Closure-Tree: An Index Structure for Graph Queries

Huahai He Ambuj K. Singh
University of California, Santa Barbara
Santa Barbara, CA 93106, USA
{huahai, ambuj}@cs.ucsb.edu

Abstract

Graphs have become popular for modeling structured data. As a result, graph queries are becoming common and graph indexing has come to play an essential role in query processing. We introduce the concept of a graph closure*, a generalized graph that represents a number of graphs. Our indexing technique, called* Closure-tree*, organizes graphs hierarchically where each node summarizes its descendants by a graph closure. Closure-tree can efficiently support both* subgraph queries *and* similarity queries*. Subgraph queries find graphs that contain a specific subgraph, whereas similarity queries find graphs that are similar to a query graph. For subgraph queries, we propose a technique called* pseudo subgraph isomorphism *which approximates subgraph isomorphism with high accuracy. For similarity queries, we measure graph similarity through edit distance using heuristic graph mapping methods. We implement two kinds of similarity queries: K-NN query and range query. Our experiments on chemical compounds and synthetic graphs show that for subgraph queries, Closure-tree outperforms existing techniques by up to two orders of magnitude in terms of candidate answer set size and index size. For similarity queries, our experiments validate the quality and efficiency of the presented algorithms.*

1 Introduction

Recent technological and scientific advances have resulted in an abundance of data that describe and model phenomena in terms of primitive components and relationships between them. Querying and mining of the resulting graphs has the potential to advance our understanding in numerous ways: understanding of new connectivity patterns and evolutionary changes, and discovery of topological features. Graph data models have been studied in the database community for semantic data modeling, hypertext, geographic information systems, XML, multimedia [1], and schema matching and integration [2]. For example, schema of heterogeneous web-based data sources and e-commerce sites can be modeled as graphs and the problem of schema matching and integration solved through graph matching. In a recent study, video data scenes were modeled as graphs of primitive objects and similarity queries were answered through graph isomorphism [3]. More broadly, graphs have been used for modeling biological pathways [4], chemical compounds [5], protein structures [6], social networks [7], and taxonomies [8, 9]. For example, a metabolic pathway is modeled as a set of reactions, enzymes, and metabolites, and an edge is placed between a reaction and a metabolite (or enzyme) if it participates in the reaction. Similarly, the 3D structure of proteins can be modeled as contact maps: atoms whose distance is less than a threshold have an edge between them.

In all the above cases, the understanding of a collection of graphs can be accelerated through the use of a graph database that supports elementary querying mechanisms. Queries in graph databases can be broadly classified into two categories. In the first category, one looks for a specific pattern in the graph database. The pattern can be either a small graph or a graph where some parts are uncertain, e.g., vertices with wildcard labels. We call this a *subgraph query*. Subgraph queries are useful in a number of applications such as finding structural motifs in protein 3D structures, and pathway discovery in protein interaction graphs. In the second query category, one looks for graphs that are similar to a given graph. We call this a *similarity query*. There are two common kinds of similarity queries: A *K-NN query* finds K nearest graphs to the query graph; a *range query* finds graphs within a specified distance to the query graph. Similarity queries can be used as a building block for applications such as schema matching and classification.

Query processing on graphs is challenging for a number of reasons. Pairwise graph comparisons are usually difficult. For subgraph queries, one faces the subgraph isomorphism problem, known to be NP-Complete. For similarity queries, it is difficult to give a meaningful definition of graph similarity. The problem is also not known to be in the complexity class *P*. Graph datasets can also be very large, leading to a number of pairwise comparisons. Reducing the number of graph comparisons through good summarization and heuristics is the primary motivation for graph index structures.

1.1 Related Work

Several indexing techniques have been developed for graph queries. Shasha et al. [10] proposed a path-based technique called GraphGrep. GraphGrep enumerates paths up to a threshold length from each graph. An index table is constructed where each row stands for a path and each column stands for a graph. Each entry in the table is the number of occurrences of the path in the graph. Queries are processed in two phases. The filtering phase generates a set of candidate graphs for which the count of each path is at least that of the query. The verification phase verifies each candidate graph by subgraph isomorphism and returns the answer set.

Yan et al. [11] proposed GIndex that uses frequent patterns as index features. Frequent patterns reduce the index space as well as improve the filtering rate. Experimental results show that their technique has 10 times smaller index size than that of GraphGrep, and outperforms GraphGrep by 3-10 times in terms of the candidate answer set size. In a subsequent paper, the authors have extended their idea to partial matches of given queries [12].

GraphGrep and GIndex have some common disadvantages. First, they do not support graphs where attributes on vertices or edges are continuous values. The underlying reason is that the index features need to be matched *exactly* with the query. Second, their index construction requires an exhaustive enumeration of paths or fragments with high space and time overhead. Third, since paths or fragments carry little information about a graph, loss of information at the filtering step appears unavoidable.

Berretti et al. [1] proposed a metric based indexing on attributed relational graphs (ARGs) for content-based image retrieval. Graphs are clustered hierarchically according to their mutual distances and indexed by M-trees [13]. Queries are processed in a top-down manner by routing the query along the reference graphs of clusters. Triangle inequality is used for pruning unnecessary nodes. More recently, Lee et al. [3] use a graphical representation for modeling foreground and background scenes in videos. These graphs are clustered using the edit distance metric, and similarity queries are answered using a multi-level index structure.

1.2 Our Approach

We develop a tree-based index called *Closure-tree*, or *C-tree*. Each node in the tree contains discriminative information about its descendants in order to facilitate effective pruning. This summary information is represented as a *graph closure*, a "bounding box" of the structural information of the constituent graphs. Our approach has a number of advantages:

1. C-tree can support both subgraph queries and similarity queries on various kinds of graphs.
2. C-tree extends many techniques developed for spatial access methods, e.g., R-trees [14, 15].
3. Graph closures capture the entire structure of constituent graphs, which implies high pruning rates.
4. Dynamic insertion/deletion and disk-based access of graphs can be done efficiently.
5. C-tree avoids an exhaustive enumeration procedure as in GraphGrep and GIndex.

The approach taken by C-tree can be contrasted by graph indexing approaches based on M-trees [1, 3], where the summary graph in the index structure (routing object) is a database graph; in our approach, this graph is a generalized graph that is a *structural union* of the underlying database graphs.

We perform pairwise graph comparisons using heuristic techniques. For subgraph queries, we tackle the subgraph isomorphism problem by an approximation technique called *pseudo subgraph isomorphism*. Pseudo subgraph isomorphism produces accurate candidate answers within a polynomial running time. For similarity queries, we define graph similarity based on edit distance, and compute it using heuristic graph mapping methods. All C-tree operations take polynomial time.

C-tree is the first index structure that efficiently supports both subgraph queries and similarity queries on graphs. For subgraph queries, our techniques outperform GraphGrep by up to two orders of magnitude in terms of candidate answer set size and index size. For similarity queries, our experiments demonstrate the quality and efficiency of our techniques. Our work also demonstrates how traditional query and indexing techniques can be extended to graph data.

The remainder of the paper is organized as follows. Section 2 defines graph mapping, distance, and similarity. Section 3 introduces the concept of graph closure. Section 4 describe heuristic graph mapping methods. Section 5 presents the design of C-tree. Section 6 presents the idea of pseudo subgraph isomorphism and query processing for subgraph queries. Section 7 presents query processing for similarity queries. Experimental results are reported in Section 8. We conclude with brief remarks in Section 9.

2 Preliminaries

We denote a graph (directed or undirected) G by (V, E) where V is a vertex set and E is an edge set. Vertices and edges have attributes denoted by *attr(v)* or *attr(e)*. A graph database is a set of graphs $D = \{G_1, G_2, ..., G_m\}$. For convenience, we focus on undirected graphs in which vertices have a single label as their attribute and edges have unspecified but identical labels. However, the concepts and techniques described can be extended to other kinds of graphs.

We assume the usual definition of *graph isomorphism*.

Definition 1 (Graph Isomorphism) *Graph G_1 is isomorphic to G_2 if there exists a bijection ϕ such that for every vertex $v \in V_1$, $\phi(v) \in V_2$ and $attr(v) = attr(\phi(v))$, and for every edge $e = (v_1, v_2) \in E_1$, $\phi(e) = (\phi(v_1), \phi(v_2)) \in E_2$, and $attr(e) = attr(\phi(e))$.*

The concept of subgraph isomorphism can be defined analogously by using an injection instead of a bijection.

Next, we define a relaxed notion of correspondence between two graphs. We extend each graph by dummy vertices and dummy edges such that every vertex and edge has a corresponding element in the other graph. This correspondence allows us to compare graphs of unequal sizes. An *extended* graph is denoted by $G^*(V^*, E^*)$. A dummy vertex or edge has a special label ε as its attribute.

Definition 2 (Graph Mapping) *A mapping between two graphs G_1 and G_2 is a bijection $\phi : G_1^* \rightarrow G_2^*$, where (i) $\forall v \in V_1^*$, $\phi(v) \in V_2^*$, and at least one of v and $\phi(v)$ is not dummy, and (ii) $\forall e = (v_1, v_2) \in E_1^*$, $\phi(e) = (\phi(v_1), \phi(v_2)) \in E_2^*$, and at least one of e and $\phi(e)$ is not dummy.*

Now, we define the notion of distance between two graphs using edit distance. Generally, the edit distance between two objects is the cost of transforming one object into the other. For graphs, the transformations are the insertion and removal of vertices and edges, and the changing of attributes on vertices and edges. The cost of these transformations can be generally regarded as a distance function between the two elements (in case of insertion and removal, the other element is a dummy). Given two graphs G_1 and G_2, we can find a mapping ϕ between G_1^* and G_2^*, and compute the distance under this mapping.

Definition 3 (Edit Distance under ϕ) *The edit distance between two graphs G_1 and G_2 under a mapping ϕ is the cost of transforming G_1 into G_2:*

$$d_\phi(G_1, G_2) = \sum_{v \in V_1^*} d(v, \phi(v)) + \sum_{e \in E_1^*} d(e, \phi(e)) \qquad (1)$$

where $d(v, \phi(v))$ and $d(e, \phi(e))$ are the vertex distance and the edge distance measures respectively.

The vertex and edge distance measures are application-specific. For simplicity, we assume a *uniform* distance measure: the distance between two vertices or two edges is 1 if they have different labels; otherwise it is 0.

We can now define edit distance between graphs.

Definition 4 (Graph Distance) *The distance between two graphs G_1 and G_2 is the minimum edit distance under all possible mappings:*

$$d(G_1, G_2) = \min_\phi \{d_\phi(G_1, G_2)\} \qquad (2)$$

Note that the distance between isomorphic graphs is zero. If the vertex and edge distances are metric, then the graph distance is also a metric. Edit distance to a null graph (having no vertices and no edges) defines the norm of a graph.

We now define the asymmetric notion of *subgraph distance*.

Definition 5 (Subgraph Distance) *The subgraph distance from G_1 to G_2 is the minimum distance between G_1 and any subgraph of G_2:*

$$d_{sub}(G_1, G_2) = min\{d(G_1, H) \mid H \subseteq G_2\} \qquad (3)$$

Subgraph distance can also be obtained by considering graph mappings and limiting distance consideration to the non-dummy vertices and edges in the first graph:

$$d_{sub}(G_1, G_2) = \min_\phi \{\sum_{v \in V_1} d(v, \phi(v)) + \sum_{e \in E_1} d(e, \phi(e))\} \qquad (4)$$

In some graph applications, graph similarity is more meaningful than distance, especially when the underlying vertex and distance measures are based on similarity. This notion of similarity is defined as follows.

Definition 6 (Graph Similarity) *The similarity between two graphs G_1 and G_2 under a mapping ϕ is the sum of similarities between vertices and edges in G_1 and their images in G_2:*

$$Sim_\phi(G_1, G_2) = \sum_{v \in V_1^*} sim(v, \phi(v)) + \sum_{e \in E_1^*} sim(e, \phi(e)) \qquad (5)$$

where $sim(v, \phi(v))$ and $sim(e, \phi(e))$ are the vertex similarity and the edge similarity measures respectively. The similarity between two graphs is the maximum similarity under all possible mappings:

$$Sim(G_1, G_2) = \max_\phi \{Sim_\phi(G_1, G_2)\} \qquad (6)$$

We use the notion of uniform similarity as well. For both vertices and edges, it is defined as one minus the distance between them.

An upper bound to similarity can be obtained by considering the vertex sets and the edge sets separately:

$$Sim(G_1, G_2) \leq Sim(V_1, V_2) + Sim(E_1, E_2) \qquad (7)$$

where $Sim(V_1, V_2)$ and $Sim(E_1, E_2)$ are the maximum similarity between two sets of vertices or edges respectively. This can be computed by constructing a bipartite graph, and finding the maximum matching.

Fig. 1 shows a sample graph database consisting of five graphs. If we use uniform distance measures, then $d(G_1, G_2) = 2$, $d_{sub}(G_1, G_2) = 0$, $Sim(G_1, G_2) = 6$, $d(G_1, G_3) = 1$, $d(G_2, G_4) = 2$, $Sim(G_4, G_5) = 5$, etc.

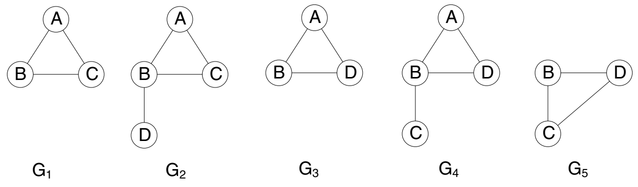


Figure 1. A Sample graph database

In practice, we cannot compute the optimal distance or similarity due to high computational complexity. However, we can find a good mapping between two graphs, and compute the approximate distance or similarity between graphs. We will discuss graph mapping methods in Section 4.

3 Graph Closures

Given two graphs and a mapping between them, if we take an elementwise union of the two graphs, then we get a new graph where the attribute of each vertex and each edge is a union of the constituent attribute values. This generalized graph captures the structural information of each graph, and serves as a bounding container. This leads to the concept of a graph closure.

Definition 7 (Vertex Closure and Edge Closure) *The closure of a set of vertices is a generalized vertex whose attribute is the union of the attribute values of the vertices. Likewise, the closure of a set of edges is a generalized edge whose attribute is the union of the attribute values of the edges.*

In particular, a vertex (or edge) closure may contain the special value ε corresponding to a dummy.

Definition 8 (Graph Closure under ϕ) *The closure of two graphs G_1 and G_2 under a mapping ϕ is a generalized graph (V, E) where V is the set of vertex closures of the corresponding vertices and E is the set of edge closures of the corresponding edges. This is denoted by $closure(G_1, G_2)$.*

The closure of two graphs depends on the underlying graph mapping. We usually leave this mapping implicit. In the case of multiple graphs, we can compute the closure incrementally, i.e., compute the closure $C_1 = closure(G_1, G_2)$, and then the closure $C_2 = closure(C_1, G_3)$, and so on.

A graph closure has the characteristics of a graph: only instead of singleton labels on vertices and edges, a graph closure can have multiple labels. The ideas of graph isomorphism, subgraph isomorphism can be extended to them easily. The notion of distance, however, needs to be reconsidered. Since a graph closure represents a set of graphs (akin to a Minimum Bounding Rectangle (MBR) in usual index structures), we define the notion of minimum distance between two graph closures (akin to the minimum distance between two MBRs).

Definition 9 (Minimum Distance between Closures under ϕ) *The minimum distance between two graph closures G_1 and G_2 under a mapping ϕ is defined as follows:*

$$d_\phi(G_1, G_2) = \sum_{v \in V_1^*} d_{min}(v, \phi(v)) + \sum_{e \in E_1^*} d_{min}(e, \phi(e)) \tag{8}$$

where the d_{min} distances are obtained using the underlying distance measures for vertices and edges.

For the case of the uniform distance measure, $d_{min}(v, \phi(v))$ and $d_{min}(e, \phi(e))$ is 0 if the closures share a label and is 1 otherwise. The minimum distance between two graph closures is defined as the minimum of the above distance under all possible mappings ϕ; this is denoted as d_{min}. The notion of maximum similarity between graph closures is defined analogously: define the maximum similarity under a mapping, and then take the maximum of these over all possible mappings. This is denoted as Sim_{max}.

In the rest of this paper, the distance between two graph closures refers to the minimum distance, and the similarity refers to the maximum similarity, unless specified otherwise. An upper bound to the similarity between two graph closures can be obtained by considering the vertex sets and the edge sets separately as in Eqn. (7).

The distance (or similarity) between a graph G and a graph closure C is a lower bound (or upper bound) to the distance (or similarity) between G and any graph H contained in C.

$$d_{min}(G, C) \le d(G, H)$$
$$Sim_{max}(G, C) \ge Sim(G, H)$$

A graph closure may contain graphs that do not actually exist in a given dataset (just as an MBR may contain points that do not exist in a given point dataset). To measure the number of graphs that a graph closure can possibly contain, we define the volume of a graph closure:

Definition 10 (Volume of Graph Closure) *The volume of a graph closure C is the product of the sizes of its vertex closures VC and its edge closures EC:*

$$volume(C) = \prod_{vc \in VC} |vc| \times \prod_{ec \in EC} |ec| \tag{9}$$

Fig. 2 shows some graph closures of the graphs in Fig. 1. The dotted edges represent the closure of a dummy and a non-dummy edge.

The closure of two graphs depends on the graph mapping. In other words, the quality of the summary achieved by a closure depends on the quality of the graph mapping. Next, we discuss different ways of obtaining graph mappings.

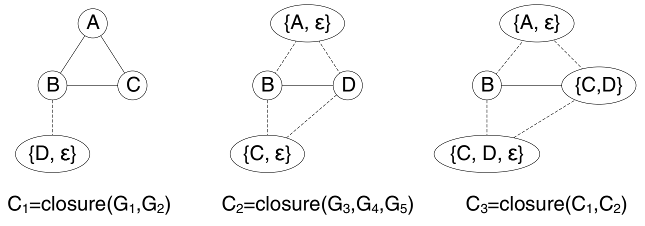


Figure 2. Sample graph closures

4 Graph Mapping Methods

We discuss three different ways of finding graph mappings. Since the problem is computationally hard, we rely on heuristics. Though presented in the context of graphs, these techniques are also applicable to graph closures.

4.1 State Search Method

A state search method can be used to find the optimal mapping between two small graphs. A branch-and-bound scheme can be implemented as follows. At each search state, we map a free vertex in one graph onto a free vertex in the other graph, and estimate an upper bound of future states using Eqn. (7). If the upper bound is less than to equal to the maximum similarity found so far, then the current state can be pruned.

The state search method works well only on small graphs, e.g., graphs of size less than 10. For larger graphs, we need approximate methods. These are discussed next.

4.2 Bipartite Method

This method constructs a bipartite graph B between two graphs G_1 and G_2. The two partitions in B are the vertices from G_1 and G_2. The edges of B are formed by connecting the vertices across the partitions. The maximum matching in B defines the graph mapping.

The bipartite graph can be either unweighted or weighted, based on the similarity measure used. If it is unweighted, then the maximum matching is computed using Hopcroft and Karp's algorithm [16]. If it is weighted, then the maximum matching is computed by the Hungarian algorithm [17, 18]. The weight between two vertices is measured by the similarity of their attributes as well as their neighbors. Using matrix iteration, weights can be propagated to all the vertices until convergence. Heymans and Singh [19] used this method to compute the similarity between metabolic pathway graphs.

4.3 Neighbor Biased Mapping (NBM)

In the bipartite method, the similarity between any two vertices is fixed during the computation of the graph mapping. There is no effort made to bias the matching towards neighbors of already matched vertices, i.e., even if two vertices have been matched, the chance that their neighbors will be matched does not increase. As a result, the common substructures of the two graphs are not captured well.

In order to find common substructures, we develop a new graph mapping method called *Neighbor Biased Mapping (NBM)* shown in Alg. 1. Initially, a weight matrix W is computed where each entry $W_{u,v}$ represents the similarity of vertex $u \in G_1$ and vertex $v \in G_2$. A priority queue PQ maintains pairs of vertices according to their weights. For each vertex in G_1, its most similar vertex is found in G_2, and the pair is added to PQ. At each iteration, the best pair (u, v) of unmatched vertices in the priority queue is chosen and marked as matched. Then, the neighboring unmatched pairs of (u, v) are assigned higher weights, thus increasing their chance of being chosen. The iterations continue until all vertices in graph G_1 have been matched.

Algorithm 1 NBM

```
Compute the initial similarity matrix $W$ for $G_1$ and $G_2$;
**for each** $u \in G_1$ **do**
    Find $v_m$ such that $W_{u,v_m} = max\{W_{u,v} | v \in G_2\}$
    PQ.Insert($W_{u,v_m}$, $\langle u, v_m \rangle$)
    $mate[u] := v_m$   // best mate of $u$
    $wt[u] := W_{u,v_m}$  // best weight of $u$
**while** PQ is not empty **do**
    $\langle u, v \rangle$ := PQ.dequeue()
    **if** $u$ is matched **then**
        **continue**
    **if** $v$ is matched **then**
        Find $v_m$ such that
        $W_{u,v_m} = max\{W_{u,v} | v \in G_2, v$ is unmatched$\}$
        PQ.Insert($W_{u,v_m}$, $\langle u, v_m \rangle$)
        $mate[u] := v_m$
        $wt[u] := W_{u,v_m}$
        **continue**
    Mark $\langle u, v \rangle$ as matched
    Let $N_u, N_v$ be the neighbors of $u, v$
    **for each** $u' \in N_u$, $u'$ is unmatched **do**
        **for each** $v' \in N_v$, $v'$ is unmatched **do**
            Add weights to $W_{u',v'}$
            **if** $W_{u',v'} > wt[u']$ **then**
                $mate[u'] := v'$
                $wt[u'] := W_{u',v'}$
        **if** $wt[u']$ has changed **then**
            PQ.Insert($wt[u']$, $\langle u', mate[u'] \rangle$)
**return** all matches
```

The time complexity of the algorithm can be computed as follows. Let n be the number of vertices and d be the maximum degree of vertices. The initial computation of matrix W and insertions into the priority queue take $O(n^2)$ time, assuming uniform distance measures. In each iteration, the algorithm removes one pair from and inserts at most d^2 unmatched pairs into the priority queue. Totally, there are $O(n)$ iterations. Thus, the time complexity is

$O(nd^2 logn)$.

5 Closure-Tree

In this section, we describe the structure of C-tree and various operations on it, including insertion, splitting, deletion, and tree construction. All the operations take polynomial time.

5.1 Tree Structure

A *C-tree* is a tree of nodes where:

1. Each node is a graph closure of its children. The children of an internal node are nodes; the children of a leaf node are database graphs.
2. Each node has at least m children unless it is root, $m \geq 2$.
3. Each node has at most M children, $\frac{(M+1)}{2} \geq m$.

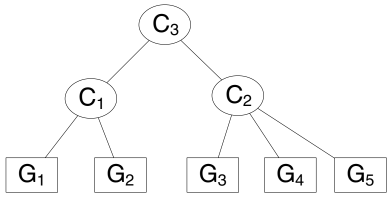


Figure 3. Closure-tree

Fig. 3 shows an example of a C-tree for the sample graph database where the graph closures are shown in Fig. 2.

The structure of C-tree is similar to that of tree-based spatial access methods, e.g., R-trees [14]. The main difference is that each node in C-tree is represented by a graph closure instead of an MBR. Operations of a C-tree are analogous to that of an R-tree.

5.2 Insertion

Given a new graph, an insertion operation begins at the root and iteratively chooses a child node until it reaches a leaf node. The given graph is inserted at this leaf node. Graph closures at each node along the path are enlarged accordingly. The main challenge of insertion is the criterion for choosing a child node. We describe several such criteria next.

- *Random selection.* Choose a child node randomly.
- *Minimizing the increase of volume.* Choose a child node that results in the minimum increase of volume (Eqn. 10).
- *Minimizing the overlap of nodes.* Choose a child node that results in the minimum increase of overlaps. The overlap of two nodes is measured by the similarity of their graph closures (Eqn. 6).

A practical consideration is to achieve a trade off between quality and running time. We choose the second criterion, which takes time linear in the number of child nodes.

5.3 Splitting

When a C-tree node has more than M child nodes because of insertion, it has to be split into two nodes. Splitting may cause the parent node to split as well and this procedure may repeat all the way up to the root. We need a criterion for partitioning the child nodes into two groups. Several options are possible:

- *Random Partitioning.* Randomly and evenly partition the child nodes into two groups.
- *Optimal partitioning.* Consider all possible partitions of child nodes and choose the one with the minimum sum of volumes.
- *Linear partitioning.* Choose a pivot and partition the child nodes based on the pivot. The idea is inspired by Fastmap [20]. The procedure is described as follows.

    1. Randomly choose a node g_0.
    2. Choose the farthest node g_1 from g_0 (using graph closure distance).
    3. Choose the farthest node g_2 from g_1. The pair of nodes (g_1, g_2) becomes the pivot.
    4. For all g_i, sort $d(g_i, g_1) - d(g_i, g_2)$ in ascending order. Then, assign the first half to one group and the rest to the other group.

Linear partitioning takes time linear in the number of child nodes. We use this criterion in our implementation.

5.4 Deletion

To delete a graph from a C-tree, we find the leaf node u where the graph is stored, and delete the graph. Graph closures at nodes along the path are shrunk accordingly. After deletion, if u has less than m entries, then u is deleted and its entries are reinserted. This procedure may propagate up to the root, but entries of non-leaf nodes are reinserted at a higher level.

5.5 Tree Construction

A straightforward approach to building a C-tree is by inserting the graphs sequentially. However, the tree structure will not be globally optimized since it is affected by the insertion order. Further, this approach involves a large number of insertion and splitting operations. An alternative approach is to build the C-tree in combination with some clustering algorithm. In our implementation, we use hierarchical clustering [21] to build the C-tree.

6 Subgraph Queries

Subgraph queries find all graphs that contain a specific subgraph. Subsection 6.1 presents our approximation algorithm for subgraph isomorphism. Subsection 6.2 describes the processing of subgraph queries on a C-tree. Finally, subsection 6.3 presents a cost model for subgraph queries.

6.1 Pseudo Subgraph Isomorphism

Since subgraph isomorphism is an NP-hard problem, avoiding complete subgraph isomorphism tests is an important concern in index construction and query processing. We tackle the problem by an approximation technique called *pseudo subgraph isomorphism*. Though the presentation in this subsection is in the context of graphs, the ideas also hold for graph closures.

Given a graph G and a vertex $u \in G$, we define a **level-n adjacent subgraph** of u as a subgraph derived from G that contains all vertices reachable from u within a distance of n. Given two graphs G_1, G_2 and two vertices $u \in G_1, v \in G_2$, u is called **level-n compatible** to v if the level-n adjacent subgraph of u is sub-isomorphic to that of v.

Based on level-n compatibility, we can construct a bipartite graph B for G_1 and G_2 as follows: the vertex sets of B are the vertex sets of G_1 and G_2; for any two vertices $u \in G_1, v \in G_2$, if u is level-n compatible to v, then (u, v) is an edge in B. If B has a semi-perfect matching, i.e., every vertex in G_1 is matched, then G_1 is called **level-n sub-isomorphic** to G_2.

When n is large enough, i.e., n equals the size of the vertex set of G_1, then level-n sub-isomorphism implies actual subgraph isomorphism. The computation of level-n sub-isomorphism is computationally intensive and does not scale for large n. Therefore, we further approximate adjacent subgraphs by adjacent subtrees.

Definition 11 (Level-n Adjacent Subtree) *Given a graph G and a vertex $u \in G$, a* **level-n adjacent subtree** *of u is a breadth-first tree on G starting at u and consisting of paths of length $\leq n$.*

Note that vertices may appear repeatedly in an adjacent subtree.

Definition 12 (Level-n Pseudo Compatible) *Vertex u is called* **level-n pseudo compatible** *to v if the level-n adjacent subtree of u is sub-isomorphic to that of v.*

Definition 13 (Level-n Pseudo Sub-Isomorphism) *Given two graphs G_1 and G_2, define a bipartite graph B as follows: the vertex sets of B are the vertex sets of G_1 and G_2; for any $u \in G_1, v \in G_2$, if u is level-n pseudo compatible to v, then (u, v) is an edge in B. G_1 is called* **level-n pseudo sub-isomorphic** *to G_2 if B has a semi-perfect matching.*

Fig. 4 outlines the approximation idea. We conceptually approximate subgraph isomorphism by level-n sub-isomorphism using adjacent subgraphs. Then, we approximate level-n sub-isomorphism by level-n pseudo sub-isomorphism using adjacent subtrees. The following lemma establishes that "sub-isomorphism" is a stronger condition than "level-n sub-isomorphism" and "level-n pseudo sub-isomorphism".

Lemma 1 *If G_1 is sub-isomorphic to G_2, then G_1 is sub-isomorphic to G_2 at any level. If G_1 is level-n sub-isomorphic to G_2, then G_1 is level-n pseudo sub-isomorphic to G_2.*

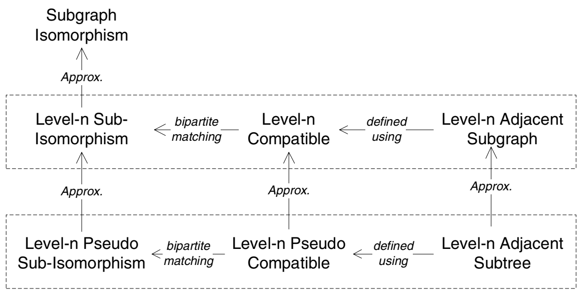


Figure 4. Relationship among the definitions

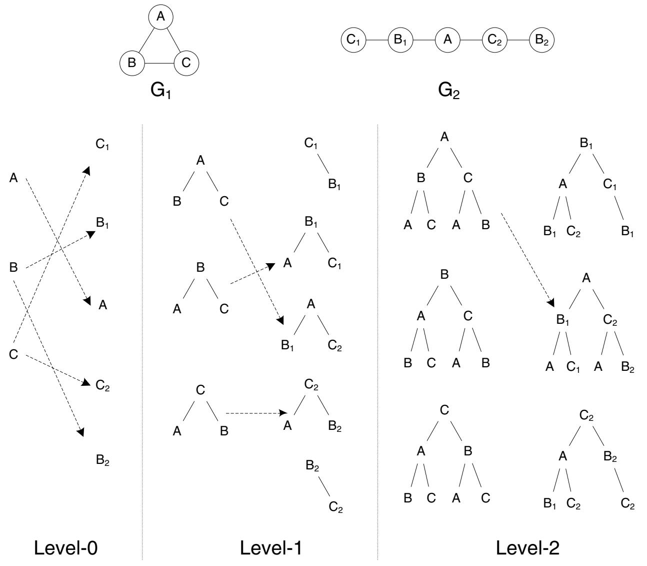


Figure 5. An example of pseudo subgraph isomorphism at levels 0, 1, 2

Fig. 5 shows an example of pseudo subgraph isomorphism. Given G_1 and G_2 (the subscripts of the labels in G_2 are used to distinguish vertices with the same label), we construct a bipartite graph starting at level 0. The "vertices" of the bipartite graph are adjacent subtrees of the vertices in G_1 and G_2. At level-0, the subtree of each vertex is a single vertex. Each vertex in G_1 is pseudo compatible to at least one vertex in G_2. At level-1, the subtree of vertex B in G_1 is not sub-isomorphic to that of vertex B_2 in G_2. Thus, B is not pseudo compatible to B_2. Nor is C to C_1. But, the bipartite graph still has a semi-perfect matching. At level-2, neither B nor C in G_1 is pseudo compatible to any vertex in G_2. The bipartite graph has no semi-perfect matchings. Therefore, G_1 is not level-2 pseudo sub-isomorphic to G_2.

The following theorem shows that level-n pseudo compatibility can be computed in a recursive way.

Theorem 1 (Level-n Pseudo Compatible) *Vertex u is **level-n pseudo compatible** to vertex v if*

1. *u is level-0 compatible to v, and*
2. *let $N_{G_1}(u)$ and $N_{G_2}(v)$ be the neighbors of u and v; define a local bipartite graph B' as follows: for any $u' \in N_{G_1}(u), v' \in N_{G_2}(v)$, (u', v') is an edge in B' if and only if u' is level-$(n-1)$ pseudo compatible to v'. Then, B' has a semi-perfect matching.*

The proof follows from the fact that level-n compatibility of a vertex is equivalent to the level-$(n-1)$ compatibility of all its neighbors and the existence of a semi-perfect matching.

Theorem 1 provides an efficient way to compute level-n pseudo compatibility and level-n pseudo subgraph isomorphism. This is encoded in Alg. 2. Initially, we compute the level-0 compatibility matrix B, where $B_{u,v} = 1$ if u can be mapped onto v, e.g., u and v share a label. Next, we refine B in procedure RefineBipartite: for each pair (u, v), a local bipartite graph B' is constructed. If B' has no semi-perfect matchings, then u is not pseudo compatible to v. The refinement continues until B converges or the given level l is reached. Finally, we check if B has a semi-perfect matching. If it does, then G_1 is pseudo sub-isomorphic to G_2.

The worst case time complexity of the pseudo subgraph isomorphism test is $O(ln_1n_2(d_1d_2 + M(d_1, d_2)) + M(n_1, n_2))$ where l is the pseudo compatibility level, n_1 and n_2 are numbers of vertices in G_1 and G_2, d_1 and d_2 are the maximum degrees of G_1 and G_2, $M()$ is the time complexity of maximum cardinality matching for bipartite graphs. Hopcroft and Karp's algorithm [16] finds a maximum cardinality matching in $O(n^{2.5})$ time.

Theorem 2 *If u is level-(n_1n_2) pseudo compatible to v, then u is pseudo compatible to v at any level.*

Proof. Consider the number of iterations in the outer loop of procedure RefineBipartite of Alg. 2. At least one entry of B is changed to zero in each iteration before B converges. There are at most n_1n_2 $1's$ in B, thus B converges in at most n_1n_2 iterations. After convergence, if $B_{u,v} = 1$, i.e., u is level-(n_1n_2) pseudo compatible to v, then $B_{u,v}$ will continue to be 1 if the iteration were continued further beyond n_1n_2 times. Therefore, u is pseudo compatible to v at any level.

Corollary 1 *If G_1 is pseudo sub-isomorphic to G_2 at level n_1n_2, then G_1 is pseudo sub-isomorphic to G_2 at any level.*

6.2 Processing of Subgraph Queries

A subgraph query is processed in two phases. The first phase traverses the C-tree, pruning nodes based on pseudo subgraph isomorphism. A candidate answer set is returned. The second phase verifies each candidate answer for exact subgraph isomorphism and returns the answers.

Algorithm 2 PseudoSubIsomorphic(G_1, G_2, l)

```
begin
  for each vertex $u \in G_1, v \in G_2$ do
    $B_{u,v} := \begin{cases} 1 & \text{if } attr(u) \bigcap attr(v) \neq \emptyset; \\ 0 & \text{otherwise.} \end{cases}$
  RefineBipartite($G_1$, $G_2$, $B$, $l$)
  $M$ := MaximumCardinalityMatching($B$)
  if $M$ is a semi-perfect matching then
    return true
  else
    return false
end
Procedure RefineBipartite($G_1$, $G_2$, $B$, $l$)
  for $i := 1$ to $l$ do
    for each vertex $u \in G_1, v \in G_2$ where $B_{u,v} \neq 0$ do
      Let $N_{G_1}(u), N_{G_2}(v)$ be the neighbors of $u$ and $v$,
      construct a local bipartite graph $B'$:
      for each $u' \in N_{G_1}(u), v' \in N_{G_2}(v)$ do
        $B'_{u',v'} := \begin{cases} 1 & \text{if } B_{u',v'} \neq 0; \\ 0 & \text{otherwise.} \end{cases}$
      $M'$ := MaximumCardinalityMatching($B'$)
      if $M'$ is NOT a semi-perfect matching then
        $B_{u,v} := 0$
    end
    if $B$ is unchanged then break
  end
```

In addition to pruning based on pseudo subgraph isomorphism, a lightweight histogram-based pruning can also be employed. The histogram of a graph is a vector that counts the number of each distinct attribute of the vertices and edges. The histogram of a node is stored at its parent node. Given a query Q and a graph G, let F_Q and F_G be their histograms. If Q is sub-isomorphic to G, then $\forall i$ $F_Q[i] \leq F_G[i]$. We use this condition to test a child node before we visit that node. Histogram tests are less accurate but faster than pseudo subgraph isomorphism tests.

Alg. 3 outlines the code for subgraph query processing. We use Ullmann's algorithm [22] for exact subgraph isomorphism. Note that the compatibility matrix B in Alg. 2 can be used to accelerate Ullmann's algorithm.

6.3 Performance Analysis

Next, we analyze the performance of subgraph queries. Table 1 defines the symbols to be used. In the search phase, $|D| \cdot \gamma$ nodes and database graphs are visited and tested by pseudo subgraph isomorphism. In the verification phase, $|CS|$ database graphs are tested by exact subgraph isomorphism. Thus, the total query time is

$$T_{query} = |D| \cdot \gamma \cdot T_{visit} + |CS| \cdot T_{isom} \qquad (10)$$

Next, we estimate the access ratio γ. Let k be the number of children of each node. At each node at level i, we test k

Algorithm 3 SubgraphQuery($query$, $ctree$)

```
begin
  $CS := \{\}$
  Visit($query$, $ctree.root$, $CS$)
  $Ans := \{\}$
  for each $G \in CS$ do
    if SubIsomorphic($query$, $G$) then
      $Ans := Ans \bigcup \{G\}$
  return $Ans$
end
Procedure Visit($query$, $node$, $CS$)
  for each child $c$ of node $node$ do
    Let $G$ be the graph or graph closure at $c$;
    if $\forall\ i\ F_Q[i] \leq F_G[i]$ then
      if PseudoSubIsomorphic($query$, $G$) then
        if $c$ is a database graph then
          $CS := CS \bigcup \{G\}$
        else
          Visit($query$, $c$, $CS$)
  end
```

Table 1. Notations

Symbol	Description
D	Graph database
CS	Candidate set
Ans	Answer set
α	*Accuracy* of the candidates, i.e., $\frac{\lvert Ans\rvert}{\lvert CS\rvert}$
R	Number of C-tree nodes and graphs visited
γ	*Access ratio*, i.e., $\frac{R}{\lvert D\rvert}$
T_{isom}	Average time for sub-isomorphism test
T_{visit}	Average time to visit a node or graph
T_{query}	Query processing time

children by histograms. Let $x(i)$ be the number of children that survive the histogram test. We then visit $x(i)$ children and test them by pseudo subgraph isomorphism. Let $y(i)$ be the number of children that survive the pseudo subgraph isomorphism test; these $y(i)$ nodes will be traced down to the next level. Let $R(i)$ be the expected number of nodes and database graphs visited below a node at level i, then

$$\begin{aligned} R(i) &= x(i) + y(i)R(i+1),\ \forall\ 0 \leq i < h \\ R(h) &= 1 \end{aligned} \tag{11}$$

where h is the height of the C-tree. Thus, $1+R(0)$ is the expected number of database graphs and nodes visited during the query. By solving Eqn. (11), we get

$$R(0) = \sum_{i=0}^{h-1} x(i) \prod_{j=0}^{i-1} y(j) + \prod_{i=0}^{h-1} y(i). \tag{12}$$

The access ratio γ can be computed as $\frac{1+R(0)}{|D|}$.

In order to gain an insight into the values $x(i)$ and $y(i)$, assume an uniform distribution of the candidate set CS of database graphs across the leaf nodes of the C-tree index structure. Assuming an uniform fan-out of k and a height h for the tree, the probability of a database graph belonging to CS is $\frac{|CS|}{k^h}$. Then, the expected number of database graphs in the candidate set under an index node at level i is $\frac{|CS|}{k^i}$. This value, and hence the probability of a node at level i having a candidate database graph in its subtree, decreases exponentially with i. Quantities $x(i)$ and $y(i)$, which denote the number of children to be inspected at level $i+1$, will be proportional to (fan-out) $\cdot$ (probability of a node at level $i+1$ having a candidate database graph in its subtree). Therefore, $x(i)$ and $y(i)$ can be evaluated as

$$x(i) = c_1 k\ \rho^{-i},\ \ y(i) = c_2 k\ \rho^{-i} \tag{13}$$

where ρ, c_1 and c_2 are constants that can be estimated empirically.

7 Similarity Queries

Similarity queries find graphs which are similar, but not necessarily isomorphic to a given query graph. Since computing exact similarity is expensive, we compute approximate graph similarity (or distance) using the heuristic graph mapping methods discussed in Section 4. We present the algorithm for K-NN query; discussion of the range query is skipped for brevity.

A K-NN query finds K nearest graphs to the query graph. We implement this query by incremental ranking [23, 24] as follows. A priority queue maintains C-tree nodes and they are visited according to their similarity to the query. Each time the top entry is chosen from the priority queue. We check whether the entry is a node. If the entry is a node, then its children are inserted into the priority queue, otherwise the entry (a database graph) is reported. The procedure stops after k database graphs have been reported.

We can accelerate the ranking using a lower bound threshold. Whenever the similarity of a child is less than or equal to the lower bound, it is discarded immediately. The lower bound is the similarity of the k^{th} nearest graph found so far. To keep track of the lower bound, we use another priority queue *PQ2* to store the k nearest graphs.

Alg. 4 outlines the code for K-NN query. Note that the similarities (Sim) and their upper bounds (Sim_{up})are computed approximately using the techniques defined in Sections 2, 3, and 4.

8 Experimental Results

In this section, we evaluate the performance of C-tree for both subgraph queries and similarity queries. We use two kinds of datasets in our experiments. One is a chemical compounds dataset and the other is a synthetic graph

Algorithm 4 K-NN query

```
PQ: Priority queue for ranking and tree traversal;
PQ2: Priority queue for pruning based on lower bounds;
$lb$: Lower bound of the $k^{th}$ nearest graph
  PQ.Insert(0, ctree.root)
  $lb := count := 0$
  **while** $count < k$ **do**
    $entry$ := PQ.dequeue()
    **if** $entry$ is a database graph **then**
      report $entry.graph$
      $count := count + 1$
    **else**
      **for each** $child$ of $entry.node$ **do**
        $sim := Sim_{up}(query, child)$ //based on Eqn. (7)
        **if** $sim \leq lb$ **then**
          continue **end if**
        **if** $child$ is a database graph **then**
          $sim := Sim(query, child)$
          **if** $sim \leq lb$ **then**
            continue **end if**
          PQ2.Insert($sim$, $child$)
          **if** $|PQ2| > k$ **then**
            PQ2.dequeue()
            $lb$ := PQ2.top().$sim$
          **end if**
        **end if**
        PQ.Insert($sim$, $child$)
      **end for**
    **end if**
  **end while**
```

dataset. For subgraph queries, we compare C-tree with an existing graph indexing technique, GraphGrep.

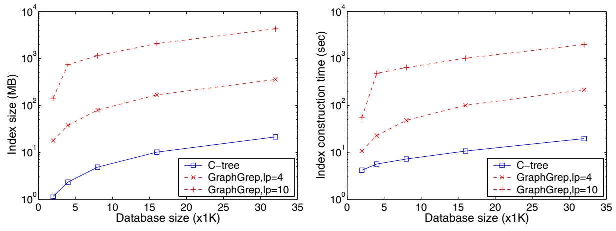
C-tree was implemented in Java and compiled with Sun JDK 1.5.0. GraphGrep was provided by Shasha and Giugno et al. [10] and compiled with gcc/g++. All experiments were done on an AMD Athlon 1.8GHz, 2GB memory workstation running Linux/Debian 3.0.

8.1 Subgraph Queries

8.1.1 Chemical Compounds Dataset

The chemical compounds dataset is an NCI/NIH AIDS Antiviral Screen dataset containing around 42,000 chemical compounds [5]. We generate the vertex-labeled graphs from the molecule structures and omit Hydrogen atoms. The graphs have an average number of 25 vertices and 27 edges, and a maximum number of 222 vertices and 251 edges. A major portion of the vertices are C, O and N. The total number of distinct labels is 62.

For C-tree, we set the minimum number of child nodes $m = 20$ and the maximum number $M = 2m - 1$. We use the NBM method described in section 4.3 to compute graph closures. For GraphGrep, there are two parameters: the length of path (lp) and the length of fingerprint (fp). We set $lp = 4, 10$ and $fp = 256$.



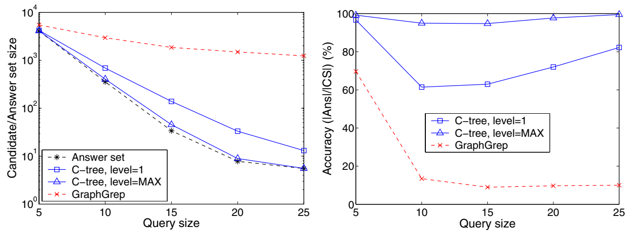
(a) Index size (b) Construction time

Figure 6. Index Size and Construction Time

Fig. 6(a) shows the index size with respect to the database size. The database sizes are 2K, 4K, 8K, 16K, and 32K, obtained by randomly selecting graphs from the original dataset. As shown in the figure, the index sizes of C-tree are at least 10 times smaller than that of GraphGrep when $lp = 4$, and 100 times smaller when $lp = 10$. For C-tree, the information stored is proportional to the database size. To the contrary, GraphGrep has to enumerate every path up to length lp and uses them as index features. In the worst case, the index size for GraphGrep grows exponentially with respect to lp.

Fig. 6(b) shows the index construction time with respect to the database size. The construction time for C-tree is in seconds. The construction time for GraphGrep is much higher because of the enumeration procedure.

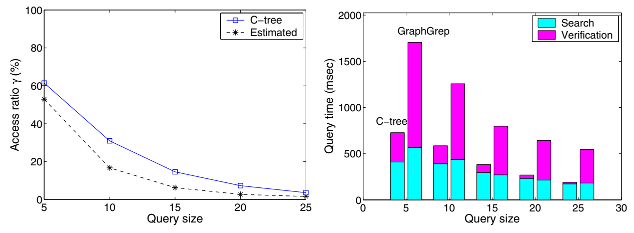
Experiments for subgraph queries are conducted as follows. We fix the database size to 10K and vary the query sizes from 5 to 25 in terms of the number of vertices. For each query size, we generate a set of 1000 queries and average the results. Each query is generated by randomly selecting a graph from the database and randomly extracting a connected subgraph from the graph. For C-tree, we set the pseudo subgraph isomorphism level to 1 and MAX respectively, where MAX refers to the level at which the compatibility matrix B converges. For GraphGrep, we set $lp = 4$; other parameters are set to default values.



(a) Candidate/Answer set (b) Accuracy

Figure 7. Accuracy of candidates

Fig. 7(a) shows the candidate and answer set size with respect to the query size. The candidate set size of C-tree decreases more steeply than that of GraphGrep with increasing query size. For large query sizes, the candidate set of C-tree is two orders smaller than that of GraphGrep. Fig. 7(b) shows the accuracy of the candidates. As shown in both figures, when the pseudo sub-isomorphism level is maximum, C-tree can generate candidates with nearly 100% accuracy.



(a) Access ratio (b) Query time

Figure 8. Access ratio and query time

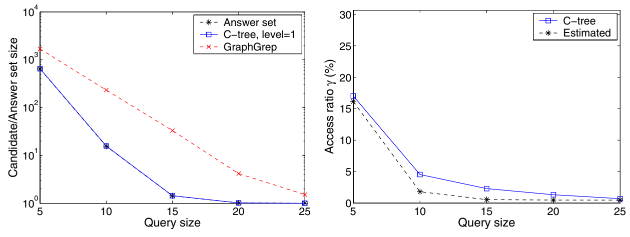
Fig. 8(a) shows the pruning effects of C-tree. The access ratio, i.e., the ratio of the number of accessed nodes and graphs to the total number of graphs, decreases with increasing query size. The underlying reason is that when the query size increases, the number of candiate/answers decreases. So, the number of nodes and graphs to be visited decreases as well. The access ratios are compared with the estimated values discussed in Section 6.3. The estimated values are close to the actual values, which justifies our performance analysis in Section 6.3.

Fig. 8(b) shows the query time, which consists of the search and verification time. The verification time of C-tree is less than that of GraphGrep in all cases, which is a direct result of smaller candidate sets generated by C-tree. The search time of C-tree decreases steadily with respect to the query size. As shown in the figure, the overall query time of C-tree is less than that of GraphGrep even though C-tree was implemented in Java, whereas GraphGrep was implemented in C/C++.

8.1.2 Synthetic Dataset

We also evaluated C-tree on a synthetic dataset. The synthetic dataset is generated by a synthetic graph generator provided by Kuramochi et al. [25]. The generator first generates a set of S seed subgraphs with mean size I and L distinct labels. Next, a set of D graphs with mean size T are generated by randomly inserting the seeds into the graph. The sizes of the seeds and the graphs conform to a Poisson distribution. A seed is inserted into a graph by finding a mapping that maximizes the overlap between the seed and the graph.

We set the parameters so that the synthetic dataset has a size similar to the real dataset: $D = 10000$, $S = 100$, $I = 10$, $T = 50$ and $L = 10$. The queries are generated in the same way as in the real dataset. For GraphGrep, we set $lp = 4$. Fig. 9(a) shows the candidate/answer set size



(a) Candidate/Answer set (b) Access ratio

Figure 9. Performance on synthetic dataset

for the synthetic dataset. The candidate set size of C-tree is up to 20 times smaller than that of GraphGrep, and the accuracy is nearly 100%. The results again demonstrate the accuracy of the pseudo subgraph isomorphism test.

Fig. 9(b) shows the pruning effects of C-tree with respect to the query size. The access ratio decreases with increasing query size. The access ratios are less than that on the real dataset. This is because the answer sets on the synthetic dataset are smaller than that on the chemical dataset.

In summary, C-tree outperforms GraphGrep by up to two orders of magnitude in terms of candidate set size and index size. C-tree performs well especially on large queries where the answer sets are small.

We can indirectly compare the performance of C-tree against GIndex [11], another state-of-the-art technique. On similar experimental settings, the candidate set sizes and the index sizes for GIndex are one order of magnitude smaller than that for GraphGrep. Since the candidate set sizes and the index sizes for C-tree are up to two orders smaller than that for GraphGrep, we can conclude that C-tree outperforms GIndex in terms of candidate set size and index size.

8.2 Similarity Queries

For similarity queries, we evaluate the quality of the graph mapping methods as well as the performance of the K-NN query. Since GraphGrep and GIndex do not support similarity queries, we do not compare C-tree to them.

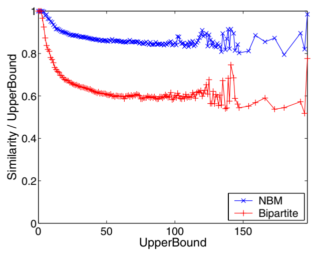
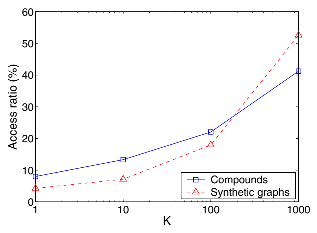
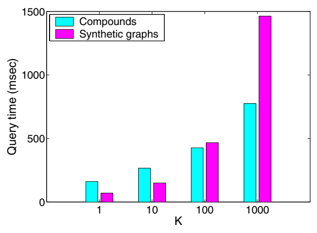


Figure 10. Quality of graph mapping methods

First, we evaluate the quality of the graph mapping methods: the NBM method and the bipartite method. Two groups of 1000 graphs are randomly selected without re-



(a) Access ratio



(b) Query time

Figure 11. K-NN query

placement from the chemical dataset. For each pair of graphs from the two groups, we compute their similarity using the graph mapping methods and compare it to the upper bound of exact similarity (Eqn. (7)). Fig. 10 shows the average ratio of similarity to the upper bound for both the methods. We do not compare it to the exact similarity since it is computationally expensive. However, the exact similarity value is, by definition, greater than the similarity reported by either of the two methods. As shown in the figure, NBM performs better than the bipartite method as its similarity scores are closer to the upper bound (and to the exact similarity). Experiments on the synthetic dataset show similar results.

K-NN queries are conducted as follows. We fix the database size to 10000, and vary K to 1, 10, 100, and 1000. For each K, we generate a set of 1000 queries, each of which is selected randomly from the dataset. We compute the average of the query results.

Fig. 11(a) shows the access ratio with respect to K on both datasets. When K=1, the access ratio is less than 10%. The access ratio scales well with increasing K. Fig. 11(b) shows the K-NN query time with respect to K. The query time is consistent with the access ratio. As shown in the figure, even when K is very large, e.g., 1000, the query time is less than 2 seconds. Therefore, it is possible to use the K-NN query as a building block for other applications, such as classification and clustering.

9 Conclusions

In this paper, we presented a comprehensive and innovative solution to graph queries. Our techniques organize a graph database into a tree using the concept of graph closure. C-tree can support various types of graphs and graph queries. Our experiments demonstrated both the high quality and the high performance of our techniques. In particular, our techniques outperform GraphGrep on subgraph queries by up to two orders of magnitude in terms of candidate set size and index size.

C-tree is the first index structure that can efficiently support both subgraph queries and similarity queries on graphs. Pseudo subgraph isomorphism is a novel approximation technique for subgraph isomorphism. For similarity queries, we measured graph distance and similarity through edit distance, and showed that it can be computed accurately using heuristic graph mapping methods.

Acknowledgements: We thank D. Shasha and R. Giugno for sharing the code of GraphGrep, and M. Kuramochi and G. Karypis for providing the synthetic graph generator. This work was supported in part by NSF grants DBI-0213903, EIA-0080134, and ANI-0123985.

References

[1] S. Berretti, A. D. Bimbo, and E. Vicario. Efficient matching and indexing of graph models in content-based retrieval. In *IEEE Trans. on Pattern Analysis and Machine Intelligence*, volume 23, 2001.

[2] E. Rahm and P. Bernstein. A survey of approaches to automatic schema matching. *VLDB J. 10(4): 334-350 (2001).*

[3] J. Lee, J. Oh, and S. Hwang. STRG-index: Spatio-temporal region graph indexing for large video databases. In *SIGMOD Conference*, 2005.

[4] KEGG. http://www.genome.ad.jp/kegg/.

[5] National Cancer Institute. http://dtp.nci.nih.gov/.

[6] H. Berman et al. The protein data bank. *Nucleic Acids Research*, (28):235–242, 2000.

[7] S. White and P. Smyth. Algorithms for estimating relative importance in networks. In *Proc. SIGKDD*, 2003.

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

[9] MeSH. http://www.nlm.nih.gov/mesh/.

[10] D. Shasha, J. T. L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. 2002.

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

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

[13] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proc. of VLDB*, 1997.

[14] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of SIGMOD*, 1984.

[15] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD Conference*, 1990.

[16] J. Hopcroft and R. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Computing*, 1973.

[17] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 1955.

[18] C. H. Papadimitriou and K. Steiglit. *Combinatorial optimization: algorithms and complexity*, pages 247–255. 1982.

[19] M. Heymans and A. K. Singh. Deriving phylogenetic trees from the similarity analysis of metabolic pathways. *Bioinformatics*, 19, 2003.

[20] C. Faloutsos and K.-I. Lin. Fastmap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *SIGMOD Conference*, 1995.

[21] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, 2000.

[22] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 1976.

[23] G. R. Hjaltason and H. Samet. Ranking in spatial databases. In *Proc. 4th Int. Symposium on Large Spatial Databases (SSD'95)*, pages 83–95, 1995.

[24] T. Seidl and H.-P. Kriegel. Optimal multi-step k-nearest neighbor search. In *SIGMOD Conference*, 1998.

[25] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proc. of ICDM*, 2001.