load the on-disk Lindex associated with sg. Finally we traverse the on-disk Lindex to find q and its value set.

3. FINDING $MAXSUB$ AND $MINSUP$

In order to find the maximal subgraphs of a query, q, previous methods like Gindex [11], MimR [8] may need to check whether all possible subgraphs of the query are indexed. The total number of subgraphs of q is exponential with respect to the size of q. The advantage of maintaining a graph lattice is that instead of constructing canonical labels for each subgraph of q and comparing them with the existing labels in the index to check if a node matches, while traversing a graph lattice, mappings constructed to check that a graph sg_t is contained in q can be extended to check whether a supergraph of sg_t in the lattice is contained in q by incrementally expanding the mappings from sg_t to q.

Since the lattice is an acyclic directed graph, one node may be visited several times during the lattice traversal. In order to prevent multiple visiting, we traverse the lattice according to a spanning tree of the lattice. A node sg_t has multiple parents in the partial lattice $\mathcal{PL}(S)$ (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.

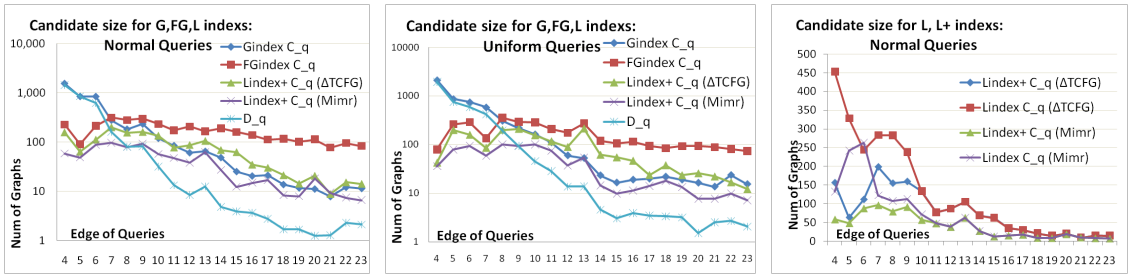
In the algorithm for maximum subgraph search of a query q, the graph lattice is traversed as follows. For the nodes (graphs) in the first-level (children of the root node of the lattice), there exists no mapping from nodes in the level above, i.e., the root node that corresponds to an empty graph. In this case, the algorithm checks whether each graph sg_c, is contained in the query graph q. If it is, all mappings of sg_c on q are temporarily 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 between sg_c and q. Since sg'_c, as a child of sg_c, is labelled as an extension of sg_c, the vertex $v_i(sg_c)$ in sg_c maps with the vertex $v_i(sg'_c)$ (the mapping can be obtained from the extension label of sg'_c). Also, based on the mappings in $M_{sg_c,q}$, vertex $v_i(sg_c)$ also maps to $v_j(q)$. Given that vertex $v_i(sg_c)$ maps with both $v_i(sg'_c)$ in sg'_c and $v_j(q)$ in the query graph, the vertex $v_i(sg'_c)$ maps to $v_j(q)$. With the bridging of sg_c, partial mappings $pM_{sg'_c,q}$ can be constructed at query-time.

For each $pM_{sg'_c,q}$, the algorithm, subSearch, checks to see if it can expand a complete mapping $M_{sg'_c,q}$. If the partial mapping can be expanded, the algorithm keeps on searching for mappings between sg'_c's children and 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 which use canonical labeling, since each canonical labeling as difficult as one subgraph isomorphism test, $T_{labeling} \approx T_{subIsomorphism}$, and an exponential number of subgraphs have to be labelled. Also, in the algorithm described above, we adopt a breath first approach and store all the mappings between one node t to the query q, $M(sg_t, q)$. However, in case of short of memory, it can be easily changed to a depth first way in which mappings are enumerated and extended one after anther.

After finding $maxSub(q)$, it is easy to find $minSup(q)$ by adopting the following filtering strategy: The set of minimal supergraph of a query q in the partial lattice, is a subset of the intersection of descendants of each subgraph node of q in the partial lattice, $minSup(q) \subseteq \cap_{g \in maxSub(q)} \text{Descendant}(g)$. This is because if the graph in the partial lattice is a supergraph of the query q, it must contain each of q's subgraphs in lattice as subgraphs of its own.

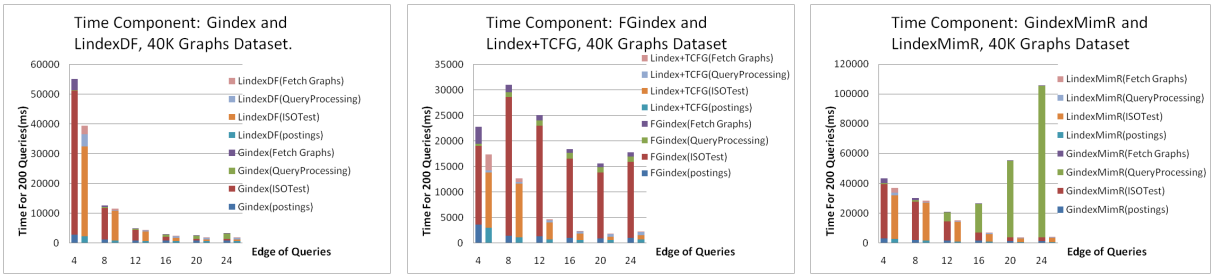
4. EXPERIMENTAL EVALUATION

We will refer to the disk-based Lindex as Lindex+ in this section, and Lindex only refers to the in memory Lindex. First, we evaluate the performance of our index on the AIDS Antiviral Screen dataset consisting of 43, 905 chemical structures, that has been used in [1, 8, 11]. For Gindex and FGindex, we set the parameters as in prescribed [11] and [1] (FGindex uses minimum support $0.03N$ where N is the dataset size). We use both δ-TCFG [1] and MimR [8] subgraph features to build Lindex. As was the case with related work, we do not have access to a real-world query workload. Thus, we used a randomly generated query set to train the MimR algorithm suggested by Sun, et al [8]. We assume that the test query size has a normal distribution with $\mu = 9$ and $\sigma^2 = 5$. In order to test the effectiveness of different indexes, we first generate a series of query sets Q_4 to Q_{23}, where the subscript denotes the size of queries in each query set. Each query set contains 55 queries generated from the graph database. We first choose 1000 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 normal distribution and a uniform distribu-



(a) (b) (c)

Figure 3: Exp on AIDS Dataset: 10,000 Graphs



(a) (b) (c)

Figure 4: Exp on AIDS Dataset: 40,000 Graphs

tion. Thus, frequent subgraphs have a higher chance to be selected. We also use a second dataset containing 5M chemical molecule graphs that we will refer to as the eMolecules Dataset [2] to test the scalability of our algorithm. The feature mining tasks were run on machines with 16G RAM and the indexing were tested with a maximum heap size of 2G.

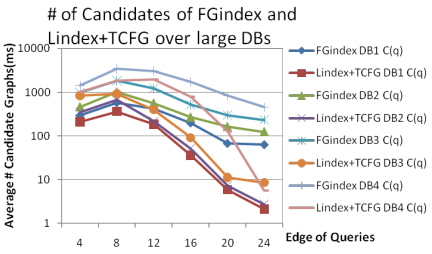
4.1 Effectiveness Test

We find 1587 discriminative features for Gindex. For FGindex, there are 2236 δ-TCFGs saved in memory and the rest of the FGs numbering 2300 are stored on disk. We use the same set of δ-TCFGs and on disk FGs to build a Lindex+. We also build a competing in-memory Lindex using 2236 mimr features. As we do in Lindex(+) and FGindex, to make the memory consumption comparable, we store the postings-lists/value-sets of Gindex features on disk. The on-disk postings is implemented using the Lucene library [3].

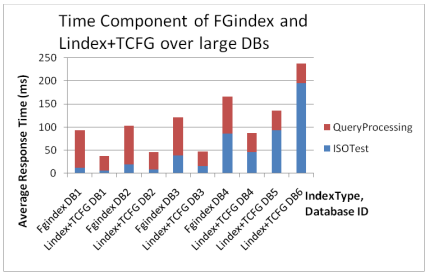
Fig. 3 shows the candidate set over a 10,000 graphs sample dataset. As in Figure. 3(a), Lindex+ with MimR features has the smallest candidate set when compared to Gindex and FGindex and Lindex+ with δ-TCFG features. Although FGindex and Lindex+(TCFG) share the same set of in-memory and on-disk features, the candidate set of Lindex+ is smaller than FGindex. This fact shows that the Lindex+ framework can result in additional benefits when used in conjunction with existing indexing algorithms like FGindex. We observed that the high filtering rate of Lindex+ arises due to the benefits of Property 2. Furthermore, FGindex could not find the whole set of maximal subgraphs features for queries resulting in the degradation of its filtering rate further. We observe similar results on queries selected uniformly as shown in Fig. 3(b). Fig. 3(c) compares the candidate set of Lindex and Lindex+. Lindex+ is mainly effective on small queries. As we further look into the query set, we find that a great portion of small queries are frequent subgraph queries that can be retrieved directly by Lindex+ and FGindex. When the size of the query grows, the size of candidate sets of Lindex and Lindex+ converges.

Fig. 4 shows the real query processing time components of different indices. The total response time includes *postings* fetch time (also include on-disk index loading time), *database graph* fetching time, *query processing* time (or *maxSub, minSup* search time), and *isomorphism test* time for verification. Lindex(+) using the same set of features, has less candidate verification time (high filtering rate) and less *maxSub and minSup* search time compared with other indices. Fig. 4(a) shows that Lindex outperforms Gindex with the same set of Discriminative and Frequent features. The query processing time of Gindex takes a small portion of the overall response time, since a frequency based apriori rule can be used to early prune the *maxSub* search space [11]. But for MimR features, this apriori rule cannot be applied and the *maxSub* search time outrages the verification time several times when the query is large, as can be seen in Fig. 4(c). By using Lindex on MimR features, the query processing time decreases significantly. Thus saves the overall response time. In FGindex, in order to control the query processing time, an incomplete *maxSub* set is returned [1]. As we can see, this incomplete *maxSub* set decreases the overall filtering rate, thus enlarges the cost on isomorphism tests. Also, when the query is large, the cost of incomplete *maxSub* search still takes twice time as in Lindex+. This means when the query processing time of FGindex dominates the overall response time (as shown in the following subsection), Lindex+(TCFG) takes only half time than FGindex using

[2] www.emolecules.com
[3] http://lucene.apache.org/java/docs/index.html



(a)



(b)

Figure 5: Large Scale Experiment on EMolecules

the same feature set.

4.2 Large Scale Experiment

In this section, we study the scalability of Lindex empirically. We randomly selected graphs from the eMolecules dataset to construct a bunch of experiment datasets ranging from 32K to 1M graphs. We build Lindex with the δ-TCFG features. In order to better test the scalability of Lindex, we generate δ-TCFG out of subgraphs with minimum support $0.01N$. The memory consumption for indexes is stable when the database grows, because the number of feature selected to build the index is stable. We observe that the average memory consumption for Lindex storing one feature is 0.2KB, Gindex is 0.26 KB, and FGindex is 0.33KB, which shows that Lindex is an compact index.

We evaluate the performance of FGindex and Lindex+(TCFG) on DB1(32K), DB2(64K), ..., and DB6(1M) graphs. Both FGindex and Lindex+TCFG store almost 10,000 TCFG features in memory and 20,000 frequent subgraph features on disk. For Datasets bigger than 128, 000 graphs, mining frequent subgraph will be out of memory, thus we use the frequent feature set of DB3 to stimulate DB4-6. As we can see from Fig. 5(b), Lindex+TCFG outperforms FGindex on both candidate verification and *maxSub* search. The tight candidate set of Lindex+TCFG can also been seen from Fig. 5(a). In Fig. 5(b), the average time cost of *maxSub* search outweighs time cost on candidate verification, and this is because the number of in-memory features (10,000) is bigger than average size of the candidate set. When the graph database doubles, the *maxSub* search cost is stable if the number of in-memory features are stable. But the candidate set doubles accordingly. Thus for graph database larger than 256K graphs, the verification time dominates the overall response time again.

For index construction, constructing the lattice takes 272 seconds for 10,000 features. This cost is basically the same as mining δ-TCFGs from all frequent subgraphs. This cost is negligible compared with the cost of building postings. As mentioned before, for large graph dataset, we usually sample the dataset and mine frequent subgraphs over this sample set first. The postings (supporting set) is constructed by scanning the graph database and adding the graph id to the postings of their subgraphs. Take DB4(256K graphs) for example, FGindex takes $53,175$ seconds to build the postings, while Lindex only took $20,270$ seconds to build the postings (including the direct or indirect value set test). This further shows the benefit of a fast maxSub(q) search algorithm.

5. CONCLUSION

We proposed Lindex to process subgraph queries efficiently. 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 index makes efficient finding of supergraphs possible. The candidate set is further reduced by partitioning the value set of each indexed feature. 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 over an existing benchmark dataset.

6. REFERENCES

[1] J. Cheng, Y. Ke, W. Ng, and A. Lu. Fg-index: Towards verification-free query processing on graph databases. In *SIGMOD*, 2007.

[2] S. A. Cook. The complexity of theorem-proving procedures. In *STOC*, pages 151–158, 1971.

[3] M. El-Mehalawi and R. A. Miller. A database system of mechanical components based on geometric and topological similarity. *J. CAD*, 35(1):83 – 94, 2003.

[4] W.-S. Han, J. Lee, M.-D. Pham, and J. X. Yu. igraph: a framework for comparisons of disk-based graph indexing techniques. *Proc. VLDB Endow.*, 3:449–459, September 2010.

[5] H. He and A. K. Singh. Closure-tree: An index structure for graph queries. In *ICDE*, page 38, 2006.

[6] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proc. VLDB Endow.*, 2008.

[7] D. Shasha, J. T. L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *PODS*, 2002.

[8] B. Sun, P. Mitra, and C. L. Giles. Irredundant informative subgraph mining for graph search on the web. In *CIKM*, 2009.

[9] D. W. Williams, J. Huan, and W. Wang. Graph database indexing using structured graph decomposition. In *ICDE*, 2007.

[10] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *ICDM*, page 721, 2002.

[11] X. Yan, P. S. Yu, and J. Han. Graph indexing: a frequent structure-based approach. In *SIGMOD*, 2004.

[12] S. Zhang, M. Hu, and J. Yang. Treepi: A novel graph indexing method. *ICDE*, pages 966–975, 2007.

[13] P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: tree + delta <= graph. In *VLDB*, pages 938–949, 2007.

[14] L. Zou, L. Chen, J. X. Yu, and Y. Lu. A novel spectral coding in a large graph database. In *EDBT*, pages 181–192, 2008.