Similarity Evaluation on Tree-structured Data

Rui Yang Panos Kalnis Anthony K. H. Tung+
School of Computing
National University of Singapore
{yangrui,kalnis,atung}@comp.nus.edu.sg
+Contact Author

ABSTRACT

Tree-structured data are becoming ubiquitous nowadays and manipulating them based on similarity is essential for many applications. The generally accepted similarity measure for trees is the edit distance. Although similarity search has been extensively studied, searching for similar trees is still an open problem due to the high complexity of computing the tree edit distance. In this paper, we propose to transform tree-structured data into an approximate numerical multidimensional vector which encodes the original structure information. We prove that the L_1 distance of the corresponding vectors, whose computational complexity is $O(|T_1| + |T_2|)$, forms a lower bound for the edit distance between trees. Based on the theoretical analysis, we describe a novel algorithm which embeds the proposed distance into a filter-and-refine framework to process similarity search on tree-structured data. The experimental results show that our algorithm reduces dramatically the distance computation cost. Our method is especially suitable for accelerating similarity query processing on large trees in massive datasets.

1. INTRODUCTION

The use of tree-structured data in modern database applications is attracting the attention of the research community. Typical examples of huge repositories of rooted, ordered and labeled tree-structured data include the secondary structure of RNA in biology or the XML data on the web. In this paper, we study the structure similarity measure and similarity search on large trees in huge datasets. These problems form the core operation for many database manipulations (e.g., approximate join, clustering, k-NN classification, data cleansing, data integration etc). Our findings are useful in numerous applications including XML data searching under the presence of spelling errors, efficient prediction of the functions of RNA molecules, version management for documents, etc.

Trees provide an interesting compromise between graphs and the linear representation of data. They allow the expression of hierarchical dependencies where the semantics are specified implicitly by the relationship between their components; thus, the structure of the tree plays an important role in differentiating the data. The most commonly used distance measure on tree-structured data is the *tree edit distance* [23]. However, computing the tree edit distance can be very expensive both in terms of CPU cost and disc I/Os, rendering it impractical for huge datasets.

In this paper, we utilize a structure transformation to develop a novel distance function for rooted, ordered, labeled trees based on both the structure and the content. We prove that the proposed distance function is a lower bound of the tree edit distance. The idea is similar to using a set of q-grams to bound the edit distance of strings and thus filter out dissimilar strings [19]. Given a string S, a q-gram is a contiguous substring of S of length q. If S_1 and S_2 are within edit distance k, S_1 and S_2 must share at least $max(|S_1|, |S_2|) - (k-1)q - 1$ common q-grams. Similarly we characterize a tree as a set of q-level binary branches, and we show that two trees T_1 and T_2 are within edit distance k precisely when they share $[4 * (q-1) + 1] * k$ q-level binary branches. Furthermore, just as string edit distance can be tightened if the positions of the q-grams in the string are also taken into account [17, 5], so too tree-edit distance can be tightened by using information detailing the positions of q-level binary branches in the trees.

By employing our distance function as the lower bound of the edit distance in the filter-and-refine framework, we can evaluate similarity queries in two steps: In the filtering step, the lower bound is used to filter out most objects which are not possible to be in the result. The remaining objects are candidates which are validated by the original complex similarity measure during the refinement step. This strategy greatly reduces the number of expensive distance computations in the original space.

Summarizing, the contributions of our paper are:

- We propose a transformation which maps tree-structured data with the edit distance measure to numerical vectors with the standard L_1 distance. The method maintains the structure information of the original data. The novel distance is proved to be the lower bound of the general tree-edit distance.
- We describe a generalization of our mapping method which approximates the tree-edit distance at different resolutions.
- By adapting q-grams methods to use the lower bound

SIGMOD 2005 June 14-16, 2005, Baltimore, Maryland, USA.

property of the new distance, we propose novel methods to embed it into the filter-and-refine framework to facilitate the processing of similarity search.

- We present a set of experiments which compare the performance of our algorithm with the histogram filtration methods proposed in Ref. [7].

The rest of the paper is organized as follows: In Section 2 we provide the background and an overview of the related work. Section 3 presents the definition of the transformed vector space and the new distance based on it, together with the formal proof of the lower bound theorem. In Section 4 we discuss how to embed our distance function as the lower bound of edit distance into the framework for similarity search, while in Section 5 we present a thorough experimental study of our algorithms. Finally, Section 6 concludes our paper.

2. PRELIMINARIES AND RELATED WORK

In this paper, we focus on the huge dataset D of rooted, ordered, labeled trees. Here, a tree is defined as a data structure $\mathcal{T} = (N,\ E,\ Root(\mathcal{T}))$. N is a finite set of nodes. E is a binary relation on N where each pair $(u,\ v) \in E$ represents the parent-child relationship between two nodes $u, v \in N$. Node u is the parent of node v and v is one of the child nodes of u. There exists only one root note, denoted as $Root(\mathcal{T}) \in N$, which has no parent. Every other node of the tree has exactly one parent and it can be reached through a path of edges from the root. The nodes which have a common parent u (i.e., all the children of u) are siblings. $|\mathcal{T}|$ is the number of nodes in tree $\mathcal{T}$, or the size of $\mathcal{T}$.

A rooted ordered tree is a tree in which the order of the siblings from left to right is significant. A rooted, ordered, labeled tree is an ordered tree structure $T = (N,\ E,\ Root(T),\ label)$, where $label:\ N \rightarrow \Sigma$ is a total function and Σ is the finite alphabet of vertex labels. In our paper, we focus on rooted ordered labeled trees. Fig. 1 is an example of two trees T_1 and T_2.

2.1 Structural Similarity Measure

The measure of similarity between two trees T_1 and T_2 has been well studied in combinatorial pattern matching. Most studies use edit distance to measure the dissimilarity between trees (notice that similarity computation is the dual problem of distance computation). Usually there are three kinds of basic edit operations on tree nodes [23]: (i) Relabeling means changing the label of nodes; (ii) Deleting a node n means assigning the children of n to be the children of the parent of n and removing n; (iii) Inserting node n under n' means assigning a consecutive subsequence of the children of n' as the children of n, and appending n under n'. As mentioned in Ref. [23], a mapping between two trees can specify the edit operations applied to each node in the two trees graphically. The mapping must be one-to-one and it must preserve the sibling order and ancestor order at the same time. If node $u \in T_1$ is mapped to node $v \in T_2$ and $label(u) \neq label(v)$, u is relabeled to v. If $u \in T_1$ has no correspondence in the mapping, u is deleted. If $v \in T_2$ has no correspondence in the mapping, v is inserted. For example, the mapping, represented by the dashed line in Fig. 1 shows a way to transform T_1 to T_2.

For the general case, functions $\gamma(e)$ are defined to evaluate each edit operation e. The cost of a sequence of edit operations $e_1,\ e_2,\ \cdots,\ e_k$ is $\sum_{i=1}^{k} \gamma(e_i)$. The edit distance between T_1 and T_2, denoted as $EDist(T_1,\ T_2)$, is defined as the minimum cost of edit operation sequences that can transform T_1 to T_2. If the cost for each operation equals 1, the edit distance, referred to as unit cost tree edit distance, is the minimum number of operations required to transform one tree into the other. The unit cost edit distance is a distance metric and is adopted in this paper for its simplicity. However, our algorithm can be easily extended to the general edit distance measure if there is a lower bound on the cost for each edit operation.

The main differences between various tree edit distance algorithms lie in the set of allowed edit operations. Earlier work allows insertion and deletion of single nodes at the leaves only and relabeling of nodes anywhere in the tree [14]. The definitions in Ref. [23, 16, 18, 22] allow insertion and deletion of single nodes anywhere in a tree. In Ref. [22] a new distance metric based on a restriction of the mappings between two trees is proposed. The intuition is that two separate sub-trees of T_1 should be mapped to two separate subtrees in T_2. In Ref. [18], on the other hand, insertion is allowed only before deletion.

Dynamic programming algorithms are frequently used to solve the edit distance problem. To the best of our knowledge, the most commonly cited algorithm to compute general edit distance between rooted ordered labeled trees appears in Ref. [23]. The algorithm runs in $O(|T_1||T_2|)$ space and $O(|T_1||T_2|\ \min(depth(T_1), leaves(T_1))\min(depth(T_2), leaves(T_2)))$ time, where $depth(T_i), leaves(T_i),\ i = 1,\ 2$ are the depth and the number of leaf nodes of tree T_i respectively. In the worst case, the time complexity of the algorithm is $O(|T_1|^2|T_2|^2)$. Obviously, the tree-edit distance computation is both CPU and I/O expensive and is not scalable for similarity search on large trees in huge datasets.

2.2 Related Work on Tree Similarity

Although similarity search on numerical multidimensional data has been extensively studied [6, 12, 2, 20, 13], similarity search on tree-structured data has only recently attracted the attention of the research community [7]. As mentioned above, similarity evaluation of tree structured data in massive datasets based on the tree-edit distance is expensive both in terms of CPU and I/O cost; therefore, most of the previous work utilizes the filter-and-refine approach. To guarantee the filtration efficiency, the lower bound function should be a relatively precise approximation of the tree-edit distance. At the same time, it should be computationally much less expensive than the real distance. The authors of Ref. [15] proposed a pivot based approximate similarity join algorithm on XML documents. In their method, XML documents are transformed into their corresponding preorder and postorder traversal sequences. Then the maximum of the string edit distance of the two sequences is used as the lower bound of the tree-edit distance. They also proposed additional heuristics to reduce the amount of edit-distance computations between pairs of trees. However, the complexity of computing the proposed lower bounds is still $O(|T_1||T_2|)$ (i.e., the complexity of sequence edit distance computation), and it is not scalable to our problem. In order to correlate XML data streams, Garofalakis et.al. [4] proposed embedding the tree-edit distance metrics (allowing a *move* operation in addition to the basic operations) into a numeric vector space with L_1 distance norm. In their method, XML

trees are hierarchically parsed into valid subtrees in different phases. Then the multi-set of valid subtrees is obtained by parsing the tree. The vector representation is defined as the characteristic vector of the multi-set. The L_1 distance of the vectors guarantees an upper bound of distance distortion between two trees. However, the method fails to give a constant lower bound on the tree-edit distance to facilitate the retrieval of exact answers to the similarity queries based on similarity measure.

In their recent work, Kailing et.al. [7] presented a set of filters grounded on structure and content-based information in trees. They proposed using the vectors of the height histogram, the degree histogram and the label histogram to represent the structure as well as content information of trees. The lower bound of the unordered-tree edit distance can be derived from the L_1 distance among the vectors. They also suggested a way to combine filtration to facilitate similarity query processing. However, their filters are for unordered trees and cannot explore the structure information implicitly depicted by the order of siblings. Moreover, their lower bounds are obtained by considering structure and content information separately. In our approach, we suggest combining the two sources of information to provide accurate lower bounds for the tree-edit distance. In Section 5 we compare the performance of our algorithm against the histogram filtration methods.

2.3 Binary Tree Representation of Forests (or Trees)

Our proposed mapping of tree structures into a numeric vector space is based on the binary tree representation of rooted ordered labeled trees. For completeness, we briefly describe the binary tree representation of forests (or trees). We cite the formal definition of the binary tree from Ref. [9]:

Definition 1 (Binary Tree) A *binary tree* consists of a finite set of nodes. It is:

1. an empty set. Or
2. a structure constructed by a root node, the left subtree and the right subtree of the root. Both subtrees are binary trees, too.

In a binary tree, the edges between parents and the left child nodes are different from those between parents and the right child nodes. We use $T_B = (N,\ E_l,\ E_r,\ Root(T))$ to represent a binary tree. $\forall u,\ v_1,\ v_2 \in N$, if v_1 (v_2 resp.) is the left (right resp.) child of u, then $\langle u,\ v_1\rangle_l \in E_l$ ($\langle u,\ v_2\rangle_r \in E_r$ resp.). A full binary tree is a binary tree in which each node has exactly zero or two children.

There is a natural correspondence between forests and binary trees. The standard algorithm to transform a forest (or a tree) to its corresponding binary tree is through the left-child, right-sibling representation of the forest (tree): (i) Link all the siblings in the tree with edges. (ii) Delete all the edges between each node and its children in the tree except those edges which connect it with its first child. Note that the transformation does not change the labels of vertices in the tree. We can transform[1] T_1 and T_2 of Fig. 1 into $B(T_1)$ and $B(T_2)$ shown in Fig. 2, respectively. The binary tree representation is denoted as $B(T) = (N,\ E_l,\ E_r,\ Root(T),\ label)$ in our paper.

[1]The appended nodes labeled ε and the numbering of the nodes are explained in sections 3.2 and 4.2, respectively.

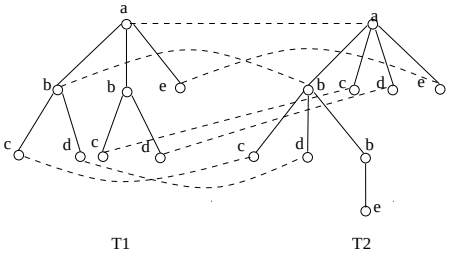


Figure 1: Tree Examples

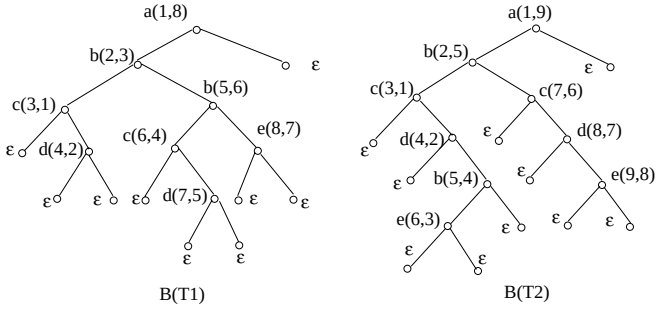


Figure 2: Normalized Binary Tree Representation

3. TREE STRUCTURE TRANSFORMATION

The key element of our algorithm is to transform rooted, ordered, labeled trees to a numeric multi-dimensional vector space equipped with the norm L_1 distance. The mapping of a tree T to its numeric vector ensures that the features of the vector representation retain the structural information of the original tree. Furthermore, the tree-edit distance can be lower bounded by the L_1 distance of the corresponding vectors. The lower bound distance evaluation is computationally much less expensive than that of $EDist(T,\ T')$. In this section, we present the transformation methods and the proof of the lower bound theorem.

3.1 Observation

We observe that edit operations change at most a fixed number of sibling relationships. This is because each node in a tree can have a varying number of child nodes but at most two immediate siblings. This is illustrated in the example of Fig. 1. The deletion of node b in T_1 incurs five changes in parent-child relationships: It destroys the (a, b), (b, c), (b, d) edges, while generating the (a, c), (a, d) edges. At the same time, this edit operation only incurs four changes in sibling relationships: The one between b and b, and the one between b and e are destroyed. The sibling relationship between b and c, and the one between d and e are generated by the deletion operation.

As mentioned in 2.3, a binary tree corresponding to a forest retains all the structure information of the forest. Particularly, in the binary tree representation, the original parent-child relationships between nodes, except the ones between each inner nodes and its first child, are removed. The removed parent-child relationships are replaced by the link

edges between the original siblings. This property makes the transformed binary tree representation appropriate for highlighting the effect of the edit-based operations on original trees.

3.2 Vector Representation of Trees

To encode the structural information we normalize the transformed binary tree representation $B(T)$ of T. In $B(T)$, for any node u, if u has no right (or left) child, we append a ϵ node (i.e., nodes labeled as ϵ do not exist in T) as u's right (or left) child. Thus we make T a full binary tree in which all the original nodes have two children and all the leaves are labeled as ϵ (as in Fig. 2). The normalized binary tree representation is defined as $B(T) = (N \bigcup \{\epsilon\},\ E_l,\ E_r,\ Root(B(T)),\ label)$, where ϵ denotes the appended nodes as well as their labels. To simplify the notation, in this paper $u \in N$ represents the node as well as its label where no confusion arises. In order to quantify change detection in a binary tree, we define the binary branch on normalized binary trees:

Definition 2 (Binary Branch) *Binary branch* (or branch for short) is the branch structure of one level in the binary tree. For a tree T, $\forall u \in N$ there is a binary branch $BiB(u)$ in $B(T)$ such that $BiB(u) = (N_u,\ E_{u_l},\ E_{u_r},\ Root(T_u))$, where $N_u = \{u,\ u_1,\ u_2\}$ $(u \in N;\ u_i \in N \bigcup \{\epsilon\},\ i = 1,\ 2)$, $E_{u_l} = \{\langle u, u_1\rangle_l\}$, $E_{u_r} = \{\langle u, u_2\rangle_r\}$ and $Root(T_u) = u$ in the normalized $B(T)$.

According to the properties of normalized binary trees, we have Lemma 3.1:

Lemma 3.1 *For each node $u \in N$ of a tree T, u may appear in at most two binary branches in the binary tree representation $B(T)$.*

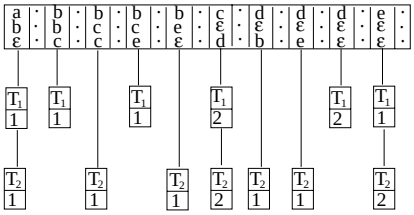
PROOF:

1. *u can occur as root in at most one binary branch. This is obvious.*
2. *u can occur as the left (or right) child in at most one binary branch. u can not occur as the left child in one branch and as the right child in another branch at the same time; otherwise, u must have two parents in $B(T)$. That is contrary to the properties mentioned in section 2.*

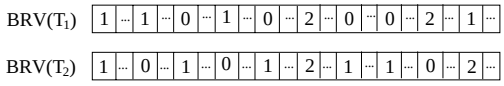
Assume that the universe of binary branches $BiB()$ of all trees in the dataset composes alphabet Γ and the symbols in the alphabet are sorted lexicographically on the string uu_1u_2. A representative vector of dimension $|\Gamma|$ can be built for each tree-structured data record, with each dimension recording the number of occurrences of a corresponding branch in the data. The formal definition of the binary branch vector is given in Definition 3.

Definition 3 (Binary Branch Vector) The *binary branch vector* $BRV(T)$ of a tree T is a vector $(b_1, b_2, \cdots b_{|\Gamma|})$, with each element b_i representing the number of occurrences of the ith binary branch in the tree. $|\Gamma|$ is the size of the binary branch space of the dataset.

We can first build an inverted file for all binary branches, as shown in Fig. 3(a). An inverted file has two main parts: a vocabulary which stores all distinct values being indexed, and an inverted list for each distinct value which stores the identifiers of the records containing the value. The vocabulary here consists of all existing binary branches in the datasets. The inverted list of each component records the number of occurrences of it in the corresponding trees. The resulting vectors of our transformation for the trees in Fig. 1 and the normalized binary trees in Fig. 2 are shown in Fig. 3(b).



(a) Inverted File



(b) Binary Branch Vectors

Figure 3: Binary Branch Vector Representation

Based on the vector representation, we define a new distance of the tree structure as the L_1 distance between the vector images of two trees:

Definition 4 (Binary Branch Distance) Let $BRV(T_1) = (b_1,\ b_2,\ \cdots,\ b_{|\Gamma|})$, $BRV(T_2) = (b'_1, b'_2, \cdots b'_{|\Gamma|})$ be the binary branch vectors of trees T_1 and T_2 respectively. The binary branch distance of T_1 and T_2 is $BDist(T_1,\ T_2) = \Sigma_{i=1}^{|\Gamma|}|b_i - b'_i|$

The binary branch distance has the properties listed below: For all T_1, T_2 and T_3 in the dataset,

1. $BDist(T_1,\ T_2) \geq 0$, and $BDist(T_1,\ T_1) = 0$
2. $BDist(T_1,\ T_2) = BDist(T_2,\ T_1)$
3. $BDist(T_1,\ T_3) \leq BDist(T_1,\ T_2) + BDist(T_2,\ T_3)$

   PROOF

   The first two properties are obvious. For the third property, let $BRV(T_i) = (b_{i1},\ b_{i2},\ \cdots,\ b_{i|\Gamma|})$ for $i = 1,\ 2,\ 3$.

$$\begin{aligned} &BDist(T_1,\ T_2) + BDist(T_2,\ T_3) \\ &= \Sigma_{j=1}^{|\Gamma|}|b_{1j} - b_{2j}| + \Sigma_{j=1}^{|\Gamma|}|b_{2j} - b_{3j}| \\ &\geq \Sigma_{j=1}^{|\Gamma|}|b_{1j} - b_{3j}| \ = \ BDist(T_1,\ T_3) \end{aligned}$$

The third property means that the binary branch distance satisfies the triangular inequality. However, $BDist(T_1,\ T_2) = 0$ cannot imply that T_1 is identical to T_2. This is illustrated in Fig. 4, where both trees have the same binary branch vector. So the binary branch distance is not a metric on tree-structured data.

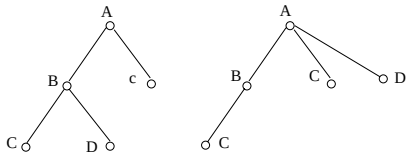


Figure 4: Trees with 0 Binary Branch Distance

3.3 Lower Bound of Edit Distance

In this section, we prove our main theorem.

Theorem 3.2 *Let T and T' be two trees. If the tree-edit distance between T and T' is $EDist(T, T')$, then the binary branch distance between them satisfies the following: $BDist(T, T') \leq 5 \times EDist(T, T')$*

PROOF: *The theorem follows if we show that at most $5 \times k$ binary branch distance is incurred by k edit operations. Assume that edit operations $e_1,\ e_2,\ \cdots,\ e_k$ transform T to T'. Accordingly, there is a sequence of trees $T = T_0 \rightarrow T_1 \rightarrow \cdots \rightarrow T_k = T'$, where $T_{i-1} \rightarrow T_i$ via e_i for $1 \leq i \leq k$. Let there be k_1 relabeling operations, k_2 insertions and k_3 deletions. $k_1 + k_2 + k_3 = k$. It is sufficient to prove the theorem for one step of the transformation.*

1. *Assume that e_i is a relabeling operation on some node v of the tree. According to Lemma 3.1, v occurs in at most two binary branches in $B(T_{i-1})$. In these two branches, $label(v)$ is changed to the new one in the target tree $B(T_i)$. Assume that the count of the two binary branches in $BRV(T_{i-1})$ is in dimension l_1 and l_2, while the two new binary branches are in dimension l_3 and l_4. Then $BRV(T_{i-1})[l_m] - BRV(T_i)[l_m] = 1$, for $m = 1, 2$. $BRV(T_{i-1})[l_{m'}] - BRV(T_i)[l_{m'}] = -1$, for $m' = 3, 4$. So, $BDist(T_{i-1}, T_i) \leq 4$.*

2. *Assume that e_j inserts a node v to transform T_{j-1} to T_j. Obviously, when v has a parent, a left sibling, a right sibling and child nodes, this operation leads to the maximum number of changes on the structure information. Fig. 5 and Fig. 6 demonstrate the insertion operation and the changes it causes on the binary tree representation. Let v be inserted under node v' and child nodes $w_{l+1}, \cdots w_{l+m}$ of v' in T_{j-1} become the child nodes of v in T_j.*

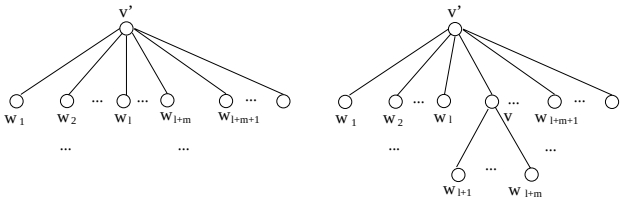


Figure 5: Insertion of Node v Under Node v'

We show that at most five changes occur on the edges of $B(T_{j-1})$: Two edges $\langle v,\ w_{l+1}\rangle_l$ and $\langle v,\ w_{l+m+1}\rangle_r$ representing the structure information rooted on v are added into the binary tree. These edges comprise the binary branch $BiB(v)$. So, assuming that it corresponds to dimension l in $BRV(T_j)$, then $BRV(T_j)[l] - BRV(T_{j-1})[l] = 1$. In addition, the sibling relationship between w_l and w_{l+1}, and between w_{l+m} and w_{l+m+1} in T_{j-1} (represented by $\langle w_l,\ w_{l+1}\rangle_r$ and $\langle w_{l+m},\ w_{l+m+1}\rangle_r$ respectively in $B(T_{j-1})$) are destroyed. This leads to the destruction of one of each binary branch $BiB_{T_{j-1}}(w_l)$ and $BiB_{T_{j-1}}(w_{l+m})$. Thus, the values for the two corresponding dimensions in $BRV(T_j)$ are less than those in $BRV(T_{j-1})$ by 1. Finally, $\langle w_l,\ w_{l+1}\rangle_r$ is replaced by $\langle w_l,\ v\rangle_r$ in $B(T_j)$ for v is the right sibling of w_l after being inserted in T_j. $\langle w_{l+m},\ w_{l+m+1}\rangle_r$ is replaced by $\langle w_{l+m},\ \epsilon\rangle_r$ for w_{l+m} is the right most child of v in T_j after insertion. Then the values of the corresponding two dimensions in $BRV(T_j)$ are larger than those in $BRV(T_{j-1})$ by 1 each. To sum up, $BDist(T_{j-1},\ T_j)$ is at most 5.

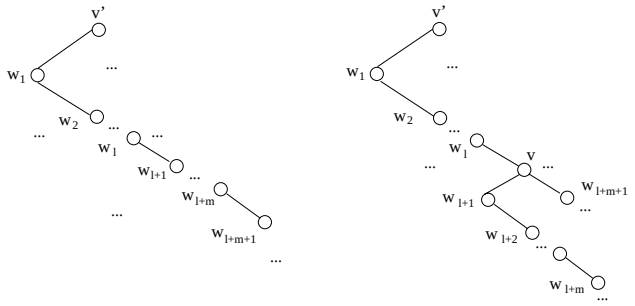


Figure 6: Changes of Binary Tree Incurred by Insertion

3. *Deletion is complementary to insertion. Therefore the number of affected binary branches must be bounded by the same amount as for insertion.*

According to the triangular inequality property of binary branch distance, we have

$$BDist(T,\ T') \leq BDist(T_0,\ T_1) + BDist(T_1,\ T_2) + \cdots + BDist(T_{k-1},\ T_k)$$
$$\leq 4 \times k_1 + 5 \times k_2 + 5 \times k_3 \leq 5 \times k$$
$$\leq 5 \times EDist(T,\ T').$$

3.4 Extended Study

Our generalized analysis is similar to the q-gram method [19] for solving the k-difference problem of strings. The number of occurrences of each q-gram (i.e., all strings of length q over the alphabet) in any two strings are counted. If two strings are similar, they have many q-grams in common. Formally, if the edit-distance of strings S_1 and S_2 is k, then they have at least $\max(|S_1|, |S_2|) - (k-1)q - 1$ q-grams in common. When applied to similarity search problems in which the full strings are involved, the q-gram method usually trades off the false positive for the false negative rate by adjusting the length of the q-gram searched [8]. Binary branches can be viewed as playing the role of q-gram structures for tree data. The vector images of trees can be extended to record multiple level binary branch profiles. We first give the formal definition of the q-level binary branch:

Definition 5 The q-level binary branch $BiB\_Q(n_0,\ n_1,\ \cdots,\ n_{2^q-2})$ is the perfect binary tree of height $q-1$, where $n_0,\ n_1,\ \cdots,\ n_{2^q-2}$ is the sequence obtained by preorder traversing the perfect binary tree (with all leaf nodes at the same depth and all internal nodes having degree 2).

The binary branch defined in the previous section is indeed the two-level binary branch. Similar to the computation of q-grams for strings, our sliding window is a perfect binary tree with height $q-1$. The sliding window shifts one level each time along the path from the root to the leaves. For each node u in the tree, there is a q-level binary branch rooted at u in the binary tree representation. If the subtree of height $q-1$ rooted at u is not a perfect binary tree in the transformed representation, we can append ϵ-nodes to complete it. The multiple level binary branch is used to maintain structures of fixed size and fixed shape in the original data. Obviously, it encodes more information than the two-level binary branch. We can extend the binary branch vector to the *characteristic vector* $BRV\_Q(T)$, which includes all the elements in the q-level binary branch space. The q-level binary branch distance $BDist\_Q(T,\ T')$ is defined as the L_1 vector distance between the images of the trees T and T' under the q-level mapping.

Theorem 3.3 *Let T and T' be two trees. If the tree-edit distance between T and T' is $EDist(T,T') = k$, then the q-level binary branch distance between them $BDist\_Q(T,T') \leq [4 \times (q-1)+1] \times k$*

The proof methods here are similar to those of Theorem 3.2. Note that each node in T appears in at most q q-level binary branches in $B(T)$. We do not include the details of the proof due to space constraints.

The binary branch distance $BDist\_Q(T_1, T_2)$ increases as the level of the binary branch q increases. This is due to the fact that the higher the level is, the more information of the tree structure is encoded in the binary branches. At one extreme, q is equal to the height of the normalized transformed binary tree; then all the structural information of the original tree is encoded. However, in such a situation, the filter algorithm is of no use. At the other extreme q is equal to 1; in this case, the filter efficiency is too low. We do not discuss this option as it records no structure information of the original tree at all. According to Theorem 3.3, $BDist\_Q(T,\ T')/[4(q-1)+1]$ can be used as a series of approximations for the tree-edit distance with different resolutions. So the level q of the binary branch can be adjusted to improve filter efficiency when solving the similarity search problem.

4. ENHANCEMENT OF SIMILARITY SEARCH

Previously, we mapped the tree structures and the tree edit distance metric to a numeric vector space and the L_1 norm distance. Although, according to its properties, the binary branch distance is not a metric, it approximates and lower bounds the tree-edit distance metric. Just as q-gram methods can be used to speed up similarity search for strings, the distance-embedded lower bounds can be integrated into the filter-and-refine framework to speed up similarity search by reducing the number of expensive similarity distance computations. In this section, we present our filter-and-refine algorithm for processing similarity search on the tree-structured data by exploiting the lower bounds.

4.1 Basic Algorithm

Similarity search on various data usually refers to range queries and k nearest neighbor queries. Range queries find all objects in the database which are within a given distance τ from a given object; k nearest neighbor (k-NN) queries find the k most similar objects in the database which are closest in distance to a given object. Other types of search can be composed by these two similarity queries. When searching tree-structured data, similarity is measured by tree-edit distance.

As mentioned above, the similarity evaluation of large trees in massive datasets based on tree-edit distance is a computationally expensive operation. Traditionally, the filter-and-refine architecture is utilized to reduce real distance computation by employing the lower bounds of the real distance [13]: In the first step (i.e., filtration), objects that cannot qualify are filtered out. In the second step (i.e., refinement), verification of the original complex similarity measure is necessary only for the candidates filtered through. The objects satisfying the query predicate are reported as results. The completeness of the results is guaranteed by the lower bound property: If the lower bound distance is greater than the query range, it is safe to filter out the data since its real edit distance cannot be less than that range.

Our method is to embed an easy-to-compute distance function that is the lower bound of the actual tree edit distance into the filter-and-refine framework. The optimistic bound used by the similarity search is based on the binary branch vector distance of the trees. In addition to the number of occurrences of individual binary branch, the positional information of the binary branch is also important in exploring the structure information of the trees. We will focus on the two-level binary branch. However, our approach can be easily generalized to q-level binary branches.

4.2 Optimistic Distance for Similarity Queries

The efficiency of the filter-and-refine architecture is based on the hypothesis that the lower bound function is much quicker to evaluate than real distance. As shown in Section 3.3, binary branch distance lower bounds edit distance effectively: The lower bound function can be computed in $O(|T| + |T'|)$ time, which is much more succinct than edit distance computation. Like using the q-gram methods to solve the approximate string matching problem, not only the occurrences of the q-grams, but also their positions can be exploited to measure the similarity of the pattern and certain subsequence of the strings [17, 5]. The idea is that: given two strings with distance less than l, two identical q-grams in the two strings respectively cannot be matched if their positions differ by more than l.

Binary branch filtration also exhibits this property. First, we give the following proposition:

Proposition 4.1 *Given T_1 and T_2 with edit distance less than l, we number each node u by its position in the preorder traverse of its tree. In the mapping corresponding to the edit distance, the node $u \in T_1$ cannot be mapped to $v \in T_2$ if the difference of the numbers of u and v is larger than l.*

PROOF(sketch): *In the preorder traversal numbering, the numbers which are smaller than that of u are assigned to ancestors of u or the nodes that are to the left of u, while the ones that are larger than that of u are assigned to descendants or the nodes to the right of u. Since the edit operation mapping preserves sibling order and ancestor order, if the two nodes are matched, and their number difference is larger than l, then there must be more than l deletions*

or insertions. This is contrary to the premise that the edit distance is less than l.

The property is also applicable to postorder numbering of the nodes.

For each binary branch $BiB(u, u_1, u_2)$, we define the positional structure $(BiB(u, u_1, u_2),\ pre(u),\ post(u))$, where $pre(u)$ and $post(u)$ denote the position of u in the preorder and the postorder traversal sequence of T respectively (resp. the preorder traverse and inorder traverse of $B(T)$). Based on the positional binary branch, we define the mapping $M(T_1, T_2, pr)$ between the positional binary branches of T_1 and T_2 with positional range pr, which is any set of pairs of positional binary branches $((BiB(u, u_1, u_2),\ pre(u),\ post(u)),\ (BiB(v, v_1, v_2),\ pre(v),\ post(v)))$ satisfying:

1. the mapping is one-to-one;
2. $BiB(u, u_1, u_2) = BiB(v, v_1, v_2)$;
3. $|pre(u) - pre(v)| \leq pr$ and $|post(u) - post(v)| \leq pr$.

For the example in Fig. 2, the numbering beside each node is the position specification of the corresponding binary branch. Assume the positional range $pr = 1$. It is obvious that $(BiB(c, \epsilon, d), 3, 1)$ in T_1 can only be mapped to $(BiB(c, \epsilon, d),\ 3, 1)$ in T_2; While $(BiB(c, \epsilon, d), 6, 4)$ and $(BiB(c,\ \epsilon, d), 7, 6)$ cannot be mapped to each other. $(BiB(e, \epsilon, \epsilon), 8, 7)$ in T_1 can be mapped to $(BiB(e, \epsilon, \epsilon), 9, 8)$ in T_2, but cannot be mapped to $(BiB(e, \epsilon, \epsilon), 6, 3)$.

For two trees T_1 and T_2, we denote the maximum-sized mapping as $M_{max}(T_1, T_2, pr)$. The subset of it which is related to a given binary branch $BiB \in \Gamma$ is denoted as $M'_{max}(T_1, T_2, BiB, pr)$. Obviously, $M'_{max}(T_1, T_2, BiB, pr)$ is the maximum-sized mapping on the binary branch BiB. Given the preorder and postorder position sequences of BiB in T_1 and T_2 in ascending order, $|M'_{max}(T_1, T_2, BiB, pr)|$ (size of $M'_{max}(T_1, T_2, BiB, pr)$) can be computed in linear time. We define a new distance between two trees based on $|M'_{max}(T_1, T_2, BiB, pr)|$:

Definition 6 (Positional Binary Branch Distance) Given two trees T_1 and T_2, their binary branch vectors $BRV(T_i) = (b_{i1}, b_{i2}, \cdots, b_{i|\Gamma|})$ $(i = 1, 2)$ and the positional range specification pr, the positional binary branch distance with range pr is $PosBDist(T_1, T_2, pr) = \sum_{j=1}^{|\Gamma|}(b_{1j} + b_{2j} - 2|M'_{max}(T_1, T_2, j, pr)|)$

Proposition 4.2 *If $PosBDist(T_1, T_2, l) > 5 \times l$, then $EDist(T_1, T_2) > l$.*

PROOF(sketch): *We prove the contrapositive proposition: If the edit distance is less than l, then $PosBDist(T_1, T_2, l) \leq 5 \times l$. According to the definition of positional binary branch, $PosBDist(T_1, T_2, l)$ differs from $BDist$ in that, in T_1 and T_2, it does not match the same binary branches whose position differences are larger than l. For any positional binary branch $(BiB(u, u_1, u_2), pre(u), post(u))$, if there is no element in $M_{max}(T_1, T_2, l)$ that corresponds to it, the node u should be changed by some edit operation. According to Definition 6, the positional binary branch distance is the sum of the differences on the binary branch incurred by the edit operations to change T_1 to T_2. And according to Theorem 3.2, one edit operation changes at most 5 binary branches. Thus $PosBDist(T_1, T_2, l) \leq 5 \times l$.*

Obviously the positional binary branch distance is related to the positional range specification. Theoretically, the positional range for two trees T_1 and T_2 can increase from $pr_{min} = 0$ to $pr_{max} = |T_1| + |T_2|$ and the positional binary branch distance decrease correspondingly. Given $pr = pr_{min}$, the corresponding positional binary branch distance computed has the maximum possible value: $PosBDist(T_1, T_2,\ pr_{min}) = PosBDist_{max}$, computed by matching only the identical binary branches which have the same positions. Apparently, $PosBDist_{max}/5 > pr_{min}$. Given $pr = pr_{max}$, the corresponding positional binary branch computed has the minimum possible value $PosBDist(T_1, T_2,\ pr_{max}) = PosBDist_{min} = BDist\ (T_1,\ T_2)$. It is obvious that

$$PosBDist_{min}/5 \leq EDist(T_1,\ T_2) \leq pr_{max}$$

Then, there must be a given positional range pr_i s.t. $pr_{min} \leq pr_i \leq pr_{max}$ which is the maximum positional range that satisfies $PosBDist(T_1, T_2, pr_i)/5 > pr_i$. According to the analysis of Proposition 4.2, $EDist(T_1,\ T_2) \geq (pr+1)$, where $pr = pr_{min}, \cdots, pr_i$. Thus, $(pr_i + 1)$ is a lower bound of edit distance. Note that $BDist(,)$ is the minimum value for $PosBDist$ and that for $pr_i + 1$, $PosBDist(T_1, T_2, pr_i + 1)/5 \leq (pr_i + 1)$, so we have:

$$BDist(T_1, T_2)/5 \leq PosBDist(T_1, T_2, pr_i + 1)/5 \leq (pr_i + 1)$$

Thus, $pr_i + 1$ is a closer lower bound of edit distance between T_1 and T_2 than $BDist(T_1,\ T_2)/5$. A better optimistic bound, pr_{opt}, of the edit distance can be obtained by searching the minimum value of the positional range pr_i $(pr_{min} \leq pr_i \leq pr_{max})$ satisfying $PostBDist(T_1, T_2, pr_i)/5 \leq pr_i$. In practice, we can reduce the search range further. Since $EDist(T_1, T_2) \geq ||T_1| - |T_2||$, $pr_{min} = ||T_1| - |T_2||$. At the same time, it is meaningless to set the pr_{max} to be larger than $max(|T_1|, |T_2|)$;

4.3 Similarity Search Algorithm

In this section, we present the algorithm for constructing vectors and a novel filter-and-refine algorithm for similarity search utilizing the positional binary branch distance. The steps of vector construction and k-NN search are listed in Algorithm 1 and Algorithm 2 respectively.

In the vector construction algorithm, we utilize an extended inverted file IFI to build the vector representation. The inverted list of each binary branch records the data record Tid, the number of occurrences of this branch and the respective positions at which it appears in the corresponding data. Firstly, each tree-structured data is recursively traversed and the IFI is constructed by calling the function $Traverse(\ )$ to obtain the binary branch information in Fig. 2. In the function, the binary branch BiB_R of the current node is built by calling the $getFirstChild()$ and $getNextSibling()$ functions of the parser. Then, in function $insertPreOrder(\ )$, the corresponding entry of BiB_R in IFI is found by some hashing function. Then the component for data T (identified by Tid) at the end of inverted list is updated: The number of occurrences is increased by 1. The preorder position of the branch is recorded. In function $insertPostOrder(\ )$, the postorder position is recorded. After the construction of IFI, we build the sparse vector representation of each data by scanning IFI: For each branch that occurs in the data, we record the id of the branch and the number of its occurrences in the vector. In addition, two arrays recording the branch positions (for preorder and

Algorithm 1 *vector construction*

Input:
The data set D
Output:
The vector representations of the data BRV,
The preorder positions $preOrderPos$,
The postorder positions $postOrderPos$,

1: initialize the inverted file index IFI to be empty;
2: **for** each record $T \in D$ **do**
3: $PrePosition = 0$;
4: $PostPosition = 0$;
5: $Traverse(Root(T), PrePosition, PostPosition, IFI)$;
6: $l = 0$;
7: **for** each entry i in IFI **do**
8: **for** each entry j in the inverted list of i **do**
9: $k = IFI[i][j].Tid$;
10: $BRV[k][l].Bib \leftarrow i$;
11: $BRV[k][l++].Count \leftarrow IFI[i][j].occurrence$;
12: Build positional sequence $preOrderPos[k]$;
13: Build positional sequence $postOrderPos[k]$;

Function:
$Traverse(R, \&\ Preorder, \&\ Postorder, IFI)$

1: construct binary branch BiB_R of R by calling $getFirstChild(R)$ and $getNextSibling(R)$; {two level binary branch}
2: $Preorder{+}{+}$;
3: $insertPreOrder(Tid, BiB_R, IFI, Preorder)$;
4: **for** each child node r_i of R **do**
5: $Traverse(r_i, Preorder, Postorder, IFI)$;
6: $Postorder{+}{+}$;
7: $insertPostOrder(Tid, BiB_R, IFI, Postorder)$;

Algorithm 2 *k-NN on tree-Structured data*

Input: The data set D,
The vector representation of data BRV,
The preorder positions $preOrderPos$,
The postorder positions $postOrderPos$,
The query T_q;
Output:
The result set of k nearest neighbors of T_q

1: construct vector and position arrays vec_{T_q}, $preOrderPos_{T_q}$ and $postOrderPos_{T_q}$ for T_q;
2: **for** each vector i in BRV **do**
3: $LowerBound[i] = SearchLBound(vec_{T_q}, BRV[i], preOrderPos[i], postOrderPos[i], preOrderPos_{T_q}, postOrderPos_{T_q})$;
4: sort the $LowerBound$ and D into $LowerBound'$ and D' in ascending order of the lower bound distances;
5: initialize the max heap KNN, s.t. capacity(KNN)=k;
6: **for** i From 0 To $|D|$ **do**
7: **if** $(KNN.size = k)$AND$(LowerBound'[i] > KNN[0].key)$ **then**
8: BREAK;
9: Retrieve the corresponding data T_i;
10: $editDist = EDIST(T_i, T_q)$;
11: **if** $KNN.size$ is less than k **then**
12: insert T_i with the key $editDist$ in KNN ;
13: **else**
14: pop up $KNN[0]$;
15: insert and push down T_i with the key $editDist$ in KNN;
16: return KNN;

Function:$SearchLBound(vec_{T_q}, BRV_i, preOrderPos_i, postOrderPos_i, preOrderPos_{T_q}, postOrderPos_{T_q})$

1: $pr_{min} = diff(size(vec_{T_q}), size(BRV_i))$;
2: $pr_{max} = max(size(vec_{T_q}), size(BRV_i))$;
3: $PosBDist_{max} = PosDiff(vec_{T_q}, BRV_i, preOrderPos_i, postOrderPos_i, preOrderPos_{T_q}, postOrderPos_{T_q}, pr_{min})$;
4: **if** $PosBDist_{max}/5 \leq pr_{min}$ **then**
5: Return pr_{min};
6: **while** $pr_{min} \leq pr_{max}$ **do**
7: $pr_{half} = (pr_{min} + pr_{max})/2$;
8: $PosBDist = PosDiff(vec_{T_q}, BRV_i, preOrderPos_i, postOrderPos_i, preOrderPos_{T_q}, postOrderPos_{T_q}, pr_{half})$;
9: **if** $PosBDist/5 \leq pr_{half}$ **then**
10: $pr_{max} = pr_{half} - 1$;
11: **else**
12: $pr_{min} = pr_{half} + 1$;
13: Return $pr_{half} + 1$;

postorder respectively) are constructed from IFI. Both are sorted according to the branches and in ascending order.

The procedure for k-NN search is shown in Algorithm 2. First, the query T_q is preprocessed to construct the vector representation and position sequences. In line 3, the optimistic bound of the distance between the query and each data object is computed by calling function $SearchLBound$; In the function, the optimistic bound is searched for in the range $[diff(size(vec_{T_q}), size(BRV[i])),\ max(size(vec_{T_q}),\ size(BRV[i]))]$. Since the search range is ordered, we use the binary search algorithm. In line 3 and line 8 of function $SearchLBound(\ )$, the distance $PosBDist()$ are computed based on $|M'_{max}()|$. After the optimistic bounds of all the vectors are obtained, at line 4 of the main function, the $LowerBound$ array and the data tree id are sorted in ascending order of the optimistic bounds to ensure that vectors of high possibility in being the results are processed before others. Second, the pruning procedure of traditional filter-and-refine similarity search steps are adopted [13, 1, 10, 11] to reduce real distance computation. Finally, refinements are done by verifying the candidates.

Range query processing is similar to k-NN query processing; The difference is that there is a specified range τ for the query. According to Proposition 4.2, $\max(PosBDist(T, T_q, \tau)/5, pr_{opt})$ should be considered as the optimistic bound in the filtering step.

4.4 Complexity Analysis

In this part, we analyze the time and space complexities of our vector construction method and optimistic bound computation method. In order to calculate running time complexity, we consider each step of the algorithm. Assume that the size of the dataset, i.e., the total number of tree data objects, is $|D|$. For record T_i, there are $|T_i|$ nodes in it. The vocabulary of inverted file IFI is implemented by one hashing function. According to Algorithm 1, function $Traverse()$ is called recursively to traverse each node and insert the binary branch information of the current node into IFI. Each time the new entries are appended at the end of the inverted list. So each update of IFI is of constant time complexity. Thus, the IFI construction is of linear complexity. As we store in IFI only the existing vocabulary of the dataset, the worst case is that all the nodes in the datasets have got different binary branches. Thus, the size of the vocabulary is at most $\sum_{i=1}^{|D|} |T_i|$. In addition, each node in each tree has one corresponding entry in the inverted list. In total, the space complexity of IFI is also $O(\sum_{i=1}^{|D|} |T_i|)$. To build the vector representation, the whole IFI has to be scanned once. So the time and space complexities of the whole vector construction algorithm are both $O(\sum_{i=1}^{|D|} |T_i|)$.

Next, we analyze the optimistic bound computation com-

plexity in our query processing method. Given one query T_q, we need to compare its vector and its positional sequence with those of each data T_i. As mentioned in section 4.3, we use the binary search algorithm to obtain the optimistic bound between $||T_i|-|T_q||$ and $max(|T_i|, |T_q|)$. Each search process is of linear complexity $O(|T_i| + |T_q|)$. Then the time complexity for this step is $O(\sum_{i=1}^{|D|}(|T_i| + |T_q|) \times log(min(|T_i|, |T_q|)))$, and the space complexity is $O(\sum_{i=1}^{|D|} |T_i|+ |T_q|)$.

5. EXPERIMENTAL RESULTS

In this section, we compare the performance of our filter-and-refine similarity search algorithm which integrates binary branch distance and the lower bound of edit distance (denoted as $BiBranch$ in Figure 7 through 15) against the histogram filtration methods proposed in [7] (denoted as $Histo$ in those figures). Through experiments on synthetic datasets, we show our algorithm's sensitivity to different features of the data. We also present our experiments on real dataset to show the algorithm's performance on different query characteristics. Finally, we discuss the effect of level q on the algorithm. We conducted all the experiments on a workstation with Intel Pentium IV 2.4GHz CPU and 1GB of RAM. We implemented our algorithms and the algorithms proposed in [7] in C++.

The synthetic data generator is similar to that of [21], except that we do not need to simulate website browsing behavior, but instead have to control the data distance. The program constructs a set of trees based on specific parameters. Four groups of parameters, the fanout of tree nodes, the size of trees, the number of labels and the edit operations are all random variables conforming to some distributions. The fanout and the size of the trees are sampled from normally distributed values, denoted by $N\{x_1, x_2\}$, where x_1 and x_2 are the mean and standard deviation of the normal distribution. The number of labels in the dataset is denoted by Ly, where y is its value. We allow multiple nodes in each tree to have the same label. For example, the specification $N\{4, 0.5\}N\{50, 2\}L8$ means that in the generated trees, the fanout of nodes conforms to normal distribution with mean 4 and variance 0.5. The total number of nodes in each tree conforms to normal distribution with mean 50 and standard deviation 2. And there are eight labels in the whole dataset. We also use another parameter Dz, the decay factor, to explicitly specify the distribution of the edit operations. The generator consists of the following steps: Firstly, a given number of seeds of the dataset are generated according to the first three groups of parameters. At the beginning of each seed generation, the maximum size is randomly sampled from $N\{50, 2\}$. Then, the tree grows by breadth first processing. The label of current node is sampled uniformly from the eight labels. Next, we check whether the current size of the tree exceeds the maximum size. If so, the process terminates. Otherwise, the number of children of current node is sampled from $N\{4, 0.5\}$. Secondly, new tree is generated from one of the seeds by changing each node of it with the probability specified by Dz. The changes are equiprobably insertion, deletion, and relabeling. The data generated from the seeds is used as the seed for the next data generation. In our experiments, we adopted 0.05 as the decay factor. Experiments with other settings had similar results.

For the real datasets, we used DBLP, which consists of bibliographic information on major computer science journals and proceedings. It is of XML document format and includes very bushy and shallow trees in the repository. The average depth is 2.902, and there are 10.15 nodes on average in each tree.

In each experiment, 100 queries were randomly selected from the dataset. The results shown in this paper were all averaged on the queries. We adopt the CPU time consumption as performance measures. As real edit distance computation is the most costly part of similarity search on tree-structured data, the percentage of data which are not filtered out and for which the real distances have to be evaluated is an important measure of the algorithm efficiency. It is defined as:

$$\left(\frac{|True\ Positive| + |False\ Positive|}{|Dataset|}\right) \times 100\%$$

Timings were based on processor time. As the source code of histogram filtration was not available, for time consumption, we compared our filter-and-refine algorithm with the sequential search algorithm.

For the histogram filtration algorithm, three types of histogram vectors are used: One histogram records the distribution of heights of every node in the tree, a second records the fanouts for each of the nodes, and a third records the distribution of labels used. As mentioned in section 4.3, in the binary branch vector, only the non-zero dimension is stored. Also, the positional information for binary branches is stored for each node which equals to the size of the trees. To use equal amount of space, we set the sum of dimension of the three type histogram vectors for one tree to be the averaged vector size plus two averaged tree size in a given dataset.

5.1 Sensitivity Test

In the first set of experiments, we carried out a series of sensitivity analysis to the parameters of the dataset. The first three arguments of the data generator were set with different distributions. All the datasets generated included 2000 trees. Figure 7 to Figure 11 show the relative performance of the methods for various parameter settings. They compare the percentage of accessed data for the binary branch filtration and the histogram filtration (shown as the bars in the figures) and the CPU time consumption of the binary branch filtration and the sequential search (shown as the lines). The results shown are for range queries as well as k-NN queries. Each range was set to be the 1/5 of the average distance among the whole datasets. For k-NN queries, we retrieved 0.25% of the trees of the dataset.

Figure 7 and figure 8 illustrate the performance of the two algorithms when the fanout varied. The mean values of it in the four datasets increased from 2 to 8 with the variance fixed to be 0.5. In order to analyze the effect of fanout, we diminished the effect of tree size and label number. The mean values of the tree size in the four datasets were all 50, and the standard deviation was limited to 2. Thus most trees in the datasets should have a size range from 46 to 54. The label number for each dataset was fixed at 8. We can see that the binary branch filtration accessed at most 3.35% of the number of data objects accessed by the histogram filtration for the range queries and at most 23.08% for the k-NN queries. We can see that when fanout was 2, both filtration methods accessed the most data. The reason is

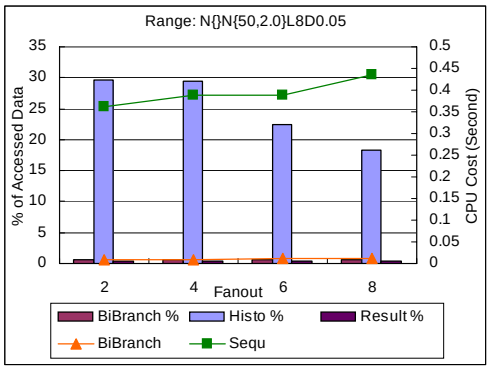


Figure 7: Sensitivity to Fanout Variation for Range Queries

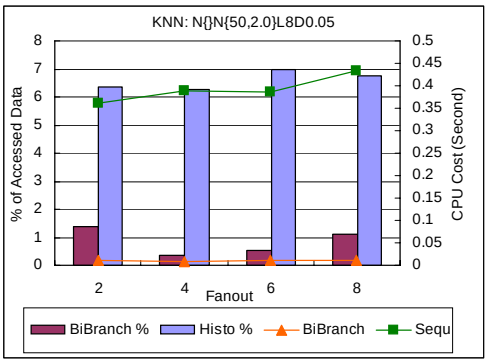


Figure 8: Sensitivity to Fanout Variation for k-NN Queries

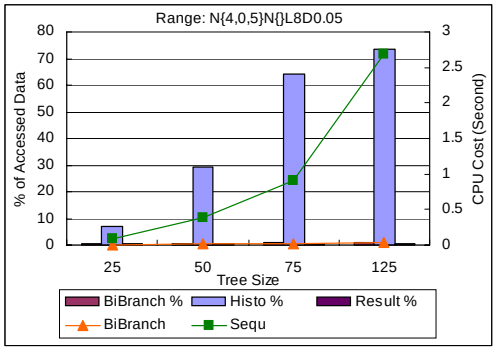


Figure 9: Sensitivity to Size of Trees for Range Queries

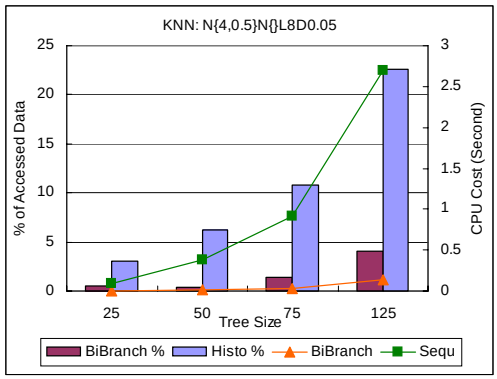


Figure 10: Sensitivity to Size of Trees for k-NN Queries

that the probability that the fanout of nodes is 0 is much higher when the mean is set to be 2. Then the structure distance in this dataset is larger since the variation of height is larger than other sets. When the fanout is increased to 4, the height difference becomes much less. We also see that with increasing fanout, the histogram filtration accessed less data for range queries. This is because degree histogram yields better filtration power for larger fanout. However, for the k-NN queries, similar trends did not appear since the mean of the real distance increased as the fanout increased, and the search radius had to grow to retrieve the k most similar data [3]. In Figure 8, for the binary branch filtration, when the access rate is only 1.4%, the time consumption of binary branch distance evaluation is only 1.92% of the CPU cost of sequential query processing. This is consistent with the theoretical analysis that real distance consumption is overwhelming. So the extra costs incurred by the filtering can be ignored.

Figure 9 and Figure 10 show the percentage of accessed data and CPU cost when the mean size of trees varied. The results of the k-NN queries are similar to that of the range queries. In these experiments, the fanout of the datasets conformed to $N\{4, 0.5\}$. The label size is set as 8. The mean tree size varied from 25 to 125, and in each of the four datasets, all the tree size values conformed to normal distribution with variance of 2. The results show that for the range queries, the percentages of accessed data with binary branch filtration were almost the same as the result size for various tree size values. Histogram filtration needed to access much more data to process the same queries on the same dataset. When the mean value of tree size was 125, the binary branch filtration outperformed histogram filtration by more than a factor of 70 for range queries. The reason is that with label number and fanout almost fixed, the height, degree and the label histograms could vary little. The histogram information blurs the distance identification. On the other hand, the increase of size led to the increase of the edit distances. So the larger size caused worse performance of both our algorithm and histogram filtration methods. However, binary branch filtration still outperformed histogram filtration for various tree size. As can be seen, when the mean values of the tree size increased, the time consumption for the computation of the real distances increased quadratically. So, although the result size was almost the same, the sequential search time was too long for the datasets with large size. Thus, our algorithm is quite efficient for the similarity search on the large trees.

Figure 11 and Figure 12 show how the algorithms performed with the number of labels in the datasets increased. The parameters for the tree size and the fanout conformed to $N\{50, 2.0\}$ and $N\{4, 0.5\}$ respectively. We chose 8, 16, 32, 64 as the size of the label universals for the four datasets. As shown in the figures, the binary branch filtration algorithm always outperformed the histogram algorithm. When there were eight labels in the dataset, the performance of histogram filtration was less effective than binary branch filtration by more than a factor of 20. In the two figures , we can see that with the increase of the number of labels from 8 to 32, the histogram filtration improved much. The reason is that the label histogram can perform better with a large label size. However, since the histogram vector size was set to be comparable to the binary branch vector represen-

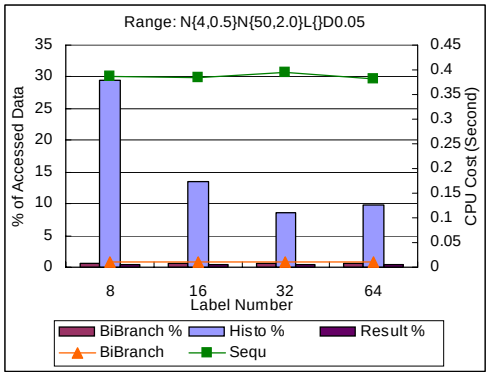


Figure 11: Sensitivity to Number of Labels in Trees for Range Queries

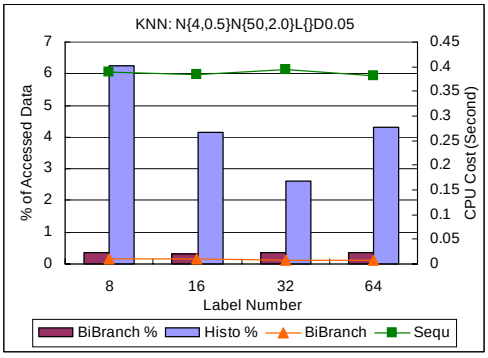


Figure 12: Sensitivity to Number of Labels in Trees for k-NN Queries

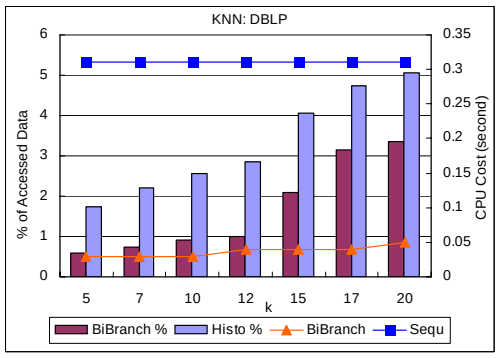


Figure 13: k-NN Searches on DBLP

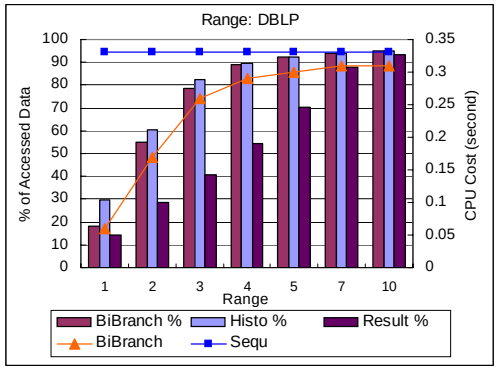


Figure 14: Range Searches on DBLP

tation, and since the mean values of the distance increased with the label size becoming larger, the performance began to degrade when the number of labels was larger than 32 for both the range queries and k-NN queries.

5.2 Similarity Query Performance

The experiments described in this part were conducted to compare the performance of the two filtration algorithms for the queries with different parameters. Figure 13 and figure 14 show the performance of the two algorithms for k-NN queries and range queries on the DBLP data. We randomly chose 2000 data objects from the whole DBLP dataset. 100 queries were randomly chosen from this set. The average tree size of the the data was 10.15; And the average distance among the data was 5.031;

Figure 13 displays the k-NN query results on DBLP data with the k varied from 5 to 20. We have also plotted the CPU time for sequential search in it. It can be seen that the binary branch filtration accessed much less data than the histogram filtration. It performed one to three times better than the histogram filtration. Since the DBLP data clustered very well, the percentage of the accessed data was small and the search time of binary branch filtration was only 1/6 of the sequential search time.

Figure 14 shows the results of range queries on DBLP. When the range remained less than the average distance among the data, the binary branch method clearly had better filtration power than the histogram method. As the range continued to increase to 10, the performance difference of the two methods decreased. The reason is that the result set was almost the whole dataset. Compared to the results of the percentage of data accessed in the previous experiments, the binary branch filtration here showed a smaller advantage over histogram filtration. This is due to the fact that the DBLP data consists of shallow and small tree data, and the relatively small size of the binary branch universal set blurs the distinctions among data.

5.3 Pruning Power With Respect To Binary Branch Levels

Fig. 15 shows the distribution of data according to distances between the data and the queries on DBLP. The results here were averaged on the query number. We plotted the data distribution on three kinds of distance: edit distance, binary branch distance ($BiBranch(2)$ in Fig. 15) and histogram distance between each data and query. We also plotted data distribution according to three and four-level binary branch distances ($BiBranch(3)$ and in $BiBranch(4)$ Fig. 15). We can see that two-level binary branch distance is a better lower bound of edit distance than the histogram distance. Thus it can filter out much more data than histogram filtration when processing similarity search. When the distance is less than 3, three and four-level binary branch distance are also better than histogram distance. When the range is larger than 3, the data distribution is almost the same for three and four-level binary branch distance and histogram filtration distance. According to the definition of the multiple level binary branch, we know that for the shallow tree-structured data like DBLP records, multiple level binary branch distance is not an efficient lower bound for edit distance.

From the above analysis, we can see that the binary branch filtration is robust since it outperforms histogram filtration

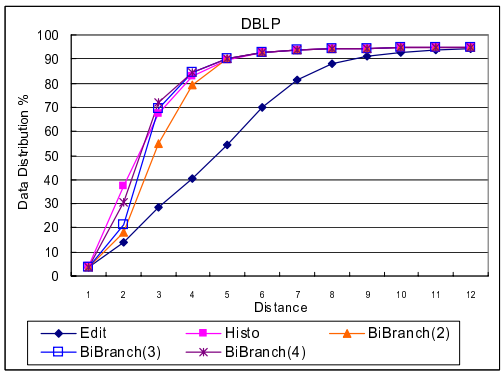


Figure 15: Data Distribution on Distance

on processing various types of datasets and on various settings of the queries. It is particularly suitable for processing real datasets in spite of their skewed nature. This may be because we encode structure information as well as the label information into the binary branch vector representations and positional sequences. In contrast, histogram filtration blurs the distinctions between trees since it uses only the histogram information, and the height, fanout and label histogram are considered separately.

6. CONCLUSION

Tree-structured data is becoming ubiquitous as it can express the hierarchical dependencies among data components and can be used to model data in many applications. Just as for other types of data, searches based on similarity measure are in the core of many operations for tree-structured data. However, the computational complexity of the general dissimilarity measure (i.e., the tree-edit distance) render the brute force methods prohibitive for processing large trees in huge datasets.

In this paper, we propose an efficient method based on the binary tree representation, where we transform trees into binary branch numerical vectors. This *characteristic vector* records the structural information of the original tree, and the L_1 norm distance on the vector space is proved to be the lower bound of the tree-edit distance. Moreover, we have generalized the vector representation of trees by using multiple level binary branches; this enables the structural information to be encoded in different granularities. Since our novel lower bound is much easier to obtain than the original distance measure, we propose embedding the new distance in the filter-and-refine architecture to reduce the computation of real edit distance between data and queries and guarantee no false negatives. We design a novel filter-and-refine similarity search algorithm which exploits the positional binary branch properties to obtain a better lower bound of edit distance. The results of our experiments show that our algorithm is robust to varying dataset features and query parameters. The pruning power of our algorithm leads to CPU and I/O efficient solutions.

7. REFERENCES

[1] C. C. Aggarwal, J. L Wolf, and P. S. Yu. A new method for similarity indexing of market basket data. In *SIGMOD*, pages 407–418, 1999.

[2] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marrquín. Proximity searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.

[3] Edgar Chavez and Gonzalo Navarro. Towards measuring the searching complexity of metric sapces. In *Proc. of the Mexican Computing Meeting*, pages 969–978, 2001.

[4] Minos Garofalakis and Amit Kumar. Correlating XML data streams using tree-edit distance embeddings. In *PODS*, pages 143–154, June 2003.

[5] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 327–340, 2001.

[6] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[7] K. Kailing, H. P. Kriegel, S. Schönauer, and T. Seidl. Efficient similarity search for hierarchical data in large databases. In *EDBT*, pages 676–693, March. 2004.

[8] Juha Kärkkäinen. Computing the threshold for q-gram filters. In *SWAT*, pages 348–357, 2002.

[9] Donald E. Knuth. *The Art of Computer Programming*. Addison-Wesley Pub Co, 1997.

[10] Nikos Mamoulis, David W. Cheung, and Wang Lian. Similarity search in sets and categorical data using the signature tree. In *ICDE*, pages 75–86, 2003.

[11] Alexandros Nanopoulos and Yannis Manolopoulos. Efficient similarity search for market basket data. *The VLDB Journal*, 11(2):138–152, 2002.

[12] Y. Sakurai, M. Yoshikawa, S. Uemura, and H. Kojima. The A-tree: An index structure for high-dimensional spaces using relative approximation. In *VLDB*, pages 516–526, 2000.

[13] Thomas Seidl and Hans-Peter Kriegel. Optimal multi-step k-nearest neighbor search. In *SIGMOD*, pages 154–165.

[14] Stanley M. Selkow. The tree-to-tree editing problem. *Information Processing Letters*, 6:184–186, December 1977.

[15] S.Guha, H. V. Jagadish, N. Koudas, D. Srivastava, and T. Yu. Approximate XML joins. In *SIGMOD*, pages 287–298, June 2002.

[16] Dennis Shasha and Kaizhong Zhang. *Approximate Tree Pattern Matching*. Oxford University, 1997.

[17] Erkki Sutinen and Jorma Tarhio. On using q-gram locations in approximate string matching. In *Proc. of 3rd Annual European Symposium*, pages 327–340, 1995.

[18] Jiang Tao, Lusheng Wang, and Kaizhong Zhang. Alignment of trees - an alternative to tree edit. In *Theoretical Computer Science*, volume 143, pages 75–86, 1995.

[19] Esko Ukkonen. Approximate string-matching with q-grams and maximal matches. *Theoretical Computer Science*, 92:191–211, 1992.

[20] R. Weber, H.-J. Schek, and S. Blott. A quantitative ananlysis and performance study for similarity search methods in high-dimensional space. In *VLDB*, pages 194–205, 1998.

[21] Mohammed J. Zaki. Efficiently mining frequent trees in a forest. In *SIGKDD*, pages 71–80, 2002.

[22] Kaizhong Zhang. Algorithms for the constrained editing distance between ordered labelled trees and related problems. In *Pattern Regonition*, volume 28, pages 463–474, 1995.

[23] Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18:1245–1262, December 1989.