GADDI: Distance Index based Subgraph Matching in Biological Networks

Shijie Zhang, Shirong Li, and Jiong Yang
Dept. of Electrical Engineering and Computer Science
Case Western Reserve University
10900 Euclid Avenue, Cleveland, OH 44106
{shijie.zhang, shirong.li, jiong.yang}@case.edu

ABSTRACT

Currently, a huge amount of biological data can be naturally represented by graphs, e.g., protein interaction networks, gene regulatory networks, etc. The need for indexing large graphs is an urgent research problem of great practical importance. The main challenge is size. Each graph may contain thousands (or more) vertices. Most of the previous work focuses on indexing a set of small or medium sized database graphs (with only tens of vertices) and finding whether a query graph occurs in any of these. In this paper, we are interested in finding all the matches of a query graph in a given large graph of thousands of vertices, which is a very important task in many biological applications. This increases the complexity significantly. We propose a novel distance measurement which reintroduces the idea of frequent substructures in a single large graph. We devise the novel structure distance based approach (GADDI) to efficiently find matches of the query graph. GADDI is further optimized by the use of a dynamic matching scheme to minimize redundant calculations. Last but not least, a number of real and synthetic data sets are used to evaluate the efficiency and scalability of our proposed method.

1. INTRODUCTION

With the emergence of bioinformatics and social science applications, a large amount of data can be represented as graphs. Thus, there is an increasing interest in developing fast indexing and matching algorithms that operate on graphs. Graphs can be used to model complex data objects in a number of applications, e.g., network intrusion detection [33, 26], the semantic web [2], behavioral modeling [43, 34], VLSI reverse engineering [49], link analysis [22, 25, 36], and chemical compound classification [9, 30, 15, 10]. In the real world, graph models are constructed to capture the structural and relational characteristics of a variety of datasets arising in other areas such as physical sciences (e.g., chemistry, fluid dynamics, astronomy, structural mechanics, and ecosystem modeling), life sciences (e.g.,biological networks), and national defense (e.g., information assurance, network intrusion, infrastructure protection, and terrorist-threat prediction/identification).

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the ACM. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permissions from the publisher, ACM.
EDBT 2009, March 24–26, 2009, Saint Petersburg, Russia.
Copyright 2009 ACM 978-1-60558-422-5/09/0003 ...$5.00

Among many graph based applications, it is quite important to retrieve the occurrences of a query graph in database graphs efficiently. To speed up this process, a number of algorithms have been designed [5, 23, 38, 46, 51, 52, 53, 17, 18, 40, 44, 47, 48, 11, 41, 7, 28, 29, 39, 37, 3, 8]. Most of these techniques can only be applied to small graphs (of tens or hundreds of vertices and edges) in a database of multiple graphs. The databases are large in the way of the number of database graphs. Some of these methods deal with whether any database graph contains the query graph or not, rather than finding all the matches of the query graph in the database graph.

In real life applications, however, we need to not only deal with large database graphs, but also find all the matches of the query graph. For example, biological networks (protein-protein interaction networks, and gene regulatory networks, etc.) are often much larger than the graphs used in previous exact indexing methods. A single biological network may contain thousands or tens of thousands of vertices. Biologists may want to find all the occurrences of a particular pattern (subgraph), e.g., protein type A interacts with protein type B and C, and protein type C interacts with protein type A and D. In different occurrences of the pattern, the exact proteins involved may be different since multiple proteins may share the same type. As a result, all occurrences of a particular pattern need to be retrieved. Therefore, we want to solve the following problem for the necessity of real life research—how to find all the matches of a query graph in a large database graph? Namely, given a query graph, possibly large, and a large database graph with thousands or tens of thousands of vertices and edges, we want to find all subgraphs in the database graph that are isomorphic to the query graph.

Subgraph isomorphism test is believed to be an NP hard problem and exact indexing huge graphs (e.g., of millions of vertices) is practically infeasible. In fact, indexing a graph with thousands of vertices is already very difficult. To the our best knowledge, many previous methods only apply to graphs of tens of vertices. In addition, most of biological networks, e.g., protein interaction networks, are of thousands of vertices. Therefore, we restrict our applications to biological networks or small social networks. Due to the inherent differences of the characteristics of the underlying datasets and the problem definition, algorithms developed for the database of multiple graphs setting may not be used to solve this single graph database problem. Thus, it is necessary to develop a novel solution.

When the database is composed of a number of graphs, many of the predominant methods are frequent-substructure based. Researchers preprocess the database and adopt a filter-and-verification process

to speed up the subgraph search. False positives are removed by a given pruning strategy. Then, a subgraph isomorphism algorithm is performed on each of the remaining candidates to obtain the final results. These methods have proven to be effective and efficient. However, since we have only a single database graph, the old definition of "frequency" cannot apply. The difficulty of defining "frequency" over a single database graph has been addressed by [32, 4, 12]. Furthermore, since the database graph is much larger than the database graphs for which the frequent subgraph mining tools designed, those tools may not work at all. The change of the problem setting requires us to define frequency from a new perspective, and at the same time segment the database graph in a meaningful way.

The inherent difficulty of indexing a single graph of thousands vertices and edges lies in the fact that the subgraph isomorphism is an NP-hard problem. In the traditional database indexing research, the data set size is very large. Thus the goal is to optimize the disk access time. However, in the graph indexing problem setting, the raw graph is not very large, e.g., in the range of megabytes. The computation time to find all occurrences of a subgraph in a graph database is very long. There exist two extreme solutions: (1) Store and index all possible subgraphs of a graph. This is not practically feasible due to the exponential number of possible subgraphs for a graph of thousands edges and nodes, which may require terabyte of storage. (2) Only store the raw database graph. Since the size of the raw database graph is small, it can be easily fit in the main memory. However, the query (matching) time will be very long due to the NP-hard complexity. As a result, we need identify a solution which lies somewhere between these two extremes, that only utilizes an index structure of a reasonable size and can provide efficient query time.

Before providing a solution, we first propose the inequality property, which is the general requirement of any distance based pruning for a matching algorithm. In this paper we present the index structure in the form of distance, because distance is much easier to compute than most substructures which may introduce tedious subgraph isomorphism tests.

Afterwards, we present our index-based method for subgraph matching, called GADDI. GADDI employs a novel graph indexing method, the NDS distance (neighboring discriminating substructure distance). Most existing graph indexing methods only index subgraphs (paths, trees or general subgraphs), which may result in a huge amount of index substructures in addition to the subgraph isomorphism tests. The indexing unit of NDS distance is closely related to a pair of neighboring vertices. The neighborhood concept captures the local graph structure between each pair of vertices, and leads to an index substructure with high pruning power. The number of indexing units is proportional to the number of neighboring vertices in the database, which allows the index to grow in a more controllable pace. To make the index construction process more efficient, we optimize the procedure by using the property of neighboring vertices.

After the part of index construction, we propose an innovative matching algorithm for query process. The algorithm is based on twoway pruning. Any database graph vertex matched to a query graph vertex needs to have some vertices around it in order to meet the inequality property. On the other hand, after a vertex has been matched, we will remove some of its neighboring vertices since they can never be matched. The convenience in employing distances as index structures accelerates the pruning process. Since we are to find all the matches of the query graph, much of the calculation may be redundant. We incorporate a dynamic matching scheme into our matching algorithm to avoid wasted calculation. The dynamic matching scheme is specifically designed for the purpose of finding all the matches of the query graph.

The main contributions of this paper are as follows:

1. We propose GADDI—an index based graph matching algorithm in a single large graph, especially for biological networks. GADDI indexes the NDS distance between pairs of neighboring vertices. It achieves high pruning power and its size scales linearly with the number of neighboring vertex pairs. We introduce an innovative graph matching algorithm, which applies a two-way pruning and incorporates a dynamic matching scheme.
2. By applying GADDI to real applications, we show its effectiveness, significant performance improvements over existing methods, and ability to efficiently accomplish query processing in a large database graph.

The remainder of this paper is organized as follows: Related work is presented in Section 2. Section 3 defines the preliminary concepts. Section 4 and 5 describe our indexing mechanism, and the matching algorithm, respectively. Experimental results are presented in Section 6, and the final conclusions are drawn in Section 7.

2. RELATED WORK

We use graphs to model complex data objects in the real world, e.g., chemical compounds, biological networks, images, XML documents and social networks. Due to its wide usage, it is important to organize, access, and analyze graph data efficiently. As a result, graph database research has attracted a large amount of attention from the database and data mining communities, such as subgraph search in a database of multiple graphs [5, 23, 38, 46, 51, 52, 53, 16], approximate subgraph matching [17, 40, 44, 47, 48, 11], frequent subgraph mining [21, 31, 45, 19, 20, 35, 27], and correlation subgraph query[24].

Regarding to the problem studied in this paper, the first category of related research lies in subgraph isomorphism algorithms. Ullmann [41] proposed a subgraph matching algorithm based on a state space search method with backtracking. However, this algorithm is prohibitively expensive for querying against a large database graph. Recently, Cordella [7] proposed a new subgraph isomorphism algorithm for large graphs. These algorithms do not utilize any indexing structures by preprocessing the database graph.

Among many graph based applications, it is quite important to retrieve those database graphs containing the query graph efficiently. This is called a subgraph search problem and is closely related to work in this paper. To speed up the subgraph search, researchers preprocess the database and adopt a filter-and-verification framework. First, false positives are removed by a pruning strategy. Then, a subgraph isomorphism algorithm is performed on each of the remaining candidates to obtain the final results. Many pruning strategies have been proposed, which can be divided into two sub-categories.

The first sub-category is the frequent discriminate substructure based filtering. The approaches in this sub-category apply data mining

techniques to extract some discriminating substructures, then build inverted index for each feature. Query graph Q is denoted as a set of features, the pruning power of which is always dependent on the set of selected features. With the inverted indexes, we can find the complete set of candidates. Many algorithms have been proposed to improve the effectiveness of the selected features, such as gIndex[46], TreePi[51], FG-Index[5] and Tree+δ[52]. In gIndex, the authors propose a discriminative ratio for features. Only frequent and discriminative subgraphs are chosen as indexed features. In TreePi, due to the manipulation efficiency of trees, frequent and discriminate subtrees are chosen as feature set. The frequent subgraphs and edges are used as indexed features in FG-Index. In Tree+δ, the author use frequent free trees and a small number of discriminative subgraphs as indexed features.

The second sub-category is the path, vertex, and neighborhood substructures based filtering, in which no data mining based feature selection is necessary. There are several representative algorithms. In GraphGrep [38, 16], the authors propose to use all paths up to $maxL$ length as index features. Similarly, GraphGrep also builds inverted index for each path. In Closure-Tree [17], a pseudo subgraph isomorphism test is performed by checking the existence of a semi-perfect match from vertices in the query graph to vertices in a data graph (or graph closure). In TALE [40], an approximate matching method was proposed for large query graphs based on neighborhood units. In [18], the authors introduced a pattern matching method based on a combination of techniques: use of neighborhood subgraphs and profiles, joint reduction of the search space, and optimization of the search order. Since paths, vertices and neighborhood units are less discriminative than the frequent substructures, these algorithms may have less pruning power but better manipulation efficiency.

Most of these techniques only apply to a database of multiple small or medium sized graphs. These databases are large in the sense that they contain many graphs. Many of these methods care more about whether any database graph contains the query graph or not, instead of finding all the matches of the query graph in a given database graph.

Another category of research related to subgraph matching is graph alignment [28, 29, 39, 37, 3, 8]. Instead of matching subgraphs in a large database graph, these methods aimed to align a pair of biological graphs. In the problem studied in this paper, the size of the query graph may be much smaller than that of the database graph. Thus, the graph alignment method may not be directly applicable.

To apply the idea of frequent discriminate substructure based filtering, we need to create a new way to define "frequency" within one large graph. There is some related work in mining frequent subgraphs in a large graph [32, 50, 6, 13, 42]. However, none of them is designed for the acceleration of graph matching.

3. PRELIMINARIES

In this section, we introduce the fundamental definitions used in this paper and give the formal problem statement. We investigate the graph matching methods for undirected and unweighted labeled graphs. Without a loss of generality, it is easy to extend our methods to directed and weighted labeled graphs.

DEFINITION 1. *A* **labeled graph** *G is a five element tuple $G = \{V, E, \Sigma_V, \Sigma_E, L_G\}$ where V is a set of vertices and $E \subseteq V \times V$ is a set of edges. Σ_V and Σ_E are the sets of vertices and edge labels, respectively. The labeling function L_G defines the mappings $V \to \Sigma_V$ and $E \to \Sigma_E$.*

DEFINITION 2. *A labeled graph $G = (V, E, \Sigma_V, \Sigma_E, l)$ is* **isomorphic** *to another graph $G' = \{V', E', \Sigma'_V, \Sigma'_E, l'\}$, denoted by $G \approx G'$, iff there exists a bijection $f : V \to V'$ s.t.*

1. *$\forall\, u \in V$, $l(u) = l'(f(u))$,*
2. *$\forall\, u, v \in V, (u, v) \in E \Leftrightarrow (f(u), f(v)) \in E'$, and*
3. *$\forall\, (u, v) \in E$, $l(u, v) = l'(f(u), f(v))$.*

S is **subgraph isomorphic** *to G', denoted as $S \subseteq G'$, if S is isomorphic to at least one subgraph G'' of G'. G'' is a* **match** *of S in G'.*

An unlabeled graph substructure S is (sub)isomorphic to another graph G without considering labeling if G is regarded as an unlabeled graph and S is (sub)isomorphic to G.

DEFINITION 3. *Given vertices v_a and v_b in a connected graph G, we define the* **shortest distance** *between v_a and v_b, denoted as $d(G, v_a, v_b)$, as the number of edges on the shortest path between v_a and v_b.*

Since G is connected and undirected, we have $d(G, v_b, v_a) < |V|$ and $d(G, v_a, v_b) = d(G, v_b, v_a)$, where V is the vertex set of G.

In this paper, if the type of distance is not otherwise specified, shortest distance is assumed.

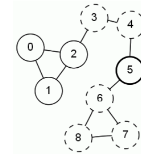


Figure 1: 2-neighborhood of vertex 5

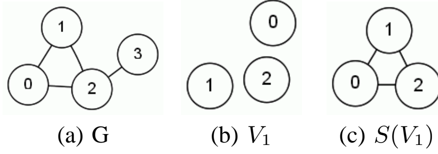
DEFINITION 4. *Given a vertex v in a graph G and an integer k, we define the* **k-neighborhood** *of v, denoted as $N_k(G, v)$, as a set of vertices in G such that*

$$\forall v' \in N_k(G, v), d(G, v', v) \leq k$$

As an example, in figure 1, the 2-neighborhood set of vertex 5 is set { 3, 4, 5, 6, 7, 8}.

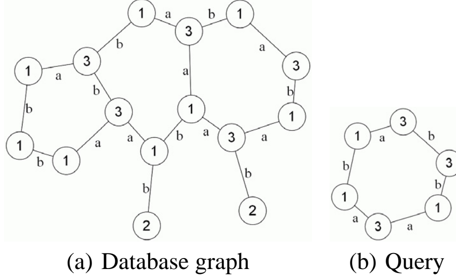
DEFINITION 5. *Given a subset V_1 of vertex set V of graph G, an* **induced subgraph** *of V_1 is the subgraph composed of V_1 and any edge whose endpoints both belong to V_1. We denote the induce subgraph of V_1 as $S(V_1)$.*

As an example, in figure 2, the induced subgraph of figure 2(b) is the graph in figure 2(c).



(a) G (b) V_1 (c) $S(V_1)$

Figure 2: Induced subgraph



(a) Database graph (b) Query

Figure 3: Database graph and query graph

Problem Statement: Given a large database graph G and a query graph Q, we aim to find all distinct matches of Q in G.

In figure 3, there is exactly one match of the query graph in the database graph. In this paper, we assume that the database graph is large, e.g., $|V| \geq 10^3$. The processing of graph queries in our paper can be divided into two major steps:

1. **Database Preprocessing.** In this step we construct the index from the database graph. We enumerate and select discriminative substructures in the database graph G. The set of discriminative substructures is denoted by $\mathrm{DS}(G)$. For a substructure $t_i \in \mathrm{DS}(G)$, we count the number of matches of t_i in the induced subgraph of intersecting neighborhood of two vertices. It is stored as the NDS distance between the pair of vertices. The NDS distances are used in the index structures.

2. **Query Processing.** In this step we match the query graph to the database graph. This step includes two repeated substeps: 1. **Vertex matching**. In this step we match a vertex in the database graph to one in the query graph. We match two vertices if they have the same label, meet the adjacency relationships, and satisfy the distance constraint with unmatched vertices. 2. **Pruning unqualified vertices by distance**. In this step, we prune the candidate vertices in the database graph by distance constraints introduced by the new matched vertex.

4. DATABASE PREPROCESSING

When the database of interests is composed of many small or medium sized graphs, GraphGrep [38, 16], GIndex [46] and TreePi [51] use substructures (paths, frequent subgraphs and trees respectively) to filter out graphs that do not match the query. There are two problems in directly applying these techniques in the new problem when the only database graph is of much larger size. First, the old concept of frequent substructure does not apply [32]. We need to develop a new definition of frequent substructures in a single large graph. Second, the final verification process of these algorithms may be slow because it might not be easy to directly apply the knowledge of substructure locations due to the difficulty of subgraph isomorphism tests.

In approximate large graph matching, TALE [40] used neighborhood units to improve performance. The neighborhood unit of a vertex consists of three elements: the degree of the vertex, the number of edges between the neighbors, and the labels of the actual neighbors. In this paper, we extend the definition of "neighborhood" in two ways. First, we extend the neighborhood unit from the set of adjacent vertices to the k-neighborhood set. Second, we extract more information from the intersection of the neighborhood units of a pair of vertices. Thus, we make the neighborhood definition a more powerful pruning tool. Moreover, by modifying the definition of a neighborhood, we are able to apply the concept of frequent substructures in this graph matching problem, which is widely adopted in subgraph search methods. We will provide more details in the remainder of this section.

In this section, we construct an index that will accelerate the query processing. First we introduce a property for the desired distance measurement. Then a new type of distance measurement between a pair of vertices is presented. Last we describe how to construct the index structure based on this distance.

4.1 The inequality property

There are many possible types of distance measurements between a pair of vertices. Only when the distance measurement meets the inequality property can we use it to reduce the search space during the query processing. Before we propose the distance measurement, we first introduce the inequality property.

If the query graph Q is subgraph isomorphic to the database graph G, for any vertices v_1, v_2, v_3, and v_4 in Q and their corresponding vertices v_1', v_2', v_3' and v_4' in G, we require a distance measurement $dist$, such that:

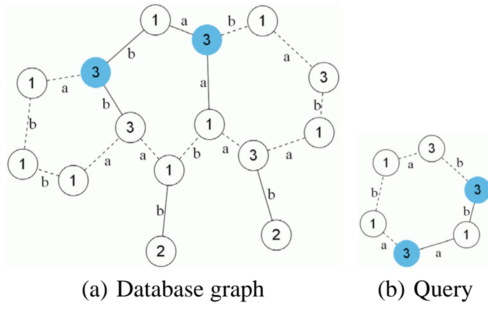
$$(dist(G, v_1', v_2') - dist(Q, v_1, v_2)) \times (dist(G, v_3', v_4') - dist(Q, v_3, v_4)) \geq 0$$

In other words, the difference between the distances between a pair of vertices in Q and its counterparts in G should either be always less than or equal to zero, or be always greater than or equal to zero.

Many types of distance measurements satisfy the inequality property, e.g., shortest distance, longest distance, second shortest distance, etc. The longest distance between any pair of vertices in the query graph, is always less than or equal to its counterparts in the database graph. In figure 4, the longest distance between two vertices (filled) is four in the query graph, while it is eleven in the database graph (the longest paths are marked by dashed lines).

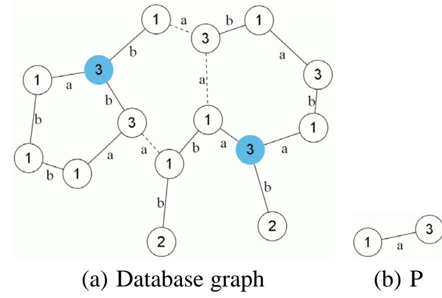
4.2 The NDS distance

In this subsection, we introduce a new distance measurement based on the frequent substructure count. This measurement satisfies the inequality property. We further explore the relationship between neighboring vertices in the database graph. We generate one subgraph from each pair of vertices and obtain a set of subgraphs. Thus, we can re-introduce the idea of frequent substructures to the new problem where the only database graph is of much larger size than the database graphs in previous methods. However, we need to restrict the size of this set of subgraphs.



(a) Database graph (b) Query

Figure 4: The definition of longest distance satisfies the inequality property



(a) Database graph (b) P

Figure 5: The definition of NDS distance satisfies the inequality property

To limit the number of subgraphs in the set, we introduce a parameter $Length$, which is the upper bound of the shortest distance between a pair of vertices to be indexed. Additionally, to reduce the average size of the subgraphs in the set, we select the intersection (instead of union) of the neighborhood sets. For a pair of vertices v_1 and v_2 within distance $Length$ in the database graph G and an integer k ($2 \times k > Length$), we generate an intersecting subgraph $Int(G, v_1, v_2)$ from v_1 and v_2 as follows.

- Generate the k-neighborhood set of v_1 and v_2, i.e., $N_k(G, v_1)$ and $N_k(G, v_2)$.
- Obtain the intersection of $N_k(G, v_1)$ and $N_k(G, v_2)$, $N_k(G, v_1) \cap N_k(G, v_2)$
- Obtain the *induced subgraph of the intersection set* as $Int(G, v_1, v_2)$, $Int(G, v_1, v_2) = S(N_k(G, v_1) \cap N_k(G, v_2))$

By the above process, we can generate a set of intersecting subgraphs from pairs of vertices within distance $Length$. When the database graph is very dense, i.e., high edge to vertex ratio, these intersecting subgraphs can be quite large, and hence may not be as useful as expected. However, the large database graph of interests is more likely to be generally sparse and locally dense (as noted in [14]). Thus, the intersecting subgraphs will be much smaller than the original database graph. In addition, the radius k can be chosen (as shown in the experimental results section) so that the number of intersecting subgraphs is too large.

With a set of intersecting subgraphs, we use frequent subgraph mining tools to find frequent substructures and select the discriminative ones. More details of how to select the discriminating substructure set $\mathrm{DS}(G)$ will be discussed in the next subsection. Let us assume that $\mathrm{DS}(G)$ is chosen for a substructure P in $\mathrm{DS}(G)$ and a pair of vertices v_1 and v_2. We define the neighboring discriminating substructure (NDS) distance of P between v_1 and v_2 as the number of matches of P in the intersecting subgraph $Int(G, v_1, v_2)$. We denote this distance as $d_{\mathrm{NDS}}(G, v_1, v_2, P)$. We should address here that in case of automorphisms, when more than one matches of P is located in the same set of database graph vertices, we count these matches separately as though they are located in different sets of vertices.

As an example, in figure 5, let us suppose that $Length = 4$, $k = 3$, the NDS distance between the two filled vertices is three, as there are three matches in their intersection subgraph. The matches of the discriminative substructure in the intersecting subgraphs are marked by dashed lines.

Next we show that the NDS distance satisfies the inequality property.

THEREOM 1. *The NDS distance satisfies the inequality property.*

PROOF. Let us suppose that there is a match of Q in G, assuming v_1 and v_2 are two vertices in Q and v_1' and v_2' are their counterparts in G.

We have $S(N_k(Q, v_1)) \subseteq S(N_k(G, v_1'))$ and $S(N_k(Q, v_2)) \subseteq S(N_k(G, v_2'))$, which leads to $Int(Q, v_1, v_2) \subseteq Int(G, v_1', v_2')$. For a discriminating substructure P, the number of matches of P in $Int(G, v_1', v_2')$ is always no less than that in $Int(Q, v_1, v_2)$. Thus, the NDS distance satisfies the inequality property. □

4.3 Selecting discriminating substructures

In the last subsection, we defined the NDS distance between two vertices. Now, we explain how to select the discriminative substructure set $\mathrm{DS}(G)$. A discriminating substructure is a small frequent substructure of intersecting subgraphs of neighboring vertices. Here we define a substructure to be frequent if it is subgraph isomorphic to no less than 50% of the intersecting subgraphs without considering labeling. We restrict the size of the substructure since the subgraph isomorphism test of large graphs is very expensive; we relax the labeling restrictions so that more frequent substructures will qualify as candidates.

Given a database graph G, and parameters $Length$ and k, we first generate a subset of around 100 intersecting subgraphs $\mathrm{IS}_r(Int(G))$ by randomly selecting pairs of vertices. Then, we use a frequent graph mining algorithm (e.g., [45]) to find unlabeled top 10 frequent substructures of either 3 or 4 edges in $\mathrm{IS}_r(Int(G))$. We record the number of matches of these frequent substructures in each intersecting subgraph. These frequent substructures distinguish the intersecting subgraphs by the number of matches, and are selected as a first set of discriminating substructures.

The large number of possible frequent substructures may significantly increase the computational and space complexity. Moreover, two substructures may carry good distinguishing information individually, but there is little gain if they are combined together. Due to the large number of possible substructures, it is hardly possible

to select a set of substructures without including any mutual correlation. Therefore, we select only the most important discriminative substructures leading to the large inter-class distance and small intra-class variance, i.e., the number of matches are far apart for intersecting subgraphs in the different clusters and are similar for intersecting subgraphs in one cluster. After selecting the initial set of discriminating subgraphs, we cluster the intersecting subgraphs into groups according to the number of matches. From the original set of frequent substructures, we select the most discriminative 3 substructures by a sequential forward selection procedure.

The sequential forward selection procedure is straightforward: first, we select the discriminating substructure with the best discriminating ability, say x_1, and then select the discriminating substructure which forms the best discriminating ability with x_1. The procedure continues until all the 3 substructures are selected from the 10 original ones. The number of combinations searched with this procedure is 27. Thus, the original 10 discriminating substructures are reduced to 3, as the final set of discriminating substructure, i.e. $\mathrm{DS}(G)$.

4.4 Constructing index

Having selected and refined the discriminative substructure set $\mathrm{DS}(G)$, for each P in $\mathrm{DS}(G)$, we calculate the NDS distance of P for every pair of neighboring vertices as our index structure. There are approximately $|V|deg^{Length}/2$ pairs of neighboring vertices, where deg is the average degree and V is the vertex set of the database graph G. To reduce the amount of calculation, we provide the following method.

Let us suppose that we have already calculated the NDS distance between vertices v_1 and v_2 for discriminative substructure P and parameter k. For another adjacent vertex of v_1, denoted as v_3, we deduce $d_{\mathrm{NDS}}(G, v_2, v_3, P)$ from $d_{\mathrm{NDS}}(G, v_1, v_2, P)$.

We can deduce that the NDS distance between v_2 and v_3 is $d_{\mathrm{NDS}}(G, v_1, v_2, P)$ plus the number of matches of P in $S((N_k(G, v_3) \cap N_k(G, v_2))$ which contain at least one vertex in $N_k(G, v_3) - N_k(G, v_1)$, and minus the number of matches of P in $S(N_k(G, v_1) \cap N_k(G, v_2))$ which contains at least one vertex in $N_k(G, v_1) - N_k(G, v_3)$. Formally, it is:

THEREOM 2.

$$M_{132} = \{m | P \approx m \subseteq S(N_k(G, v_3) \cap N_k(G, v_2)), \; m \cap (N_k(G, v_3) - N_k(G, v_1)) \neq \varnothing\}$$
$$M_{312} = \{m | P \approx m \subseteq S(N_k(G, v_1) \cap N_k(G, v_2)), \; m \cap (N_k(G, v_1) - N_k(G, v_3)) \neq \varnothing\}$$
$$d_{\mathrm{NDS}}(G, v_2, v_3, P) = d_{\mathrm{NDS}}(G, v_1, v_2, P) + |M_{132}| - |M_{312}|$$

PROOF. Let us suppose that the set of matches in $Int(G, v_2, v_3)$ is T_{23} and in $Int(G, v_1, v_2)$ is T_{12}. By definition, we have $T_{23} \cup M_{312} = T_{12} \cup M_{132}$. By set theory, $T_{23} \cap M_{312} = \varnothing$ and $T_{12} \cap M_{132} = \varnothing$. So we have $|T_{23}| + |M_{312}| = |T_{12}| + |M_{132}|$, and consequently $|T_{23}| = |T_{12}| + |M_{132}| - |M_{312}|$. Replacing the sets with NDS distances we have

$$d_{\mathrm{NDS}}(G, v_2, v_3, P) = d_{\mathrm{NDS}}(G, v_1, v_2, P) + |M_{132}| - |M_{312}|$$

□

By the shortest distance definition, the set of $N_k(G, v_3) - N_k(G, v_1)$ and $N_k(G, v_1) - N_k(G, v_3)$ can be retrieved as follows:

$$N_k(G, v_3) - N_k(G, v_1) = \{v | v \in V, d(G, v_1, v) = k+1, d(G, v_3, v) = k\}$$
$$N_k(G, v_1) - N_k(G, v_3) = \{v | v \in V, d(G, v_3, v) = k+1, d(G, v_1, v) = k\}$$

That is, any vertex v in $N_k(G, v_3)$ but not in $N_k(G, v_1)$ should satisfy the requirement that the shortest distance from v to v_1 is $k+1$ and the distance from v to v_3 is k, and vice versa.

To obtain $d_{\mathrm{NDS}}(G, v_3, v_1, P)$, we only need to search those matches containing vertices whose distances to v_1 and v_3 are exactly k and $k+1$. Considering that the discriminating substructures we use are of small size, we can improve the efficiency of the index construction by theorem 2.

To store the NDS distance between pairs of neighboring vertices, we use one array for each vertex of a discriminating substructure P. In the array, we store the NDS distance from the vertex to each of its neighboring vertices, which are sorted by the vertex indices. The space requirement for the index structure is $|\mathrm{DS}(G)||V|deg^{Length}$.

5. MATCHING ALGORITHM

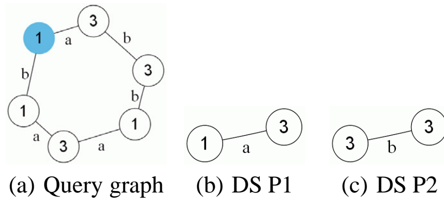
In this section, we introduce the subgraph matching algorithm. We start with a depth first matching algorithm, and then we provide a dynamic matching method to improve overall efficiency.

5.1 Depth first matching algorithm

Given a query graph Q, we need to find all matches of Q in the database graph G. In this subsection, we introduce the matching algorithm by depth first search. The algorithm is based on the following two observations.

1. To match a database graph vertex v_g to a query graph vertex v_q, we will need to check the neighboring vertices of v_g in G and those of v_q in Q. For each neighboring vertex v of v_q, we want to find at least one neighboring vertex v' of v_g, such that v and v' have the same label, and the NDS distance for any discriminating substructure between v' and v_g is greater than or equal to that between v and v_q while the shortest distance between v' and v_g is less than or equal to that between v and v_q. Formally, given v_g in G and v_q in Q, $\forall v \in Q$, $d(Q, v_q, v) \leq Length$, we are to find $v' \in G, d(G, v_g, v') \leq Length$, such that (1) $L(v) = L(v')$, (2) $d(Q, v_q, v) \geq d(G, v_g, v')$, and (3) $d_{\mathrm{NDS}}(Q, v_q, v) \leq d_{\mathrm{NDS}}(G, v_g, v')$. Based on the inequality property, if such a neighboring vertex (v') does not exist in the database graph, then we cannot match v_g to v_q.

2. Having matched a database graph vertex v_g to a query graph vertex v_q, we can prune those vertices in the database graph that cannot appear in the current match based on the matching between v_g and v_q. For any neighboring vertex v of v_g, we need to find at least one vertex v' in Q, such that v and v' are of the same label, and the NDS distance for any discriminative substructure is less than or equal to that between v and v_g while the shortest distance is greater than or equal to that between v and v_g. Formally, given v_g in G and v_q in Q, $\forall v \in G$, $d(G, v_g, v) \leq Length$, we are to find $v' \in Q$, such that (1) $L(v) = L(v')$, (2) $d(Q, v_q, v') \geq d(G, v_g, v)$, and (3) $d_{\mathrm{NDS}}(Q, v_q, v') \leq d_{\mathrm{NDS}}(G, v_g, v)$. (Remember that when $d(Q, v_q, v') > Length$, $d_{\mathrm{NDS}}(G, v_g, v) = 0$.) Based on the inequality property, if such a vertex (v') does not exist



(a) Query graph (b) DS P1 (c) DS P2

Figure 6: Query graphs and the discriminating substructure

in the query graph, then we cannot match the database graph vertex v to any vertex in the query graph, and we remove v from the candidate set. After removing all such database graph vertices, we also remove any vertex unconnected to the matched vertex.

Although these two observations look similar, they are not the same. In observation one, we fix the query graph vertex and then search for database graph vertices, so that we can find a potential match for the query graph vertex. In observation two, we fix the database graph vertex and search for query graph vertices, so that we can decide whether the database graph vertex can be removed. This twoway pruning strategy is highly effective. It is also very efficient because the shortest distance and the NDS distance can be retrieved quickly in the database graph.

Let us suppose that we have the two discriminative substructures shown in figure 6(b) and figure 6(c) and $Length = 3$, $k = 2$. To match a database graph vertex in figure 3(a) to the filled vertex in figure 6(a), we check the neighboring vertices (within distance $Length$) of each vertex in figure 3(a) with label "1". In the query graph, there are three vertices with the label "3" of the filled vertex. The NDS distance of the discriminating substructure P1 between any of the filled vertices to the filled vertex is 1, and the shortest distances are 1, 2, and 2 respectively. By applying this information, we can significantly reduce the set of candidate database vertices for the filled query graph vertex from the whole set of labeled "1" vertices to a small vertex set. If we use the discriminative substructure in figure 6(b), we can reduce the candidate set to the set of filled vertices in figure 7. On the other hand, if we use the discriminative substructure in figure 6(c), we can reduce the candidate set to two vertices (the rightmost filled vertex in figure 7 would be removed).

In the same example, having matched the filled database graph vertex in figure 8 to the filled query graph vertex in figure 6(a), we remove those database graph vertices which are impossible to be matched based on the information from the new matched vertex. The greatest shortest distance from the filled vertex to any labeled "1" vertex is 3 in the query graph, when the NDS distance of both P1 and P2 should be at least 1. Similarly, the greatest shortest distance from the filled vertex to any vertex labeled "3" is 2 in the query graph, when the NDS distance of either P1 or P2 should be at least 1. Thus, we remove all unqualified vertices in the database graph and as well as the unconnected ones. The remaining database graph is shown in figure 8.

From these two observations, we introduce a new index based subgraph matching algorithm. To find a match for the query graph in the database graph, we match one pair of vertices a time. The database graph vertices are chosen in a depth first fashion. Without loss of generality, we assign a unique vertex ID to each query graph vertex, $v_{q1}, ..., v_{qn}$, where n is the number of vertices in

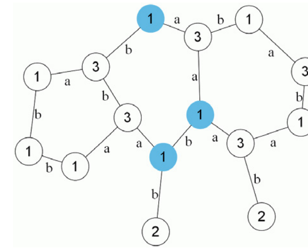


Figure 7: Reduced candidate vertex set (filled)

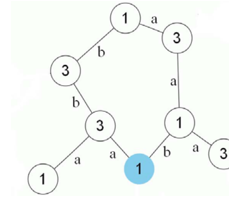


Figure 8: The pruned database graph (pruned by the filled vertex)

the query graph and for any i, $1 \leq i \leq n$, the induced subgraph $S(v_{q1}, ..., v_{qi})$ is connected. The query graph vertices are matched in the order of their vertex ID.

At the beginning of any match, the candidate set (the set of database graph vertices which can appear in the match) is the whole vertex set of the database graph. To match a database graph vertex to a query graph vertex, we check: (1) if they have the same label; (2) whether the database graph vertex is adjacent to the corresponding matched vertices based on the structure of the query graph; and (3) if the neighboring vertices satisfy the inequality property (by observation one). If all these three requirements are satisfied, we match the database graph vertex to the query graph vertex. After matching the vertex, we remove those vertices that cannot appear in the current match from the database graph based on the information introduced by the new matched vertex. We also remove any vertices that are unconnected to the matched vertex.

When the algorithm finishes searching all database graph vertices in the candidate set from match a query graph vertex v_{qi}, we delete the last matched vertex from the current match, recover the removed vertices to the candidate set, and match another database graph vertex to the query graph vertex v_{qi-1}. If all the query graph vertices are matched successfully, we output the current match and start another match with a different database graph vertex. If such a database graph vertex does not exist, the algorithm ends since all matches have been found.

5.2 Dynamic matching

To find multiple matches for the query graph, we use a depth first algorithm to find all possible matches. We can accelerate the algorithm by avoiding redundant calculations. In observation one, in order to match a database graph vertex to a query graph vertex, we check their neighboring vertices. This procedure is complicated but can be reduced. After a database graph vertex is matched to a query graph vertex in one match, we record the vertex-vertex matching relationship. To find another match of the query graph, if the same vertex pair appears again, we do not calculate the neighboring vertices again since that has already been cached.

Algorithm 1 *GADDI*

Input: database graph G, query graph Q
Output: matches of Q in G

```
1: match M ← ∅, candidate set C ← V
2: DynamicMatching(0, M, C)
```

Subprocedure name: $DynamicMatching(i, M, C)$
Input: No. of matched vertices i, current match M, candidate set C
Output: final matches of Q in G

```
1: if i = n then
2:    output M
3:    RETURN
4: end if
5: for each vertex v in C do
6:    if v and v_{qi+1} have the same label and adjacency relationships then
7:       if v_{qi+1} is in the false list of v then
8:          Continue
9:       end if
10:      if v_{qi+1} is in the true list of v then
11:         shrink C by observation two
12:         add v to M,
13:         DynamicMatching(i + 1, M, C)
14:         Continue
15:      end if
16:      if the neighboring vertices of v satisfy inequality property then
17:         add v_{qi+1} to the true list of v
18:         shrink C by observation two
19:         add v to M,
20:         DynamicMatching(i + 1, M, C)
21:      else
22:         add v_{qi+1} to the false list of v
23:      end if
24:   end if
25: end for
```

In the beginning of the algorithm, we create two empty lists for each database graph vertex. The first list (the true list) represents the vertices which the database graph vertex can be matched to, the other list (the false list) represents the vertices which the database graph vertex cannot be matched to. For any database graph vertex v_g, when we match it to a query graph vertex v_{qi} for the first time, if v_g and v_{qi} have the same vertex label, and v_g is adjacent to a set of vertices that match to the neighboring vertices of v_{qi} in the query graph, we check whether the neighboring vertices of v_g satisfy the inequality property (based on observation one). If they do, then we add v_{qi} to the true list of v_g; otherwise we add v_{qi} to the false list. Another match may attempt to match v_g with v_{qi} again. Then, we only need to check the true and false lists, instead of checking the neighboring vertices again.

The true and false lists are only modified when necessary. At all times, the intersection of the true and false lists of any database graph vertex is the empty set. For example, assuming the vertex ID of filled vertex in figure 6(a) is 0 and it is the first vertex to be matched in the algorithm. After we try to match this query graph vertex, the true list of the three filled vertices and the false list of all other vertices in figure 7 is changed to $\{0\}$ from the empty set.

After using this dynamic matching techniques, we show our algorithm in algorithm 1.

5.3 Parameter Setting

In the GADDI algorithm, there are two important parameters, $Length$ (upper bound of the distance between a pair of indexed vertices) and k (radius) mentioned above. The range of $Length$ is from 1 to $2k$ and can be set adaptively. First $Length$ is set to 1 and it can gradually increase. Let us assume that we are matching v_q in Q to nodes in G, if the pruning power is too low (i.e., too many[1] possible matches), we increase $Length$ and redo the process for the vertex v_q with higher $Length$. Otherwise, we will try to match another vertex in Q. Parameter k is more difficult to set. It is highly dependent on the characteristics of the database graph, e.g., the average degree of a vertex.

As the radius k increases, the number of vertices in the intersection area also increases. The increasing rate of construction time depends on the average degree deg of the graph. Usually the number of vertices in a circle with radius k is deg^k, which means the number of vertices in either a single circle or the intersection of two circles increases exponentially with k and polynomially with deg. Thus when deg is large, even a small increase of k increases the index construction time dramatically.

However, the pruning power does not linearly increase with k. When k is small, the number of vertices in the intersection of two circles in the query graph tends to increase as k increases. But when k becomes larger, most vertices in query graph are already covered by the intersection area, and the number of vertices in the intersection area does not change much. However, for the database graph, due to its large size, the number of vertices in the intersection area increases significantly with deg and k. The increasing potential of the NDS distance in the database graph is much higher than it is in the query graph. Thus, when k is low, the pruning power of the NDS distance usually increases with k. On the other hand, when k is high, the pruning power does not usually improve with larger k. Therefore, k is fixed at 2.

6. EXPERIMENTAL RESULTS

In this section, we empirically analyze the performance of GADDI against TALE [40], one of the most recent subgraph matching tools that designed for large graphs. TALE is very efficient and effective in index construction and matching, which is quite important to subgraph matching in graphs of large size. We also compare GADDI with another graph indexing tool GraphGrep [16], for its widely recognized performance. GADDI and TALE are implemented with C++ code and we obtained GraphGrep from the authors of [16]. They were all run on a Dell PowerEdge 2950, with two 3.0 GHZ dual-core CPUs and 16 GB main memory, using Linux 2.6.16.21-0.8-smp.

First, we compare the performance of GADDI with the other algorithms on two real data sets, a protein interaction network and a social network. Then, a large number of synthetic data sets are employed to show the efficiency and scalability of these three methods. Without loss of generality, we select three discriminating substructures for all experiments, which are triangles, quadrilaterals, and stars of size 3.

6.1 Real Data Sets

In this set of experiments, two graphs generated from real data sets are utilized. The first graph is generated from a subset of the protein-protein interaction network for homo sapiens. There are 6410 vertices, 53844 edges, and the average degree of a vertex is 8.4. Each vertex represents a protein and the label of the vertex is its gene ontology term from [54]. There are a total of 632 distinct

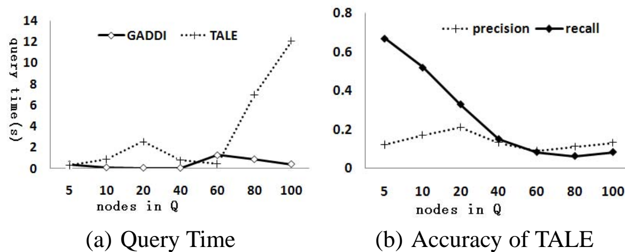
[1]This threshold is application dependant and can be set by domain experts. In this paper, we set threshold value to 20.

labels. An edge in the graph represents an interaction between the two proteins it connects. Although our major application domain is biological networks, we also compare GADDI with other methods in a social network graph, where the average degree is much higher. The social network graph is obtained from a social network in [55]. There are 297 vertices, 4158 edges, and the average degree of a vertex is 14. Each vertex represents a person (with a unique vertex label), and an edge corresponds to communication between two persons.

For the protein interaction network, we vary the number of vertices and edges in the query graph. GADDI spends about 35 minutes to construct an index of 100MB and it can find all matches of a pattern correctly with efficient execution time and memory usage. This means that the precision and recall of GADDI is 100%. On the other hand, GraphGrep enumerates all possible paths up to the length $MaxL$ in the database graphs to construct the index. The index space is $O(Ndeg^{MaxL})$ where N and deg are the number of vertices and the average degree of a vertex, respectively. We have experimented with various value for $MaxL$. We found that when $MaxL$ is small, e.g., less than 4, GraphGrep does not perform very effectively. This may be due to the fact that when the paths are too short, one path in the query graph may have too many matches in the database graph. Thus, we use the default setting of $MaxL = 4$ in the GraphGrep program for all the experiments.

It is expected that when the number of edges increases in the database graph, the size of index will also grow dramatically especially when $MaxL$ is large. Thus, GraphGrep consumes over 20 gigabytes of memory during the construction of the index for the protein interaction graph and the program crashed. Even had it not crashed, it would have generated a very large indexing structure, which would not have fit in main memory. Thus, it would have to be loaded into main memory whenever it was needed. This would increase the query time dramatically (with disk accesses).

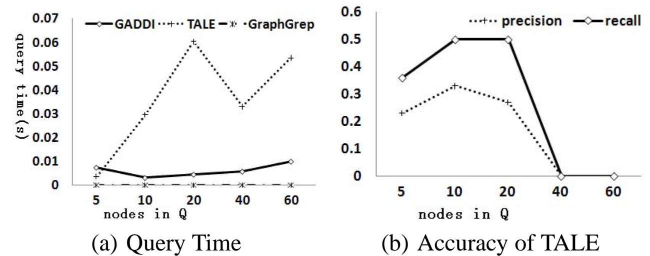
For TALE, on the other hand, since it is an approximate algorithm, it may not correctly find all matches of a pattern. We therefore show both the execution time of GADDI and TALE and the precision and recall [1] of TALE in Figure 9. The precision and recall of TALE are much lower than GADDI, particularly on large graphs. We found another very interesting phenomenon: the query execution time of GADDI is usually less than that of TALE when the query graph size is not very small, e.g., larger than 5 vertices. This is due to the fact that GADDI uses an indexing structure with more information about the database graph. During the query time, the information captured in the indexing structure can be utilized more efficiently, which leads to a faster response time.



(a) Query Time (b) Accuracy of TALE

Figure 9: Protein Interaction

For our social network graph, we also vary the number of vertices and edges in the query graph. The execution time of GADDI and TALE and the precision and recall of TALE are shown in Figure 10. GADDI is able to build the index structure within 300 ms, with the size of less than 1MB. The average execution time is less than 10 ms. GraphGrep spends almost 4 minutes in order to build its index structure, which is 1.2GB in size. However, it finishes the actual execution in under 1 millisecond. With the lowest precision and recall, TALE spent the least 3ms. The TALE index structure is also very small, e.g., 1 MB. However, its query time is the longest of the three when the size of the query graph is not tiny, e.g., 5 vertices or larger. Once again, this result is due to the fact that GraphGrep keeps the most information in its index while TALE keeps the least.



(a) Query Time (b) Accuracy of TALE

Figure 10: Social Network

Since we only need to build an index structure for each database graph once, query time is much more important than index building time. From the perspective of precision and recall, GADDI performs better than TALE in both of the real data sets. GraphGrep has equal precision to GADDI, and we note that the query time of GraphGrep on the social network graph is shorter than that of GADDI. However, GraphGrep generates a much larger index structure than GADDI does. When the database graph is large, the indexing structure of GraphGrep does not fit in the main memory leading to the dynamic loading of indexing pages and in turn significantly increasing the query execution time.

6.2 Synthetic Data Sets

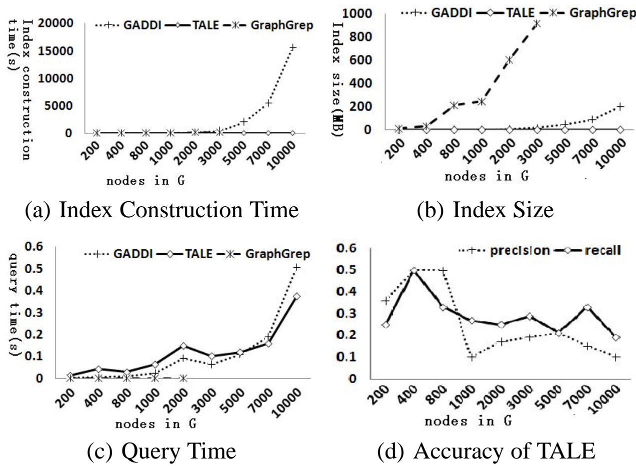
We analyze the performance of the three methods by separately varying each of six parameters in experiments run on a set of synthetically generated graphs. Since GADDI finds all the matches of the Q in G, its precision and recall are all 100 percents. So we only show TALE's accuracy in the following experiments. To systematically analyze the performance of GADDI, we vary one parameter at a time. GraphGrep can only be applied to a small database graph G, and thus is only involved in part of tests. For the remainder of experiments, GraphGrep crashes due to the extremely high space utilization. The default parameter values are listed in Table 1.

Table 1: Default Parameter Value

Parameter	Default Value
Number of vertices in G	5000
Average Degree in G	8
Number of vertices in Q	20
Average Degree in Q	4
Number of Labels	250

The first parameter is the number of vertices in G. We test GADDI and TALE, 200 to 10,000 vertices and GraphGrep from 200 to 3,000 vertices(Grapgrep crashes when generating index on graphs

with larger number of vertices). GraphGrep still spends the most amount of time in generating the largest indexing structure, but spends the least amount of time in querying. The query time of GADDI is less than that of TALE when the number of vertices in G is less than or equal to 5,000. On the other hand, the precision and recall of TALE degrade dramatically when the number of vertices in G increases. When the number of vertices in G is 10,000, GADDI takes 7,828 seconds to build a 200 MB index. This is much less than GraphGrep and is affordable. GADDI needs to calculate the shortest distance and NDS distance between any pair of vertices in G, a calculation with a time complexity of $O(n^3)$ and space complexity of $O(n^2)$. So the index construction time and size increases polynomially (with a low power) with number of vertices in G. Figure 11(a) and (b) show the index construction time and size of all three methods. Figure 11(c) and (d) show the query time of all three methods, and the precision and recall of TALE.



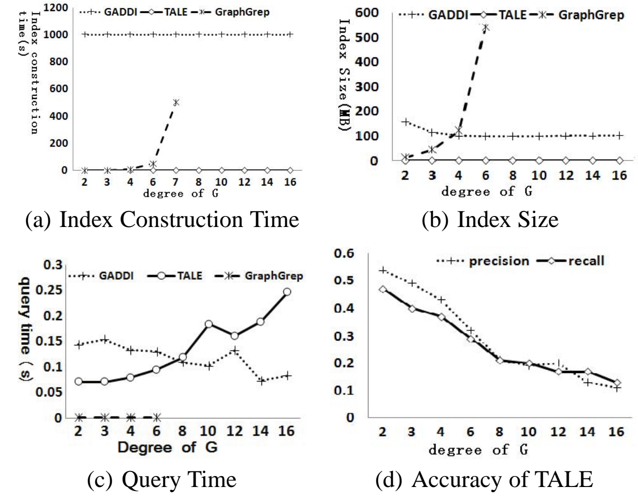
(a) Index Construction Time (b) Index Size

(c) Query Time (d) Accuracy of TALE

Figure 11: Number of Vertices in G

The second parameter we vary is the average degree of a vertex in G. GraphGrep is tested on G with average degrees of 2, 3, 4, 6 and 7(GraphGrep crashes when the average degree is greater than 7). Figure 12 (a) and (b) show the index construction time and size for all three methods. It is evident that TALE spends the least amount of time in constructing the smallest index. On the other hand, when the average vertex degree is low, the number of paths in G is also small. Thus, GraphGrep spends a small amount of time in building the relatively small indexes. However, with the increase of the average vertex degree, the number of paths increase significantly, Grephgrep builds much larger indexing structure and crashes when the degree reaches 8. The GADDI index construction time contains two parts, the shortest path and the NDS calculations. The shortest path calculation time depends on the number of vertices in G while the NDS calculation time does not. However, the NDS calculation time is much less than the shortest path computation time. Thus the GADDI index construction time primarily depends on the number of vertices in G.

Figure 12 (c) and (d) show the query time of all three methods, and the precision and recall of TALE, respectively. In general, the query time of GADDI is less affected by the average vertex degree in G. On the other hand, the query time of TALE increases with the average vertex degree because it needs to match more vertices and edges. In addition, the accuracy of the query results from TALE degrades significantly with the average vertex degree due to the fact that the probability of a mismatch is greater.

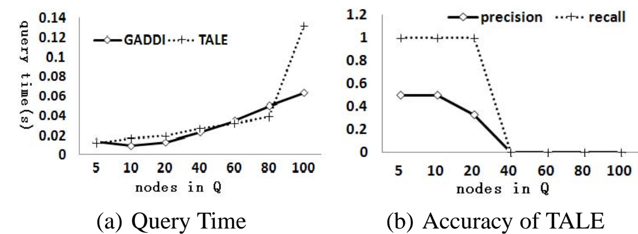


(a) Index Construction Time (b) Index Size

(c) Query Time (d) Accuracy of TALE

Figure 12: Average Degree of a Vertex in G

The third parameter we vary is the number of vertices in the query graph Q. We only run GADDI and TALE, which are shown in Figure 13. (GraphGrep crashed during this sets of experiments.) With more vertices in Q, more vertices and edges need to be compared in the query process, so the query times of both GADDI and TALE increases. GADDI has a faster or similar query time as TALE for varying number of vertices in Q. Since TALE is an approximation match algorithm, the probability of it finding wrong matches increases with the number of vertices in Q. So the accuracy of TALE degrades when number of vertices in Q increases.

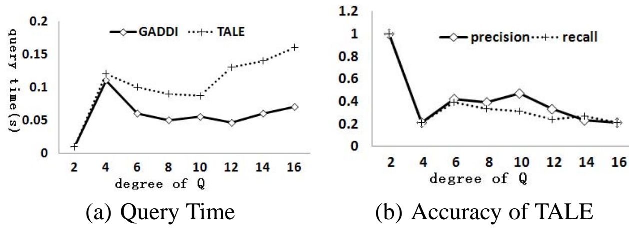


(a) Query Time (b) Accuracy of TALE

Figure 13: Number of Vertices in Q

The forth parameter is the average degree of a vertex in Q. The query time of GADDI and TALE, and the precision and recall of TALE are shown in Figure 14. It is obvious that the higher the vertex average degree Q has, the more information that Q possesses for pruning vertices in G. However, a high vertex degree also requires more edges to be compared when querying. When the average degree of Q is 2, there are few edges to be compared and both GADDI and TALE perform very quickly. When the average degree goes from 4 to 8, the pruning power of both methods increases and more vertices in G are pruned during query. However, when the vertex degree of Q increases from 10 to 16, the additional number of edges that need to be compared plays a key role and query time increases. GADDI outperforms TALE with a wide margin on various average degree values.

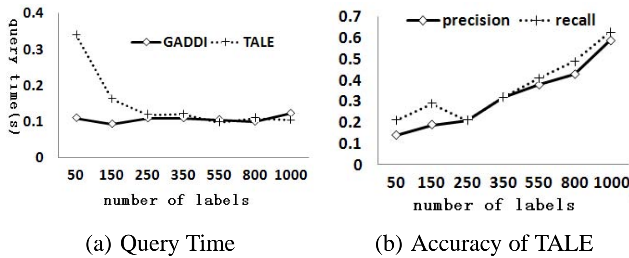
The fifth parameter is the number of distinct labels. From Figure 15, we can see that the precision and recall of TALE increases



(a) Query Time (b) Accuracy of TALE

Figure 14: Average Degree of a Vertex in Q

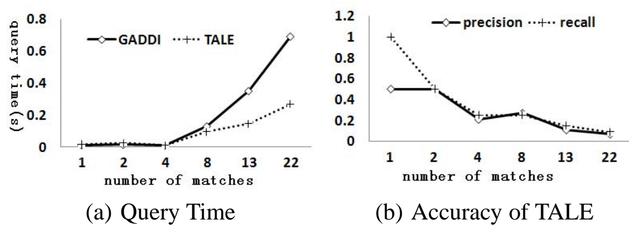
with the number of distinct labels. More labels in G increases the pruning power of both GADDI and TALE. Increasing the number of distinct labels reduces the number of candidate matches between any pair of vertices in G and Q, which leads to a shorter execution time.



(a) Query Time (b) Accuracy of TALE

Figure 15: Distinct types of labels

The sixth parameter we vary is the number of matches of Q in G. Figure 16 the query time of TALE and GADDI, and the precision and recall of TALE. With more matches in G, GADDI requires more time to find all the matches. Since TALE only finds some of the matches, the number of matches does not affect the query time of TALE as much as it does for GADDI. On the other hand, the accuracy of TALE degrades since the number of matches TALE finds does not increase with the number of true matches.



(a) Query Time (b) Accuracy of TALE

Figure 16: Number of matches of Q in G

This concludes the results of the experiments we ran on these three methods using various real and synthetic data sets. GraphGrep stores a larger amount of information about the database graph in its index than TALE and GADDI do. Thus, the index size of GraphGrep increases more quickly with the database graph size (e.g., the number of vertices and average degree of a vertex) while TALE and GADDI maintain a relatively small indexing structure. When the graph is very large, GraphGrep often fails to build the index structure due to running out of space. On the other hand, the query time of GraphGrep is usually shorter when its index can be built. The main difference between TALE and GADDI is in accuracy. TALE is an approximate method which does not find all matches of a pattern correctly while GADDI has no such problem. Thus, if the database graph is relatively small (e.g., a couple thousand vertices and less than ten thousand edges) and index size is not a big concern, then GraphGrep is preferable over the other two methods. On the other hand, if the database graph is very large, e.g., several thousand vertices and tens of thousand edges or more, then GADDI should be deployed.

7. CONCLUSION

In this paper, we have proposed an indexed based graph matching method (GADDI) to find all the matches of any query graph in a single large database graph. GADDI indexes the NDS distance between pairs of neighboring vertices. The NDS distance satisfies the proposed inequality property which is the basis of our pruning method. GADDI achieves high pruning power and its size scales linearly with the number of neighboring vertex pairs. We introduce an innovative graph matching method, which applies a twoway pruning and incorporates a dynamic matching scheme.

With a large set of real and synthetic data sets, we demonstrate that the GADDI approach can outperform the alternative methods both in efficiency and accuracy. Overall, GADDI is a very useful tool of subgraph matching in biological networks.

8. ACKNOWLEDGEMENT

We sincerely thank the authors of GraphGrep for providing their implementation of GraphGrep. This project was partially supported by the grant of NSF0551603 and a Case PRI award.

9. REFERENCES

[1] R. Baeza-Yates and B. Ribeiro-Neto. Modern Information Retrieval. New York: ACM Press, Addison-Wesley, 1999.

[2] B. Berendt, A. Hotho, and G. Stumme. Towards semantic web mining. *Proc. of* ISWC, 2002.

[3] J. Berg, and M. Lassig, Local graph alignment and motif search in biological networks, PNAS, 2004.

[4] B. Bringmann, and S. Nijssen: What Is Frequent in a Single Graph? *PAKDD* 2008.

[5] J. Cheng, Y. Ke, W. Ng, and A. Lu. FG-Index: Towards verification-free query processing on graph databases. *Proc. of SIGMOD*, 2007.

[6] D. Cook, L. Holder, and S. Djoko. Knowledge discovery from structural data. Journal of Intelligent Information Systems, 1995.

[7] L. Cordella, P. Foggia, C. Sansone, and M. Vento. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. PAMI, 2004.

[8] T. Dandekar, S. Schuster, B. Snel, M. Huynen, and P. Bork, Pathway alignment : application to the comparative analysis of glycolytic enzymes, Biochem, 1999.

[9] L. Dehaspe, H. Toivonen, and R. King. Finding frequent substructures in chemical compounds. *Proc. of KDD*, 1998.

[10] M. Deshpande, M. Kuramochi, G. Karypis. Frequent sub-structure based approaches for classifying chemical compounds. *Proc. of ICDE*, 2003.

[11] B. Dost, T. Shlomi, N. Gupta, E. Ruppin, V. Bafna, and R. Sharan. QNet: A Tool for Querying Protein Interaction Networks, *Proc. of RECOMB*, 2007.

[12] M. Fiedler, and C. Borgelt: Subgraph Support in a Single Large Graph. *Proc. of ICDM Workshops*, 2007.

[13] S. Ghazizadeh, and S. Chawathe. SEuS: Structure extraction using summaries. *Proc. of ICDS*, 2002.

[14] D. Gibson, R. Kumar, and A. Tomkins. Discovering Large Dense Subgraphs in Massive Graphs, *Proc. of VLDB,* 2005.

[15] J. Gonzalez, L. Holder, and D. Cook. Application of graph-based concept learning to the predictive toxicology domain. Proc. of the Predictive Toxicology Challenge Workshop, 2001.

[16] R. Giugno, D. Shasha, GraphGrep:, A Fast and Universal Method for Querying Graphs. *Proc. of ICPR*, 2002.

[17] H. He and A. K. Singh. Closure-Tree: an index structure for graph queries. *Proc. of ICDE*, 2006.

[18] H. He and A. K. Singh. Graphs-at-a-time: Query Language and Access Methods for Graph Databases, *Proc. of SIGMOD*, 2008.

[19] J. Huan, W. Wang, J. Prins, and J. Yang. SPIN: mining maximal frequent subgraphs from graph databases. *Proc. of KDD,* 2004.

[20] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. *Proc. of ICDM,* 2003.

[21] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. *Proc. of PDKK,* 2000.

[22] D. Jensen, and H. Goldberg. Artificial Intelligence and Link Analysis. AAAI Press, 1998.

[23] H. Jiang, H. Wang, P. Yu, and S. Zhou. Gstring: A novel approach for efficient search in graph databases. *Proc. of ICDE*, 2007.

[24] Y. Ke, J. Cheng, and W. Ng. Correlation search in graph databases. *Proc. of SIGKDD*, 2007.

[25] J. Kleinberg, R. Kumar, P. Raghavan,S. Rajagopalan, and A. Tomkins. The Web as a graph: Measurements, models and methods. Lecture Notes in Computer Science, 1999.

[26] C. Ko. 2000. Logic induction of valid behavior specifications for intrusion detection. In IEEE Symposium on Security and Privacy, 2000.

[27] M. Koyuturk, A. Grama, and W.Szpankowski. An efficient algorithm for detecting frequent subgraphs in biological networks. *Bioinformatics,* 2004.

[28] M. Koyuturk, A. Grama, and W. Szpankowski. Pairwise Local Alignment of Protein Interaction Networks Guided by Models of Evolution. RECOMB 2005.

[29] M. Koyuturk, Y. Kim, U. Topkara, S. Subramaniam, W. Szpankowski, and A. Grama. Pairwise Alignment of Protein Interaction Networks, RECOMB 2005 .

[30] S. Kramer, L. De Raedt, and C. Helma. Molecular feature mining in HIV data. *Proc. of KDD*, 2001.

[31] M. Kuramochi, and G. Karypis. Frequent subgraph discovery. *Proc. of ICDE,* 2001.

[32] M. Kuramochi, G. Karypis, Finding Frequent Patterns in a Large Sparse Graph. *DMKD,* 2005.

[33] W. Lee, and S. Stolfo. A framework for constructing features and models for intrusion detection systems. ACM Transactions on Information and System Security, 2000.

[34] R. Mooney, P. Melville, L. Tang, J. Shavlik, I. Castro Dutra, and D. Page. Relational data mining with inductive logic programming for link discovery. AAAI Press/The MIT Press, 2004.

[35] S. Nijssen and J. Kok. A quick start in frequent structure mining can make a difference *Proc. of KDD,* 2004.

[36] C. Palmer, P. Gibbons, and C. Faloutsos. ANF: A fast and scalable tool for data mining in massive graphs. *Proc. of KDD*, 2002.

[37] R. Pinter, O. Rokhlenko, E. Yeger-Lotem and M. Ziv-Ukelson, Alignment of metabolic pathways, Bioinformatics, 2005.

[38] D. Shasha, J. Wang, and R. Giugno. Algorithmic and applications of tree and graph searching. PODS, 2002.

[39] R. Singh, J. Xu, and B. Berger. Pairwise Global Alignment of Protein Interaction Networks by Matching Neighborhood Topology, RECOMB 07'.

[40] Y. Tian and J. Patel. TALE: A Tool for Approximate Large Graph Matching, *Proc. of ICDE*, 2008.

[41] J. Ullmann. An algorithm for subgraph isomorphism. J. ACM, 1976.

[42] N. Vanetik, E. Gudes, and S. Shimony. Computing frequent graph patterns from semistructured data. *Proc. of ICDM*, 2002.

[43] S. Wasserman, K. Faust, and D. Iacobucci. Social network analysis : Methods and applications. Cambridge University Press, 1994.

[44] D. Williams, J. Huan, and W. Wang. Graph database indexing using structured graph decomposition. *Proc. of ICDE*, 2007.

[45] X. Yan and J. Han. gSpan: graph-based substructure pattern mining, *Proc. of ICDM,* 2002.

[46] X. Yan, P. Yu, and J. Han. Graph indexing, a frequent structure-based approach. *Proc. of Sigmod*, 2004.

[47] X. Yan, P. Yu, and J. Han. Substructure similarity search in graph databases. *Proc. of Sigmod*, 2005.

[48] X. Yan, F. Zhu, J. Han, and P. Yu. Searching substructures with superimposed distance. *Proc. of ICDE*, 2006.

[49] K. Yoshida, and H. Motoda. CLIP: Concept learning from inference patterns. Artificial Intelligence, 1995.

[50] K. Yoshida, H. Motoda, and N. Indurkhya, Graph-based induction as a unified learning framework. Journal of Applied Intelligence, 1994.

[51] S. Zhang, M. Hu, and J. Yang. Treepi: a new graph indexing method. *Proc. of ICDE,* 2007.

[52] P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: tree + delta <= graph. *Proc. of VLDB*, 2007.

[53] Lei Zou, Lei Chen, J. Yu, Y. Lu, A Novel Spectral Coding in a Large Graph Database, *Proc. of EDBT*, 2008.

[54] Gene Ontology. http://www.geneontology.org/.

[55] Social Network.
http://www-personal.umich.edu/ mejn/netdata/.