Towards Graph Containment Search and Indexing *

Chen Chen[1] Xifeng Yan[2] Philip S. Yu[2] Jiawei Han[1] Dong-Qing Zhang[3] Xiaohui Gu[2]

[1]University of Illinois at Urbana-Champaign
{cchen37, hanj}@cs.uiuc.edu
[2]IBM T. J. Watson Research Center
{xifengyan, psyu, xiaohui}@us.ibm.com
[3]Thomson Research
{dong-qing.zhang}@thomson.net

ABSTRACT

Given a set of model graphs $\mathcal{D}$ and a query graph q, *containment search* aims to find all model graphs $g \in \mathcal{D}$ such that q contains g ($q \supseteq g$). Due to the wide adoption of graph models, fast containment search of graph data finds many applications in various domains. In comparison to *traditional graph search* that retrieves all the graphs containing q ($q \subseteq g$), containment search has its own indexing characteristics that have not yet been examined. In this paper, we perform a systematic study on these characteristics and propose a *contrast subgraph*-based indexing model, called **cIndex**. Contrast subgraphs capture the structure differences between model graphs and query graphs, and are thus perfect for indexing due to their high selectivity. Using a redundancy-aware feature selection process, cIndex can sort out a set of significant and distinctive contrast subgraphs and maximize its indexing capability. We show that it is NP-complete to choose the best set of indexing features, and our greedy algorithm can approximate the one-level optimal index within a ratio of $1 - 1/e$. Taking this solution as a base indexing model, we further extend it to accommodate hierarchical indexing methodologies and apply data space clustering and sampling techniques to reduce the index construction time. The proposed methodology provides a general solution to containment search and indexing, not only for graphs, but also for any data with transitive relations as well. Experimental results on real test data show that cIndex achieves near-optimal pruning power on various containment search workloads, and confirms its obvious advantage over indices built for traditional graph search in this new scenario.

1. INTRODUCTION

Graph data is ubiquitous in advanced data applications, including the modeling of wired or wireless interconnections (communication), 2D/3D objects (pattern recognition), chemical compounds or protein networks (chem/bio-informatics), circuits (computer-aided design), loosely-schemaed data (XML), social or informational networks (Web), *etc.*. With the tremendous amount of structured or networked data accumulated in large databases, supporting scalable graph search becomes an emerging database system research problem.

Given a graph database and a graph query, one could formulate two basic search problems: (1) (*traditional*) *graph search*: Given a graph database $\mathcal{D} = \{g_1, \ldots, g_n\}$ and a graph query q, find all graphs g_i in $\mathcal{D}$ s.t. q is a subgraph of g_i; and (2) *graph containment search*: Given a graph database $\mathcal{D} = \{g_1, \ldots, g_n\}$ and a graph query q, find all graphs g_i in $\mathcal{D}$ s.t. q is a supergraph of g_i.

The first problem, *i.e.*, graph search, including both exact and approximate search, has been extensively studied in the database community. For example, there are many indexing methods proposed to optimize XML queries (usually based on path expressions) against tree-structured data [9, 20]. In order to handle more complex graph queries, GraphGrep [21] and gIndex [25] use graph fragments such as paths and small subgraphs as indexing features. Both of them can achieve promising performances. Based on GraphGrep and gIndex, approximate graph search algorithms are also successfully developed, *e.g.*, Grafil [26]. The key idea of these algorithms is the *feature-based pruning* methodology: Each query graph is represented as a vector of features, where *features are subgraphs in the database*. If a target graph in the database contains the query, it must also contain all the features of the query. As long as one feature is missing, the target graph can be safely pruned without testing subgraph isomorphism between the query and itself.

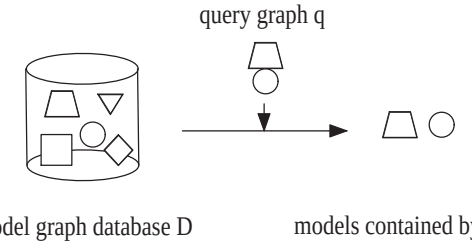


Figure 1: Graph Containment Search

The second problem, named *graph containment search*, though having rarely been systematically studied before,

*Work supported in part by the U.S. National Science Foundation (NSF) IIS-05-13678/06-42771 and BDI-05-15813.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
VLDB '07, September 23-28, 2007, Vienna, Austria.
Copyright 2007 VLDB Endowment, ACM 978-1-59593-649-3/07/09.

finds many applications in chem-informatics, pattern recognition [7] (visual surveillance, face recognition), cyber security (virus signature detection [6]), information management (user-interest mapping [19]), *etc.*. Figure 1 illustrates the scenario of containment search. A model graph database $\mathcal{D}$ is searched against the query graph q to find all models contained in q. For example, in chemistry, a descriptor (a.k.a. model graph) is a set of atoms with designated bonds that has certain attributes in chemical reactions. Given a new molecule, identifying "descriptor" structures can help researchers to understand its possible properties. In computer vision, attributed relational graphs (ARG) [7] are used to model images by transforming them into spatial entities such as points, lines, and shapes. ARG also connects these spatial entities (nodes) together with their mutual relationships (edges) such as distances, using a graph representation. The graph models of basic objects such as humans, animals, cars, airplanes, are built first. A recognition system could then query these models by the scene to identify foreground objects, or perform large-scale video search for specific models if the key frames of videos are represented by ARGs. Such a system can also be used to automatically recognize and classify basic objects in technical drawings from mechanical engineering and civil engineering.

In contrast to traditional graph search which finds all graphs that q is contained in ($q \subseteq g$), containment search has its distinguishing characteristics. Briefly speaking, the pruning strategy employed in traditional graph search has *inclusion logic*: Given a query graph q and a database graph $g \in \mathcal{D}$, if a feature $f \subseteq q$ and $f \not\subseteq g$, then $q \not\subseteq g$. That is, if feature f is in q then the graphs not having f are pruned. The inclusion logic prunes by considering features **contained** in the query graph. On the contrary, containment search has *exclusion logic*: If a feature $f \not\subseteq q$ and $f \subseteq g$, then $q \not\supseteq g$. That is, if feature f is not in q then the graphs having f are pruned.

Owing to the inclusion logic, feature-based pruning and indexing [25] is effective for traditional graph search because if a query graph q contains a feature f, then the search can be confined to those database graphs that contain the feature f (*i.e.*, indexed by f). However, such a framework does not work for containment search which uses the exclusion logic, because the exclusion logic relies on features **not contained** in q to implement the pruning. Also, as the space of features not contained in q is infinite, it is impossible to enumerate them for a given query graph. Therefore, a different indexing methodology is needed to support containment search.

Exclusion logic is the starting point for graph containment search. Here, instead of comparing a query graph q with database graphs, we compare q with features. Given a model graph database $\mathcal{D}$, the best indexing features are those subgraphs contained by lots of graphs in $\mathcal{D}$, but unlikely contained by a query graph. This kind of subgraph features are called *contrast features (or contrast subgraphs)*. The term has been used by emerging pattern mining [5, 23] in a different context, where it is defined as patterns that are frequent in positive data but not frequent in negative data. In this paper, the concept serves a different objective function, which will become clear in Sections 4 and 5.

There are two potential issues of finding and using contrast subgraphs for the purpose of indexing. First, there is an exponential number of subgraphs to examine in the model graph database, most of which are not contrastive at all. Practically, it is impossible to search all subgraphs that appear in the database. We found that there is a connection between contrast subgraphs and their frequency: Both infrequent and very frequent subgraphs are not highly contrastive, and thus not useful for indexing. So, we propose to use frequent graph pattern mining and pick those mid-frequent subgraphs. Second, the number of contrast subgraphs could be huge. Most of them are very similar to each other. Since the indexing performance is determined by a set of indexed features, rather than a single feature, it is important to find a set of contrast subgraphs that collectively perform well. We develop a redundancy-aware selection mechanism to sort out the most significant and distinctive contrast subgraphs.

Our proposed techniques, including contrast subgraphs and redundancy-aware feature selection, form a contrast feature-based index system, **cIndex**, which can achieve near-optimal pruning performance on real datasets. This system can be further extended to support approximate containment search. Specifically, we make the following contributions in this paper.

- We systematically investigate graph indexing towards containment search and propose a contrast subgraph-based indexing framework, cIndex. To support this framework, a new concept called the contrast graph matrix is developed. cIndex virtually implements contrast graph matrix by factorizing it into two smaller matrices, which leads to huge space economization.
- We show that it is NP-complete to choose the best set of contrast features, and prove that our greedy algorithm can approximate the one-level optimal index within a ratio of $1 - 1/e$.
- Based on cIndex, we develop data space reduction techniques such as clustering and sampling to speed up the index construction, and extend cIndex to support hierarchical indexing models: cIndex-BottomUp and cIndex-TopDown, which further improve the pruning capability.
- Our proposed methodology provides a general indexing solution to containment search, not only for graphs, but also for any data with transitive relations as well.

The rest of the paper is organized as follows. Section 2 gives preliminary concepts. Section 3 presents the basic framework, cIndex, for contrast subgraph-based containment search and indexing. The key concept of contrast subgraphs is introduced in Section 4. Section 5 illustrates the necessity and details of redundancy-aware feature selection. Section 6 explores hierarchical indexing models: cIndex-BottomUp and cIndex-TopDown. Index maintenances are discussed in Section 7. We report empirical studies, and give related work in Sections 8 and 9, respectively. Section 10 concludes this study.

2. PRELIMINARIES

In this paper, we use the following notations: For a graph g, $V(g)$ is its vertex set, $E(g) \subseteq V(g) \times V(g)$ is its edge set, and l is a label function mapping a vertex or an edge to a label. Due to the application background of containment search, a graph in the database is also called a *model graph* in our setting.

DEFINITION 1. *(Subgraph Isomorphism). For two labeled graphs g and g', a* subgraph isomorphism *is an injective function $f : V(g) \rightarrow V(g')$, s.t.,: first, $\forall v \in V(g), l(v) = l'(f(v))$; and second, $\forall (u,v) \in E(g), (f(u), f(v)) \in E(g')$ and $l(u,v) = l'(f(u), f(v))$, where l and l' are the labeling functions of g and g', respectively. Under these conditions, f is called an* embedding *of g in g'.*

DEFINITION 2. *(Subgraph and Supergraph). If there exists an embedding of g in g', then g is a* subgraph *of g', denoted by $g \subseteq g'$, and g' is a* supergraph *of g.*

The graph search and containment search problems are formulated as follows.

DEFINITION 3 (GRAPH SEARCH PROBLEM). *Given a graph database $\mathcal{D} = \{g_1, \ldots, g_n\}$ and a graph query q, find all graphs g_i in $\mathcal{D}$, s.t., $q \subseteq g_i$.*

DEFINITION 4 (GRAPH CONTAINMENT SEARCH PROBLEM). *Given a graph database $\mathcal{D} = \{g_1, \ldots, g_n\}$ and a graph query q, find all graphs g_i in $\mathcal{D}$, s.t., $q \supseteq g_i$.*

The definitions can be generalized to any partially ordered data with transitive relations, such as the following.

DEFINITION 5 (GENERAL CONTAINMENT SEARCH). *Given a transitive relation $\succeq$, a database $\mathcal{D} = \{g_1, g_2, \ldots, g_n\}$ and a query q, find all instances g_i in $\mathcal{D}$, s.t., $q \succeq g_i$.*

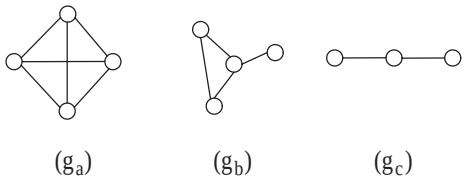


Figure 2: A Sample Database

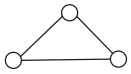


Figure 3: A Query Graph

EXAMPLE 1. *Figure 2 (g_a-g_c) is a sample dataset with three graphs and Figure 3 shows a query graph, where all vertices and edges share the same label. For graph search, graphs g_a and g_b contain the query graph; whereas for graph containment search, graph g_c is the answer.* ■

A naïve solution to the graph containment search problem, called SCAN, examines the database $\mathcal{D}$ sequentially and compares each graph g_i with the query graph q to decide whether $q \supseteq g_i$. Such a brute-force scheme is not efficient for large-scale databases or online applications. An interesting question is: Can we reduce the number of isomorphism tests? Intuitively, similar model graphs g_i and g_j are likely to have similar isomorphism testing results with regard to the same query graph. Let f be a common substructure shared by g_i and g_j. If $f \nsubseteq q$, then, $g_i \nsubseteq q$ and $g_j \nsubseteq q$. That is, we can save one isomorphism test (of course, it has overhead if $f \subseteq q$). This intuition leads to the design of our exclusion logic-based indexing methodology.

3. BASIC FRAMEWORK

The exclusion logic is as follows: Instead of comparing a query graph q with database graphs, we compare q with features. If feature f is not in q, then the graphs having f are pruned. The saving will be significant when f is a subgraph of many graphs in the database.

The basic index framework using exclusion logic has three major steps:

1. Off-line index construction: Generate and select a feature set $\mathcal{F}$ from the graph database $\mathcal{D}$. For feature $f \in \mathcal{F}$, $\mathcal{D}_f$ is the set of graphs containing f, *i.e.*, $\mathcal{D}_f = \{g | f \subseteq g, g \in \mathcal{D}\}$, which can be represented by an inverted ID list over $\mathcal{D}$. In the index, each feature is associated with such an ID list.

2. Search: Test indexed features in $\mathcal{F}$ against the query q which returns all $f \nsubseteq q$, and compute the candidate query answer set, $\mathcal{C}_q = \mathcal{D} - \bigcup_f \mathcal{D}_f$ $(f \nsubseteq q, f \in \mathcal{F})$.

3. Verification: Check each graph g in the candidate set $\mathcal{C}_q$ to see whether g is really a subgraph of q.

The above framework outlines the basic components of cIndex. Within this framework, given a query graph q and a set of features $\mathcal{F}$, the search time can be formulated as:

$$\sum_{f \in \mathcal{F}} T(f,q) + \sum_{g \in \mathcal{C}_q} T(g,q) + T_{index} \quad (1)$$

where $T(f,q)$ is the time to retrieve a feature and check whether it is a subgraph of q (each feature will incur one I/O and one isomorphism test), $T(g,q)$ is the time to retrieve a candidate model graph and check whether it is a subgraph of q (each candidate will incur one I/O and one isomorphism test), and T_{index} includes the cost to load the ID lists and the cost of index operations that compute $\mathcal{C}_q$.

According to the basic index framework, various kinds of index schemes can be formulated which use different feature sets. T_{index} is negligible as it represents operations on the ID lists (disk access and union calculation), while subgraph isomorphisms are of NP-hard complexity. This fact is further confirmed by our experiments. For simplicity, we shall focus on the time caused by $T(f,q)$ and $T(g,q)$, which is roughly proportional to the total number of features and model graphs retrieved and tested,

$$|\mathcal{F}| + |\mathcal{C}_q|. \quad (2)$$

In this study, we are going to use this simplified cost model (Equation 2) to illustrate the ideas behind our index design. Nonetheless, the algorithms developed here can be easily revised according to the original cost model of Equation 1.

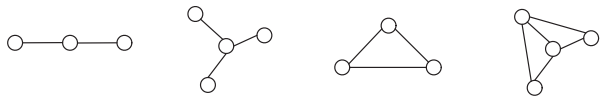
So far, we have not discussed how to generate and select a feature set $\mathcal{F}$ for index construction. In the next section, we are going to introduce the concept of contrast features, which can measure the pruning power of a single feature. After that, a redundancy-aware feature selection algorithm will be presented, which is able to select a set of significant and distinctive contrast features to maximize their collective pruning capability.

4. CONTRAST FEATURES

The exclusion logic suggests that every subgraph f satisfying $f \nsubseteq q$ is a potential candidate, with its pruning power

determined by the fraction of database containing f and the probability that a query graph does not contain f.

Figure 4 shows several subgraph features extracted from the sample graph database in Figure 2.



(f1) (f2) (f3) (f4)

Figure 4: Features

We can build a feature-graph matrix [21] to display the containment relationship between features and graphs, whose (i, j)-entry tells whether the j_{th} model graph has the i_{th} feature. Figure 5 shows a feature-graph matrix for features in Figure 4 and graphs in Figure 2. For instance, f_3 is contained in g_a.

	g_a	g_b	g_c
f_1	1	1	1
f_2	1	1	0
f_3	1	1	0
f_4	1	0	0

Figure 5: Feature-Graph Matrix

Consider the query shown in Figure 3, we can prune g_a and g_b immediately based on feature f_2, according to the exclusion logic. However, it is also possible that some overhead may arise if a wrong feature is selected. For example, suppose f_1 is selected, then the number of subgraph isomorphism tests will be 4 (three plus the extra testing between f_1 and the query graph), even worse than the naïve SCAN.

The above example shows that features might have different pruning power. Formally, let the frequency of a feature f in the database graphs be p, and the probability that a query graph having f be p'. Given a query graph, the expected number of subgraph isomorphism tests that can be saved from SCAN is as follows,

$$J_f = np(1-p') - 1, \tag{3}$$

where n is the total number of graphs in $\mathcal{D}$. We call the subgraphs with high value of J_f *contrast subgraph (or contrast feature)*. Here is the intuition: Contrast subgraphs are those subgraphs contained by a lot of graphs in $\mathcal{D}$, but unlikely contained by a query graph. It is obvious that we should index those contrast subgraphs since they provide the best pruning capability.

Equation 3 shows that features frequently (or infrequently) appearing in both database graphs and query graphs are useless. As query graphs and database graphs often share some similarity and thus a feature's frequency in query graphs is usually not too different from that in database graphs (*i.e.*, $p \approx p'$), J_f will achieve its maximum at $p = 1/2$ and monotonically decreases as $p \to 1$ or $p \to 0$. This, to some extent, indicates an observation that good features should be frequent, but not too frequent in the database. Since the pruning power of infrequent subgraphs are limited, we can use (closed) frequent subgraph mining algorithms,*e.g.*, FSG [16], GASTON [18], and gSpan [24], to generate an initial set of frequent subgraphs and then filter out those contrast ones. This approach is much easier than directly finding contrast features from the exponential set of subgraphs in $\mathcal{D}$.

5. REDUNDANCY-AWARE FEATURE SELECTION

In order to maximize the pruning capability of the basic framework introduced in Section 3, it is necessary to use a set of contrast subgraphs. However, there might be thousands, or millions of contrast subgraphs; it would be unrealistic to index all of them. On the other hand, contrast subgraphs could be similar to each other. If two contrast subgraphs are similar, then the pruning power of one subgraph will be shadowed by the other. Therefore, it is important to remove redundant features and use a set of distinctive contrast subgraphs. In this section, we introduce a redundancy-aware selection mechanism to sort out such features.

Given a SINGLE query graph q, we define the *gain* J of indexing a feature set $\mathcal{F}$ as the number of subgraph isomorphism tests that can be saved from SCAN:

$$\begin{aligned} J &= |\mathcal{D}| - |\mathcal{C}_q| - |\mathcal{F}| \\ &= |\cup_{f \not\subseteq q, f \in \mathcal{F}} \mathcal{D}_f| - |\mathcal{F}| \end{aligned} \tag{4}$$

For an initial set of features $\mathcal{F}_o = \{f_1, f_2, \ldots, f_m\}$, we would like to select a subset $\mathcal{F} \subseteq \mathcal{F}_o$ to maximize J. According to Equation 4, we should index a feature as long as q does not have it and it covers at least one database graph which has not yet been covered by other features (a feature *covers* a graph g if g has it). Thus, what we need to solve is exactly the *set cover* problem. Let

$$\mu_{ij} = \begin{cases} 1 & f_i \subseteq g_j, \\ 0 & otherwise. \end{cases}$$

The formulation is

$$\begin{aligned} \text{minimize} \quad & \sum_{i, f_i \not\subseteq q} x_i \\ \text{subject to} \quad & \forall j, \sum_i x_i \mu_{ij} \geq 1, \text{ if} \sum_i \mu_{ij} \geq 1, \\ & x_i \in \{0, 1\}. \end{aligned} \tag{5}$$

where x_i is a boolean value indicating whether feature f_i has been selected.

The solution to the above optimization problem can help us find an ideal feature set for a single query graph. However, in practice, the index should not be optimized for one query graph, but for a set of query graphs. We are going to discuss how to find a feature set that can provide the best pruning power for multiple queries.

5.1 Feature Selection

When multiple query graphs are taken into consideration, we can select features based on a probabilistic model or an exact model. In a probabilistic model, we use the probability for a feature contained in a query; while in an exact model, we do not compute probabilities but instead plug in a set of query graphs and see what happens. It seems that the probabilistic model is more restrictive: The joint probability

for multiple features contained in a query is hard to derive accurately, while it is required to determine whether one feature is redundant with respect to the others. Therefore, our study employs an exact model, which uses a query log $\mathcal{L}$ as the training data.

Given a SET of queries $\{q_1, q_2, \ldots, q_r\}$, an optimal index should be able to maximize the total gain

$$J_{total} = \sum_{l=1}^{r} |\cup_{f \sqsubseteq q_l, f \in \mathcal{F}} \mathcal{D}_f| - r|\mathcal{F}| \quad (6)$$

which is a summation of the gain in Equation 4 over all queries.

The optimization over J_{total} can be written as an integer programming problem as follows,

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n}\sum_{l=1}^{r} z_{jl} - r\sum_{i=1}^{m} x_i \\ \text{subject to} \quad & \sum_{i,\mu_{ij}=1} x_i \nu_{il} \geq z_{jl}, j = 1, \ldots, n \text{ and } l = 1, \ldots, r, \\ & x_i \in \{0,1\}, i = 1, \ldots, m, \\ & z_{jl} \in \{0,1\}, j = 1, \ldots, n \text{ and } l = 1, \ldots, r. \end{aligned} \quad (7)$$

where

$$\nu_{il} = \begin{cases} 1 & f_i \not\sqsubseteq q_l, \\ 0 & otherwise. \end{cases}$$

shows whether query graph q_l has feature f_i, and z_{jl} indicates whether database graph g_j is pruned for query graph q_l. The optimization of Equation 7 is related to set cover, but not straightforwardly.

Let us introduce the concept of contrast graph matrix, which is derived from the feature-graph matrix but with its i_{th} row set to $\overrightarrow{0}$ if the query has feature f_i. Figure 6 shows three query graphs and their corresponding contrast graph matrix for the 4 features and 3 model graphs shown in Figure 4 and Figure 2.

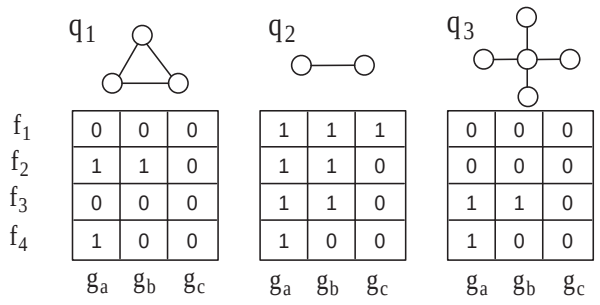


Figure 6: Contrast Graph Matrix

For each query graph q_l, we can construct a corresponding $m \times n$ contrast graph matrix M_l, where m is the number of features in the initial feature set $\mathcal{F}_o$ and n is the number of graphs in the database $\mathcal{D}$. These matrices are then column-concatenated to form a global $m \times nr$ matrix $\mathbf{M}$ for the whole query log set $\mathcal{L}$, where r is the number of queries in $\mathcal{L}$.

Definition 6 (Maximum Coverage With Cost). *Given a set of subsets $\mathbf{S} = \{S_1, S_2, \ldots, S_m\}$ of the universal set $U = \{1, 2, \ldots, n\}$ and a cost parameter λ associated with any $S_i \in \mathbf{S}$, find a subset $\mathbf{T}$ of $\mathbf{S}$ such that $|\cup_{S_i \in \mathbf{T}} S_i| - \lambda|\mathbf{T}|$ is maximized.*

Casting to a contrast graph matrix $\mathbf{M}$, U corresponds to the column index of $\mathbf{M}$, $\mathbf{S}$ corresponds to non-zero entries in each row of $\mathbf{M}$, $\mathbf{T}$ corresponds to the selected rows (*i.e.*, features), and λ corresponds to the number of queries r. Take the matrix in Figure 6 as an example: $U = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ because there are 9 columns in the concatenated matrix, $\mathbf{S} = \{\{4, 5, 6\}, \{1, 2, 4, 5\}, \{4, 5, 7, 8\}, \{1, 4, 7\}\}$, *e.g.*, $\{4, 5, 6\}$ corresponds to the first row in the matrix, $\lambda = 3$ since there are 3 query graphs. With this translation, the integer programming problem in Equation 7 becomes the problem of maximum coverage with cost.

Theorem 1. *Maximum coverage with cost is NP-complete.*

Proof. Reduction from set cover. Given an instance of set cover: Subsets $\mathbf{S} = \{S_1, S_2, \ldots, S_m\}$ of the universal set $U = \{1, 2, \ldots, n\}$, we further let $\lambda = 1$ and construct an instance of maximum coverage with cost. Assume we solve this instance of maximum coverage with cost and get a solution $\mathbf{T}$, which covers the universal set U except a portion U'. Now as any element $j \in U'$ must be covered by at least one subset S_{i_j} from $\mathbf{S}$ (otherwise, $\mathbf{S}$ cannot set cover U), we will show that $\mathbf{T} \cup \{S_{i_j} | j \in U'\}$ is a minimal set covering U. If this is true, then the NP-complete set cover can be solved via maximum coverage with cost through a polynomial-time reduction, which means that maximum coverage with cost itself must be NP-complete.

Proof by contradiction. Suppose that $\mathbf{T}_{min}$ is a minimal set covering U, while $\mathbf{T} \cup \{S_{i_j} | j \in U'\}$ is not. Plug $\mathbf{T}_{min}$ into the objective function of Definition 6:

$$\begin{aligned} |\cup_{S_i \in \mathbf{T}_{min}} S_i| - \lambda|\mathbf{T}_{min}| &= |U| - |\mathbf{T}_{min}| \\ &> |U| - |\mathbf{T} \cup \{S_{i_j} | j \in U'\}| \\ &= |U| - |U'| - |\mathbf{T}| \\ &= |\cup_{S_i \in \mathbf{T}} S_i| - \lambda|\mathbf{T}| \end{aligned} \quad (8)$$

we obtain $\mathbf{T}_{min}$ as a better solution than $\mathbf{T}$ for this particular instance of maximum coverage with cost. Contradiction!

To see Equation 8, we have: No $S_i \notin \mathbf{T}$ can cover at least 2 elements of U' (otherwise, $\{S_i\} \cup \mathbf{T}$ is a better solution than $\mathbf{T}$ for the problem of maximum coverage with cost, which can increase the objective function by at least 1), implying that each S_{i_j} can only cover 1 element of U', and thus it must be true that $|\{S_{i_j} | j \in U'\}| = |U'|$. ∎

Having seen the connection between set cover and maximum coverage with cost, it is reasonable to follow the same greedy heuristic used to approximate set cover in treating this new NP-complete problem. In each iteration, we select a row with the most number of non-zero entries from the matrix $\mathbf{M}$; however, since now including a feature is associated with cost r, it may not be good to completely cover each column of $\mathbf{M}$, which can potentially generate a heavy negative entry $-r|\mathcal{F}|$ in J_{total}. Rather, we set a fixed K, and choose the best K features. This can be regarded as a mechanism of *redundancy-aware feature selection*, which aims at finding those distinctive features that are able to cover database graphs not covered by previous features.

How good is this greedy feature selection process? Note that, as K is fixed and r is a constant, we only need to maximize the first part of Equation 6 so that the performance

metric J_{total} is optimized. Define that part:

$$\sum_{l=1}^{r} |\cup_{f \not\subseteq q_l, f \in \mathcal{F}} \mathcal{D}_f|$$

as the *coverage* of the indexed feature set $\mathcal{F}$ composed of K features. We have the following theorem.

THEOREM 2. *In terms of coverage, the greedy feature selection process can approximate the optimal index with K features within a ratio of $1 - 1/e$.*

PROOF. When the cost parameters $\lambda|\mathbf{T}|$ are dropped, we are indeed treating the problem of maximum coverage (but with no cost associated): Given m subsets $\mathbf{S} = \{S_1, S_2, \ldots, S_m\}$ of the universal set $U = \{1, 2, \ldots, n\}$, find K of them which can cover the most number of elements in U. [12] mentions a greedy algorithm as same as ours and proves that it can achieve an approximation ratio of $1 - 1/e$. Details of the proof are neglected, see [12] for references. ∎

What if the number of features K is not fixed as given? Straightforwardly, if the number of non-zero entries in a row is greater than the cost r, adding it to $\mathcal{F}$ can further bring down J_{total}. Overall, we refer to this feature selection algorithm as **cIndex-Basic**. Algorithm 1 outlines its pseudo-code. Given a contrast graph matrix, it first selects a row that hits the largest number of columns, and then removes this row and all columns covered by it. This process repeats until no row can cover at least r columns.

Algorithm 1 cIndex-Basic

Input: Contrast Graph Matrix $\mathbf{M}$.
Output: Selected Features $\mathcal{F}$.

1: $\mathcal{F} = \varnothing$;
2: **while** $\exists i, \sum_j \mathbf{M}_{ij} > r$ **do**
3: select row i with most non-zero entries in $\mathbf{M}$;
4: $\mathcal{F} = \mathcal{F} \cup \{f_i\}$;
5: **for each** column j s.t. $\mathbf{M}_{ij}$ is not zero **do**
6: delete column j from $\mathbf{M}$;
7: delete row i;
8: **return** $\mathcal{F}$;

5.2 Complexity and Virtualization

LEMMA 1. *cIndex-Basic's time complexity is $O(|\mathcal{F}_o||\mathcal{D}||\mathcal{L}|)$.*

PROOF. For each row i, we keep track of how many non-zero entries u_i exist. For each column deleted on line 6 of Algorithm 1, influenced u_i's must be modified to reflect the change, and the number of such u_i's is upperbounded by $|\mathcal{F}_o|$. As there are in total $|\mathcal{D}||\mathcal{L}|$ columns, maintaining all the u_i's throughout the algorithm takes $O(|\mathcal{F}_o||\mathcal{D}||\mathcal{L}|)$ time. On line 3, selecting the row with the currently highest u_i needs $O(|\mathcal{F}_o|)$ time. Because in each iteration we delete at least one column, all re-selections take at most $O(|\mathcal{F}_o||\mathcal{D}||\mathcal{L}|)$ time. In total, the time complexity is $O(|\mathcal{F}_o||\mathcal{D}||\mathcal{L}|)$. ∎

The contrast graph matrix shown in Figure 6 might be too large with a size of $|\mathcal{F}_o||\mathcal{D}||\mathcal{L}|$. As shown below, there is no need to materialize the contrast graph matrix. In fact, we can use a compact (contrast graph) matrix, together with the feature-graph matrix, to replace it.

	q_1	q_2	q_3
f_1	0	3	0
f_2	2	2	0
f_3	0	2	2
f_4	1	1	1

Figure 7: Compact Contrast Graph Matrix

Figure 7 shows a compact matrix M^c for the 4 features and 3 queries in Figure 4 and Figure 6. The (i, l)-entry of M^c is $\sum_{j=1}^{n} \mu_{ij}\nu_{il}$, where $\mu_{ij} = 1$ if the j_{th} model graph has the i_{th} feature and $\nu_{il} = 1$ if the l_{th} query graph does not have the i_{th} feature.

We can mimic the feature selection process on the original contrast graph matrix by manipulating the compact matrix M^c correspondingly: In each iteration, we select row i with the maximum sum of entries in M^c and update M^c according to its definition in above. In this way, we only need to keep two matrices for index construction. This virtual implementation of contrast graph matrix can reduce the space usage from $O(|\mathcal{F}_o||\mathcal{D}||\mathcal{L}|)$ to $O(|\mathcal{F}_o||\mathcal{D}| + |\mathcal{F}_o||\mathcal{L}|)$.

5.3 Non-binary Features

The analysis presented so far does not consider the plurality of features: In fact, features are not limited to 0-1 variables if we take into account the number of their occurrences (*i.e.*, non-automorphic embeddings) in a graph. Assume f_i occurs in g for $n(f_i, g)$ times. If $n(f_i, g) > n(f_i, q)$ (which substitutes the clause: $f_i \subseteq g$ & $f_i \not\subseteq q$ in the binary setting), then $g \not\subseteq q$. The basic framework can be slightly adjusted to accommodate non-binary features: Given a query log entry q, the contrast graph matrix for q is now defined as: $M_{ij} = 1$ if $n(f_i, g) > n(f_i, q)$; and when incoming queries are being processed, we allow pruning if $n(f_i, g) > n(f_i, q)$. Nothing else needs to be changed.

5.4 Time Complexity Reduction

As analyzed in Section 5.2, the time complexity of the greedy algorithm for feature selection is linear to the size of initial feature set $\mathcal{F}_o$, database $\mathcal{D}$, and query log $\mathcal{L}$. However, in practice, the product of $|\mathcal{F}_o|$, $|\mathcal{D}|$, and $|\mathcal{L}|$ may still be too large. If we can substantially reduce the cardinality of $\mathcal{F}_o$, $\mathcal{D}$, or $\mathcal{L}$, it would be of great help for fast index construction.

We propose two data reduction techniques: sampling and clustering. Due to space limitation, here we only discuss the sampling technique, *e.g.*, we might sample a small set of model graphs from the database and a small set of queries from the query log. Clustering can be similarly pursued if we treat it as a form of advanced sampling that selectively picks out representative data points. One should be cautious about the optimization function in Equation 6 when performing sampling on $\mathcal{D}$. Let $\widehat{\mathcal{D}}$ be a sample of $\mathcal{D}$; and based on this sample, we are going to build an index for $\mathcal{D}$. According to Equation 6, the first part of J_{total}:

$$\sum_{l=1}^{r} |\cup_{f \not\subseteq q_l, f \in \mathcal{F}} \mathcal{D}_f|$$

now approximately shrinks to $\frac{|\widehat{\mathcal{D}}|}{|\mathcal{D}|}$ times as before. In order to make the optimization solution in line with the original

one without sampling, the second part of J_{total} should be reduced by the same factor accordingly. Therefore, the total gain depicted in Equation 6 becomes

$$J_{total} = \sum_{l=1}^{r} |\cup_{f \not\subseteq q_l, f \in \mathcal{F}} \widehat{\mathcal{D}}_f| - \frac{|\widehat{\mathcal{D}}|}{|\mathcal{D}|} \cdot r|\mathcal{F}|$$

6. HIERARCHICAL INDEXING MODELS

The cIndex-Basic algorithm introduced in Section 5 builds a flat index structure, where each feature is *tested sequentially and deterministically* against any input queries. This mechanism has two potential disadvantages.

First, the index, consisting of a set of features $\mathcal{F}$ and their corresponding inverted ID lists, is flat-structured. Given a query graph, we need to sequentially test the containment relationship between q and each of the features in $\mathcal{F}$, which is equivalent to executing the naïve SCAN approach on $\mathcal{F}$. Note that, $\mathcal{F}$ is nothing else but a set of graphs. In order to avoid naïve SCAN, it is interesting to view $\mathcal{F}$ itself as another graph database on which a second-level index can be built. This process can be repeated to generate a hierarchical index structure. Figure 8 shows *a bottom-up hierarchical index* based on this design strategy.

Second, cIndex-Basic follows a deterministic order when testing its indexed features. Specifically, each feature in $\mathcal{F}$ must be retrieved and compared with any given query q, which does not adapt to different inputs flexibly. Now, suppose we have tested a feature f_1 and are going to test another feature f_2, then the testing result between f_1 and q, *i.e.*, $f_1 \subseteq q$ or $f_1 \not\subseteq q$, can give us hints about how the query looks like, which may affect our choice of the second feature. In cIndex-Basic, f_2 is fixed no matter $f_1 \subseteq q$ or $f_1 \not\subseteq q$. Here, we can relax this constraint and allow them to be two different features: f_2 and f_2'. Would this strategy help improve the performance? The answer is yes! Following exactly the same idea to select more features engenders *a top-down hierarchical index*, as shown in Figure 9.

In this section, we will examine these two hierarchical indexing models. Using a conventional search tree to index the inverted list associated with each feature, they can be easily implemented over a graph database.

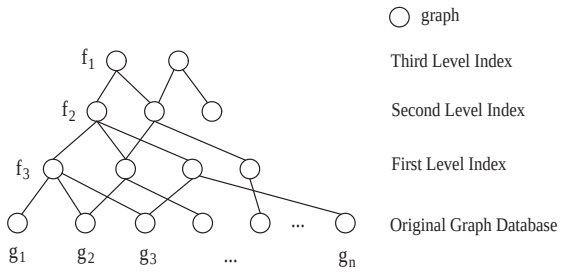


Figure 8: Bottom-up Hierarchical Indexing

6.1 Bottom-up Hierarchical Index

Bottom-up means that the index is built layer by layer starting from the bottom-level graphs. Figure 8 shows a bottom-up hierarchical index where the i^{th}-level index $\mathcal{I}_i$ is built by applying cIndex-Basic to features in the $(i-1)^{th}$-level index $\mathcal{I}_{i-1}$. For example, the first-level index $\mathcal{I}_1$ is built on the original graph database by cIndex-Basic. Once this is done, the features in $\mathcal{I}_1$ can be regarded as another graph database, where cIndex-Basic can be executed again to form a second-level index $\mathcal{I}_2$. Following this manner, we can continue building higher-level indices until the pruning gain becomes zero.

This method is called **cIndex-BottomUp**. Note that in a bottom-up index, features on the i^{th}-level must be subgraphs of features on the $(i-1)^{th}$-level. In Figure 8, subgraph relationships are shown as edges. For example, f_1 is a subgraph of f_2, which is in turn a subgraph of f_3. Given a query graph q, if $f_1 \not\subseteq q$, then the tree covered by f_1 need not be examined due to the exclusion logic. This phenomenon is called the *cascading* effect. Since the index on each level will save some isomorphism tests for the graphs it indexes, it is obvious that cIndex-BottomUp should outperform the flat index of cIndex-Basic.

The bottom-up index has a potential problem on the arrangement of features in the hierarchy. At the first glance, the cascading effect can avoid examinations of many features in the index if a higher-level feature is not contained in the query. However, this will not happen often because higher-level features are in general weaker and thus much easier to be contained: During the construction of a bottom-up index, the best features are entered into lower levels because they are built first. Intuitively, it would be more profitable to put high-utility features in higher-level indices, which can prune the database graphs as early as possible. This idea leads us to the following top-down hierarchical index design.

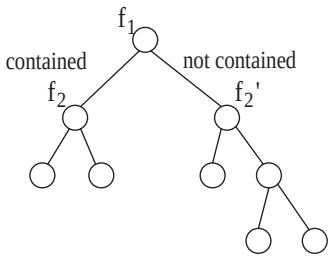


Figure 9: Top-down Hierarchical Indexing

6.2 Top-down Hierarchical Index

The top-down hierarchical index first puts f_1, the feature with the highest utility, at the top of the hierarchy. Given a query graph q, if f_1 is contained by q, we go on to test f_2; if f_1 is not contained by q, we at first prune all model graphs indexed by f_1, and then pick f_2' as the second feature. In a flat index built by cIndex-Basic, f_2 and f_2' are forced to be the same: No matter whether f_1 is contained by q, the same feature will be examined next. However, in a top-down index, they can be different.

Generally speaking, the top-down index adopts a *differentiating* strategy. Suppose that a query log set $\mathcal{Q}$ is clustered into c groups: $\mathcal{Q}_1, \mathcal{Q}_2, \ldots, \mathcal{Q}_c$, based on their isomorphism testing results against a set of features. We can uniformly choose K features $\mathcal{F}_\mathcal{Q}$ with regard to $\mathcal{Q}$, or we can specifically choose K features $\mathcal{F}_{\mathcal{Q}_l}$ for each of $\mathcal{Q}_l$ $(l = 1, 2, \cdots, c)$. Likely, the latter strategy will achieve better performance due to its finer granularity.

Figure 10 illustrates the process of building a top-down hierarchical index. For the three query graphs shown in Figure 6, we first construct their contrast graph matrix. Based on

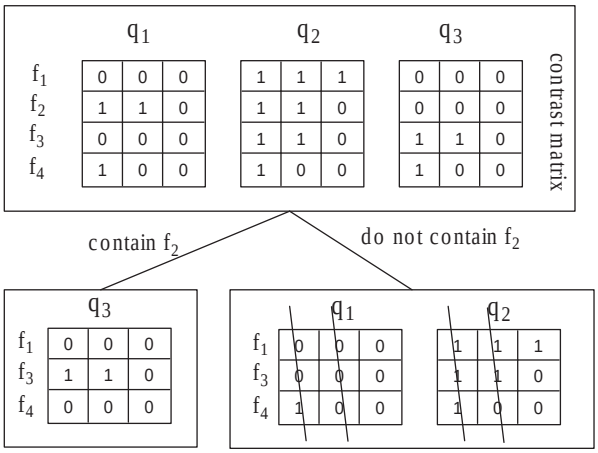


Figure 10: Top-down Index Construction

the greedy feature selection algorithm, feature f_2 will be selected first (f_3 is a tie), where $\{q_1, q_2, q_3\}$ is divided into two groups $\{q_3\}$ (those contain f_2) and $\{q_1, q_2\}$ (those do not contain f_2). In the right branch, columns covered by f_2 are removed from the contrast graph matrix.

The above partitioning process is iterated until we reach a leaf node of the tree. In order to avoid too deep branches that may cause overfitting, a smallest number (min_size) of queries must be retained within each leaf. Now the hierarchical structure is clear: at an internal node N_i, we select one feature f_i to split the query log set into two groups; at a leaf node N_l, cIndex-Basic is applied to build a flat index. We call this hybrid approach **cIndex-TopDown**.

Formally, given a query log set $\mathcal{L}_i$, a feature set $\mathcal{F}_i$, and a contrast graph matrix $\mathbf{M}_i$ at an internal node N_i, we run cIndex-Basic on $\mathbf{M}_i$ to get the feature with the highest utility. Let this feature be f. f splits $\mathcal{L}_i$ into two groups $\mathcal{L}'_i = \{q | f \subseteq q, q \in \mathcal{L}_i\}$ and $\mathcal{L}''_i = \{q | f \not\subseteq q, q \in \mathcal{L}_i\}$, which form the two branches of node N_i. $\mathbf{M}_i$ is also split into two parts: $\mathbf{M}'_i$ and $\mathbf{M}''_i$, where $\mathbf{M}'_i$ is the contrast graph matrix for the query log set $\mathcal{L}'_i$ and $\mathbf{M}''_i$ is the matrix for $\mathcal{L}''_i$. In the "do not contain" branch, all columns covered by feature f are removed from $\mathbf{M}''_i$.

7. INDEX MAINTENANCES

The indexing methodology we have examined so far is suitable within a static environment. When updates take place in the database $\mathcal{D}$ or query graphs deviate away from previously logged entries, how should the index react in order to maintain its performance?

First, we can take an "ostrich" strategy: Stick with the same set of selected features and the same hierarchical structure built. Whenever an insertion/deletion to $\mathcal{D}$ takes place, we simply add/remove a corresponding item to/from the inverted ID lists of all involved features. If the changed model graph database and query log is not significantly deviated from the original, it is quite reasonable that the index thus maintained will continue to perform well. This aspect of the "ostrich" algorithm is further confirmed in our experimental results.

Unfortunately, there are also cases where a previously built index becomes completely outdated after a substantial amount of updates. This may be due to dramatic changes to database $\mathcal{D}$, log $\mathcal{L}$, or both, which makes it inappropriate to still take the original database and query log as references. In practice, such scenarios must be monitored so that some counter-measures can be taken.

Let $\mathcal{I}$ be the index originally built. We periodically take small samples $\mathcal{D}_s$ and $\mathcal{L}_s$ from $\mathcal{D}$ and $\mathcal{L}$, mine a set of frequent subgraphs $\mathcal{F}_s$ out of $\mathcal{D}_s$, and calculate a new index $\mathcal{I}_s$ based on the triplet $(\mathcal{F}_s, \mathcal{D}_s, \mathcal{L}_s)$. The mining step can even be bypassed, because frequent patterns represent the intrinsic trends of data and are relatively stable in spite of updates. The sampling ratios can be set according to the updating rates of $\mathcal{D}$ and $\mathcal{L}$. Since queries are continuously being logged in an online fashion, we can also apply strategies such as tilted time windows [3] to over-sample the query stream in the recent past so that the evolutions of query distribution are better reflected. As will be shown in the experiment section, the performance of $\mathcal{I}$ and $\mathcal{I}_s$ can be systematically gauged: If $\mathcal{I}$ has not degraded much from $\mathcal{I}_s$, which roughly represents an up-to-date index, the "ostrich" strategy is taken; otherwise, we may consider to replace $\mathcal{I}$ with $\mathcal{I}_s$ or reconstruct it from a larger sample to assure even higher accuracy. Here, we take advantage of the data space reduction techniques introduced in Section 5.4. They are very effective in building an index with similar quality as the one built on the whole dataset, while the index construction time is greatly reduced. This is ideal for online performance monitoring purposes.

8. EMPIRICAL STUDIES

In this section, we present our empirical studies on real datasets. We first demonstrate cIndex's effectiveness by comparing its query performance with that of the naïve SCAN method and the indexing structure provided by gIndex [25], a state-of-art algorithm proposed for traditional graph search, followed by examinations on scalability, maintenances, *etc.*. As gIndex is not designed for containment search, we use the features selected by gIndex, and feed them into the containment search framework given in Section 3. This strategy is named as FB (Feature-Based) to avoid possible confusions with cIndex's contrast feature-based methodology.

Our experiments are done on a Windows XP machine with a 3GHz Pentium IV CPU and 1GB main memory. Programs are compiled by Microsoft Visual C++ 6.

8.1 Experiment Settings

Graph containment search has many applications. In this experiment, we are going to explore two of them, fast chemical descriptor search and real-time object recognition search, on two real datasets. The first one is an AIDS anti-viral screen dataset, which contains the graph structures of chemical compounds. This dataset is publicly available on the website of Developmental Therapeutics Program, which was also used in the testing of gIndex [25]. The second one is taken from TREC Video Retrieval Evaluation (*i.e.*, TREC-VID, url: `http://www-nlpir.nist.gov/projects/trecvid`), which is a public benchmark for evaluating the performance of high-level concept (*e.g.*, objects in scenes) detection and image search [27, 10].

8.2 Chemical Descriptor Search

The AIDS anti-viral screen dataset contains more than

40,000 chemical compounds. We randomly pick 10,000 graphs to form a dataset, which is denoted as W. W is divided to a query log set $\mathcal{L}$ and a testing query set $\mathcal{Q}$ based on five-fold cross-validation. That is, we randomly partition W into five pieces $W_1, \ldots, W_5$ of equal size, and each time use four pieces as query log and one piece as testing query. Results are reported as an average over five folds.

In chem-informatics, a model graph database usually includes a set of fundamental substructures, called descriptors. These descriptors, shared by specific groups of known molecules, often indicate particular chemical and physical properties. Given a molecule, fast searching for its "descriptor" substructures can help researchers to quickly predict its attributes. In order to build a descriptor (model) graph database $\mathcal{D}$, we apply frequent subgraph mining on W and generate 5,000 distinctive frequent subgraphs whose frequency ranges from 0.5% to 10%.

During experiments, we keep the index built by cIndex on disk. When a query q comes, we load the indexed features and compare them with q (this represents the $|\mathcal{F}|$ part of Equation 2). For those features $f \nsubseteq q$, their associated ID lists are then loaded into memory and merged to compute the candidate query answer set (this represents the T_{index} part of Equation 1). Finally, these candidate graphs are retrieved from the database for further verification (this represents the $|\mathcal{C}_q|$ part of Equation 2).

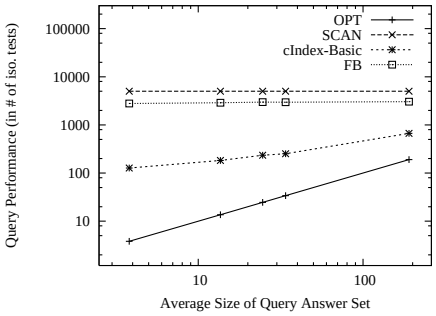


Figure 11: Query Performance of cIndex-Basic, in terms of Subgraph Isomorphism Test Numbers

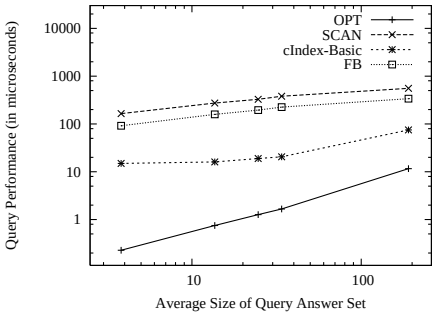


Figure 12: Query Performance of cIndex-Basic, in terms of Query Processing Time

Figure 11 and Figure 12 compare the query performance of cIndex-Basic with that of SCAN and FB. Here, 2,000 queries are taken from the testing fold of W, which are divided into five bins: $[0, 10), [10 - 20), [20 - 30), [30, 40), [40, \infty)$, based on the size of the query answer set, *i.e.*, the number of database graphs that are contained in the query. The x-axis in Figure 11 shows the average answer set size of the queries within each bin, and the y-axis depicts the average querying cost of each method in terms of isomorphism test numbers. Figure 12 has the same x-axis as Figure 11, but its y-axis shows the corresponding query processing time instead of isomorphism test numbers. In summary, Figure 11 is gauging based on Equation 2 of the basic framework, while Figure 11 is gauging based on Equation 1.

The cost of SCAN is obviously $|\mathcal{D}| = 5,000$ tests for Figure 11 and the total time needed to load and check all these 5,000 database graphs for Figure 12. As Equation 1 indicates, the cost of cIndex-Basic and FB consist of three parts: the first part is testing indexed features against the query, the second part is testing each graph in the candidate answer set for verification, and the third part is the index overhead. Since in the second part, we always need to verify those database graphs really contained in the query, which can never be saved by any indexing efforts, this portion of cost is marked as OPT in both figures.

To make the comparison fair, we select the same number of features for FB and cIndex-Basic. As shown in the figure, cIndex-Basic performs significantly better than naïve SCAN and FB, while itself achieves near-optimal pruning power. Note the log scale of y-axis, cIndex-Basic in fact increases the performance by more than one order of magnitude. The above result well testifies the necessity of a new indexing scheme aiming at containment search, which is the major motivation of this study.

Furthermore, trends shown in Figure 11 and Figure 12 are very similar to each other, which implies that the simplified cost model we proposed in Equation 2 is quite reasonable. Actually, we also measured the differential part: T_{index} in our experiments: Looking at Figure 12, cIndex-Basic in general costs several tens of microseconds per query; while in comparison, T_{index} is in the range of 0.3ms-0.4ms. We did not draw this adjustment in the picture, as it will become too small to be discernable. Having this in mind, we shall stick with the isomorphism test numbers (*i.e.*, follow Equation 2) to gauge query performance in below.

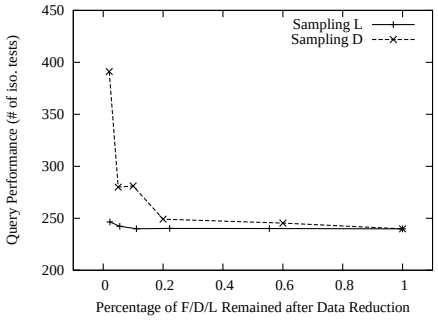


Figure 13: Effectiveness of Data Space Reduction

Figure 13 shows the effectiveness of the data space reduction techniques proposed in Section 5.4. Here, we sample $\mathcal{D}$ and $\mathcal{L}$ to reduce the data size to a specific percentage shown on the x-axis. y-axis shows the query performance of the index built on the sampled model graphs and query logs. It seems that all strategies perform well as long as the remained data size is above some threshold and thus can

well reflect the data distribution. Sampling on query log $\mathcal{L}$ is very stable, indicating that the amount of query log data needed is moderate. Furthermore, as suggested by Lemma 1, we observe that the index construction time is linearly scalable with regard to the sample size $|\mathcal{D}|$ and $|\mathcal{L}|$; while the figure is omitted here due to lack of space.

In addition, if it is hard to obtain query logs, cIndex can perform a "cold start": As the database distribution is often not too different from the query distribution, we might use the database itself as a pseudo log to initiate an index so that the system can start to operate. After that, as real queries flow in, the maintenance scheme described in Section 7 can be applied. Using this alternative, the average querying cost on the set of 2,000 queries is degraded from 239.8 tests to 340.5 tests, which we believe is affordable. This means that cIndex can work well even without query log.

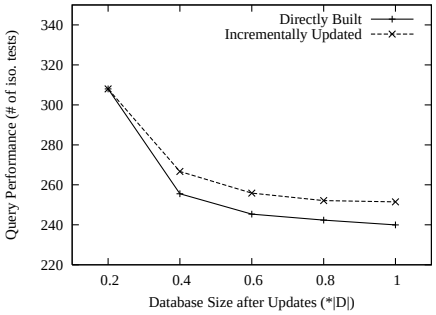


Figure 14: Index Maintenances

Figure 14 shows the situation when database updates take place. To simulate such a scenario, a sample $\mathcal{D}_4 = \frac{4}{5}|\mathcal{D}|$ is taken from $\mathcal{D}$; similarly, we sample $\mathcal{D}_3$ out of $\mathcal{D}_4$, $\mathcal{D}_2$ out of $\mathcal{D}_3$, and $\mathcal{D}_1$ out of $\mathcal{D}_2$, so that $|\mathcal{D}_i| = \frac{i}{5}|\mathcal{D}|(i = 1, \ldots, 5)$, where $\mathcal{D}_5 = \mathcal{D}$. By doing this, $\mathcal{D}_j$'s $(j > i)$ can be viewed as a database incrementally updated from $\mathcal{D}_i$ through a batch of insertions. We compare two curves here: One curve shows the performance of the index directly constructed for $\mathcal{D}_i$, while the other shows the performance of the index originally constructed for $\mathcal{D}_1$ but maintained afterwards to fit $\mathcal{D}_i$ ("ostrich" updating). Performance is still gauged using the same set of 2,000 queries. It can be seen that two curves start from the same point at $\mathcal{D}_1$, and keep quite close to each other for all the following updated databases. This well validates the feasibility of the proposed "ostrich" index maintenance scheme.

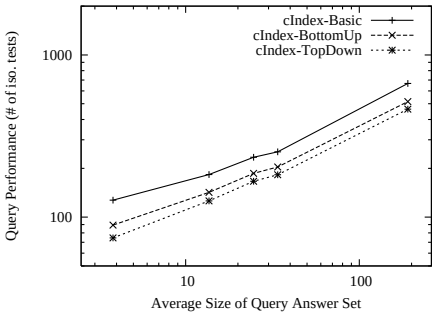


Figure 15: Performance of Hierarchical Indices

In the next several experiments, we examine hierarchical indices. Figure 15 depicts the query performance of all three indices: cIndex-Basic, cIndex-BottomUp, and cIndex-TopDown. We implement a 2-level bottom-up index, and set the top-down index's stop-split query log *min_size* at 100. Compared to cIndex-Basic, a bottom-up index can approximately save 25% by providing another index for the first-level features, while a top-down index can save by nearly one half, due to its differentiating effect.

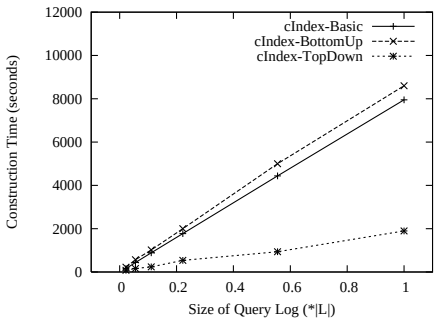


Figure 16: Scalability of Hierarchical Indices

Figure 16 compares the scalability of hierarchical indexing algorithms to that of cIndex-Basic, with respect to the size of query log: $|\mathcal{L}|$. It is observed that, cIndex-BottomUp takes a little bit more time, due to the additional computation it devotes to the second-level index. Surprisingly, cIndex-TopDown runs even faster than cIndex-Basic. This is due to the fact that splitting at an internal node removes part of the contrast graph matrix, and thus the time spent in choosing deeper-level features is reduced. The results are quite similar when we vary $|\mathcal{F}_o|$ and $|\mathcal{D}|$.

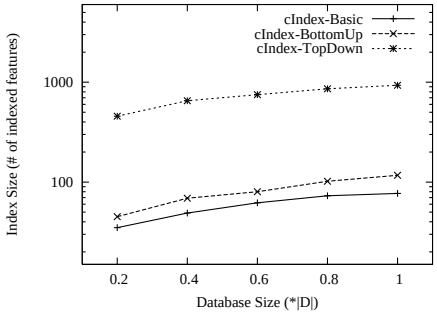


Figure 17: Index Size

Figure 17 compares the index size by measuring the number of indexed features. It is shown that the increase of index size is sub-linear to the increase of database size, which is a nice property when dealing with large databases. cIndex-TopDown somehow indexes more features since it selects a potentially different feature set for different subgroups of the query log. Both cIndex-BottomUp and cIndex-TopDown exchange some space cost for the improvement of query performance.

8.3 Object Recognition Search

In object recognition search, labeled graphs are used to model images by transforming pixel regions into entities.

These entities (taken as vertices) are connected together according to their spatial distances. The model graphs of basic objects, *e.g.*, humans, animals, cars, airplanes, *etc.*, are stored in a database, which are then queried by the recognition system to identify foreground objects that appear in the scene. Certainly, such querying should be conducted as fast as possible in order to support real-time applications.

We download more than 20,000 key frame images from the TREC-VID website. A labeled graph is generated for each image through salient region detection, followed by a quantization step that converts continuous vertex/edge features into integer values. The scale-saliency detector is developed by Kadir *et al.* [14], which outputs the locations and scales of detected regions. After that, color moment features are extracted from each region, where regions and their color moment features form vertices. Edges of a graph are generated by connecting nearby vertices, which are then associated with corresponding spatial distances. The graphs transformed from the key frame images are taken as query graphs, while the model graphs are extracted as annotated regions in the key frames. The TREC-VID benchmark provides many concept image regions, including face, building, table, monitor, microphone, baseball, bicycle, *etc.*, that are annotated by humans. We collected several thousand model graphs from these concepts. Since one concept might have different appearances, there are tens of, or even hundreds of model graphs for each concept.

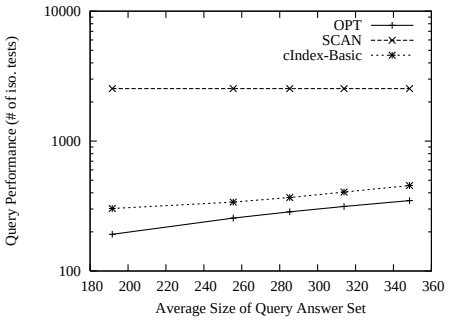


Figure 18: Query Performance – TREC-VID

Figure 18 shows the result. The database contains 2,535 model graphs. We use cIndex-Basic to build an index, whose performance is then tested against 3,000 querying key frames. It can be seen that as same as in chemical descriptor search, the system provides near-optimal pruning capability and is an order of magnitude faster than the naïve SCAN approach. We observe that, in our case, the average time needed by SCAN to process one query is about 1 second. Obviously, such costs would be unaffordable for large-scale applications that require real-time responses.

9. RELATED WORK

Graph search, both the traditional and containment one, are needed in various application domains, though they may be associated with different names, such as graph matching.

For traditional graph search, there have been a lot of studies under the term "graph indexing". In XML, where the structures encountered are often trees and lattices, query languages built on path expression become popular [9, 20]. For more complicated scenarios such as searching a chemical compound database by small functional fragments, Shasha *et al.* [21] extend the path-based technique for full-scale graph retrieval; Yan *et al.* propose gIndex [25] and use frequent subgraphs as indexing features, which is shown to be more effective. There are also a few studies dealing with approximate graph search. He *et al.* [11] develop a closure tree with each node being a closure (*i.e.*, summarization) of its descendant nodes. This tree is utilized as an index to perform (approximate) graph search. Tian *et al.* [22] define a more sophisticated similarity measure, and design a fragment (*i.e.*, feature-based) index to assemble an approximate match. However, all these methods target traditional graph search. In order to tackle graph containment search, a new methodology is needed.

Graph matching has been a research focus for decades [2], especially in pattern recognition, where the wealth of literature cannot be exhausted. Conte *et al.* give a comprehensive survey on graph matching [4], Gold *et al.* [8] apply a graduated assignment algorithm to quickly match two graphs, and Messmer *et al.* [17] place all permutations (*i.e.*, isomorphisms) of each database graph in a decision tree for online retrieval. In bio-informatics, how to align two biological networks is a challenging issue. Koyutürk *et al.* [15] propose a method for aligning two protein networks using a distance based on evolutionary models characterizing duplication and divergence. However, to the best of our knowledge, no algorithm considers the possibility of using a contrast feature-based methodology for graph containment search and indexing, which is our focus in this work.

Fang *et al.* propose to use string matching algorithms for virus signature detection in network packet streams [6]. Compared with bit-sequence signatures, if structured signatures are used instead to capture richer information, the methodology developed in this paper would be needed. Petrovic *et al.* [19] model the online information dissemination process as a graph containment search problem. Users subscribe their interests, represented as an RSS (RDF Site Summary) graph model, to an intermediate broker, who checks each incoming publication to see whether it covers the interest of any subscription: If it does, the publication is disseminated to the right users. Our cIndex framework can definitely benefit such user-interest mapping systems.

As for graph mining, there have been a sequence of methods developed [13, 16, 24, 18] that can efficiently extract (closed) frequent subgraphs from a database. They act as a firm basis for the feature selection process of cIndex.

Putting graph in the context of partially ordered data, we can examine containment search at a higher level of abstraction. Based on the transitive relation: $\alpha \preceq \beta, \beta \preceq \gamma \Rightarrow \alpha \preceq \gamma$, a lattice can be built for the data space, and contrast feature-based indexing can be regarded as a method of finding those elements existing in the middle of the lattice which are best for indexing. Besides this, there are other ways of doing it for special data types. One well-known example is the Aho-Corasick string matching algorithm [1], which can locate all occurrences of a set of keywords (database) in a input string (query) within $O(n+m)$ time, where n is the total size of key words and m is the input length. Unfortunately, in the context of graph containment search, a similar solution is hard to derive since it is infeasible to design a graph serialization technique so that subgraph isomorphism check can be replaced by substring matching test: Subgraph isomorphism is NP-complete [12], which cannot be solved via

polynomial substring matching. Rather, in this study, we approach contrast feature-based indexing as a methodology which do not rely on the specialties of data, and thus serve the general principles of indexing partially ordered data.

10. CONCLUSIONS

Different from traditional graph search that has been extensively studied, graph containment search was rarely discussed before as a database management problem, though it is needed in various domains such as chem-informatics, pattern recognition, cyber security, and information management. It is desirable to quickly search a large database for all the model graphs that a query graph contains. This motivates us to develop a novel indexing methodology, cIndex, for efficient query processing. Our empirical studies also confirm that traditional graph search indices do not fit the scenario of containment search.

As a unique characteristic of containment search, cIndex relies on exclusion logic-based pruning. We propose a contrast subgraph-based indexing framework, develop a redundancy-aware process to select a small set of significant and distinctive index features, and further improve the performance by accommodating two hierarchical indexing models. Other techniques and issues, such as data space reduction for fast index construction, and index maintenances in front of updates, are also discussed. The methods proposed are general for any data with transitive relations. Applying the cIndex methodology to new data types and extending it to support approximate search are interesting topics for future research.

11. REFERENCES

[1] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

[2] H. Bunke. Graph matching: Theoretical foundations, algorithms, and applications. In *Vision Interface*, pages 82–88, 2000.

[3] Y. Chen, G. Dong, J. Han, B. W. Wah, and J. Wang. Multi-dimensional regression analysis of time-series data streams. In *VLDB*, pages 323–334, 2002.

[4] D. Conte, P. Foggia, C. Sansone, and M. Vento. Thirty years of graph matching in pattern recognition. *IJPRAI*, 18(3):265–298, 2004.

[5] G. Dong and J. Li. Efficient mining of emerging patterns: Discovering trends and differences. In *KDD*, pages 43–52, 1999.

[6] Y. Fang, R. H. Katz, and T. V. Lakshman. Gigabit rate packet pattern-matching using tcam. In *ICNP*, pages 174–183, 2004.

[7] K. S. Fu. A step towards unification of syntactic and statistical pattern recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*, 8(3):398–404, 1986.

[8] S. Gold and A. Rangarajan. A graduated assignment algorithm for graph matching. *IEEE Trans. Pattern Anal. Mach. Intell.*, 18(4):377–388, 1996.

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

[10] X. Gu, Z. Wen, C. Lin, and P. S. Yu. Vico: an adaptive distributed video correlation system. In *ACM Multimedia*, pages 559–568, 2006.

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

[12] D. S. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing, MA, 1997.

[13] L. B. Holder, D. J. Cook, and S. Djoko. Substucture discovery in the subdue system. In *KDD Workshop*, pages 169–180, 1994.

[14] T. Kadir and M. Brady. Saliency, scale and image description. *International Journal of Computer Vision*, 45(2):83–105, 2001.

[15] M. Koyutürk, A. Grama, and W. Szpankowski. Pairwise local alignment of protein interaction networks guided by models of evolution. In *RECOMB*, pages 48–65, 2005.

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

[17] B. T. Messmer and H. Bunke. A decision tree approach to graph and subgraph isomorphism detection. *Pattern Recognition*, 32(12):1979–1998, 1999.

[18] S. Nijssen and J. N. Kok. A quickstart in frequent structure mining can make a difference. In *KDD*, pages 647–652, 2004.

[19] M. Petrovic, H. Liu, and H.-A. Jacobsen. G-topss: fast filtering of graph-based metadata. In *WWW*, pages 539–547, 2005.

[20] C. Qun, A. Lim, and K. W. Ong. D(k)-index: An adaptive structural summary for graph-structured data. In *SIGMOD Conference*, pages 134–144, 2003.

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

[22] Y. Tian, R. C. McEachin, C. Santos, D. J. States, and J. M. Patel. SAGA: a subgraph matching tool for biological graphs. *Bioinformatics*, pages 232–239, 2006.

[23] R. M. H. Ting and J. Bailey. Mining minimal contrast subgraph patterns. In *SDM*, 2006.

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

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

[26] X. Yan, P. S. Yu, and J. Han. Substructure similarity search in graph databases. In *SIGMOD Conference*, pages 766–777, 2005.

[27] D. Zhang and S.-F. Chang. Detecting image near-duplicate by stochastic attributed relational graph matching with learning. In *ACM Multimedia*, pages 877–884, 2004.