Efficiently Indexing Large Sparse Graphs for Similarity Search

Guoren Wang, Bin Wang, Xiaochun Yang, *Member*, *IEEE Computer Society*, and Ge Yu, *Member*, *IEEE*

Abstract—The graph structure is a very important means to model schemaless data with complicated structures, such as protein-protein interaction networks, chemical compounds, knowledge query inferring systems, and road networks. This paper focuses on the index structure for similarity search on a set of large sparse graphs and proposes an efficient indexing mechanism by introducing the Q-Gram idea. By decomposing graphs to small grams (organized by κ-Adjacent Tree patterns) and pairing-up on those κ-Adjacent Tree patterns, the lower bound estimation of their edit distance can be calculated for candidate filtering. Furthermore, we have developed a series of techniques for inverted index construction and online query processing. By building the candidate set for the query graph before the exact edit distance calculation, the number of graphs need to proceed into exact matching can be greatly reduced. Extensive experiments on real and synthetic data sets have been conducted to show the effectiveness and efficiency of the proposed indexing mechanism.

Index Terms—Graph indexing, similarity search, κ-adjacent tree.

1 Introduction

Graph similarity matching over a set of large graphs is a key issue in many graph-based applications. For example, the entities of a relational database can be considered as the vertices of a graph while the relation between those entities can be regarded as the edges across the corresponding vertices. Bioinformatical data such as protein-protein interaction networks and chemical compounds, Web and XML documents, and road networks also have similar structures to graphs.

Conceptually, all data with a discrete structure can be represented in the form of graph, and the problem of designing and maintaining a graph index for similarity matching is quite prevalent in domains of many subjects.

There are various research problems related to graphs. For example, for a query graph, how to find out all the graphs in a graph set which are similar to the query graph. Whether two graphs are similar to each other can be judged by the size of their maximum common subgraph, but only theoretically, since the subgraph isomorphism test has been proved to be an NP-Complete problem [1]. The sequential searching from a set of large graphs introduces a huge computational cost. Due to this low efficiency of a sequential search, a *filter-and-verification* method is usually employed to speed up the search efficiency of graph similarity matching over a graph set and an index on the graph set can be used to filter the graph set to reduce candidates.

• The authors are with the College of Information Science and Engineering, Northeastern University, Wenhua Road 11-3, Heping District, Shenyang, Liaoning, P.R. China. E-mail: {wangbin, yangxiaochun}@ise.neu.edu.cn, {wanggr, yuge}@mail.neu.edu.cn.

Manuscript received 16 Apr. 2009; revised 6 Sept. 2009; accepted 7 Dec. 2009; published online 4 Feb. 2010.
Recommended for acceptance by X. Zhou.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2009-04-0356.
Digital Object Identifier no. 10.1109/TKDE.2010.28.

Similar to the Q-Gram [2] index on strings, our indexing method focuses on how to break graphs into pieces, and evaluates the similarity of them through pairing-up corresponding pieces between the database and the query. However, indexing graphs is more challenging than indexing strings: the decomposition of strings is relatively simple, we only need to let a Q-length sliding window go from the beginning to the end, but decomposing graphs becomes more cumbersome. The first problem is how to define the "length" of the graph pieces. The second problem lies in the fact that we have to make sure the uniqueness of the decomposition: two isomorphic graphs should have the same decomposition. Furthermore, a minor number of modifications to a graph should only lead to a relatively small number of modified pieces, that is, the number of pairing-ups on the pieces should be correlated with the similarity of the graphs. Finally, the decomposition and pairing-up procedure should be efficient. For this purpose, an index should be designed to accelerate similarity.

Sparse graph can be used to model data in many applications such as road networks [3], natural language processing [4], and protein interaction networks [5]. Beasley and Christofides [3] use a sparse feasibility graph to model road networks to solve the problems of vehicle routing. Nallapati et al. [4] propose sparse word graph to describe the correlations between words. Lin et al. [5] propose a prediction algorithm of protein functions based on protein interaction data, which can be represented large sparse graphs. In these sparse graph applications, graph similarity search is an important issue. For example, a biology scientist might want to know the relationship between two protein interaction networks. Another example can be found in the field of chemistry, where one might want to find all compounds from a database with a certain substructure. Motivated by these applications, we propose κ-Adjacent Tree (κ-*AT*) pattern for graph decomposition and indexing to solve the similarity search of sparse graphs. Given two graphs, we decompose them into κ-*AT* patterns (like Q-Gram decomposition of strings), then use the number of their common κ-*AT* patterns for edit distance estimation. Experimental results show that this indexing method for filtering sparse graphs is relatively fast and has a good accuracy when the querying edit distance threshold is not large.

The rest of this paper is organized as follows: Section 2 introduces the research works related to this paper. Section 3 presents the κ-adjacent tree index, including basic definitions, the edit operations adopted in this paper, problem statement, and the filtering principle based on the concept of κ-adjacent tree. Section 4 describes the implementation techniques of κ-adjacent tree index, including the construction and maintenance algorithms of κ-adjacent tree index. Section 5 gives the performance evaluation on the proposed index. Finally, Section 6 concludes the paper and highlights the future works.

2 Related Work

Because subgraph isomorphism is an NP-complete problem [1], a *filter-and-verification* method is usually employed to speed up the search efficiency of graph similarity matching over a graph set. Since the filtering phase is the key issue to improve the search efficiency, a lot of indexing techniques have been proposed recently for speeding up filtering [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18]. Most of the existing research works on graph indexing can be grouped into two categories: frequent-subgraph-based indexing and graph-decomposition-based indexing.

Motivated by *path-based* indexing methods [19] for managing semistructured data, Yan et al. [6], [7] introduce a novel indexing structure referred to as *G-Index* based on frequent subgraph patterns. It makes use of frequent subgraph structures as the basic indexing features. Shang et al. [8] propose a novel indexing technique *QuickSI* to efficiently compute verification phase for testing subgraph isomorphism and a new feature-based indexing method is developed to accommodate *QuickSI* in the filtering phase. Yan et al. [9] propose a structure filtering algorithm *Grafil* to filter out graphs without performing pairwise similarity computations. They point out that the filtering effectiveness and efficiency greatly depend on the quality of selected features. Cheng et al. [10], [11] propose a *nested inverted-index* called *FG-index* to avoid candidate verification by exploiting frequent subgraphs and edges as indexing features. However, when encountering infrequent queries, the method performs poorly, as infrequent subgraphs are not incorporated into the FG-Index. On the other hand, Chen et al. [12] investigate an interesting graph containment search problem, and a contrast subgraph-based indexing model *cIndex* is proposed to solve this problem.

Frequent-subgraph-based indexing methods are also called feature-based indexing methods. They have two main shortcomings. One is that the effectiveness and efficiency of such kind of methods depend on the quality of selected features. The other is that it is difficult to construct and maintain the index because the frequent subgraph mining algorithm usually takes a very long time to compute [20].

To compensate for the latter shortcoming, the authors of [13] and [14] propose frequent-subtree-based indexing methods called *TreePi* and *Closure-Tree*. The *TreePi* approach indexes a graph set by using only subtree patterns to accelerate the filtering procedure, while the *Closure-Tree* approach uses the data structure called *ClosureTree* to organize a B-tree-like index on the graph set. These new works also lead to discussions on the optimum pattern selection on graph indexing. Zhao et al. [15] discuss the filtering ability among Path, Tree, and Graph, based on the study on many experiment results, and a conclusion of $\mathrm{Tree} + \Delta \geq \mathrm{Graph}$ is drawn by the authors. Although subtree-based indexing methods alleviate the latter shortcoming, they still suffer from the former shortcoming like frequent-subgraph-based methods.

In graph-decomposition-based indexing methods, graph decomposition is applied directly to answering questions of isomorphism and subgraph isomorphism. Graph decomposition has been adopted before to the problems of isomorphism and subgraph isomorphism for planar graphs [16], [17]. The method proposed in [16] partitions a planar graph into pieces of small trees, and dynamic programming is applied within each piece. With the method, the subgraph isomorphism problem can be solved in linear time. Kukluk et al. [17] have proposed an algorithm for isomorphism testing of planar graphs based on the decomposition of a graph into biconnected components. Williams et al. [18] have developed three kinds of graph decomposition schemas, *Clique Decomposition*, *Modular Decomposition*, and *Node Label Decomposition (NLD)* [21] to decompose a graph set, and Directed Acyclic Graphs (DAGs) are constructed to describe the results of a graph decomposition and a *Graph Decomposition Index (GDI)* is proposed to support graph similarity search. Tian and Patel [22] have proposed an indexing method by incorporating graph structural information in a hybrid index structure called *NH-index*.

Graph decomposition indexing methods suffer from two main drawbacks. The first is that they have to enumerate all connected subgraphs, and therefore, complexity of graph decomposition is exponential to the graph size that is being decomposed. The other is that the frequency information existing in the graph decomposition results is not utilized for improving the efficiency of graph similarity search.

For either frequent subgraphs or decomposed subgraphs, they can be represented as sequences based on graph encoding techniques and graph similarity searching can be converted into subsequence matching. Jiang et al. [23] have proposed a sequencing method called *gString* based on the chemical semantics in chemical compound databases. Zou et al. [24] have proposed a spectral graph encoding method called *GCoding*, and a two-step filtering process is proposed based on the encoding schema to reduce the checking cost in the filtering phase.

In this paper, we propose a novel graph decomposition method called κ-*Adjacent Tree* (κ-*AT*). The method is inspired by the idea of "Q-Gram" from string matching area. A graph is decomposed into a set of κ-Adjacent Trees and the decomposed results are indexed by a κ-*AT* index. A lower bound of the edit distance between graphs is derived and used for filtering graphs. The lower bound lemma guarantees the absence of false negatives. The proposed method incorporates both graph decomposition methods and frequent subgraph methods. Therefore, it can both control the results of decomposed subgraphs by tuning the κ value and make full use of the frequency information about decomposed subgraphs to improve the efficiency of the filtering phase.

In summary, we make the following contributions:

- A novel graph decomposition method called κ-*Adjacent Tree* (κ-*AT*) is proposed to index graphs for graph similarity search. The method is inspired by the idea of "Q-Gram" from string matching area.
- We get an inequality between the edit distance and the number of the common subgraphs of the two given graphs. With the inequality, we can filter a lot of graphs and improve the performance during graph similarity searching.
- Extensive experiments have been conducted to validate the effectiveness and efficiency of the proposed method.

3 Principle of κ-Adjacent Tree Index

3.1 Preliminaries

First, we introduce the basic definitions in graph theory for graph representation and comparison. Whether two graphs are equal to each other is judged by a (sub)graph isomorphism test; it will be necessary to introduce the concept of (sub)graph isomorphism in the following.

A graph $G(V, E)$ is a tuple containing two sets, the set of vertices V and the set of edges E, and each edge e is associated with two vertices in the set of vertices, denoted by $e(u, v), u, v \in V$.

Definition 1 (Attributed Graph). *A labeled graph $G(V, E, W_v, W_e, \Sigma_V, \Sigma_E)$ is a graph, where each vertex and edge are labeled with words from the alphabets Σ_V and Σ_E, respectively. $W_v : V \to \Sigma_V$ is a surjection from $V(G)$ to Σ_V and $W_e : E \to \Sigma_E$ is a surjection from $E(G)$ to Σ_E. Σ_V and Σ_E are the label alphabets for vertices and edges, respectively.*

Definition 2 (Subgraph Isomorphism). *Let G and G' be two graphs. A subgraph isomorphism [6] from G to G' is an injection $f : V(G) \to V(G')$, iff it satisfies 1) $\forall u \in V(G)$, $f(u) \in V(G'), W_v(u) = W_v(f(u))$ and 2) $\forall \varepsilon(u, v) \in E(G)$, $e(f(u), f(v)) \in E(G')$, and $W_e(e(u, v)) = W_e(e(f(u), f(v)))$.*

Definition 3 (Graph Isomorphism). *Let G and G' be two graphs. An isomorphism [6] from G to G' is a bijection $f : V(G) \to V(G')$, iff it satisfies 1) $\forall u \in V(G), f(u) \in V(G'), W_v(u) = W_v(f(u))$ and 2) $\forall e(u, v) \in E(G), e(f(u), f(v)) \in E(G')$, and $W_e(e(u, v)) = W_e(e(f(u), f(v)))$.*

It is denoted by $G \subset G'$. Graph G isomorphic to G' is denoted by $G \cong G'$ if graph G is subgraph isomorphic to G' and it is denoted by $G \cong G'$ if graph G is isomorphic to G'. If two graphs are isomorphic to each other, they can be considered to be equal.

3.2 Problem Statement

We use graph edit distance to measure the similarity of two graphs. Intuitively, the edit distance of the two graphs is the minimum number of edit operations to convert one into another.

Definition 4 (Graph Edit Operation). *A Graph Edit Operation (GEO) is an edit operation to transform one graph to another. The GEO can be one of the following six operations, as defined in [14], [20]:*

1. *Delete an edge from the graph.*
2. *Insert an edge between two disconnected vertices.*
3. *Delete an isolated vertex from the graph.*
4. *Insert an isolated vertex into the graph.*
5. *Change the label of a vertex.*
6. *Change the label of an edge.*

It is obvious that we can convert any graph into another by a finite number of GEOs. To judge the similarity of two graphs, we define the Edit Distance between two graphs, which is the minimum number of GEOs needed to transform a graph to another.

Definition 5 (Graph Edit Distance). *The Graph Edit Distance $\Delta_G(G_1, G_2)$ between two graphs G_1 and G_2 is the minimum number of GEOs needed to transform G_1 to a graph isomorphic to G_2.*

The definition of edit distance of two graphs gives us a measurement to quantify the difference of two graphs. Based on the definition above, we formally define the problem as follows:

Problem statement (Δ_G Query). Given a graph set D, a query graph Q, and a given threshold ϵ, find out all the graphs G in D satisfying the inequality $\Delta_G(Q, G) \leq \epsilon$. The returned graphs are the answer set of Q.

3.3 Filtering Principle

So far, we have introduced the measurement for quantifying the similarity between graphs and the problem definition of similarity search on graph sets. Next, we will discuss how to use κ-Adjacent Tree pattern decomposition for Δ_G lower bound estimation. To give readers a clear overview of our index method, we will use Q-Gram index as a comparison.

The main idea of Q-Gram is as follows: For each letter in a string, keep its small continuous gram (a short substring with this letter as its center). As such, an N-length string can have N grams in its Q-Gram set, each gram is of length Q. Q-Gram decomposition has overlapping, but it can make sure that 1) two same strings have identical Q-Gram sets and 2) any edit operation will only affect a limited number of grams. These two properties make it feasible to perform string edit distance query filtering, since the number of matched Q-Grams is correlated with the edit distance.

Instinctively, one can apply the reasoning behind the Q-Gram method to the problem of graph similarity matching. For each vertex of a graph, we can identify a small subgraph within the graph with this vertex as its center, and let all those subgraphs to be the "Grams" of the graph. This kind of "Gram" obviously satisfies the two properties stated above, so the pairing-up number of "Grams" between two graphs can be used to evaluate their graph edit distance.

This is the basic idea of our index, but is technically not good enough, since the "Grams" are still graph patterns, and the pairing-up on those patterns can still be slow (depending on slow isomorphism test). To avoid the structural complexity of graph patterns, we use the *Adjacent Tree* patterns for index construction. An adjacent tree is a redundant subtree of a graph, which is able to preserve its structural information well. Furthermore, it is much faster to do matching on adjacent trees than on graph grams.

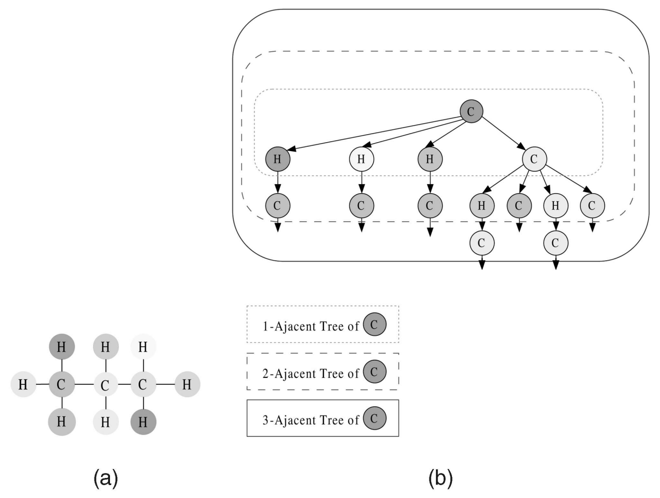


Fig. 1. Example of κ-adjacent tree.

Definition 6 (Adjacent Tree (AT)). *The adjacent tree of a vertex v ($AT(v)$) in $G(V,E)$ is a breadth-first search tree of vertex v, the children of each node of $AT(v)$ are sorted by their labels in the graph.*

Definition 6 indicates that $AT(v)$ is infinitely expanding, as shown in Fig. 1. To be of practical use, we only take the subtree of the top-κ level of the adjacent tree for index construction.

Definition 7 (κ-Adjacent Tree (κ-AT)). *The κ-adjacent tree of a vertex v (κ-$AT(v)$) in graph G is the top κ-level subtree of $AT(v)$.*

Fig. 1 shows a κ-AT example, some of the adjacent trees of the graph shown in Fig. 1a are shown in Fig. 1b. With κ increases, the more and more adjacent vertices are included into the κ-AT, which indicates that the "Gram" becomes larger with larger κ. In Q-Gram index, either a too large or small Gram-length will weaken the strength of the index. This property also holds in κ-AT index, there is a trade-off in κ selection, as we will see it in the experimental part of this paper.

The concept of κ-AT is inspired on [14]. He and Singh [14] use pseudoisomorphism to do fast graph matching. Here, we extend level-n adjacent subtrees by sorting the sibling nodes into an ordered sequence. Presorting the siblings can be convenient for unique sequentializing a κ-AT. In Section 4, we will see how this presorted structure benefits the κ-AT sequentializing.

It is simple to decompose a graph into κ-ATs. For each vertex, its κ-AT can be generated by a width-first searching on it. Fig. 2 gives an example of the κ-ATSs of two graphs, the tables below graphs G_1 and G_2 record the different 1-ATs and corresponding numbers.

For a graph G, let κ-$ATS(G)$ (κ-Adjacent Tree Set) of it be $\{\kappa\text{-}AT(v)|\forall v \in G\}$, κ-ATS serves as "Q-Gram set" for graph edit distance estimation. Intuitively, if the number of edit operations exerted on a graph is small, there must be considerable number of its κ-ATs remain unchanged. Next, we will see how to use the number of common κ-ATs of two graphs to estimate the lower bound of Δ_G of them.

The matching number of two graphs' κ-AT can be described as the intersection size of their κ-ATS. Lemma 1 gives the basic filtering principle of κ-AT index:

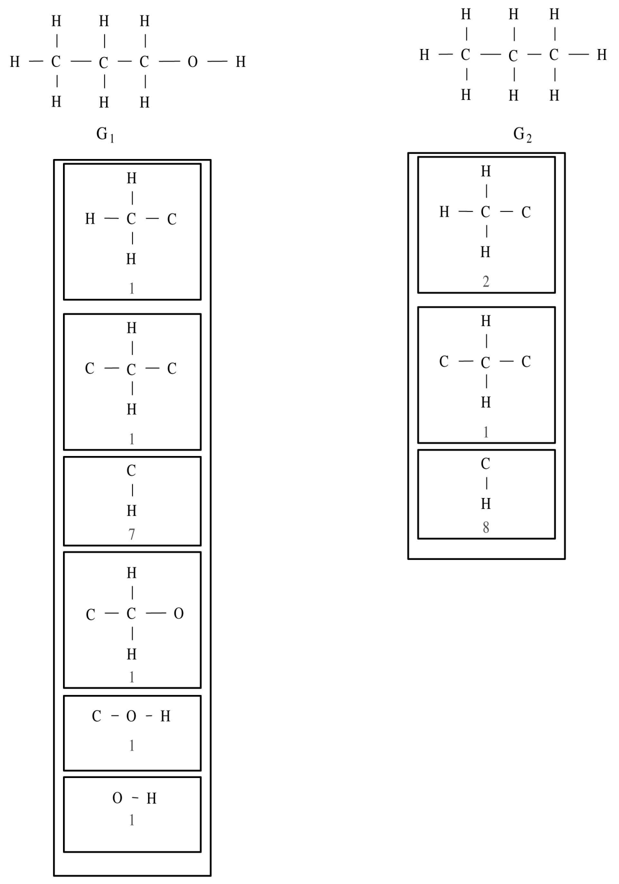


Fig. 2. Example of κ-adjacent tree set.

Lemma 1. *For two given graphs g_1 and g_2, let $\delta(g_1)$ and $\delta(g_2)$ be the maximum degree of g_1 and g_2, respectively. If the maximum degree of the two graphs g_1 and g_2 satisfies $\delta(g_1) > 1$ and $\delta(g_2) > 1$, then $|\kappa\text{-}ATS(g_1) \cap \kappa\text{-}ATS(g_2)|$ and $\Delta_G(g_1, g_2)$ satisfy the following inequality*[1]*:*

$$|\kappa\text{-}ATS(g_1) \cap \kappa\text{-}ATS(g_2)| \geq |V(g_1)| - \Delta_G(g_1,g_2) \cdot 2(\delta(g_1)-1)^{\kappa-1}. \quad (1)$$

Proof. Here, we use the technique of mathematical induction. First, we verify when $\Delta_G(g_1,g_2)=0$, $|\kappa\text{-}ATS(g_1) \cap \kappa\text{-}ATS(g_2)| \geq |V(g_1)|$. If the two graphs g_1 and g_2 are isomorphic to each other, they must have the same properties in every respect, so the number of common κ-ATs in their κ-ATS will be the number of the vertices they have; thus,

$$|\kappa\text{-}ATS(g_1) \cap \kappa\text{-}ATS(g_2)| \geq |V(g_1)|.$$

Suppose when $\Delta_G(g_1,g_2)=n$, the lemma holds: $|\kappa\text{-}ATS(g_1) \cap \kappa\text{-}ATS(g_2)| \geq |V(g_1)| - n \cdot 2(\delta(g_1)-1)^{\kappa-1}$.

Induction step: We need to prove that when $\Delta_G(g_1, g_2) = n+1$, the lemma still holds. Because there is one more GEO operation by changing the edit distance from n to $n+1$, we discuss the correctness of the induction in situations of the six different edit operations defined in Definition 4.

1. We can see from the proof for the lemma that $2(\delta(g_1)-1)^{\kappa-1}$ is the upper bound number of the influenced κ-ATs of g_1 by an edit operation on g_1.

- **Delete an edge from the graph:** Consider that the edit operation is that of deleting edge e from graph g_1. Since there are k paths including e in g_1, deleting an edge from g_1 will influence at most $2(\delta(g_1)-1)^{\kappa-1}\kappa{-}AT$s of g_1.[2] So, we have

$$\begin{aligned}|\kappa\text{-}ATS(g_1)\cap\kappa\text{-}ATS(g_2)| &\geq |V(g_1)| \\ &\quad - n\cdot 2(\delta(g_1)-1)^{\kappa-1} - 2(\delta(g_1)-1)^{\kappa-1} \\ &= |V(g_1)| - (n+1)\cdot 2(\delta(g_1)-1)^{\kappa-1}.\end{aligned}$$

  That is, Lemma 1 holds when $\Delta_G(g_1,g_2)=n+1$.
- **Insert an edge between two vertices:** Insertion of an edge is the reverse operation of deletion of an edge, so the proof is the same.
- **Delete an isolated vertex from the graph:** Deleting an isolated vertex from the graph will influence only 1 $\kappa\text{-}ATS$ of g_1, so we have

$$\begin{aligned}&|\kappa\text{-}ATS(g_1)\cap\kappa\text{-}ATS(g_2)| \\ &\quad\geq |V(g_1)| - n\cdot 2(\delta(g_1)-1)^{\kappa-1} - 1 \\ &\quad\geq |V(g_1)| - (n+1)\cdot 2(\delta(g_1)-1)^{\kappa-1}.\end{aligned}$$

- **Insert an isolated vertex into the graph:** Insertion of an isolated vertex is the reverse operation of deletion of an isolated vertex, so the proof is the same.
- **Change the label of a vertex:** Changing the label of vertex v in g_1 at most influences $(\delta(g_1)-1)^{\kappa-1}\kappa$-$AT$s of g_1, so we have

$$\begin{aligned}|\kappa\text{-}ATS(g_1)\cap\kappa\text{-}ATS(g_2)| &\geq |V(g_1)| \\ &\quad - n\cdot 2(\delta(g_1)-1)^{\kappa-1} - (\delta(g_1)-1)^{\kappa-1} \\ &= |V(g_1)| - (n+1)\cdot 2(\delta(g_1)-1)^{\kappa-1}.\end{aligned}$$

- **Change the label of an edge:** Changing the label of edge e in g_1 at most influences $2(\delta(g_1)-1)^{\kappa-1}\kappa$-$AT$s of g_1, so we have

$$\begin{aligned}&|\kappa\text{-}ATS(g_1)\cap\kappa\text{-}ATS(g_2)| \\ &\quad\geq |V(g_1)| - n\cdot 2(\delta(g_1)-1)^{\kappa-1} - (\delta(g_1)-1)^{\kappa-1} \\ &\quad= |V(g_1)| - (n+1)\cdot 2(\delta(g_1)-1)^{\kappa-1}.\end{aligned}$$

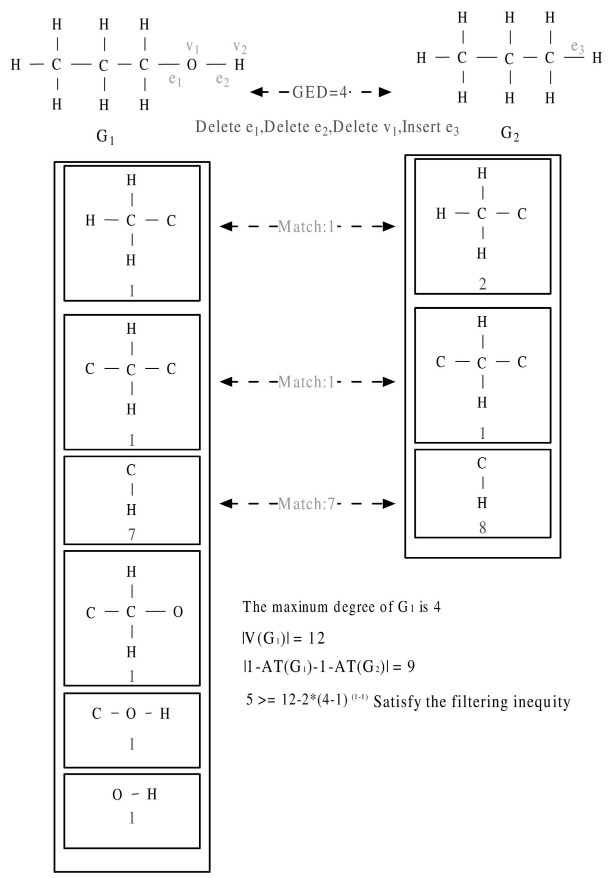
In a conclusion to the six situations of induction, we have discussed above. When $\Delta_G(g_1,g_2)=n+1$, there will be

$$\begin{aligned}&|\kappa\text{-}ATS(g_1)\cap\kappa\text{-}ATS(g_2)| \\ &\quad\geq |V(g_1)| - (n+1)\cdot 2(\delta(g_1)-1)^{\kappa-1}.\end{aligned}$$

□

Lemma 1 can be used to filter the candidate for Δ_G query. First, we pre-generate the set of k-ATs from all graphs in the graph set. For a query (Q,ϵ), we keep as the candidates of the query the graphs G for which the following condition holds: $|\kappa\text{-}ATS(Q)\cap\kappa\text{-}ATS(G)|\geq|V(Q)|-\Delta_G(Q,G)\cdot 2(\delta(Q)-1)^{\kappa-1}$. Suppose that the query Q has the number of vertices 100 and the maximum degree 3, Δ_G threshold is 2, G is a graph in the set, and Q and G have 30 common 3-ATs. In this case, the upper bound becomes $30 < 100 - 2\times 2\times(3-1)^{(3-1)}$, and G cannot be the answer of query Q based on Lemma 1. Using the filtering ability of Lemma 1, we can avoid many exact edit distance calculations.

2. In the case of $\kappa=1$, deleting an edge $e(v_1,v_2)$ from g_1 influences $2=2(\delta_1(g_1)-1)^{1-1}1-AT$s of g_1 in which the roots are v_1 and v_2, respectively. In the case of $\kappa=\kappa+1$, the number of the influenced $(\kappa{+}1)$-ATs is that of the influenced κ-ATs multiplied by $(\delta-1)$. This is because $(\kappa+1)$-ATs can be acquired by extending each node in κ-ATs with all its neighbors and at least one of these is already included in the κ-ATs.



GED=4
Delete e1,Delete e2,Delete v1,Insert e3
G1
G2
Match:1
Match:1
Match:7
The maxinum degree of G1 is 4
|V(G1)| = 12
|1-AT(G1)-1-AT(G2)| = 9
5 >= 12-2*(4-1)^(1-1) Satisfy the filtering inequity

Fig. 3. Example of Lemma 1.

Fig. 3 gives an example of how Lemma 1 works in candidate filtering. We mark the properties of the graphs such as $|V(G_1)|$, $\delta(G_1)$, $\Delta_G(G_1,G_2)$ in the figure, and verify Lemma 1 through a simple calculation of inequality (1).

3.4 Discussions

In this section, we give a brief analysis of how κ and graph density can influence the filtering ability of the index.

Let the κ-AT Candidate Set of a Δ_G Query (Q,ϵ) in graph set D is

$$\begin{aligned}Cand_\kappa = \{G|G\in D, &|\kappa\text{-}ATS(Q)\cap\kappa\text{-}ATS(G)| \\ &\geq |V(G)| - \epsilon\cdot 2(\delta(G)-1)^{\kappa-1}\},\end{aligned}$$

the Candidate Set is the result set returned by κ-AT filtering on the graph set.

It is necessary to introduce the definition of sparse graph. A sparse graph is a graph G with $|E(G)|\ll|V(G)|^2$, based on handshaking lemma, $E(G)=(V(G)\delta(G))/2$, so we have $\delta(G)\ll|V(G)|$. Nearly all biological and chemical compounds are sparse graphs, for the number of chemical bonds of each atom is constant, while the vertices number can be very large, so there is always $\delta(G) \ll |V(G)|$.

The density of a graph can also be measured by the average degree of it. It is obvious that a sparse graph has a relatively low density.

There are two problems related to our lower bound estimation in Lemma 1. The first problem is that the estimation is too loose if the density of the graph is large. Can it be further tightened? Take a worst-case example, suppose the two graphs are both complete graphs, then even a single deletion on an edge will change all the vertices' κ-AT when $\kappa > 1$. It is difficult to further tighten the lower bound estimation, but we can see that the filtering ability will be better on sparse graphs, the smaller the $\delta(G)$ is, the smaller is the number of the k-ATs influenced by a single GEO.

The second problem is how to choose the proper κ to guarantee the filtering ability of the index for a specific graph set. Suppose that the graphs in the set are all sparse graphs, i.e., $\delta(G) \ll |V(G)|, \forall G \in D$. From

$$|Cand_\kappa| = |\{G|G \in D, |\kappa\text{-}ATS(Q) \cap \kappa\text{-}ATS(G)| \geq |V(G)| - \epsilon \cdot \delta(G)^\kappa\}|,$$

we can see that $|V(G)| - \epsilon \cdot \delta(G)^\kappa$ should be greater than zero to guarantee the filtering ability of the index. Rearranging this expression leads to the following condition: $\kappa \leq log_{\delta(G)}(|V(G)|/\epsilon)$. But is a smaller κ necessarily better? The answer is negative because a matching on small adjacent tree sets may lead to mismatches on the whole graph. For example, two graphs can have all their 1-ATs matched with the other, but all in the wrong places. This can be analogized with the Q-Gram method on string matching, both too large and too small length Q-Grams can weaken the filtering ability of Q-Gram index. This is because too small grams cannot reflect the global structure of the string while too large grams may not be preserved under minor edit operations, this causes the trade-off in gram size selection. Due to the structural complexity of graph, it is difficult to give a theoretical evaluation on how to choose the optimum κ, and we will show some empirical results in the experiment section.

From the inequality $1 \leq \kappa \leq log_{\delta(G)}(|V(G)|/\epsilon)$, we can also see why the index is more effective for sparse graphs because only $log_{\delta(G)}(|V(G)|/\epsilon) \geq 1$ guarantees the existence of a proper κ. If we do not want κ to be too small, the condition $\delta(G) \ll V(G)$ must hold, and thus, $|E(G)| \ll |V(G)|^2$.

However, from Inequality (1), we can see when $\kappa = 1$, the parameter $\delta(g_1)$ is eliminated. This leads to a solution for nonsparse graph. To index nonsparse graphs, we can choose 1-AT index, without worrying about that the large δ will weaken the filtering ability of the index. Although this is a feasible solution for nonsparse graphs, we still suggest to use this index only on sparse graphs because when κ is too small, the index sometimes can show unsatisfactory performance on candidate filtering.

4 κ-AT Index Implementation

4.1 κ-AT Index Construction

The discussion in Section 3 gives us a method to organize the index structure for Δ_G query. First, we generate all the κ-ATs of each graph in the graph set and store them in a table. For a query Q, we also generate its κ-ATs, and for each graph G in the graph set, we calculate the number of common κ-ATs of Q and G. Next, we use Inequality (1) to test if G belongs to the candidate set of the query.

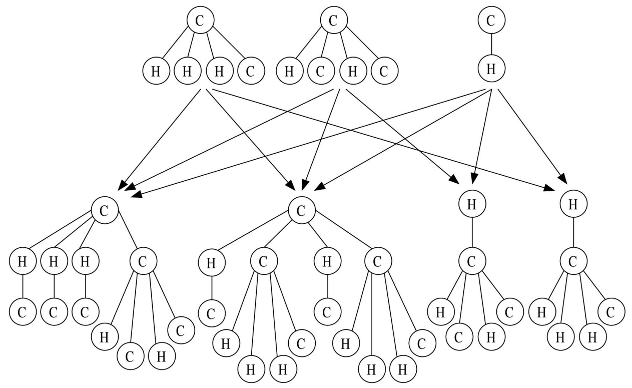


Fig. 4. The hierarchical structure of κ-AT.

This approach is faster than the naive searching method, but it is far from satisfying. First, κ-AT is a highly redundant structure. As κ increases, the space needed to store a κ-AT exponentially expands. Second, even though we have enough space to store κ-ATs, but the isomorphism test on large trees could also be slow enough to challenge our patience.

The bottleneck for the current κ-AT index as stated above forces us to improve the redundant structure of κ-AT index. It is noted that a κ-AT is made up by several $(\kappa - 1)$-ATs, so actually we do not need to store every κ-AT of the graph set, but only 1-ATs; every larger κ-AT can be represented by smaller ones, and we can organize them in a κ-AT lattice.

For example, the $1, 2$-ATs of G_2 in Fig. 2 are shown in Fig. 4, and we can see that the κ-ATs have a tidy hierarchical structure, each κ-AT is made of several $(\kappa - 1)$-ATs in the upper layer of the lattice.

To compact the index size, for each 1-AT shown in Fig. 4, we give them a unique ID, then use the IDs of the 1-ATs to represent 2-ATs, and give each 2-AT a unique ID (Our "unique" means that the ID of the AT in the same layer of the lattice must be different) to represent 3-ATs, and so on. Each κ-AT can be represented in a hierarchical structure finally linked to 1-ATs (shown in Fig. 5). We call this index structure "κ-AT lattice."

Fig. 5 shows how to dispatch unique IDs for each κ-AT and how to use IDs of $(\kappa - 1)$-ATs to represent κ-ATs based on the example $1, 2$-ATs shown in Fig. 4.

We can generate the κ-AT lattice in a relabeling procedure of each graph in the graph set. For a graph set, we first generate all its 1-ATs, and dispatch IDs for each unique 1-AT. After that, we store them in the first layer of the lattice, and then relabel each vertex in the graph set by the ID of their 1-AT. Next, we generate all 1-ATs of the new graph set and dispatch them unique IDs. These 1-ATs and their IDs are the element of the second layer of the lattice, this procedure continues until the layer number reaches the κ we set up.

Next, we introduce a technique to sequentialize a 1-AT. It is a difficult task to find out an adaptive sequentializing method for a general graph. Only very few methods can solve this problem using polynomial-time complexity. We can see, however, that a 1-AT can be organized in a star-like shape due to the special structure of the 1-AT. This gives rise to a method to transform a 1-AT into a sequence efficiently. For any 1-AT,

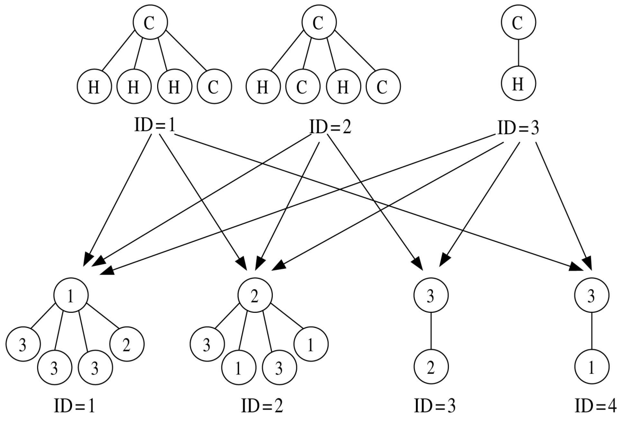


Fig. 5. How to compact the κ-AT lattice by "relabeling."

suppose that its root vertex is v, child vertices are $v_1, v_2, \ldots, v_n$, and the edges connecting the root vertex and child vertices are $e_1, e_2, \ldots, e_n$. We then group each child vertex and the corresponding edge connecting it to the root vertex into a 2-tuple; there will be n such tuples $(e_1, v_1), (e_2, v_2), \ldots, (e_n, v_n)$. Within these tuples, we define a partial ordering: if $W_e(e_1) < W_e(e_2)$ or $(W_e(e_1) = W_e(e_2)$ and $W_{ex}(v_1) < W_{ex}(v_2))$, then $(e_1, v_1) \prec (e_2, v_2)$, else $(e_1, v_1) \succ (e_2, v_2)$. In order to generate the sequence of a 1-AT and make sure that isomorphic 1-ATs should be mapped into the same sequence, we first sort the sequence of the tuples $(e_1, v_1), (e_2, v_2), \ldots, (e_n, v_n)$ based on the partial order defined above, and then insert the root vertex into the head of the sequence. Finally, we get a sequence of the form $(v)(e_1, v_1), (e_2, v_2), \ldots, (e_n, v_n)$.

We do not use the definition of level-n adjacent subtree in [25] since we need to fix the order of a node's siblings for convenience of the sequentializing procedure of 1-AT. If the sibling nodes of a κ-AT are already sorted, the sequentialization of a κ-AT will become much easier.

Algorithm 1 describes the procedure of generating a κ-AT lattice of a graph set.

```
Algorithm 1: GenLattice($D$,$L$,$\kappa$)
  Data: $D$:(graph set), $\kappa$(depth of the lattice).
  Result: $L$($\kappa$-$AST$ lattice).
1 Initiate $L = \phi, i = 1$;
2 while ($i \leq depth$) do
3     Generate all $i$-$AT$s of Graphs in D, sequentialize and
      store these $i$-$AT$s into $L[i]$ and dispatch them a
      unique ID;
4     for ($\forall G \in D$) do
5         for ($\forall$ vertex $v$ of $G$) do
6             $W_v(v)$=ID of $i$-$AT$ in $L[i]$;
7     $i = i + 1$;
```

For a given graph, let the κ-AT table of the graph be a table recording all the κ-AT IDs and the frequency they appear in the graph. For example, Table 1 shows the κ-AT table for the graph in Fig. 1a.

We maintain the κ-AT table for each graph in the graph set for κ-AT set join. For each query Q, first generate its κ-AT table based on the existing κ-AT lattice, then for each graph G in the graph set, calculate κ-$ATS(Q) \cap \kappa$-$ATS(G)$ by comparing the $(2\kappa - 1, 2\kappa)$ columns of their κ-AT tables.

TABLE 1
Example of κ-AT Table

ID[1]	Freq[1]	ID[2]	Freq[2]
1	2	1	2
2	1	2	1
3	7	3	6
		4	2

The usage of the κ-AT table allows us to store each graph's adjacent tree by their ID in the κ-AT lattice; this greatly reduces the space needed to store each graph's κ-ATs. In the query processing, the incoming query needs to be decomposed into κ-AT by a bottom-up search on the κ-AT lattice. The ID and number of its κ-AT can be collected during the search to generate the query's κ-AT table.

Algorithm 2 uses the κ-AT table for candidate filtering of Δ_G query (Q, ϵ) on the graph set.

```
Algorithm 2: Filter($Q$, $D$, $\epsilon$, $R$, $\kappa$)
  Data: $Q$(query graph), $\epsilon$($\Delta_G$ threshold), $D$(graph set),
        $\kappa$(index depth).
  Result: $R$(result set).
1 Initiate $R = \phi, j = 0$;
2 Generate Q's $\kappa$-$AT$ table $ind_Q$;
3 for ($G \in D$) do
4     Compare $ind_Q$ and $G$'s table $ind_G$ to calculate the
      number of their common $\kappa$-$AT$: $n_\kappa$;
5     if $n_\kappa \geq |V(Q)| - \epsilon \cdot (\delta(Q) - 1)^{\kappa-1}$ then
6         Add($ind_G$,$R$);
```

The set R returned by Algorithm 2 is the candidate answer set of the query.

4.2 κ-AT Index Maintenance

There are two steps in the maintenance of the index. The first step is the κ-AT lattice maintenance. Any nonpreviously existing κ-AT appearing in a new graph added to the set should be inserted into the lattice, and this can be done by decomposing the new graph and searching the κ-AT in the old lattice. If they do not exist, we insert them into the corresponding layer of the lattice. The other step is to generate the κ-AT table for the new graph. The κ-AT table can be attained by the searching and relabeling procedure.

The deletion procedure is similar to the insertion procedure. To do so, we generate all κ-ATs of the graph (these κ-ATs all have a 1-AT shape because we use the technique of searching and relabeling as defined before on the κ-AT lattice). Then for each κ-AT, we test if it is unique in the lattice, if so, we delete those κ-ATs from the κ-AT lattice, and delete the graph and its κ-AT table from the graph set.

It will cost a complete search on the graph set to test if a κ-AT is unique on the lattice; so for each κ-AT in the lattice, we maintain a "counter" that records the number of occurrences. If a new κ-AT is inserted into the lattice, we initialize its $\text{counter} = 1$. Every subsequent insertion of this k-AT, the counter gets increased by 1. When a deletion is encountered, we subtract the counter by 1. When the counter equals 1, and another deletion operation is encountered, then we will know that this κ-AT should be deleted from the lattice, and as such, the slow process of searching on the graph set can be avoided.

4.3 Combining κ-AT Index with Inverted Index

Although searching on the κ-AT index will be much faster than doing the Δ_G calculation, it is still a sequential search. It inevitably has to access all the κ-AT tables in the graph set, and thus, this process has a higher computational cost. If we regard each graph in the graph set as a keyword set with their κ-ATs as their keywords, then we can organize the κ-AT index into an inverted index to avoid the slowly sequential search. Sarawagi and Kirpal [25] give several approaches to use an inverted index for joining a set of keywords. After inverted index construction, we use the *Probe-Count* Algorithm [25] to calculate $|\kappa\text{-}ATS(g_1) \cap \kappa\text{-}ATS(g_2)|$, $\forall g_2 \in D$, and g_1 is the query. For each κ, we maintain an inverted table for all the κ-ATs (κ-inverted table). We use their IDs as the keywords and record their references in the graph set into the inverted list of this keyword. During a query processing, we first generate all the κ-ATs of the query and use the ID of the κ-AT as the keywords to pick up specific inverted lists in the κ-inverted table, then execute a join operation on those inverted lists using the *Probe-Count* Algorithm.

Algorithms 3 and 4 are used to construct an inverted index from a graph set and execute candidate filtering on the inverted index. The inverted index technique can greatly improve the time efficiency of the filtering algorithm. In Section 5, we will see that after applying the inverted index to the original κ-AT tables, the query processing time will remain constant with respect to the graph set size.

```
Algorithm 3: Inverted_Index_Construction($D$,$\kappa$,$inv\_idx$)
  Data: $D$(graph set), $\kappa$(index depth)
  Result: $inv\_idx$(inverted index)
1 Initiate $T = \phi, t = \phi$;
2 for $G \in D$ do
3     $T$=G's $\kappa$-$AT$ table;
4     for $i = 1$ to $length(T)$ do
5         $t = (T[2\kappa-1, i], T[2\kappa, i])$;
6         Add($inv\_idx$,$t[0]$,($ref(G)$,$t[1]$));
7 Sort each inverted-list in the index;
```

```
Algorithm 4: Filter_Inv($Q$,$inv\_idx$,$\epsilon$,$R$,$\kappa$)
  Data: $Q$(query graph), $\epsilon$($\Delta_G$ threshold),
        $inv\_idx$(inverted-index), $\kappa$(index depth)
  Result: $R$(result set)
1 Initiate $R = \phi, j = 0, ind\_Q = \phi$;
2 Generate Q's $\kappa - AT$ table $ind\_Q$;
3 for $i = 1$ to $length(ind\_Q)$ do
4     Pick up inverted-list of $ind\_Q[2\kappa - 1]$ in $inv\_idx$;
5 Execute Probe-Count join on those inverted-lists with
  parameter $\epsilon$, save the result to $R$;
6 return $R$;
```

5 Performance Evaluation

In this section, we evaluate the performance of κ-AT index by comparing with FG-$index$ [10] and DAG^3 index [18]. FG-$index$ is a kind of frequent-subgraph-based indexing technique while DAG index is a kind of graph-decomposition-based indexing technique. Note that δ is set to 0.1 for the FG-$index$ method. Since our proposed method is a hybrid of graph-decomposition-based indexing method and frequent-subgraph-based indexing method, we choose these two indexing techniques to compare with our proposed method. In the query performance experiments, we randomly pick up one graph from the graph set to form a query for each query performance experiment, and it is running three times to get the average response time of the query. Similar ways are adopted in the literature [10], [14], [18], [23], [24].

5.1 Data Sets and Settings

We use three kinds of data sets in our experiments: two real data sets and a synthetic data set, described as follows:

1. NCI/NIH AIDS Antiviral Screen data set. AIDS is a widely used real chemical compounds data set [6], [7], [8], [9], [10], [12], [14], [23], [24]. This data set is available publicly on the Website of the Developmental Therapeutics Program (http://dtp.nci.nih.gov/). All data are stored in SDFile format. We generate from the AIDS data set more than 40,000 graphs, which have an average number of 20 vertices and 26 edges and a maximum number of 25 vertices and 35 edges.
2. Protein Interaction Data set DIP (short for Database of Interacting Proteins). DIP is a protein interaction network data set [5], [22]. This data set is available on the Website of the Database of Interacting Proteins (http://dip.doe-mbi.ucla.edu/). We download from the Website all the eight newly species protein interaction data sets, including *D. melanogaster (fruit fly), S. cerevisiae (baker's yeast), E. coli, C. elegans, H. sapiens (Human), H. pylori, M. musculus (house mouse), and R. novegicus (Norway rat)*. From the eight protein interaction data sets, we generate more than 5,000 graphs that have an average number of 12 vertices and 36 edges and a maximal number of 17 vertices and 51 edges. Protein interaction networks are typical large sparse graphs [5].
3. Synthetic data set. The synthetic data are randomly generated by the algorithm used in the paper [26]. There are five parameters used in the generator: the database size $|D|$, the average graph size $|T|$, the number of seeds $|L|$, the size of seed graphs $|I|$, and the number of distinct edge and vertex labels N. The generated process is described as follows: a) Initialize the information for seed graphs, including the size and probability of each seed graph. The size of a seed graph is determined by a Poisson distribution with mean $|I|$ and the probability of the seed graph

3. The index proposed in [18] is called GDI, we would like to name it DAG since the paper uses Directed Acyclic Graph to represent decomposed subgraphs.

is generated randomly. b) Select a seed graph using the probability information of the seeds. c) For the selected seed graph, a corresponding graph is generated, the size of which is determined by a Poisson distribution with mean $|T|$. In our experiments, the parameters used for the generator are set as: $|D| = 10k{-}100k$, $|T| = 10$, $|L| = 200$, $|I| = 10$, and $N = 5$. The generated graphs have an average number of 10 vertices and 40 edges and a maximal number of 16 vertices and 48 edges when the parameter *graphsize* is fixed at 10.

Table 2 gives the settings of the four parameters used in our experiments, including ranges and default values. κ is a key parameter in our proposed indexing method. It can be used to control the index size and tune the index performance. Δ is the edit distance threshold. *Graphsize* is the average number of nodes of the graphs in the graph database. *DBSize* is the number of graphs in the graph database. The default values of *Graphsize* and *DBSize* are set as 10 and 10,000 since the *DAG* indexing method takes a very long time to build the indexes with large graph sizes and large database size.

All the algorithms were implemented using C++. The experiments were run on a PC with an 3.0 GHz CPU and 3GB memory, running a Windows XP operating system.

TABLE 2
Parameter Settings

Parameter	Range	Default Value
κ	1-5	3
Δ	1-5	3
GraphSize	10-180	10
DBSize	10000-100000	10000

5.2 Performance of Index Construction

In this section, we compare the index construction performance of our proposed *κ-AT* method with *DAG* and *FG-index* on three data sets in terms of index size and index construction time.

Figs. 6 and 7 show the index size and construction time of three data sets. We can see in Fig. 6 that the *DAG* indexing method has the worst performance in index size. This is because *DAG* needs to index all the decomposed subgraphs. Moreover, *DAG* will take more space in the case of larger *GraphSize* since the number of the decomposed subgraphs is exponential to *GraphSize*. Our indexing method *κ-AT* incorporates graph-decomposition-based indexing techniques with frequent-subgraph-based indexing techniques; therefore, it has less space costs than *DAG*. Among the three indexing methods, *FG-index* has the best performance in index size, since the method only needs to index some frequent subgraphs. Fig. 7 shows that *κ-AT* is superior to the other two indexing methods in index construction time. In the procedure of constructing indexes, the main CPU cost is subgraph isomorphism test. *DAG* has to invoke the subgraph isomorphism test operation to clean the duplicates of subgraphs and *FG-index* has to invoke the subgraph isomorphism test operation to find frequent subgraphs, while *κ-AT* invokes the subtree isomorphism test to collect the frequency information of the decomposed subtrees. It is obvious that subtree isomorphism test is faster than subgraph isomorphism test. *DAG* is the slowest in index construction time since the number of the decomposed subgraphs is much more than that of the found

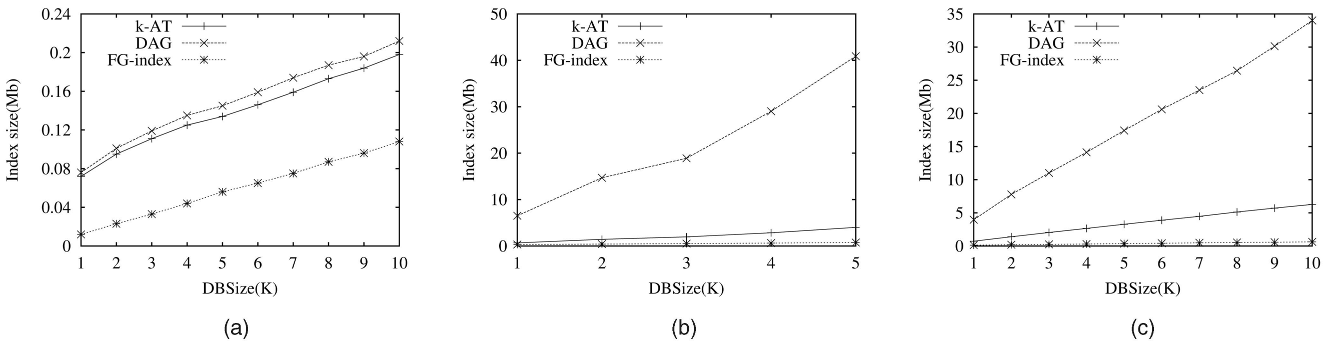


Fig. 6. Index size of three data sets. (a) Index size of AIDS. (b) Index size of DIP. (c) Index size of synthetic data set.

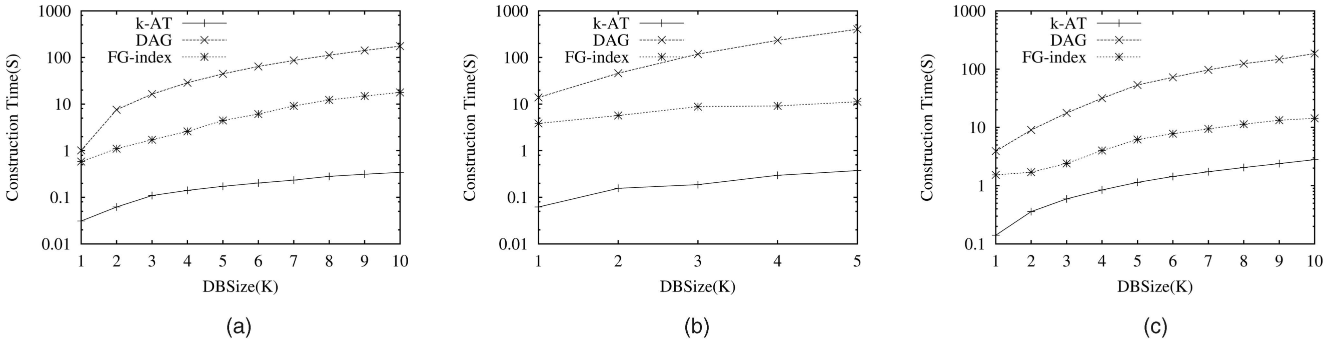


Fig. 7. Index construction time of three data sets. (a) Index construction time of AIDS. (b) Index construction time of DIP. (c) Index construction time of synthetic data set.

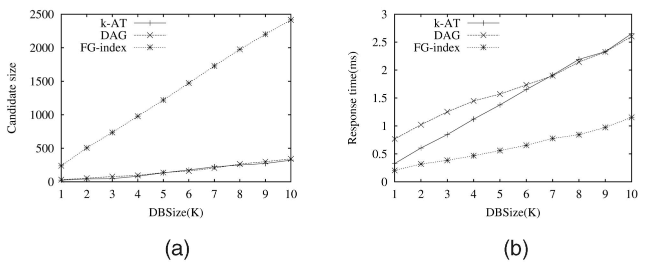


Fig. 8. Query processing performance and filtering ability of AIDS. (a) Filtering ability. (b) Query processing response time.

frequent subgraphs in *FG-index* and that of the decomposed subgraphs in κ-*AT*.

5.3 Performance of Query Processing and Filtering Ability

In this section, we compare the query processing and filtering performance of our proposed κ-*AT* method with *DAG* and *FG-index* on the three data sets.

Fig. 8 shows the performance of query processing and filtering ability of the three indexing techniques on the AIDS data set. We can see in Fig. 8a that *DAG* has the worst filtering performance, and κ-*AT* has almost the same filtering ability as *FG-index* but κ-*AT* is slightly better than *FG-index*. Fig. 8b shows that κ-*AT* has better query processing performance than *DAG* but slower than *FG-index*. This is because there are a few frequent subgraph patterns in the AIDS data set and a few of subgraph isomorphism test operations are invoked in *FG-index*. It can be noted that all the three methods have good query processing performance since all the response times are no more than 3 ms.

Fig. 9 shows the performance of query processing and filtering ability of three indexing techniques on the DIP data set. We can see in Fig. 9a that *DAG* has the worst filtering performance and *FG-index* has the best filtering performance on the DIP data set. κ-*AT* becomes better than *FG-index* when *DBSize* is larger than 5,000. Fig. 9b shows that in query processing performance, κ-*AT* and *DAG* are far superior to *FG-index* and κ-*AT* is slightly better than *DAG*. Since there are only a few protein interactions, only a few distinct labels of edges exist in the DIP data set. This leads to inefficient query processing on infrequent edges, and therefore, *FG-index* gets very bad query processing performance in the DIP data set.

Fig. 10 shows the performance of query processing and filtering ability of three indexing techniques on the synthetic data set. We can see in Fig. 10a that κ-*AT* has almost the same filtering performance as *DAG*. Fig. 10b

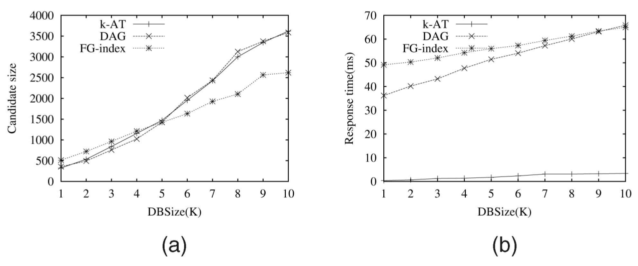


Fig. 10. Query processing performance and filtering ability of the synthetic data set. (a) Filtering ability. (b) Query processing response time.

shows that the query processing performance of κ-*AT* is far superior to *FG-index* and *DAG*. This is because *FG-index* and *DAG* have to invoke a lot of subgraph isomorphism test operations in the synthetic data set while κ-*AT* invokes subtree isomorphism test operations. The cost of subtree isomorphism test is much faster than that of the subgraph isomorphism test as pointed out before.

5.4 Performance of Scalability

In this section, we test the scalability performance of our proposed indexing methods κ-*AT* (with inverted) and κ-*AT* (without inverted) on the synthetic data set. The range of parameter *DBSize* is set as 10 k to 100 k. In the previous experiments, it is set as 1 to 10 k since the *DAG* method will take a very long time to build the index when the database is very large.

Fig. 11 shows the scalability performances of κ-*AT* when κ varies. We can see in Fig. 11a that κ-*AT* has a good filtering ability when κ is smaller than 4. This is why we set the default value of κ as 3. Fig. 11b shows that the query processing response time increases linearly as κ increases. The optimized method with inverted index improves the performance by more than 40 percent.

Fig. 12 shows the scalability performance of our proposed indexing method κ-*AT* when the edit distance threshold Δ varies. In Fig. 12a, we can see that the candidate size increases as Δ increases, since the size of the results is also increasing as Δ increases. Fig. 12b shows that the query processing response time is constant as Δ increases, since the query processing cost is not depending on Δ. The optimized method with inverted index decreases the response time by about two times.

Fig. 13 shows the scalability performance of our proposed indexing method κ-*AT* as *GraphSize* varies (the range of variation 20 to 180). In the figures, we can see that both candidate size and response time increase linearly as *GraphSize* varies. Fig. 13b shows that our proposed indexing

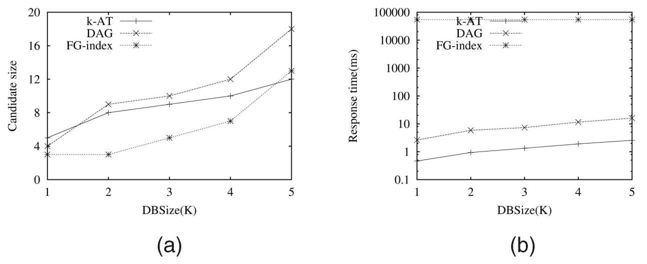


Fig. 9. Query processing performance and filtering ability of DIP. (a) Filtering ability. (b) Query processing response time.

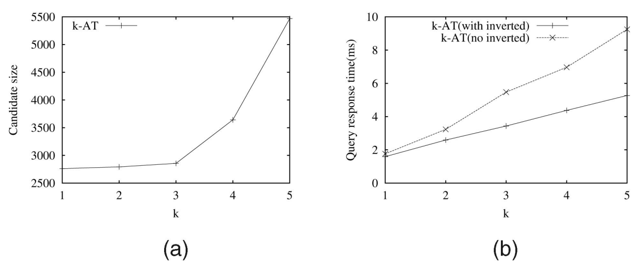


Fig. 11. Scalability versus κ. (a) Candidate size. (b) Query response time.

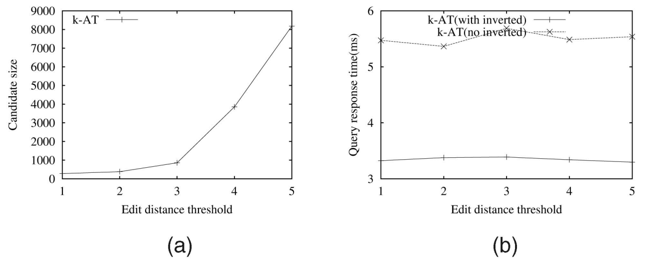


Fig. 12. Scalability versus edit distance threshold. (a) Candidate size. (b) Query response time.

method κ-AT has a very good scalability performance versus graph size, and the optimized method with inverted index improves the response time about 30 percent.

Fig. 14 gives the scalability performance of κ-AT as database size varies. Fig. 14a shows that the candidate size increases linearly as database size increases. This is because the number of answers increases as database size increases. Fig. 14b shows that the optimized method with inverted index improves the response time about 20 percent and the improvement increases gradually as database size increases.

6 Conclusions and Future Work

We have developed a graph set indexing method for similarity matching. By decomposing the graphs into small pieces (κ-ATs), and pairing-up these pieces, we evaluate the global similarity between them. In order to seek for a compromise between frequent-subgraph-based indexing methods and graph-decomposition-based indexing methods, we use the redundant subtree structure: κ-AT pattern for index construction. κ-AT records more structural information on each vertex than a normal graph-decomposition-based indexing method, and while maintaining the simple structure of tree. By calculating the number of common κ-ATs of two graphs, we can estimate the graph edit distance between them. This gives us a method for indexing and candidate filtering in a graph set for similarity matching. Experimental results evince that when applied to large sparse graph sets and when the Δ_G threshold is not too large, prefiltering on κ-AT index can be both fast and accurate.

There are still several opportunities for future improvements on the κ-AT index.

1. The lower bound estimation is loose on nonsparse graphs because we assume that the edit operation can happen on each V or E in a graph at a same probability. In fact, due to the unbalanced structure of a graph, the probability of different V or E, which will be covered by the edit operations, is also different; so we actually can apply probability model to our index to further tighten the lower bound estimation, under a different assumption of the distribution of edit operations. Also we can apply some machine learning methods to calculate the distribution from statistical information about large amounts of queries.
2. The κ-AT model still suffers from more false positives than the graph pattern model in the filtering phase because tree patterns are still "weak" compared with graph patterns. For this reason, other methods should be investigated, which can combine them together to further optimize the performance of the index such as the method proposed in [15].
3. Paralleling to the edit distance threshold query, some different kinds of querying forms, such as k-NN query and top-k query, can be discussed under our filtering principle and indexing model. This opens up many interesting topics in the similarity matching on graph sets.

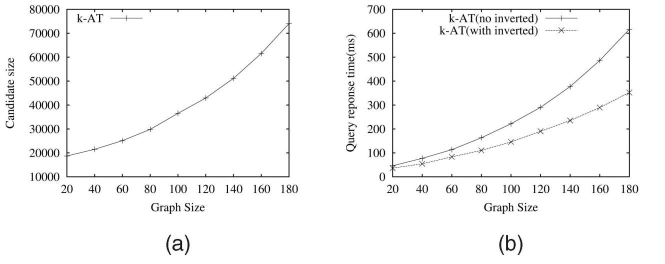


Fig. 13. Scalability versus graph size. (a) Index construction time. (b) Query response time.

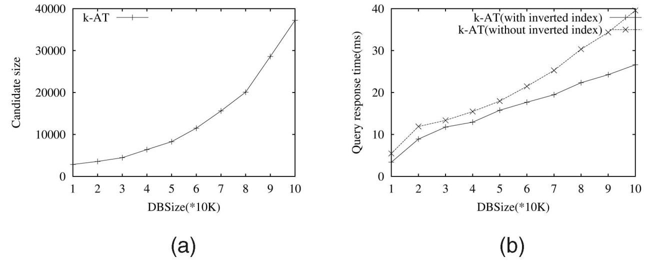


Fig. 14. Scalability versus database size. (a) Candidate size. (b) Query response time.

Acknowledgments

The authors thank a lot to Mr. Zinan Xu and Mr. Miaoqiang Su for their hard work on the experiments. This research was partially supported by the National Natural Science Foundation of China under Grant Nos. 60873011, 60933001, 60828004, and 60973018; the National Basic Research Program of China under Grant No. 2006CB303103; and the 863 Program under Grant Nos. 2007AA01Z192 and 2009AA01Z131.

References

[1] T.H. Cormen, "Np Completeness," *Introduction to Algorithms*, W. Yu, ed., second ed., vol. 7, pp. 620-630. China Machine Press, 2007.
[2] E. Sutinen and J. Tarhio, "On Using q-Gram Locations in Approximate String Matching," *Proc. Third Ann. European Symp. Algorithms*, pp. 327-340, 1995.
[3] J. Beasley and N. Christofides, "Theory and Methodology: Vehicle Routing with a Sparse Feasibility Graph," *European J. Operational Research*, vol. 98, no. 3, pp. 499-511, 1997.
[4] R. Nallapati, A. Ahmed, W. Cohen, and E. Xing, "Sparse Word Graphs: A Scalable Algorithm for Capturing Word Correlations in Topic Models," *Proc. Seventh IEEE Int'l Conf. Data Mining Workshops (ICDMW '07)*, pp. 343-348, 2007.
[5] C. Lin, D. Jiang, and A. Zhang, "Prediction of Protein Function Using Common-Neighbors in Protein-Protein Interaction Networks," *Proc. Sixth IEEE Int'l Symp. BioInformatics and BioEng. (BIBE '06)*, pp. 251-260, 2006.

[6] X. Yan, P.S. Yu, and J. Han, "Graph Indexing: A Frequent Structure-Based Approach," *Proc. ACM SIGMOD,* pp. 335-345, 2004.
[7] J.H. Xifeng Yan and P.S. Yu, "Graph Indexing Based on Discriminative Frequent Structure Analysis," *ACM Trans. Database Systems,* vol. 30, no. 4, pp. 960-993, 2005.
[8] H. Shang, Y. Zhang, X. Lin, and J.X. Yu, "Taming Verification Hardness: An Efficient Algorithm for Testing Subgraph Isomorphism," *Proc. 34th Int'l Conf. Very Large Data Bases,* pp. 364-375, 2008.
[9] X. Yan, P.S. Yu, and J. Han, "Substructure Similarity Search in Graph Databases," *Proc. ACM SIGMOD,* pp. 766-777, 2005.
[10] J. Cheng, Y. Ke, W. Ng, and A. Lu, "Fg-Index: Towards Verification-Free Query Processing on Graph Databases," *Proc. ACM SIGMOD,* pp. 857-872, 2007.
[11] J. Cheng, Y. Ke, and W. Ng, "Efficient Query Processing on Graph Databases," *ACM Trans. Database Systems,* vol. 34, no. 1, pp. 1-44, 2009.
[12] C. Chen, X. Yan, and P.S. Yu, "Towards Graph Containment Search and Indexing," *Proc. 33rd Int'l Conf. Very Large Data Bases,* pp. 926-937, 2007.
[13] S. Zhang, M. Hu, and J. Yang, "Treepi: A Novel Graph Indexing Method," *Proc. IEEE 23rd Int'l Conf. Data Eng.,* pp. 966-975, 2007.
[14] H. He and A.K. Singh, "Closure-Tree: An Index Structure for Graph Queries," *Proc. 22nd Int'l Conf. Data Eng.,* p. 38, 2006.
[15] P. Zhao, J.X. Yu, and P.S. Yu, "Graph Indexing: Tree + Delta $\geq$ Graph," *Proc. 33rd Int'l Conf. Very Large Data Bases,* pp. 938-949, 2007.
[16] D. Eppstein, "Subgraph Isomorphism in Planar Graphs and Related Problems," *J. Graph Algorithms and Applications,* vol. 3, no. 3, pp. 1-27, 1999.
[17] J.P. Kukluk, L.B. Holder, and D.J. Cook, "Algorithm and Experiments in Testing Planar Graphs for Isomorphism," *J. Graph Algorithms and Applications,* vol. 8, no. 3, pp. 313-356, 2004.
[18] D.W. Williams, J. Huan, and W. Wang, "Graph Database Indexing Using Structured Graph Decomposition," *Proc. 23rd Int'l Conf. Data Eng.,* pp. 976-985, 2007.
[19] D. Shasha, J.T.-L. Wang, and R. Giugno, "Algorithmics and Applications of Tree and Graph Searching," *Proc. 21st ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems,* pp. 39-52, 2002.
[20] D. Justice and A. Hero, "A Binary Linear Programming Formulation of the Graph Edit Distance," *IEEE Trans. Pattern Analysis and Machine Intelligence,* vol. 28, no. 8, pp. 1200-1214, Aug. 2006.
[21] O. Johansson, "Graph Decomposition Using Node Labels," doctoral dissertation, Royal Inst. of Technology, 2001.
[22] Y. Tian and J.M. Patel, "Tale: A Tool for Approximate Large Graph Matching," *Proc. 24th Int'l Conf. Data Eng.,* pp. 963-972, 2008.
[23] H. Jiang, H. Wang, P.S. Yu, and S. Zhou, "Gstring: A Novel Approach for Efficient Search in Graph Databases," *Proc. 23rd Int'l Conf. Data Eng.,* pp. 566-575, 2007.
[24] L. Zou, L. Chen, J.X. Yu, and Y. Lu, "A Novel Spectral Coding in a Large Graph Database," *Proc. 11th Int'l Conf. Extending Database Technology,* pp. 181-192, 2008.
[25] S. Sarawagi and A. Kirpal, "Efficient Set Joins on Similarity Predicates," *Proc. ACM SIGMOD,* pp. 743-754, 2004.
[26] M. Kuramochi and G. Karypis, "Frequent Subgraph Discovery," *Proc. 2001 IEEE Int'l Conf. Data Mining,* pp. 313-320, 2001.

Guoren Wang received the PhD degree from Northeastern University, China, in 1996. He is currently a professor and the director of the Computer System Institute of the College of Information Science and Engineering at Northeastern University. His research interests include XML data management, query processing and optimization, bioinformatics, high-dimensional indexing, parallel database systems, and P2P data management. He has published more than 100 research papers in international conferences and journals. He is a senior member of the CCF.

Bin Wang received the PhD degree in computer science from Northeastern University in 2008. He is currently an associate professor in the Computer System Institute at Northeastern University. His research interests include design and analysis of algorithms, databases, data quality, and distributed systems. He is a member of the CCF.

Xiaochun Yang received the PhD degree in computer science from Northeastern University, China, in 2001. She is a professor in the Department of Computer Science at Northeastern University, China. Her research interests include data quality, data privacy, and distributed data management for sensor networks and P2P networks. She has received a China Program Award for New Century Excellent Talents in Universities, a China National Natural Science Foundation Grant, a Fok Ying Tong Education Foundation Grant, and a China Ministry of Education Grant. She is a member of the ACM, the IEEE Computer Society, and a senior member of the CCF.

Ge Yu received the BE and ME degrees in computer science from Northeastern University of China in 1982 and 1986, respectively, and the PhD degree in computer science from Kyushu University of Japan in 1996. He has been a professor at Northeastern University of China since 1996. His research interests include database theory and technology, distributed and parallel systems, embedded software, and network information security. He is a member of the IEEE, the ACM, and a senior member of the CCF.