G-Hash: Towards Fast Kernel-based Similarity Search in Large Graph Databases

Xiaohong Wang[1],Aaron Smalter[1],Jun Huan[1], Gerald H. Lushington[2]
[1]Department of Electrical Engineering and Computer Science,
University of Kansas,Lawrence, KS, USA
[1]{xwang85,asmalter,jhuan}@eecs.ku.edu
[2]Molecular Graphics and Modeling Laboratory,
University of Kansas,Lawrence, KS, USA
[2]glushington@ku.edu

ABSTRACT

Structured data including sets, sequences, trees and graphs, pose significant challenges to fundamental aspects of data management such as efficient storage, indexing, and similarity search. With the fast accumulation of graph databases, *similarity search* in graph databases has emerged as an important research topic. Graph similarity search has applications in a wide range of domains including cheminformatics, bioinformatics, sensor network management, social network management, and XML documents, among others.

Most of the current graph indexing methods focus on subgraph query processing, i.e. determining the set of database graphs that contains the query graph and hence do not directly support similarity search. In data mining and machine learning, various graph kernel functions have been designed to capture the intrinsic similarity of graphs. Though successful in constructing accurate predictive and classification models for supervised learning, graph kernel functions have (i) high computational complexity and (ii) non-trivial difficulty to be indexed in a graph database.

Our objective is to bridge graph kernel function and similarity search in graph databases by proposing (i) a novel kernel-based similarity measurement and (ii) an efficient indexing structure for graph data management. Our method of similarity measurement builds upon local features extracted from each node and their neighboring nodes in graphs. A hash table is utilized to support efficient storage and fast search of the extracted local features. Using the hash table, a graph kernel function is defined to capture the intrinsic similarity of graphs and for fast similarity query processing. We have implemented our method, which we have named G-hash, and have demonstrated its utility on large chemical graph databases. Our results show that the G-hash method achieves state-of-the-art performance for k-nearest neighbor (k-NN) classification. Most importantly, the new similarity measurement and the index structure is scalable to large database with smaller indexing size, faster indexing construction time, and faster query processing time as compared to state-of-the-art indexing methods such as C-tree, gIndex, and GraphGrep.

EDBT 2009, March 24–26, 2009, Saint Petersburg, Russia.

Keywords

graph similarity query, graph classification, hashing, graph kernels, k-NNs search.

1. INTRODUCTION

Structured data including sets, sequences, trees and graphs, pose significant challenges to fundamental aspects of data management such as efficient storage, indexing, component search (e.g.subgraph/supergraph search) and similarity search. With the fast accumulation of graph databases, *similarity search* in graph databases has emerged as an important research topic. Graph similarity search has applications in a wide range of domains including cheminformatics, bioinformatics, sensor network management, social network management, and XML documents, among others. For example, in chemistry and pharmacy, there is a fast accumulation of chemical molecule data. Once a new chemical is synthesized, the properties of the chemical may be revealed through querying existing chemicals with known properties. Fast similarity search in large graph databases enable scientists and data engineers to build accurate models for graphs, identify the intrinsic connections between graph data, and reduce the computational cost of processing large databases.

Graph queries can be classified into two categories: (i) subgraph query and (ii) similarity query. *Subgraph query* aims to identify a set of graphs that contain a query graph [12]. *Similarity query* aims to identify similar graphs in a graph database to a query, according to a distance metric. There are two types of similarity query, i.e. k-NNs query and range query. In k-NNs query, the k most similar graphs are reported. In range query, all graphs within a predefined distance to the query graph are reported. In this paper we address the problem of k-NN similarity search in large databases of graphs.

Similarity search on graphs is challenging. We argue that an ideal design of fast similarity search should achieve the following three related (sometimes contradicting) objectives: accurate, running-time efficient, space efficient. By accurate, we emphasize that the similarity measurement should capture the intrinsic similarity of objects. By running-time

efficient, it is well-known that operations on graphs, such as subgraph isomorphism, are NP-complete problems [6] which require us to design efficient algorithms to avoid exhaustive search as much as possible. By space efficient, the index structure should not add a significant storage overhead to graph databases.

The most straightforward approach for similarity measurement is to embed a graph in a high dimensional Euclidian space, known as the *feature space*, and use spatial indexing techniques for similarity search. Current feature extraction methods operate in two different ways: (i) enumeration of substructures in each graph separately (e.g. generating a set of random walks from a graph) [15], and (ii) enumeration of substructures from a set of graphs (e.g. mining frequent patterns) [30, 4, 32, 29]. Though widely used, there are several limitations of adapting the feature extraction and feature indexing strategy for similarity search. First, the feature extraction process is computational intensive, especially for mining large graph databases. Second, feature extraction may produce many features that occupy a large amount of memory. Different feature selection methods have been devised to identify "discriminative" features [30]. However features that are efficient for database search may not be equally good for similarity search and a trade-off is needed.

Here we explore a new way of graph similarity search where similarity is defined using graph kernel functions. Rather than extracting features explicitly, a *kernel function* maps data objects to a high-dimensional functional space and measure the similarity of objects by computing the inner product of the objects in the functional space. The advantage of kernel function based similarity measurement is that kernel function usually have high statistical power, i.e. affording high classification accuracy. The major difficulty of applying kernel function for database search is that (i) kernel functions for graphs are expensive to compute and (ii) there is no clear way to index the kernel function computation over a large graph database.

Our approach, called G-hash, aims to devise a kernel function that can be efficiently computed over a large graph database. In our model, graphs are reduced to point sets that are compared directly via a kernel function. Typically such an approach would lose a great deal of information in the rich graph structure, but we avoid this by compressing much of the topological information into feature vectors describing each graph vertex. This approach provides a compact graph representation that is information rich, yet easy to compare. We then hash graph objects using the compressed set representation. The hash keys are based these sets and hence similar objects in the hash table are positioned in the same or nearby cells. Once we have hashed graphs in a database into the table, we can find all similar nodes and then calculate the distances between the query graph and the graphs in the database based on them and kernel function to obtain the k-NNs of the query graph.

In summary, our contributions in this papers are:

- Devised a graph kernel function and related index structure for fast graph similarity search
- Our index has linear time to compute (in terms of total number of nodes in a graph database) and may be constructed on-line with dynamic insertion and deletion
- We have proved that the new graph kernel function and its related index structure achieved a better trade-off between capturing the intrinsic similarity of graphs and fast computation for large graph databases.

This paper is organized as follows. We review related work in the areas of hashing, indexing, and kernels for graphs in section 2. Next, we formally define graphs and graph similarity search in section 3. We discuss the details of our index structure and kernel function in section 4. Finally we present a comprehensive experimental study using our methods and competing methods, and conclude with a few remarks on the study and future work.

2. RELATED WORK

In this section we discuss related work, starting from indexing in graph databases in general, including subgraph search, approximate subgraph search, and graph similarity search, and move to graph kernel functions.

2.1 Subgraph Search

Many of the recent methods for subgraph search adopt a similar framework, decomposing graphs into a set of smaller pieces, treating each piece as a feature, and building a feature-based index structure for subgraph query. Methods that belong to this category include *GraphGrep* [21], *gIndex* [30], *FG-Index* [4], *Tree+Delta* [32], and *GDIndex* [29].

The simplest type of feature in use for graph indexing is walks (including path as special cases) as pioneered in *GraphGrep* [21]. Path are easy to retrieve and easy to work with. The simplicity of paths limit their expressiveness. For example, using paths, we could not distinguish the topology of a ring and a chain (where all paths from the two graphs are paths with different sizes).

Recognizing the limitation of paths, *gIndex* [30] and *FG-Index* [4] build indices using general subgraphs, which can easily distinguish between paths and cycles in graphs and hence are more powerful. The limitation of subgraph features is that subgraph enumeration and matching are computational intensive procedures. In order to manage these obstacles, these methods extract only *frequent* subgraph features. Similar methods, *Tree+Delta* [32] and *TreePI* [31] use frequent tree patterns (as well as some discriminative graph features) instead of frequent subgraph patterns.

The method *GDIndex* [29] also uses subgraphs as the basic index feature, but does not restrict itself to *frequent* subgraph features. In addition to a subgraph-based index, this method incorporates a hash table of subgraphs for fast isomorphism lookup. While this method's focus is subgraph search, it supports similarity search as well.

2.2 Approximate Subgraph Search

Besides strict subgraph search, some methods relax the isomorphism matching constraint and allow partial or approximate matches. This is a relatively new direction, and hence not many methods currently address the problem. One such method, *SAGA* [24], was designed for biological pathway analysis. First, it builds an index based on graph fragments. It then uses a graph distance measure to allow for vertex mismatches and gaps when when matching candidate graphs to a query.

Another method *gApprox* [3] is similar to the *gIndex* [30] method, in spirit and name, as well as authors. This approach seeks to mine frequent *approximate* patterns from a

graph database and use these for indexing. They also explore the notion of *approximately frequent*.

The method TALE [25] is also designed for approximate graph matching. It's focus, however, is on handling large graphs with thousands of vertices and edges.

2.3 Graph Similarity Search

There are three commonly used ways to measure graph similarity. The first is *edit distance*. That is, given a set of operations on graph vertices and edges (such as insertion, deletion, relabeling), how many many such operations are required to transform graph G into another graph, G'. We can parameterize the method by assigning different costs to different operations and summing over the total cost of all operations. Edit distance is an intuitively attractive approach to graph similarity, but unfortunately in practice it is costly to compute (NP-hard). *C-Tree* [12] is a widely used graph indexing scheme that also does not use graph pieces as features. Instead, it organizes database graphs in tree based structure, where interior nodes are *graph closures*, and leaf nodes are database graphs. Importantly, *C-Tree* also supports similarity queries where the previous two methods, *GraphGrep* and *gIndex*, do not.

One method, *GString* [13] is a subgraph similarity query method and uses graph fragments as features as well. The approach is somewhat different than the previous two feature-based subgraph search methods. Complex graphs are first reduced into connected graphs of fewer nodes, each of which represents a specific fragment. Canonical node numbering is used to create a string representation for each graph in a database. An index that supports similarity search is then constructed in the form of a suffix tree. This method combines the expressive power of subgraphs and simplified graphs with the speed of string querying and matching.

In addition, maximal common subgraph [2] and graph *alignment* [9, 27] are used to measure graph similarity. Unfortunately, there is no easy way to index both measurements for large graph databases.

2.4 Graph Kernel Functions

Several graph kernel functions have been studied. The pioneering work was done by Haussler in his work on *R-convolution* kernel, providing a framework for many current graph kernel functions to follow [11]. The R-convolution kernel is based on the notion of decomposing a discrete structure (e.g. a graph) into a set of component objects (e.g. subgraphs). We can define many such decompositions, as well as kernels between pairs of component objects. The R-convolution framework ensures that no matter the choice of decompositions or component kernels, the result is always a symmetric, positive semi-definite function, or a kernel function between compound objects. This key insight allows the problem of finding kernel functions for discrete structures to be reduced to those of finding decompositions and kernel functions between component objects. The R-convolution kernel can be extended to allow weighting of the kernel between various components, via the Weighted Decomposition Kernel [18].

Recent progresses of graph kernel functions could be roughly divided into two categories. The first group of kernel functions consider all possible components in a graph (e.g. all possible paths) and hence measure the global similarity of two graphs. These include product graph kernels [10], random walk based kernels [15], and kernels based on shortest paths between pair of nodes [1]. The second group of kernel functions try to capture the local similarity of two graphs by specifying a (finite) subset of components and counting the shared components only according to the finite subset of components. These include a large class of graph kernels called spectrum kernels [8] and recently frequent subgraph kernels [23]. The most efficient kernel function that we notice is proposed by Vishwanathan [28] for global similarity measurement with complexity $O(n^3)$ where n is the maximal number of nodes in graphs. Different from global similarity measure, local similarity capturing is known to be expensive since subcomponent matching (e.g. subgraph isomorphism) is an NP-hard operation.

We adopt a recently develop graph wavelet matching kernel and make it scalable for large databases.

3. BACKGROUND

Before we proceed to discuss the algorithmic details, we present some general background regarding a computational analysis of graphs which includes (i) graph kernel functions, and (ii) graph wavelet analysis.

3.1 Graphs

A *labeled graph* G is described by a finite set of nodes V and a finite set of edges $E \subset V \times V$. In most applications, a graph is labeled, where labels draw from a label set λ. A labeling function $\lambda : V \cup E \rightarrow \Sigma$ assigns labels to nodes and edges. In *node-labeled graphs*, labels are assigned to nodes only and in *edge-labeled graphs*, labels are assigned to edges only. In *fully-labeled graphs*, labels are assigned to nodes and edges. We may use a special symbol to represent missing labels. If we do that, node-labeled graphs, edge-labeled graphs, and graphs without labels are special cases of fully-labeled graphs. Without loss of generality, we deal with fully-labeled graphs only in this paper. For the label set Σ we do not assume any structure of Σ now; it may be a field, a vector space, or simply a set.

Following convention, we denote a graph as a quadruple $G = (V, E, \Sigma, \lambda)$ where V, E, Σ, λ are explained before. A graph $G = (V, E, \Sigma, \lambda)$ is a *subgraph* of another graph $G' = (V', E', \Sigma', \lambda')$, denoted by $G \subseteq G'$, if there exists a 1-1 mapping $f : V \rightarrow V'$ such that

- for all $v \in V, \lambda(v) = \lambda'(f(v))$
- for all $(u, v) \in E, (f(u), f(v)) \in E'$
- for all $(u, v) \in E, \lambda(u, v) = \lambda'(f(u), f(v))$

In other words, a graph is a subgraph of another graph if it preserve the node labels, edge relations, and edge labels.

A *walk* of a graph is a list of node $v_1, v_2, \ldots, v_n$ such that v_i and v_{i+1} is connected for all $i \in [1, n-1]$. A *path* is a walk which contains no repeated nodes, i.e. for all $i \neq j$ we have $v_i \neq v_j$

3.2 Reproducing Kernel Hilbert Space

Kernel functions are powerful computational tools to analyze large volumes of graph data [11]. The advantage of kernel functions is due to their capability to map a set of data to a high dimensional Hilbert space without explicitly computing the coordinates of the data. This is done through a special function called a *kernel* function.

A binary function $K : X \times X \rightarrow \mathbb{R}$ is a *positive semi-definite* function if

$$\sum_{i,j=1}^{m} c_i c_j K(x_i, x_j) \geq 0 \quad (1)$$

for any $m \in \mathbb{N}$, any selection of samples $x_i \in X$ ($i = [1, n]$), and any set of coefficients $c_i \in \mathbb{R}$ ($i = [1, n]$). In addition, a binary function is *symmetric* if $K(x, y) = K(y, x)$ for all $x, y \in X$. A symmetric, positive semi-definite function ensures the existence of a Hilbert space $\mathcal{H}$ and a map $\Phi : X \rightarrow \mathcal{H}$ such that

$$k(x, x') = \langle \Phi(x), \Phi(x') \rangle \quad (2)$$

for all $x, x' \in X$. $\langle x, y \rangle$ denotes an inner product between two objects x and y. The result is known as the Mercer's theorem and a symmetric, positive semi-definite function is also known as a Mercer kernel function [19], or *kernel* function for simplicity.

By projecting the data space to a Hilbert space, kernel functions provide a uniformed analytical environment for various data types including graphs, regardless of the fact that the original data space may not look like a vector space at all. This strategy is known as the "kernel trick" and it has been applied to various data analysis tasks including classification [26], regression [5] and feature extraction through principle component analysis [20], among others.

3.3 Graph Wavelets Analysis

Wavelet functions are commonly used as a means for decomposing and representing a function or signal as its constituent parts, across various resolutions or scales. Wavelets are usually applied to numerically valued data such as communication signals or mathematical functions, as well as to some regularly structured numeric data such as matrices and images. Graphs, however, are arbitrarily structured and may represent innumerable relationships and topologies between data elements. Recent work has established the successful application of wavelet functions to graphs for multiresolution analysis. Two examples of wavelet functions are the *Haar* and the *Mexican hat*.

Crovella et al. [7] have developed a multi-scale method for network traffic data analysis. For this application, they are attempting to determine the scale at which certain traffic phenomena occur. They represent traffic networks as graphs labeled with some measurement such as bytes carried per unit time.

Maggioni et al. [17] demonstrate a general-purpose biorthogonal wavelet for graph analysis. In their method, they use the dyadic powers of an diffusion operator to induce a multiresolution analysis. While their method applies to a large class of spaces, such as manifolds and graphs, the applicability of their method to attributed chemical structures is not clear. The major technical difficulty is how to incorporate node labels in a multiresolution analysis.

4. FAST GRAPH SIMILARITY SEARCH WITH HASH FUNCTIONS

As discussed above, current graph query methods provide fast query time but not good similarity measurements. Kernel functions can provide better similarity measurement but the kernel matrix calculation is time-consuming so it is hard to build index structure by using them directly. To address this problem, we propose a new method, G-hash. Current methods usually focus on either accuracy or speed. Our proposed method defines similarity based on Wavelet Graph matching kernels (WA) and uses hash table as index structure to speed up graph similarity query. Below we first give an introduction to WA method.

4.1 Introduction to Wavelet Graph matching kernels

The idea behind WA method is to first convert the graph into sets by compressing property information in the neighborhood around each vertex, and then apply non-recursive alignment kernel to compute similarity between graphs. This method contains two important concepts: *h-hop neighborhood* and *discrete wavelet functions*. The *h-hop neighborhood*of one node v, denoted by $N_h(v)$, refers to a set of nodes which are h hops away from the node v according to the shortest path. *Discrete wavelet functions* refer to the defined wavelet functions, shown in equation 3, applying to h-hop neighborhood.

$$\psi_{j,k} = \frac{1}{h+1} \int_{j/(k+1)}^{(j+1)/(k+1)} \varphi(x) dx \quad (3)$$

where $\varphi(x)$ is *Haar* or *Mexican Hat* wavelet function and h is the hth partition after $\varphi(x)$ is partitioned into h+1 intervals on the domain [0,1) and j is between 0 and h.

Based on the above two definitions, we can now apply wavelet analysis to graphs. Wavelet functions are used to create a measurement summarizing the local topology of a node. Equation 4 shows such a wavelet measurement, denoted by $\Gamma_h(v)$, for a node v in a graph G.

$$\Gamma_h(v) = C_{h,v} \times \sum_{j=0}^{k} \psi_{j,k} \times \bar{f}_j(v) \quad (4)$$

where $C_{h,v}$ is a normalization factor with

$$C_{h,v} = (\sum_{j=0}^{h} \frac{\psi_{j,h}^2}{|N_h(v)|})^{-1/2}, \quad (5)$$

and $\bar{f}_j(v)$ is the average feature vector value of atoms that are at most j-hop away from v with

$$\bar{f}_j(v) = \frac{1}{|N_j(v)|} \sum_{u \in N_j(v)} f_u. \quad (6)$$

and f_u denotes the feature vector value of the node v. Such feature vector value can be one of the following four types:nominal, ordinal, internal and ratio. For ratio and internal node features, we directly apply the above wavelet analysis to get local features. For nominal and ordinal node features, we could first build a histogram and then use wavelet analysis to extract local features. After the node v is analyzed, a list of vectors $\Gamma^h(v) = \{\Gamma_1(v), \Gamma_2(v), \cdots, \Gamma_h(v)\}$, called wavelet measurement matrix, can be obtained. In this way, a graph can be decomposed into a set of node vectors. Since the wavelet has strongly positive and strongly negative regions, these wavelet-compressed properties represent a comparison between the local and distant vertex

neighborhood. Structural information of a graph has therefore been compressed into the vertex properties through wavelets. Hence, we can now ignore the topology and focus on matching vertices. The kernel function is defined on these sets. Given two graphs G and G' for example, the graph matching kernel is

$$k_m(G, G') = \sum_{(u,v)\in V(G)\times V(G')} K\left(\Gamma^h(u), \Gamma^h(v)\right), \quad (7)$$

$$K(X, Y) = e^{\frac{-\|X-Y\|_2^2}{2}}. \quad (8)$$

The WA methods shows a good definition of similarity between graphs through kernel functions, as validated in the experimental part [22]. One issue, however, is that the overall time complexity of the wavelet-matching kernel is $O(m^2)$, and that of the kernel matrix is $O(n^2 \times m^2)$, where n is the size of the database and m is the average node number of all graphs. When the size of the database increases, the kernel matrix calculation time grows very quickly.

4.2 Fast graph similarity search with hash functions

Following the idea of using a function to map each node in a graph to a feature space, we may design a kernel function for fast similarity search. Specifically, we have the following two observations.

- When the node vector of the node u in the graph G is dramatically different from that of the node v in the graph G', the RBF kernel value between the node u and node v is small and has little contribution to the graph kernel. So if we just count those pairs of nodes which have similar node vectors, the kernel matrix will reflect the similar similarity measurement between two graphs to that of WA method and the time will be saved.
- Similar objects in the hash table are positioned closer if the hash keys are based on the node vectors. So the hash table can help us to find similar node pairs rapidly. In addition, if all graphs in the database are hashed into the table, one cell may contain many similar nodes which belong to different graphs. Since these nodes are all similar, only one time RBF kernel calculation using two nodes of them is enough. Node overlay provides another chance to save time.

Based on the above two observations, we introduce our method, called hash table based k-NNs query (G-hash). G-hash is based on WA method to provide an accurate similarity measurement and use hashing to improve time complexity. The entire process is described as follows.

4.2.1 Index construction

First, we need to decompose all graphs in the library database into node vectors by using wavelet transformation same as that in the method of WA. Since the WA method is relatively insensitive to small perturbation of the hop distance parameter and the use of different wavelet functions makes little difference [22], h is picked as 2 and the *Haar* wavelet is used for simplification.

After node vectors are obtained, the hash table will be built using the graphs in the database. At this time, a hash function needs to be constructed to make sure that similar nodes can be hashed into the same cell. That means the hash keys should be associated with node vectors. Since we obtains the node vectors according to the node label and the neighboring information, we can construct hash keys in the same idea. We discretize each feature in the node vector to an integer. We encode a node label directly as a n bit-string with all zeros except a single 1 which indicate the label. Other features are rounded to the nearest integer. After the node vector is changed to a list of integer numbers, we then convert a node vector to a string by represent each integer number using its binary format and concatenate these obtained bit strings delimited by underscore. Such string is the hash key of the corresponding node. Take the graph P shown in figure 1 for example.

EXAMPLE 4.1. *We pick node labels and the number of neighboring nodes with different labels as node features. So there are a total of four node features. The node features for this graph just belong to one types of value: nominal. To get the node vectors, we first obtain histogram of node features. The histogram is shown in figure 1. Then we use wavelet function to extract local features. Take node P_3 for example, the local obtained feature vector after using wavelet analysis for h=0 is $[b, 2, 0, 1]$. The sample hash table is shown in figure 1.*

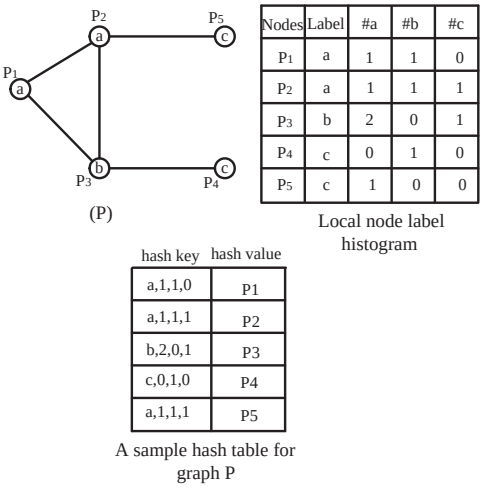

Figure 1: Example graph.

Notice: In this framework, we assume no information for query graphs when we build the hash table for indexing.

4.2.2 k-NNs Query Processing

To obtain the k-NNs of a given query graph, we need to calculate the distance between it and all graphs in the database. Here the distance is defined using the kernel function between these two graphs.

$$\begin{aligned} d(G, G') &= \sqrt{\|\phi(G) - \phi(G')\|_2^2} \\ &= \sqrt{\langle \phi(G) - \phi(G'), \phi(G) - \phi(G') \rangle} \\ &= \sqrt{\langle \phi(G), \phi(G) \rangle + \langle \phi(G'), \phi(G') \rangle - 2\langle \phi(G), \phi(G') \rangle} \\ &= \sqrt{k_m(G, G) + k_m(G', G') - 2k_m(G, G')}. \end{aligned} \quad (9)$$

where $k_m(G,G)$ is the kernel function between the graph G and itself, $k_m(G',G')$ is the kernel function between the graph G' and itself, and $k_m(G,G')$ is the kernel function between the graph G and G'). In the following section, we will discuss how to calculate all of them.

Though hashing the nodes of the query graph into the hash table, we can get the kernel function

$$k_m(G,G') = \sum_{v\in G', u\in simi(v)} K\left(\Gamma^h(u), \Gamma^h(v)\right), \qquad (10)$$

Where $simi(v)$ are a set containing the nodes in graph G hashed to the same cell as the node v does. We use the following decoding to get the graph number containing these nodes and the node number.

Clearly, the similarity of two graphs is determined only by similar node pairs instead of all node pairs, which will save computational time. Since similar nodes also may be hashed into the neighboring cells, to increase the accuracy, we also count the nodes in neighboring cells when the size of graph is a larger (e.g. greater than 40).

Since only similar nodes are involved into the kernel calculation, $K\left(\Gamma^h(u), \Gamma^h(v)\right) \approx 1$ if RBF kernel is used. So the equation 10 can be written into

$$k_m(G,G') \approx \sum_{v\in G', u\in simi(v)} 1 = \sum_{v\in G'} |simi(v)|, \qquad (11)$$

where $|simi(v)|$ is the number of nodes contained in $simi(v)$. That means that we only need to count the number of similar nodes, belonging to the graph G, of each node in the query graph G' and sum them to get the kernel. Similarly, we can calculate the kernel between each graph and itself.

After the above calculations, we obtain a distance vector with each value corresponding to the distance between the query graph and a graph in the database. Through sorting this distance vector, we can obtain the k-NNs of this given query graph.

4.2.3 Dynamic insertion and deletion

To insert a new graph into the database, we hash all nodes of the new graph into the hash table. After insertion, only those cells associated with these nodes contain more nodes and all other cells has no changes. In addition, since the new graph has a limited number of nodes, insertion operations involve less time and the size of index little. Deletion operations are similar to insertion. To delete a graph from the database, we calculate the key corresponding to each node in this graph and then delete each node from the cell containing them.

5. EXPERIMENTAL STUDY

We have performed a comprehensive evaluation of our method by evaluating its effectiveness (in classification), efficiency, and scalability. We will apply our methods on chemical databases. For chemical compounds, the node features include numeric features and boolean atom features. Numeric features include element type, atom partial charge, atom electron affinity, atom free electrons count and atom heavy valence,etc. Boolean atom features include atom in acceptor, atom in terminal carbon, atom in ring,atom is negative, atom is axial,etc. Here, we just use a single of atomical feature: element type.

We have compared our methods with the Wavelet Alignment Kernel [22], C-tree [12],GraphGrep [21] and gIndex [30] as performance benchmarks. Our method, WA method, GrapGrep and gIndex are developed in C++ and compiled using g++. C-tree was developed in Java and compiled using Sun JDK1.5.0. All experiments were done on an Intel Xeon EM64T 3.2GHz, 4G memory cluster running Linux.

The parameters for WA, G-hash, C-tree, GraphGrep and gIndex are set in the following way. we set $h = 2$ and use *haar* wavelet function for WA and G-hash. For C-tree, we choose the default values, namely, setting the minimum number of child node $m = 20$, the maximum number $M = 2m - 1$ and the NBM method [12] is used for graph mapping. For GraphGrep and gIndex, we use default parameters.

5.1 Data sets

We chose a number data sets for our experiments. The first five data sets are established data taken from Jorisson/Gilson Data Sets[14]. The next six data sets are manually extracted from BindingDB data sets [16]. The last one is NCI/NIH AIDS Antiviral Screen data set (`http://dtp.nci.nih.gov/.`). Table 1 shows these data sets and their statistical information.

5.1.1 Jorissen sets

The Jorissen data sets contain information about chemical-protein binding activity. The target values are drug's binding affinity to a particular protein. There are five proteins for which 100 chemical structures are selected with 50 chemical structures clearly bind to the protein(called "active" ones) and the other 50 ones similar to the active ones but clearly not bind to the target protein. See [14] for the further details.

5.1.2 BindingDB sets

The BindingDB database contains data for proteins and chemicals that bind to the proteins. We manually selected 6 proteins with a wide range of known interacting chemicals (ranging from tens to several hundreds). For the purpose of classification, we convert the real-valued binding activity measurements to binary class labels. This is accomplished by dividing the data set into two equal parts according to the median activity reading (we also deleted compounds whose activity value is equal to zero).

5.1.3 NCI/NIH AIDS Antiviral Screen data set

NCI/NIH AIDS Antiviral Screen data set contains 42,390 chemical compounds retrieved from DTP's Drug Information System. There is a total 63 types of atoms in this data set; the most frequent ones are C, O, N, and S. The data set contains three types of bonds: single-bond, double-bond and aromatic-bond. We selected all chemicals to build our graph database and randomly sampled 1000 chemicals as the query data set.

5.2 Similarity Measurement Evaluation with Classification

we compared classification accuracy using k-NN classifier on Jorissen sets and BindingDB sets with difference similarity measurement. For the WA method, we use wavelet matching kernel function to obtain kernel matrix, and then calculate distance matrix to obtain k nearest neighbors. For

Table 1: Data sets statistics.#S:total number of compounds, #P:number of positive compounds,#N:number of negative compounds,#Node: average number of nodes, #Edge: average number of edges.

data set	#S	#P	#N,	#Node	#Edge
PDE5	100	50	50	44.7	47.2
CDK2	100	50	50	38.4	40.6
COX2	100	50	50	37.7	39.6
FXa	100	50	50	45.75	48.03
AIA	100	50	50	48.33	50.61
AChE	183	94	89	29.1	32.0
ALF	151	61	60	23.8	25.2
EGF-R	497	250	247	24.6	27.1
HIV-P	267	135	132	43.0	46.2
HSP90	109	55	54	29.84	32.44
MAPK	336	168	168	28.0	31.1
HIV-RT	482	241	241	22.18	24.39

G-hash, we compute graph kernel according to our algorithmic study section and then calculate the k nearest neighbors. For C-tree, we directly retrieve the nearest neighbor. We use standard 5-fold cross validation to obtain classification accuracy, which is defined as $(TP+TN)/S$ where TP stands for true positive, TN stands for true negative and S is the total number of testing samples. We report the average accuracy. In our experiments, we set $k = 5$.

The accuracy results are shown in figure 2. The accuracy, precision, and recall statistical information is shown in Table 2, 3 and 4. From figure 2, we know that G-hash outperforms C-tree on all twelve data sets, with at least 8% improvement on all of them. The average accuracy difference between G-hash and C-tree is about 13%. WA method outperforms G-hash, the average difference between them is about 2% because, most likely because we adopt some simplifications on distance matrix calculation. From what is discussed above, it is clear that kernel based similarity measurement is better than edit distance based similarity measurement. Since the accuracy of k-NN classifier is associated with the value of k, we also study the accuracy with respect to the value of k on these data sets to test whether the parameter k has any effect on accuracy performance comparison. Results show that accuracy performance comparison is insensitive to the parameter k.

Table 2: Accuracy results statistical information for G-hash, C-tree and WA on all data sets.

method	G-hash	C-tree	WA
average	64.55	51.64	66.23
derivation	2.68	5.95	4.83

5.3 Scalability

5.3.1 Index Construction

In this section, we apply G-hash, WA [22], C-tree [12], GraphGrep [21] and gIndex [30] on NCI/NIH AIDS Antiviral Screen data set.

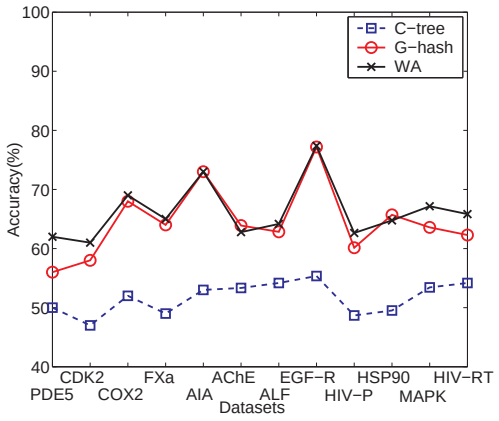

Figure 2: Comparison of averaged classification accuracy over cross validation trials.

Table 3: Average Precision for different data sets. Asterisk ($*$) denotes the best precision for the data sets among WA, G-hash and C-tree methods.

dataset	WA	G-hash	C-tree
PDE5	83.16*	75.78	31.2
CDK2	73.81*	67.42	51.82
COX2	75.88*	66.98	54.85
FXa	95.78*	91.19	29.36
AIA	98.93*	98.33	36
AChE	66.46	73.59*	62.63
ALF	72.14*	69.82	32.59
EGF-R	72.75	80*	55.41
HIV-P	56.9	64.64*	40.81
HSP90	58.19	73.63*	48.72
MAPK	66.31*	66.21	53.25
HIV-RT	69.38*	61.87	54.11

We compare index size and average index construction time for different methods. Towards that end, we sampled different number of graphs ranging from 10,000 to 40,000. Figure 3 shows the index construction time in milliseconds with respect to the size of database for G-hash, C-tree, GraphGrep and gIndex. The construction time for G-hash is much lower than those for other three methods because of the adoption of a hash table. In addition, when the data set size increases, the construction time for C-tree, GraphGrep and gIndex grows faster than that for G-hash since the construction of C-tree, GraphGrep and gIndex involve relatively complicated index structure. So G-hash outperforms C-tree, GraphGrep and gIndex on index construction time.

Figure 4 shows index size with respect to database size. The index size of G-hash shows a steady growth with increasing database size while that of C-tree increases sharply since C-tree need to save the whole tree structure while G-hash just need to save the hash table.

5.3.2 Query Processing Time

Figure 5 shows the query time in milliseconds with respect to the size of database. When the size of database increases,

Table 4: Average recall for different data sets. Asterisk (∗) denotes the best recall for the data sets among WA, G-hash and C-tree methods.

dataset	WA	G-hash	C-tree
PDE5	58.06*	56.2	46.93
CDK2	55.87*	53.54	46.7
COX2	63.57	68.06*	51.46
FXa	58.23	62.41*	42.06
AIA	64.81	66.27	55.33
AChE	63.63*	62.82	44.15
ALF	61.25	66.16*	53.83
EGF-R	79.64*	77.51	55.81
HIV-P	63.4*	61.96	47.62
HSP90	63.4*	61.96	47.62
MAPK	70.52	73.6*	72.16
HIV-RT	67.78*	66.83	56.78

G-hash scales better than C-tree. There is no direct way that we could compare G-hash and subgraph indexing methods such as G-index and Graphgrep since G-hash search for similar graph and G-index (and Graphgrep) searches for the occurrences of a subgraph in a database.

In the following, we sketch one way to use subgraph indexing methods for similarity search. This method contains three steps: (i) randomly sample subgraphs from a query, (ii) use those subgraphs as features and compute the occurrences of the subgraphs in graph databases, and (iii) search for nearest neighbors in the obtained feature space. Clearly the overall query processing time depends on (i) how many subgraphs we use and (ii) how fast we can identify the occurrences of the subgraphs in a graph database. We estimate the lower bound of the overall query processing time by randomly sampling a SINGLE (one) subgraph from each of the 1000 querying graph and use subgraph indexing method to search for the occurrence of the subgraph. We record the average query processing time for each query. This query processing time is clearly the lower bound since we use only one subgraph from the query graph.

Figure 5 shows the experimental results of comparing C-tree, GraphGrep and gIndex. When the size of database is 40,000, the query time for C-tree, Graphgrep and gIndex are nearly 8 times,10 times and 100 times as that for G-hash respectively.

Finally, we compared C-tree and G-hash with varying k values for k-NN search. the results are shown in Figure 6. The query time of C-tree increases with the increasing k and the running time of G-hash is insensitive to the number of k.

6. CONCLUSIONS AND FUTURE WORKS

Graphs are a kind of general structural data have been widely applied in many fields such as cheminformatics and bioinformatics, among others. A lot of significant researchers have been attracted to current data management and mining technique. Proposing an efficient similarity graph query method is a significant challenge since most existed methods focus on speed and provide poor accuracy. In order to address this problem, we have presented a new graph query method, G-hash. Through our experimental study, we have

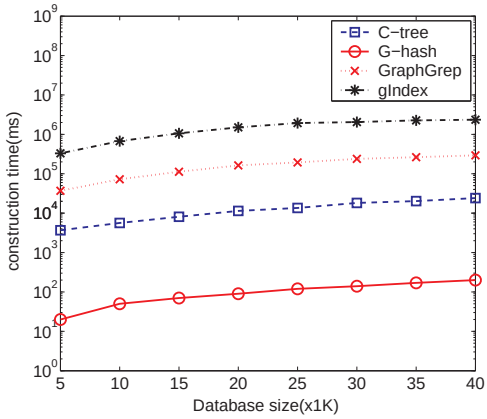

Figure 3: Index construction time for NCI/NIH AIDS data set.

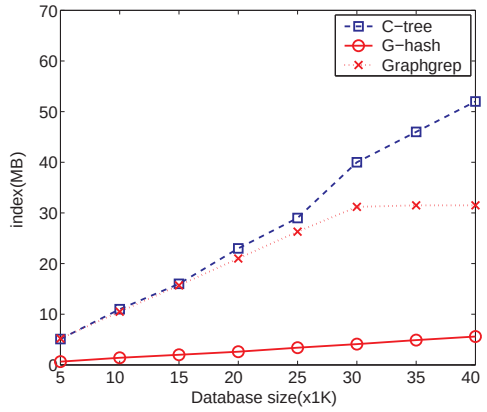

Figure 4: Index size for NCI/NIH AIDS data set.

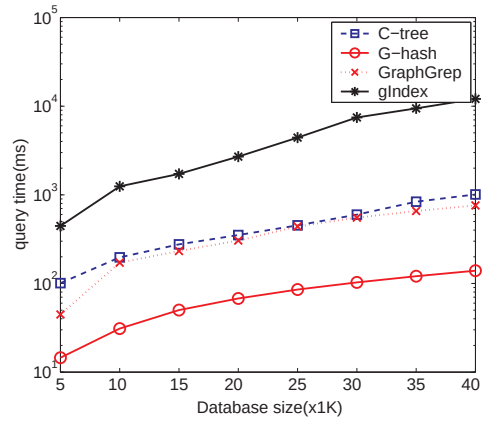

Figure 5: Query time for NCI/NIH AIDS data set.

shown that compared to C-tree [12], G-hash provides about a 13% improvement to accuracy. The query time for G-hash is much less than that for C-tree [12], GraphGrep[21] and gIndex[30] especially when the database size becomes larger. In addition, G-hash shows a better scalability on index con-

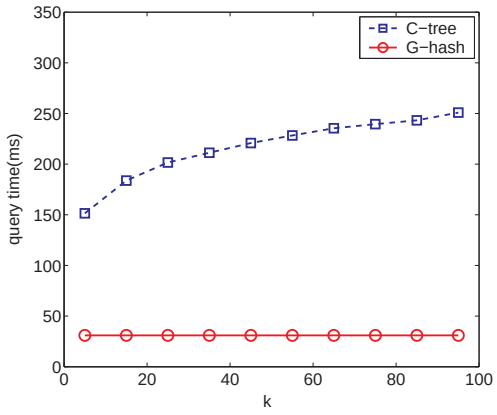

Figure 6: Query time with respect to different k for k-NN query.

struction time and efficiency in space usage. This means G-hash can support large graph database search.

Acknowledgments

We thank H. He and A. K. Singh for sharing the code of C-tree,D. Shasha, J.T.L. Wang and R. Giugno for sharing the code of GraphGre, X. Yan, P. S. Yu and J. Han for sharing the code of GIndex. This work was supported by the KU Center of Excellence for Chemical Methodology and Library Development (NIH/NIGM award #P50 GM069663)and an NIH grant #R01 GM868665.

7. REFERENCES

[1] K. Borgwardt and H. Kriegel. Shortest-path kernels on graphs. In *Proceedings of the International Conference on Data Mining (ICDM)*, 2005.

[2] Y. Cao, T. Jiang, and T. Girke. A maximum common substructure-based algorithm for searching and predicting drug-like compounds. *Bioinformatics*, 24(13):i366–74, 2008.

[3] C. Chen, X. Yan, F. Zhu, and r. Jiawei Han. gapprox: Mining frequent approximate patterns from a massive network. In *Proc. 2007 Int. Conf. on Data Mining (ICDM'07)*, 2007.

[4] H. Cheng, X. Yan, J. Han, and C.-W. Hsu. Discriminative frequent pattern analysis for effective classification. In *Proceedings of the 23rd IEEE International Conference on Data Engineering (ICDE)*, 2007.

[5] R. Collobert and S. Bengio. Svmtorch: Support vector machines for large-scale regression problems. *Journal of Machine Learning Research*, 21, 2001.

[6] S. A. Cook. The complexity of theorem-proving procedures. In *STOC*, 1971.

[7] M. Crovella and E. Kolaczyk. Graph wavelets for spatial traffic analysis. *Infocom*, 3:1848–1857, 2003.

[8] M. Deshpande, M. Kuramochi, and G. Karypis. Frequent sub-structure-based approaches for classifying chemical compounds. *IEEE Transactions on Knowledge and Data Engineering*, 2005.

[9] H. Fröhlich, J. K. Wegner, F. Sieker, and A. Zell. Optimal assignment kernels for attributed molecular graphs. In *Proceedings of the 22nd international conference on Machine learning*, 2005.

[10] T. Gärtner, P. Flach, and S. Wrobel. On graph kernels: Hardness results and efficient alternatives. In *Sixteenth Annual Conference on Computational Learning Theory and Seventh Kernel Workshop*, 2003.

[11] D. Haussler. Convolution kernels on discrete structures. *Technical Report UCSC-CRL099-10, Computer Science Department, UC Santa Cruz*, 1999.

[12] H. He and A. K. Singh. Closure-tree: an index structure for graph queries. In *Proc. International Conference on Data Engineering'06 (ICDE)*, 2006.

[13] H. Jiang, H. Wang, P. S. Yu, and S. Zhou. Gstring: A novel approach for efficient search in graph databases. In *Proc. International Conference on Data Engineering'07 (ICDE)*, 2007.

[14] R. Jorissen and M. Gilson. Virtual screening of molecular databases using a support vector machine. *J. Chem. Inf. Model.*, 45(3):549–561, 2005.

[15] H. Kashima, K. Tsuda, and A. Inokuchi. Marginalized kernels between labeled graphs. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2003.

[16] T. Liu, Y. Lin, X. Wen, R. N. Jorrisen, and M. Gilson. Bindingdb: a web-accessible database of experimentally determined protein-ligand binding affinities. *Nucleic Acids Research*, 35:D198–D201, 2007.

[17] M. Maggioni, J. B. Jr, R. Coifman, and A. Szlam. Biorthogonal diffusion wavelets for multiscale representations on manifolds and graphs. In *Proc. SPIE Wavelet XI*, volume 5914, 2005.

[18] S. Menchetti, F. Costa, and P. Frasconi. Weighted decomposition kernels. In *Proceedings of the Twenty-second International Conference on Machine Learning*, pages 585–592, 2005.

[19] B. Schölkopf and A. J. Smola. *Learning with Kernels*. the MIT Press, 2002.

[20] B. Schölkopf, A. J. Smola, and K.-R. Müller. Kernel principal component analysis. *Advances in kernel methods: support vector learning*, pages 327–352, 1999.

[21] D. Shasha, J. T. L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *Proceeding of the ACM Symposium on Principles of Database Systems (PODS)*, 2002.

[22] A. Smalter, J. Huan, and G. Lushington. Graph wavelet alignment kernels for drug virtual screening. In *Proceedings of the 7th Annual International Conference on Computational Systems Bioinformatics*, 2008.

[23] A. Smalter, J. Huan, and G. Lushington. Structure-based pattern mining for chemical compound classification. In *Proceedings of the 6th Asia Pacific Bioinformatics Conference*, 2008.

[24] Y. Tian, R. C. McEachin, D. J. States, and J. M.Patel. SAGA: a subgraph matching tool for biological graphs. *Bioinformatics*, 23(20:232–239, 2007.

[25] Y. Tian and J. Patel. TALE: a tool for approximate large graph matching. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2008.

[26] V. Vapnik. *Statistical Learning Theory*. John Wiley, 1998.

[27] J.-P. Vert. The optimal assignment kernel is not positive definite. Technical Report HAL-00218278, French Center for Computational Biology, 2008.

[28] S. V. N. Vishwanathan, K. M. Borgwardt, and N. N. Schraudolph. Fast computation of graph kernels. In *In Advances in Neural Information Processing Systems*, 2006.

[29] D. Williams, J. Huan, and W. Wang. Graph database indexing using structured graph decomposition. In *Proceedings of the 23rd IEEE International Conference on Data Engineering (ICDE)*, 2007.

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

[31] S. Zhang, M. Hu, and J. Yang. treepi: A new grah indexing method. In *ICDE*, 2007.

[32] P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: Tree + delta $\geq$ graph. In *VLDB*, 2007.