TALE: A Tool for Approximate Large Graph Matching

Yuanyuan Tian, Jignesh M. Patel

EECS Department, University of Michigan,
Ann Arbor, Michigan, USA
{ytian, jignesh}@eecs.umich.edu

Abstract

Large graph datasets are common in many emerging database applications, and most notably in large-scale scientific applications. To fully exploit the wealth of information encoded in graphs, effective and efficient graph matching tools are critical. Due to the noisy and incomplete nature of real graph datasets, approximate, rather than exact, graph matching is required. Furthermore, many modern applications need to query large graphs, each of which has hundreds to thousands of nodes and edges.

This paper presents a novel technique for approximate matching of large graph queries. We propose a novel indexing method that incorporates graph structural information in a hybrid index structure. This indexing technique achieves high pruning power and the index size scales linearly with the database size. In addition, we propose an innovative matching paradigm to query large graphs. This technique distinguishes nodes by their importance in the graph structure. The matching algorithm first matches the important nodes of a query and then progressively extends these matches. Through experiments on several real datasets, this paper demonstrates the effectiveness and efficiency of the proposed method.

1 Introduction

Graphs provide a natural way to model data in a wide variety of applications, such as social networks, road networks, network topology, protein interaction networks and protein structures. Many graph databases are growing rapidly in size. The growth is both in the number of graphs and the sizes of graphs (the number of nodes and the number of edges). For example, the number of interactions (edges in protein interaction networks) in the BIND database [3] grew about 10 folds from 2002 September to 2004 September, and almost doubled after that. The number of protein structures (graphs) in the ASTRAL database [8] has increased more than 3 folds since 2002. There is a critical need for efficient and effective graph querying tools for querying and mining these growing graph databases.

The database community has had a long-standing interest in querying graph databases [6, 9, 17, 19–25]. These previous studies have mostly been carried out within the context of precise graph data, and have focused on exact graph or subgraph matching queries. However, many real graph datasets are noisy and incomplete in nature. For example, it is well known that protein interaction networks produced by high-throughput methods contain many false positives [18]. Moreover, the discovered interactions only represent a small fraction of the true network. As a result, exact graph or subgraph matching often fails to produce useful results.

In contrast, approximate graph or subgraph matching plays a critical role in these applications. Approximate matching allows node/edge insertions and deletions, and node/edge mismatches. Furthermore, many new graph applications prefer approximate matching results rather than exact ones as they can provide more information such as what might be missing or spurious in a query or a database graph.

In addition, most existing graph matching methods are applicable to databases that contain graphs with small sizes, i.e. each graph has a small number (tens) of nodes and edges. Moreover, the query graphs allowed in these methods are also small in size. However, in many new applications, both the query and database graphs are "large". Each graph can contain hundreds to thousands of nodes and edges. For example, in life sciences applications, protein interaction networks for individual species are often matched to determine similarities and differences across species. Each protein interaction network is large, and typically contains hundreds to thousands of nodes and edges in each graph.

The problem that we address in this paper is approximate subgraph matching of large query graphs. Namely, given a *large* query graph, with hundreds to thousands of nodes and edges, and a database of large graphs, we want to find the subgraphs in the database that are similar to the query.

In this paper we present an index-based method for approximate subgraph matching, called TALE (a **T**ool for **A**pproximate Subgraph Matching of **L**arge Queries **E**fficiently). TALE employs a novel graph indexing method, called NH-Index (Neighborhood Index). Most existing graph indexing methods only index subgraphs (paths, trees or general subgraphs), which can lead to index sizes that are exponential in the database size. The indexing unit of NH-Index is the neighborhood of each database node. The neighborhood concept captures the local graph structure around each node, and results in an index with a high pruning power. At the same time, the number of indexing units is equal to the number of nodes in the database, which allows the index to grow linearly with the database size. Furthermore, NH-Index is a disk-based index, which allows it to handle graph databases that do not fit in memory. It employs a hybrid index that uses existing common disk-based index structures, which makes implementation in existing DBMSs straightforward.

We also propose an innovative matching paradigm for querying large graphs. Unlike most previous graph matching tools which treat every node in a graph equally, this matching technique distinguishes nodes by their importance in the graph structure. The algorithm first probes the NH-Index to match the important nodes in a query graph, and then progressively extends the matches by enclosing satisfiable nearby nodes of already matched nodes.

We have applied TALE to three real biological datasets. Our experiments demonstrate that TALE is able to produce useful and meaningful results in all the three cases. In addition, our experimental evaluation shows that TALE is very efficient for large queries, and that the execution time grows gracefully with increasing number of graphs in the database. Through comparisons with other existing tools, we also show that TALE is significantly faster than existing methods.

The main contributions of this paper are as follows:

(1) We propose TALE – a general tool for *approximate* subgraph matching of *large* graph queries. TALE uses a novel disk-based indexing method, which indexes the neighborhood of each database node. It achieves high pruning power and its size scales linearly with the database size. We introduce an innovative graph matching paradigm, which distinguishes nodes by their importance in the graph structure, and accordingly treats them differently in the matching process.

(2) By applying TALE to real applications, we show its effectiveness, significant performance improvements over existing methods, and ability to gracefully handle large graph queries and databases.

The remainder of this paper is organized as follows: Related work is presented in Section 2. Section 3 defines the preliminary concepts. Section 4 describes our indexing mechanism, and Section 5 introduces the TALE algorithm. Experimental results are presented in Section 6, and Section 7 contains our conclusions and directions for future work.

2 Related Work

There is a long history of database research on methods for querying graphs. However, most previous works have focused on exact graph or subgraph matching, i.e. graph or subgraph isomorphism. Subgraph isomorphism was proved to be NP-complete in [5]. 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 database with a large number of graphs. To reduce the search space, GraphGrep [17], GIndex [22] and TreePi [25] index substructures of the database (paths, frequent subgraphs and trees respectively) to filter out graphs that do not match the query.

Several index-based methods for approximate subgraph matching have also been proposed. However, most of these techniques only apply to small graphs and allow limited approximation. Grafil [23] and PIS [24] are both built on top of the exact subgraph matching method GIndex. However, neither method allows node insertion or deletion in their match models. CDIndex [21] only applies to graphs with limited sizes, as it exhaustedly enumerates and indexes all the subgraphs in the database. GString [9] utilizes sequence matching to answer graph queries, but it only applies to applications in which the graphs contain a small number of basic substructures. C-Tree [6], which employs an R-tree like index structure, is a more general tool than the above methods. In Section 6, we compare TALE with C-Tree. A recent method [19], called SAGA, employs a flexible graph similarity model. While SAGA is very efficient for small graph queries, it is computationally expensive when applied to large graphs. In contrast, TALE focuses on approximate matching for large graph queries. In Section 6, we also compare TALE with SAGA.

The life science community has produced vast amount of protein interaction networks. Several tools for comparing protein interaction networks have been proposed. These include PathBlast [10], its successor NetworkBlast [16], MaWIsh [13], and Graemlin [4]. Of these, Graemlin is the latest method and in many ways superior to the other methods for comparing protein interaction networks. In Section 6, we compare TALE with Graemlin.

3 Preliminaries

A graph G is denoted as (V, E), where V is the set of nodes and $E \subseteq V \times V$ is the set of (directed or undirected) edges. Nodes and edges can have labels specified by mappings $\phi : V \rightarrow \Sigma_v$ and $\psi : E \rightarrow \Sigma_e$ respectively, where Σ_v is the set of node labels and Σ_e is the set of edge labels. In order to uniquely identify a node, we assign an unique id to each node in a graph. We also impose an order on the ids. Our indexing method and matching algorithm support both directed and undirected graphs with labeled nodes and/or labeled edges. For ease of presentation, we present our method using undirected graphs with labeled nodes. Adaptations of our method to other graph types are fairly straightforward unless discussed. The simple adaptations are omitted in the interest of space.

Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graphs. An exact graph match (graph isomorphism) is a bijection mapping function $\lambda : V_1 \leftrightarrow V_2$, in which for every $v \in V_1$, $\phi(v) = \phi(\lambda v)$, and $(u, v) \in E_1$ if and only if $(\lambda u, \lambda v) \in E_2$. An exact subgraph match (subgraph isomorphism) from G_1 (the query) to G_2 (the target) is defined as $\exists G'_2 \subseteq G_2$, and G'_2 is an exact graph match for G_1.

Approximate graph matching allows node mismatches (i.e. $\phi(v) \neq \phi(\lambda v)$), and node/edge insertions and deletions. We define an approximate graph match as a bijection mapping $\lambda : V'_1 \leftrightarrow V'_2$, where $V'_1 \subseteq V_1$ and $V'_2 \subseteq V_2$. Similarly, an approximate subgraph match from G_1 (the query) to G_2 (the target) is defined as $\exists G'_2 \subseteq G_2$, and G'_2 is an approximate graph match for G_1.

An approximate subgraph matching tool often returns a large number of matches for a query. Often the user is only interested in the top-K results. To return the top-K results, TALE has to sort the matches based on their similarities to the query. Several graph similarity or distance models have been proposed, e.g. [2, 19]. Each model is meaningful for some applications, but there is no "universal" model that fits all applications. We do not want to limit the generality of TALE by tailoring it to a particular similarity model. Instead, we let the users customize the similarity method that best models their application, thereby allowing TALE to serve as flexible graph matching tool that can be used in a variety of graph matching applications. Section 6 shows examples of how this similarity model can be customized in practice.

4 The NH-Index

In this section, we introduce the novel indexing technique, Neighborhood Index (NH-Index).

4.1 Indexing Unit

The first question that arises with a graph indexing method is the graph entities, e.g. nodes, edges, subgraphs, etc., that should be indexed. The NH-Index is used by the matching algorithm to match the important nodes in the query graph. These initial matches for the important nodes are then extended to produce the final matching results. A naive indexing method is to index all the nodes in the database. This method has the benefit that the index size grows linearly with the number of nodes in the database, but suffers from low pruning power, as each query node can have many false positive matches (matches that cannot be extended later). Our NH-Index size is linear in the number of nodes in the database and also has a high pruning power. NH-Index achieves this by incorporating neighborhood information into the naive node indexing method. When matching a query node, instead of looking at the node in isolation, NH-Index also considers its neighborhood. A database node matches the query node, only if the two nodes match *and* their neighborhoods also match. Using this technique, a large fraction of false positives can be eliminated.

A *neighborhood* is defined as the induced subgraph of a node and its neighbors (adjacent nodes). There are three main properties that characterize the neighborhood of a node: the number of neighbors, how the neighbors connect to each other, and the labels of the actual neighbors. The number of neighbors is simply the degree of the node. To quantify the "connectedness" amongst the neighbors, we define *neighbor connection* as the number of edges between the neighbors. For example, the neighbor connection of the black node in Figure 1 is 5.

To capture the neighbors of a node, a naive method is to simply enumerate the labels of the neighbors. However, this naive approach results in variable-length index entries as well as large index size (in the worst case of a clique, the storage cost is $O(n^2)$, where n is the number of nodes in the database). An alternative to the naive approach is to use a compact bit array to capture the neighbors set. In the simple case when the

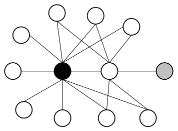

Figure 1. An example graph

total number of different labels in the problem domain is small (i.e. $|\Sigma_v|$ is small), we can use a deterministic bit array to store the neighbors. The size of the bit array is equal to $|\Sigma_v|$, and each bit in the array indicates whether a neighbor with a specific label exists (set to 1) or not (set to 0). We call this bit array *neighbor array.* When $|\Sigma_v|$ is a large number, using a deterministic bit array is very expensive. To handle this situation, we employ the Bloom filter approach [1]. We fix the size of the bit array to be S_{bit}, where S_{bit} is a user-controllable parameter. A hash function is utilized to map a node label to a bit array position. To improve precision, multiple bit arrays and hash functions can be used to characterize the neighbors of a node. For simplicity, we only use one bit array to store the neighbor information in this work.

In summary, the indexing unit of the NH-Index contains the following information: *(label, degree, nbConnection, nbArray)*, where *nbConnection* is the neighbor connection of the node, and *nbArray* is the neighbor array.

4.2 Matching a Query Node

In the previous section, we discussed the indexing unit of the NH-Index. Next, given a query node, we examine how our method finds the matching database nodes. For ease of presentation, we first investigate the matching conditions for exact subgraph matching, and then extend it to approximate subgraph matching.

For exact subgraph matching, in order to match a query node to a database node, the two nodes must have the same label. The degree of the query node should be no more than that of the database node. The same condition holds for neighbor connections. Besides, the neighbors of the query node should have corresponding matching nodes in the neighborhood of the database node.

For approximate matching, we want to tolerate some misses in the match. We introduce a single user-defined parameter ρ, which is used to control the degree of approximation. Intuitively, ρ is the percentage of neighbors of a query node that can have no corresponding matches in the neighborhood of a database node. In other words, $nb_{miss} = (\rho \times N_q.degree)$ neighbors of the query node can be missing in the match to a database node. If nb_{miss} nodes are allowed to be missing, then at most $nbc_{miss} = nb_{miss} \times (nb_{miss} - 1)/2 + (N_q.degree - nb_{miss}) \times nb_{miss}$ neighbor connections are allowed to be missing in the match, i.e. in the worst case, the nb_{miss} nodes all connect to each other, and also connect to all of the remaining $(N_q.degree - nb_{miss})$ nodes.

Note that we also support node mismatches (nodes with different labels are matched) in TALE. For ease of presentation, we delay the discussion of node mismatches to Section 4.5.1, and for now assume that matching nodes are required to have the same label.

Formally, the conditions for matching a query node to an NH-index entry for approximate subgraph matching is:

$$N_{db}.label = N_q.label \tag{4.1}$$

$$N_{db}.degree \geq N_q.degree - nb_{miss} \tag{4.2}$$

$$\sum_{i=1}^{S_{bit}} Miss(N_{db}.nbArray[i], N_q.nbArray[i]) \leq nb_{miss} \tag{4.3}$$

$$N_{db}.nbConnection \geq N_q.nbConnection - nbc_{miss} \tag{4.4}$$

The $Miss$ function in Equation 4.3 is defined as follows:

$$Miss(x, y) = \begin{cases} 1 & \text{if } x = 0 \text{ and } y = 1 \\ 0 & \text{otherwise} \end{cases}$$

In fact, exact subgraph matching can be viewed as a special case of approximate subgraph matching when $\rho = 0$.

Note that the conditions expressed in Equations 4.1 to 4.4 can result in producing some false positives. Our index serves as a filtering mechanism to prune the search space. These matches are then refined in the matching algorithm (Section 5).

4.2.1 Node Match Quality

Given a query node, there can be more than one database node that satisfies the conditions specified in Equations 4.1 to 4.4. Each of these matches can have a different match quality. Therefore, we need to measure the quality of the node matches. This quality metric will then be used at a later step (see Section 5.2) following the index probe. In this section, we describe the match quality computation.

Let $\widetilde{nb}_{miss}$ be the actual number of missing neighbors in the node match, and $\widetilde{nbc}_{miss}$ be the actual number of missing neighbor connections. Then the fraction of missing neighbors of the query node can be defined as $f_{nb} = \frac{\widetilde{nb}_{miss}}{N_q.degree}$. And the fraction of missing neighbor

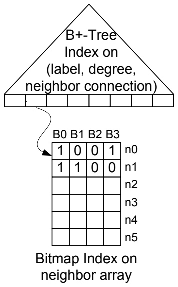

Figure 2. The hybrid index structure

connections can be defined as $f_{nbc} = \frac{\widetilde{nbc}_{miss}}{N_q.nbConnection}$. Then, we define the *quality* of a node match, w, as:

$$w = \begin{cases} 2 - f_{nbc} & \text{if } \widetilde{nb}_{miss} = 0 \\ 2 - (f_{nb} + \frac{f_{nbc}}{\widetilde{nb}_{miss}}) & \text{otherwise} \end{cases} \qquad (4.5)$$

Note that f_{nbc} is correlated with f_{nb}, as more missing neighbors is likely to result in more missing neighbor connections in the match. Therefore, we amortize f_{nbc} by the number of missing neighbors $\widetilde{nb}_{miss}$ in Equation 4.5. The value of $(f_{nb} + \frac{f_{nbc}}{\widetilde{nb}_{miss}})$ falls between 0 and 2. We subtract this value from 2, so that higher w value means a better node match.

4.3 Index Structure

Next, we consider the index structure to implement the NH-index. Rather than designing a new index structure, which makes adoption and implementation hard, it is desirable to consider using existing index structures that can implement the NH-index efficiently. A suitable index structure needs to support the conditions specified in Equations 4.1 through 4.4. We propose a simple hybrid index structure (see Figure 2) for the NH-Index.

This hybrid index structure has two levels. The highest level of the index structure is a B+-tree index on node label, degree and neighbor connection. This part of the index is used for fast evaluation of the equality search on node labels (Equation 4.1), range search on node degrees (Equation 4.2) and neighbor connections (Equation 4.4). Each leaf entry in the B+-tree index points to a second-level index. This second-level index has two components. The first is a list of database node ids that are represented by the B+-tree leaf index entry. (Recall from Section 3 that every database graph node has a unique node id.) These nodes have the same unique label, degree and neighbor connection. The second component is a bitmap index for the neighbor arrays of these database nodes. Each node has one corresponding bit array in the bitmap. Figure 2 shows an example bitmap index for a B+-tree leaf entry that is mapped to six distinct database nodes with the same label, degree and neighbor connection. The bitmap index is used to expedite the evaluation of Equation 4.3 using Algorithm 1 (discussed in detail below).

Note that our hybrid index structure is easily implemented in existing relational systems. The second level indices can be implemented simply as a relation with two attributes: one that stores the list of database nodes, and the other that stores a bitmap (using an extensible data type). The first level index is simply a B+-tree built on this table. This simple implementation is robust and allows us to easily realize the NH-Index.

4.4 Index Probing

Given a query node, we first utilize the label, degree and neighbor connection information to probe the B+-Tree index. Then, we obtain a list of bitmaps that must be further examined using the conditions specified in Equation 4.3. An efficient algorithm for this evaluation is shown in Algorithm 1. This algorithm contains two steps. The first step (line 1 to 17) counts the number of missing neighbors of the query node in the match to each database node in a bitmap. The second step (line 18 to 30) prunes all the database nodes with the number of missing neighbors higher than the user threshold. We discuss these two steps in detail below.

If a position in the query neighbor array is set to 1, but the corresponding position in a database neighbor array is 0, we count it as one miss. Step 1 of Algorithm 1 simulates the binary addition operation to count the total number of misses. We keep a counter of $countSize + 1$ bits ($countSize = \lfloor \log_2(nb_{miss}) \rfloor + 1$) for each database node to record the number of misses. These counters are stored in the $countSize + 1$ bit vectors $Count[0]$ to $Count[countSize]$, i.e. vector $Count[0]$ stores the bit position 0 for all the counters, and so on. The algorithm scans through the query neighbor array from the lowest bit (position 0) to the highest bit (position $S_{bit} - 1$). If the current bit is 1, then the algorithm negates the bits in the corresponding column of the bitmap index and adds all the bit values to the counters of the database nodes. To avoid overflow, the highest bit $Count[countSize]$ for a database node is set to 1 when the number of misses exceeds $countSize$ bits. An example of the first step is shown in Step 1 of Figure 3.

The second step of Algorithm 1 prunes all the database nodes with more than nb_{miss} misses. We use two bit vectors $Result_{eq}$ and $Result_{lt}$ to record the nodes with nb_{miss} misses and less than nb_{miss} misses,

Algorithm 1 Bitmap Probe for Approximate Subgraph Matching (N_q, $Bitmap$, ρ)

Input: N_q is the query node, $Bitmap$ is the bitmap index to be probed, assuming that there are n nodes in the bitmap index and the size of neighbor array is S_{bit}, ρ is the percentage of neighbors of a query node that can be missing in the match to a database node

Output: $Result_{le}$ is the bit vector indicating which nodes satisfy the query

```
// [Step 1]: count the number of missing neighbors
$nb_{miss} = \lfloor \rho \times N_q.degree \rfloor$ // the threshold for the number of missing neighbors
$countSize = \lfloor \log_2(nb_{miss}) \rfloor + 1$
for $i$ from 0 to $countSize$ do
  $Count[i] = (0, 0, ..., 0)$ // $Count[i]$ is a bit vector of size $n$
end for
for $j$ from 0 to $S_{bit} - 1$ do
  if $N_q.nbArray[j] = 1$ then
    $Carries$ = NOT $Bitmap.B_j$
    for $k$ from 0 to $countSize - 1$ do
      $Temp = Count[k]$ AND $Carries$
      $Count[k] = Count[k]$ XOR $Carries$
      $Carries = Temp$
    end for
    $Count[countSize] = Count[countSize]$ OR $Carries$
  end if
end for
// [Step 2]: only return nodes with no more than $nb_{miss}$ missing neighbors
$Result_{lt} = (0, 0, ..., 0)$ // $Result_{lt}$ is a bit vector of size $n$
$Result_{eq} = (1, 1, ..., 1)$ // $Result_{eq}$ is a bit vector of size $n$
for $k$ from $countSize$ to 0 do
  if bit $k$ of $nb_{miss}$'s binary format is 1 then
    $Result_{lt} = Result_{lt}$ OR ($Result_{eq}$ AND (NOT $Count[k]$))
    $Result_{eq} = Result_{eq}$ AND $Count[k]$
  else
    $Result_{eq} = Result_{eq}$ AND (NOT $Count[k]$)
  end if
end for
$Result_{le} = Result_{lt}$ OR $Result_{eq}$
return $Result_{le}$
```

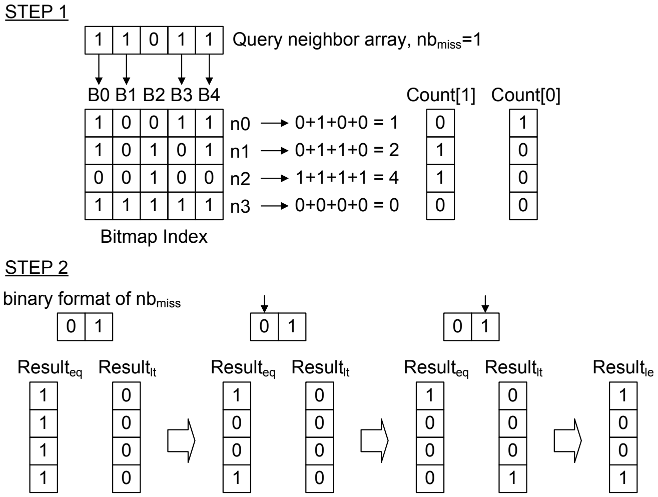
STEP 1
1 1 0 1 1 Query neighbor array, nb_miss=1
B0 B1 B2 B3 B4
Count[1]
Count[0]
1 0 0 1 1 n0 → 0+1+0+0 = 1
1 0 1 0 1 n1 → 0+1+1+0 = 2
0 0 1 0 0 n2 → 1+1+1+1 = 4
1 1 1 1 1 n3 → 0+0+0+0 = 0
0 1 1 0
1 0 0 0
Bitmap Index
STEP 2
binary format of nb_miss
0 1
Result_eq
Result_lt
Result_le

Figure 3. Example demonstrating Algorithm 1

respectively. As the algorithm scans the binary format of nb_{miss} from the highest bit (position $countSize$) to the lowest bit (position 0), it updates $Result_{eq}$ and $Result_{lt}$. Finally, the bitwise OR of the two vectors gives us the right answer. Each position in the result vector indicates whether the corresponding database node is in the query result or not. Figure 3 also shows an example of this step. This second step of Algorithm 1 resembles the algorithm for range predicate with a bit-sliced index introduced in [15].

Next, we analyze the complexity of Algorithm 1. This algorithm takes $O(S_{bit} \times \log(\rho \times d))$ bitwise operations in step 1, where d is the degree of the query node. And step 2 takes $O(S_{bit})$ bitwise operations. Therefore, the complexity of Algorithm 1 is $O(S_{bit} \times \log(\rho \times d))$ bitwise operations on bit vectors. Usually, $\rho \times d$ is very small value, thus $\log(\rho \times d)$ is even smaller, and often negligible.

We have also compared Algorithm 1 with a naive bitmap index probing method, which scans through every neighbor array in the bitmap index, and decides whether the neighbor array satisfies the condition specified in Equation 4.3. We set up a simulation to test the efficiency of Algorithm 1 against this naive method. We randomly generated 12 bitmap indexes with increasing sizes. The smallest bitmap index contains neighbor arrays for 16 nodes, while the largest one contains neighbor arrays for 32768 nodes. Each neighbor array in the bitmap has 32 bits. We use 50 randomly generated query neighbor arrays to probe these bitmap indexes. Algorithm 1 shows significant performance advantage over the naive method – the speedup ranges from 2X for the smallest index to more than 12X for the largest index.

4.5 Extensions to the Basic Approach

Next we introduce several extensions to the basic indexing technique to improve the basic approach and handle more general cases.

4.5.1 Node Mismatches

In the above indexing method, TALE requires two matching nodes to have the same label. However, real applications often need to allow matchings between nodes with different labels. We adopt the node mismatch model introduced in [19], which implicitly groups nodes based on a specific notion of similarity. In this model, the grouping of nodes is defined based on the application domain, and two nodes are allowed to match only if they belong to the same group. For example, if a node represents a gene, then its group membership is defined by the orthologous group that it belongs to (orthologous groups are organized based on similar gene functionalities), and two nodes match if they belong to the same orthologous group. To accommodate this model, we extend the basic indexing approach by replacing the node labels with their corresponding group labels and hashing the group labels for the bit arrays. The remaining indexing method remains unchanged. In Section 6, we show how TALE can be applied to real applications using this node mismatch model.

4.5.2 Directed Graphs

The above indexing method works for undirected graphs. However, it is fairly easy to extend it to handle directed graphs. In a directed graph, every edge has direction. Given a node, an adjacent edge either goes towards the node or away from the node. Therefore, the indexing unit becomes *(label, in-degree, out-degree, in-nbConnection, out-nbConnection, in-nbArray, out-nbArray)*. For the index structure, we can build one B+-Tree index on label, in-degree, out-degree, in-nbConnection and out-nbConnection. And each leaf entry in the B+-Tree points to one bitmap index for the in-nbArray and another bitmap index for the out-nbArray. Other candidate index structures are also possible.

4.5.3 Edge Labels

A simple extension can be made to the basic indexing method proposed above to handle graphs with labeled edges. In the basic method, we hash the labels of neighbors to get the neighbor array. To handle labeled edges, we hash (node label, edge label) pairs to produce the neighbor arrays. The remaining index method is unchanged.

Any of the above extensions can be combined together to meet the requirement of different applications.

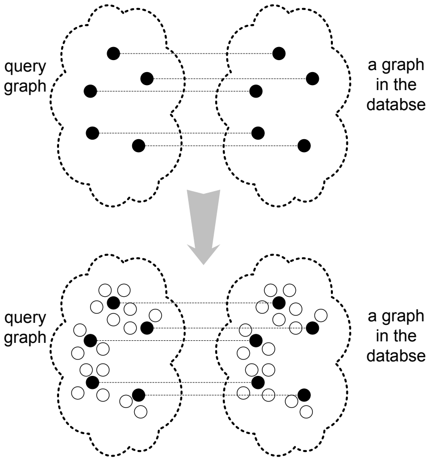

Figure 4. Overview of the matching algorithm

5 The Matching Algorithm

In this section, we introduce the approximate subgraph matching algorithm. We first start with an overview of this algorithm in Section 5.1, and then describe the algorithm in detail in Sections 5.2 and 5.3.

5.1 Algorithm Overview

Our approximate subgraph matching algorithm is based on the following two observations.

Observation 1: Some nodes in a graph play more importance roles in the graph structure than others. As shown in Figure 1, some nodes (e.g. the black node) connect to many other nodes. If these nodes are absent, then the graph structure quickly gets fragmented. In contrast, some nodes (e.g. the gray node) sit on the periphery of the graph and only connect to few other nodes. The overall graph structure will not be dramatically affected by removing these nodes. There are various ways of measuring the importance of a node in a graph. For simplicity, we use the degree centrality measure in this work. In this measure, nodes with high degrees are considered more important than nodes with low degrees. In Section 6.5, we will evaluate the effectiveness of this importance measure. Note that the definition of “importance” is flexible in TALE and customizable for specific application needs. TALE can be easily extended to use other measures of node importance, such as closeness, betweenness, and eigenvector centralities.

```
Algorithm 2 GrowMatch ($G_q$, $G_{db}$, $M_{imp}$)
Input: $G_q$ is the query graph, $G_{db}$ is the database graph, $M_{imp}$ contains the matches for the important nodes in $G_q$
Output: $M$ contains the node matches for the resulting graph match
put all node matches from $M_{imp}$ to a priority queue $Q$ sorted by their qualities
while $Q$ is not empty do
    pop up the best node match ($N_q$, $N_{db}$) from $Q$
    put ($N_q$, $N_{db}$) into $M$
    ExamineNodesNearBy($G_q$, $G_{db}$, $N_q$, $N_{db}$, $M$, $Q$) // finding new matches for nodes nearby $N_q$
end while
return $M$
```

Observation 2: A good approximate match should be more tolerant towards missing unimportant nodes in the query than missing important nodes. In other words, most of the important nodes in the query should be present in the match, while missing unimportant nodes is more tolerated. In addition, the number of matched important nodes, and the qualities of these node matches can be used to estimate the quality of an approximate subgraph match.

Based on these two observations, we introduce a new approximate subgraph matching algorithm. The overview of this algorithm is as follows: First, the algorithm selects a number of important nodes from the query based on the specified importance measure (degree centrality in this work), and then probes the NH-Index to find matching nodes for these important query nodes. These matching node pairs serve as anchor points for producing graph matches. In the second step, for each matching database graph, the algorithm extends the graph match from the anchor points by progressively adding satisfiable nearby nodes of already matched nodes. The entire matching process is outlined in Figure 4.

5.2 Step 1: Match the Important Nodes

In this step, the algorithm selects a number of important nodes from the query and probes the NH-Index to match these important nodes.

The algorithm first needs to decide how many nodes count as important nodes. We introduce a parameter P_{imp}, defined as the fraction of important nodes in the query. Given P_{imp}, we sort the nodes in the query by their importance (degree centrality in this work) and select the top P_{imp} percent as the important nodes. In Section 6.2, we show how to choose the P_{imp} value based on graph properties of specific applications.

```
Algorithm 3 ExamineNodesNearBy ($G_q$, $G_{db}$, $N_q$, $N_{db}$, $M_c$, $Q_c$)
Input: $G_q$ is the query graph, $G_{db}$ is the database graph, $N_q$ is a node in $G_q$, $N_{db}$ is the node in $G_{db}$ matched to $N_q$, $M_c$ contains all the current node matches found so far, $Q_c$ contains all the candidate node matches to be examined
$NB1_q$ = immediate neighbors of $N_q$ that have no matches in $M_c$
$NB2_q$ = nodes two hops away from $N_q$ that have no matches in $M_c$
$NB1_{db}$ = immediate neighbors of $N_{db}$ that have no matches in either $M_c$ or $Q_c$
$NB2_{db}$ = nodes two hops away from $N_{db}$ that have no matches in either $M_c$ or $Q_c$
MatchNodes($G_q$, $G_{db}$, $NB1_q$, $NB1_{db}$, $M_c$, $Q_c$)
MatchNodes($G_q$, $G_{db}$, $NB1_q$, $NB2_{db}$, $M_c$, $Q_c$)
MatchNodes($G_q$, $G_{db}$, $NB2_q$, $NB1_{db}$, $M_c$, $Q_c$)
```

After selecting the important nodes, the algorithm probes the NH-Index for each important node as discussed in Section 4.4. After the index probe, we obtain a list of database graphs that have matches for some or all of the important nodes in the query. A match score is also calculated for each matching node pair using Equation 4.5. In the results produced by the index probes, a single important query node can be mapped to multiple database nodes and vice versa. Since the main purpose of this first step is to find the anchor points that can be expanded in the next step, we need to find one-to-one node mappings between the query and database nodes. For this part, we use a maximum weighted bipartite graph matching algorithm (using node match scores as weights) from the LEDA-R 3.2 library (`http://www.algorithmic-solutions.com/index.htm`).

5.3 Step 2: Extend the Match

Step 1 of the matching algorithm produces a list of candidate database graphs. For each candidate graph, Step 2 of the algorithm utilizes the node matches produced by Step 1 as anchor points to match the remaining nodes in the database and query graphs, and produces the final graph match.

The overall idea of this step is as follows. For each node that is already matched, we try to match its "nearby" nodes (as described below these includes not just the adjacent nodes, but also nodes that are two hops away). We perform this extension progressively until no more nodes can be added to the match. The detailed algorithm is shown in Algorithm 2, 3 and 4.

Algorithm 2 is the main procedure for step 2. It first puts all the important node matches (the anchor points) into a priority queue sorted by the qualities of the node matches (cf. Section 4.2.1). In each iteration of the loop, we pop up the best node match (with the highest quality) from the queue and put it into the final graph match. In addition, we examine the nearby nodes of the query node, as well as the nearby nodes of the database node, to see whether any of them can be matched. If so, we add these new node matches to the priority queue. This process ends when the priority queue is empty.

Algorithm 3 implements the ExamineNodesNearBy function called by Algorithm 2. Based on a pair of already matched nodes, this function tries to match their nearby nodes. In order to allow more flexibility in the approximate matching, we do not limit the matching extensions to just adjacent nodes of the query node and the database node. Instead, this algorithm examines nodes at most two-hops away from the query node and the database node. Note that this algorithm is generic. It can be easily extended to match nodes more than two-hops away to allow more approximation (at the expense of an increased computational cost).

Algorithm 4 shows the details of the MatchNodes function called by Algorithm 3. For each node from the given set of query nodes, this algorithm finds the best matching node from the set of database nodes. If the new node match does not conflict with any existing ones in the priority queue, it is simply put into the priority queue. However, if this node match is better than an existing match in the queue, the existing one is replaced with the new one.

Note that our algorithm only produces one match for each database graph. In some applications, users may want more than one match for each database graph. In this case, we can extend our matching algorithm to retain more than one set of anchor points (in step 1, instead of only retaining the maximum weighted bipartite matching, also retain other high weighted maximal bipartite matchings) and then extend each of them to produce a match.

6 Evaluation

In this section, we apply TALE to three real biological applications, and present results evaluating TALE with three measures: effectiveness (whether the results produced by the tool are useful and meaningful in real life applications), efficiency and scalability.

Algorithm 4 MatchNodes(G_q, G_{db}, S_q, S_{db}, M_c, Q_c)

```
Input: $G_q$ is the query graph, $G_{db}$ is the database
    graph, $S_q$ is a set of nodes in $G_q$, $S_{db}$ is a set of
    nodes in $G_{db}$, $M_c$ contains all the current matches
    found so far, $Q_c$ contains all the candidate matches
    to be examined
for every node $N_q$ in $S_q$ do
    $N_{db}$=the best mapping of $N_q$ in $S_{db}$
    if $N_{db}$=null then
        continue
    end if
    if $N_q$ has no matches in $Q_c$ then
        put ($N_q$, $N_{db}$) into $Q_c$
        remove $N_{db}$ from $S_{db}$
    else if ($N_q$, $N_{db}$) is a better node match then
        remove the existing match of $N_q$ from $Q_c$
        put ($N_q$, $N_{db}$) into $Q_c$
        remove $N_{db}$ from $S_{db}$
    end if
end for
```

Note that while the applications discussed in this paper are from life sciences, TALE can be applied to any area in which there is a need for approximate subgraph matching. Other such areas include comparing RDF graphs in semantic web applications, and comparing parse trees produced by natural language parsers for literature mining. We have chosen to focus on life sciences applications since we have actual collaborators who have ready applications for our tool.

TALE is implemented in C++ on top of PostgreSQL (`http://www.postgresql.org`). The execution times reported in this section correspond to the running times of this C++ program including the DBMS access times. All experiments were run on a 2.8GHz Pentium 4 Fedora Core 2 machine, with 2GB memory, and a 250GB SATA disk. We use PostgreSQL version 8.1.3 and set the buffer pool size to 512MB.

6.1 Experimental Datasets

BIND Dataset: We use the BIND [3] dataset (version May 25, 2006) to demonstrate the application of TALE for comparing Protein Interaction Networks (PINs). A PIN is a large graph, in which nodes represent proteins and edges indicate protein-protein interactions. Comparing PINs of different species allows a biologist to discover the evolutionary conserved functional units across species. However, due to the high error rate of detection methods, PINs are noisy in nature [18]. Therefore, approximate subgraph matching is useful for comparing PINs.

KEGG Dataset: This dataset consists of biologi-

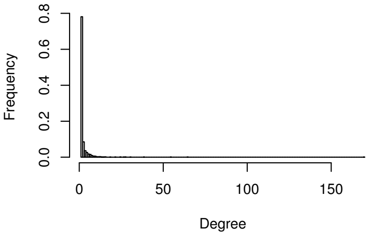

Figure 5. Degree distribution for the BIND dataset

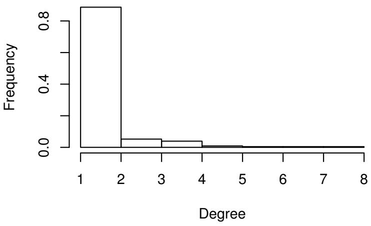

Figure 6. Degree distribution for the KEGG dataset

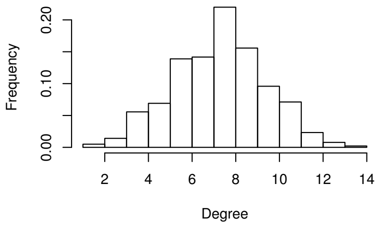

Figure 7. Degree distribution for the ASTRAL dataset

cal pathways from the well-known KEGG database [12] (downloaded on Feb 28, 2007). We use this dataset to demonstrate the application of TALE for biological pathways analysis. A pathway is a directed graph with nodes representing cellular entities such as proteins and regulatory elements, and edges representing their interactions. The graph shows the sequence of actions that lead to different cellular entities interacting to carry out some function. Similar to PINs, each interaction in a pathway is based on an experimentally observed phenomenon. Therefore, pathway data are often noisy and incomplete. Nature is very effective at modularizing complex actions and reusing subcomponents. There are a lot of common building blocks in the cellular machinery that often get "reused" in different pathways. By investigating the similarities shared by different pathways, a biologist can generate various hypotheses that can help refine the understanding of a pathway of interest.

ASTRAL Dataset: To demonstrate the potential application of TALE for Protein Structure Matching, we use the ASTRAL [8] dataset (version 1.71). This dataset contains the 3D structures of protein domains. A domain is an independent, self-stabilizing unit of a protein, usually pertinent to the function of the protein they belong to. In biology, structure similarity is often a good indicator of function similarity. 3D structures can be translated into contact graphs, and structure matching can be achieved by approximate subgraph matching on the corresponding contact graphs. In a contact graph, nodes represent amino acids (since there are 20 different kinds of amino acids, there are 20 distinct node labels) and edges indicate that the corresponding amino acids physically interact with each other. This physical interaction is usually decided by a threshold of the contact distance. In our experiment, we used the widely used $7\mathring{A}$ threshold [7] to convert each domain 3D structure into a contact graph.

6.2 Parameterizations

In this section, we demonstrate how to choose the values of the parameters used in TALE for the three experimental datasets. TALE requires the setting of the following three parameters: the neighbor array size S_{bit} in the NH-Index, the approximation ratio ρ, and the fraction of important nodes P_{imp} in a query graph.

The size of the neighbor bit array is related to the number of node labels (or the number of group labels when allowing node mismatches, cf. Section 4.5.1) in an application. For protein structure matching, there are only 20 amino acids. Two nodes of the contact graphs can be matched only if they represent the same amino acid. Therefore, we set S_{bit} to be 32 (to make it fit in an 32-bit integer), and we use the exact value of the amino acids to set the bit array (instead of using a hash function).

For pathway analysis and protein interaction networks comparison, we need to match nodes based on function or sequence similarity. For the KEGG dataset, we utilize the KEGG Orthologous group (which classifies proteins based on function similarity). Two nodes can be matched only if they belong to the same KEGG Orthologous group. For the BIND dataset, we used CD-HIT [11] to cluster the proteins based on their sequence similarity[1]. And two nodes can be matched only if they belong to the same cluster. There are totally 8814 KEGG Orthologous groups and 22311 CD-HIT clusters. We set S_{bit} to be 64 and 96 for KEGG and BIND datasets, respectively. In fact, we have experimented with other S_{bit} values, and there is no significant difference in performance for different S_{bit} values. Therefore, in the interest of space, we do not show the effect of different neighbor array sizes.

The approximation ratio ρ indicates the percentage of neighbors of a query node that can have no corresponding matchings in the neighborhood of a database

[1] Proteins in each cluster share at least 40% sequence identity.

Table 1. PINs of human, mouse and rat

	# nodes	# edges
human	8470	11260
mouse	2991	3347
rat	830	942

node. It is related to the similarity requirement of a specific application. For simplicity, we set this parameter to be 25% for all three applications.

The fraction of important nodes in a query graph P_{imp} is highly associated with the graph properties in an application. As we use degree centrality to measure the importance of a node, we study the degree distributions of the 3 applications. The representative degree distributions of the 3 datasets are shown in Figures 5 through 7. The degree distribution for the KEGG and BIND datasets is highly right skewed. In fact, studies have shown that both pathways and PINs show power-law degree distribution [14]. Only very small fraction of nodes have high degrees. We set $P_{imp} = 15\%$ for the KEGG and BIND datasets. For the ASTRAL dataset, the degree distribution is bell shape. Around 25% of nodes have degree more than 8, which we consider as important nodes ($P_{imp} = 25\%$).

6.3 Effectiveness Evaluation

In this section, we present results examining the effectiveness of TALE. We also compare TALE to C-Tree [6], SAGA [19], and Graemlin [4].

6.3.1 Protein Interaction Networks Comparison

Graph matching techniques are used on PINs to find conserved components shared between the query network and each network in the database. The PIN for a well studied species is usually a large graph with hundreds to thousands of nodes and edges. C-Tree [6] is not applicable for comparing PINs as the implementation does not allow node mismatches (nodes with different labels to be matched), which is a requirement for this application. On the other hand, TALE handles node mismatches by utilizing the group labels produced by existing protein clustering tools (see Section 6.2). SAGA can be used for querying PINs, but querying such large graphs using SAGA is prohibitively expensive.

For comparing PINs, the tools most closely related to TALE are NetworkBlast [16], MaWISh [13] and Graemlin [4]. Since these tools largely deal with pairwise comparison, we only focus on pairwise PIN comparison in this experiment. In [4], the authors showed that Graemlin is better at identifying conserved functional modules than the other methods. Therefore, we only compare TALE with Graemlin.

Table 2. Effectiveness for comparing PINs

	# KEGGs hit	KEGG coverage	time (sec)
rat vs. human			
Graemlin	0	NA	910.0
TALE	6	3.2%	0.3
mouse vs. human			
Graemlin	18	5.0%	16305.5
TALE	42	13.6%	0.8

We choose the PINs of three well studied mammals: human, mouse and rat for this experiment. The statistics for these three networks are described in Table 1.

We use both TALE and Graemlin (using code download from `http://graemlin.stanford.edu/`) to query the rat and the mouse PINs against the human PIN. We compare the two methods using the effectiveness measures: the number of KEGGs hit and the average KEGG coverage as proposed in [4]. The number of KEGGs hit is the number of pathways in the KEGG database [12] aligned between 2 species. A KEGG pathway is considered as a hit if at least 3 proteins in the pathway are aligned to their counterparts in the pathway of the other species. KEGG coverage is the fraction of proteins aligned within a pathway.

As shown in Table 2, TALE achieves significant larger number of KEGGs hit and better average KEGG coverage than Graemlin. Most noticeable is the big difference in running time. TALE only takes about 1 second for the two queries while Graemlin takes 4.8 hours. In addition, TALE only takes about 1 second to build the index on the human PIN.

Table 3. The statistics of KEGG pathways for the 7 well-studied model species

	#pathways	avg #nodes	avg #edges
human	173	83.3	38.5
mouse	169	83.8	38.3
rat	161	83.6	30.3
fly	103	97.3	12.7
worm	97	100.4	13.4
yeast	87	106.1	20.8
ecoli	95	102.1	25.1
total	885	91.2	28.3

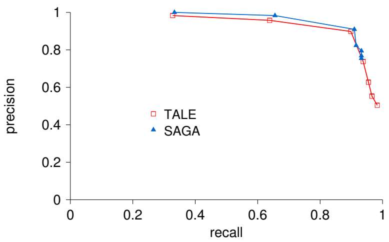

Figure 8. ROC curves for human pathways

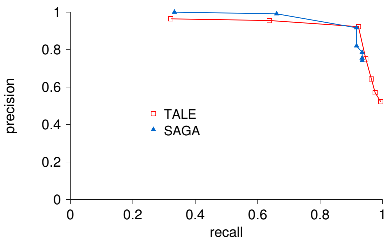

Figure 9. ROC curves for mouse pathways

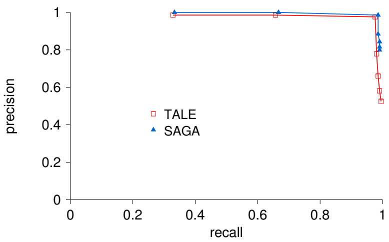

Figure 10. ROC curves for rat pathways

6.3.2 Biological Pathways Analysis

This experiment uses the KEGG pathway dataset. Again since we allow node mismatches in pathway analysis, C-Tree is not applicable. Therefore for this experiment, we only compare TALE with SAGA. SAGA was configured using the parameter settings in [19].

Close related species in evolution (e.g. mouse and rat) share significant similarities in their corresponding pathways. For example, the WNT pathways of mouse and rat are very similar to each other, but are more different from the WNT pathway of fly. This provides us a way to evaluate the effectiveness of TALE for the application of biological pathways analysis.

For this experiment, we choose the pathways of the 7 well-studied model species in KEGG: human, mouse, rat, fly, worm, yeast and ecoli. The statistics of this dataset is summarized in Table 3. Human, mouse and rat are more closely related to each other in evolutionary than the other 4 species. Therefore, we expect the results produced by TALE can reflect the fact that the pathways for these 3 species will be more similar to each other.

We used every pathway for human, mouse and rat to query the database. Note that for some large queries (e.g. human fatty acid biosynthesis pathway with 163 nodes and 151 edges), SAGA could not finish in a reasonable amount of time (taking over 1 hour), while TALE can finish every query within 1.8 seconds. For the queries SAGA can finish within 1 hour, the average running time for SAGA is about 12 seconds, while the average time for TALE is 0.14 seconds.

To evaluate the effectiveness of the results, we employ the measures: recall and precision. Recall is defined as the fraction of the retrieved relevant results out of all the relevant results. Precision is the fraction of the retrieved relevant results out of all the retrieved results. A matching result is considered relevant if it is the same pathway from a species close in evolution. For example, if the query is human WNT pathway, then a relevant result can be human, mouse or rat WNT pathway.

To keep this experiment manageable, we kill any SAGA query if it runs over 1 hour. For fair comparison, we only compare TALE with SAGA for the query results that SAGA can finish within the time limit. We employ the SAGA distance model (using the default parameters in [19]) to rank the results returned by both TALE and SAGA. We compute the average precision and recall values for human, mouse and rat pathway queries. The ROC curves are shown in Figure 8 to Figure 10. SAGA and TALE show very comparable effectiveness for pathway analysis, with SAGA having a slight advantage.

6.3.3 Protein Structure Matching

In this experiment, we evaluate the effectiveness of TALE for protein structure matching using the ASTRAL dataset.

This application generally does not require node mismatches, therefore we can compare TALE with C-Tree. However, the C-Tree implementation that we got from the authors is memory-based. In other words, the whole index needs to reside in memory for query processing. Naturally, as the database size increases, the index will soon grow out of memory. For example, C-Tree cannot build an index on the entire ASTRAL dataset (which has 75626 domains). In contrast, NH-Index is a disk-based index technique and is not limited by the memory size. As we will show in Section 6.4.3, TALE can easily handle the entire ASTRAL dataset, and our disk-based index structure scales nicely with increasing database sizes. For a fair comparison, we employ the similarity model used by C-Tree [6] to rank the matching results.

ASTRAL contains 75626 domains, which are classified into 7275 families. Domains in each family present significant structural similarity. This provides us with a way of evaluating the effectiveness of TALE: large fraction of the top matching results are expected to belong to the same family of the query domain.

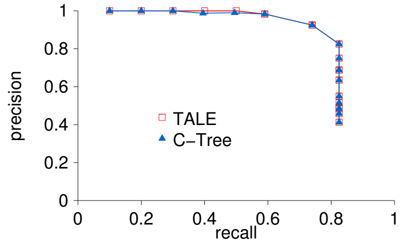

Figure 11. ROC curves using the ASTRAL dataset

We test TALE and C-Tree on a subset of ASTRAL, so that C-Tree can hold the index in memory. The dataset is created as follows: We randomly choose 1300 families (with more than 10 domains in each family), and then randomly choose 10 domains from each family. The average number of nodes and edges for each graph are 186.6 and 734.2, respectively.

We randomly choose 20 queries (with 346.4 nodes and 971.6 edges per graph on average) from the 13000 domains. We gradually increase the number of results returned by TALE and C-Tree, and measure the mean recall and mean precision for both methods. The recall and precision ROC curves are shown in Figure 11. The precision for both methods stays very high until the recall reaches round 0.6. This is because both methods return relevant results as their top results. However, as the recall further increases, the precision drops more steeply. After the recall reaches around 0.8, returning more results will not improve the recall any more. This is because the classification system in ASTRAL is not purely based on structure similarity, but also on extensive domain knowledge. No method based on pure structural similarity is likely to perfectly match this classification system. However, TALE could potentially be used for classifying novel family members in combination with the domain knowledge provided by experts.

Although TALE and C-Tree are very comparable in their effectiveness for this dataset, TALE is faster than C-Tree. The average running time for the 20 queries is 34.8 seconds using TALE, but 61.9 seconds using C-Tree. TALE is almost 2 times faster than C-Tree (even though it is a disk-based implementation and is going through PostgreSQL).

Table 4. Four BIND sub-datasets for the scalability experiment

	#graphs	avg #nodes	avg #edges	index size	index time
D1	10	939.1	1093.2	1.4MB	13.2s
D2	20	938.5	1691.9	2.9MB	31.1s
D3	30	939.5	1920.7	4.5MB	50.4s
D4	40	940.1	1743.6	5.7MB	62.7s

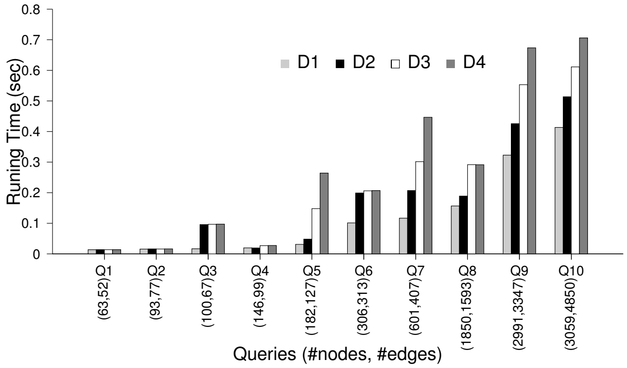

Figure 12. Scalability Experiment using the BIND dataset

6.4 Efficiency and Scalability Evaluation

In this experiment, we test the efficiency and scalability of TALE for the three applications.

6.4.1 Experiment on BIND Dataset

In this experiment, we evaluate the efficiency and scalability of TALE on the BIND dataset. BIND has PINs for 757 species, but most PINs are incomplete. We choose the largest 40 PINs from BIND. The largest graph contains 8470 nodes and 11260 edges. The smallest of these 40 PINs contains 45 nodes and 105 edges. On average, each graph has 940.1 nodes and 1743.6 edges. The characteristic of this data is that it contains large-sized graphs. To measure the scalability of TALE, we formed 4 datasets D1 to D4 with increasing sizes [2]. The statistics of the four datasets are summarized in Table 4. The index sizes and the index construction times are also shown in this table. As the database size increases, the index size grows at a near-linear rate and the index construction time increases steadily.

We choose the 10 graphs in dataset D1 as the queries. For this experiment, we do not restrict the number of results returned by each query. The execution time for the 10 queries on the 4 datasets is shown in Figure 12. Even for the largest query with 3059

[2] The 4 datasets are formed as follows. We first divide the 40 PINs into 4 balanced groups each with 10 PINs and roughly same total number of nodes. We randomly select one group as D1, randomly add another group to D1 to form D2, then randomly add one of the remaining groups to D2 to form D3, finally D4 contains all the 4 groups.

nodes and 4850 edges on the largest D4 dataset, the query executes in about 0.7 seconds. The execution time grows as the size of the database increases. For most queries, the growth ratio shows near-linear trend. Note that query execution time is not just influenced by the query and database sizes, but also by the result cardinality. In Figure 12, Q2, Q3 and Q4 increase in the query size, but the execution time increases from Q2 to Q3 while decreases from Q3 to Q4 for D2, D3 and D4 datasets. The reason is that Q3 has more database matches than Q2 and Q4. (Recall that in this experiment, we do not restrict the number of results returned by each query.) For Q3, there is a jump from D1 to D2, because more matching graphs are found in D2. But the number of matches remain roughly the same from D2 to D4 (and so does the execution time). Similar explanations apply to other queries in this figure.

6.4.2 Experiment on KEGG Dataset

In this section, we test the efficiency and scalability of TALE on increasing sizes of KEGG pathway databases. The smallest dataset contains all the human and mouse pathways. We increase the database size by including pathways of more species until it contains all the 41550 KEGG pathways of 538 species. The index construction time and index sizes for these increasing databases are shown in Figure 13 and Figure 14, respectively. Our indexing technique indexes the neighborhood of each database node. This novel technique gives us the near linear increase in the index construction time and index size as shown in Figure 13 and Figure 14.

To test query execution time, we randomly selected 20 human pathways (76.7 nodes and 29.9 edges per graph on average). For each query, we ran TALE to get the top 20 results. The average running time for the 20 queries on increasing database sizes is reported in Figure 15. As shown in this figure, the query execution time increases steadily with the database size.

6.4.3 Experiment on ASTRAL Dataset

In this experiment, we evaluate the efficiency and scalability of TALE on the ASTRAL datasets with increasing sizes. The smallest dataset contains 200 graphs, while the largest one contains all the 75626 graphs in ASTRAL. As shown in Figure 16 and Figure 17, the index construction time and index size show steady growth with increasing database size.

We randomly selected 20 queries (153.1 nodes and 592.0 edges per graph on average) from the smallest dataset, and ran it on the increasing sized databases. For each query, we only retain the top 20 results. The average execution time for the 20 queries is shown in Figure 18. The running time scales nicely with the database size.

6.5 Discussion and Summary

We note that TALE is a heuristic algorithm. It does not guarantee that it will find the best or all matches. However, given that finding the best/all matches is NP-hard [2] and infeasible in practice, heuristics are inevitable. For most real graphs, our heuristics achieve high accuracy compared with existing tools, as shown in our experiments.

In this work, we have used degree centrality to measure the importance of nodes. To show the effectiveness of this measure, we compare TALE to a variant called TALE-Random, where the "important" nodes are simply a randomly selected subset of the nodes. We ran the BIND mouse vs human test (Table 2, Row 3) using TALE-Random. We compare the number of matching nodes, the number of matching edges, the number of KEGGs hit and the average KEGG coverage for the two methods. The results are 106, 61, 42, 13.6% for TALE and 85, 24, 8, 5.8% for TALE-Random. This test shows the effectiveness of this node importance measure for this application.

To summarize the experimental section, our extensive empirical evaluation demonstrates the effectiveness, efficiency and scalability of TALE. We have compared TALE to three existing tools, SAGA, C-Tree and Graemlin. TALE is a flexible tool and the only tool that can easily be applied across the three applications considered in our evaluation. Furthermore, TALE produces useful and meaningful results for all the three applications, and is also significantly faster than these existing tools. Our results also show that TALE is scalable for large queries and large databases.

7 Conclusions and Future Work

In this paper we have presented TALE – an approximate subgraph matching tool for matching graph queries with a large number of nodes and edges. TALE employs a novel indexing technique, which achieves a high pruning power and scales linearly with the database size. This index structure can be easily implemented in existing relational systems. The innovative matching algorithm used by TALE distinguishes nodes by their importance to the graph structure. This algorithm first matches the important nodes in the query, and then extends them to produce larger graph matches. TALE is a general tool for approximate subgraph matching queries, and can be easily customized to meet the requirement of different applications. Our

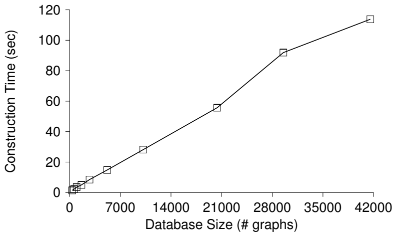

Figure 13. Index Construction Time with Increasing KEGG Database Size

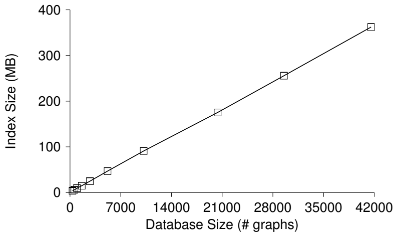

Figure 14. Index Size with Increasing KEGG Database Size

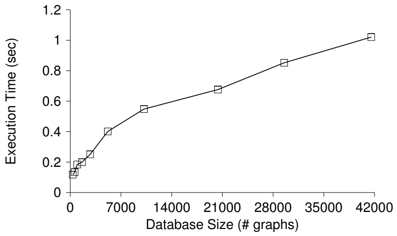

Figure 15. Query Execution Time with Increasing KEGG Database Size

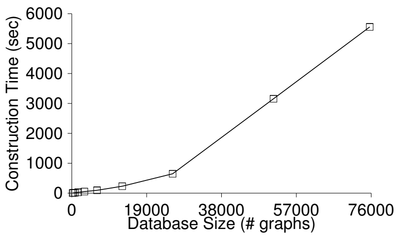

Figure 16. Index construction time for the ASTRAL dataset

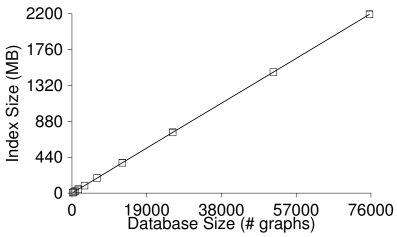

Figure 17. Index size for the ASTRAL dataset

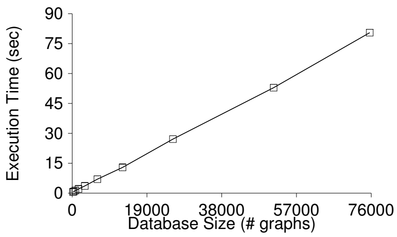

Figure 18. Query execution time for the ASTRAL dataset

empirical evaluations demonstrate the improved effectiveness and efficiency of TALE over existing methods. As part of future work, we plan on applying TALE to other applications, such as social networks and RDF graph datasets, to further evaluate the generality of TALE.

Acknowledgment

This research was primarily supported by the National Science Foundation under grant DBI-0543272, the National Institutes of Health under grant 1-U54-DA021519-01A1 and by an unrestricted research gift from Microsoft Corp.

References

[1] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[2] H. Bunke. On a relation between graph edit distance and maximum common subgraph. *Pattern Recogn. Lett.*, 18(8):689–694, 1997.

[3] C. Alfarano et al. The biomolecular interaction network database and related tools 2005 update. *Nucleic Acids Res.*, 33:D418–D424, 2005.

[4] J. Flannick, A. Novak, B. S. Srinivasan, H. H. McAdams, and S. Batzoglou. Græmlin: General and robust alignment of multiple large interaction networks. *Genome Res.*, 16:1169–1181, 2006.

[5] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1979.

[6] H. He and A. K. Singh. Closure-tree: an index structure for graph queries. In *ICDE*, 2006.

[7] J. Hu, X. Shen, Y. Shao, C. Bystroff, and M. J. Zaki. Mining protein contact maps. In *BIOKDD*, 2002.

[8] J. Chandonia et al. The astral compendium in 2004. *Nucleic Acids Res.*, 32:D189–D192, 2004.

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

[10] B. P. Kelley, B. Yuan, F. Lewitter, R. Sharan, B. R. Stockwel, and T. Ideker. Pathblast: a tool for alignment of protein interaction networks. *Nucleic Acids Res.*, pages W83–W88, 2004.

[11] W. Li, L. Jaroszewski, and A. Godzik. Clustering of highly homologous sequences to reduce the size of large protein database. *Bioinformatics*, 17:282–283, 2001.

[12] M. Kanehisa et al. The kegg resources for deciphering the genome. *Nucleic Acids Res.*, 32:D277–D280, 2004.

[13] M. Koyuturk et al. Pairwise alignment of protein interaction networks. *Journal of Computational Biology*, 13(2):182–199, 2006.

[14] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45:167, 2003.

[15] P. O'Neil and D. Quass. Improved query performance with variant indexes. In *SIGMOD*, 1997.

[16] R. Sharan et al. Conserved patterns of protein interaction in multiple species. *PNAS*, 102:1974–1979, 2005.

[17] D. Shasha, J. T.-L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *PODS*, 2002.

[18] E. Sprinzak, S. Sattath, and H. Margalit. How reliable are experimental protein-protein interaction data? *Journal of Molecular Biology*, 327(5):919–923, 2003.

[19] Y. Tian, R. C. McEachin, C. Santos, D. J. States, and J. M. Patel. SAGA: a subgraph matching tool for biological graphs. *Bioinformatics*, 23(2):232–239, 2007.

[20] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.

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

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

[23] X. Yan, P. S. Yu, and J. Han. Substructure similarity search in graph databases. In *SIGMOD*, 2005.

[24] X. Yan, F. Zhu, J. Han, and P. S. Yu. Searching substructures with superimposed distance. In *ICDE*, 2006.

[25] S. Zhang, M. Hu, and J. Yang. Treepi: A new graph indexing method. In *ICDE*, 2007.