Substructure Similarity Search in Graph Databases*

Xifeng Yan† Philip S. Yu‡ Jiawei Han†

†University of Illinois at Urbana-Champaign, {xyan, hanj}@cs.uiuc.edu
‡IBM T. J. Watson Research Center, psyu@us.ibm.com

ABSTRACT

Advanced database systems face a great challenge raised by the emergence of massive, complex structural data in bioinformatics, chem-informatics, and many other applications. The most fundamental support needed in these applications is the efficient search of complex structured data. Since exact matching is often too restrictive, *similarity search of complex structures* becomes a vital operation that must be supported efficiently.

In this paper, we investigate the issues of *substructure similarity search using indexed features* in graph databases. By transforming the edge relaxation ratio of a query graph into the maximum allowed missing features, our structural filtering algorithm, called Grafil, can filter many graphs without performing pairwise similarity computations. It is further shown that using either too few or too many features can result in poor filtering performance. Thus the challenge is to design an effective feature set selection strategy for filtering. By examining the effect of different feature selection mechanisms, we develop a multi-filter composition strategy, where each filter uses a distinct and complementary subset of the features. We identify the criteria to form effective feature sets for filtering, and demonstrate that combining features with similar size and selectivity can improve the filtering and search performance significantly. Moreover, the concept presented in Grafil can be applied to searching approximate non-consecutive sequences, trees, and other complicated structures as well.

1. INTRODUCTION

Database research has been facing a new challenge raised by the emergence of massive, complex structural data, in the form of sequences, trees, and graphs. Among all the complex structured data, graph is the most sophisticated and general form of structure. Graphs have broad applications and have become the first-class citizens in widely used datasets in chem-informatics and bioinformatics, such as ChemIDplus [16], PDB [2], and KEGG [11].

Graph database has been largely involved in the development of chemical structure search and registration systems. In chemistry, the structures and properties of newly discovered or synthesized chemical molecules are studied, classified, and recorded for scientific and commercial purposes. ChemIDplus [16], a free data service offered by the National Library of Medicine (NLM), provides access to structure and nomenclature information. Users can query molecules by their names, structures, toxicity, and even weight in a flexible way through its web interface. Given a query structure, it can quickly identify a small subset of molecules for further analysis [8, 24], thus shortening the discovery cycle in drug design and other scientific activities. Nevertheless, the usage of a graph database as well as its query system is not confined to chemical informatics only. In computer vision and pattern recognition [17, 13, 1], graphs are used to represent complex structures such as hand-drawn symbols, 3D objects, and medical images. Researchers extract graph models from various objects and compare them to identify unknown objects and scenes. The developments in bioinformatics also call for efficient mechanisms in querying a large number of biological pathways and protein interaction networks. These networks are usually very complex with multilevel structures embedded [11]. All these applications indicate the importance and the broad usage of graph database and its accompanied similarity search system.

While the motif discovery in graph datasets has been studied extensively, a systematic examination of graph query becomes equally important. A major kind of query in graph databases is *searching topological structures*, which cannot be answered efficiently using the existing database infrastructure. The indices built on the labels of vertices or edges are usually not selective enough to distinguish complicated, interconnected structures.

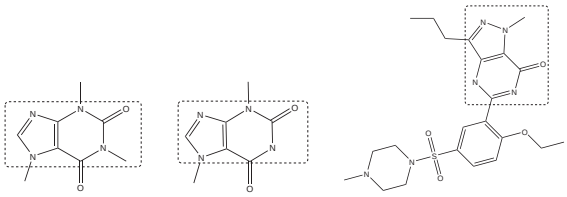
Due to the limitation of processing graph queries using the existing database techniques, tremendous efforts have been put into building practical graph query systems. In the past decade, sophisticated methods were developed to handle different kinds of structure search queries. Most of them fall into the following three categories: (1) full structure search: find the structure exactly the same as the query graph [1]; (2) substructure search: find structures that con-

*The work of the first and third authors was supported in part by NSF IIS-02-09199/03-08215, an IBM Faculty Award, and an IBM Summer Internship. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

SIGMOD 2005 June 14-16, 2005, Baltimore, Maryland, USA

tain the query graph, or vice versa [19, 20, 25]; and (3) full structure similarity search: find structures that are similar to the query graph [17, 24, 18]. These kinds of queries are very useful within their own applications. For example, in substructure search, a user may not know the exact composition of the full structure he wants, but requires that it contain a set of small functional fragments.

A common problem in substructure search is: what if no matches occur for a given query graph? In this situation, a subsequent query refinement process has to be taken in order to find the structures of interest. Unfortunately, it is often too time-consuming for a user to perform manual refinements. One solution is to ask the system to find graphs that nearly match the query. This similarity search strategy is more appealing since the user can first define the portion of the query for exact matching and let the system change the remaining portion slightly. The query could be relaxed progressively until a relaxation threshold is reached or a reasonable number of matches are found.



(a) caffeine (b) thesal (c) viagra

Figure 1: A Chemical Database

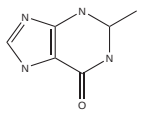


Figure 2: A Query Graph

EXAMPLE 1. *Figure 1 is a chemical dataset with three molecules. Figure 2 shows a substructure query. Obviously, no match exists for this query graph. If we relax the query with one edge miss, caffeine and thesal in Figures 1(a) and 1(b) will be good matches. If we relax the query further, the structure in Figure 1(c) could also be an answer.* ∎

Unfortunately, few systems are available for this kind of search scheme in large scale graph databases. The existing tools such as ChemIDplus could only provide the full structure similarity search and the exact substructure search. Other studies usually focus on how to compute the substructure similarity between two graphs efficiently [15]. This leads to the linear complexity with respect to the graph database size since each graph in the database has to be checked.

Given that the pairwise substructure similarity computation is very expensive, practically it is not affordable in a large database. A naïve solution is to form a set of subgraph queries with one or more edge deletions and then use the exact substructure search. This does not work when the number of deletions is more than 1. For example, if we allow three edges to be deleted in a 20-edge query graph, it may generate $\binom{20}{3} = 1140$ substructure queries, which is too expensive to check. Therefore, a better solution is greatly preferred.

Our Contributions. In this paper, we propose a feature-based structural filtering algorithm, called Grafil (**Gra**ph Similarity **Fil**tering) to perform substructure similarity search in a large scale graph database. Grafil models each query graph as a set of features and transforms the edge deletions into the feature misses in the query graph. With an upper bound on the maximum allowed feature misses, Grafil can filter many graphs directly without performing pairwise similarity computation. As a filtering technology, Grafil will improve the performance of the existing pairwise substructure similarity search systems as well as the naïve approach discussed above in large graph databases.

To facilitate the feature-based filtering, we introduce two data structures, feature-graph matrix and edge-feature matrix. The feature-graph matrix is an index structure to compute the difference in the number of features between a query graph and graphs in the database. The edge-feature matrix is built on the fly to compute a bound on the maximum allowed feature misses based on a query relaxation ratio.

It is shown that using too many features will not improve the filtering performance due to a *frequency conjugation* phenomenon identified through our study. This counter intuitive result inspires us to identify better combinations of features for filtering purposes. Therefore, we develop a multi-filter composition strategy, where each filter uses a distinct and complimentary subset of the features. The filters are constructed by a hierarchical, one dimensional clustering algorithm that groups features with similar selectivity into a feature set. The experimental result shows that the multi-filter strategy can improve performance significantly for a moderate relaxation ratio. To our best knowledge, we are not aware of any previous work using feature clustering to improve the filtering performance.

A significant contribution of this study is an examination of an increasingly important search problem in graph databases and the proposal of a feature-based filtering algorithm for efficient substructure similarity search. The development of our method explores the structural filtering algorithm in this new field. Moveover, the concept presented in Grafil can be applied to searching approximate, non-consecutive sequences, trees, and other complicated structures as well.

The rest of the paper is organized as follows. Section 2 defines the preliminary concepts. We introduce our structural filtering technique in Section 3, followed by an exploration of feature set selection using clustering techniques in Section 4. Section 5 describes the algorithm implementation, while our performance study is reported in Section 6. Related work is presented in Section 7. Section 8 concludes our study.

2. PRELIMINARY CONCEPTS

Graphs are widely used to represent complex structures that are difficult to model. In a labeled graph, vertices and edges are associated with attributes, called *label*s. The attributes could be tags in XML documents, atoms and bonds in chemical compounds, genes in biological networks, and object descriptors in images. Using labeled graphs or unlabeled graphs depends on the application need. However, the filtering algorithm we proposed in this paper can handle both types efficiently.

The *vertex set* of a graph G is denoted by $V(G)$ and the

edge set by $E(G)$. A label function, l, maps a vertex or an edge to a label. The *size* of a graph is defined by the number of edges it has, written as $|G|$. A graph G is a *subgraph* of G' if there exists a subgraph isomorphism from G to G', denoted by $G \subseteq G'$. G' is called a *supergraph* of G.

DEFINITION 1 (SUBGRAPH ISOMORPHISM). *A* subgraph isomorphism *is an injective function $f : V(G) \rightarrow V(G')$, such that (1) $\forall u \in V(G)$, $f(u) \in V(G')$ and $l(u) = l'(f(u))$, and (2) $\forall (u, v) \in E(G)$, $(f(u), f(v)) \in E(G')$ and $l(u, v) = l'(f(u), f(v))$, where l and l' is the label function of G and G', respectively. f is called an* embedding *of G in G'.*

Given a graph database and a query graph, we may not find a graph (or a few graphs) in the database that contains the whole query graph. Thus, it would be interesting to find graphs that contain the query graph approximately, which is a *substructure similarity search* problem. Based on our observation, this problem has two scenarios, similarity search and reverse similarity search.

DEFINITION 2 (SUBSTRUCTURE SIMILARITY SEARCH). *Given a graph database $D = \{G_1, G_2, \ldots, G_n\}$ and a query graph Q,* similarity search *is to discover all the graphs that approximately contain this query graph.* Reverse similarity search *is to discover all the graphs that are approximately contained by this query graph.*

Each type of search scenario has its own applications. In chemical informatics, similarity search is more popular, while reverse similarity search has key applications in pattern recognition. In this paper, we develop a structural filtering algorithm for similarity search. Nevertheless, our algorithm can also be applied to reverse similarity search with slight modifications.

To distinguish a query graph from the graphs in a database, we call the latter *target graphs*. The question is how to measure the substructure similarity between a target graph and the query graph. There are several similarity measures. We can classify them into three categories: (1) physical property-based, *e.g.*, toxicity and weight; (2) feature-based; and (3) structure-based. For the feature-based measure, domain-specific elementary structures are first extracted as features. Whether two graphs are similar is determined by the number of common elementary structures they have. For example, we can compare two compounds based on the number of benzene rings they have. Under this similarity definition, each graph is represented as a feature vector, $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T$, where x_i is the frequency of feature f_i. The distance between two graphs is measured by the distance between their feature vectors. Because of its efficiency, the feature-based similarity search has become a standard retrieval mechanism [24]. However, the feature-based approach only provides a very rough measure on structure similarity since it loses the global structural connectivity. Moreover, it is hard to build an "elementary structure" dictionary for a graph database, due to the lack of domain knowledge.

In contrast, the structure-based similarity measure directly compares the topology of two graphs, which is often costly to compute. However, since this measure takes structure connectivity fully into consideration, it is more accurate than the feature-based measure. Bunke and Shearer [3] used the maximum common subgraph to measure the full structure similarity. Researchers also developed the concept of graph edit distance and graph alignment distance by simulating the graph matching process in a way similar to the string matching process (akin to string edit distance and alignment distance). No matter what the definition is, the matching of two graphs can be regarded as a result of three edit operations: insertion, deletion, and relabeling. According to the substructure similarity search, each of these operations relaxes the query graph by removing or relabeling one edge (insertion does not change the query graph). Without loss of generality, we take *the percentage of maximum retained edges in the query graph* as a similarity measure.

DEFINITION 3 (RELAXATION RATIO). *Given two graphs G and Q, if P is the maximum common subgraph*[1] *of G and Q, then the substructure similarity between G and Q is defined by $\frac{|E(P)|}{|E(Q)|}$, and $1 - \frac{|E(P)|}{|E(Q)|}$ is called relaxation ratio.*

EXAMPLE 2. *Consider the target graph in Figure 1(a) and the query graph in Figure 2. Their maximum common subgraph has 11 edges. Thus, the substructure similarity between these two graphs is around 92% with respect to the query graph. That also means if we relax the query graph by 8%, Figure 1(a) contains the relaxed query graph. Note that the relaxation ratio is not symmetric. We can also compute the similarity of graphs in Figures 1(b) and 1(c) with the query graph. It is 92% and 67%, respectively.* ■

In this paper, we want to examine how to build a connection between the structure-based measure and the feature-based measure so that we can use the feature-based measure to screen the database before performing the expensive pairwise structure-based similarity computation. Using this strategy, we take the advantages of both measures: efficiency from the feature-based measure and accuracy from the structure-based measure.

3. STRUCTURAL FILTERING

Given a relaxed query graph, the major target of our algorithm is to filter as many graphs as possible using a feature-based approach. The features discussed here could be paths [19], discriminative frequent structures [25], elementary structures, or any structures indexed in graph databases. Previous work did not investigate the connection between the structure-based similarity measure and the feature-based similarity measure. In this study, we explicitly transform the query relaxation ratio to the misses of indexed features, thus building a connection between these two measures.

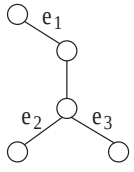


Figure 3: A Sample Query

Let us first check an example. Figure 3 shows a query graph and Figure 4 depicts three structural fragments. Assume that these fragments are indexed as features in a graph

[1]The maximum common subgraph is not necessarily connected.

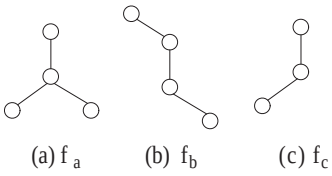


Figure 4: Features

database. For simplicity, we ignore all the label information in this example. The symbols e_1, e_2, and e_3 in Figure 3 do not represent labels but edges themselves. Suppose we cannot find any match for this query graph in a graph database. Then a user may relax one edge, e_1, e_2, or e_3, through a deletion or relabeling operation. He/she may deliberately retain the middle edge, because the deletion of that edge may break the relaxed query graph into pieces, which certainly should be avoided. Because the relaxation can take place among e_1, e_2, and e_3, we are not sure which feature will be affected by this relaxation. However, no matter which edge is relaxed, the relaxed query graph should have at least three occurrences of these features. Equivalently, we say that the relaxed query graph may *miss* at most four occurrences of these features in comparison with the original query graph, which have seven occurrences: one f_a, two f_b's, and four f_c's. Using this information, we can discard graphs that do not contain at least three occurrences of these features. We name the above filtering concept *feature-based structural filtering*.

3.1 Feature-Graph Matrix Index

Here we introduce an index structure, referred to as the *feature-graph matrix*, to facilitate the feature-based filtering. Each column of the matrix corresponds to a target graph in the graph database, while each row corresponds to a feature being indexed. Each entry records the number of the embeddings of a specific feature in a target graph. Suppose we have a sample database with four graphs, G_1, G_2, G_3, and G_4. Figure 5 shows an example of feature-graph matrix index. For instance, G_1 has two embeddings of f_c. The feature-graph matrix index is easily maintainable: as each time a new graph is added to the graph database, only an additional column needs to be added.

	G_1	G_2	G_3	G_4
f_a	0	1	0	0
f_b	0	0	1	0
f_c	2	3	4	4

Figure 5: Feature-Graph Matrix Index

Using the feature-graph matrix, we can apply the feature-based filtering on any query graph against a target graph in the database using any subset of the indexed features. Consider the query shown in Figure 3 with one edge relaxation. According to the feature-graph matrix in Figure 5, even if we do not know the structure of G_1, we can filter G_1 immediately based on the features included in G_1, since G_1 only has two of all the embeddings of f_a, f_b, and f_c. This feature-based filtering process is not involved with any costly structure similarity checking. The only computation needed is to retrieve the features from the index that belong to a query graph and compute the possible feature misses for a relaxation ratio. Since our filtering algorithm is fully built on the feature-graph matrix index, we need not access the physical database unless we want to calculate the accurate substructure similarity.

We implement feature-graph matrix based on a list, where each element points to an array representing the row of the matrix. Using this implementation, we can flexibly insert and delete features without rebuilding the whole index. In the next subsection, we will present the general framework of processing similarity search, and illustrate the position of our structural filtering algorithm in this framework.

3.2 Framework

Given a graph database and a query graph, the substructure similarity search can be performed in the following four steps.

1. **Index construction**: Select small structures as features in the graph database, and build the feature-graph matrix between the features and the graphs in the database.

2. **Feature miss estimation**: Determine the indexed features belonging to the query graph, select a feature set (i.e., a subset of the features), calculate the number of selected features contained in the query graph and then compute the upper bound of feature misses if the query graph is relaxed with one edge deletion or relabeling. This upper bound is written as d_{max}. Some portion of the query graph can be specified as not to be altered, *e.g.*, key functional structures.

3. **Query processing**: Use the feature-graph matrix to calculate the difference in the number of features between each graph G in the database and query Q. If the difference is greater than d_{max}, discard graph G. The remaining graphs constitute a candidate answer set, written as C_Q. We then calculate substructure similarity using the existing algorithms and prune the false positives in C_Q.

4. **Query relaxation**: Relax the query further if the user needs more matches than those returned from the previous step; iterate Steps 2 to 4.

The feature-graph matrix in Step 1 is built beforehand and can be used by any query. The similarity search for a query graph takes place in Step 2 and Step 3. The filtering algorithm proposed should return the candidate answer set as small as possible since the cost of the accurate similarity computation is proportional to the size of the candidate set. Quite a lot of work has been done at calculating the pairwise substructure similarity. Readers are referred to the related work in [15, 8, 18].

In the step of feature miss estimation, we calculate *the number of features* in the query graph. One feature may have multiple occurrences in a graph; thus, we use *the number of embeddings of a feature* as a more precise term. In this paper, these two terms are used interchangeably for convenience.

In the rest of this section, we will introduce how to estimate feature misses by translating it into the maximum coverage problem. The estimation is further refined through a branch-and-bound method. In Section 4, we will explore the opportunity of using different feature sets to improve filtering efficiency.

3.3 Feature Miss Estimation

Substructure similarity search is akin to approximate string matching. In approximate string matching, filtering algorithms such as q-gram achieve the best performance because they do not inspect all the string characters. However, filtering algorithms only work for the moderate relaxation ratio and need a validation algorithm to check the actual matches [14]. Similar arguments also apply to our structural filtering algorithm in substructure similarity search. Fortunately, since we are doing substructure search instead of full structure similarity search, usually the relaxation ratio is not very high in our problem setting.

A string with q characters is called a *q-gram*. A typical q-gram filtering algorithm builds an index for all q-grams in a string database. A query string Q is broken into a set of q-grams, which are compared against the q-grams of each target string in the database. If the difference in the number of q-grams is greater than the following threshold, Q will not match this string within k edit distance.

Given two strings P and Q, if their edit distance is k, their difference in the number of q-grams is at most kq [21].

It would be interesting to check *whether we can similarly derive a bound for size-q substructures*. Unfortunately, we may not draw a succinct bound like the one given to q-grams due to the following two issues. First, in substructure similarity search, the space of size-q subgraphs is explosive with respect to q. This contrasts with the string case where the number of q-grams in a string is linear to its length. Secondly, even if we index all of the size-q subgraphs, the above q-gram bound will not be valid since the graph does not have the linearity that the string does.

	f_a	$f_{b(1)}$	$f_{b(2)}$	$f_{c(1)}$	$f_{c(2)}$	$f_{c(3)}$	$f_{c(4)}$
e_1	0	1	1	1	0	0	0
e_2	1	1	0	0	1	0	1
e_3	1	0	1	0	0	1	1

Figure 6: Edge-Feature Matrix

We use an *edge-feature matrix* to build a map between edges and features for a query graph. In this matrix, each row represents an edge while each column represents an embedding of a feature. Figure 6 shows the matrix built for the query graph in Figure 3 and the features shown in Figure 4. All the embeddings of features are recorded. For example, the second and the third columns are two embeddings of feature f_b in the query graph. The first embedding of f_b *cover*s edges e_1 and e_2 while the second covers edges e_1 and e_3. The middle edge does not appear in the edge-feature matrix if a user prefers retaining it. We say that an edge e_i *hit*s a feature f_j if f_j covers e_i.

It is not expensive to build the edge-feature matrix on-the-fly as long as the number of features is small. Whenever an embedding of a feature is discovered, we append a new column to the matrix. The feature miss estimation problem can be stated as follows: *Given a query graph Q and a set of features contained in Q, if the relaxation ratio is θ, what is the maximal number of features that can be missed*? In fact, it is the maximum number of columns that can be hit by k rows in the edge-feature matrix, where $k = \lfloor \theta \cdot |G| \rfloor$. This is a classic maximum coverage (or set k-cover) problem, which has been proved NP-complete. The optimal solution that finds the maximal number of feature misses can be approximated by a greedy algorithm. The greedy algorithm first selects a row that hits the largest number of columns and then removes this row and the columns covering it. This selection and deletion operation is repeated until k rows are removed. The number of columns removed by this greedy algorithm provides a way to estimate the upper bound of feature misses.

Algorithm 1 GreedyCover

Input: Edge-feature Matrix M,
Maximum edge relaxations k.
Output: The number of feature misses W_{greedy}.

1: let $W_{greedy} = 0$;
2: **for each** $l = 1 \ldots k$ **do**
3: select row r_l that maximizes $|M(r_l, \cdot)|$;
4: $W_{greedy} = W_{greedy} + |M(r_l, \cdot)|$;
5: **for each** column c s.t. $M(r_l, c) = 1$ **do**
6: set $M(\cdot, c)$=0;
7: **return** W_{greedy};

Algorithm 1 shows the pseudo-code of the greedy algorithm. Let $M(r, c)$ be the entry in the r_{th} row, c_{th} column of matrix M. $M(r, \cdot)$ denotes the r_{th} row vector of matrix M, while $M(\cdot, c)$ denotes the c_{th} column vector of matrix M. $|M(r, \cdot)|$ represents the number of non-zero entries in $M(r, \cdot)$.

THEOREM 1. *Let W_{greedy} and W_{opt} be the total feature misses computed by the greedy solution and by the optimal solution. We have*

$$W_{greedy} \geq [1 - (1 - \frac{1}{k})^k] \, W_{opt} \geq (1 - \frac{1}{e}) \, W_{opt}, \quad (1)$$

where k is the number of edge relaxations.

Proof. [4] ∎

Theoretic result shows that the optimal solution cannot be approximated in polynomial time within a ratio of $(e/(e-1) - o(1))$ unless $P = NP$ [5]. We rewrite the inequality in Theorem 1.

$$\begin{aligned} W_{opt} &\leq \frac{1}{1 - (1 - \frac{1}{k})^k} \, W_{greedy} \\ W_{opt} &\leq \frac{e}{e - 1} \, W_{greedy} \\ W_{opt} &\leq 1.6 \, W_{greedy} \end{aligned} \quad (2)$$

Traditional applications of the maximum coverage problem focus on how to approximate the optimal solution as

better as possible. Here we are only interested in the upper bound of the optimal solution. Let $\max_r |M(r,\cdot)|$ be the maximum number of features that one edge hits. Obviously, W_{opt} should be less than k times of this number,

$$W_{opt} \leq k \times \max_r |M(r,\cdot)|. \quad (3)$$

3.4 Estimation Refinement

A tight bound of W_{opt} is critical to the filtering performance. A tighter bound often leads to a smaller set of candidate graphs. Although the bound derived by the greedy algorithm cannot be improved asymptotically any more, we may still improve the greedy algorithm in practice.

Let $W_{opt}(M,k)$ be the optimal value of the maximum feature misses for k edge relaxations. Suppose row r_l maximizes $|M(r_l,\cdot)|$. Let M' be M except $M'(r_l,\cdot) = 0$ and $M'(\cdot,c) = 0$ for any column c that is hit by r_l, and M'' be M except $M''(r_l,\cdot) = 0$.

Any optimal solution that leads to W_{opt} should be in the following two cases: (Case 1) r_l is selected in this solution; or (Case 2) r_l is not selected (we call r_l disqualified for the optimal solution). In the first case, the optimal solution should also be the optimal solution for the remaining matrix M'. That is, $W_{opt}(M,k) = |M(r_l,\cdot)| + W_{opt}(M', k-1)$. $k-1$ means that we need to remove the remaining $k-1$ rows from M' since row r_l is selected. In the second case, the optimal solution for M should be the optimal solution for M'', *i.e.*, $W_{opt}(M,k) = W_{opt}(M'',k)$. k means that we still need to remove k rows from M'' since row r_l is disqualified. We call the first case the *selection step*, and the second case the *disqualifying step*. Since the optimal solution is to find the maximum number of columns that are hit by k edges, W_{opt} should be equal to the maximum value returned by these two steps. Therefore, we can draw the following conclusion.

LEMMA 1.

$$W_{opt}(M,k) = max \begin{cases} |M(r_l,\cdot)| + W_{opt}(M', k-1), \\ W_{opt}(M'', k). \end{cases} \quad (4)$$

Lemma 1 suggests a recursive solution to calculating W_{opt}. It is equivalent to enumerating all the possible combinations of k rows in the edge-feature matrix, which may be very costly. However, it is worth exploring the top levels of this recursive process, especially for the case where most of the features intensively cover a set of common edges. For each matrix M' (or M'') that is derived from the original matrix M after several recursive calls in Lemma 1, M' encountered interleaved selection steps and disqualifying steps. Suppose M' has h selected rows and b disqualified rows. We restrict h to be less than H and b to be less than B, where H and B are predefined constants, and $H + B$ should be less than the number of rows in the edge-feature matrix. In this way, we can control the depth of the recursion.

Let $W_{apx}(M,k)$ be the upper bound of the maximum feature misses calculated using Equations (2) and (3), where M is the edge-feature matrix and k is the number of edge relaxations. We formulate the above discussion in Algorithm 2. Line 7 selects row r_l while Line 8 disqualifies row r_l. Lines 7 and 8 correspond to the selection and disqualifying steps shown in Lemma 1. Line 9 calculates the maximum value of the result returned by Lines 7 and 8. Meanwhile, we can also use the greedy algorithm to get the upper bound of W_{opt} directly, as Line 10 does. Algorithm 2 returns the best estimation we can get. The condition in Line 1 will terminate the recursion when it selects H rows or when it disqualifies B rows. Algorithm 2 is a classical branch-and-bound approach.

Algorithm 2 $W_{est}(M,k,h,b)$

Input: Edge-feature Matrix M,
Number of edge relaxations k,
h selection steps and b disqualifying steps.
Output: Maximum feature misses W_{est}.

1: **if** $b \geq B$ or $h \geq H$ **then**
2: **return** $W_{apx}(M,k)$;
3: select row r_l that maximizes $|M(r_l,\cdot)|$;
4: let $M' = M$ and $M'' = M$;
5: set $M'(r_l,\cdot)$=0 and $M'(\cdot,c)$=0 for any c if $M(r_l,c) = 1$;
6: set $M''(r_l,\cdot)$=0;
7: $W_1 = |M(r_l,\cdot)| + W_{est}(M', k, h+1, b)$;
8: $W_2 = W_{est}(M'', k, h, b+1)$;
9: $W_a = \max(W_1, W_2)$;
10: $W_b = W_{apx}(M,k)$;
11: **return** $\min(W_a, W_b)$;

We select parameters H and B such that H is less than the number of edge relaxations, and $H + B$ is less than the number of rows in the matrix. Algorithm 2 is initialized by $W_{est}(M,k,0,0)$. The bound obtained by Algorithm 2 is not greater than the bound derived by the greedy algorithm since we intentionally select the smaller one in Lines 10-11. On the other hand, $W_{est}(M,k,0,0)$ is not less than the optimal value since Algorithm 2 is just a simulation of the recursion in Lemma 1, and at each step, it has a greater value. Therefore, we can draw the following conclusion.

LEMMA 2. *Given two non-negative integers H and B in Algorithm 2, if $H \leq k$ and $H + B \leq n$, where k is the number of edge relaxations and n is the number of rows in the edge-feature matrix M, we have*

$$W_{opt}(M,k) \leq W_{est}(M,k,0,0) \leq W_{apx}(M,k). \quad (5)$$

Given a query Q and the maximum allowed selection and disqualifying steps, H and B, the cost of computing W_{est} is irrelevant to the number of the graphs in a database. Thus, the cost of feature miss estimation remains constant with respect to the database size.

3.5 Frequency Difference

Assume that f_1, f_2, ..., f_n form the feature set used for filtering. Once the upper bound of feature misses is obtained, we can use it to filter graphs in our framework. Given a target graph G and a query graph Q, let $\mathbf{u} = [u_1, u_2, \ldots, u_n]^T$ and $\mathbf{v} = [v_1, v_2, \ldots, v_n]^T$ be their corresponding feature vectors, where u_i and v_i are the frequency (i.e., the number of embeddings) of feature f_i in graphs G and Q. Figure 7 shows the two feature vectors $\mathbf{u}$ and $\mathbf{v}$. As mentioned before, for any feature set, the corresponding feature vector of a target graph can be obtained from the feature-graph matrix directly without scanning the graph database.

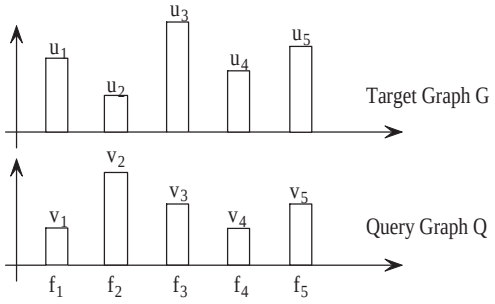


Figure 7: Frequency Difference

We want to know how many more embeddings of feature f_i appear in the query graph, compared to the target graph. Equation (6) calculates this frequency difference for feature f_i,

$$r(u_i, v_i) = \begin{cases} 0, \ if \ u_i \geq v_i, \\ v_i - u_i, \ otherwise. \end{cases} \quad (6)$$

For the feature vectors shown in Figure 7, $r(u_1, v_1) = 0$; we do not take the extra embeddings from the target graph into account. The summed frequency difference of each feature in G and Q is written as $d(G, Q)$. Equation (7) sums up all the frequency differences,

$$d(G, Q) = \sum_{i=1}^{n} r(u_i, v_i). \quad (7)$$

Suppose the query can be relaxed with k edges. Algorithm 2 estimates the upper bound of allowed feature misses. If $d(G, Q)$ is greater than that bound, we can conclude that G does not contain Q within k edge relaxations. For this case, we do not need to perform any complicated structure comparison between G and Q. Since all the computations are done on the preprocessed information in the indices, the filtering actually is very fast.

4. FEATURE SET SELECTION

In Section 3, we have explored the basic filtering framework and our bounding technique. For a given feature set, the performance could not be improved further unless we have a tighter bound of allowed feature misses. Nevertheless, we have not explored the opportunities of composing filters based on different feature sets. An interesting question is "*should we use all the features together in a single filter*?" or "*does a filter achieve good filtering performance if all the features are used together*?" Intuitively, such a strategy would improve the performance since all the available information is used. Unfortunately, very different results are observed in our experiments: Using all the features together in one filter will deteriorate the performance rather than improve it. This counter-intuitive result is observed universally. In this section, we will explain the reason behind this phenomenon and discuss how to solve this issue by separating features with different characteristics to construct multiple filters.

Let $\mathbf{u} = [u_1, u_2, \ldots, u_n]^T$ and $\mathbf{v} = [v_1, v_2, \ldots, v_n]^T$ be the feature vectors built from a target graph G and a query graph Q. Assume that d_{max} is the maximum allowed feature misses. The feature space of a candidate graph can be described as follows,

$$r(u_1, v_1) + r(u_2, v_2) + \ldots + r(u_n, v_n) \leq d_{max}. \quad (8)$$

Any graph whose feature vector satisfies the above inequality is a candidate answer for the query graph. Let P be the maximum common subgraph of G and Q. Vector $\mathbf{u}' = [u'_1, u'_2, \ldots, u'_n]^T$ is its feature vector. If G contains Q within the relaxation ratio, P should contain Q within the relaxation ratio as well, *i.e.*,

$$r(u'_1, v_1) + r(u'_2, v_2) + \ldots + r(u'_n, v_n) \leq d_{max}. \quad (9)$$

Since for any feature f_i, $u_i \geq u'_i$, we have

$$\begin{aligned} r(u_i, v_i) &\leq r(u'_i, v_i), \\ \sum_{i=1}^{n} r(u_i, v_i) &\leq \sum_{i=1}^{n} r(u'_i, v_i). \end{aligned}$$

Inequality (9) is stronger than Inequality (8). Mathematically, we should check Inequality (9) instead of Inequality (8). However, we do not want to calculate P, the maximum common subgraph of G and Q, beforehand, due to its computational cost. Inequality (8) is the only choice we have. Assume that Inequality (9) does not hold for graph P, and furthermore, there exists a feature f_i such that its frequency in P is too small to make Inequality (9) hold. However, we can still make Inequality (8) true for graph G, if we compensate the misses of f_i by adding more occurrences of another feature f_j in G. We call this phenomenon *feature conjugation*. Feature conjugation likely takes place in our filtering algorithm since the filtering does not distinguish the misses of a single feature, but a collective set of features. As one can see, because of feature conjugation, we may fail to filter some graphs that do not satisfy the query requirement.

EXAMPLE 3. *Assume that we have a graph G that contains the sample query graph in Figure 3 with edge e_3 relaxed. In this case, G must have one embedding of feature f_b and two embeddings of f_c (f_b and f_c are in Figure 4). However, we may slightly change G such that it does not contain f_b but has one more embedding of f_c. This is exactly what G_4 has in Figure 5. The feature conjugation takes place when the miss of f_b is compensated by the addition of one more occurrence of f_c. In such a situation, Inequality (8) is still satisfied for G_4, although Inequality (9) is not.*

However, if we can divide the features in Figure 4 into two groups, we can partially solve the feature conjugation problem. Let group A contain feature f_a and f_b, and group B contain feature f_c only. For any graph containing the query shown in Figure 3 with one edge relaxation (edge e_1, e_2 or e_3), it must have one embedding in Group A. Using this constraint, we can drop G_4 in Figure 5 since G_4 does not have any embedding of f_a or f_b.

The above example also implies that the filtering power may be weakened if we deploy all the features in one filter. A feature has filtering power if its frequency in a target graph is less than its frequency in the query graph; otherwise, it does not help the filtering. Unfortunately, a feature that is good for some graph may not be good for other graphs in the database. Therefore, we want to find a set of features that are uniformly good for a large number of graphs. We use *selectivity* defined below to measure the filtering power

of a feature f for all the graphs in the database. Using the feature-graph matrix, it takes little time to compute since we need not access the physical database.

Definition 4 (Selectivity). *Given a graph database D, a query graph Q, and a feature f, the selectivity of f is defined by its average frequency difference within D and Q, written as $\delta_f(D,Q)$. $\delta_f(D,Q)$ is equal to the average of $r(u,v)$, where u is a variable denoting the frequency of f in a graph belonging to D, v is the frequency of f in Q, and r is defined in Equation (6).*

To put features with the same filtering power in a single filter, we have to group features with similar selectivity into the same feature set. Before we elaborate this idea, we first conceptualize three general rules that provide guidance on feature set selection.

Rule 1. Select a large number of features.

Rule 2. Make sure features cover the query graph uniformly.

Rule 3. Separate features with different selectivity.

Obviously, the first rule is necessary. If only a small number of features are selected, the maximum allowed feature misses may become very close to $\sum_{i=1}^{n} v_i$. In that case, the filtering algorithm loses its pruning power. The second rule is more subtle than the first one, but based on the same intuition. If most of the features cover several common edges, the relaxation of these edges will make the maximum allowed feature misses too big. The third rule has been examined above. Unfortunately, these three criteria are not consistent with each other. For example, if we use all the features in a query, the second and the third rules will be violated since sparse graphs such as chemical structures have features concentrated in the graph center. Secondly, low selective features deteriorate the potential filtering power from high selective ones due to frequency conjugation. On the other hand, we cannot use the most selective features alone because we may not have enough highly selective features in a query.

Since using a single filter with all the features included is not expected to perform well, we devise a multi-filter composition strategy: Multiple filters are constructed and coupled together, where each filter uses a distinct and complimentary feature set. The three rules we have examined provide general guidance on how to compose the feature set for each of the filters. The task of feature set selection is to make a trade-off among these rules. We may group features by their size to create feature sets. This simple scheme satisfies the first and the second rules. Usually the selectivity of features with varying sizes is different. Thus it also roughly meets the third rule. This simple scheme actually works as verified by our experiments. However, we may go one step further by first grouping features with similar size and then clustering them based on their selectivity to form feature sets.

We devise a simple hierarchical agglomerative clustering algorithm based on the selectivity of the features. The final clusters produced represent the distinct feature sets for the different filters. The algorithm starts at the bottom, where each feature is an individual cluster. At each level, it recursively merges the two closest clusters into a single cluster. The "closest" means their selectivity is the closest. Each cluster is associated with two parameters: the average selectivity of the cluster and the number of features associated with it. The selectivity of two merged clusters is defined by a linear interpolation of their own selectivity,

$$\frac{n_1\delta_1 + n_2\delta_2}{n_1 + n_2}, \tag{10}$$

where n_1 and n_2 are the number of features in the two clusters, and δ_1 and δ_2 are their corresponding selectivity.

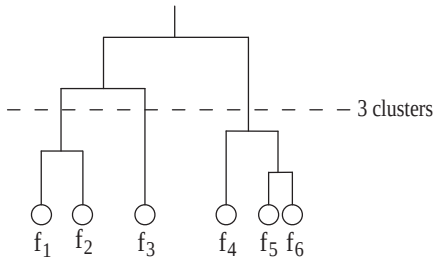


Figure 8: Hierarchical Agglomerative Clustering

Features are first sorted according to their selectivity and then clustered hierarchically. Assume that $\delta_{f_1}(D,Q) \leq \delta_{f_2}(D,Q) \leq \delta_{f_3}(D,Q) \leq \ldots$. Figure 8 shows a hierarchical clustering tree. In the first round, f_5 is merged with f_6. In the second round, f_1 is merged with f_2. After that, f_4 is merged with the cluster formed by f_5 and f_6 if f_4 is the closest one to them. Since the clustering is performed in one dimension, it is very efficient to build.

5. ALGORITHM IMPLEMENTATION

In this section, we formulate our filtering algorithm, called Grafil (**Gra**ph Similarity **Fil**tering).

Grafil consists of two components: a base component and a clustering component. Both of them apply the multi-filter composition strategy. The base component generates feature sets by grouping features of the same size and uses them to filter graphs based on the upper bound of allowed feature misses derived in Section 3.4. It first applies the filter using features with one edge, then the one using features with two edges, and so on. We denote the base component by Grafil-base. The clustering component combines the features whose size differs at most by 1, and groups them by their selectivity. Algorithm 3 sketches the outline of Grafil. F_i in Line 2 represents the set of features with i edges. Lines 2-4 form the base component and Lines 5-11 form the clustering component. Once the hierarchical clustering is done on features with i edges and $i+1$ edges, Grafil divides them into three groups with high selectivity, medium selectivity, and low selectivity. A separate filter is constructed based on each group of features. For the hierarchical clusters shown in Figure 8, Grafil will choose f_1 and f_2 as group 1, f_3 as group 2, and f_4, f_5 and f_6 as group 3.

To deploy the multiple filters, Grafil can run in a pipeline mode or a parallel mode. The diagram in Figure 9 depicts the pipeline mode, where the candidate answer set returned from the current step is pumped into the next step.

Algorithm 3 is written in the pipeline mode. We can change it to the parallel mode by replacing Line 4 and Line 11 with the following statement,

$$C_Q = \{\, G | d(G,Q) \leq d_{max}, G \in D \,\},$$

Algorithm 3 Grafil

Input: Graph database D, Feature set F,
Maximum feature size $maxL$, and
A relaxed query Q.
Output: Candidate answer set C_Q.

1: let $C_Q = D$;
2: **for each** feature set F_i, $i \leq maxL$ **do**
3: calculate the maximum feature misses d_{max};
4: $C_Q = \{ G | d(G, Q) \leq d_{max}, G \in C_Q \}$;
5: **for each** feature set $F_i \bigcup F_{i+1}$, $i < maxL$ **do**
6: compute the selectivity based on C_Q;
7: do the hierarchical clustering on features in $F_i \bigcup F_{i+1}$;
8: cluster features into three groups, X_1, X_2, and X_3;
9: **for each** cluster X_i **do**
10: calculate the maximum feature misses d_{max};
11: $C_Q = \{ G | d(G, Q) \leq d_{max}, G \in C_Q \}$;
12: **return** C_Q;

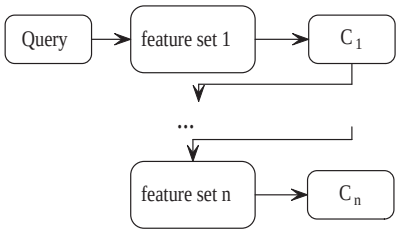


Figure 9: Filtering Pipeline

and C_Q in Line 6 with D. With these modifications, Grafil can be parallelized directly. The final candidate answer set is the intersection of the candidate sets returned by each filter. However, there is a slight difference between the pipeline mode and the parallel mode. Grafil in the pipeline mode can achieve a smaller candidate answer set. The reason is the clustering component (Line 6) in the pipeline mode calculates the selectivity based on the candidate graphs returned in the previous step, while the parallel mode does not. We will show the performance impact raised by this difference in the next section.

6. EMPIRICAL STUDY

In this section, we conduct several experiments to examine the properties of Grafil. The performance of Grafil is compared with two algorithms based on a single filter: one using individual edges as features (denoted as Edge) and the other using all features of a query graph (denoted as Allfeature). Many similarity search algorithms [8, 18] can only apply the edge-based filtering mechanism since the mapping between edge deletion/relabeling and feature misses was not established before this study. In fact, the edge-based filtering approach can be viewed as a degenerate case of the feature-based approach using a filter with single edge features. By demonstrating the conditions where Grafil can filter more graphs than Edge and Allfeature, we show that Grafil can substantially improve substructure similarity search in large graph databases.

Two kinds of datasets are used through our empirical study: one real dataset and a series of synthetic datasets. The real dataset is an AIDS antiviral screen dataset containing the topological structures of chemical compounds. This dataset is available on the website of the Developmental Therapeutics Program (NCI/NIH)[2]. In this dataset, thousands of compounds have been checked for evidence of anti-HIV activity. The dataset has around 44,000 structures. The synthetic data generator was kindly provided by Kuramochi et al. [12]. The generator allows the user to specify various parameters, such as the database size, the average graph size, and the label types, to examine the scalability of Grafil.

We built Grafil based on the gIndex algorithm [25]. gIndex first mines frequent subgraphs with size up to 10 and then retains discriminative ones as indexing features. We thus take the discriminative frequent structures as our indexing features. Certainly, other kinds of features can be used in Grafil too, since Grafil does not rely on the kinds of features to be used. For example, Grafil can also take paths [19] as features to perform the similarity search.

Through our experiments, we illustrate that

1. Grafil can efficiently prune the search space for substructure similarity search and generate up to 15 times less candidate graphs than the alternatives in the chemical dataset.
2. Bound refinement and feature set selection for the multiple filter approach developed by Grafil are both effective.
3. Grafil performs much better for graphs with a small number of labels.
4. The single filter approach using all features together does not perform well due to the frequency conjugation problem identified in Section 4. Neither does the approach using individual edges as features due to their low selectivity.

Experiments on the Chemical Compound Dataset.

We first examine the performance of Grafil over the AIDS antiviral database. The test dataset consists of 10,000 graphs that are randomly selected from the AIDS screen database. These graphs have 25 nodes and 27 edges on average. The maximum one has 214 nodes and 217 edges in total. Note that in this dataset most of the atoms are carbons and most of the edges are carbon-carbon bonds. This characteristic makes the substructure similarity search very challenging. The query graphs are directly sampled from the database and are grouped together according to their size. We denote the query set by Q_m, where m is the size of the graphs in Q_m. For example, if the graphs in a query set have 20 edges each, the query set is written as Q_{20}. Different from the experiment setting in [25], the edges in our dataset are assigned with edge types, such as single bond, double bond, and so on. By doing so, we reduce the number of exact substructure matches for each query graph. This is exactly the case where the substructure similarity search will help: find a relatively large matching set by relaxing the query graph a little bit. When a user submits a substructure similarity query, he may not allow arbitrary deletion of some critical atoms and bonds. In order to simulate this constraint, we retain 25% of the edges in each query graph.

[2]http://dtpsearch.ncifcrf.gov/FTP/AIDO99SD.BIN

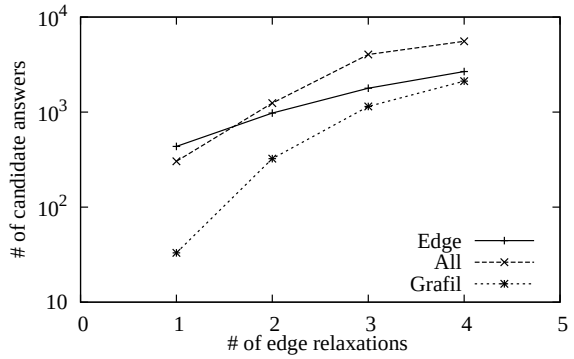


Figure 10: Structure Query with 16 edges

We made some slight modifications on the Allfeature approach by removing features whose size is greater than the query graph size divided by the number of edge relaxations. This modification improves the performance of Allfeature. Figure 10 depicts the performance of Edge, Allfeature and Grafil for the query set Q_{16}. The X-axis shows the number of edge relaxations done for a query graph. The Y-axis shows the average number of candidate graphs returned by each algorithm. As explained in Section 1, it is always preferable to filter as many graphs as possible before performing real similarity computation. The accurate pairwise similarity checking is very time-consuming. If we allow one edge to be lost for queries with 16 edges, the Edge approach can prune 96% of the dataset while Grafil can prune 99.7%. If a user wants to check whether there are real matches in the remaining 0.3% of the dataset, they can apply the similarity computation tools developed in [8, 18] to check them. If they are not satisfied with the result, the user can relax the edge loss to 3. The Edge approach will return 18% of the dataset and Grafil will return 11% of the dataset. The running time of Grafil is negligible in comparison to the accurate substructure similarity computation. Using the feature-graph matrix, the filtering stage takes less than 1 second per query for this query set.

Figure 10 demonstrates that Grafil outperforms Edge and Allfeature significantly when the relaxation ratio is within 2 edges by a factor of 5-10 times. However, when the relaxation ratio increases, the performance of Grafil is close to Edge. The reason is very simple. The structures of chemical compounds are very sparse, and are mainly tree-structured with several loops embedded. If we allow three edges to be deleted or relabeled for a query that has 16 edges, it is likely that the relaxed edges divide the query graph into four pieces. Each piece may only have 4-5 edges in total which virtually cannot hold any significant features. These small pieces have very weak pruning power. Eventually, only the number of remaining edges in a query graph counts. Therefore, we expect that when the relaxation ratio increases, Grafil will have the performance close to Edge. However, at this time the number of matches will increase dramatically. Since a user may not like the relaxed query graph to deviate too far away from her/his need, she/he may stop with a small relaxation ratio. Take the query set Q_{16} as an example, on average it only has 1.2 exact substructure matches. If we allow two edges to be relaxed, it has 12.8 matches on average, which may be enough for examination. And this figure is proportional to the number of graphs in the database.

The above result is further confirmed through another experiment. We test the queries that have 20 edges. Figure 11 shows the performance comparison among these three algorithms. Again, Grafil outperforms Edge and Allfeature.

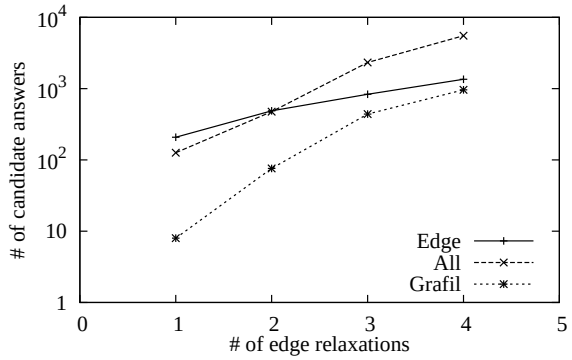


Figure 11: Structure Query with 20 edges

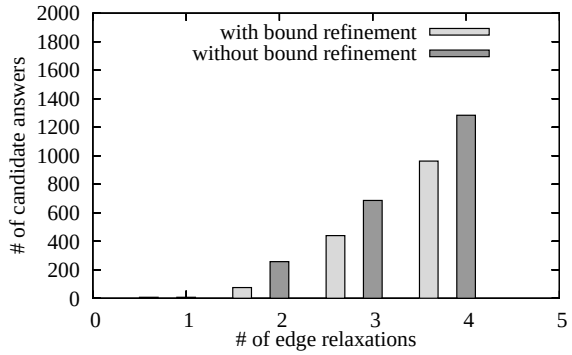


Figure 12: Feature Miss Estimation Refinement

Having examined the overall performance of Grafil in comparison with the other two approaches, we test the effectiveness of each component of Grafil. We take Q_{20} as a testing set. Figure 12 shows the performance difference before and after we apply the bound refinement in Grafil. In this experiment, we set the maximum number of selection steps (H) at 2, and the maximum number of disqualifying steps (B) at 6. It seems that the bound refinement makes critical improvement when the relaxation ratio is below 20%. At the high relaxation ratio, bound refinement does not have apparent effects. As explained in the previous experiments, Grafil mainly relies on the edge feature set to filter graphs when the ratio is high. In this case, bound refinement will not be effective at all. In summary, it is worth doing bound refinement for the moderate relaxation ratio.

Figure 13 shows the filtering ratio obtained by applying the clustering component in Grafil. Let C_Q and C'_Q be the candidate answer set returned by Grafil (with the clustering component) and Grafil-base (with the base component only), respectively. The filtering ratio in the figure is defined by $\frac{|C'_Q|}{|C_Q|}$. The test is performed on the query set Q_{20}. Overall, Grafil with the clustering component is 40%–120% better

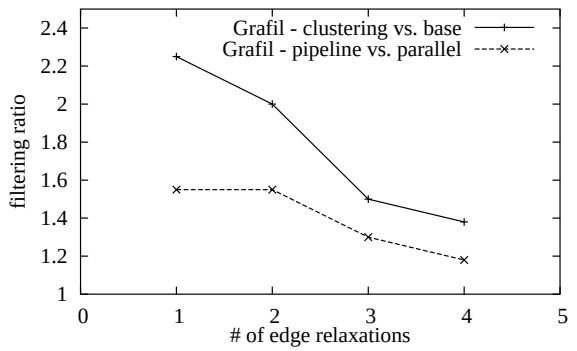


Figure 13: Clustering and Pipeline Improvement

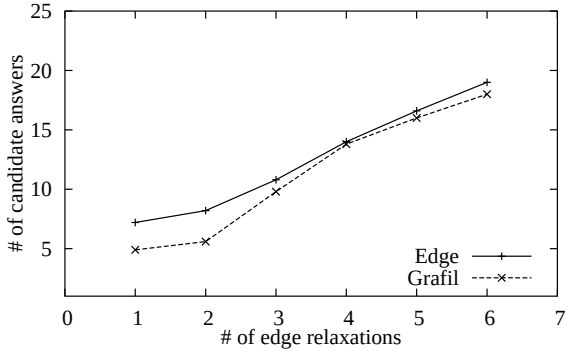


Figure 14: Numerous Types of Labels

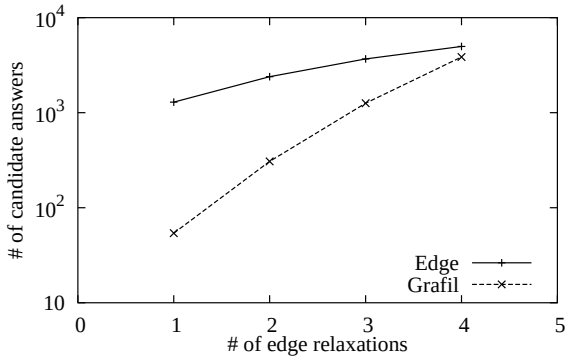


Figure 15: Few Types of Labels

than Grafil-base. We also do a similar test to calculate the filtering gain achieved by the pipeline mode over the parallel mode. The pipeline mode is 20%–60% better.

Experiments on the Synthetic Datasets.

The synthetic data generator first creates a set of seed structures randomly. Seed structures are then randomly combined to form a synthesized graph. Readers are referred to [12] for details about the synthetic data generator. A typical dataset may have 10,000 graphs and use 200 seed fragments with 10 kinds of nodes and edges. We denote this dataset by $D10kI10T50L200E10V10$. $E10$ ($V10$) means there are 10 kinds of edge labels (node labels). In this dataset, each graph has 50 edges ($T50$) and each seed fragment has 10 edges ($I10$) on average.

Since the parameters of synthetic datasets are adjustable, we can examine the conditions where Grafil outperforms Edge. One can imagine that when the types of labels in a graph become very diverse, Edge will perform nearly as well as Grafil. The reason is obvious. Since the graph will have less duplicate edges, we may treat it as a set of tuples $\{node1\_label,\ node2\_label,\ edge\_label\}$ instead of a complex structure. This result is confirmed by the following experiment. We generate a synthetic dataset, $D10kI10T50L200E10V10$, which has 10 edge labels and 10 node labels. This setting will generate $(10 \times 11)/2 \times 10 = 550$ different edge tuples. Most of graphs in this synthetic dataset have 30 to 100 edges. If we represent a graph as a set of edge tuples, few edge tuples will be the same for each graph in this dataset. In this situation, Edge is good enough for similarity search. Figure 14 shows the results for queries with 24 edges. The two curves are very close to each other, as expected.

We then reduce the number of label types in the above synthetic dataset and only allow 2 edge labels and 4 vertex labels. This setting significantly increases the self similarity in a graph. Figure 15 shows that Grafil outperforms Edge a lot in this dataset. We can further reduce the number of label types. For example, if we ignore the label information and only consider the topological skeleton of graphs, the edge-based filtering algorithm will not be effective at all. In that situation, Grafil has more advantages than Edge.

7. RELATED WORK

Structure similarity search has been studied in various fields. Willett et al. [24] summarized the techniques of fingerprint-based and graph-based similarity search in chemical compound databases. Raymond et al. [18] proposed a three tier algorithm for full structure similarity search. Recently, substructure search has attracted lots of attention in the database research community. Shasha et al. [19] developed a path-based approach for substructure search, while Srinivasa et al. [20] built multiple abstract graphs for the indexing purpose. Yan et al. [25] took the discriminative frequent structures as indexing features to improve the search performance.

As to substructure similarity search, in addition to graph edit distance and alignment distance, maximum common subgraph is used to measure the similarity between two structures. Unfortunately, it is NP-complete [6]. Nilsson[15] presented an algorithm for the pairwise approximate substructure matching. The matching is greedily performed to minimize a distance function for two structures. Hagadone [8] recognized the importance of substructure similarity search in a large set of graphs. He used the atom and edge label to do screening. Holder et al. [9] adopted the principle of minimum description length for approximate graph matching. Messmer and Bunke [13] studied the reverse substructure similarity search problem in computer vision and pattern recognition. These methods did not explore the potential of using more complicated structures to improve the filtering performance, which is studied extensively by our work. In [19], Shasha et al. also extended their substructure search algorithm to support queries with wildcards, *i.e.*, don't care nodes and edges. Different from their similarity model, we do not fix the positions of wildcards, thus allowing a general and flexible search scheme.

Besides the full-scale graph search problem, researchers also studied the approximate tree search problem. Wang et al. [23] designed an interactive system that allows a user to search inexact matchings of trees. Kailing et al. [10] presented new filtering methods based on tree height, node degree and label information.

The structural filtering approach presented in this study is also related to string filtering algorithms. A comprehensive survey on various approximate string filtering methods was presented by Navarro [14]. The well-known q-gram method was initially developed by Ullmann [22]. Ukkonen [21] independently discovered the q-gram approach, which was further extended in [7] against large scale sequence databases. These q-gram algorithms work for consecutive sequences, not structures. Our work generalized the q-gram method to fit structural patterns of various sizes.

8. CONCLUSIONS

In this study, we have investigated the problem of substructure similarity search in large scale graph databases, a problem raised by the emergence of massive, complex structural data. Different from the previous work, our solution explored the filtering algorithm using indexed structural patterns, without doing costly structure comparisons. The successful transformation of the structure-based similarity measure to the feature-based measure renders our method attractive in terms of accuracy and efficiency. Our filtering algorithm is fully built on the feature-graph matrix, thus performing very fast without accessing the physical database. We show that the multi-filter composition strategy adopted by Grafil is far superior to the single filter approach using all features together due to the frequency conjugation problem identified in this paper. We also demonstrated the direct usage of clustering techniques in feature set selection, which can leverage the filtering performance further. Moreover, the new concept developed in Grafil can be directly applied to searching inexact non-consecutive sequences, trees, and other complicated structures as well.

9. REFERENCES

[1] S. Beretti, A. Bimbo, and E. Vicario. Efficient matching and indexing of graph models in content based retrieval. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 23:1089–1105, 2001.

[2] H. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. Bhat, H. Weissig, I. Shindyalov, and P. Bourne. The protein data bank. *Nucleic Acids Research*, 28:235–242, 2000.

[3] H. Bunke and K. Shearer. A graph distance metric based on the maximal common subgraph. *Pattern Recognition Letters*, 19:255 – 259, 1998.

[4] D. Hochbaum (ed.). *Approximation Algorithms for NP-Hard Problems*. PWS Publishing, MA, 1997.

[5] U. Feige. A threshold of ln n for approximating set cover. *Journal of the ACM*, 45:634 – 652, 1998.

[6] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman & Co., New York, 1979.

[7] L. Gravano, P. Ipeirotis, H. Jagadish, N. Koudas, S. Muthukrishnan, L. Pietarinen, and D. Srivastava. Using q-grams in a dbms for approximate string processing. *Data Engineering Bulletin*, 24:28–37, 2001.

[8] T. Hagadone. Molecular substructure similarity searching: efficient retrieval in two-dimensional structure databases. *J. Chem. Inf. Comput. Sci.*, 32:515–521, 1992.

[9] L. Holder, D. Cook, and S. Djoko. Substructure discovery in the subdue system. In *Proc. AAAI'94 Workshop on Knowledge Discovery in Databases (KDD'94)*, pages 169 – 180, 1994.

[10] K. Kailing, H. Kriegel, S. Schnauer, and T. Seidl. Efficient similarity search for hierarchical data in large databases. In *Proc. 9th Int. Conf. on Extending Database Technology (EDBT'04)*, pages 676–693, 2004.

[11] M. Kanehisa and S. Goto. Kegg: Kyoto encyclopedia of genes and genomes. *Nucleic Acids Research*, 28:27–30, 2000.

[12] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proc. 2001 Int. Conf. on Data Mining (ICDM'01)*, pages 313–320, 2001.

[13] B. Messmer and H. Bunke. A new algorithm for error-tolerant subgraph isomorphism detection. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 20:493 – 504, 1998.

[14] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33:31 – 88, 2001.

[15] N. Nilsson. *Principles of Artificial Intelligence*. Morgan Kaufmann, Palo Alto, CA, 1980.

[16] National Library of Medicine. http://chem.sis.nlm.nih.gov/chemidplus.

[17] E. Petrakis and C. Faloutsos. Similarity searching in medical image databases. *Knowledge and Data Engineering*, 9(3):435–447, 1997.

[18] J. Raymond, E. Gardiner, and P. Willett. Rascal: Calculation of graph similarity using maximum common edge subgraphs. *The Computer Journal*, 45:631–644, 2002.

[19] D. Shasha, J. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *Proc. 21th ACM Symp. on Principles of Database Systems (PODS'02)*, pages 39–52, 2002.

[20] S. Srinivasa and S. Kumar. A platform based on the multi-dimensional data model for analysis of bio-molecular structures. In *Proc. 2003 Int. Conf. on Very Large Data Bases*, pages 975–986, 2003.

[21] E. Ukkonen. Approximate string matching with q-grams and maximal matches. *Theoretic Computer Science*, pages 191–211, 1992.

[22] J. Ullmann. Binary n-gram technique for automatic correction of substitution, deletion, insertion, and reversal errors in words. *The Computer Journal*, 20:141–147, 1977.

[23] J. Wang, K. Zhang, K. Jeong, and D. Shasha. A system for approximate tree matching. *IEEE Trans. on Knowledge and Data Engineering*, 6:559 – 571, 1994.

[24] P. Willett, J. Barnard, and G. Downs. Chemical similarity searching. *J. Chem. Inf. Comput. Sci.*, 38:983–996, 1998.

[25] X. Yan, P. Yu, and J. Han. Graph indexing: A frequent structure-based approach. In *Proc. 2004 ACM Int. Conf. on Management of Data (SIGMOD'04)*, pages 335 – 346, 2004.