A Tree Distance Function Based on Multi-sets

Arnoldo José Müller-Molina, Kouichi Hirata, and Takeshi Shinohara

Department of Artificial Intelligence, Kyushu Institute of Technology
Kawazu 680-4 Iizuka 820-8502, Japan
arnoldo@daisy.ai.kyutech.ac.jp
{hirata,shino}@ai.kyutech.ac.jp

Abstract. We introduce a tree distance function based on multi-sets. We show that this function is a metric on tree spaces, and we design an algorithm to compute the distance between trees of size at most n in $O(n^2)$ time and $O(n)$ space. Contrary to other tree distance functions that require expensive memory allocations to maintain dynamic programming tables of forests, our function can be implemented over simple and static structures. Additionally, we present a case study in which we compare our function with other two distance functions.

Key words: Tree edit distance, Program matching, Triangle inequality, Metric

1 Introduction

Analysis of tree structured data is required in many fields [6, 9, 18]. Our research group is particularly interested in the field of approximate binary program matching for the detection of Open Source/Libre software license violations [11, 12]. A recent technique in this field works by generating fragments, that are regarded as trees, from control flow graphs. By means of a tree distance function, the similarity of two programs can be computed. We are interested in distance functions that satisfy the triangle inequality. If this property is satisfied, then we can employ dimension reduction techniques like SMAP [14], and therefore matching speed on massive databases improves considerably. Additionally, it is desirable that the distance function is a metric. Specifically, it is necessary for the distance function to satisfy the property: $d(x, y) = 0$ if and only if $x = y$. If this property is satisfied, the matching of programs is “safer” and false positives are reduced.

An overview of current literature [2] presents a tree edit distance function and several variants. To compute every distance function that is metric in [2], it is necessary at least $O(n^2)$ time where n is the maximum number of nodes in two given trees. In this paper, we propose a new tree distance function based on multi-sets that is simple to compute, and that does not require dynamic programming or intensive memory allocations. The main contribution of this paper is to introduce a metric for rooted ordered labeled trees that can be computed in $O(n^2)$ time and $O(n)$ space. We denote this metric as *mtd*. The rest of the paper is organized as follows. Section 2 introduces related works. In Section 3, the proposed distance *mtd* is described and in Section 4, a naive algorithm to compute *mtd* is presented. Finally in Section 5, we present a case study.

2 Related Works

The generally accepted similarity measure for trees is the tree-edit distance metric (*ted*) introduced by Tai[15] in 1979. Klein[10] proposed an $O(n^3 \log(n))$ algorithm to compute *ted*. This problem is known to be NP-hard for unordered trees[19]. Currently, the fastest algorithm for ordered trees is $O(n^3)$[5].

Chawathe *et al.*[3] introduced an approach in which the set of edit operations is extended. This distance is NP-complete, and the fastest heuristic runs in $O(n^3)$ time. A different variation [4] is linear, however it is not clear if these similarity functions are metrics nor if they satisfy the triangle inequality.

The tree alignment distance introduced by Jiang *et al.*[7] is not a metric. Finally, the fastest metric that we are aware of is the constrained edit distance[17], with time $O(n^2)$ and space $O(n \log(n))$. The distance functions described above can be seen as an edit script minimization problem. On the other hand, functions based on q-grams [1, 13, 16] are a vector feature distance minimization. These functions have linear complexity, but they are not metrics.

2.1 Approximate Program Matching

The function presented in this paper, is intended to be used in the context of approximate binary program matching field. Müller-Molina and Shinohara introduced a technique based on extracting *program fragments*[11] from binary programs. A program fragment is a tree whose nodes are machine-level instruction codes. They are extracted from program control flow graphs. By analyzing the frequency distribution of the fragments, it is possible to employ "standard" information retrieval techniques to rank programs by similarity. The technique also uses tree distance functions to find similar pairs of program fragments. In this way, even if fragment normalization rules fail, the distance function can compensate for differences introduced by a program transformation. This process is analogous to the stemming process web information retrieval engines apply to each word of a natural language query. The distance introduced in this paper has been employed successfully in this context, but its properties were not studied formally.

3 Tree Similarity Distance

A *tree* is a connected graph without cycles. For a tree $T = (V, E)$, we sometimes denote $v \in T$ instead of $v \in V$, and $|T|$ instead of $|V|$. A *rooted tree* is a tree with one node r chosen as its *root*.

Let r be a root of T and v a node in T. Then, the *parent* of $v(\neq r)$ is its adjacent node in a path from v to r in T, and the *ancestors* of $v(\neq r)$ are the nodes contained in a path from the parent of v to r in T. The parent and the ancestors of the root r are undefined. We say that u is a *child* of v if v is the parent of u, and u is a *descendant* of v if v is an ancestor of u. A *leaf* is a node having no children. Furthermore, a *complete subtree* of T at v is a subtree of T of which its root is v and that contains all descendants of v.

A rooted tree $T = (V, E)$ is *labeled* (by an alphabet Σ of labels) if there exists an onto function $l : V \rightarrow \Sigma$ such that $l(v) = a$ ($v \in V, a \in \Sigma$). A tree is *ordered* if a left-to-right order for the children of each node is given; *unordered* otherwise. In this paper, we call a rooted labeled ordered tree simply by a tree.

Next, we introduce the notions of *multi-sets*. Intuitively, a *multi-set* is a set that allows an element to occur more than once. The *multiplicity* $m_A(x)$ of an element x in a multi-set A is the number of the occurrences of x in A. For a (standard) set A, it holds that $m_A(x) = 1$ for every $x \in A$. It is clear that $x \notin A$ if $m_A(x) = 0$. The *set view* $v(A)$ of a multi-set A is a set $\{x \in A \mid m_A(x) \geq 1\}$. For example, for a multi-set $A = \{a, a, a, b, c, c\}$, it holds that $v(A) = \{a, b, c\}$. Additionally, the *cardinality* $|A|$ of a multi-set A is defined as $\sum_{x \in v(A)} m_A(x)$.

We now introduce the multi-set operations. Let A and B be multi-sets. Then, the *intersection* $A \sqcap B$, the *union* $A \sqcup B$ and the *additive union* $A \uplus B$ of A and B are defined as follows.

$$\begin{aligned} A \sqcap B &= \{x \in v(A) \cap v(B) \mid m_{A \sqcap B}(x) = \min\{m_A(x), m_B(x)\}\}, \\ A \sqcup B &= \{x \in v(A) \cup v(B) \mid m_{A \sqcap B}(x) = \max\{m_A(x), m_B(x)\}\}, \\ A \uplus B &= \{x \in v(A) \cup v(B) \mid m_{A \uplus B}(x) = m_A(x) + m_B(x)\}. \end{aligned}$$

For example, let $A = \{a, a, b, c, c\}$ and $B = \{a, b, b, c\}$. Then, it holds that $A \sqcap B = \{a, b, c\}$, $A \sqcup B = \{a, a, b, b, c, c\}$ and $A \uplus B = \{a, a, a, b, b, b, c, c, c\}$.

Based on the previous operations, we define a similarity measure for multi-sets:

$$\delta(A, B) = |A \sqcup B| - |A \sqcap B|. \tag{1}$$

Finally, we introduce the function $s(T)$ which is the multi-set of all complete subtrees of T. For example, in Figure 1 the tree $C(B)$ is not a complete subtree of T_1, but $C(B, E)$ is a complete subtree of T_1. The function $n(T)$, is the multi-set of all the nodes of T.

3.1 Distance definition

The first step to calculate our similarity function is to convert a tree into two multi-sets, one multi-set of complete subtrees and another multi-set that contains all the nodes (without children) of the tree. This can be achieved by invoking functions s and n. Once the multi-sets of the trees have been generated, the function *mtd* can be computed. The function *mtd* is defined as:

$$d_s(T_1, T_2) = \delta(s(T_1), s(T_2)), \tag{2}$$

$$d_n(T_1, T_2) = \delta(n(T_1), n(T_2)), \tag{3}$$

$$mtd(T_1, T_2) = \frac{d_s(T_1, T_2) + d_n(T_1, T_2)}{2}. \tag{4}$$

The *mtd* function is composed of two different distance operations: d_s and d_n. The first function d_s is a measure based on the number of equal complete subtrees between T_1 and T_2. Note that a change of only one leaf node can have multiple repercussions in all its parents. In general, this measure is very sensitive to changes and quickly the intersection $s(T_1) \sqcap s(T_2)$ becomes void. Nevertheless, an important reason to have this intersection is that it makes sure that $mtd(T_1, T_2) = 0$ iff $T_1 = T_2$. Additionally, trees that share many common complete subtrees will receive a high score.

The second function d_n is a measure that is likely to find matches since only individual nodes are considered. Note that d_n is Kailing's distance based on the label histogram [8]. Using only this measure would make trees very similar to each other, however it is necessary to balance the strictness of d_s.

To summarize, the intuitive idea of our matching procedure is that a very strict matching (d_s) combined with a permissive matching (d_n) brings a balance to the scoring technique. In the context of binary program matching, we have found that both *mtd* and d_s return acceptable results. Function *mtd* produces slightly better results than d_s.

Note that *mtd* always returns a natural number. Precisely, we have the following proposition:

Proposition 1. *$d_s(T_1, T_2) + d_n(T_1, T_2)$ is even for any trees T_1 and T_2.*

Proof. For multi-sets A and B, since $|A \sqcup B| + |A \sqcap B| = |A \uplus B|$, it holds that $|A \sqcup B| - |A \sqcap B| = |A \uplus B| - 2|A \sqcap B|$. Then, for multi-sets $s(T_i)$, and $n(T_i)$ $(i = 1, 2)$, the following equation holds:

$$|s(T_1) \sqcup s(T_2)| - |s(T_1) \sqcap s(T_2)| + |n(T_1) \sqcup n(T_2)| - |n(T_1) \sqcap n(T_2)| \tag{5}$$

$$= |s(T_1) \uplus s(T_2)| - 2|s(T_1) \sqcap s(T_2)| + |n(T_1) \uplus n(T_2)| - 2|n(T_1) \sqcap n(T_2)| \tag{6}$$

Since $|s(T_i)| = |n(T_i)|$, let $|n(T_i)| = |s(T_i)| = k_i$. Hence, we can obtain the following even expression:

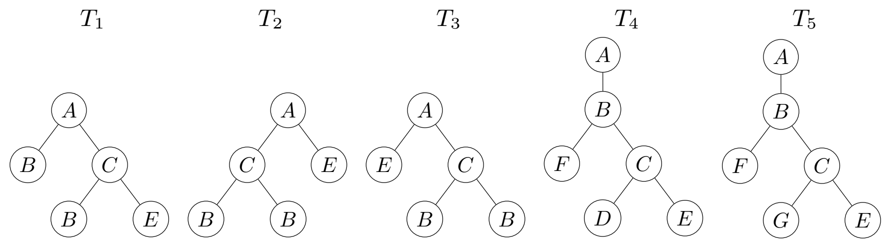
$$2k_1 + 2k_2 - 2|s(T_1) \sqcap s(T_2)| - 2|n(T_1) \sqcap n(T_2)|. \tag{7}$$

□

In what follows, we give some examples of *mtd*.

In Figure 1 three examples are displayed. In the first example, $mtd(T_1, T_2)$, returns 2 because it does not consider in which place a complete subtree is found (multi-sets do not record the position of the subtrees). The function $ted(T_1, T_3)$ returned 2 because "C" is deleted, and a new "C" node is inserted over the two "B" nodes.

In the second example, $mtd(T_1, T_3)$ is calculating the distance of two node renaming operations. The result is again the same for $ted(T_1, T_3)$. In the third example, $mtd(T_4, T_5)$ returns 5 because all of the complete subtrees of T_4 except "E" and "F" are not in T_5. The deeper the modification is, the greater the distance will be. In the context of binary program matching[12] this behavior is desirable because changes in the deepest part of an expression are likely to change more drastically the "meaning" of the expression than changes in upper layers of the expression.



$s(T_1) = \{A(B, C(B, E)), B, C(B, E), B, E\}$
$s(T_2) = \{A(C(B, B), E), C(B, B), B, B, E\}$
$s(T_3) = \{A(E, C(B, B)), E, C(B, B), B, B\}$
$s(T_4) = \{A(B(F, C(D, E))), B(F, C(D, E)), F, C(D, E), D, E\}$
$s(T_5) = \{A(B(F, C(G, E))), B(F, C(G, E)), F, C(G, E), G, E\}$

$n(T_1) = \{A, B, C, B, E\}$
$n(T_2) = \{A, C, B, B, E\}$
$n(T_3) = \{A, E, C, B, B\}$
$n(T_4) = \{A, B, F, C, D, E\}$
$n(T_5) = \{A, B, F, C, G, E\}$

$|s(T_1) \sqcap s(T_2)| = 3$ $\quad$ $|s(T_1) \sqcap s(T_3)| = 3$ $\quad$ $|s(T_4) \sqcap s(T_5)| = 2$
$|n(T_1) \sqcap n(T_2)| = 5$ $\quad$ $|n(T_1) \sqcap n(T_3)| = 5$ $\quad$ $|n(T_4) \sqcap n(T_5)| = 5$

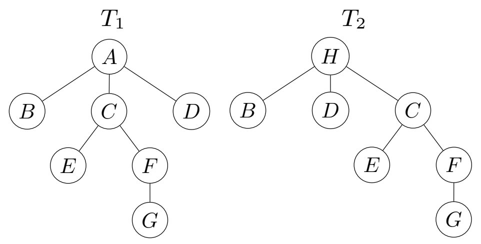
$$mtd(T_1, T_2) = \frac{(7-3)+(5-5)}{2} = 2 \qquad ted(T_1, T_2) = 2$$
$$mtd(T_1, T_3) = \frac{(7-3)+(5-5)}{2} = 2 \qquad ted(T_1, T_3) = 2$$
$$mtd(T_4, T_5) = \frac{(10-2)+(7-5)}{2} = 5 \qquad ted(T_4, T_5) = 1$$

Fig. 1. This example calculates $mtd(T_1, T_2)$, $mtd(T_1, T_3)$, and $mtd(T_4, T_5)$. T_1 and T_2 illustrate the cost of complete subtree movement. T_1 and T_3, illustrate two label modification operations. T_3 and T_4 depict the cost of one node modified at deep levels of the tree.

In Figure 2, an example in which *mtd* returns a smaller value than *ted* is shown. In this case, $mtd(T_1, T_2)$ returns a smaller distance than $ted(T_1, T_2)$ because swapped nodes (rooted in "C" and "D") were children of a node whose label was modified. Because the algorithm does not differentiate when the root node *and* the children are modified, the score is lower. The function $ted(T_1, T_2)$ returns 3 because "A" is renamed by "H", "D" is deleted, and finally, "D" is inserted to the left of "C".

Proposition 2. *For any trees T_1, T_2 and T_3, the following statements hold:*

1. $mtd(T_1, T_2) \geq 0$	*non-negativity*
2. $mtd(T_1, T_2) = mtd(T_1, T_2)$	*symmetry*
3. $mtd(T_1, T_2) = 0 \iff T_1 = T_2$	*identity of indiscernibles*
4. $mtd(T_1, T_3) \leq mtd(T_1, T_2) + mtd(T_2, T_3)$	*triangle inequality*



$s(T_1) = \{A(B, C(E, F(G)), D), B, C(E, F(G)), E, F(G), G, D\}$ $\quad$ $n(T_1) = \{A, B, C, E, F, G, D\}$
$s(T_2) = \{H(B, D, C(E, F(G))), B, D, C(E, F(G)), E, F(G), G\}$ $\quad$ $n(T_2) = \{H, B, D, C, E, F, G\}$

$|s(T_1) \sqcap s(T_2)| = 6$ $\quad$ $|n(T_1) \sqcap n(T_2)| = 6$
$mtd(T_1, T_2) = \frac{(8-6)+(8-6)}{2} = 2$ $\quad$ $ted(T_1, T_2) = 3$

Fig. 2. In this example, *mtd* returns a smaller distance than *ted*.

Proof. Properties 1 and 2 are obvious. Property 3 can be deduced from the fact that T_1 and T_2 are returned by the d_s function. If there is any change between the trees, the original tree will not be matched in the intersection operation and a distance greater than zero will be computed.

We now prove property 4. It is sufficient to prove that δ satisfies the triangle inequality, $\delta(A, C) \leq \delta(A, B) + \delta(B, C)$.

The function $\delta(A, B)$ can be rewritten as:

$$\delta(A, B) = \sum_{x \in v(A) \cup v(B)} |m_A(x) - m_B(x)|. \tag{8}$$

Since:

$$|m_A(x) - m_C(x)| \leq |m_A(x) - m_B(x)| + |m_B(x) - m_C(x)|, \tag{9}$$

the triangle inequality holds. Therefore *mtd* is a metric on tree spaces.

□

4 Naive Algorithm for computing *mtd*

In what follows, we describe a naive algorithm for computing δ and *mtd*. Once δ is obtained, *mtd* can be computed easily. In lines 3 to 7 of Algorithm 1, the tree v is compared against all the trees of size $|v|$ of the multi-set A. This optimization is necessary to keep the complexity quadratic as we shall see later. The function δ (lines 12 to 14) calculates the intersection between the multi-sets A and B by obtaining the minimum of the multiplicity of the common elements. Finally in line 15, the cardinality of the union of $|T_1|$ and $|T_2|$ is subtracted from the cardinality of the intersection of T_1 and T_2 and the result is returned to the caller.

We now proceed to analyze the complexity of algorithm *mtd*. The following lemma is useful:

Algorithm 1 δ and m functions implementation

```
    Receives a multi-set an a tree
1:  function $m(A, v)$                                           ▷ Returns $m_A(v)$
2:      $c \leftarrow 0$                                         ▷ Multiplicity counter
3:      for $m \in A$ such that $|v| = |m|$ do        ▷ Only compare trees of size $|v|$.
4:          if $m = v$ then
5:              $c \leftarrow c + 1$
6:          end if
7:      end for
8:      return c
9:  end function
    Receives two multi-sets
    $v(x)$ receives a multi-set $x$ and returns a set "view" of $x$.
10: function $\delta(A, B)$                                      ▷ Returns $\delta(A, B)$
11:     $c \leftarrow 0$                                         ▷ Intersection counter
12:     for $v \in v(A)$ do
13:         $c \leftarrow c + min(m_A(v), m_B(v))$
14:     end for
15:     return $|A \sqcup B| - c$
16: end function
```

Lemma 1. *Every tree of size m has at most $\frac{m}{n}$ complete subtrees of size n.*

Theorem 1. *The distance function $mtd(F, G)$ can be computed in $O(|T_1| \times |T_2|)$ time and $O(|T_1| + |T_2|)$ space.*

Proof. It is necessary to consider the cost involved in comparing the equality of two trees. For two trees T_1 and T_2, it is obvious that checking whether or not $T_1 = T_2$, can be computed in $O(|T_1|)$ time. The computation of s and n can be achieved in linear time. Additionally, the space required by the multi-sets returned by s and n is linear because it is sufficient to store pointers to the original trees. Regarding m, we can see in line 3 of Algorithm 1 that for a complete subtree $v \in A$, the function will only compare complete subtrees of size $|v|$ in B.

Since d_n matches only nodes (trees of size 1), it can be computed in linear time, therefore we will focus only on d_s. Each equality comparison for any complete subtree $v \in F$ will take at most $|v|$ steps, and will be executed at most $\frac{|T_2|}{|v|}$ times. This is because we will only match complete subtrees in F of size $|v|$ (Algorithm 1, line 3) and because Lemma 1 guarantees that $|T_2|$ has only $\frac{|T_2|}{|v|}$ complete subtrees of size $|v|$. Therefore, the multiplicity $m_{s(T_2)}(v)$ can be computed in $O(|v| \times \frac{|T_2|}{|v|}) = O(|T_2|)$ time. Since the multiplicity function m will be called by δ at most $|T_1|$ times, the function $mtd(T_1, T_2)$ can be computed in $O(|T_1| \times |T_2|)$ time. Since it is possible to create a multi-set of pointers to the original nodes of the tree, the space complexity for $mtd(T_1, T_2)$ is $O(|T_1| + |T_2|)$. □

Our current implementation uses hash tables to improve performance. Hash codes have greater pruning power than the pruning by tree size described above, however this is not enough to lower the complexity of the algorithm. In the experiments of section 5, we will see that in practice, our hash-based *mtd* implementation matches the performance of a linear q-gram based distance function.

5 Case Study

In this section, we compare our function against *BDist* [16] and *ted* [15]. Currently, we are not able to run *ted* on our approximate program matching framework[1] because of its enormous computational cost. Therefore, our experiments were executed on real tree data extracted from program fragments[12] but focus on the distance functions only. The procedure to extract these fragments is detailed in [12]. Our data-set[2] includes 244668 trees. The average depth is 4.75, the average number of nodes is 11.11, and the number of nodes range between 1 to 20 nodes. In the experiment, we randomly selected 1000 trees (queries) from the data-set and compared them against the rest of the data-set. About 244 million tree comparisons were performed.

We implemented the *ted* [5] that can be computed in $O(n^3)$ time, *BDist*[16] that can be computed in $O(n)$ time, and *mtd* that has a time complexity of $O(n^2)$. The distance functions were implemented in Java. The source code of the functions is available under the GPL license[3]. For the performance benchmarks, we loaded all the trees into memory and the time is counted after all the trees have been loaded. The experiments were executed on a Intel(R) Xeon(R) CPU 2.66 GHz with 4 processors. Four threads were created to reduce execution time. Each thread performs the same job.

Regarding the *q*-gram function, we used Yang's *BDist* function[16]. In this case, *BDist* will create $|T|$ grams from a tree T by creating for each node $v \in T$, a triple of the left child of v, v, and the right sibling of v. The original motivation of the paper was to create a similarity search engine by creating an inverted file of each q-gram. In this paper we simply calculated *BDist* in a sequential manner to show the real cost of a linear distance function. By using the similarity index presented by Yang, better performance results can be achieved for *BDist*. Finally, it can be proved that $BDist(T_1, T_2) \leq 5 \times ted(T_1, T_2)$[16].

Distance Evaluation

In Figure 3, we show the distribution of data according to distances between data and queries. In the y axis, the percentage of calculations that returned the distance shown in the x axis is displayed. We can see that all the functions maintain a similar, normal distribution.

[1] `http://www.furiachan.org`

[2] `http://furia-chan.googlecode.com/files/trees1-20.tar.bz2`

[3] `http://furia-chan.googlecode.com/files/mtd-1.tar.bz2`

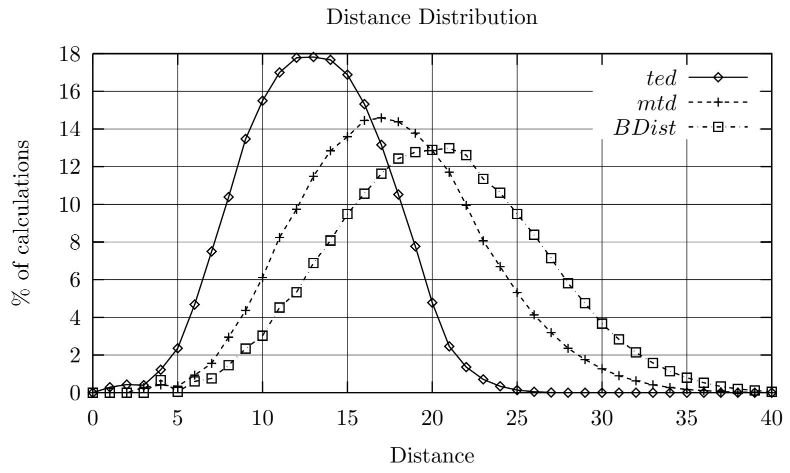


Fig. 3. Distribution of data according to distances between data and queries.

In Figures 4(a) and 4(b), we compare respectively, how *mtd* and *BDist* vary from the result returned by *ted*. For *mtd*, the mean and standard deviation are closer than *BDist*. Additionally, maximum values for *mtd* tend to be smaller than for *BDist*. On the other hand, *Bdist*'s minimum values seem to be closer to *ted*. When *BDist* is used in its similarity search framework (and when a range r is provided), additional pruning can be performed by the pre-order and post-order counts of the node. In this case, *BDist* cannot satisfy the triangle inequality and that is why we did not include this pruning. *BDist* is returning bigger results than *mtd* because subtree movements return lower values than for *BDist* as siblings are not taken into account. For example in Figure 2, the distance returned by *BDist* is 8 for T_1 and T_2. Since the shape of the distribution and the overall distance results are closer to *ted*, we can conclude that *mtd* performs better than *BDist*.

In Figures 5(a) and 5(b), we compare how d_s and d_n vary from the result returned by *ted*. We can see how d_s has bigger maximum values than *mtd* (Figure 4(a)). On the other hand, d_n has smaller minimum values than *mtd* or d_s. Finally, we can see that *mtd*'s minimum and maximum range is improved by the combination of d_s and d_n. The mean and standard deviation for *mtd* is centered between d_s and d_n. If we base our comparison on *ted*, the use of d_n is justified. It is interesting to see that the closest function to *ted* when considering only the standard deviation is d_n.

Benchmarks

For the same experiment described at the beginning of the section, we recorded the execution time for *ted*, *BDist* and *mtd*. The function *ted* took about 85 hours to complete, *BDist* took 8 minutes, and *mtd* took 7.4 minutes. The small difference between *BDist* and *mtd* might be related to the hash function being

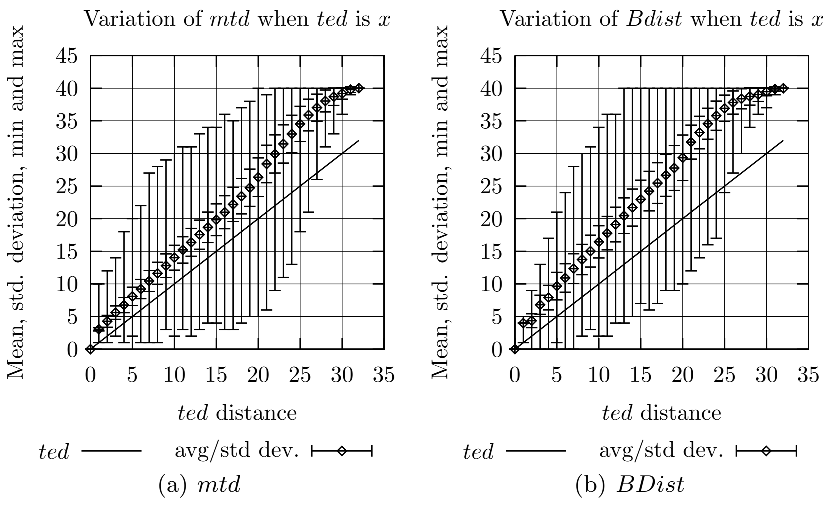


Fig. 4. Variation of mtd and $BDist$ with respect to the result x of ted

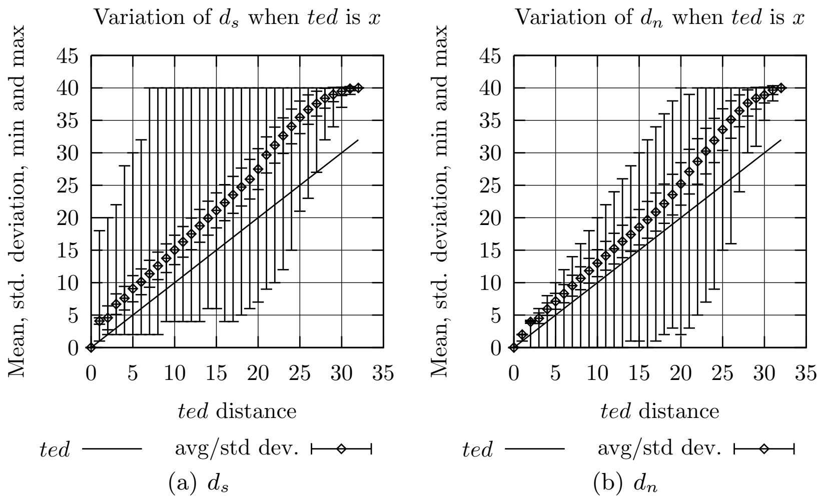


Fig. 5. Variation of d_s and d_n with respect to the result x of ted

called more times in *BDist* than in *mtd*. On the other hand, *mtd* was implemented recursively and there is much room for improvement. One of the reasons *ted* is performing so poorly is that for each distance computation, many objects are being created (dynamic programming table objects and forests created and destroyed during the search). On the contrary, *mtd* and *BDist* only have to access some static structures that are computed once during the lifetime of a tree. From the benchmarking results, can see how the actual performance of our function seems to be on par with a linear q-gram distance function.

6 Conclusions and Future Work

We have introduced a tree distance function that is based on multi-sets. When compared to *ted*, our function is more sensitive to changes. This is in general an undesirable property, however the performance gains are considerable enough to make it a worthwhile candidate for matching trees. Our function is fast because it can perform matchings without resorting to expensive dynamic programming table memory allocations. Our hash table based implementation achieved similar performance than *BDist*[16], a q-gram based function that runs in time $O(n)$. Our distance function also has the added property of being a metric.

Regarding future works, because *mtd* is faster than *ted*, it can be employed by techniques like SMAP [14] to find a subset of the data suitable for pivoting. From that subset, a set of pivots based on *ted* could be computed. Finally, we would like to study the effect of adding N-grams greater than 1 to function n.

References

[1] N. Augsten, M. Bhlen, and J. Gamper. Approximate matching of hierarchical data using pq-grams. In *VLDB '05*, pages 301–312, 2005.

[2] P. Bille. A survey on tree edit distance and related problems. *Theoretical Computer Science*, 337(1-3):217–239, 2005.

[3] S. S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. *SIGMOD Rec.*, 26(2):26–37, 1997.

[4] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. *SIGMOD Rec.*, 25(2):493–504, 1996.

[5] E. Demaine, S. Mosez, B. Rossman, and O. Weimann. An optimal decomposition algorithm for tree edit distance. In *ICALP '07, LNCS*, volume 4596, pages 146–157. Springer-Verlag, 2007.

[6] M. Garofalakis and A. Kumar. Xml stream processing using tree-edit distance embeddings. *ACM Trans. Database Syst.*, 30(1):279–332, 2005.

[7] T. Jiang, L. Wang, and K. Zhang. Alignment of trees - an alternative to tree edit. *Theoretical Computer Science*, 143(1):157–148, 1995.

[8] K. Kailing, H.-P. Kriegel, S. Schoenauer, and T. Seidl. Efficient similarity search for hierarchical data in large databases. In *EDBT '04 LNCS*, volume 2992, pages 676–693, 2004.

[9] P. Klein, S. Tirthapura, D. Sharvit, and B. Kimia. A tree-edit-distance algorithm for comparing simple, closed shapes. In *SODA '00*, pages 696–704, Philadelphia, USA, 2000. Society for Industrial and Applied Mathematics.

[10] P. N. Klein. Computing the edit-distance between unrooted ordered trees. In *ESA '98, LNCS*, volume 1461, pages 91–102. Springer-Verlag, 1998.

[11] A. J. Müller-Molina and T. Shinohara. On approximate matching of programs for protecting libre software. In *CASCON '06*, pages 275–289. ACM Press, 2006.

[12] A. J. Müller-Molina and T. Shinohara. Fast approximate matching of programs for protecting libre/open source software by using spatial indexes. In *SCAM '07*, pages 111–122. IEEE Computer Society, 2007.

[13] N. Ohkura, K. Hirata, T. Kuboyama, and M. Harao. The q-gram distance for ordered unlabeled trees. In *Discovery Science LNAI*, volume 3735, pages 189–202, 2004.

[14] T. Shinohara and H. Ishizaka. On dimension reduction mappings for approximate retrieval of multi-dimensional data. In *Progress in Discovery Science, LNCS*, volume 2281, pages 224–231. Springer-Verlag, 2002.

[15] K.-C. Tai. The tree-to-tree correction problem. *JACM*, 26(3):422–433, 1979.

[16] R. Yang, P. Kalnis, and A. K. H. Tung. Similarity evaluation on tree-structured data. In *SIGMOD '05*, pages 754–765, 2005.

[17] K. Zhang. Algorithms for the constrained editing distance between ordered labeled trees and related problems. *Pattern Recognition*, 28(3):463–474, 1995.

[18] K. Zhang. Computing similarity between rna secondary structures. *INTSYS '98*, pages 126–132, 1998.

[19] K. Zhang, R. Statman, and D. Shasha. On the editing distance between unordered labeled trees. *Information Processing Letters*, 42(3):133–139, 1992.