Mining and Indexing Graphs for Supergraph Search

Dayu Yuan* Prasenjit Mitra*,† C. Lee Giles*,†

* Department of Computer Science and Engineering
† College of Information Sciences and Technology
The Pennsylvania State University, University Park, PA, 16802, USA
{duy113, pmitra, giles}@ist.psu.edu

ABSTRACT

We study *supergraph search* (SPS), that is, given a query graph q and a graph database G that contains a collection of graphs , return graphs that have q as a supergraph from G. SPS has broad applications in bioinformatics, cheminformatics and other scientific and commercial fields. Determining whether a graph is a subgraph (or supergraph) of another is an NP-complete problem. Hence, it is intractable to compute SPS for large graph databases. Two separate indexing methods, a "filter + verify"-based method and a "prefix-sharing"-based method, have been studied to efficiently compute SPS. To implement the above two methods, subgraph patterns are mined from the graph database to build an index. Those subgraphs are mined to optimize either the *filtering gain* or the *prefix-sharing gain*. However, no single subgraph-mining algorithm considers both gains.

This work is the first one to mine subgraphs to optimize both the *filtering gain* and the *prefix-sharing gain* while processing SPS queries. First, we show that the subgraph-mining problem is NP-hard. Then, we propose two polynomial-time algorithms to solve the problem with an approximation ratio of $1-1/e$ and $1/4$ respectively. In addition, we construct a lattice-like index, LW-index, to organize the selected subgraph patterns for fast index-lookup. Our experiments show that our approach improves the query processing time for SPS queries by a factor of 3 to 10.

1. INTRODUCTION

A number of recent studies have proposed algorithms to address the graph search problem [3, 5, 14, 17, 19, 20, 21]. There are two common search scenarios: *subgraph search* and *supergraph search*. In a subgraph search, given a query graph q, the algorithm searches for all graphs that have q as a subgraph, from a graph database [5, 14, 17, 19]. A supergraph search, on the other hand, retrieves all the database graphs that have q as a supergraph [3, 4, 20, 21].

Supergraph search (SPS) has broad applications in various scientific and commercial fields. In Cheminformatics, molecules are modeled as undirected graphs with nodes representing atoms and edges representing chemical bonds [15]. SPS is commonly used for synthetic planning [15] and for deriving generic reaction knowledge in the process of synthesizing new compounds with already known chemical molecules [22]. In Computer-Aided Design (CAD), 3-D mechanical parts are often represented as attributed graphs, in which each node represents a "face" of the solid model and each edge represents an arc between two connecting faces [8]. SPS is commonly used to find the existing components within a newly designed part to facilitate manufacturing planning and cost estimation. In image processing and computer vision, attributed graphs are also used to model figures [10]. For each figure, a set of visual features are first extracted, and then an attributed graph is constructed with those features as nodes. An edge is drawn to connect two visual features if they are close in the figure. In figure retrieval, SPS is used to search for simple fragments contained in the query figures [3]. Graphs are also constructed to model the system call dependencies with nodes representing system calls and directed edges representing dependencies [1]. In malware detection, SPS is used to search for system-call patterns contained in malicious softwares.

Proceedings of the VLDB Endowment, Vol. 6, No. 10

1.1 Filtering with Graph Index

Detecting the containment relationship between two graphs, also referred to as the subgraph isomorphism test (SGI), is an NP-complete problem. Using SGI to sequentially check each database graph is computationally expensive for large databases. A *filter+verify* indexing method has been proposed to quickly process SPS queries [3, 4, 20, 21]. Given a query graph q and a graph database G, the *filter* sub-routine first prunes false answers and generates a candidate set $C(q)$, where $|C(q)| \ll |G|$. Then in the *verify* step, only the candidate graphs are verified with SGIs. An *exclusive-logic* method is proposed to filter the false answers: if a subgraph p is not contained in the query graph q, then all database graphs containing p can not have q as a supergraph, and thus can be safely filtered [3].

The time taken to process SPS queries depends on the patterns selected for indexing [3, 4, 20, 21]. A set of good quality index patterns filters false answers aggressively, and consequently saves the query-processing time by computing fewer SGIs. We refer to the savings of the query-processing time due to the exclusive-logic filtering as the *filtering gain*. Chen, et al., showed that mining index patterns to optimize the filtering gain on a set of training queries Q is NP-hard [3]. They proposed a polynomial-time greedy solution with an approximation ratio of $1-1/e$ [3]. Cheng, et al., further proposed a heuristics-based algorithm [4]. The algorithm mines index patterns from an *integrated graph* instead of frequent subgraphs in order to save subgraph-mining time. An integrated graph is a compact representation of a set of graphs G, such that the commonality of G are shared [4].

1.2 Prefix Sharing with Graph Index

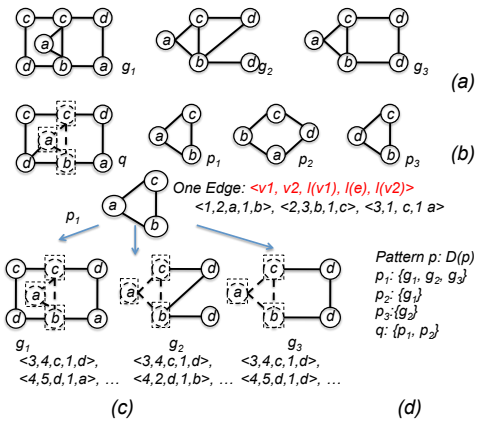


Figure 1: An example of the SPS: (a) database graphs, (b) query q and patterns, (c) prefix index, (d) filter index

Zhang et al. [20] and Zhu et al. [21] proposed prefix-sharing indexes to reduce the time of each SGI and thereby save the verification time of processing SPS queries. A prefix of graphs is a *common subgraph* of those graphs. For example, in Figure 1(c), prefix p_1 is a common subgraph of database graphs $\{g_1, g_2, g_3\}$. Assume that all of $\{g_1, g_2, g_3\}$ must be checked with SGIs in the verify step. To reduce the verification time, the prefix-sharing algorithms first search for the mappings from a common prefix subgraph p_1 to the query q (dashed lines in Figure 1(b)) with one subgraph isomorphism test (SGI). Then, the algorithms reuse these mappings while searching for subgraph isomorphism mappings from $\{g_1, g_2, g_3\}$ to q. As a result, the time taken to verify $\{g_1, g_2, g_3\}$ are subgraphs of q decreases from $\sum_{i=1}^{3} c(g_i)$ to $c(p_1) + \sum_{i=1}^{3} c(g_i \setminus p_1)$, where $c(\cdot)$ is the time for computing SGI and $c(g_i \setminus p_1)$ is the time for computing SGI from g_i to q by extending the mappings from p_1 to q. One database graph g contains multiple subgraphs, each of which can be g's prefix. However, to construct the prefix index, we can only choose one subgraphs p_i as g's prefix and serialize g to a string with p_i as the string prefix [20, 21]. Choosing multiple prefixes and thus serializing a graph in multiple ways will increase the size of the graph database and the size of index multiple times without significant gains in query-processing time [20, 21].

The choice of the prefix influences the query-processing time of SPS [20, 21]. For example, in Figure 1, we can choose either p_1 or p_2 as the prefix of the database graph g_1. When p_2 is chosen as the prefix instead of p_1, the time of verifying the graphs $\{g_1, g_2, g_3\}$ changes to $c(p_1) + \sum_{i=2}^{3} c(g_i \setminus p_1) + c(p_2) + c(g_1 \setminus p_2)$. We refer to the savings in the query-processing time using prefix-sharing as *prefix-sharing gain*. Zhang et al. showed that mining index patterns to optimize the prefix-sharing gain is NP-hard [20]. They proposed a polynomial-time solution with an approximation ratio of $1/2$. Zhu et al. studied empirical improvements to the polynomial-time algorithm [21].

1.3 Our Method

No previous algorithms have used both the *filtering gain* and the *prefix-sharing gain* together for pattern mining. In this paper, we propose to mine index patterns to optimize an objective function that evaluates both the *filtering gain* and the *prefix-sharing gain*. We first show the NP-hardness of the pattern-mining problem referred herein. Then, we propose a greedy algorithm with an approximation ratio of $1 - 1/e$. This algorithm has a time complexity of $O(k|H||G||Q|)$, for selecting k patterns out of all frequent subgraphs H on a graph database G and training queries Q. The algorithm is not scalable to large-scale datasets. To improve its scalability, we propose a more efficient objective function to approximate the original one. Using the approximate objective function reduces the runtime complexity of the greedy algorithm to $O(k|H||G|)$. The greedy algorithm has a space complexity of $O(|H||G|)$. Given a large number of frequent subgraphs H, the algorithm will run out of memory for a large dataset G. To alleviate the memory bottleneck, we further propose a memory-efficient algorithm with space complexity $O(k|G|)$ ($k \ll |H|$). The memory-efficient algorithm has an approximation ratio of $1/4$. However, our empirical studies show that the processing time for SPS queries using an index constructed by the memory-efficient algorithm is almost the same as that using an index constructed using the greedy algorithm.

The objective function of our pattern-mining algorithm requires training queries. Training queries are commonly used for query optimization [2] and graph mining [3, 14, 19]. First, we show the merit of training queries by proving that it is insufficient to mine patterns solely based on the database graphs. Then, we show that our algorithm does not overfit to a certain set of queries. When the training queries are not available, we use the graph database as a surrogate of the training queries for a "cold" start, assuming that both the queries and the database graphs have similar statistical distributions. Similar assumptions are made in related works [3, 14]. Our experiment results show that the patterns mined with a cold start are as effective as patterns mined with training queries on processing SPS queries.

Besides mining index patterns to reduce the verification time for processing SPS queries, we also propose an index structure, LW-index, to reduce the index-lookup time, which is the other component of the query-processing time. We design LW-index by organizing index patterns in a graph lattice (see Section 3.2). A query q is processed as follows (1) lookup index patterns not contained in q, $P(q) = \{p \in P | p \not\subseteq q\}$ (2) fetch the value for each $p \in P(q)$, and construct the candidate graphs as $C(q) = G - \cup_{p \in P(q)} G(p)$, where $G(p)$ contains all database graphs containing p. LW-index is designed such that the time costs for both (1) and (2) are significantly reduced.

Table 1 summarizes related works. See Section 7 for details. The contributions of this work are as follows:

1. Our algorithm is the first to use both the filtering gain and the prefix-sharing gain for index pattern mining. A greedy pattern-mining algorithm is proposed with an approximation ratio of $1 - 1/e$. Further, a memory-efficient algorithm is proposed to reduce the memory consumption from $|H||G|$ to $k|G|$.

2. We design LW-index to support fast index lookup.

3. Our approach outperforms previous works by a factor of 3-10 on processing SPS queries on real-world data.

Organization: First, we introduce the preliminary in Section 2. Next, we introduce the query-processing framework and LW-index structure in Section 3. In Section 4, we introduce the pattern-mining algorithms . We discuss the use of training queries in Section 5. In Section 6, we evaluate the performance of both the pattern-mining algorithms and the LW-index structure.We discuss related works in Section 7 and finally conclude in Section 8.

2. PRELIMINARY

A graph is a triple $g(V, E, L)$ defined on a set of vertices V and a set of edges E such that $E \subseteq V \times V$. A label $\ell(u)$ or $\ell(e) \in L$

Table 1: SPS Solutions

Methods	Index	Patterns	Mining Time	Comments
cIndex	Top down and bottom up index	Mine index pattern considering filtering gain only.	$O(\|H\|\|G\|\|Q\|)$	No prefix sharing for pattern selection or query processing
GPTree	CRGraph, FGPTree	Mine significant patterns to maximize filtering gain (based on heuristics). Mine another set of patters for prefix sharing.	$O(\|H\|\|G\|)$	Two sets of patterns. Extra time on index-lookup. Num of patterns explodes with decreasing minSupport
PrefixIndex	Prefix Index & Hierarchical Prefix Index	Mine patterns considering only prefix-sharing gain.	$O(\|H\|\|G\|^2)$	Low filtering power since patterns are selected without optimizing filtering gain
IGQuery	Integrated Graph IG	Mine discriminative patterns for filtering based on heuristics	$O(\|IG\|)$	Effective for batch queries. Perform similar to cIndex and GPTree on single query
LW-index	Lattice-like index	Mine patterns considering both prefix-sharing and filtering gain.	$O(k\|H\|\|G\|\|Q\|)$(Exact) $O(k\|H\|\|G\|)$(Aprox)	

is associated with each node v or edge e. In this paper, we address only *undirected, labeled, and connected* graphs for simplicity; however, our methods with minor modifications are applicable to other graphs. A *subgraph* of a graph $g(V, E, L)$ is a graph $h(V', E', L)$ with a vertex set $V' \subseteq V$ and edge set $E' \subseteq E$. A graph g is a *supergraph* of h if h is a subgraph of g. We denote the subgraph relationship between g and h as $g \supset h$ and $h \subset g$. Graphs are commonly serialized to a list of tuples, each of which represents one edge [16, 12, 20]. For example, the tuple for the edge $e(v_1, v_2)$ is $(v_1, v_2, \ell(v_1), \ell(e), \ell(v_2))$. A k-tuple graph p is a prefix of a serialization of g if p is a subgraph of g and the first k tuples in the serialization of g equals p.

DEFINITION 1 (SUBGRAPH ISOMORPHISM). *Given two graphs $g_1 = (V_1, E_1, L_1)$ and $g_2 = (V_2, E_2, L_2)$, an isomorphism between g_1 and g_2 is a bijection between V_1 and V_2 that preserves the labels and connectivity of the two graphs. This bijection is a mapping $m(g_1, g_2)$, which maps any adjacent vertices u_1, v_1 in g_1 to adjacent vertices u_2, v_2 in g_2 with labels $\ell(u_1) = \ell(u_2)$, $\ell(v_1) = \ell(v_2)$, and $\ell(e(u_1, v_1)) = \ell(e(u_2, v_2))$, and vice versa. A Subgraph Isomorphism between g_1 and g_2 is an isomorphism between g_1 and one of $g_2's$ subgraph.*

A partial mapping $pm(g_1' \subset g_1, g_2' \subset g_2)$ is a mapping from a graph g_1' to a graph g_2', where g_1' is a subgraph of g_1 and g_2' is a subgraph of g_2. $g_1 \subseteq g_2$ if there exists a partial mapping that can grow to a full-coverage mapping that covers all nodes and edges of g_1. We use $m(g_1, g_2)$ to represent a full-coverage (sub)graph mapping from g_1 to g_2, and use $M(g_1, g_2)$ to refer to all the full-coverage mappings from g_1 to g_2 since there are multiple ones. The search for a mapping $m(g_1, g_2)$ can start from a pre-computed partial mapping $pm(p \subset g_1, g_2)$, $p \prec g_1$. This procedure saves computational cost when more than two graphs having p as a prefix are tested against g_2 for subgraph isomorphism.

Frequent subgraphs are commonly used as index patterns in processing graph search [3, 5, 17, 20, 21]. In a graph database G, the database graphs containing sg comprise the *supporting set* of sg, $G(sg)$. The *support* of sg is $|G(sg)|$. The subgraph sg is a *frequent subgraph* if and only if $support(sg) \geq \delta \cdot |G|$, where δ is a tunable parameter, named as the minimum support.

3. PROCESSING SPS QUERIES

Addressing SPS querying involves three steps: (1) mine index patterns, (2) construct a graph index, and (3) process SPS queries using the graph index. We study all three steps in this paper. In this section, we discuss (2) and (3).

3.1 Framework

In this subsection, we study how to process SPS queries on pattern-based indexes. No-pattern indexes have been shown to perform worse than pattern-based indexes [7].

A *graph index* with index patterns P contains (key, value) pairs where a key is a pattern $p_i \in P$ and its value is all database graphs containing p_i, that is, $G(p_i)$. Each database graph g is also associated with a prefix pattern p_j, $p_j \subseteq g$ and $p_j \in P$. The graph g is labeled by a string with p_j as the string prefix. This graph index can be used for both filtering and prefix-sharing.

Algorithm 1 Process SPS Queries

Input: Graph Index I with Patterns P, Query graph q,
Graph Database G, Supporting Set for Each Pattern $p \in P$, $G(p)$
Output: Graphs contained in q

1: $An(q) \leftarrow \emptyset$ (An(q) is the answer set)
2: patterns not contained in q, $P(q) = \emptyset$.
3: hashtable storing the embeddings, $T = \emptyset$
4: **for** each index pattern $p \in P$ **do**
5: **if** $p \subseteq q$ **then**
6: $T \leftarrow T \cup$ (key $= p$, value$= M(p, q)$), where $M(p, q)$ is the set of mappings from p to q
7: **else**
8: $P(q) \leftarrow P(q) \cup \{p\}$
9: **end if**
10: **end for**
11: fetch the value set for each pattern $p \in P(q)$, $G(p)$
12: **for** each graph $g \in G - \cup_{p \in P(q)} G(p)$ **do**
13: $M(p_j, q) = T.get(p_i)$, where p_j is the prefix of g
14: search for a mapping $m(g, q)$ by extending each $m(p_j, q) \in M(p_j, q)$.
15: **if** $\exists m(g, q)$, that is $g \subseteq q$ **then**
16: put g to $An(q)$.
17: **end if**
18: **end for**
19: return $An(q)$.

Algorithm 1 describes the use of the graph index on processing SPS queries. The algorithm integrates both the "filter+verify" methods (line 11) and the "prefix-sharing" methods (lines 13-17). Algorithm 1 comprises two parts: **index-lookup** (lines 4-11) and **verify** (lines 12-18). Given a query graph q, first, the SPS algorithm looks up the graph index for the index patterns that are not subgraphs of the query, $P(q) = \{p \in P | p \not\subseteq q\}$ (line 8) (see Section 3.2 for details). Then, the algorithm fetches the value set of each $p \in P(q)$, and filters graphs in $\cup_{p \in P(q)} G(p)$ based on the exclusive logic. Finally, in the verify step, each candidate graph in $G - \cup_{p \in P(q)} G(p)$ is tested with SGI to decide whether it is subgraph isomorphic to the query or not. For a graph g labeled with a

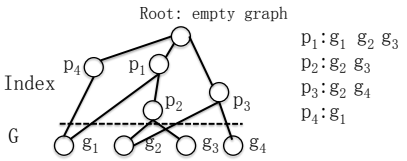


Figure 2: An example of LW-index

pattern p_j as its prefix, the mappings from p_j to q can be shared and reused to compute the subgraph isomorphism from g' to q, where g' is another candidate graph labeled with p_j as its prefix.

Correspondingly, the time of processing SPS queries comprises **index-lookup time** and **verification time**. The index-lookup time can be reduced by the structural design of the graph index, as introduced in Section 3.2. The verification time can be reduced by filtering and prefix sharing. As shown in Algorithm 1, the verification time depends on the index patterns P and prefix graphs $pref(g)$. We discuss how to mine index patterns and assign prefixes such that the verification time is optimized in section 4.

3.2 LW-index and Fast Index-lookup

We introduce a lattice-like index structure, LW-index, that supports fast index-lookup. The graph index should support two operations: (1)**key-lookup**: given the query graph q, identify the index patterns p (keys) that are subgraph isomorphic to q, $p \subset q$, or $p \not\subset q$ (lines 4-10). (2)**value-lookup**: given all the identified keys, $p \not\subset q$, retrieve and unite the value sets of those keys (line 11).

To support fast **key-lookup**, LW-index adopts Lindex, a lattice index structure we introduced [18]. Due to space limitations, we give only a brief description of Lindex here; for details please refer to the original work (see [18]). In LW-index, index patterns are organized in a lattice where the partial order is defined using the subgraph relationship. That is, if there is an edge from p_i pointing to p_j in Lindex, then $p_i \subset p_j$. Figure 2 shows an example of LW-index. For simplicity, all edges that can be obtained by transitivity are ignored. For example, if $p_i \subset p_x \subset p_j$, then the edge $e = (p_i, p_j)$ is not included in the lattice. The root node of the lattice is an empty pattern $p_0 = \emptyset$. The advantage of organizing index patterns in a lattice is that by moving down the lattice, the algorithm can grow a mapping from a parent pattern p_i to q to generate a mapping from a child pattern p_j to q, if $p_i \prec p_j$. In other words, the mappings $M(p_j, q)$ can be obtained by extending the mappings $M(p_i, q)$ instead of computing them afresh. Hence, the key-lookup time is largely reduced by using Lindex [18].

In LW-index, all the index patterns are organized in a lattice and stored in memory for fast index-lookup, as shown in Figure 2. The value associated with each key p is a list of database graphs containing p. The values can be either stored on disks for large G or in memory. Note that LW-index is different from CRGraph [20], which also organizes index patterns in a lattice. In CRGraph, there is an edge from p_i to p_j iff $G(p_i) \supseteq G(p_j)$. One major drawback of CRGraph is that it has to be re-constructed when the database G updates. In LW-index, only the value sets of the patterns are updated in response to database insertion, deletion or update. The index needs re-construction only when the index patterns P are remined. To support fast **value-lookup**, LW-index filters the union of value sets of *minimum non-subgraph patterns*.

DEFINITION 2 (MIN-NON-SUBGRAPHS). *Given a LW-index with index patterns P, for a query graph q, the set of minimum non-subgraph patterns of q is $minP(q) = \{p \in P | p \not\subseteq q \wedge \nexists p' \not\subseteq q,\ s.t.\ p' \subset p\}$.*

For example, in Figure 2, if a query q does not contain both p_1 and p_2 as subgraphs, then only p_1 is a minimum non-subgraph of q. p_2 is not a *minimum* non-subgraph, because $p_1 \subset p_2$. $minP(q)$ can be found by walking through the lattice of LW-index. When the algorithm reached an index pattern $p \not\subseteq q$, and each of p's ancestor p' is subgraph isomorphic to q, $p' \subset q$, then, p is found as a minimum non-subgraph pattern of q and put into $minP(q)$. Then, all of p's descendants are marked false so that they can be skipped because they are not the *minimum* non-subgraphs of q.

4. MINING PATTERNS FOR SPS

The processing times for SPS queries depends upon the subgraph patterns selected for indexing. In this section, we study the problem of subgraph pattern mining. We first formulate the problem and show it is NP-hard. Then, we propose a polynomial-time greedy algorithm for pattern mining. We show that the algorithm has an approximation ratio of $1 - 1/e$ because the objective function is submodular (see Section 4.2). The greedy algorithm is time consuming with a time complexity of $O(k|H||G||Q|)$. To improve its scalability, we propose an approximate objective function and reduce the time complexity of the greedy algorithm to $O(k|H||G|)$. The greedy algorithm is memory consuming because it needs to pre-mine and store all frequent subgraphs H in memory. To alleviate the memory bottleneck, we further propose a memory-efficient algorithm that stores only k index patterns in memory, where k is the number of selected patterns. We show that the memory-efficient algorithm has an approximation ratio of $1/4$. Although its theoretical approximation ratio is worse than that of the greedy algorithm, the memory-efficient algorithm empirically mines patterns as good as the greedy algorithm (see Section 6).

4.1 The Pattern Mining Problem

It is impossible to index all subgraphs given the limited memory. A pattern p can help to filter false graphs if it is *frequent* in G and *infrequent* in the queries Q. A pattern p can help to reduce the verification time by prefix sharing if p is *frequent* in both G and Q. Therefore, a pattern p is a candidate to be indexed if and only if it is frequent in G. However, there are still a large number of frequent subgraphs and most of them are redundant to each other, e.g., two frequent subgraphs with similar supporting sets are redundant, because they filter similar graphs for similar queries. Furthermore, given two frequent subgraphs $p_1 \subset p_2$ and $G(p_1) \approx G(p_2)$, p_1 is redundant for prefix-indexing $g \in G(p_1)$ given that p_2 is selected for indexing, because p_1 can not further reduce the verification time.

The task of pattern mining is to select index patterns P, such that the query-processing time over the training queries Q is minimized.

DEFINITION 3 (PATTERN MINING). *Given a dataset G, training queries Q, and frequent subgraphs H, we mine k subgraphs P for indexing, such that the time saving by filtering and prefix-sharing is maximized:*

$$P = \underset{P \subset H, |P| \leq k}{argmax}\ Gain(P) \tag{1}$$

$$= \underset{P \subset H, |P| \leq k}{argmax} \sum_{q \in Q} \sum_{g \in G} w(g, q), \tag{2}$$

$$w(g,q) = \begin{cases} c(pref(g), q), & g \in (G - \cup_{p \not\subset q} G(p)) \\ c(g, q) & otherwise \end{cases} \tag{3}$$

$c(g, q)$ is the time required to test the subgraph isomorphism from g to q. $c(pref(g), q)$ is the prefix-sharing gain achieved by indexing g with its prefix, $pref(g)$, when the graph g needs to be verified

with SGI. $c(g, q)$ is the filtering gain when the graph g can be filtered for query q.

We drop the second parameter of $c(g, q)$ based on the assumption that the time for subgraph isomorphism test from a graph g to different queries q is approximately the same.

The value of k is constrained by the available memory or according to an *index-lookup cost* given unlimited memory. The index-lookup cost, $|Q|r$, is the average time taken to look up one index pattern to process training queries. The total time cost of looking up an index with k patterns P_k is $k|Q|r$. And the total saving of the query-processing time is $Gain(P_k) - k|Q|r$. We stop selecting more patterns if the increase of the $Gain$ function by selecting one additional index pattern is smaller than the increase of the index-lookup cost. The algorithm selects k' patterns, $P_{k'}$, such that k' is the smallest number with $Gain(P_{k'+1}) - Gain(P_{k'}) < |Q|r$. The total saving of the query-processing time decreases if we select more than k' patterns, because $\forall k'' > k'$, $Gain(P_{k''+1}) - Gain(P_{k''}) \leq Gain(P_{k'+1}) - Gain(P_{k'}) < |Q|r$. The above inequality holds because $Gain(P)$ is a submodular function (see Section 4.2).

In related works [3, 20, 21], the index-lookup cost is set to be the time cost of the subgraph isomorphism test from p to queries. In this paper, *the index-lookup cost is largely reduced because of the use of LW-index* (Section 3.2). Thus, we set r to be a small constant, e.g., 2 in Section 6.

NP-Hardness: Before showing the pattern-mining problem is NP-hard, we first re-write the gain function defined in Equation 3.

$$w(g, q) = max_{p \in P} w(g, q, p) \tag{4}$$

$$w(g, q, p) = \begin{cases} c(g) & p \subseteq g \wedge p \not\subseteq q \\ c(p) & p \subseteq g \wedge p \subseteq q \\ 0 & p \not\subseteq g \end{cases}$$

Equation 4 is equivalent to Equation 3, because (1) if there exists an index pattern $p_i \in P$ that can help filter the graph g for query q, $p_i \subseteq g \wedge p_i \not\subseteq q$, then $w(g, p) = c(g) \geq max_{\forall p \subseteq g} c(p)$. (We define $c(g)$ as a monotonically increasing function, that is, $c(g) \geq c(p)$ if $g \supseteq p$). (2) otherwise, if g can not be filtered for the query q, then, $w(g, p) = c(pref(g)) = max_{p \subseteq g} w(g, q, p) = c(p_j)$, and g chooses pattern p_j as its prefix. Hence,

$$\begin{aligned} P &= \underset{P \subset H, |P| \leq k}{argmax}\ Gain(P) \\ &= \underset{P \subset H, |P| \leq k}{argmax} \sum_{q \in Q} \sum_{g \in G} w(g, q) \\ &= \underset{P \subset H, |P| \leq k}{argmax} \sum_{q \in Q} \sum_{g \in G} max_{p \in P} w(g, q, p) \end{aligned} \tag{5}$$

To show that the pattern-mining problem is NP-hard, we give a reduction from a known NP-hard problem, the weighted max-k-coverage problem.

DEFINITION 4 (WEIGHTED MAX-K-COVERAGE). *Given a universe of n elements $U = \{e_1, e_2, \ldots, e_n\}$, each of which has a weight $w(e_j)$, and a collection of sets $\{S_1, S_2, \ldots, S_m\}$, where $S_i \subseteq U$, find k sets such that $\sum w(e_i)y(e_i)$ is maximized, where $y(e_i) = 0$ if e_i is not covered by any of the selected sets, or $y(e_i) = 1$ otherwise.*

CLAIM 1. *max-k-coverage is polynomial-time (Karp) reducible to Pattern Mining.*

PROOF. The pattern mining problem is a generalization of the weighted max-k-coverage problem. In the pattern-mining problem, the universe contains all (g, q) pairs, $U = \{(g, q) | g \in G, q \in Q\}$, and each pattern is a set $S(p) \subseteq U$ where $S(p)$ covers the element (g_i, q_j) with weight $w(g_i, q_j, p)$ if $w(g_i, q_j, p) > 0$. Hence, we can reduce the max-k-coverage problem to the pattern-mining problem by first representing each element e_ℓ to a (g, q) pair, $e_\ell = e_{(g_i, q_j)}$, and then assigning the weight of $w(e_{(g_i, q_j)})$ to $w(e_\ell)$, $\forall i, j$, in polynomial time. □

4.2 A Greedy Algorithm

We propose a greedy algorithm to solve the pattern-mining problem with approximation guarantee. The algorithm greedily chooses an unselected pattern in each iteration maximizing the increase of the objective function. Algorithm 2 describes the procedure in detail. At line 5, the pattern with the maximum increase, $Gain(P \cup \{h\}) - Gain(P)$, is selected, where P is the set of patterns selected before the start of this iteration. The algorithm terminates when k patterns are selected (line 4), or no candidate patterns have $Gain(P \cup \{h\}) - Gain(P)$ greater than $|Q|r$, where $|Q|r$ is the index-lookup cost (lines 6-8).

Algorithm 2 Greedy Algorithm for Pattern Mining

Input: Frequent Subgraph H
Output: k Index Patterns P, cost r
1: $P \leftarrow \emptyset$
2: //Objective function defined in Equation 5
3: Define $Gain(P) = \sum_{q \in Q} \sum_{g \in G} max_{p \in P} w(g, q, p)$
4: **while** $|P| < k$ **do**
5: Find the pattern $h \in H \setminus P$ with the maximum increase of the objective function, $Gain(P \cup \{h\}) - Gain(P)$
6: **if** increase $< |Q|r$ **then**
7: break
8: **else**
9: $P \leftarrow P \cup h$
10: **end if**
11: **end while**
12: return P

The approximation ratio of the greedy algorithm is $1 - 1/e$, which means $Gain(P_{gre}) \geq (1 - 1/e) Gain(P_{opt})$, where P_{gre} is the set of patterns found by the greedy algorithm and P_{opt} is the optimal solution. This is because the objective function $Gain(P)$ is a *monotone submodular* function. $Gain(P)$ is a monotonically increasing function because $Gain(P)$ will not decrease by indexing more patterns. Next, we prove $Gain(P)$ is submodular. See [6] for the proof of the approximation ratio.

CLAIM 2. *The Gain function is submodular, that is, $Gain(P \cup \{h\}) - Gain(P) \leq Gain(O \cup \{h\}) - Gain(O)$, given $O \subset P$ and $\forall h \notin P$.*

PROOF. Based on Equation 5,

$$\begin{aligned} &Gain(P \cup \{h\}) - Gain(P) \\ &= \sum_{q \in Q} \sum_{g \in G} (max_{p \in (P \cup \{h\})} w(g, q, p) - max_{p \in P} w(g, q, p)) \\ &= \sum_{q \in Q} \sum_{g \in G} max\{0, (w(g, q, h) - max_{p \in P} w(g, q, p))\}, \end{aligned}$$

because in $P \cup \{h\}$, if h does not achieve $max_{p \in (P \cup \{h\})} w(g, q, p)$ then some p in P provides it and the difference is 0. Similarly,

$$\begin{aligned} &Gain(O \cup \{h\}) - Gain(O) \\ &= \sum_{q \in Q} \sum_{g \in G} max\{0, (w(g, q, h) - max_{p \in O} w(g, q, p)\}. \end{aligned}$$

$max_{p \in P} w(g, q, p) \geq max_{p \in O} w(g, q, p)$, because $P \supset O$. Hence, $Gain(P \cup \{h\}) - Gain(P) \leq Gain(O \cup \{h\}) - Gain(O)$. □

Time complexity: Algorithm 2 computes the increase of the objective function for any $h \in H$ in each iteration of the while loop (lines 4-10), and it takes $O(|G||Q|)$ time to compute the increase. Therefore, the greedy algorithm has a time complexity of $O(k|H||G||Q|)$, which is not scalable to large datasets. In order to reduce the running time, we introduce an easy-to-compute function to approximate the objective function, so that the greedy algorithm optimizing the approximate objective function can be applied to large datasets.

4.3 An Approximate Objective Function

Computing $Gain(P)$ is time consuming because it is the summation of $w(g_i, q_j)$ for all $g_i \in G$ and $q_i \in Q$ (Equation 4). In this subsection, we propose an easy-to-compute score, $appGain$, to approximate the $Gain$ function. $appGain$ is defined with the distributions of queries instead of using the training queries explicitly. The increase of the approximate objective function $appGain(P \cup \{p\}) - appGain(P)$ (line 5 of Algorithm 2) can be computed in $O(|G|)$ time, which is much smaller than the $O(|G||Q|)$ time complexity of computing the increase of the $Gain$ function.

DEFINITION 5 (APPROXIMATE OBJECTIVE FUNCTION).

$$appGain(P) = \sum_{g \in G} w'(g) = \sum_{g \in G} max_{p \in P} w'(g, p) \quad (6)$$

$$w'(g,p) = \begin{cases} |Q|(Pr(q \supseteq p) \cdot c(p) + Pr(q \not\supseteq p) \cdot c(g)) & p \subseteq g \\ 0 & p \not\subseteq g \end{cases}$$

$Pr(q \supseteq p)$ is the probability that a query q contains a pattern p as a subgraph.

We explain the definition of $w'(g, p)$ by relating it to the function $w(g, q, p)$.

$$w'(g,p) = \sum_{q \in Q} w(g,q,p). \quad (7)$$

The derivation is as follows:

(1) when $p \not\subseteq g$, $w'(g,p) = \sum_{q \in Q} w(g,q,p) = 0$;

(2) when $p \subseteq g$,

$$\begin{aligned} w'(g,p) &= |Q|(Pr(q \supseteq p) \cdot c(p) + Pr(q \not\supseteq p) \cdot c(g)) \\ &= \sum_{q \supseteq p} c(p) + \sum_{q \not\supseteq p} c(g) = \sum_{q \in Q} w(g,p,q). \end{aligned}$$

However, $w'(g) \leq \sum_{q \in Q} w(g, q)$ because

$$\begin{aligned} w'(g) &= max_{p \in P} w'(g,p) = max_{p \in P} \sum_{q \in Q} w(g,q,p) \\ &\leq \sum_{q \in Q} max_{p \in P} w(g,q,p) = \sum_{q \in Q} w(g,q). \end{aligned}$$

Hence, $appGain(P) < Gain(P)$. However, we argue that patterns selected to maximize $appGain(P)$ are similar to those selected to optimize $Gain(P)$. The reasons are twofold: firstly, $w'(g,p) = \sum_{q \in Q} w(g,q,p)$. That is, $w'(g)$ equals to $w(g, q)$ if P contains non-overlapping patterns. Two patterns are non-overlapping if $\nexists (g, q)$ that is covered by both patterns with non-zero weight. Secondly, patterns selected are irredundant. For example, if two redundant patterns p_i and p_j have similar supporting sets on both G and Q, then, for each database graph g, the score $w'(g, p_i) \approx w'(g, p_j)$. Assuming p_i is first selected, then p_j will not be selected because the approximate objective will not increase by selecting p_j, given $\sum_{g \in G} max_{p \in P} w'(g,p) \approx \sum_{g \in G} max_{p \in P \cup \{p_j\}} w'(g,p)$.

This approximate objective function is a monotone submodular function, as can be shown similarly as Claim 2. Hence, the same greedy algorithm can be applied to optimize $appGain$ with an approximation ratio of $1 - 1/e$. The greedy algorithm runs with time $O(k|H||G|)$ and space $O(|H||G|)$.

4.4 A Memory-Efficient Algorithm

Algorithm 2 pre-computes and stores all frequent subgraphs H and their supporting sets in-memory for random access. This is memory inefficient and not scalable for large datasets, because the number of frequent subgraphs grows exponentially with the decrease of the minimum support. The worst-case space complexity of a pattern mining algorithm is $O(k|G|)$ because the memory should be big enough to hold all the k selected patterns and their value sets. In this section, we propose a memory-efficient algorithm, which only stores k index patterns in memory. The algorithm replaces an in-memory pattern p' with a new subgraph h if h is "better" than p' for filtering and prefix-sharing.

Algorithm 3 enumerates frequent subgraphs one after another according to the DFS code order [16]. The first k enumerated subgraphs are first selected and stored in memory. Then, for each new subgraph h, the algorithm computes the value $Gain(P \cup \{h\}) - Gain(P)$, where P is the k subgraphs in memory. The algorithm also finds an in-memory subgraph $p' \in P$, $p' = argmin_{p \in P}$ $(Gain(P) - Gain(P \setminus \{p\}))$. The algorithm replaces p' with h if $Gain(P \cup \{h\}) - Gain(P) > 2 \cdot (Gain(P) - Gain(P \setminus \{p'\}))$, or discards h otherwise. The factor 2 is set to archive a $1/4$ approximation ratio (see Appendix A). The algorithm enumerates the next subgraph and goes through the same procedure until no subgraphs can be enumerated.

Algorithm 3

Input: Current Database G, Recent Queries Q
Output: Selected Patterns P

```
1: A Subgraph Enumerator E
2: P ← The first k enumerated subgraphs
3: while E has next subgraph do
4:    h = E.nextsubgraph
5:    Inc ← appGain(P ∪ {h}) − appGain(P)
6:    Find p' = argmin_{p∈P}(appGain(P) − appGain(P \ {p})).
7:    if Inc > 2 · (appGain(P) − appGain(P \ {p'})) then
8:       Replace p' with h
9:    else
10:      Discard h
11:   end if
12: end while
```

Algorithm 3 is adapted a swap-based algorithm for the stream set-cover problem [11] and it has an approximation ratio of $1/4$. We prove the approximation guarantee in Appendix A. Although the worst-case approximation ratio of the memory-efficient algorithm is smaller than that of the greedy algorithm, $1/4 < 1 - 1/e$, their empirical performance are almost the same (Section 6).

Time Complexity: Algorithm 3 has a time complexity of $O(k \cdot s|G| + |H||G|)$. $O(|H||G|)$ time is spent on computing Inc for each frequent subgraph $h \in H$ (line 5). $O(k \cdot s|G|)$ time is spent on finding the pattern with the minimum score, where s is the number of times that the algorithm searches for the pattern with the minimum score (line 6). The algorithm recomputes the min-score pattern only after it replaces the min-score pattern with a newly enumerated pattern (line 8). Therefore, $s \ll |H|$.

Large Dataset: Given a large-scale database G that cannot fit in the memory, the memory-efficient algorithm is still time and space consuming to run. To overcome this difficulty, we propose

a two-step pattern-mining and index-construction algorithm: (1) we randomly pick a set of graphs $G' \subset G$ and mine index patterns P out of G' by Algorithm 2 or Algorithm 3, (2) we build the value sets of P on G. The rationale behind this strategy is that G' preserves the structural properties of the whole database G; hence, the index patterns mined out of G' are close to those mined from G [17]. *In addition, specifically for prefix sharing, each database graph g chooses its prefix $p_i \in P$ according to $c(p_i)$, $p_i = argmax_{p \subset g \wedge p \in P} c(p)$. Therefore, the prefix selection can be applied after the pattern-selection algorithm find the k index patterns.* Hence, the construction of the index can be split into the above two steps.

5. QUERY-WORKLOAD VERSUS NOQUERY

Previous approaches such as GPTree [20], PrefixIndex [21] and IGQuery [4] do not use any query workload. We will refer to such approaches as *NoQuery approaches*. In essence, these algorithms use the distribution of subgraphs in the database as a surrogate to having a query workload. Our algorithm can take a query workload and create an index based on that. Or, it can create the index by selecting patterns based on distribution of subgraphs in the database.

We acknowledge that the use of the training queries is a double-edged sword. On one hand, it can help mine better index patterns to process SPS queries faster than NoQuery approaches that do not use a query workload. On the other hand, using a query workload may cause overfitting. Also, when the query interest changes, patterns mined with the old training queries may not perform well on the new queries. However, we show the results for a "cold" start where a query workload is not available in Section 6. Our algorithm works reasonably well.

5.1 NoQuery Approaches

The patterns mined by the NoQuery approaches do not optimize the real query-processing time. The savings of the real query-processing time by using GPTree patterns and PrefixIndex patterns vary based on the query workload. The analysis on GPTree (significant subgraphs) can be applied on IGQuery patterns to show that the filtering gain of IGQuery is also query dependent.

Prefix-Sharing Gain: Both GPTree [20] and PrefixIndex [21] select patterns, P, to maximize the *prefix-sharing gain*, which is defined as $\sum_{p \in P}(|E(p)|-1)\cdot c(p)$, where $E(p)$ is the set of database graphs having p as the prefix [20, 21]. As each database graph chooses only one prefix [20, 21], it can be shown that $\sum_{p \in P}|E(p)| = |G|$. The prefix-sharing gain measures the savings enabled by prefix-sharing over the time taken to verify all database graphs G. In the filter+verify framework, most of the false graphs are filtered and only the candidate graphs are verified. Therefore, the above objective function failed to capture the real saving of the time by prefix sharing when filtering is also used. The candidate graphs verified using an isomorphism test depends upon the query workload. In other words, the set of index patterns mined without training queries does not optimize the real prefix-sharing gain and thereby does not create an optimal index.

Filtering Gain: GPTree selects *significant subgraph* patterns for filtering based on heuristics [20]. A pattern p_i is significant if $|G(p_i)| \geq \delta_{min}|\cup_{p_j \in P, p_j \supset p_i} G(p_j)|$. *Significant patterns are not mined to optimize the real filtering gains* because of two reasons. First, GPTree is constructed by only removing the redundancy between patterns with containment relationships. Hence, given two patterns p_1 and p_2 with similar supporting sets on both G and Q, and $p_1 \not\subseteq p_2 \wedge p_2 \not\subseteq p_1$, then GPTree will select both patterns for indexing, although selecting both will not save more verification time than selecting only one of the two. Second, the filtering gain of indexing significant subgraphs depends on the query distribution $Pr_Q(q)$. That is, given the complete set of frequent subgraphs H, the filtering gain of indexing all H is $filG = \sum_q Pr_Q(q)(|G|-|C(q)|)$, where $C(q)$ is the set of candidate graphs that need to be verified by SGIs. For exclusive-logic filtering, $C(q) = G - \cup_{p \in nosub(q,H)} G(p) = G - G(nosub(q,H))$, where $nosub(q,H) = \{p \in H, p \not\subseteq q\}$. Therefore,

$$filG = \sum_q Pr_Q(q)|G(nosub(q,H))| = \sum_{\forall F \subset H} Pr_Q(F)|G(F)|,$$

where F is a subset of patterns and $Pr_Q(F)$ denotes the probability that a query graph q uses patterns F for filtering.

It is not practical to index all frequent subgraphs. Instead, a small set of patterns P is mined for indexing. Correspondingly, instead of F, $F_P = F \cap P$ is used for filtering. Inevitably, the verification time increases because the candidate set generated by F_P is not as tight as that generated by F. Significant subgraphs are mined to minimize the verification time increase heuristically.

$$filG(H) - filG(P) = \sum_{\forall F \subset H} Pr_Q(F)|G(F)|(1 - \frac{|G(F_P)|}{|G(F)|})$$

GPTree removes insignificant pattern p from F if $|G(p)| < \delta_{min}|\cup_{p_j \in P, p_j \supset p} G(p_j)|$, which is a heuristic to confine $\frac{|G(F_P))|}{|G(F)|} > {\delta_{min}}^{-n}$ when $|F - F_P| = n$. Therefore,

$$filG(H) - filG(P) < \sum_{\forall F \subset H} Pr_Q(F)|G(F)|(1 - {\delta_{min}}^{-n})$$

Thus, both the increase of the verification time and its upper bound depend on the underlying query distribution Pr_Q. Therefore, the real filtering gain of the GPTree patterns is query dependent.

5.2 Training Queries

No training queries: When there are no training queries available, it is common to use the graph database as a surrogate of the training queries, based on the assumption that the database graphs have similar statistical distributions to the queries. This strategy has been prevalently used in graph pattern mining [14, 3, 19], database tuning and other related fields and can also be used with our method.

Overfitting: To avoid overfitting index patterns to a special set of queries, our objective function is defined by $Pr(q \supset p)$, the probability of a query containing a pattern (Equation 6). Hence, instead of fitting to a set of training queries, our index patterns are mined to fit to a class of queries that have similar statistical properties.

Change of query interest: When the query interest changes, index patterns mined with the old training queries may not perform well on the new queries. In this case, the index patterns need to be re-mined to accommodate the query change. One key challenge is to monitor queries and decide the time to invoke an update of the index. $Pr(q \supset p)$ can be used to monitor the shift of the query interest. When the probability in the recent queries q_{new} differs largely from that in the training query workload, the index patterns need to be re-mined. KL divergence can be used to measure the differences. Monitoring the divergence does not incur significant extra cost except for maintaining the probability $Pr(q \supset p), \forall p \in P$. Yuan, et al. have proposed pattern and index updating algorithms for subgraph search [19]. Similar approaches can be applied to update supergraph search patterns. The pattern update algorithm is beyond the scope of this paper and we plan to study in the future work.

When the query interest changes, the real time savings of the NoQuery approaches [20, 21, 4] vary based on the query workload (see Section 5.1). Because the NoQuery approaches do not model query distribution, they can not be updated to accommodate the change of the query interest.

6. EXPERIMENT

We compare LW-index with cIndex [3], GPTree [20], and PrefixIndex [21]. We focus on studying the performance of single queries and hence, do not study IGQuery because it outperforms other indexes mainly on batch-query processing [4]. We evaluate both cIndex-BottomUp (c(BU)) and cIndex-TopDown(c(TD)). For LW-index, we evaluate both the greedy algorithm (Algorithm 2) and the memory-efficient algorithm (Algorithm 3) and denote them as LW(G) and LW(M) respectively. We use the greedy algorithm, LW(G), by default on moderate-sized datasets. We use LW$_k$ to denote the case when k is specified.

Parameters: For cIndex-TopDown, we set the number of training queries in each leaf node to be 100, as suggested in the original work [3]. For PrefixIndex, we consider the 2-level Hierarchical PrefixIndex. The exact pattern-mining algorithm is used in GPTree. The index-lookup cost of our algorithms, r, is set to 2 except when noted differently.

Dataset: We test our algorithm on four datasets. (1) The AIDS dataset [20, 21, 3] contains more than 40,000 chemical molecules. First, we mine frequent subgraphs from the AIDS dataset with a minimum support of 0.5% and obtain more than $500,000$ subgraphs. Then, we uniformly sample 5 databases from the subgraphs, with size $20,000$, $40,000$, $60,000$, $80,000$ and $100,000$ respectively. We randomly pick $10,000$ graphs from the AIDS dataset as training queries, and pick another $2,000$ for testing. This setting is similar to that in related works [3, 4, 20, 21]. (2) To study the scalability of LW-index on large graphs, we construct a graph database with the AIDS graphs and generate queries by merging the AIDS graphs using a graph integration algorithm [4]. Each query graph is generated by first randomly selecting n database graphs, where $n \sim \mathcal{N}(6, 2)$, and then merging them into a query graph. $10,000$ training queries and $2,000$ test queries are generated in this way with an average edge count of 216 and an average node count of 184. Each query graph contains more than 40 AIDS graphs as subgraphs. (3) We also study the system call dependency graphs of 2631 malware samples [1] with nodes representing system calls and directed edges representing dependencies. We mine frequent subgraphs with the minimum support $\delta = 0.05$ and then randomly sample $40,000$ to create the database. We generate 14973 queries by partitioning each dependency graph into densely connected communities. We randomly pick $8,000$ queries for training and another $2,000$ for testing. (4) We randomly generate four synthetic graph datasets with density $0.2, 0.3, 0.4$ and 0.5 respectively, where the density is defined as the ratio between the edge count and the square of node count [5]. For each dataset, we select $8,000$ graphs as training queries and $2,000$ for testing. Then, we construct a $40,000$-graph database for each dataset by randomly picking frequent subgraphs mined from the synthetic graphs with the minimum support $\delta = 0.05$.

We implement GPTree, PrefixIndex, and cIndex within the same framework for fair comparison. We store the value sets in memory for fast index-lookup. We use the ParMol package [9] to parse chemical molecules and mine frequent subgraphs. We tested the pattern-mining algorithms on a machine with 32G RAM, and the indexes are tested with a maximum heap size of 4G.

6.1 AIDS Small Graphs

6.1.1 LW-index Vs. Others

In this subsection, we mine index patterns on the dataset G_4 (40,000 graphs) with minimum support 0.01.

Table 2: Index Construction on G_4 with MinSup 0.01

Index	Freq Time	pattern Count	Mine Time	Mine Space
LW-index	60 s	641	8.3 s	11.3 MB
PrefIndex	30 s	706/87	10.5 s	11.2 MB
GPTree	30 s	972/228/1267	6.2 s	11.2 MB
c(BU)	60 s	264	1807 s	64 MB
LW-index$_k$	60 s	264	4.1 s	11.3 MB
c(TD)	60 s	1130	455 s	48MB

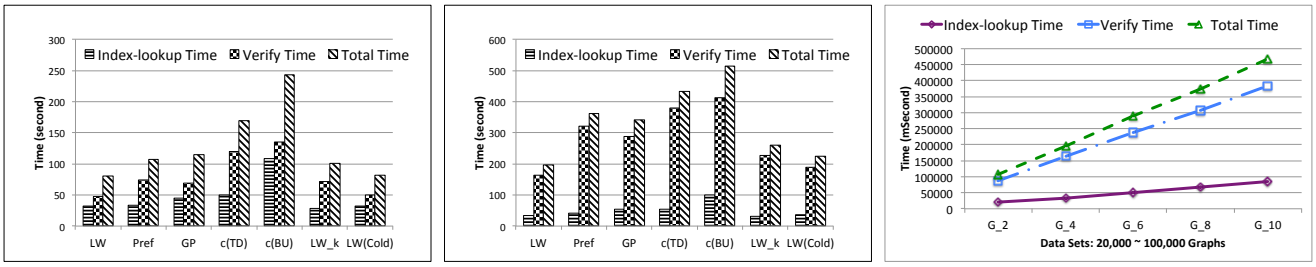
Pattern Mining: Table 2 shows the performance of various indexes on pattern mining. We mine the frequent subgraphs using the GSpan algorithm [16] and found 1511 frequent subgraphs. The frequent-subgraph-mining time is shown in Column 2. cIndex(BU/TD) and LW-index take longer than GPTree and PrefixTree. This increase is because cIndex(BU/TD) and LW-index also compute the containment relationship between the frequent subgraphs and the training queries. Column 3 shows the number of selected index patterns. For PrefixIndex, 706 patterns were selected for the base-level index and 87 more patterns were selected for the second-level index. For GPTree, 972 significant patterns were mined to build the CRGraph, 228 patterns were mined to build the FGPTree and 1267 patterns were mined as prefix patterns in GPTree. c(BU) mines patterns with $r = |Q|$ (as specified in the original work [3]), so that the mining algorithm stops when none of unselected subgraphs can increase the objective function by r. As a result, 264 patterns were selected. To make the results comparable, we also build LW-index$_k$ with $k = 264$ index patterns (same as c(BU)). Column 4 shows the running time of the pattern-mining algorithms. LW-index has comparable running time with PrefixIndex and GPTree. c(BU) runs 200 times slower than LW-index on pattern selection. c(TD) runs faster than c(BU) on pattern selection, and this is coherent to the original work [3]. Column 5 shows the memory consumption of the pattern selection. LW-index, PrefixIndex and GPTree have similar memory costs because they all have a space complexity of $O(|H||G|)$.

Query Processing: Figure 3(a) shows the time taken to process $2,000$ test queries, which have an average edge count of 25. LW-index and LW-index$_k$ have the smallest verification time, which proves the effectiveness of our pattern selection algorithm. In addition, LW-index has the smallest index-lookup time. Figure 3(b) shows the time taken to process $2,000$ test queries, which have an average edge count of 35. In comparison to Figure 3(a), the verification time increases on larger queries and it dominates the overall query-processing time. The 35-edge test queries and the 25-edge training queries have different edge counts, but they are sampled from the same dataset and they have similar probability of containing subgraphs. Therefore, the index patterns mined with edge-25 training queries work well on queries with 35 edges. LW-index is the fastest on query processing in comparison to other indexes.

Cold Start: We also mine index patterns with a cold start when no training queries are available. LW-index(Cold) is constructed with those index patterns. As shown in Figure 3(a) and 3(b), LW(Cold) has similar index-lookup time to LW-index. In addition, the verification time for LW(Cold) is only slightly higher than that of LW-index, both of which are much smaller than that of other indexes.

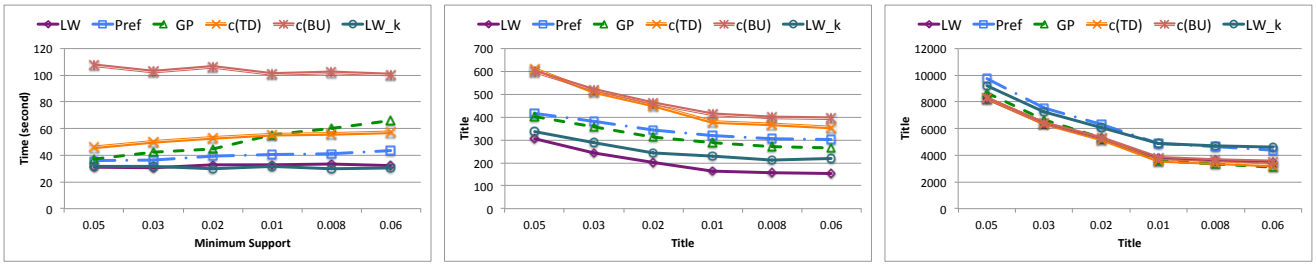
6.1.2 Varing-MinSupport Patterns

We mine index patterns with minimum support, δ, ranging from 0.006 to 0.05.



(a) dataset G_4, 25-edge queries (b) dataset G_4, 35-edge queries (c) 35-edge queries

Figure 3: Performance of Index Patterns Mined with MinSupport 0.01 on the AIDS Small Datasets



(a) Index-Lookup Time (b) Verification Time (c) Number of Verified Graphs

Figure 4: Performance of Index Patterns Mined with Different MinSupports on G_4, 35-edge Testing Queries

(1) LW-index (both LW and LW$_k$) maintains a much smaller index-lookup time than other indexes over different δ, as shown in Figure 4(a).

(2) In addition, as shown in Figure 4(b), LW-index has the smallest verification time of all the indexes and the verification time decreases with the decrease of δ as more subgraph patterns are selected for indexing. The reasons for the small verification cost are twofold: first, LW-index patterns can prune more false graphs than other patterns, as shown in Figure 4(c); second, LW-index patterns are effective in prefix indexing candidate graphs, so that the verification time is further reduced. GPTree filters a comparable number of false answers as LW-index (Figure 4(c)). However, GPTree indexes a large amount of *significant subgraphs* and thus its index-lookup time is high, as can be seen in Figure 4(a) and Figure 5(a).

(3) LW$_k$ achieves the second smallest verification time (Figure 4(b)). Although, as shown in Figure 4(c), LW$_k$ does not filter as many false graphs as cIndex(TD/BU), the verification time of LW$_k$ is much lower than that of cIndex. This is because LW$_k$ also optimizes the prefix-sharing gain, which is not considered in cIndex(TD/BU). This observation further proves the benefit of the prefix-sharing methods. In addition, LW$_k$ filters less false graphs than GPTree, because it indexes much less patterns than GPTree. For example, 3468 patterns are indexed by GPTree at $\delta = 0.006$, in comparison to the 283 patterns indexed by LW$_k$. However, the verification time of LW$_k$ is lower than that of GPTree. This reduction is because LW$_k$ optimizes the prefix-sharing gain over candidate graphs, but GPTree optimizes the prefix-sharing gain over all database graphs (Section 5). In the filter+verify framework, only the candidate graphs need to be verified. Thus, patterns mined by GPTree do not perform as well as those mined by LW-index.

(4) PrefixTree does not filter as many false graphs as other methods, as shown in Figure 4(c), because the index patterns are mined considering only the prefix-sharing gain. The prefix-sharing gain is measured over all database graphs and not only on candidate graphs. Correspondingly, the verification time of PrefixTree is higher than that of GPTree, LW-index and LW$_k$, as shown in Figure 4(b).

6.1.3 *Varying-sized Datasets*

Table 3: Index Large Datasets

Data	Selected patterns	Selection Time	Space	Lucene Time
G_2	479	3 s	6MB	26s
G_6	762	14 s	12MB	78s
G_8	861	20s	22MB	111s
G_{10}	927	24 s	28MB	143s

We study the performance of LW-index on graph databases of different sizes. We mine index patterns with a minimum support of 0.01, and obtain 1500 to 1600 frequent subgraphs. Table 3 shows the cost for index pattern mining. We show the results of the experiments using dataset G_4 in Table 2. As can be seen, the running-time and space cost of the greedy pattern-mining algorithm is linear with respect to the size of the database. Figure 3(c) shows the time for processing 2000 queries that have an average edge count of 35. The query-processing time is also linear with respect to the size of the graph database.

6.2 AIDS Large Graphs

In this section, we study large database graphs. We set the minimum support as $\{0.05, 0.03, 0.02, 0.01, 0.008\}$. Correspondingly, 4098, 9873, 18629, 51353, and 69777 frequent subgraphs are mined. The number of frequent subgraphs grows significantly with the decrease of the minimum support. This shows the necessity of pattern mining and selection. We omit c(BU) because it does not perform as well as c(TD) as shown in the previous experiments and related work [20].

6.2.1 *LW-index Vs. Others*

Pattern Mining: Table 4 shows the time costs of index pattern mining. The time for frequent subgraph mining is shown in Column 3. Column 4 shows the number of selected index patterns.

Table 4: Pattern Mining on Large Graphs with Diff δ

δ	Index	T	Selected Count	Time
0.05	LW-index	1,753 s	349	45 s
	PrefIndex	206 s	442/44	34 s
	GPTree	206 s	2227/99/1247	51 s
	c(TD)	1,753 s	3072	6,982s
0.03	LW-index	3225 s	464	90 s
	PrefIndex	324 s	530/61	99 s
	GPTree	324 s	5524/184/1883	303 s
	c(TD)	3225	4839	12860s
0.02	LW-index	4,635 s	539	171 s
	PrefIndex	403 s	602/70	217 s
	GPTree	403 s	10759/298/2334	1,126 s
	c(TD)	4,635 s	5809	19,683 s
0.01	LW-index	7,686 s	731	310 s
	PrefIndex	561 s	664/100	679 s
	GPTree	561 s	31144/632/3149	8,472 s
	c(TD)	7,686 s	6713	36,319 s
0.008	LW-index	8,838 s	794	381 s
	PrefIndex	612 s	702/107	1,007 s
	GPTree	612 s	43002/800/3370	15,916 s
	c(TD)	8,838 s	6900	43,582 s

Similar to that in Table 2, for PrefixIndex, the two numbers together denote the pattern count in the two-level indexes. As can be seen, both LW-index and PrefixIndex selects index patterns judiciously, so that the number of index patterns grows stably with the decrease of the minimum support, although the total number of frequent subgraphs increases significantly. On the contrary, the number of index patterns mined by GPTree increases significantly with the decrease of the minimum support. This increase leads to a cumbersome index, which is costly to construct and lookup with respect to time. The running time for pattern selection algorithms is shown in Column 5. LW-index is faster than the other algorithms. c(TD) is almost 200 time slower than LW-index on pattern selection. GPTree mines significant and frequent subgraphs by first finding the containment relationships of all frequent subgraphs. It is generally slow given large number of frequent subgraphs. LW-index is 2 to 3 times faster than PrefixIndex in practice. This is because the PrefixIndex constructed here is a 2-level hierarchy index, and the running time also includes the frequent subgraph mining and pattern selection time for the level-2 patterns.

Query Processing: As seen in Figure 5(a), the index-lookup time increases for all indexes with the decrease of the minimum support. Figure 5(b) shows the decrease of the verification time for all indexes with increasing number of index patterns. Figures 5(a) and 5(b) show that indexing more patterns reduces the verification time but increases the index-lookup time. LW-index balances these two time components using the index-lookup penalty r. For indexes storing the value sets on disk, r should be set to a larger value because the value-lookup time increases with intensive disk operations. For GPTree, because the significant and frequent patterns are selected based on heuristics, the number of selected patterns is hard to control. As shown in Figure 5(a), the index-lookup time for GPTree grows significantly when a large number of index patterns are selected at low minimum support. Although the verification time decreases with more indexing patterns, it does not compensate for the increase of the index-lookup time. Hence, the overall query-processing time of GPTree increases significantly, as shown in Figure 5(c).

6.2.2 *Greedy Vs. Memory-Efficient*

In this section, we study the performance of the two pattern selection algorithms, the greedy algorithm (LW(G)) and the memory-efficient algorithm (LW(M)) on the large dataset.

Memory: As can be seen in Figure 6(a), the number of frequent subgraphs, $|H|$, grows significantly when minimum support δ decreases. Accordingly, the memory usage of LW(G) grows significantly because the greedy algorithm has a $O(|H||G|)$ space complexity. However, the number of selected patterns, k, does not change significantly, as in Table 4. As a result, the memory usage of the memory-efficient algorithm only grows slightly, as shown in Figure 6(a). This slow growth is because LW(M) has a theoretical space complexity of $k|G|$.

Time: Recall that the greedy algorithm has a time complexity of $O(k|H||G|)$ and the memory-efficient algorithm has a time complexity of $O((k \times s + |H|)|G|)$, where s is the number of times that the algorithm replaces a selected pattern with a new subgraph (section 4.4). The empirical running time of the two algorithms is consistent with their theoretical time complexity, as shown in Figure 6(b). The running time of LW(M) grows at a much slower rate than the running time of LW(G) because $k \times s + |H| \ll k|H|$, given that $s \ll |H|$.

Pattern Quality: Although the memory-efficient algorithm only has an approximation ratio of $1/4$, which is smaller than the $1 - 1/e$ approximation ratio of the greedy algorithm, the index patterns mined by the memory-efficient algorithm perform as well as those mined by the greedy algorithm on filtering and prefix-sharing, as shown in Figure 5(b).

6.3 Other Datasets

Figure 7 shows the performance of our algorithm on the **malware dataset**. Index patterns are mined with the minimum support $\delta = 0.02$. LW-index outperforms all other methods on the index-lookup time. LW-index has similar verification time as GPTree. However, because GPTree indexes many more patterns than LW-index, its index-lookup time is significantly higher than that of LW-index. As a result, GPTree has the longest query-processing time.

We also study the performance of LW-index on **synthetic datasets**. We change the graph density from 0.2 to 0.5, where the density is defined as the ratio between the edge count to the square of the node count. Figure 8 shows the index-lookup time (F), verification time (V) and the total query-processing time (T) of LW-index, PrefixIndex and GPTree separately. As can be seen, LW-index has the smallest index-lookup time and total query-processing time. Although GPTree has the smallest verification time, its total query-processing time is higher than that of LW-index because its index-lookup time is significantly longer than that of LW-index. PrefixIndex has the longest verification time because it does not select index-patterns to optimize the filtering gain. These observations are consistend with what we observed on the AIDS dataset.

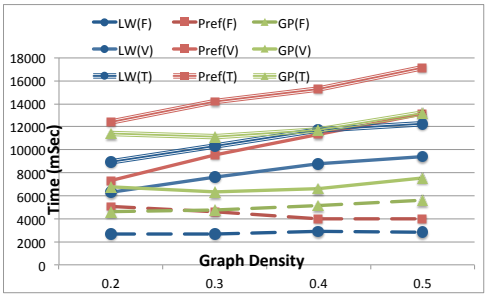
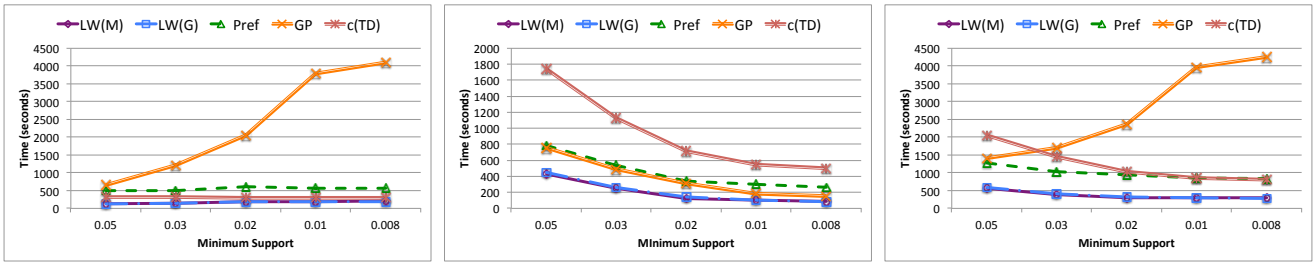


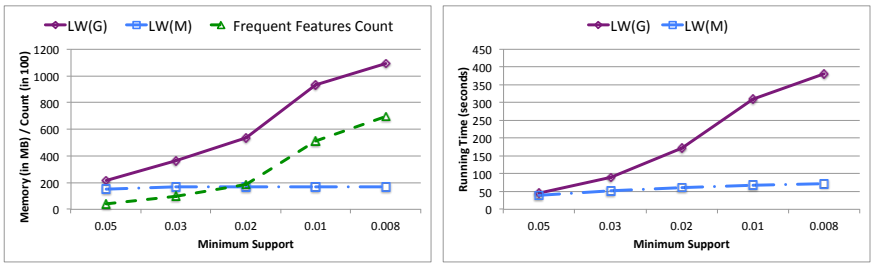
Figure 8: Synthetic Dataset with Varying Density

7. RELATED WORKS



(a) Index-Lookup Time (b) Verification Time (c) Total Time

Figure 5: Performance of Index Patterns Mined on Large Graphs with Different minSupports



(a) Memory for Pattern Mining (b) Time for Pattern Mining

Figure 6: The Greedy Algorithm and The Memory-Efficient Algorithm

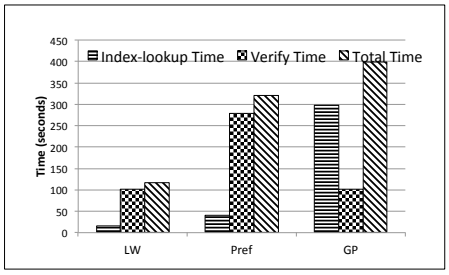


Figure 7: Malware Dataset, MinSupport 0.02

Chen, et.al, first modeled the pattern-mining problem for SPS as a NP-hard optimization problem [3]. They proposed a greedy algorithm to solve the problem with an approximation ratio of $1 - 1/e$. This model has three drawbacks. First, it only measures the filtering gain; it does not consider the prefix-sharing gain. Second, the model assumes that all SGIs take the same amount of time. Clearly, verifying a 10-edge graph takes more time than verifying a 2-edge graph. Third, the time complexity for the greedy pattern-selection algorithm is $O(|H||G||Q|)$, which is not scalable to large datasets and query sets. Zhang, et al., proposed a heuristic-based algorithm, selecting *significant subgraph* patterns for indexing [20]. A pattern p_i is significant if $|G(p_i)| \gg |\cup_{p_j \in P, p_j \supset p_i} G(p_j)|$. Although fast, this heuristic-based pattern-mining algorithm only removes the redundancy between patterns with containment relationships. In addition, the number of selected patterns increases significantly when the number of candidate patterns H increases, because all the *maximal frequent subgraph* patterns are selected. A frequent subgraph sg is *maximal* if no supergraphs of sg are frequent.

Zhang, et al. [20], and Zhu, et al. [21], proposed to solve the SSP problem using prefix sharing. The query-processing time depends on the prefix assigned to each database graph. Zhang, et al., proved that the pattern-mining problem optimizing the prefix-sharing gain is NP-hard and proposed a greedy algorithm with an approximation ratio of $1/2$ [20]. Zhu, et al., further improved the algorithm empirically by using a different greedy metric [21]. However, the prefix-sharing gain that both algorithms optimize is measured over all database graphs, and the real prefix-sharing gain should be measured over candidate graphs only, because the candidate graphs (not the whole graph database) are verified in the query-processing algorithm (see Section 5).

Cheng, et al., proposed a fast way of processing SPS queries in batches [4]. Given a batch of queries Q, the algorithm first identifies the common subgraphs of Q, $SubQ$, and the common supergraphs of Q, $SupQ$. Based on the definition of the SPS, the answer $An(q)$ for each query $q \in Q$ is lower bounded by $An(SubQ)$ and upper bounded by $An(SupQ)$, given $subQ \subseteq q \subseteq supQ$. Hence, the batch queries can be solved by first finding $An(SubQ)$ and $An(SupQ)$, and then calculating $An(q)$ for each query q, so that candidate $g \in An(SubQ) \subseteq An(q)$ can be directly included in the answer set without verification and $g \notin An(SupQ)$ can be pruned, thereby significantly reduces the number of SGIs. This strategy can be applied to other algorithms, including LW-index. Shang, et al., extended the SPS to similarity search and proposed a no-pattern index, a global SG-Enum index, to facilitate the similarity search [13].

8. CONCLUSION

We study the problem of selecting a set of optimal features for an index in a graph database to facilitate answering supergraph search queries. We address this pattern-mining problem so as to optimize the savings of the query-processing time using both filtering and prefix-sharing. Two algorithms are proposed to solve the problem with approximation guarantees. In addition, we introduce an LW-index to reduce the time of index-lookup. We show that the LW-index with index patterns mined by our algorithms outperforms other existing methods, including cIndex [3], PrefixIndex [21], and GPTree [20], on processing SPS queries on a query workload over an existing benchmark dataset.

9. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 0845487 and by the Dow Chemical Company.

10. REFERENCES

[1] D. Babić, D. Reynaud, and D. Song. Malware Analysis with Tree Automata Inference. In *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 116–131, 2011.

[2] N. Bruno and S. Chaudhuri. Exploiting statistics on query expressions for optimization. SIGMOD, pages 263–274, New York, NY, USA, 2002.

[3] C. Chen, X. Yan, P. S. Yu, J. Han, D.-Q. Zhang, and X. Gu. Towards graph containment search and indexing. VLDB, pages 926–937. VLDB Endowment, 2007.

[4] J. Cheng, Y. Ke, and W. Ng. Efficient query processing on graph databases. *TODS*, 34:2:1–2:48, 2009.

[5] J. Cheng, Y. Ke, W. Ng, and A. Lu. Fg-index: Towards verification-free query processing on graph databases. In *SIGMOD*, pages 857–872, 2007.

[6] M. Fisher, G. Nemhauser, and L. Wolsey. An analysis of approximations for maximizing submodular set functions. In M. Balinski and A. Hoffman, editors, *Polyhedral Combinatorics*, volume 8 of *Mathematical Programming Studies*, pages 73–87. 1978.

[7] 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, 2010.

[8] L. Ma, Z. Huang, and Y. Wang. Automatic discovery of common design structures in cad models. *Computers and Graphics*, 34(5):545 – 555, 2010.

[9] T. Meinl, M. Wrlein, O. Urzova, I. Fischer, and M. Philippsen. The parmol package for frequent subgraph mining. *ECEASST*, 1, 2006.

[10] S. Nowozin, K. Tsuda, T. Uno, T. Kudo, and G. Bakir. Weighted substructure mining for image analysis. In *CVPR*, pages 1–8, 2007.

[11] B. Saha and L. Getoor. On maximum coverage in the streaming model & application to multi-topic blog-watch. In *SDM*, pages 697–708, 2009.

[12] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proc. VLDB Endow.*, pages 364–375, 2008.

[13] H. Shang, K. Zhu, X. Lin, Y. Zhang, and R. Ichise. Similarity search on supergraph containment. In *ICDE*, pages 637–648, 2010.

[14] B. Sun, P. Mitra, and C. L. Giles. Independent informative subgraph mining for graph information retrieval. In *CIKM*, pages 563–572, 2009.

[15] W. T. Wipke and D. Rogers. Artificial intelligence in organic synthesis. sst: starting material selection strategies. an application of superstructure search. *J Chem Inform Comput Sci*, 24(2):71–81, 1984.

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

[17] X. Yan, P. S. Yu, and J. Han. Graph indexing: a frequent structure-based approach. In *SIGMOD*, pages 335–346, 2004.

[18] D. Yuan and P. Mitra. Lindex: a lattice-based index for graph databases. *The VLDB Journal*, 22(2):229–252, 2013.

[19] D. Yuan, P. Mitra, H. Yu, and C. L. Giles. Iterative graph feature mining for graph indexing. In *ICDE*, pages 198 – 209, 2012.

[20] S. Zhang, J. Li, H. Gao, and Z. Zou. A novel approach for efficient supergraph query processing on graph databases. In *EDBT*, pages 204–215, 2009.

[21] G. Zhu, X. Lin, W. Zhang, W. Wang, and H. Shang. Prefindex: an efficient supergraph containment search technique. In *SSDBM*, pages 360–378, 2010.

[22] Q. Zhu, Yao, Yuan, F. Li, Chen, W. Cai, and Q. Liao. Superstructure searching algorithm for generic reaction retrieval. *JCIM*, 45(5):1214–1222, 2005.

APPENDIX

A. APPROXIMATION GUARANTEE OF THE MEMORY-EFFICIENT ALGORITHM

We show that Algorithm 3 has an approximation ratio at least $1/4$. Our proof strategy follows from that in Section 2.4 [11]. First, let each subgraph pattern p be modeled as a set s_p, covering items (g, q) with weight $w(g, q, p)$ if $w(g, q, p) > 0$, as in the proof of Claim 1. Let $P_t = \{p_1, p_2, \ldots, p_k\}$ be the in-memory patterns selected by Algorithm 3 at iteration t. We omit t when its value can be derived from the context. We define $r_{p_j} = \{(g, q) \in s_{p_j} | w(g, q, r_{p_j}, P) > 0\}$, where $w(g, q, r_{p_j}, P) = w(g, q, p_j) - max_{p_i \in P \wedge i \neq j} w(g, q, p_i)$. That is, the set r_{p_j} includes all (g, q) elements, the weight of which decreases from $w(g, q, p_j)$ to $max_{p_i \in P \wedge i \neq j} w(g, q, p_i)$ after p_j is removed from P. Hence, $w(P) - w(P \setminus \{p_j\}) = w(r_{p_j}) = \sum_{(g,q) \in r_{p_j}} w(g, q, r_{p_j}, P)$, where $w(P)$ is the $Gain$ function for patterns P. Similarly, for a new pattern h, we define $b_h = \{(g, q) \in s_h | w(g, q, b_h, P) = w(g, q, h) - max_{p_i \in P} w(g, q, p_i) > 0)\}$. Accordingly, $w(P \cup \{h\}) - w(P) = w(b_h) = \sum_{(g,q) \in b_h} w(g, q, b_h, P)$. Without loss of generality, we assume the optimal sets are pair-wise disjoint, hence $w(O) = \sum_{j=1}^{k} w(o_j)$.

We define element charge and set charge similar to [11]. *Element charge:* When an optimal set o_i comes, a weight of $w(g, q, b_{o_i}, P)$ is charged on each of its element if o_i is selected. Otherwise, a weight of $w(g, q, b_{o_i}, P)$ is charged on each element of $\bigcup_{s_j \in S} s_j \cap o_i$. *Set charge:* If the optimal set o_i is not selected by the algorithm, charge each element of $r_{p_j}, \forall p_j \in P$ with $\frac{w(b_{o_i}) w(g,q,r_{p_j},P)}{w(P)}$.

LEMMA 1. *Each in-memory set s_{p_j} gets a set charge of at most $\frac{2w(r_{p_j})}{k}$ for each optimal set o_i which is not selected by Algorithm 3. The total set-charge in the final selected patterns P is at most $2w(P)$.*

PROOF. By definition, $r_{p_j} \subseteq s_{p_j}$ and $r_{p_j} \cap r_{p'_j} = \emptyset$ for $j \neq j'$. As o_i is not selected, we have, $w(b_{o_i}) \leq 2 \cdot w(r_{p_j}), \forall p_j \in P$. Hence, $k \cdot w(b_{o_i}) \leq 2 \sum_{j=1}^{k} w(r_{p_j})) \leq 2w(P)$. Therefore, any element $e \in \bigcup_{j=1}^{k} r_{p_j} \setminus o_i$, set-charge on e is $\frac{w(b_{o_i}) w(g,q,b_{o_i},P)}{w(P)}$, which is at most $\frac{2w(g,q,b_{o_i},P)}{k}$. Set-charge on s_{p_j} is at most $\sum_{e \in r_{p_j} \setminus o_i} \frac{2w(g,q,r_{p_j},P)}{k} \leq \frac{2w(r_{p_j})}{k} \leq \frac{2w(s_{p_j})}{k}$.

Transfer of set-charge follows a similar way as [11]: (1) If some set h replaces s_{p_j} in later iterations, for each element $(g, p) \in s_{p_j} \setminus r_{p_j}$, the set charge on s_{p_j} for (g, p) can be transferred to $s_{p'_j}$, where $w(g, p, s_{p'_j}) = w(g, p, s_{p_j})$. After the charge transfer, the charge on the set $s_{p'_j}$ is at most $2w(r_{p'_j})/k$. (2) Transfer the rest of the set charge on s_j to h. Because $w(b_h) \geq w(r_{p_j})$, the set charge transferred to h is less than $w(r_{p_j})$, herein less than $w(b_h) \leq w(r_h)$. Therefore, the inequality holds after set-charge transfer. All in all, the total set-charge is at most $\sum_{i=1}^{k} \frac{2w(r_{p_j})}{k} \leq 2w(P)$. □

LEMMA 2. *The total element-charge in the final selected patterns P is at most $2w(P)$.*

PROOF. Let P_0 be the first k subgraph selected by Algorithm 3 and P_f be the final set of subgraphs selected. Then $w(P_f) = w(P_0) + \sum_{t \geq 1} (w(b_{h_t}) - w(r_{p'_t}))$, where h_t is the subgraph selected in iteration t and p'_t is the in-memory subgraph swapped out. Notice that $w(b_{h_t}) \geq 2w(r_{p'_t})$. Hence, $\sum_{t \geq 1} w(r_{p'_t})) \leq w(P_f) - w(P_0) < w(P_f)$. Therefore, the element-charge on removed elements is at most $w(P_f)$. Moreover, the final cover can have an additional element-charge of at most $w(P_f)$. □

Combining Lemma 1 and Lemma 2, we know that, the total set-charge and element-charge of the final cover is at most $4w(P)$. Moreover, it is at least $w(O)$. Hence, $w(O) \leq 4w(P)$. We conclude that the memory-efficient algorithm has an approximation ratio $1/4$.