Computing the edit-distance between unrooted ordered trees

Philip N. Klein*

Department of Computer Science, Brown University

Abstract. An ordered tree is a tree in which each node's incident edges are cyclically ordered; think of the tree as being embedded in the plane. Let A and B be two ordered trees. The *edit distance* between A and B is the minimum cost of a sequence of operations (contract an edge, uncontract an edge, modify the label of an edge) needed to transform A into B. We give an $O(n^3 \log n)$ algorithm to compute the edit distance between two ordered trees.

1 Introduction

A tree is said to be *ordered* if each node is assigned a cyclic ordering of its incident edges. Such an assignment of cyclic orderings constituted a combinatorial planar embedding of the tree (and is called a *rotation system*; see [1]). Several application areas involve the comparison between planar embedded trees. Two examples are biochemistry (comparing the secondary structures of different RNA molecules) and computer vision (comparing trees that represent different shapes).

One way of comparing such trees is by their edit distance: the minimum cost to transform one tree into another by elementary operations. The edit distance between two trees can be computed using dynamic programming. This paper provides a faster dynamic-programming algorithm than was previously known.

Let A and B be ordered trees. We assume in this paper that the edges are labeled; node labels can be handled similarly. Two kinds of elementary operations are allowed: label modification and edge contraction. We assume that two subroutines (or tables) have been provided. The first subroutine, given two labels, outputs the cost of changing one label into the other. The second, given a label, outputs the cost of contracting an edge with that label. We assume that the costs are all nonnegative. The algorithmic goal is to find a minimum-cost set of operations to perform on A and B to turn them into the same tree. The left of Figure 1 gives an example.

In stating the time bounds for algorithms, we assume that the cost subroutines take constant time.

A more familiar edit-distance problem is computing *string edit-distance* [7]. The edit-distance between two strings is the minimum cost of a set of symbol-deletions and symbol-modifications required to turn them into the same string.

* research supported by NSF Grant CCR-9700146

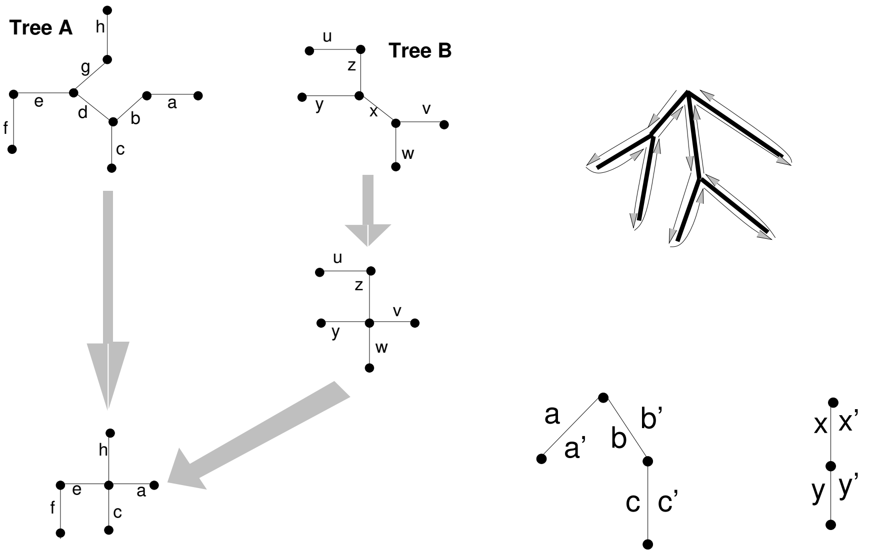


Fig. 1. The diagram on the left shows the comparison between two planar-embedded trees A and B. They can be transformed into the same tree as follows. In A, contract the edges f, d, b. In B, contract the edge x, and then change labels u, v, w, y, z to f, h, a, c, e.

On the top-right is shown a rooted tree (in bold) and the corresponding Euler string of darts (indicated by arrows).

On the bottom-right, two small trees are shown with labeled darts. The Euler tour of the left tree is $aa'bcc'b'$ and that of the second is $xyy'x'$.

Edit-distance between trees that are simple paths is equivalent to the string edit-distance problem. There is a simple $O(ab)$-time algorithm to compute the distance between a length-a string and a length-b string. Thus the worst-case time bound is $O(n^2)$.

A modification of the string edit-distance problem is the *cyclic* string edit-distance problem. In this problem, the strings are considered to be cyclic, and an algorithm must determine the best alignment of the beginnings of the two strings. One can arbitrarily fix the beginning of the first string, and try all possibilities for the beginning of the second string: for each, one computes an ordinary edit-distance. This brute-force approach reduces a cyclic edit-distance instance to n ordinary edit-distance instances, and hence yields an $O(n^3)$-time algorithm. An algorithm due to Maes [2] takes $O(n^2 \log n)$ time.

The problem of *rooted* ordered tree edit-distance has previously been considered. This problem arises in settings where, e.g., parse trees need to be compared, such as in natural language processing and image understanding.

For this problem, the input consists of two rooted trees A and B where each node's children are ordered left to right. The fastest previously known algorithm is due to Zhang and Shasha; it runs in time

$$O(|A|\ |B|\ \mathrm{LR\_colldepth}(A)\ \mathrm{LR\_colldepth}(B))$$

where $|T|$ denotes the size of a tree T and $\mathrm{LR\_colldepth}(T)$ is a quantity they define, called the *collapsed depth*. Zhang and Shasha bound the collapsed depth of a tree T by

$$\min\{\text{depth of } T,\ \text{number of leaves of } T\}$$

However, in the worst case, the collapsed depth of a tree T is $\Omega(|T|)$. Thus in the worst case their algorithm runs in time $O(n^4)$ where n is the sum of the sizes of the two trees.

Unrooted ordered trees are to rooted ordered trees as cyclic strings are to ordinary strings, and it is possible to use a similar brute-force reduction from edit-distance on unrooted trees to edit-distance on rooted trees. The brute-force reduction would yield an algorithm that in the worst case required $O(n^5)$ time.

We give an algorithm that runs in $O(n^3 \log n)$ time. It solves both the rooted and the unrooted tree edit-distance problems. Thus it improves the worst-case time on rooted trees by nearly a factor of n (although depending on the input trees Zhang and Shasha's algorithm may be faster). It beats the naive $O(n^5)$-time algorithm for the unrooted case by nearly an n^2 factor.

In particular, for trees A and B, our algorithm runs in time $O(|A|^2|B|\log|B|)$. Our algorithm uses essentially the same approach as the algorithm of Zhang and Shasha. We define a variant of collapsed depth that is always at most logarithmic, and we generalize their algorithm to work with this variant. Loosely speaking, the complexity of analyzing B is thus reduced from $|B|\ \mathrm{LR\_colldepth}(B)$ to $|B|\log|B|$. The price we pay, however, is that the complexity of analyzing A goes from $|A|\mathrm{LR\_colldepth}(A)$ to $|A|^2$. The consolation is that within this bound we can consider all possible roots of A; thus we can solve the unrooted problem within the same bounds.

1.1 Notation

For a rooted tree T, the root of T is denoted $\mathrm{root}(T)$. For any node v in T, the subtree of T consisting of v and its descendents is called a *rooted subtree* of T, and is denoted $T(v)$. A special case arises in which the tree is a descending path P (the first node of P is taken as the root of the tree): in this case, $P(v)$ denotes the subpath beginning at v.

For an edge e of T, we let $T(e)$ denote the subtree of T rooted at whichever endpoint of e is farther from the root of T. Note that e does not occur in $T(e)$.

Given an ordered, rooted tree T, replace each edge $\{x, y\}$ of T by two oppositely directed arcs (x, y) and (y, x), called *darts*. The depth-first search traversal of T (visiting each node's children according to their order) defines an Euler tour of the darts of T. Each dart appears exactly once. (See the top-right of Figure 1.) We interpret the tour as a string, the *Euler string* of T, and we denote this string by $E(T)$. The first dart of the string goes from the root to the leftmost child of the root.

For a dart a, the oppositely directed arc corresponding to the same edge will be denoted a^M (here M stands for "mate") and will be called the *mate* of a.

A *substring* of a string is defined to be a consecutive subsequence of the string.

The *reverse* of a string s is denoted s^R. Thus s^R contains the same elements as s but in the reverse order. If the dart a is the first or last symbol of s, we use $s - a$ to denote the substring obtained from s by deleting a.

We use Λ to denote the empty string, and we use $\log x$ to denote $\log_2 x$.

2 Euler strings: parenthesized strings for representing trees

For now, we take as our goal calculating the edit-distance between two *rooted* trees. For this purpose, it is notationally and conceptually useful to represent the trees by their Euler strings. We can thus interpret the edit-distance problem on trees as an edit-distance problem on strings. However, this string edit-distance problem is not an ordinary one; each dart occurring in a tree's Euler string has a mate, and the pairing of darts affects the edit-distance calculation.

Think of each pair of darts as a pair of parentheses. The Euler string of a tree is then a nesting of parentheses. In comparing one Euler string to another, we must respect the parenthesization. Contracting an edge in one tree corresponds to deleting a pair of paired parentheses in the corresponding string. Matching an edge in one tree to an edge in the other corresponds to matching the pairs of parentheses and then matching up what is inside one pair against what is inside the other pair.[1] (See Figure 2.)

3 Comparing substrings of Euler strings

The subproblems arising in comparing two Euler strings involve comparing substrings of these strings.

For the purpose of comparing one substring s to another, we ignore each dart in s whose mate does not also occur in s.

We are interested in measuring the distance between substrings s and t of the Euler strings of trees A and B. The operations allowed are: delete a pair of parentheses in one of the strings (i.e. contracting the corresponding edge), and match a pair of parentheses in one string to a pair in the other (i.e. match an edge in one tree to an edge in the other).

In this subsection, we give a recurrence relation for the edit distance $\text{dist}(s, t)$ between two substrings. This recurrence relation implies a dynamic program—in

[1] Shapiro [3] compares trees by comparing their Euler strings. However, he does not seem to treat paired darts in any special way; he compares the strings using an ordinary string-edit distance algorithm. Thus he does not compute the true tree edit-distance.

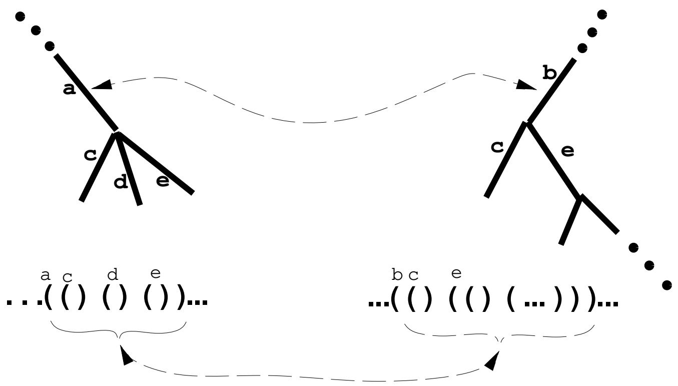


Fig. 2. Matching an edge in one tree to an edge in the other corresponding to matching a pair of parentheses in one string to a pair in the other. Note that if we are to match edge a to edge b, then we must somehow match the interior of the pair of parentheses corresponding to the edge a to the interior of the pair corresponding to b.

fact, it is a simplified (and less efficient) version of the algorithm of Zhang and Shasha[2]

We first give an auxiliary definition

match(s,t)=
  If s has the form $s_1(\!(s_2)\!), t_1[\![t_2]\!]$
    then $\mathrm{dist}(s_1,t_1) + \mathrm{dist}(s_2,t_2) + \mathrm{cost}(\text{change } (\!(\ )\!) \text{to } [\![\ ]\!])$
    else ∞

where the "cost" term represents the cost of changing the label of the edge in tree A corresponding to $(\!(\ )\!)$ into the label of the edge in tree B corresponding to $[\![\ ]\!]$.
Now we give the recurrence relation. The base case is

$$\mathrm{dist}(\Lambda,\Lambda) = 0$$

The recursive part is

$\mathrm{dist}(s,t) = \min\{\mathrm{match}(s,t),$
  if $t = \Lambda$ then ∞ else $\mathrm{dist}(s, t - \mathrm{last}(t)) + \mathrm{cost}(\text{delete last dart of } t)$,
  if $s = \Lambda$ then ∞ else $\mathrm{dist}(s - \mathrm{last}(s), t) + \mathrm{cost}(\text{delete last dart of } s)\}$

where the cost of the deletion is zero if the last dart's mate does not appear in the string. We use the notation $t - \mathrm{last}(t)$ to denote the string obtained from t by deleting its last dart.

[2] Note, however, that we use notation very different from that of Zhang and Shasha. They described their algorithm in terms of (disconnected) forests induced by subsequences of the nodes ordered by preorder, whereas we use substrings of Euler strings.

As an example of how the recurrence is applied, consider the two trees at the bottom-right of Figure 1. Applying the recurrence relation to the Euler strings of these trees, we obtain dist$(aa'bcc'b', xyy'x')$=min{dist$(aa'bcc', xyy'x')$+cost(delete b'), dist$(aa'bcc'b', xyy')$ + cost(delete x'), match$(aa'bcc'b', xyy'x')$}.

Invoking the definition of *match*, the last term is equal to dist(aa', A) + dist(cc', yy') + cost(change bb' to xx').

The correctness of the recurrence relation is based on the following observation, which in turn is based on the fact that deletions and label modifications do not change the order among remaining symbols in a string.

Proposition 1 (Zhang and Shasha). *Consider the cheapest set of operations to transform s and t into the same string x. If the last dart of s is not deleted and the last dart of t is not deleted, then these two darts must both correspond to the last dart in x.*

The value of dist(s, t) is the minimum over at most three expressions, and each depends on the distance between smaller substrings. Therefore, to compute the distance between Euler strings $E(A)$ and $E(B)$, we can use a dynamic program in which there is a subproblem "compute dist(s, t)" for every pair of substrings s, t of $E(A)$ and $E(B)$. The subproblems are solved in increasing order of $|s|$ and $|t|$. The number of pairs s, t of substrings is $O(|A|^2|B|^2)$, and each value dist(s, t) can be calculated in constant time. Thus the time required is $(|A|^2|B|^2)$, which is $O(n^4)$.

4 Obtaining a faster dynamic program

Zhang and Shasha take advantage of the fact that not all substring pairs s, t need be considered, and thereby obtain an algorithm that, depending on the input trees, can be much faster than the naive algorithm. However, in the worst case their algorithm takes $\Omega(n^4)$ time like the naive algorithm.

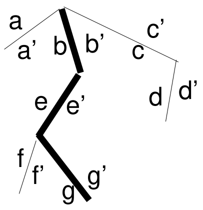
We modify some of their ideas to obtain an algorithm that takes $O(n^3 \log n)$ time. In this section, we present our new ingredients and the algorithm that employs them. In the next section, we will discuss how this algorithm relates to that of Zhang and Shasha.

We start by presenting the ingredients, a sequence of substrings of a tree's Euler string and a decomposition of a tree into paths. Then we show how to combine these ingredients to obtain the algorithm.

4.1 Special substrings

Let T be a tree, and let P be a path starting at the root of T and descending to a leaf. We shall define a sequence of substrings of $E(T)$, called the *special substrings*. The first special substring is simply $E(T)$ itself; each subsequent substring is obtained by deleting either the first or last dart of the previous substring. It is therefore convenient to define the sequence of substrings by defining the sequence of darts to be deleted, which we call the *difference sequence*. The recursive procedure below defines this sequence. (We use $\circ$ to denote concatenation of sequences.)

Diff(T, P):
  Let $r := \mathrm{root}(T)$.
  If r has no children then return the empty sequence
  Else let v be r's child in P.
    Let T_{left} denote the prefix of $E(T)$ ending on (r, v).
    Let T_{right} denote the suffix of $E(T)$ beginning on (v, r).
    Return $T_{\mathrm{left}} \circ T^R_{\mathrm{right}} \circ \mathrm{Diff}(T(v), P(v))$



Let $e_1, \ldots e_{|E(T)|+1}$ denote the difference sequence. For example, for the tree shown to the right of the procedure, the difference sequence is $aa'bc'd'dcb'ee'ff'gg'$. Now we can define the special substrings $t_0, t_1, \ldots, t_{|E(T)|+1}$ of T with respect to P. Substring t_i is obtained from $E(T)$ by deleting $e_1, \ldots, e_i$.

Lemma 1. *The sequence of special substrings t_i of T with respect to P has the following properties.*

1. *For $i = 1, \ldots, m$, the substring t_i is a substring of t_{i-1} and is shorter than t_{i-1} by one dart e_i.*
2. *Suppose e_i is an dart not on P and $e_j = e_i^M$ $(j > i)$. Then t_{i-1} is either t_j $\boldsymbol{e_j}$ $E(T(e_i))$ $\boldsymbol{e_i}$ or $\boldsymbol{e_i}$ $E(T(e_i))$ $\boldsymbol{e_j}$ t_j.*
3. *For each node v of P, the string $E(T(v))$ is one of the special strings.*

Definition 1. *For a nonempty special substring t_i, define the* successor *of t_i to be t_{i+1}, and define the* difference dart *of t_i to be e_{i+1}.*

Thus the successor of t_i is obtained from t_i by deleting the difference dart of t_i. Note that the difference dart of t_i is either the leftmost or the rightmost dart occuring in t_i.

4.2 Decomposition of a rooted tree into paths

The next idea is the employment of a tree decomposition into *heavy paths*. This decomposition is used, e.g., in the dynamic-tree data structure [4]. Given a rooted tree T, define the *weight* of each node v of T be the size of the subtree rooted at v. For each nonleaf node v, let heavy(v) denote the child of v having greatest weight (breaking ties arbitrarily). The sequence of nodes

$$r, \mathrm{heavy}(r), \mathrm{heavy}(\mathrm{heavy}(r)), \ldots$$

defines a descending path which is called the *heavy path*; we denote this path by $P(T)$.

Each of the subtrees hanging off of $P(T)$ has size at most $|T|/2$ (for otherwise the heavy path would enter the subtree). We recursively define the tree decomposition of T to include the path $P(T)$ together with the union of the tree decompositions of all the subtrees hanging off of $P(T)$.

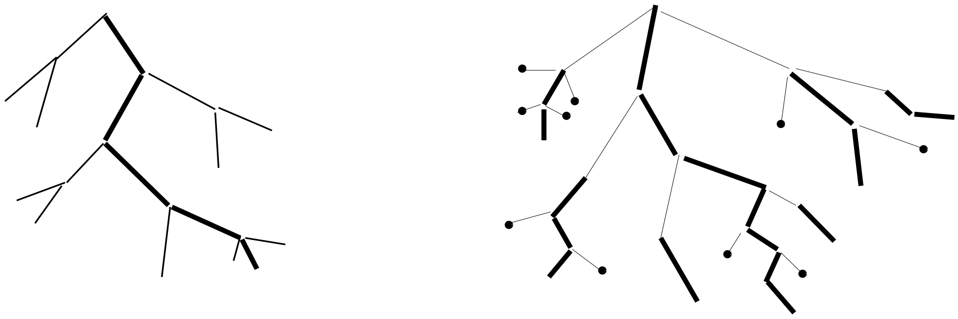


Fig. 3. In the left picture, a tree's heavy path is indicated in bold. On the right is depicted the decomposition of a tree into heavy paths and associated special subtrees. The dots indicate trivial, one-node heavy paths.

Let $P_1, \ldots, P_k$ be the descending paths comprising the tree decomposition of T, and let $r_1, \ldots, r_k$ be the first nodes of these paths. For example, r_1 is the root of T. Define the *collapsed depth* of a node v in T to be the number of ancestors of v that are members of $\{r_1, \ldots, r_k\}$.

Lemma 2 (Sleator and Tarjan, 1983). *For any node v, the collapsed depth of v is at most* $\log |T|$.

For $i = 1, \ldots, k$, let T_i be the subtree of T rooted at r_i. We call each T_i a *special subtree*. We use $P(T_i)$ to denote the heavy path P_i that starts at r_i.

4.3 Special substrings of special subtrees

For a tree T equipped with a decomposition into heavy paths, we define the *relevant* substrings of $E(T)$ to be the union, over all special subtrees T' of T, of the special substrings of T' with respect to $P(T')$.

Lemma 3. *The number of relevant substrings of T is at most* $2|T| \log |T|$

Proof. The proof consists in combining a slight modification of Lemma 7 of Zhang and Shasha [10] with our Lemma 2.

The analogue of Zhang and Shasha's lemma states that

$$\sum_{\text{special subtree } T'} |T'| = \sum_{v \in T} \text{collapsed depth of } v \tag{1}$$

To prove this equality, note that for each node v, the number of special subtrees T' containing v is the collapsed depth of v. Thus v contributes the same amount to the left and right sides.

For each special subtree T', the number of special substrings is one plus the number of darts in T', which is $1 + 2(|T'| - 1)$. Thus the total number of relevant substrings is at most the sum, over all special subtrees T', of $2|T'|$. By combining (1) with Lemma 2, we bound this sum by $2|T| \log |T|$.

Lemma 4. *For every node v of T, $E(T(v))$ (the Euler string of the subtree rooted at v) is a relevant substring.*

Proof. Every node v occurs in the heavy path P of some special subtree T'. By part 3 of Lemma 1, $E(T'(v))$ is a special substring of T', hence a relevant substring of T.

4.4 Unrooted, ordered trees

Given an unrooted, ordered tree T, let $E^*(T)$ denote the Euler tour of the darts of T, interpreted as a cyclic string. For each dart d, we can obtain a non-cyclic string from $E^*(T)$ by designating d as the starting dart of the string. Each non-cyclic string thus obtained is the Euler string of one of the *rooted* versions of T, and conversely each rooted version of T can be obtained in this way. Let $R(T)$ denote the set of these Euler strings. Note that $|R(T)| = O(|T|)$.

4.5 The new dynamic program

We finally give the new algorithm for computing the edit distance between trees A and B. Essentially the same algorithm is used for the rooted case and the unrooted case.

- If B is unrooted, root it arbitrarily.
- Find a heavy-path decomposition of B, and then identify the relevant substrings of each special subtree of B.
- By dynamic programming, calculate $\text{dist}(s,t)$ for every substring s of the cyclic string $E^*(A)$ and every relevant substring t of B.
- For the rooted distance, output $\text{dist}(\bar{s},\bar{t})$, where $\bar{s}$ is the Euler string of the (rooted) tree A, and $\bar{t}$ is that of B.
- For the unrooted distance, output $\min_{s\in R(A)} \text{dist}(s,\bar{t})$, where $\bar{t} = E(B)$. Note that the min is over all Euler strings of rooted versions of A.

For the unrooted edit-distance between A and B, we see that the algorithm arbitrarily roots B and compares it (using *rooted* edit-distance) to every rooted version of A. The correctness of this approach is intuitively evident, and has been formally proved by Srikanta Tirthapura [6]

Now we consider the analysis. The dominant step is the dynamic programming. The number of substrings s of $E^*(A)$ is $O(|A|^2)$. By Lemma 3, the number of relevant substrings t of B is $O(|B|\log|B|)$. We show below how each value $\text{dist}(s,t)$ can be calculated in constant time from a few "easier" values $\text{dist}(s',t')$. Hence the time (and space) required is $O(|A|^2|B|\log|B|)$, which is $O(n^3\log n)$.

We must show that the answer to every subproblem can be computed in constant time from the answers to "easier" subproblems.

Note that the recurrence relation in Section 3, whose correctness is based on Observation 1, relies on deletion of the rightmost darts of substrings. We can invoke a symmetric version of Observation 1 to justify an alternative recurrence relation based on deletion of the leftmost darts.

The ability to delete from the left gives us freedom which we exploit as follows. Our goal is to compute $\text{dist}(s,t)$ from a few "easier" values $\text{dist}(s',t')$ (i.e. where $|s'|+|t'|$ is smaller than $|s|+|t|$). We need to ensure that for each such value we use, the substring t' is a relevant substring of B. Since t is itself relevant, the successor of t is such a relevant substring t'. However, the successor of t is either the substring obtained from t by deleting the last dart in t or the substring obtained by deleting the first dart of t.

We now give a formula for computing $\text{dist}(s,t)$. It is computed as the minimum of three terms.

$$\text{dist}(s,t) = \min\{\text{Delete-From-}s(s,t), \text{Delete-From-}t(s,t), \text{Match}(s,t)\}$$

We proceed to define the terms.

Delete-From-$t(s,t)$:
If t is the empty string, return ∞
Let e be the difference dart of t.
If e^M occurs in t, return $\text{dist}(s, t-e) + \text{cost}(\text{delete } e \text{ from } t)$
else return $\text{dist}(s, t-e)$

Delete-From-$s(s,t)$:
If s is the empty string, return ∞.
If t is the empty string, let e be the rightmost dart of s
else if the difference dart of t is the rightmost dart of t
then let e be the rightmost dart of s.
else let e be the leftmost dart of s.
If e^M occurs in s, return $\text{dist}(s-e, t) + \text{cost}(\text{delete } e \text{ from } s)$
else return $\text{dist}(s-e, t)$

Match(s,t):
If s or t is empty, return ∞.
Let e be the difference dart of t.
If t has the form $t'\ \boldsymbol{e^M}\ t''\ \boldsymbol{e}$ then write s as $s'\ \boldsymbol{e'^M}\ s''\ \boldsymbol{e'}$.
If t has the form $\boldsymbol{e}\ t''\ \boldsymbol{e^M}\ t'$ then write s as $\boldsymbol{e'}\ s''\ \boldsymbol{e'^M}\ s'$.
Return $\text{dist}(s',t') + \text{dist}(s'',t'') + \text{cost}(\text{match } \boldsymbol{e} \text{ in } \boldsymbol{t} \text{ with } \boldsymbol{e'} \text{ in } \boldsymbol{s})$

The correctness of the formula follows from Observation 1 and its symmetric version.

It remains to verify that, for each "easier" distance subproblem $\text{dist}(s',t')$ appearing in the formula, t' is a relevant substring.

In Delete-From-t, the substring t' is $t-e$ where e is the difference dart of t. Hence $t-e$ is the successor of t, and is therefore relevant.

In Match, we have to check the substrings t' and t''. Since t'' is the Euler string of a rooted subtree, it is relevant by Lemma 4. We use part 2 of Lemma 1 to show that t' is relevant. Assume t has the form $t'\ \boldsymbol{e^M}\ t''\ \boldsymbol{e}$ (the other case is symmetric). Say e is the i^{th} difference dart, so t is the $i-1^{st}$ special substring, t_{i-1}. Then by part 2, t' is the j^{th} special substring t_j, where e^M is the j^{th} difference dart.

5 Concluding remarks

5.1 The algorithm of Zhang and Shasha

We have presented an algorithm that solves the edit-distance problem for both rooted and unrooted ordered trees. Our algorithm has a better worse case bound than the previous algorithm for rooted trees, that of Zhang and Shasha, and is especially suitable for the unrooted case, for which Zhang and Shasha's algorithm alone is insufficient.

However, for the rooted case Zhang and Shasha's algorithm may run faster depending on the structure of the trees being compared. For this reason, it may be useful for future research to interpret their algorithm in the present framework; perhaps someone can combine the advantages of the two algorithms.

In the algorithm of Zhang and Shasha, the analogue of decomposition into heavy paths might be called decomposition into *leftmost* paths. The leftmost path descends via leftmost children. The disadvantage of this decomposition is that it does not guarantee small collapsed depth; indeed, the collapsed depth can be $\Omega(n)$.

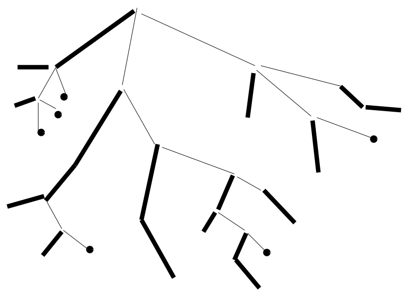


Fig. 4. The decomposition of a tree into leftmost paths is depicted. The dots indicate trivial, one-node leftmost paths.

The leftmost decomposition has a considerable benefit, however. One can define the special substrings of a subtree to be the prefixes of the Euler string of the subtree. The successor of a special substring is obtained by deleting its last symbol. The advantage is that only rightmost deletes are needed in the algorithm. For this reason, the decomposition idea can be applied to *both* trees A and B being compared, not just B. The number of subproblems is therefore $O(|A|\,\mathrm{LR\_colldepth}(A)\,|B|\,\mathrm{LR\_colldepth}(B))$ instead of $O(|A|^2\,|B|\,\mathrm{LR\_colldepth}(B))$.

One must verify that the analogue of Lemma 1 holds for this set of special substrings. Part 1 holds trivially. Part 2 is easy to verify. Part 3 does not hold; however, because of the leftmost decomposition, something just as useful does hold. Say two substrings of $E(T)$ are *equivalent* if upon removal of unmatched darts the strings become equal. Note that we ignore such darts in computing edit-distance; thus the edit-distance between equivalent substrings is zero. The notion of equivalence gives us a variant of part 3: for each node v in the leftmost path of a subtree T', there is a special substring (a prefix of $E(T')$) that is equivalent to $E(T(v))$.

5.2 Related problems on trees

Zhang and Shasha point out that their dynamic program can be adapted to solve similar problems, and give as examples two possible generalizations of approximate string matching to trees; these problems involve finding a modified version of a pattern tree in a text tree.

For some applications, comparison of *unordered* trees would make more sense. Unfortunately, computing edit-distance on unordered trees is NP-complete, as shown by Zhang, Statman, and Shasha [12]. Zhang has given an algorithm [9] for computing a kind of constrained edit-distance between unordered trees.

One might consider generalizing from edit-distance between ordered trees to edit-distance between planar graphs. However, the problem of finding a Hamiltonian path in a planar graph can be reduced to finding the edit-distance between planar graphs (by using the dual graph).

5.3 Acknowledgements

Many thanks to Srikanta Tirthapura for his perceptive remarks and his skepticism. He has been a great help in this research.

References

1. J. L. Gross and and T. W. Tucker, *Topological Graph Theory*, Wiley, 1987.
2. M. Maes, "On a cyclic string-to-string correction problem," *Information Processing Letters* 35 (1990), pp. 73-78.
3. B. A. Shapiro, "An algorithm for comparing multiple RNA secondary structures," *Computer Applications in the Biosciences* (1988), pp. 387-393.
4. D. D. Sleator and R. E. Tarjan, "A data structure for dynamic trees," *Journal of Computer and System Sciences* 26 (1983), pp. 362-391.
5. K.-C. Tai, "The tree-to-tree correction problem", *Journal of the Association for Computing Machinery* 26 (1979), pp. 422-433.
6. Srikanta Tirthapura, personal communication, 1998.
7. R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *Journal of the Association for Computing Machinery* 21, (1974), pp. 168-173.
8. J. T.-L. Wang, K. Zhang, K. Jeong, and D. Shasha, "A system for approximate tree matching," *IEEE Transactions on Knowledge and Data Engineering* 6 (1994), pp. 559-571. 5
9. K. Zhang, "A constrained edit distance between unordered labeled trees," *Algorithmica* 15 (1996), pp. 205-222.
10. K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems, *SIAM Journal on Computing* 18 (1989), pp. 1245-1262.
11. K. Zhang and D. Shasha, "Approximate tree pattern matching," Chapter 14 of *Pattern Matching Algorithms*, Oxford University Press (1997)
12. K. Zhang, R. Statman and D. Shasha, "On the editing distance between unordered labeled trees," *Information Processing Letters* 42 (1992), pp. 133-139