SAPPER: Subgraph Indexing and Approximate Matching in Large Graphs

Shijie Zhang, Jiong Yang, Wei Jin
EECS Dept., Case Western Reserve University,
{shijie.zhang, jiong.yang, wei.jin}@case.edu

ABSTRACT

With the emergence of new applications, e.g., computational biology, new software engineering techniques, social networks, etc., more data is in the form of graphs. Locating occurrences of a query graph in a large database graph is an important research topic. Due to the existence of noise (e.g., missing edges) in the large database graph, we investigate the problem of approximate subgraph indexing, i.e., finding the occurrences of a query graph in a large database graph with (possible) missing edges. The SAPPER method is proposed to solve this problem. Utilizing the hybrid neighborhood unit structures in the index, SAPPER takes advantage of pre-generated random spanning trees and a carefully designed graph enumeration order. Real and synthetic data sets are employed to demonstrate the efficiency and scalability of our approximate subgraph indexing method.

1. INTRODUCTION

Graph data has appeared in many recent applications, ranging from bioinformatics, software engineering to social networks. Managing, processing, and analyzing these graph data becomes an urgent practical problem. Subgraph query is one of the most fundamental procedures in managing graphs. In many applications, e.g., biological networks, graphs are large with thousands or tens of thousands of vertices and millions of edges. A subgraph query is to identify the occurrences of the query subgraph in the database graph.

Although subgraph query has been studied previously [24], the basic assumption is that the networks of interest are perfectly clean. In order to qualify an occurrence of a query subgraph q, all edges of the query graph have to occur in the database graph G. In other words, the occurrence has to be exact. However, noise commonly exists in many applications or the approximate matches themselves are more interesting. For example:

1. A challenging problem in the computational biology is to annotate, index and search subgraphs in large networks generated with high throughput experiments. Specifically, the problem is to search for well characterized pathways/patterns in a less studied model organism [7]. Subgraph Indexing is useful in querying for pathways/patterns from well studied model organisms in other unfamiliar organisms with known protein-protein interaction networks where vertices and edges represent proteins and interactions, respectively. However, due to possible errors in data collection and different thresholds used in experiments, the data are highly noisy. Missing interactions are common and it is very difficult to clean the data. By discovering and analyzing the approximate matches, biologists would generate solid hypotheses for future studies in understanding and identifying pathways/patterns in not so well studied model organisms.

2. In object-oriented programming, developers and testers handle multiple objects of the same or different classes. The object dependency graph of a program run, where each vertex is an object and each edge is an interaction between two objects through a method call or a field access, helps developers and testers understand the flow of the program and identify bugs. The patterns to be queried that are confirmed by the developers as typical object usages can be used to automatically detect the locations in programs that deviate from them (that is similar to the pattern but not exactly the same) [8]. Hence, by retrieving the approximate occurrences of a typical pattern, developers and testers can quickly locate where the possible bugs are.

In this paper, we investigate the problem of discovering the occurrences of a query graph q in G. The query graph may contain dozens of vertices. Subgraph indexing has been studied before [24, 19, 9]. In previous work, to qualify an occurrence of q in G, all edges of q have to occur. On the other hand, we are studying the subgraph indexing problem in the context of noises, e.g., missing edges. Therefore, in this paper, an approximate match model is developed. In this model, the *edge edit distance* (i.e., the number of edge modifications needed to transform one graph to another) is used to qualify an occurrence of q. If the edge distance between the query graph q and a subgraph q' of G is no more than some threshold θ, then q' is considered as an approximate occurrence of q. This approximate matching model takes into account missing edges in the database graph G. Note that we do not consider the approximate matches with additional edges to the query graphs because such matches are always contained by the matches of the query graphs.

We do not consider label mismatches because the number of possible candidate graphs with label mismatches to a given query graph can be huge. For example, let us assume the size of the query graph is n and the number of vertex labels in the database graph is m, then the total number of candidate graphs with only two label mismatches is $n \times (n-1) \times (m-1)^2$ even without considering any missing edges.

There is a very straightforward solution for approximate query matching. We can first find all graphs whose edge edit distance to

q is no more than θ. Next, for each of these graphs q', the exact occurrences of q' in G can be discovered. In this way, the approximate subgraph matching can be reduced to the problem of exact subgraph matching and previous existing methods, e.g., GADDI [24] can be applied. However, this approach has two shortcomings. First, the exact subgraph matching itself is a very difficult problem since subgraph isomorphism is known to be an NP-hard problem. Secondly, there could be potentially a large number of graphs whose edge edit distance is no more than θ away from q (Denote these graphs as $AI(q, \theta)$). For instance, if q has m edges, then the number of graphs in $AI(q, \theta)$ could be $O(m^\theta)$, which could be very large. Thus, it is crucial to devise an efficient way to process the group of queries.

In this paper, we aim to solve the above two problems. To efficiently identify the occurrences of one subgraph, a novel indexing structure, hybrid neighborhood unit (HNU), is devised. Let $N_i(v, G)$ be the set of vertices u in G such that there exists an i-edge path in G that connects u and v. For each vertex v in the database graph G, HNU stores the degree of v and the labels of v, v's neighbors ($N_1((v, G))$, and v's neighbors' neighbors ($N_2(v, G)$). In most cases, $N_1(v, G)$ is a relatively small set, but $N_2(v, G)$ could be large. For a graph with average degree d, there could be d^2 vertices in $N_2(v, G)$. During the query time, when matching one vertex u in q to a vertex v in G, we need to find out whether the labels in $N_2(u, q)$ are a subset of those in $N_2(v, G)$, which could be costly if these sets are large. To efficiently determine the set relationship, the *bloom filter* [3] data structure is used to represent the labels in $N_2(v, G)$. The bloom filter is an L-bit vector which can be used to determine whether one set is a subset of another. It has the following advantages. It is time efficient and space compact. Moreover, it has no false negatives and only a small rate ($\leq 1\%$) of false positives. Therefore, the vertices in the query graph q can be efficiently matched to the vertices in G with a high accuracy.

To improve the efficiency of processing a set of subgraph queries (graphs in $AI(q, \theta)$), we make the following observation. Although there could be m^θ graphs in $AI(q, \theta)$, these graphs are highly overlapped. Therefore, it is beneficial to query the overlapping parts first since they have the greatest pruning power, i.e., can be used in many of graphs in $AI(q, \theta)$. As a result, the spanning trees of q are used for the query first because (i) many graphs in $AI(q, \theta)$ contain some spanning tree of q and (ii) the time to identify a tree in G is quite small. Based on the matches of the spanning trees, we can map vertices in q to vertices in G.

The graph occurrences have a similar property as the Apriori Property [2] because an occurrences of a supergraph has to contain an occurrence of a subgraph. Therefore, finding the matches of graphs in $AI(q, \theta)$ is similar to that of discovering frequent patterns. As a result, a depth-first enumeration order similar to that of FP-tree [10] is constructed for matching graphs in $AI(q, \theta)$ so that previous discovered occurrences of a graph q' can be used for the matching of later enumerated supergraphs of q'.

The remainder of this paper is organized as follows. Section 2 is the related work and section 3 is the preliminaries. We present how to preprocess the database and construct the index in section 4. Section 5 describes the query processing. The experiment results are presented in section 6. Last, the final conclusion is drawn in section 7.

2. RELATED WORK

These days graph database research has attracted great attention, related works of subgraph indexing for approximate graph matching include subgraph isomorphism algorithms, graph indexing and subgraph indexing, approximate subgraph matching and graph similarity search.

The first category of related research lies in subgraph isomorphism algorithms. Ullmann [20] 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 graph. Cordella [6] proposed a new subgraph isomorphism algorithm for large graphs. These algorithms do not utilize any index structure by preprocessing the database graphs.

Many index-based graph matching and searching schemes have been proposed to find where the query graph occurs in the graph databases [4, 12, 18, 22, 23, 9], which can be further divided into the graph indexing and subgraph indexing. In graph indexing, e.g., gIndex[22], TreePi[23], FG-Index[4], the graph database consists of a set of small graphs. The graph indexing aims to find all database graphs that contain or are contained by a given query graph. On the other hand, in the subgraph indexing e.g., GraphGrep [9], TALE [19], GADDI [24], the goal is to index a very large database graph, so that we can find all or a subset of the matches of a given query graph efficiently in the very large database graph. The proposed method, SAPPER, also deals with subgraph indexing in a very large database graph, and thus falls into this category.

Recently, a number of algorithms are proposed which support approximate graph matching or similarity search through different means [7, 11, 12, 19, 21, 15, 9, 16]. C-Tree[11] organizes database graphs in a tree based structure, where interior nodes are graph closures, and leaf nodes are database graphs. The design of its data structure enables it to perform similarity queries efficiently. In TALE [19], important nodes are matched first and then the match is progressively extended. The method is very effective and fast in approximately finding matches in a large graph. In G-Hash [21], wavelet graph matching kernels are applied along with a hashing scheme. In [15], a top-k query scheme is proposed to find the most similar k answers. However, most of these algorithms are not designed for finding all approximate matches for the query graph with a given threshold in a very large graph. In [16], the authors aim to find the database graphs that are similar to the query graph. Since the database graphs and the query graph are all small, they transform the approximate graph matching to the SET-COVER problem.

Another category of research related to the subgraph matching is graph alignment [14, 17]. 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.

3. PRELIMINARIES

In this section, we introduce the fundamental definitions used in this paper and give the formal problem statement. We investigate the approximate 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 $V \rightarrow \Sigma_V$ and $E \rightarrow \Sigma_E$.*

DEFINITION 2. *The **edge edit distance** from graph g_1 to g_2 is defined as the minimum number of added edges required to transform g_1 into g_2. We denote the edge edit distance as $D_{edit}(g_1, g_2)$.*

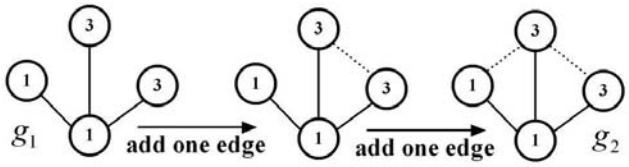


Figure 1: An Example of the Edge Edit Distance

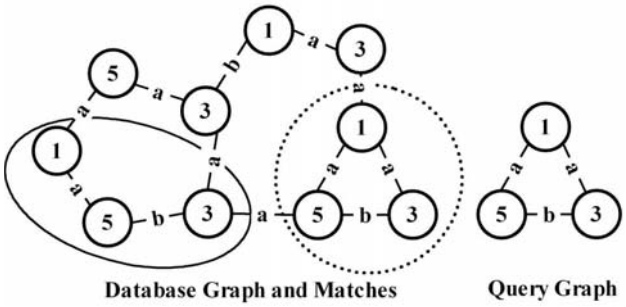


Figure 2: The Database Graph, Query Graph and Matches

For example, in Figure 1, by adding two edges to g_1, we can transform g_1 to g_2. This leads to $D_{edit}(g_1, g_2) = 2$. Edge edit distance is not symmetric, i.e., $\forall g_1, g_2, D_{edit}(g_1, g_2) \neq D_{edit}(g_2, g_1)$. When a graph g_a is not possible to be transformed to another graph g_b by adding edges, we have $D_{edit}(g_a, g_b) = +\infty$.

DEFINITION 3. *Given a database graph G, a connected query graph q, and an integer θ as threshold, a connected subgraph s of G is defined as an* ***approximate match*** *of q in G if and only if $D_{edit}(s, q) \leq \theta$; any graph isomorphic to s is defined as* ***approximately isomorphic*** *to q. The set of graphs approximately isomorphic to q is denoted as $AI(q, \theta)$. If the edge edit distance from an approximate match m to q is exactly zero, m is an* ***exact match*** *of q in G.*

Apparently, the set of approximate matches of any query graph is the superset of the set of exact matches of the same query graph.

In this paper, two restrictions on the approximate match are imposed: (i) the approximate match has to be connected and (ii) only edge additions are considered, but not the edge deletions. A brief discussion on approximate matches without these two restrictions is presented in the appendix.

Problem Statement: We aim to solve the following two problems. (1) Given a large database graph G, we want to construct an index. (2) Given a query graph q and a threshold integer θ, we want to efficiently find all matches of graphs that are approximately isomorphic to q in G with the help of the indexed information. Our goal is not to find some of the matches to the graphs in $AI(q, \theta)$, but to find all matches to the graphs in $AI(q, \theta)$. The word "approximate" refers to the matches of graphs that are approximately isomorphic to the query graph.

In Figure 2, given the query graph and threshold $\theta = 1$, two distinct approximate matches exist in the database graph. The edge edit distance from the left approximate match to the query graph is one, while it is zero for the right match, which is also an exact match.

Before presenting the approximate subgraph indexing method, we will introduce the *subgraph matching* property which will be used extensively later in this paper.

PROPERTY 1. *Given a query graph q and a database graph G, for any exact match g of q in G, let q' be a subgraph of q, g must contain a match of q' in G.*

Figure 2 also illustrates this property: the right exact match in database graph contains any match of a subgraph q' of the query graph q. This property is similar to the Apriori property in the frequent pattern mining [2]. With this property, we can devise an algorithm that searches the matches of a subgraph first. By refining these matches, we can build the matches for larger subgraphs.

The processing of graph queries in our paper can be divided into two major steps. In the first step we construct the index from the database graph. The hybrid neighborhood unit (HNU) is used to store the useful local information for each vertex. In the second step, approximate matches of the query graph q are identified.

4. HYBRID NEIGHBORHOOD UNIT INDEX

In GraphGrep [9], the effectiveness of paths is first revealed, while in TALE [19], neighboring unit proves to be a compact and powerful index unit. In GADDI [24], neighboring distances based index shows its strength in graph matching in a single large graph. Taking the usefulness of these three models into account, we create a new index unit, called hybrid neighborhood unit (HNU). For each vertex v in G, let $N_i(v, G)$ be the set of vertices u in G such that there exists a path of i edges between u and v. For example, $N_1(v, G)$ is the set of vertices that are adjacent to v in G. For the database graph G, we construct the HNU for each vertex v in G. The HNU of v includes four parts: the label v, the degree of v, the labels of vertices in $N_1(v, G)$ and the labels of vertices in $N_2(v, G)$. The first three parts are easy to compute and efficient to store. However, the last part could be too large. For a graph with the average degree of d, $|N_2(v, G)|$ could be $O(d^2)$. The bloom filter [3] is used to store the labels in $N_2(v, G)$.

A **bloom filter** B is an L-bit vector and a set of m independent hash functions $\{f_1, f_2, \ldots, f_m\}$. It is used to determine whether an element x is a member of a set X. Each of the m hash functions f_i maps an element into an integer between 1 and L. Initially, all bits in B are set to 0. If f_i maps an element in X into the integer k, then the kth bit in B ($B[k]$) is set to 1. After mapping every element in X with m hashing functions, some bit in B is 1 while others are 0. To determine whether x is in X, x is mapped to m integers with the m independent hash functions. Assume that $f_i(x) = k_i$. If $x \in X$, then $B[k_i]$ has to be 1 for all k_i ($1 \leq i \leq m$). If $\exists k_i$, $B[k_i] = 0$, then x can not be a member of X. There is no false negative in the bloom filter. However, there could be false positive, i.e., if all mapped bits of x are 1 in B, then there is still a chance that x is not a member of X. The error rate depends on L, $|X|$ (number of elements in X), and m. The optimal number of independent hash functions is approximately $0.7 \times L/|X|$. In addition, if the positive error rate is set to 1%, then $L/|X|$ should be 9.6 [5]. Since X are the labels of vertices in $N_2(v, G)$, $|X|$ can be approximated by d^2 where d is the average degree of a vertex in G. Without a loss of the generality, we choose L and m to be $\lceil 9.6d^2 \rceil$ and 7, respectively to ensure the false positive rate no more than 0.01. If a lower false positive rate is needed, each time we add about 4.8 bits per element to the length of the bloom filter, the false positive rate is reduced by ten times. In the HNU of vertex v, the labels of $N_2(v, G)$ are collected and an L-bit bloom filter is built during index construction time.

The time complexity to obtain the first three parts of the HNU is $O(d)$ for each vertex while the bloom filter takes $O(d^2 \times m + L)$ time to build. Since L is in the order of d^2, the time complexity of bloom filter construction can be simplified as $O(md^2)$. Thus, the

total index construction time for all vertices in G is $O(md^2 \times |V_G|)$ where $|V_G|$ is the number of vertices in G.

5. SAPPER QUERY PROCESSING

In this section, we introduce the approximate subgraph matching algorithm, namely SAPPER. During the query of a subgraph q in G, SAPPER consists of four main steps: vertex matching, constructing random spanning trees of q, generating a matching order of graphs in $AI(q, \theta)$, and the final graph matching. SAPPER first finds candidate matches of each vertex $v_q \in q$ to vertices in G based on the HNUs. Next, we randomly generate a set of spanning trees of q. The matches of the spanning trees are discovered based on the vertices match. The spanning tree matches are used for matching the approximate graphs. Since there are multiple graphs need to be matched, an order on matching these graphs is determined. Finally, matches of all these graphs are discovered.

5.1 Vertex Matching

For each vertex v_q in the query subgraph q, we search for its matches in G based on the HNUs. A vertex v_G in G is a match of v_q if all the following conditions are satisfied: 1) The label of v_q is the same as that of v_G. 2) The degree of v_q is less than or equal to that of v_G. 3) The labels of vertices in $N_1(v_q, q)$ is a subset of those of $N_1(v_G, G)$. 4) The labels of vertices in $N_2(v_q, q)$ is a subset of those of $N_2(v_G, G)$. In the last step, the bloom filter B is employed. Each label in $N_2(v_q, q)$ is hashed via the m hash functions and check whether the corresponding bits in B of v_G are 1. After this step, each v_q is associated with a set of matched vertices in G, denoted as $M(v_q)$.

The total time complexity in this step is $O(d^2 m|V(q)||V(G)|)$ where d, m, $|V(q)|$, and $|V(G)|$ are the maximum of the average degree of G and q, the number of hash functions for the bloom filter, the number of vertices in q, and the number of vertices in G, respectively. There are some false positives in the fourth step due to the bloom filter. The total false positive rate is $1 - (1 - e)^l$ where e and l are the false positive rate of determining whether one element is in the bloom filter and the number of distinct labels in $N_2(v_q, q)$, respectively. This is because if any label out of the l labels is reported as a false positive by the bloom filter of v_G, then v_G is a false positive match of v_q. If e and l are 0.01 and 10, then the total false positive rate is less than 0.1. Since the vertex matching is to find a *candidate* set of matches for a vertex in q, the false positive rate is well in the tolerance.

5.2 Random Spanning Tree Generation and Matching

Although matches for vertices have been discovered, these matches are determined based on the local information (within a 2-edge distance). It is possible that some of these matches are false positives. Therefore, more information needs to be used to prune the matches. Since our ultimate goal is to find matches for all graphs in $AI(q, \theta)$, it is desirable to use the global information existing in a large number of the graphs of $AI(q, \theta)$. All graphs in $AI(q, \theta)$ are θ or less edge edit distance away from q, and hence they are heavily overlapped. Therefore, spanning trees of q will be used for the global information because graphs in $AI(q, \theta)$ would share many spanning trees. In addition, we want each edge in q to have the same probability to be selected into a spanning tree. This could ensure that each graph in $AI(q, \theta)$ would contain a similar number of spanning trees, and thus have a similar amount of pruning power.

A *random spanning tree* T of q has the following property: *each edge e in q has the same probability to be selected into T* [1]. For a graph q with vertices $V(q)$ and edges $E(q)$, a random spanning tree T of q is constructed via a random walk. A random walk on q is a discrete-time Markov chain with the following transition probabilities from a vertex v to another vertex w: $P(v, w) = 1/d_v$ (d_v is the degree of vertex v) if there is an edge from v to w. Otherwise, $P(v, w) = 0$. Initially, a vertex $v_0 \in V(q)$ is randomly chosen as the starting point and the spanning tree T only contains vertex v_0. The random walk starts at v_0. An edge (v, w) is randomly chosen based on the probability P. If w is not in T, edge (v, w) and w are inserted into T. Otherwise, no edge will be added into T. Next the walk is repeated on w. This process terminates until T includes all vertices of V_q. The formal random tree construction algorithm is described in Algorithm 1 in Appendix and an example is depicted in Figure 3. In the example, at time step t_0, T only includes v_0 and no edge. In t_1 time stamp, the edge (v_0, v_1) and v_1 are added into T and T contains vertices v_0, v_1, and one edge. At the time stamp t_2, no edge or vertex is added into T since the random walk is back to v_0. At t_3, the edge (v_0, v_2) and vertex v_2 are added into T and a spanning tree is formed.

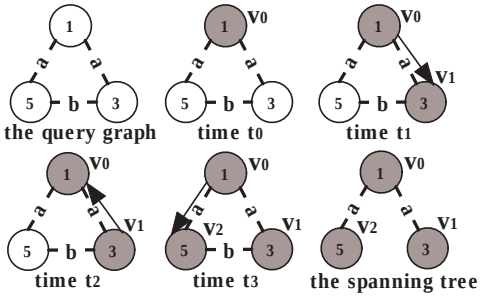


Figure 3: The Random Spanning Tree Generation

A tree generated by this random walk algorithm is a uniform random spanning tree, i.e., the probability of a spanning tree t of a graph q to be generated by Algorithm 1 is $1/TN(q)$, where $TN(q)$ is the number of distinct spanning trees of q. This can be proved by showing that the set of trees constructed by a random walk has a stationary distribution proportional to the degree of the vertex from which it starts. The detailed proof was presented in [1].

In this step, we generate $|V(q) + 1|$ random spanning trees so that (1) each edge has 85% probability to be included in at least one of the spanning trees and (2) the complexity is still not too large. A vertex v in q is randomly chosen as the **prime** vertex. For each generated spanning tree T, we find its matches in G based on the vertices match. The matching starts from the prime vertex v in T and tries to match v's neighbors in T. For example, let's assume that v's matches in G are $M(v) = \{u_1, u_2\}$ and v is connected to v_1 in T. Then we try to see whether v_1 matches to any neighbor of u_1 or u_2 in G. In other words, we want to see whether any neighbor of u_1 or u_2 is in $M(v_1)$. If v_1 only could be matched to some neighbor of u_1, but not any neighbor of u_2, we know that u_2 could not be a match to v for the occurrence of T in G and hence, u_2 could be removed from the match for v of T. The process continues until all matches of T are located. The matching process is performed in a depth-first traversal manner. Since the tree is a very special form of a graph, the match of a tree in G is rather efficient and simple. Due to the space limitations, we omit the details of tree matching in this paper. After the matching process for T, the prime vertex v has a set of matched vertices in G for T. $M(v, T_i)$ is denoted as the set of vertices in G that could be matched to the prime vertex v for the query graph T_i. For example, in Figure 5, v

has label 3 in the query graph, then $M(v, T_1) = \{5, 10\}$ (circled by the solid ellipses), where 5 and 10 are the ids of the mapped vertices of v in the two matches of the spanning tree T_1. Since there are $|V(q) + 1|$ spanning trees, there are $|V(q) + 1|$ sets of $M(v, T_i)$. These matched sets of v serve as the starting point for the later graph matching.

Given a query graph q, and threshold θ, there are approximately $\binom{|E(q)|}{\theta}$ subgraphs of q of $|E(q)| - \theta$ edges. After generating $|V(q)|+1$ spanning trees, a subgraph of q with $|E(q)|-\theta$ edges has the probability P to contain at least one of these random spanning trees, where P is

$$P = 1 - (1 - (\frac{|E(q)| - \theta}{|E(q)|})^{|V(q)|-1})^{|V(q)|+1}.$$

For instance, if q consists of 10 vertices and 20 edges and θ is 2, P would be larger than 0.995. This means that most of these graphs could utilize the match information of the spanning trees.

5.3 Query Graph Enumeration Order

Since there are many graphs in $AI(q, \theta)$, we need devise an order on enumerating these graphs. This problem is similar to that of frequent pattern mining in the data mining field. There are two main approaches to enumerate patterns in frequent pattern mining: breadth-first enumeration and depth-first enumeration. In the bread-first enumeration [2], all patterns (graphs) with i items (edges) are first enumerated. Based on the occurrences (matches) of these pattern (graphs), their super-patterns (super graphs) with one extra item (edge) are enumerated and so on. In the depth-first pattern (graph) enumeration [10], one pattern (subgraph) is generated first, if it has sufficient occurrences (matches), one item (edge) is added into the pattern (subgraph), and the occurrences (matches) of the new pattern is searched and so on. It has been shown that the depth-first enumeration has an advantage over the breadth-first search because in a depth-first search, (1) pattern generation is simpler and more efficient, (2) the match of a pattern can be directly built on its predecessor, and (3) many patterns are not enumerated. Based on this knowledge, we devise a depth-first enumeration of our graphs in $AI(q, \theta)$.

We assign a unique id to each edge in q and a lexicographical order is assumed on these edge ids. Assume that there are z edges in q, whose ids are $e_1 < e_2 < \cdots < e_z$ according to the lexicographical order. (We will discuss how to assign the lexicographical order shortly.) Thus, each graph in $AI(q, \theta)$ can be uniquely represented by a sequence of edges (sorted according to the lexicographical order of the edges). The order of two distinct graphs q' and q'' in $AI(q, \theta)$ can be determined based on their corresponding edge lists. Let edge lists of q' and q'' be $e'_1, e'_2, \ldots, e'_i$ and $e''_1, e''_2, \ldots, e''_j$. respectively. If one sequence is a prefix of another, e.g., q' is a prefix of q'', then we define $q' < q''$. Otherwise, there exists an integer k ($k \leq i$ and $k \leq j$) such that $e'_k \neq e''_k$, then the order of q' and q'' can be determined as follows. Let k be the smallest integer such that $e'_k \neq e''_k$. $q' < q''$ if and only if $e'_k < e''_k$.

By defining the lexicographical order of graphs, the graphs in $AI(q, \theta)$ can be enumerated in a depth-first manner from the lexicographically smallest to the lexicographically largest. First, the edge sequence (graph) with the smallest lexicographical order q_1 is enumerated, which is $e_1, e_2, \ldots, e_l$ ($l = |E(q)| - \theta$). If q_1 has at least one match, then an edge with the smallest lexicographical order after e_l is appended into q_1 to form a new graph q_2 as described in Algorithm 2 in Appendix. (This procedure is illustrated as *next* in Figure 4.) This process continues on q_2 until no edge can be appended into q_2 or there is no match for q_2. In such a case, it is not necessary to enumerate any edge sequences containing q_2 as a prefix. The enumeration process will resume from the lexicographically smallest graph that does not contain q_2 and is larger than q_2. This procedure is described in Algorithm 3 in Appendix. (This procedure is illustrated as *jump* in Figure 4.)

Let's take a look at an example. Assume that q consists of four edges $e_1 < e_2 < e_3 < e_4$ and $\theta = 2$. The lexicographically smallest graph in $AI(q, \theta)$ is (e_1, e_2). If (e_1, e_2) has at least one match, then e_3 is appended and (e_1, e_2, e_3) will be enumerated next. In the case of (e_1, e_2, e_3) has no match, then any sequence whose prefix is (e_1, e_2, e_3) will not be enumerated, namely the sequence (e_1, e_2, e_3, e_4). Next, the lexicographically smallest graph that does not contain (e_1, e_2, e_3) as a prefix and is larger than (e_1, e_2, e_3) is enumerated, which is (e_1, e_2, e_4). Figure 4 shows the enumeration order of graphs in this example. The graphs are enumerated from a top-down and left-to-right fashion. In this method, each graph in $AI(q, \theta)$ will be enumerated or reached at most once. Thus, at most $AI(q, \theta)$ graphs will be enumerated under this method.

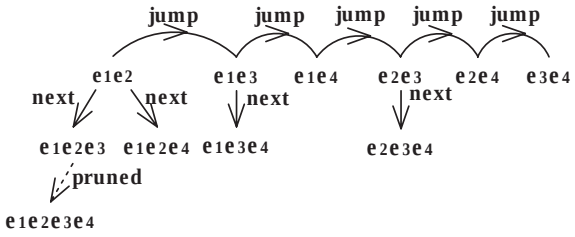


Figure 4: The Enumeration Order

Although any lexicographical order among edges will work, our goal is to prune the graphs in $AI(q, \theta)$ as early as possible. As a result, it is beneficial to search the graphs with the smallest number of matches first so that it can prune the graphs in $AI(q, \theta)$ the most. Therefore, the lexicographical order of edges is set according to the number of matches of each edge. $e_i < e_j$ if edge e_i occurs less times in G than e_j. If two edges have the same number of occurrences/matches, than an order is assigned arbitrarily.

5.4 Graph Matching

After determining the enumeration order of query graphs, we continue to match these graphs in the enumeration order. When matching a graph q_1, there are two cases: q_1 is connected and q_1 is not connected. In the case that q_1 is not connected, it is not necessary to find matches of q_1 since we are only interested in connected query graphs. However, it is possible that some supergraph of q_1 is connected. Thus, we *pretend* there is a match of q_1 (without searching for the matches of q_1), and continue to enumerate the supergraphs of q_1 by appending an edge to q_1.

In the second case that q_1 is connected, we need to find matches of q_1 in G. The matching process can be divided into two cases again according to q_1, (1) we have not yet searched any prefix of q_1 and (2) we have found match(es) of some prefix of q_1. In the first subcategory, since q_1 is very likely to contain at least one pregenerated spanning tree. Thus, the matching of q_1 often could start from the spanning trees. In the rare scenario that q_1 does not contain any randomly generated spanning tree, the match has to start from the vertex matches without the help of the spanning trees. The vertices are matched in a depth-first order. To match a database graph vertex v_g and a query graph vertex v_q, we require that (1) v_g is in $M(v_q)$ and (2) for each edge adjacent to v_q in q (v_q, u_q), there exists a vertex u_g such that the edge label of (v_g, u_g) is the same as (v_q, u_q) and u_g is matched to u_q. This process is similar

to other existing graph matching algorithms, e.g., GADDI [24] and hence we will not present it here due to the space limitations.

When q_1 contains at a least one spanning trees, the following procedure is employed. First, the spanning trees contained by q_1 will be identified via the edges in q_1 and those contained in the spanning trees. Assume q_1 contains r spanning trees $T_1, T_2, \ldots, T_r$. Each match of q_1 has to contain at least one occurrence of T_1, T_2, $\ldots$, and T_r. Therefore, the matched vertices of the prime vertex v for q_1 should be in $M(v, T_i)$ for all $1 \leq i \leq r$. Thus, $M(v, q_1) = \cap_{i=1}^{r} M(v, T_i)$ will serve as the starting point for finding the matches of q_1 in G. Based on the match set of $M(v, q_1)$, we search for the matches of q_1's neighbors and so on. After finding the matches of q_1. For each match of q_1 in G, we keep the mapping from the vertices in the match of q_1 to vertices in q_1. Figure 5 shows an example of matching q_1 based on the matches of the spanning trees. We can see that $M(v, T_1) = \{5, 10\}$ (circled by the solid ellipses) and $M(v, T_2) = \{8, 10\}$ (circled by the dotted ellipses). The intersection of the two sets is $\{10\}$, which is the starting point to match q_1.

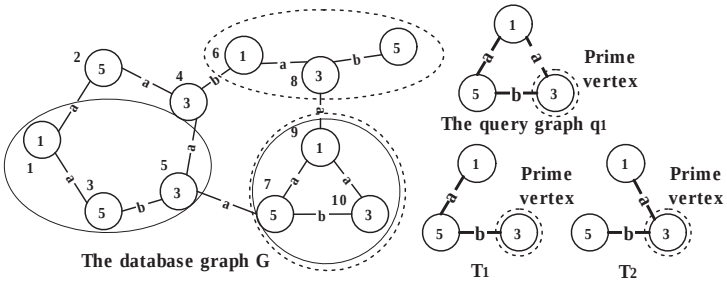


Figure 5: Matching q_1 based on the matches of the spanning trees q_1 contains

In the second sub-category, matches of some of q_1's prefix have been discovered. Let q_2 be the longest prefix of q_1 such that the matches of q_2 have been identified. Also denote that $e^1, e^2, \ldots, e^i$ be the edges in q_1, but not in q_2. For each match of q_2, we check whether $e^1, e^2, \ldots, e^i$ exist in G. If so, this will be a match of q_1. Otherwise, this match of q_2 could not be extended to a match of q_1. This process continues until all matches of q_2 are examined. The formal algorithm is described in Algorithm 4. Figure 6 depicts an example of matching q_1 based on its subgraph q_2 corresponding to the longest prefix of q_1. Then when matching q_1, we only need to check the matches of q_2.

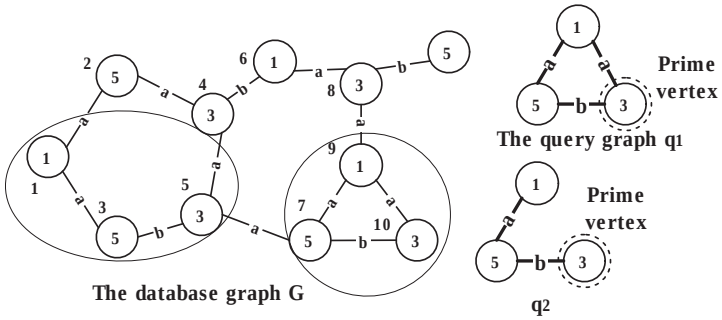


Figure 6: Matching q_1 based on its subgraph q_2

Although the SAPPER algorithm employs approximation to accelerate the matching process, it can find all matches to the graphs that are approximately isomorphic to a query graph. Due to the space limitations, the proof is in the Appendix. It is difficult to determine the exact time complexity of the SAPPER method since it depends on how many graphs in $AI(q, \theta)$ are enumerated. Since the subgraph isomorphism test is an NP-hard problem, the worst case time complexity is exponential. We will empirically analyze the time efficiency and scalability of the SAPPER method in the next section.

6. EXPERIMENTAL RESULTS

In this section, we empirically analyze the performance of SAPPER against TALE, GADDI, two of the most recent subgraph matching tools that designed for large graphs, and Basic SAPPER (BSAPPER). TALE is efficient in index construction and heuristically finds the approximate matches of the query graph. GADDI enumerates all possible approximate isomorphic graphs ($AI(q, \theta)$) of the query graph and finds all exact matches for each of these graphs. To show the pruning power of the random spanning trees and lexicographical order, we also include BSAPPER in the comparison results. BSAPPER employs the same indexing structure as SAPPER, but it differs from SAPPER in the following two aspects. (i) BSAPPER does not use spanning trees. (ii) BSAPPER uses a breadth-first enumeration order similar to the level-wise search algorithm in [2]. In the first level, all the graphs θ edge edit distance away from the query graph q are enumerated and queried. Next it enumerates graphs $\theta - 1$ edge edit distance away in the second level, a graph will be enumerated in the second level if there exists at least one match for all its subgraphs in the first level. This process continues until either the level containing q or no graph can be enumerated based on the subgraph property. The performance difference between BSAPPER and SAPPER is essentially the effects of the random spanning trees and the lexicographical order query graph enumeration while the performance difference between BSAPPER and GADDI is the effects of the bloom filters. All methods are implemented with C++ and run on a Dell PowerEdge 2950, with two 3.0 GHZ dual-core CPUs and 16 GB main memory, and Linux 2.6.16.21-0.8-smp system.

6.1 Protein Interaction Network

In this set of experiments, the graph is generated from a subset of the protein interaction network for homo sapiens. Each vertex represents a protein and the label of the vertex is its gene ontology term from [25]. An edge in the graph represents an interaction between the two proteins it connects. There are 6410 vertices, 53844 edges, and the average degree of a vertex is 16.8. There are a total of 632 distinct labels.

SAPPER spends about 25 minutes to construct an index of 60MB, while TALE spends 10 minutes to construct an index of 15MB, and GADDI spends 35 minutes to construct an 100MB index. As SAPPER processes more information than TALE, it takes more time to construct the index. Since we only need to build an index structure for each database graph once, the query time is much more important than the index building time.

To evaluate these four methods, we use eight known signal transduction pathways from the KEGG database [13] to query the protein interaction network. These known pathways are from species other than homo sapiens, e.g., flies and yeast, etc. Since some protein interaction only exists in yeast or flies and does not exist in human, there are missing edges in the homo sapiens protein interaction network. If θ is set to 2, all eight signal transduction pathways should be recovered in our homo sapiens protein interaction network. Thus, we use these eight pathways as the query graphs and

set θ to 2. SAPPER, BSAPPER and GADDI find all these eight pathways successfully. Among these three methods, SAPPER is much faster than the remaining two due to its advanced pruning techniques. Since TALE is a heuristic algorithm, it only finds two out of these eight pathways. Although TALE runs very fast, its accuracy (e.g., recall) is not high. The execution time of SAPPER, BSAPPER, GADDI, and TALE is shown in Figure 7. The number of vertices on the eight known pathways are 9, 10, 11, 12, and 14. Thus, we report the average execution time with respect to the number of vertices in each query graph.

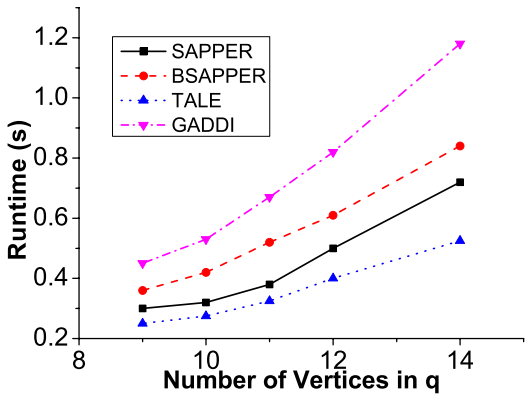


Figure 7: The Performances of the Queries on a Protein Interaction Network

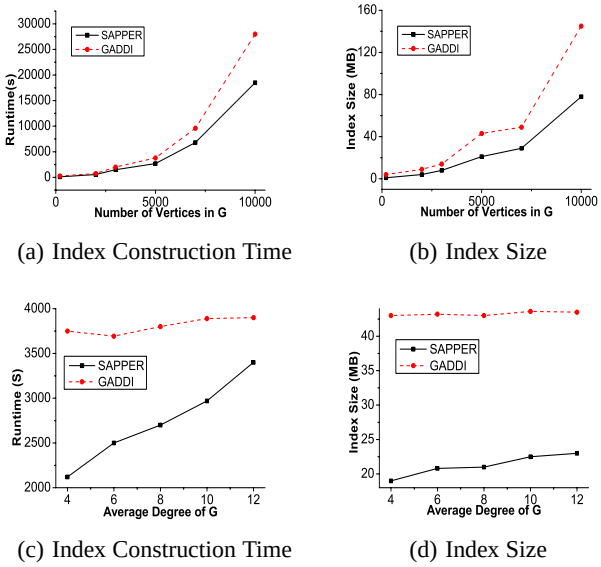
6.2 Synthetic Data Sets

In this portion of the experimental studies, we analyze the performance of SAPPER, BSAPPER and GADDI by independently varying each of **six** parameters on a set of synthetically generated graphs. We do not include TALE because although it can efficiently finish the queries, only around 20% of all the approximate matches are discovered by TALE as shown in the real data set. To systematically analyze the performance of these methods, we vary one parameter at a time. The default values of the parameters are listed in Table 1.

Table 1: Default Parameter Value

Parameter	Default Value
Number of vertices in G	5000
Number of vertices in q	20
Number of Labels	250
θ	1
Average Degree of G	8
Average Degree of q	4

The index construction comparisons are shown in in Figure 8. We first vary the number of vertices in G. GADDI needs more time to construct the index than SAPPER because it needs to calculate the NDS distances for neighboring vertices. Due to the nature of the compactness of the bloom filter, the size of the index of SAPPER is consistently smaller than that of GADDI. When the number of vertices in G is 10,000, SAPPER takes around 18000 seconds to build an 80 MB index. Next, we vary the average vertex degree of G. This affects SAPPER more on the index construction time since the number of 2-hop neighbor vertices grows exponentially with respect to the average degree.



(a) Index Construction Time (b) Index Size

(c) Index Construction Time (d) Index Size

Figure 8: Comparisons of the Indices

Now the average query time of these 3 methods on different parameters are analyzed. The first parameter is the number of vertices in G. The $|V(G)|$ is varied from 200 to 10,000. SAPPER and BSAPPER achieve better matching efficiency than GADDI as they can quickly match vertices by the index and optimize the approximation matching process. SAPPER outperforms BSAPPER due to the effectiveness of the random spanning trees and lexicographical order pruning techniques. The results are shown in Figure 9(a).

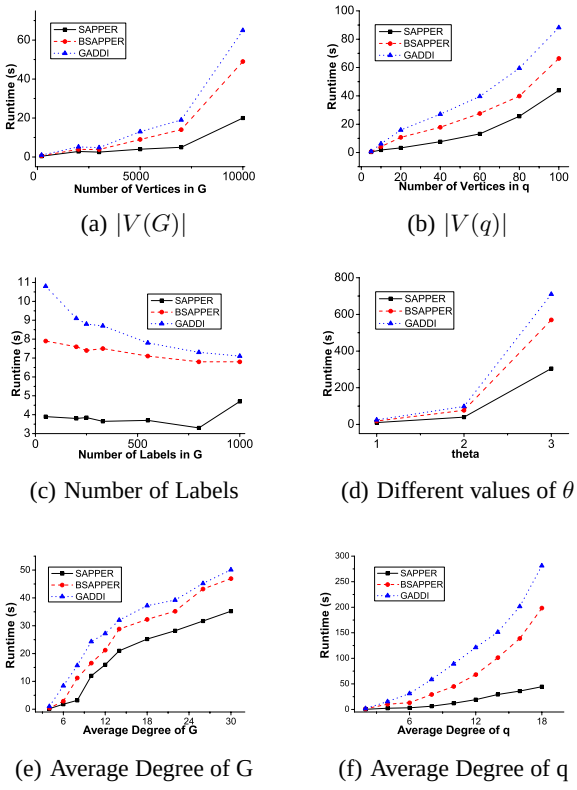
Next we vary the number of vertices in the query graph q. We show the result in Figure 9 (b). With more vertices in q, more vertices and edges need to be compared in the query process, so the query times of all three methods increases. The increase is more evident with $|V(q)| \geq 40$, as the methods need to find all approximate matches, especially GADDI, which processes more candidate graphs for large query graph without pruning techniques.

The third parameter we vary is the number of distinct labels. From Figure 9 (c), we can see that more labels in G increases the pruning power of GADDI, but has a mixed effect on SAPPER. This may be due to the fact that SAPPER only indexes a subset of labels of neighboring vertices. Increasing the number of distinct labels reduces the number of candidate matches between any pair of vertices in G and q, but also decreases the pruning power of SAPPER's index.

The approximate threshold parameter θ is varied and the results are shown in Figure 9 (d). With the increase of θ, the query time of SAPPER is still less than GADDI and BSAPPER because GADDI needs to generate all possible candidate graphs, whose number increases dramatically with θ. On the other hand, due to the use of the advanced pruning techniques, the query time of SAPPER increases at a slower pace.

The fifth parameter we vary is the average degree of G and the results are shown in Figure 9 (e). The high degree in G means more edges have to be examined when matching a pattern and basically the query time of these three methods grows at a similar rate.

Last we vary the average degree of a vertex in q. The results are shown in Figure 9 (f). It is obvious that the higher average degree



(a) $|V(G)|$ (b) $|V(q)|$

(c) Number of Labels (d) Different values of θ

(e) Average Degree of G (f) Average Degree of q

Figure 9: Query Time on Different Parameters

of q is, the more information that q possesses for pruning vertices in G. However, a high vertex degree will also generate more potential candidate query graphs since the number of candidate query graphs is exponential to the average degree of q. When the average degree of q is 2, there are few edges to be examined and all algorithms are efficient. When the average vertex degree of q is larger than 6, the number of edges that need to be compared grows exponentially, which results in GADDI's long response time.

The main difference between TALE and SAPPER is the accuracy. TALE is a heuristic method which does not find all approximate matches of a pattern while SAPPER is an exact method to find the complete set of the approximate matches. Thus, if the goal is to take a quick look of the approximate matches of any query graph in the database, TALE is an efficient and convenient tool. On the other hand, SAPPER is a better choice if the complete set of approximate matches needs to be retrieved. The main difference between GADDI and SAPPER is the efficiency. Although GADDI can find all approximate matches by enumerating all approximate isomorphic graphs of the query graph, this is a very time consuming process. The performance of BSAPPER is between GADDI and SAPPER since it utilizes the bloom filter to match vertices and the subgraph property to prune query graphs without the help of the random spanning trees and lexicographical order. Therefore, when the goal is to discover all approximate matches, SAPPER is preferred.

7. CONCLUSION

Due to the existence of noises (e.g., missing edges) in the large database graph, we are investigating the problem of approximate subgraph indexing, i.e., finding the occurrences of a query graph in a large database graph with (possible) missing edges. In this paper, we have proposed a subgraph indexing and matching method (SAPPER) to find all approximate matches of a query graph. SAPPER constructs the HNU index to accelerate query processing. During the query time, SAPPER improves matching efficiency by using pre-generated random spanning trees and a lexicographical query graph enumeration order. To the best of our knowledge, this is the first attempt to find the complete set of approximate matches in a single large graph. With a large set of real and synthetic data, we demonstrate that the SAPPER approach can outperform the alternative methods in accuracy while achieve a good efficiency.

8. REFERENCES

[1] D. J. Aldous, The random walk construction of uniform spanning trees and uniform labelled trees, *SIAM J. Discrete Math*, 1990.

[2] R. Agrawal and R. Srikant, Fast algorithms for mining association rules, *Prof. of VLDB*, 1994.

[3] B. H. Bloom, Space/time trade-offs in hash coding with allowable errors, *Communications of the ACM 13 (7)*, 1970.

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

[5] B. Chazelle, J. Kilian, R. Rubinfeld and A. Tal, The bloomier filter: an efficient data structure for static support lookup tables, *Proc. of 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2004.

[6] L. Cordella, P. Foggia, C. Sansone and M. Vento, A (sub)graph isomorphism algorithm for matching large graphs. PAMI, 2004.

[7] 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.

[8] T. Nguyen, H. Nguyen, N. Pham, J. AI-Kofahi and T. Nguyen, Graph-based mining of multiple object usage patterns, *Proc. of the Joint Meeting of ESEC and ACM SIGSOFT*, 2009.

[9] R. Giugno and D. Shasha, GraphGrep: A fast and universal method for querying graphs. *Proc. of ICPR*, 2002.

[10] J. Han, J. Pei and Y. Yin, Mining frequent patterns without candidate generation, *Proc. of SIGMOD*, 2000.

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

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

[13] M. Kanehisa and S. Goto, KEGG: Kyoto encyclopedia of genes and genomes, *Nuc. Ac. Res*, 2000, 28:27-30

[14] M. Koyuturk, A. Grama and W. Szpankowski, Pairwise local alignment of protein interaction networks guided by models of evolution. *Proc. of RECOMB*, 2005.

[15] F. Mandreoli, R. Martoglia, G. Villani and W. Penzo, Flexible query answering on graph-modeled data. *Proc. of EDBT*, 2009.

[16] M. Mongiovi, R. Natale, R. Giugno, A, Pulvirenti, and A. Ferro. A set-cover-based approach for inexact graph matching. *Proc. of CSB*, 2009.

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

[18] H. Shang, Y. Zhang, X. Lin, and J. Yu, Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *PVLDB*, 2008.

[19] Y. Tian and J. Patel, TALE: a tool for approximate large graph matching, *Proc. of ICDE*, 2008.

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

[21] X. Wang, A. Smalter, J. Huan, and G. Lushington, G-Hash: towards fast kernel-based similarity search in large graph databases, *Proc. of EDBT*, 2009.

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

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

[24] S. Zhang, S. Li, and J. Yang, Gaddi: distance index based subgraph matching in biological networks. *Proc. of EDBT*, 2009.

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

APPENDIX

A. FORMAL ALGORITHM DESCRIPTION

Algorithm 1 *Generating Random Spanning Tree*

Input: graph q.
Output: a Random Spanning Tree t of q.

Construct transition matrix P from q.
Vertex set $S \leftarrow \varnothing$, edge list $E \leftarrow \varnothing$.
randomly select a vertex X_0 of q.
$S \leftarrow S + X_0$.
$v \leftarrow X_0$.
while $S < |V(q)|$ **do**
  randomly select vertex w by P, e_{vw} exists.
  if $!w \in S$ **then**
    $E \leftarrow E + e_{vw}$
    $S \leftarrow S + w$
  end if
  $v \leftarrow w$
end while
Output the graph composed of edge list E.

Algorithm 2 LEXI_Next

Input: sequence s_1, edge list $EL = \{e_1, ..., e_l\}$, threshold θ.
Output: the next sequence of s_1.

$L \leftarrow$ Length(s_1)
if $s_1(L) < e_l$ **then**
  $e_x \leftarrow s_1(L)$
  return Sequence $s_1(1), ..., s_1(L)e_{x+1}$
end if
LEXI_JUMP(s_1,EL, θ)

Algorithm 3 LEXI_Jump

Input: sequence s_1, edge list $e_1, ..., e_l$, threshold θ.
Output: the next sequence of s_1 which is not a super-sequence of s_1.

if $\exists i$, s.t. $s_1(i) < e_{l-(L-i)}$ **then**
  $x \leftarrow \text{MAX}\{i : s_1(i) < e_{l-(L-i)}\}$
  $e_t \leftarrow s_1(x)$
  if $x \geq l - \theta$ **then**
    return Sequence $s_1(1), ..s_1(x-1)e_{t+1}$
  end if
  return Sequence $s_1(1), ..s_1(x-1)e_{t+1}e_{t+2}...e_{t+l-\theta-x}$
end if
return `end`

B. PROOF OF CORRECTNESS OF SAPPER

The proof of the correctness of SAPPER is divided into two parts. First, we prove that given a query graph q, a database graph G, and an approximation threshold θ, for every connected graph s where $Dist_e(s, q) \leq \theta$ and there exists at least one match of s in G, SAPPER will enumerate s (described in Section 5.3). Second, we want to prove that if s is enumerated in Section 5.3, all of its matches in G will be discovered.

Lemma 1: SAPPER enumerates every candidate graph s of query graph q such that $D_{edit}(s, q) \leq \theta$ and s has at least one exact match in G.

Proof: The lexicographical order enumerates every graph s' such that $D_{edit}(s', q) \leq \theta$ in a depth first style. When we find that such a graph (denoted as s'') does not have any exact match in the database graph, we perform a *jump* procedure. The graphs we skip are all supergraphs of s'', which cannot have any exact match in the database graph, and hence are not candidate graphs. Therefore, we enumerate all candidate graphs s of query graph q such that $D_{edit}(s, q) \leq \theta$ and s has at least one exact match in G.

Lemma 2: SAPPER finds all exact matches of any candidate graph s.

Proof: For a candidate graph s, if we have not yet searched for any prefix of s and s does not contain any pre-generated random spanning trees, then we would perform a depth first matching for s, which will not miss any exact match of s. Otherwise, we start the search from either the matches of the prefix candidate graph of s or the intersection of matches of the pre-generated random spanning trees contained by s. Either the prefix candidate graph of s or a pre-generated random spanning tree contained by s is a subgraph of s. Since any exact match of s must contain at least one exact mach of any subgraph of s based on Property 1, we will not miss any exact match of s in this scenario either. Therefore, SAPPER can find all exact matches of any candidate graph s.

Theorem 1: SAPPER finds all approximate matches of query graph q.

Proof: From Lemma 1, we prove that SAPPER can enumerate all candidate graphs of the query graph. From Lemma 2, we prove that for any candidate graph s, SAPPER finds all matches of s. By the definition of approximate matches, SAPPER can find all approximate matches of q.

Algorithm 4 *Algorithm SAPPER*

Input: database graph G, query graph q, threshold θ.
Output: approximate matches of q.

Sort q's edges decreasingly by their number of matches in G, $l \leftarrow |E(q)|$
edge list $EL \leftarrow e_1, ..., e_l$, ($\forall i, e_i \in q$).
$s \leftarrow e_1, ..s_{l-\theta}$
while $s \neq end$ **do**
  if The graph corresponding to the longest prefix of s is not matched yet **then**
    Find and output the exact matches of $g(s)$ with the help of matches of the spanning trees if it contains any
  else
    Find and output the exact matches of $g(s)$ according to the matches of the graph corresponding to the longest prefix of s
  end if
  if $g(s)$ has no match **then**
    $s \leftarrow$ LEXI_JUMP(s, EL, θ)
  else
    $s \leftarrow$ LEXI_Next(s, EL, θ)
  end if
end while

C. EDGE ADDITIONS/DELETIONS AND DISCONNECTED MATCHES

In this paper, we focus on approximate matches with the following two restrictions: (1) the match has to be connected and (2) only edge additions but not edge deletions are consider ed. The rationale behind these two restrictions are the following. If unconnected matches are considered, there could be too many of these matches. Moreover, these unconnected matches may not be useful in many applications. Thus, in this paper, we focus on finding connected matches.

Edge deletions could be as important as edge additions. In most cases, a match with edge deletions is a super-graph of some other approximate matches. For instance, if g_2 can be obtained by deleting some edge from g_1, then g_1 has to contain g_2. For an approximate match g_2, if the edit distance between g_2 and the query graph q is less than θ, then by adding different edges to g_2, a (potential) large number of matches will be discovered and all these matches

contain g_2 as a subgraph. These matches may not be interesting to users.

However, there is only one exception: g_2 is unconnected. For example, assume that the query graph q is $a-b-c-d$ and $\theta = 2$. The graph $c-d-a-b$ can be considered as an approximate match of q (deletion of edge $d-a$ and addition of edge $b-c$). Since unconnected matches are not discovered by SAPPER, this type of matches could not be retrieved.

Assume that both edge additions and deletions are allowed in our approximate match model. Let g be an approximate match of q in the new model. g can be transformed to q by adding a set of edges E_1 and deleting a set of edges E_2. The **core** of a match g is defined as a graph of $g - E_2$. By addition edges in E_1 to the core of g, we will recover the query graph q. For example, in the previous example, the core of $c-d-a-b$ is $a-b, c-d$ ($E_1 = \{b-c\}$ and $E_2 = \{d-a\}$). All approximate matches of q can be divided into two categories according to their cores: connected cores and unconnected cores.

For matches with connected cores, their cores (subgraphs) will be discovered by SAPPER, and therefore, it is very easy to discover these matches by extending from their connected cores. Since SAPPER does not discover unconnected approximate matches, locating matches with unconnected cores is more complicated. If discovering these matches is useful, the following method can be used. In the case that removing at most $\theta - 1$ edges from q, and q becomes a set of disconnected components, we need locate the matches for each unconnected component of q. Next we examine whether the matches of these unconnected components can be linked together by inserting edges. If so, an approximate match is discovered. There could be more optimization techniques. For example, different cores may share some same unconnected components. By locating the matches of a component in multiple cores, it can save a significant amount of computation time. The optimization techniques for discovering approximate matches with unconnected cores is complicated and could be a future research direction. Thus, we will not elaborate more in this paper.

D. NDS VS. BLOOMING FILTER

There are three main contributions on SAPPER: Blooming filter data structures representing the neighborhoods of a vertex, random spanning trees, and enumeration order of candidate query patterns. It is possible to extend GADDI to approximate subgraph matching with the random spanning trees and the query pattern enumeration order. We call this extended GADDI as GADDI2. The main difference between GADDI2 and SAPPER is how the neighborhoods of a vertex is represented. In SAPPER, the bloom filter is used while in GADDI2, the index of neighborhood discriminative substructures (NDS) is employed. NDS is usually much larger than that of blooming filter as shown in Figure 8. When the indexing structures can be stored in the main memory, then NDS has a better pruning power than bloom filters with about 10% execution time improvement. However, when the database graph is very large, the NDS may not be fit in the memory and thus the thrashing may occur. As a result, the execution time of GADDI2 could become much larger than that of SAPPER with large database graphs as shown in Figure 10.

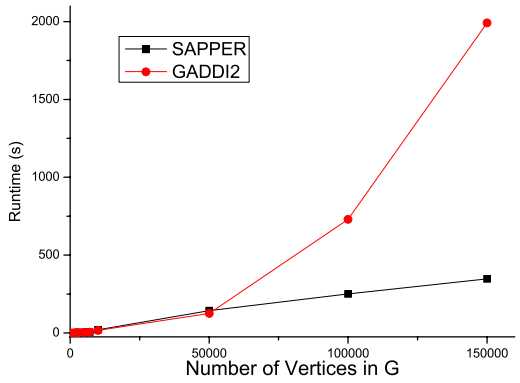


Figure 10: The Performance Comparison Involved with GADDI2