Graph Database Indexing Using Structured Graph Decomposition

[1]David W. Williams, [2]Jun Huan, [1]Wei Wang
[1]*Department of Computer Science*
University of North Carolina, Chapel Hill
[2]*Department of Electrical Engineering and Computer Science*
University of Kansas
[1]*{dwwill, weiwang}@cs.unc.edu,* [2]*jhuan@ittc.ku.edu*

Abstract

We introduce a novel method of indexing graph databases in order to facilitate subgraph isomorphism and similarity queries. The index is comprised of two major data structures. The primary structure is a directed acyclic graph which contains a node for each of the unique, induced subgraphs of the database graphs. The secondary structure is a hash table which cross-indexes each subgraph for fast isomorphic lookup. In order to create a hash key independent of isomorphism, we utilize a code-based canonical representation of adjacency matrices, which we have further refined to improve computation speed. We validate the concept by demonstrating its effectiveness in answering queries for two practical datasets. Our experiments show that for subgraph isomorphism queries, our method outperforms existing methods by more than an order of magnitude.

1. Introduction

A graph describes relationships over a set of entities. With node and edge labels, a graph can describe the attributes of both the entity set and the relation. Labeled graphs appear in many research domains such as drug design [1], protein structure comparison [2], video indexing [3], and web information [4].

Only recently have graph data management and mining techniques attracted significant research attention in the database community. For example, more than 10 algorithms for mining recurring patterns in graph databases [5] were developed in the past 5 years. In addition, a growing body of research focuses on graph database queries, which can be roughly divided into two classes: pattern matching and similarity search.

In pattern matching, one determines the set of graphs in a graph database which match a query pattern P. The term match is made specific by using a matching condition to decide if a pattern subgraph occurs in a graph. The most commonly used matching condition is the subgraph isomorphism test, which determines that a pattern P matches a graph G if and only if P is a subgraph of G [6].

In similarity search, one looks for graphs in a database which are similar to a query graph. Here, the term similarity is defined by a specific procedure. Graph edit distance [7] is a common way to measure the (dis)similarity between two graphs, though other measures exist [8].

Graph similarity search can be divided into two subgroups. A K-NN query reports the K graphs in a graph database that are most similar to the query, while a range query determines all graphs in the database whose similarity score is within a user-specified tolerance.

In this paper, we focus on deriving efficient index structure for graphs with limited sizes. The applications of small graphs include:

- Protein structure motifs. A structural motif is a recurring geometric arrangement of amino acid residues in proteins. In general, structural motifs have 4 to 6 residues. Recent work has discovered millions of such motifs. Therefore, automated structural motif recognition within protein structures is important for a number of applications, including protein function prediction [9].
- Chemical structures. Most of the chemical structures, such as those from the NCI/NIH AIDS antiviral screen test [10], have only a handful of nodes and edges. As studied in [5], most of these chemical graphs are sparse and the majority of recurring components are tree or tree-like patterns. Pattern matching and similarity search in chemical databases are important for drug design.

1-4244-0803-2/07/$20.00 ©2007 IEEE.

1.1. Challenges

Though studied previously, the search for efficient methods of answering graph queries remains open. Essential problems include: (1) how to store graph databases efficiently, (2) how to define similarity between graphs, and (3) how to create efficient index structure to accelerate pattern matching and graph similarity search.

A primary challenge in pattern matching is that pair-wise comparisons of graphs are usually hard problems. Subgraph isomorphism is known to be NP-complete [11]. As a result, most meaningful definitions of similarity will result in NP-hard problems. Even a relatively simple comparison, graph isomorphism, defies a polynomial bound for the general case [12]. These costly pair-wise comparisons, when combined with the increasing size of modern graph databases, makes finding efficient search techniques difficult. Even more complications arise from the many classifications of graphs and the fact that some techniques apply only for a specific class.

Furthermore, the application of graphs to a wide variety fields implies that graphs themselves have a wide variety of interpretations. This poses a challenge when defining useful similarity measures, even when restricted to a single domain, such as chemical structures [13]. As mentioned earlier, edit distance is a commonly used metric, but other metrics, such as one based upon maximal common subgraphs, may be more meaningful [8].

1.2. Our Solution

This paper presents the use of a novel graph decomposition procedure which facilitates the processing of subgraph isomorphism and similarity queries for graph databases. Graph decomposition is the process of deriving component parts from a graph by utilizing a given set of operations. Several graph decomposition schemes have been developed, including clique decomposition, modular decomposition, and NLC decomposition [14].

Previous work has typically demonstrated its effectiveness on sparse graphs or planar graphs with unlabeled edges. The techniques we present in this paper, although applicable to any small graph, are notably effective for processing dense graphs with labeled edges. The algorithms presented herein quickly identify sub-isomorphic relationships between the database graphs, allowing for more compact indexes when the graphs have a high degree of similarity. Moreover, our indexing scheme is particularly well-suited to answer subgraph isomorphism queries using far less computation time than other methods.

1.3. Related Work

Several recent research efforts have focused on preprocessing graph databases with the goal of improving query times.

B.T. Messmer et al. proposed a decision-tree approach for indexing models for isomorphism and subgraph isomorphism [3]. This method generates answers in polynomial time, at the cost of an index which is exponential in size with respect to database size.

GraphGrep, developed by Shasha et al., indexes graphs by enumerating paths through each graph in a database [15]. GraphGrep has a notable advantage in that its index is not exponential with respect to graph size. However, it is exponential with respect to path length, which is a primary factor in the power of the index.

Yan et al. introduced a method for indexing subgraph isomorphism queries based on selectively using frequently occurring subgraphs as features [16]. By enumerating subgraphs instead of paths, their filtering methods are more selective. When compared with GraphGrep, their search engine, gIndex, achieves significant improvements in both index size and query time. Recently, they have extended the concept to produce an engine named Grafil, which processes similarity queries [17].

He et al. indexed graphs using a novel data structure, a closure tree, also referred to as a C-tree [7]. C-trees use graph closures, which is a form of bounding box for graphs. The usage of C-trees closely parallels that of R-trees.

Another approach, used by Srinivasa et al., uses multiple reference techniques, including an allowance for user-defined schema [18,19]. The result is a flexible database system, GRACE, which enables users to limit query search spaces to achieve faster results.

In addition to the more generalized work on graph indexing, graph decomposition has been directly applied to answering questions of isomorphism and subgraph isomorphism. Graph decomposition into trees has been used for these problems, notably for planar graphs [20,21]. In the case of database indexing, Wang et al. have detailed methods for matching three-dimensional graphs with model graphs, via decomposition to rigid structures [22].

1.4. Organization

This paper is organized as follows. In Section 2, we present the preliminary concepts of labeled graphs, subgraph isomorphism, graph canonical representations, and subgraph mismatch scores. Readers familiar with these concepts may skip the section. Section 3 presents our graph index. Section 4 and Section 5 apply our graph index for subgraph query and similarity search,

respectively. In Section 6, we outline our experimental results with a real-world graph database and a synthetic database. Section 7 concludes our paper.

2. Preliminary Concepts

This section presents the key concepts, notations, and terminology used in this paper, which include: labeled graphs, subgraph isomorphism, graph canonical representation, and subgraph mismatch score.

2.1 Labeled Graphs

A *labeled graph* is a four-element tuple $G = (V, E, \Sigma, \lambda)$ where V is a set of vertices and $E \subseteq V \times V$ is a set of undirected edges joining two distinct vertices. Σ is the set of vertex and edge labels and $\lambda : V \cup E \rightarrow \Sigma$ maps vertices and edges to their labels. The *size* of a graph is the number of its vertices. A *graph database* is a collection of labeled graphs. Figure 1 provides an example of a graph database composed of four graphs, $\{G_1, G_2, G_3, G_4\}$.

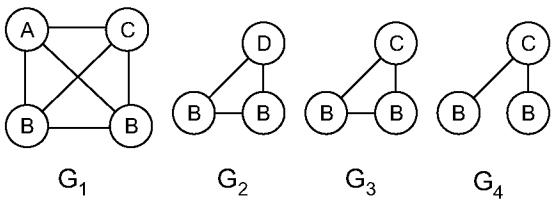


Figure 1. A graph database.

2.2. Subgraph Isomorphism

Given two graphs G, G' we define that the graph $G=(V, E, \Sigma, \lambda)$ is *subgraph isomorphic* to $G'=(V', E', \Sigma', \lambda')$, denoted by $G \subseteq G'$, if there exists a 1-1 mapping $f: V[G] \rightarrow V[G']$ such that:

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

where V and E are the vertex and edge sets of the graph G, respectively. E' is the edge set of G'. The mapping f is defined as a *subgraph isomorphism* from G to G'. A graph G is an *induced subgraph* of G' if $G \subseteq G'$ and G preserves all edges in G', i.e. $E[G] = V[G] \times V[G] \cap E[G']$.

In Figure 1, G_3 and G_4 are subgraph isomorphic to G_1, but G_2 is not subgraph isomorphic to G_1. Further, G_3 is an induced subgraph of G_1, while G_4 is not.

2.3. Graph Canonical Form

We represent a graph G by an adjacency matrix M. Slightly different from an unlabeled graph, a diagonal entry of M in our representation is the label of the corresponding vertex in G and every off-diagonal entry is the label of the corresponding edge in G, or zero if there is no edge. For graphs with unlabeled edges, a one denotes that the edge exists, while a zero indicates it does not.

Given an $n \times n$ adjacency matrix M of a graph with n vertices, we define the *code* of M, denoted by code(M), as the sequence of lower triangular entries of M (including the diagonal line) in the order:

$$M_{1,1}, M_{2,1}, M_{2,2}, \ldots, M_{k,1}, M_{k,2}, \ldots, M_{k,k}, \ldots, M_{n,n}$$

where $1 \le k \le n$ and $M_{i,j}$ is the entry at the ith row and jth column in M. We assume the rows (columns) in M are numbered 1 through n from top to bottom (from left to right). For a graph G, the canonical code of G, denoted by $\varphi(G)$, is the maximal code among all of its possible codes, when the codes are compared lexicographically. This canonical representation is used in [23].

Assuming a lexicographical ordering B > C > 1 > 0, the possible codes for G_4 in Figure 1 are B1C01B, B0B11C, and C1B10B. Of these, B1C01B is the canonical code.

2.4. Subgraph Mismatch Score

Given two simple graphs g and g', we can construct two equivalent complete graphs G,G' in which we use edges with null labels to represent missing edges in g and g'.. Given G, G',, and an injective mapping $f: V[G] \rightarrow V[G']$, we define the *mapping-induced subgraph mismatch score* from G to G' as:

$$d_f(G, G') = \Sigma[\lambda(u) \neq \lambda'(f(u))] + \Sigma[\lambda(u,v) \neq \lambda'(f(u,v))], \quad \forall\ (u \in V[G], v \in E[G]).$$

That is, the mapping-induced subgraph mismatch score is the number of mismatched vertex and edge labels under a mapping. The *subgraph mismatch score* from G to G', denoted as $d(G, G')$ is the minimal mapping-induced subgraph mismatch score for all possible mappings. Note that, if the size of G is larger than the size of G', no injective mapping exists. In this case, the subgraph mismatch score from G to G' is infinite.

In Figure 1, $d(G_2, G_1) = 1$, $d(G_3, G_2) = 1$, $d(G_4, G_3) = 1$, and $d(G_3, G_1) = 0$.

3. Graph Decomposition

For the remainder of this paper, we use the term *graph decomposition* to denote the enumeration of all connected, induced subgraphs of a given graph. Each enumerated subgraph is unique with respect to isomorphism; only the canonical instance is enumerated for each automorphism group. Note that any graph decomposition will contain both the original graph, as well as a null graph with no vertices or edges.

A graph of size n decomposes into at most 2^n subgraphs. This occurs in the case of a complete graph in which each of the vertices has a unique label. Due to isomorphism, a complete graph with multiple occurrences of the same label may decompose into fewer subgraphs. If all labels are identical, a complete graph of size n decomposes into just n+1 subgraphs (one for each size, size = 0 to n).

3.1. Notations

G	A graph database
P, Q, G	Single graphs
$\varphi(G)$	The canonical code of G
$d(P, Q)$	Subgraph mismatch score between P and Q
GDI	Graph decomposition index
$\mathcal{H}$	The hash table in GDI
DAG	Directed acyclic graph

3.2. Graph Decomposition DAG

We construct a Directed Acyclic Graph (DAG) to describe the results of a graph decomposition of a graph G in the following way.

- Each node is a subgraph P of G
- For two nodes P and Q, there is a directed link from P to Q if
  - $P \subset Q$ and
  - There exists no graph P' such that $P \subset P' \subset Q$.

We call such constructed DAG the *graph decomposition DAG*. In the graph decomposition DAG of a graph G, there is always one node that represents G, and one node that represents the null graph. The *children* of a node P are all graphs Q for which (P, Q) is a directed link in the DAG. The *descendents* of a node P are all nodes that are reachable from P in the DAG.

For brevity, we use the term *node* to refer to a node within a DAG, and the term *vertex* to refer to a component of a graph that is being decomposed. Similarly, we use the term *link* to denote a connection between nodes in a DAG, and the term *edge* to denote a connection between vertices. Further, nodes have the properties of their represented graphs. Namely, nodes have a size which is equivalent to the size of their represented graph. In addition, a node is considered a subgraph of any node for which the relation holds with respect to their represented graphs.

Figure 2 provides an example of the graph decomposition DAG of a graph. For purposes of illustration, we use a complete graph without edge labels. Each node in the graph decomposition DAG is labeled with a shorthand notation which denotes the vertex labels of its represented graph.

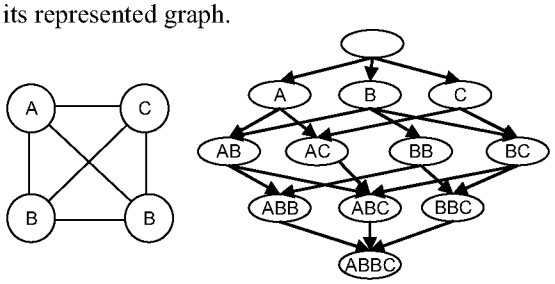


Figure 2. Decomposition of a complete graph.

A decomposition DAG of an n-sized graph is described as being composed of n+1 tiers. The m^{th} tier is comprised of all nodes of size m. The 0^{th} tier always contains one node, which is the null graph. Similarly, the n^{th} tier also contains one node, which represents the entire original graph. The combinatorial nature of the decomposition leads to the $(n/2)^{th}$ tier being the largest, and organizing the DAG into horizontal rows based on tiers results in a roughly diamond pattern.

3.3. Hashing Graph Decompositions

We use a hash table to index the subgraphs enumerated during graph decomposition. To hash a graph, we compute the canonical form of its adjacency matrix. The hash key is then determined from the string given by the canonical code. Using this method, all isomorphic graphs produce the same hash key. All entries in the hash table are in canonical form, and only one entry is made for each unique canonical code.

The hash table enables a lookup function to quickly locate a node in the decomposition DAG which is isomorphic to a query graph, if it exists. Doing so is a two-step process. First, we compute the hash key from the query's canonical code. From this, we obtain candidate matches and their canonical codes. In the second step, we verify candidate canonical codes with the query's canonical code. If the codes match exactly, then this indicates that the candidate is an isomorphic match to the query graph.

The validation step described above is necessary for two reasons. First, hash keys are not guaranteed to be unique to a given canonical code; canonical codes are

sequences of variable length and thus cannot be uniquely mapped to hash keys represented by far fewer bits. Second, the size of the hash table is limited to an a priori estimate of the number of entries it will contain. Therefore, mapping hash keys to the hash table may cause several graphs to be within a single table entry. Comparing canonical codes removes this ambiguity.

3.4. Graph Decomposition Indexes

While a graph decomposition DAG can represent the decomposition of a single graph, it can also be applied to a collection of graphs in order to provide an indexing structure. A Graph Decomposition Index (GDI) contains two indexing structures. The first structure is a graph database DAG (or simply a DAG) which is merged from the graph decomposition DAGs of all the database graphs. The second structure is a hash table that cross-references nodes in the database DAG. Algorithm 1 outlines a method for constructing a GDI from a database of graphs.

Algorithm 1. GDI Construction

```
Construct(G)
  H := ∅
  DAG := ∅
  for each G ∈ G do
     V[DAG] := V[DAG] ∪ {G}
     H[φ(G)] = G
     Decompose(G, GDI, H)
  end do
return (GDI, H)

Decompose(G, DAG, H)
  for each v ∈ V[G] do
     G' := G – v
     V[DAG] := V[DAG] ∪ {G}
     E[DAG] := E[DAG] ∪ {(G', G)}
     H[φ(G')] = G'
     decompose(G', DAG, H)
  end do
```

Here we use the notation $G' := G - v$ to denote an operation that creates a new graph G' from G by deleting a node v and all the edges incident with v in G.

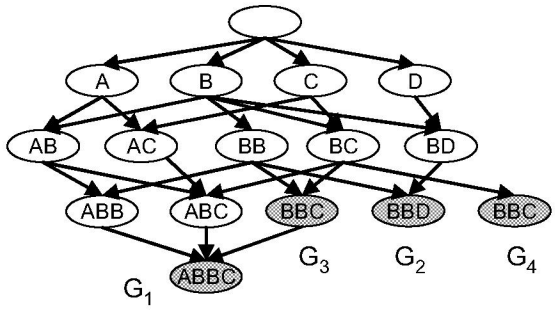


Figure 3. A graph database DAG.

Figure 3 illustrates the DAG index constructed for the graph database illustrated in Figure 1. In the illustration, nodes that represent the four original database graphs are shaded. The other nodes in the DAG are used to index those graphs.

The total number of nodes in the GDI is bounded by $O(k2^n)$, where n is the maximum size of any database graph and k is the number of graphs in the database. Any given node of size m has at most m parents, each corresponding to the removal of a different vertex. This limit bounds the maximum number of links in the GDI. Since any node in the GDI can be stored using a reference to database graph and the subset of the m vertices which it includes, a node can be stored in $O(m)$ space. Therefore, the space overhead of the DAG index is $O(kn(2^n))$.

Although the index requirement is exponential in the general case, a limit placed on the size of the database's graphs results in a space requirement which is linear with respect to database size. In addition, any subgraphs common to more than one database graph are instantiated only once, which can substantially reduce the size of the index. Thus, a GDI can be used for any database containing a large number of relatively small graphs, especially when these graphs have substantial overlap.

4. Subgraph Isomorphism Queries

Algorithm 2 sketches a method to answer a subgraph isomorphism query quickly through the use of a pre-computed GDI. From the definition of subgraph isomorphism, the query must be isomorphic to a subgraph of each graph in the answer set. Therefore, it is sufficient to locate the node in the GDI which is isomorphic to the query, and report all descendents of the node which correspond to database graphs. When there is no matching to the query, the answer set is empty.

Algorithm 2. Subgraph Isomorphism Query

```
SubGraph Isomorphism Search(G)
  ans := ∅
  visited := ∅
  v := H(φ(G))
  if v exists then Visit(v, ans, visited)
  return ans

Visit(v, ans, visited)
  visited := visited ∪ {v}
  if v represents a database graph G then
    ans := ans ∪ {G}
  endif

  for each child u of v do
   if u ∉ visited then Visit(u, ans)
  end for
```

5. Similarity Queries

The answer set of a similarity range query is the set of all database graphs for which the subgraph mismatch score from the query graph to the database graph is less than or equal to the query's range. Based upon our definition of subgraph mismatch score in Section 2.4, vertex and edge substitutions are allowed at unit cost. Any number of vertices, along with their associated edges, may be added to the query at no cost. However, no vertices may be deleted. For this reason, similarity queries can be conceptualized as being determined by subgraph similarity; subgraph mismatch score is the minimum number of vertex and edge mismatches between the query graph and any identically-sized subgraph of a candidate graph, over all possible mappings.

The algorithms presented herein process similarity queries using only comparisons of graphs of equal size. This is accomplished by decomposing the query itself and sequentially comparing its component subgraphs with subgraphs stored in the GDI. The goal of the search is to identify the set of all nodes which are of the same size as the query graph and within the specified range. Once the algorithm identifies this set of nodes, it visits the nodes sequentially and reports any database graphs that they or their descendents represent.

Due to the inherent NP-hardness of the problem, it may be necessary to perform an exhaustive search in order to find an optimal graph mapping. This exhaustive search can become computationally prohibitive. Approximate mapping techniques, such as the neighbor-based mapping algorithm [7], can alleviate this problem. However, our work has been focused on guaranteeing complete answers.

Fortunately, in the case of near-neighbor searches, where the query range is relatively small, exhaustive search for the optimal mapping is not required. Thus, we present two algorithms that process similarity queries. The first is an efficient algorithm for near-neighbor searches. The second algorithm processes similarity queries for any range. Finally, we outline an additional approach for far-neighbor searches. .

5.1. Near-Neighbor Queries

Given a similarity query with range d and size s, we define it as a near-neighbor query if d is less than s. When the relation $d < s$ holds, any graph in the answer set must share a subgraph in common with the query. Further, this common subgraph must be of at least size s-d.

As an informal proof of this fact, observe that for any query graph P and answer graph G, there exists an optimal mapping for which the subgraph mismatch score from P to G is less than or equal to d. From our definition of subgraph mismatch score, there exist at most d mismatched vertex/edge labels. Any chosen mismatch can be eliminated by removing a single mapped pair of vertices from P and G. Iteratively eliminating mismatches in this manner removes at most d mapped pairs of vertices, leaving subgraphs of P and G, which are isomorphic and of minimum size s-d.

Algorithm 3 presents a "quick start" method that utilizes common subgraphs to quickly search for near-neighbors. First, it decomposes the query into a query DAG (and its related hash table). It then locates matches for the query's subgraphs of size s-d in the decomposition DAG, using any such matches and their known isomorphic mappings as a basis for the search.

From each isomorphic node in the decomposition DAG, the algorithm performs a depth-first search by testing the mapping-induced subgraph mismatch score from all children to all of the children of the corresponding node in the query DAG. When a pair-wise score is within the query's range, the search continues by advancing to the next tier. Any path that progresses past the s^{th} tier indicates that an answer has been found. In this case, the algorithm records the GDI node for later visitation.

Whenever the depth-first search reaches a pair of nodes on the s^{th} tier, or when there are no remaining pairs of children to compare, the search algorithm backtracks. It returns to the previous tier and continues comparing that tier's children were it had left off.

When determining the subgraph mismatch score between two children, Algorithm 3 avoids an exhaustive search for the optimal mapping. At each step, the mapping between the children is determined by using the mapping between their parents. The new vertices, one of

which belongs to each child, form a new mapped vertex pair that is added to the previous mapping. Because the initial mapping is given from an isomorphic match, the result is that the mapping is always known.

If graphs are decomposed without testing for connectivity (or in the case that database contains only complete graphs), the completeness of the answer is guaranteed, due to the redundancy present in the DAG. Observe that from any node to any of its descendents, there exists a path which represents the addition of vertices for any ordering of those vertices. Hence, from a given node, Algorithm 3 considers all possible mappings for new vertices. Further, since it uses every subgraph of the query of size *s-d*, it considers all mappings between each candidate answer and the query which are necessary to produce a complete answer.

Algorithm 3. Near-Neighbor Similarity Query

```
nnSearch(G, d)
  vis := ∅
  qDAG := ∅
  qℋ := ∅

  decompose(G, qDAG, qℋ)
  for each Q in qDAG and |Q| = |G| – d do
      P := ℋ(φ(Q))
      if P exists and d(P, Q) <= d then
          DFS(P, Q, G, d, vis)
      end if
   end for

  ans := ∅
  for each v in vis do
    visited := ∅
    Visit(v, visited, ans)
  end do
  return ans

DFS(P, Q, G, d, vis)
  if Q = G then
      vis := vis + {P}
      return
  else
    for each (P', Q') ∈ C[P] × C[Q] do
        if d(P', Q') <= d then
          DFS(P', Q', G, d, vis)
        end if
    end for
  end if
```

We use $C[G]$ to denote the children of a node G in the DAG of the related GDI index. The function d computes the subgraph mismatch score of graphs, as defined in Section 2.4.

5.2. Queries for Greater Ranges

When query range exceeds the query size, then there is no guarantee that a graph in the answer set has an induced subgraph that is identical to a subgraph of the query. Thus, the information stored in the GDI is of less use for such queries. This is an inherent difficulty when applying index structures to answer similarity queries. In addition, the larger mismatch tolerances mean that branch-and-bound techniques are able to prune fewer possibilities when searching for the optimal mapping between candidates and the query.

Algorithm 4 shows a method to compute the complete answer set for queries of any range. The basic approach is to compare the query to all nodes in the GDI that are of identical size. A depth-first branch-and-bound search is used to find a mapping which meets the range requirement. If such a mapping is found, then the GDI node is visited to expand the answer set. Using this method, it is possible to report multiple answers from a single comparison of two small graphs.

Algorithm 4. General Similarity Query

```
SimilaritySearch(G, d)
  ans := ∅
  visited := ∅
  for each G' in DAG such that |G'| = |G| do
    F := ∅
    if match(G, G', F, d) then
      Visit(G', ans, visited)
    end if
  end do
  return ans

Match(G, G', F, d)
  If ( G := ∅) then
    return true
  end if
  for each (u, v) ∈ V[G] × V[G'] do
      F := F ∪ {(u,v)}
      if ( d_F(G, G') ≤ d ) then
        P := G – u
        Q := G' – v
        if ( Match(P, Q, F, d) ) then
          return true
        end if
      end if
      F := F - {(u,v)}
    end do
    return false
```

We construct an injection f from F by letting $f(u) = v$ for all $(u, v) \in F$. Here, d_F is the mapping-induced subgraph mismatch score, defined in Section 2.4.

5.3 Queries for Far-Neighbors

It is possible to optimize searches for far-neighbor searches. For example, a query might be interested in locating all database graphs which share no vertex labels in common with the query graph. Such a query can be performed by processing all size 1 nodes in the DAG and marking each node that has a vertex matching any vertex of the query graph. The descendents of these marked nodes contain all database graphs which have at least one vertex in common with the query graph. Thus, the answer set for the query is the set of all database graphs minus these descendents. This approach of computing an answer set and taking its compliment can be generalized to an algorithm for Far-neighbor searches.

6. Experimental Results

In order to evaluate the performance of the algorithms presented in Sections 3 through 5, we developed a search engine named *GDIndex*. *GDIndex* was implemented in C++ and compiled using Microsoft Visual Studio 6.0.

We used C-tree, which was developed and provided by He et al. [7], as a performance comparison. C-tree was implemented in Java and compiled using Sun JDK 1.5.0.

All experiments were made using a 3 GHz Pentium 4 workstation with 1 GB of memory and Windows XP. Index construction times reported for both *GDIndex* and C-tree include time required to write the index to disk. The reported index space requirements are the size of the index files created by their respective applications. Time performance measurements for C-tree were reported by C-tree itself. Query times were measured while maintaining the index in main memory.

6.1. Datasets

Performance was measured using two datasets. The first was a protein motif dataset, derived from protein structure information obtained from the Protein Data Bank [24]. The protein motif dataset is a collection of graphs constructed from mining protein graphs for frequent patterns [23]. Each graph encodes the structure of a reoccurring three-dimensional protein structure. The vertices are used to represent amino acid residues, with discrete edge labels encoding all pair-wise distances between them. The second dataset was a synthetic dataset generated using software developed by Kuramochi et al. [25].

The protein motif database represents a database of complete, edge-labeled graphs. It consists of 10000 complete graphs with an average of 6.3 vertices. The maximum graph size, a primary factor in the performance of GDIndex, is 11. In contrast, the synthetic dataset represents a database of sparse graphs without edge labels. It consists of 10000 graphs with an average of 9.27 vertices and 10.65 edges. The maximum graph size for this data set is 21.

For both datasets, we report the index size and index construction times for random subsets of the dataset. Our results also show the average query times for various sizes of subgraph isomorphism queries. In the case of similarity queries, we demonstrate the effects of varying range as well. Query graphs were obtained randomly from the set of all canonical subgraphs that are represented in the dataset.

6.2. Protein Motif Index Performance

For the protein motif dataset, Figure 4 shows a comparison of the index size and index construction time for both GDIndex and C-tree. The superior performance of GDIndex is primarily attributable to the ability of GDIndex to identify sub-isomorphic relationships between the database graphs.

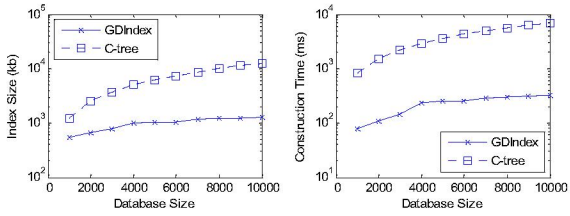
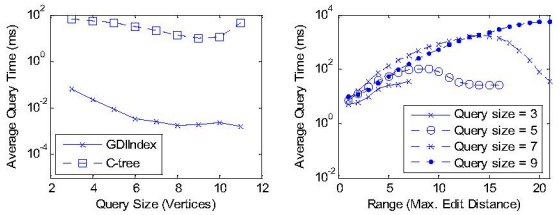


Figure 4. Index Size(a) and Construction Time(b).

In Figure 5, we show the performance of GDIndex as measured by average query time. The subgraph isomorphism test was conducted using the set of all subgraphs of the database graphs. For the similarity search, we tested performance using 100 random database subgraphs for each size, s = {3, 5, 7, 9}. Note that as the range increases beyond a certain threshold, query times begin to decrease. Since most of the graphs in the database are obviously within the search range, there are many mappings between the graph and the query which meet such lax criteria. Once any such mapping is discovered, that graph is declared a match and no further mappings need to be explored.



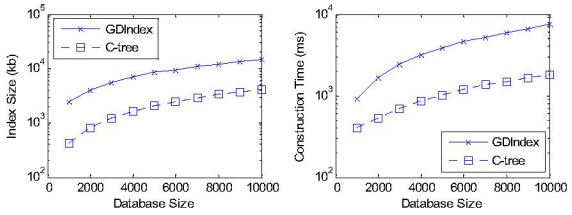
(a) (b)

Figure 5. Subgraph Isomorphism Query Time(a), Range Similarity Query (b).

Because the implementation of C-tree does not handle edge labels, it was necessary to insert an additional vertex for each edge to encode this information. Since the edge and vertex labels were drawn from disjoint sets, there could be no ambiguity between edges and vertices. This enabled a performance comparison to be made in the case of subgraph isomorphism queries, where the computed answers were the same. However, the translation is not valid for similarity queries, since they permit the insertion and deletion of edges. Hence, we present only the performance of GDIndex with regard to similarity queries.

6.3. Synthetic Dataset

The synthetic dataset contains larger graphs than the graphs in the protein motif dataset. It is primarily for this reason that GDIndex creates a larger index than C-tree, as shown in Figure 6.

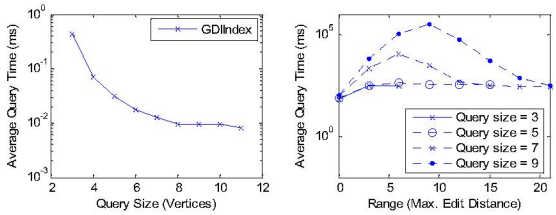


(a) (b)

Figure 6. Index Size(a) and Construction Time(b).

For the synthetic dataset, GDIndex computes answer sets which differ from the answer sets given by C-tree. This is because C-tree tests for subgraph isomorphism, while GDIndex tests for *induced* subgraph isomorphism, and because the queries are no longer complete graphs, as is the case with the protein motif dataset. Thus, the subgraph isomorphism times for C-tree are not presented with the times for GDIndex in Figure 7. Nevertheless, Figure 7 shows that query times average significantly less than a millisecond.

Similarity query responses also differed in this test. GDIndex computes a complete answer for similarity queries, while C-tree computes an approximate answer. For this dataset, there were many cases for which C-tree failed to return a single graph which was within the range, although hundreds of valid answers existed. Since it was observed that C-tree finds far fewer matches than GDIndex (albeit in much less time), it is inappropriate to compare query times. Thus, Figure 1 shows the query times for GDIndex only.



(a) (b)

Figure 7. Subgraph Isomorphism Query Time(a), Range Similarity Query Time(b).

7. Conclusions and Future Work

Our graph indexing approach showed dramatic improvements in query times for subgraph isomorphism queries. It was also able to generate complete answers for a meaningful range of similarity searches within 1 second. In addition, in the case of the protein motif database, it accomplished these times using a significantly smaller index than C-tree.

In the future, we will attempt to improve query times for similarity queries with greater ranges. Additionally, since our current approach is limited to databases containing only small graphs (less than ~20 nodes), we will explore methods intended to extend this technique to databases with larger graphs.

Our method of graph decomposition appears well-suited to answering questions concerning common subgraphs. For this reason, we will investigate its use in other applications, such as pattern mining. This may also lead to exploring the use of similarity metrics other than edit distance, such as the metric presented in [8].

Finally, we will investigate the use of yet more, domain-specific, similarity metrics. In particular, we will work to develop similarity metrics useful for comparing protein structures.

8. Acknowledgements

This work is supported by NSF grants IIS-0513789 and CCF-0523875, a Microsoft eScience Award, and a Microsoft New Faculty Fellowship. We thank H. He and A. K. Singh for providing the code of C-tree, and M. Kuramochi and G. Karypis for providing the synthetic graph generator.

9. References

[1] Christian Borgelt, and Michael R. Berthold. "Mining Molecular Fragments: Finding Relevant Substructures of Molecules", ICDM, 2002, pp. 51-58.

[2] J. Huan, W. Wang, D. Bandyopadhyay, J. Snoeyink, J. Prins, and A. Tropsha, "Mining Protein Family Specific Residue Packing Patterns from Protein Structure Graphs", *In Proceedings of the 8th Annual International Conference on Research in Computational Molecular Biology (RECOMB)*, 2004, pp. 308-315.

[3] B. T. Messmer, and H. Bunke, "A Decision Tree Approach to Graph and Subgraph Isomorphism Detection", *Pattern Recognition*, December 1999, Vol. 32, No. 12, pp. 1979-1998.

[4] S. Raghavan and H. Garcia-Molina, "Representing Web Graphs", *In Proceedings of the IEEE Intl. Conference on Data Engineering*, 2003.

[5] J. Huan, W. Wang, J. Prins, and J. Yang, "SPIN: Mining Maximal Frequent Subgraphs from Graph Databases", *In Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2004, pp 581-586.

[6] J. R. Ullman, "An Algorithm for Subgraph Isomorphism", *Journal of the Association for Computing Machinery*, 1976, Vol. 23, pp. 31-42.

[7] H. He, and A. K. Singh, "Closure-Tree: An Index Structure for Graph Queries", *ICDE'06*, Atlanta, Georgia, 2006.

[8] H. Bunke, and K. Shearer, "A Graph Distance Metric Based on the Maximal Common Subgraph", *Pattern Recognition Letters*, 1998, Vol. 19, No. 3-4, pp. 255-259.

[9] D. Banyopadhyay, J. Huan, J. Liu, J. Prins, J. Snoeyink, A. Tropsha, and W. Wang, "Using Fast Subgraph Isomorphism Checking for Protein Functional Annotation Using *SCOP* and *Gene Ontology*", *UNC CS Technical Report*, 2005.

[10] National Cancer Institute, http://www.nci.nih.gov/.

[11] M. R. Garey, and D. S. Johnson, *Computers and Intractability*, W. H. Freeman and Company, New York, New York, 1979.

[12] S. Fortin, "The Graph Isomorphism Problem", *Technical Report,* The University of Alberta, July 1996.

[13] P. Willett, J. Barnard, and G. Downs, "Chemical similarity searching", *J. Chem. Inf. Comput. Sci.*, 1998, Vol. 38, No. 6, pp. 983-996.

[14] O. Johansson, "Graph Decomposition Using Node Labels", *Doctoral Dissertation*, Royal Institute of Technology, Stockholm, 2001.

[15] D. Shasha, J. T. L. Wang, and R. Giugno, "Algorithmics and Applications of Tree and Graph Searching", 2002.

[16] X. Yan, P. S. Yu, and J. Han, "Graph Indexing Based on Discriminative Frequent Structure Analysis", ACM Transactions on Database Systems (TODS), December 2005.

[17] X. Yan, P. S. Yu, and J Han, "Substructure Similarity Search in Graph Databases", *SIGMOD Conference*, ACM Inc., Baltimore, Maryland, 2005.

[18] S. Srinivasa, M Maier, and M. Mutalikdesai, "LWI and Safari: A New Index Structure and Query Model for Graph Databases", *COMAD 2005*, Goa, India, January 2005.

[19] S. Srinivasa, and M. Harjinder Singh, "GRACE: A Graph Database System", *COMAD 2005b*, Hyderabad, India, December 2005.

[20] J. P. Kukluk, L. B. Holder, and D. J. Cook, "Algorithm and Experiments in Testing Planar Graphs for Isomorphism", *Journal of Graph Algorithms and Applications*, 2004, Vol. 8, No. 2, pp. 101-104.

[21] D. Eppstein, "Subgraph Isomorphism in Planar Graphs and Related Problems", *Journal of Graph Algorithms and Applications*, 1999, Vol. 3, No. 3, pp. 1-27.

[22] X. Wang, J. T. L. Wang, D. Shasha, B. Shapiro, I. Rigoutsos, and K. Zhang, "Finding Patterns in Three Dimensional Graphs: Algorithms and Applications to Scientific Data Mining", *IEEE Transactions on Knowedge and Data Engineering*, July/August 2002, Vol. 14, No. 4, pp. 731 – 749.

[23] J. Huan, W. Wang, and J. Prins, "Comparing Graph Representations of Protein Structure for Mining Family-Specific Residue-Based Packing Motifs", *Journal of Computational Biology(JCB)*, Vol. 12, No. 6, pp.657-671, 2005.

[24] The Protein Data Bank, http://www.pdb.org/.

[25] M. Kuramochi and G. Karypis, "Frequent Subgraph Discovery", *ICDM*, 2001.