The Editing Distance Between Trees *

Carsten Isert
Institut für Informatik
Technische Universität München
isert@in.tum.de

Abstract

The similarity between two given structures can be of vital interest in various problem definitions. An example with a large application basis — tolerant full text searches — is the similarity of strings. We present the extension of this problem to trees, where applications lie in comparing grammar parses for error recovery, image clusters, chemical structures, or most important RNA (Ribonucleic Acid) secondary structures. In all these cases, the trees have the property that they are labeled and ordered, i.e. the left-to-right order among siblings is significant. Among the different possible metrics we present in detail efficient ways for computing the tree edit distance by a dynamic programming approach and we will give a brief overview of tree alignment.

1 Introduction

Generally, there are many ways to measure the similarity — or distance — between structures like strings, trees, or graphs. In this paper we will be particularly concerned about trees where we can employ measures like the largest common subtree, the smallest common supertree, tree edit distance, transferable ratio, or tree alignment. For the problem of RNA secondary structure analysis, which is the main motivation for dealing with this problem, tree edit distance and tree alignment are good and sensible measures. The RNA secondary structure can be represented by an ordered labeled tree with the five component types: stem (S), hairpin (H), bulge (B), interior loop (I), and multi-branch loop (M). Because different sequences of RNA can produce similar secondary structures it is necessary for understanding their comparative functionality to measure the similarity at the level of the secondary, tree like structure. Additionally, the secondary structure influences translation rates from RNA to proteins.

Figure 1 depicts a sample secondary structure generated by a computer program and clearly visualizes the meaning of the previously mentioned components and the tree like structure. The connections between the various *circles* correspond to the stem type and are not annotated in the figure.

Previously, these structures were represented as parenthesized strings and string distance metrics were used. Although this works in some cases, the problem in trees is harder because ancestor relationships must be preserved and certain prefixes conditions of strings can no longer be exploited.

In the remainder of this paper we will describe, proof, and analyze a simple dynamic programming algorithm for solving the tree edit distance problem. Therefore, we will specify the edit operations modify, insert, and delete and give some of the necessary definitions in Section 2, describe how a mapping between two trees can be constructed, list some properties of mappings, and motivate their actual function in Section 3, present a detailed description of the simple dynamic programming algorithm with various additional definitions, Lemmata, and an example in Section 4, analyze the time complexity of this algorithm to be $O(|T_1| \times |T_2| \times \min(depth(T_1), leaves(T_1)) \times \min(depth(T_2), leaves(T_2)))$ and the space complexity to be $O(|T_1| \times |T_2|)$ in Section 5, and optimize these bounds with certain extensions in Section 6. Furthermore, we will give a short introduction and overview of tree alignment, how it works and give an example in Section 7. Finally, we summarize and conclude our observations in Section 8.

*This report is an extension to the presentation given at the Ferienakademie 1999, Sarntal, Italy for course 2: Bäume: Algorithmik und Kombinatorik

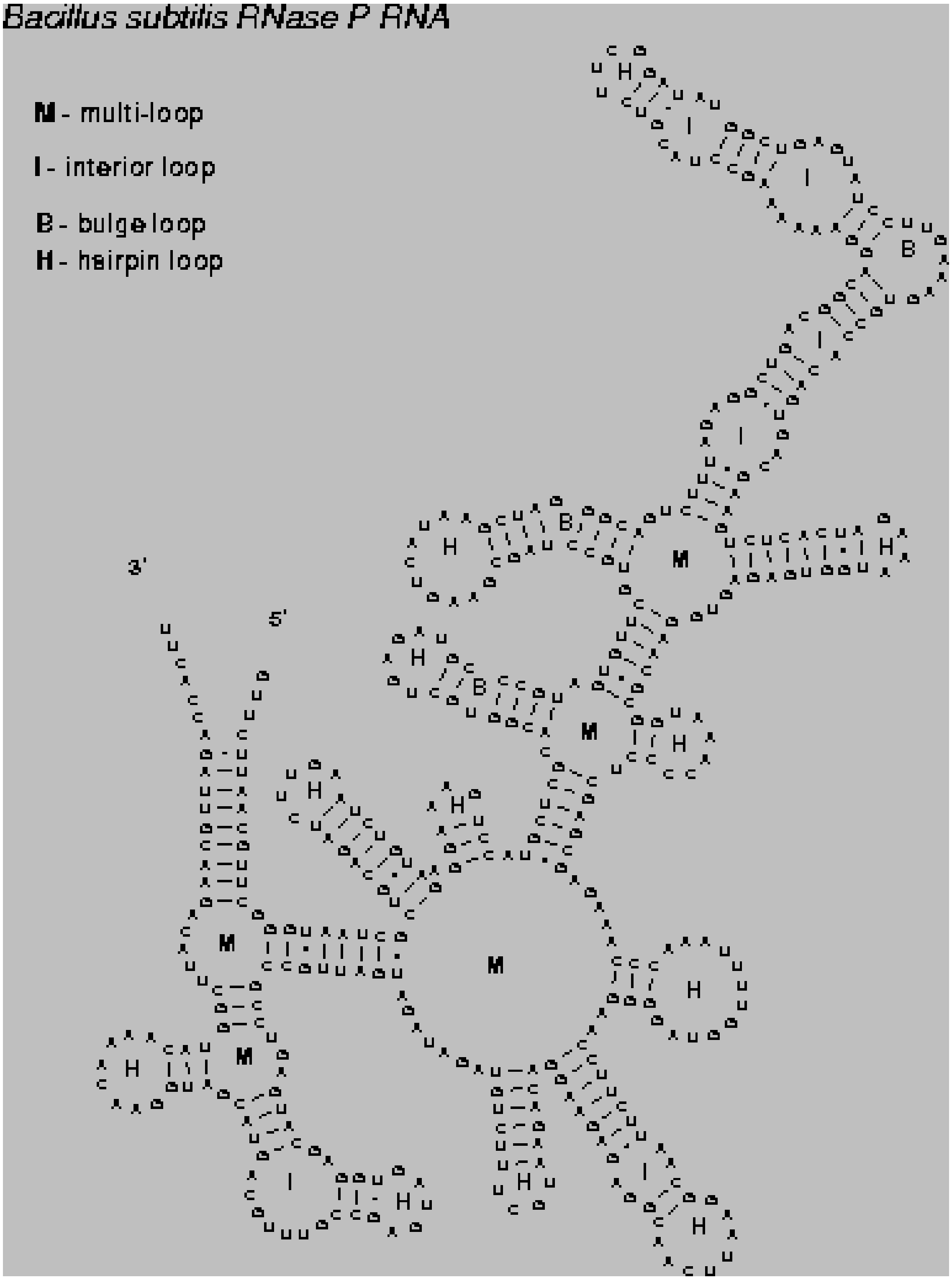

Figure 1: Example of RNA Secondary Structure [5]

2 Definitions

A necessary precondition for an edit algorithm is to clearly specify its basis, i.e. which edit operations are available and to formally define them. Specifically, in our case these operations are modify, insert, and delete.

The modify operation is depicted in Figure 2 and changes the label of a node from a to b. It will be formally represented by the notation $(a \mapsto b)$ or (a, b). Ambiguity between node labels arises when multiple nodes can have the same label, e.g. in RNA secondary structure analysis this is the rule. This can be avoided by associating each label with the unique number of its node, but throughout this paper it will always be clear from the context which node is meant and therefore, we omit this explicit notation.

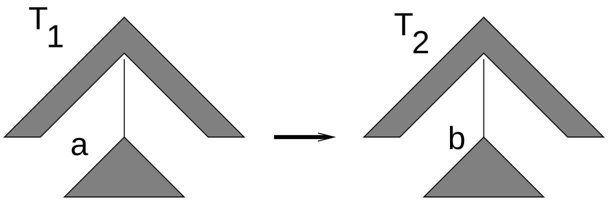

Figure 2: Modify

The insert operation is depicted in Figure 3 and inserts a node b as an additional son of node a and makes a consecutive number of children of node a become children of node b. Insert will be fomally represented by the notation $(\Lambda \mapsto b)$ or (Λ, b). Again, this notation is not unambiguous when looking at tree T_1 alone because it is unclear which set of children should be assigned to node b and which remain at node a. However, if the operation includes both trees, the children of node b in tree T_2 specify the set needed in tree T_1. As we will see later, this information is always available in our approach. Additionally, it is noteworthy that it is also possible to insert nodes above the root node.

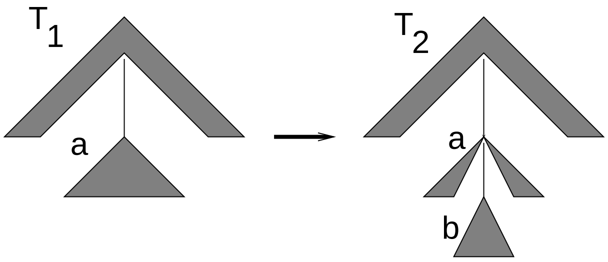

Figure 3: Insert

The delete operation is depicted in Figure 4 and deletes son b from node a and makes the children of node b become additional children of node a. This operation will be formally represented by the notation $(b \mapsto \Lambda)$ or (b, Λ). It is basically the complement of the insert operation. If the root has only one son then it is also allowed to delete the root.

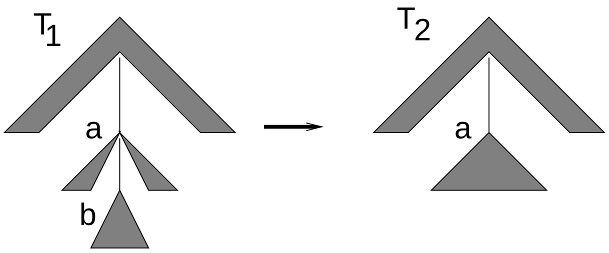

Figure 4: Delete

Each of the previously defined edit operations has an associated cost $\gamma(a \mapsto b)$ or $\gamma(a, b)$. For our algorithm to work, this cost function is required to be a distance metric, i.e. it must have the following properties:

- $\gamma(a \mapsto b) \geq 0; \quad \gamma(a \mapsto a) = 0$
- $\gamma(a \mapsto b) = \gamma(b \mapsto a)$
- $\gamma(a \mapsto c) \leq \gamma(a \mapsto b) + \gamma(b \mapsto c)$

Definition. Let $S = s_1, \ldots, s_k$ be a sequence of edit operations. The S-derivation from tree T_1 to tree T_2 be is characterized by a sequence of trees $U_0 \ldots U_k$ where $U_0 = T_1$, $U_k = T_2$, and $U_{i-1} \mapsto U_i$ via s_i for $1 \leq i \leq k$ The cost of S is $\gamma(S) = \sum_{i=1}^{i=|S|} \gamma(s_i)$
Definition. This allows us to write the distance between the two trees T_1 and T_2 as:

$$\delta(T_1, T_2) = \min\{\gamma(S)| \text{ S is a sequence taking } T_1 \text{ to } T_2\}$$

3 Mappings

Before describing the formal definitions of mappings, we will first describe the intuitive meaning of a mapping by the example in Figure 5.

To uniquely identify nodes in a tree we will use a *left-to-right postorder numbering* of nodes throughout this paper. $T[i]$ will denote the ith node in this postorder numbering and can be found inside the nodes in Figure 5. Different node colours represent different labels.

A mapping is basically a graphical specification of edit operations where a dotted line between two nodes indicates that the modify operation must be applied if the labels of $T_1[i]$ and $T_2[j]$ mismatch. The nodes of T_1 that are not touched by a dotted line have to be deleted from T_1 and the nodes of T_2 not touched by a dotted line must be inserted into T_1. As an example consider the mapping of Figure 5 where we will have to delete node 5, insert node 4, and relabel node 6 to transform tree T_1 into T_2.

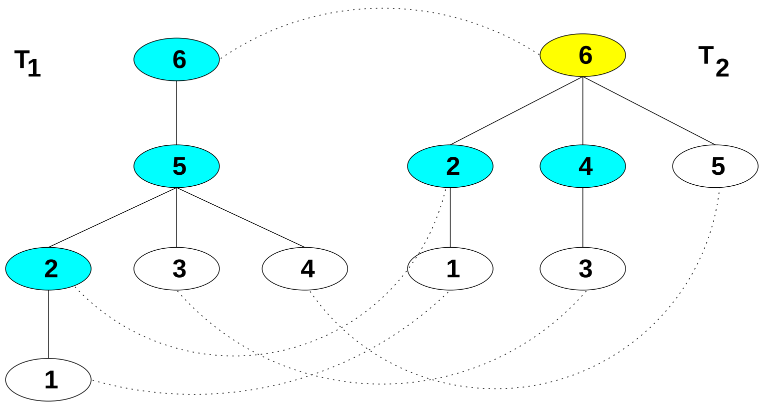

Figure 5: Mapping

Definition. M is a mapping if the following conditions hold for all $(i_1, j_1), (i_2, j_2) \in M$ with $1 \leq i_k \leq |T_1|$ and $1 \leq j_k \leq |T_2|$ and $k \in \{1, 2\}$:

- One-to-one: $i_1 = i_2$ iff $j_1 = j_2$
- Sibling order preserved: $T_1[i_1]$ is to the left of $T_1[i_2]$ iff $T_2[j_1]$ is to the left of $T_2[j_2]$
- Ancestor order preserved: $T_1[i_1]$ is an ancestor of $T_1[i_2]$ iff $T_2[j_1]$ is an ancestor of $T_2[j_2]$

Definition. This gives rise to an extension of the cost function. Let I and J be the sets of nodes not touched by mapping M in T_1 and T_2 respectively, then:

$$\gamma(M) = \sum_{(i,j)\in M} \gamma(T_1[i] \mapsto T_2[j]) + \sum_{i\in I} \gamma(T_1[i] \mapsto \Lambda) + \sum_{j\in J} \gamma(\Lambda \mapsto T_2[j])$$

Definition. Let M_1 be a mapping from T_1 to T_2 and M_2 a mapping from T_2 to T_3, then we define $M_1 \circ M_2 = \{(i,j) | \exists k \text{ such that } (i,k) \in M_1 \text{ and } (k,j) \in M_2\}$.
Mappings as defined above have several properties which we will need in the development process of our algorithm.

Lemma 1 *$M_1 \circ M_2$ is a mapping and $\gamma(M_1 \circ M_2) \leq \gamma(M_1) + \gamma(M_2)$*

Proof. For the first part it is easy to check that all of the definitions of a mapping are satisfied by $M_1 \circ M_2$ because of the *iff* conditions. Considering the composed mapping $M_1 \circ M_2$ and the respective insertion and deletion sets I and J there are only three possible situations for indices $i \in T_1$ and $j \in T_2$: First, i can be an element of I is not touched by a dotted line. Second, i is touched by a dotted line and therefore has an associated node $j \in T_2$, i.e. $(i,j) \in M_1 \circ M_2$. Third, there can be nodes $j \in T_2$ not touched by a dotted line, i.e. $j \in J$. No other situations can occur because $M_1 \circ M_2$ is a mapping. Each of these situations corresponds to an edit operation and each edit operation preserves the triangle inequality. Therefore, the above assumption is correct.

Lemma 2 *For a sequence $S = s_1, \ldots, s_k$ from T_1 to T_2, there exists a mapping M such that $\gamma(M) \leq \gamma(S)$. For any mapping M there exists a sequence S such that $\gamma(M) = \gamma(S)$*

Proof. The first part is proven by induction on k. The base case is $k = 1$, i.e. it consists of only one edit operation. It is clear that each edit operation alone satisfies the above assumption. The general case starts with the assumption that the sequence $S_1 = s_1, .., s_{k-1}$ has a corresponding mapping M_1 such that $\gamma(M_1) \leq \gamma(S_1)$. Let M_2 denote the mapping for the final edit operation s_k. Applying Lemma 1 yields $\gamma(M_1 \circ M_2) \leq \gamma(M_1) + \gamma(M_2) \leq \gamma(S_1) + \gamma(s_k) = \gamma(S)$ q.e.d. The proof for the second part is constructive. First, perform all deletes, then all modifies, then all inserts. This results in a sequence S that has the same costs as the mapping.
Definition. The equivalence of Lemma 2 makes the definition of distances constructive and lets us exactly define what we mean by a minimum cost mapping between trees T_1 and T_2:

$$\delta(T_1, T_2) = \min\{\gamma(M) | M \text{ is a mapping from } T_1 \text{ to } T_2\}.$$

4 Simple Dynamic Programming Algorithm

Some more definitions are necessary before we can actually start with the algorithm. Figure 6 visualizes some of these definitions.

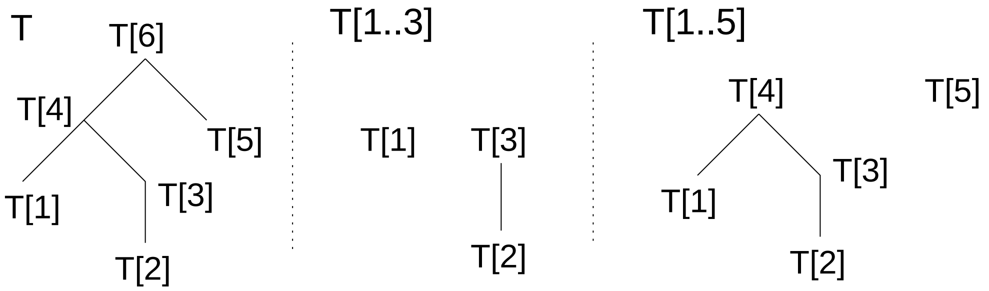

Figure 6: Forests

Definitions. As defined earlier, $T[i]$ refers to the ith node in the left-to-right postorder numbering. As our algorithm will compute not only distances between trees, but also between ordered forests, we define $T[i..j]$ to be the forest of nodes numbered from i to j inclusively. Figure 6 shows three examples of forests, all beginning with node 1 (in this case $T[1..i]$ is also called $forest(i)$). If $i > j$ then $T[i..j] = \emptyset$ which simplifies the notion and proof of our algorithm, but must be considered explicitly in an implementation.
Definitions. $l(i)$ is leftmost leaf descendant of the subtree rooted at $T[i]$. This notation will be used extensively in the remainder of this paper to specify reasonable sets of ordered forest.
Definitions. The parent of node $T[i]$ is denoted $p(i)$ and the set of ancestors is given by $anc(i) = \{p^k(i)|0 \leq k \leq depth(i)\}$ where $p^0(i) = i, p^k(i) = p(p^{k-1}(i))$.
Definitions. $fdist(i'..i, j'..j)$ is the distance between the forests $T_1[i'..i]$ and $T_2[j'..j]$ and $tdist(i,j)$ is the distance between the subtrees rooted at $T_1[i]$ and $T_2[j]$. Clearly, the value of $tdist$ at the roots gives us the solution to our problem. Therefore, it is useful to compute the values of the subtrees in $tdist$ and between certain forests in $fdist$.

Lemma 3 *For $i_1 \in anc(i)$ and $j_1 \in anc(j)$ the distance $fdist$ between the two forests $T_1[l(i_1)..i]$ and $T_2[l(j_1)..j]$ satisfies:*

- $fdist(\emptyset, \emptyset) = 0$
- $fdist(T_1[l(i_1)..i], \emptyset) = fdist(T_1[l(i_1)..i-1, \emptyset]) + \gamma(T_1[i], \Lambda)$
- $fdist(\emptyset, T_2[l(j_1)..j]) = fdist(\emptyset, T_2[l(j_1)..j-1]) + \gamma(\Lambda, T_2[j])$

Proof. The first case is obvious as no operations are executed. In the second case the only operations that can be performed to transform the forest $T_1[l(i_1)..i]$ to the empty set are deletions. Therefore, the costs for deleting all elements are summed up recursively. For the third case, all nodes must be inserted to the empty set and the respective costs are added up. The particular choice of $l(i_1)$ and $l(j_1)$ as left bounds is not really needed for this Lemma, but the reasons will become apparent in the next Lemmata.

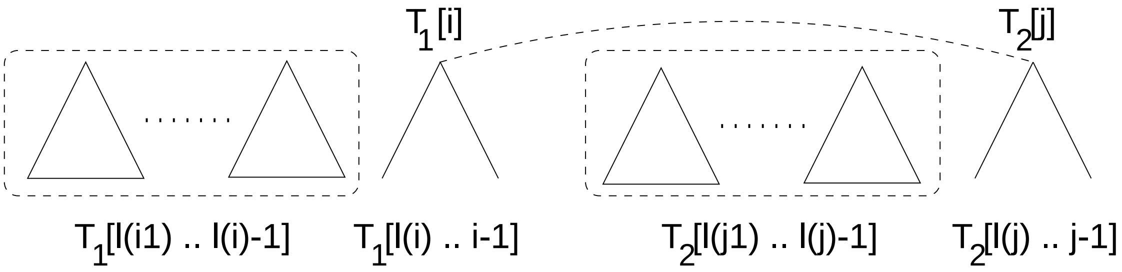

Figure 7: Visualization of $fdist(l(i_1)..i, l(j_1)..j)$

Figure 7 shows the situation for intermediate values of $fdist$ with the previously mentioned left bounds $l(i_1)$ and $l(j_1)$ with $i_1 \in anc(i)$ and $j_1 \in anc(j)$. In the following explanations we will only deal with T_1, but all results are equally applicable to T_2 as well. From the definition of $l(k)$ it is clear that $l(i_1) \leq i$. To compute the intermediate values for $fdist$ we need a good partition of the problem so that it is easy to solve each part. The tree rooted at $T_1[i]$ contains as children the elements numbered from $l(i)$ to $i-1$ (if these elements exist). The remaining elements of the set $l(i_1)..i$ are organized in a forest and contain elements from $l(i_1)$ to $l(i)-1$ which is the highest numbered element not in the subtree rooted at $T_1[i]$. This partition is shown for T_1 and T_2 in Figure 7.

Looking at a mapping between these two forest, corresponding to the three different edit operations three different situations can occur. If node $T_1[i]$ is not touched by a dotted line it has to be deleted. In this case the distance between the forests without $T_1[i]$ can be directly added to the costs for deletion. Insertion of $T_2[j]$ is just the sum of the old mapping plus the costs for insertion. A somewhat different situation occurs when both nodes are connected by a dotted line. It is clear that the costs for changing the label (if they are unequal) must contribute to the new value. However, because both forests are affected by this operation we must perform separate lookups for the forests to the left of $T_1[i]$ and $T_2[j]$ and for the children of $T_1[i]$ and $T_2[j]$. The operation with the least cost is selected. These facts are formalized by the following Lemma.

Lemma 4 *The intermediate solutions to the distance between the two forests $T_1[l(i_1)..i]$ and $T_2[l(j_1)..j]$ with $i_1 \in anc(i)$ and $j_1 \in anc(j)$ can be computed by*

$$fdist(l(i_1)..i, l(j_1)..j) = \min \begin{cases} fdist(l(i_1)..i-1, l(j_1)..j) + \\ \quad \gamma(T_1[i] \mapsto \Lambda) \\ fdist(l(i_1)..i, l(j_1)..j-1) + \\ \quad \gamma(\Lambda \mapsto T_2[j]) \\ fdist(l(i_1)..i-1, l(j_1)..j-1) + \\ \quad fdist(l(i)..i-1, l(j)..j-1) + \\ \quad \gamma(T_1[i] \mapsto T_2[j]) \end{cases}$$

Proof. If we look at the associated mapping M, three cases are possible:

- $T_1[i]$ is not touched by a line in M, then the first case is true and we delete the node from T_1.
- $T_2[j]$ is not touched by a line in M, then the second case is true and we insert the node into T_1.
- $T_1[i]$ and $T_2[j]$ are both touched by a line in M, then $(i, j) \in M$ because of the ancestor and sibling conditions.

For an implementation it is convenient to split up the third case of Lemma 4 in a way that distinguishes between proper forests and proper trees. The condition for the set of nodes numbered $l(i_1)..i$ to be a proper tree is that $l(i_1) = l(i)$. Otherwise it is a proper forest. This distinction is motivated by the dynamic programming technique we will use for the algorithm. Distances between trees are our primary interest and we will store them in a permanent array for later lookup during the computation of the forest distances as needed by Lemma 4 and depicted in Figure 7. This will naturally be done in a bottom-up fashion. Distances between forests, on the other hand, are stored only temporarily as they are gathered into tree results as soon as possible. These observations are formalized by the following Lemma.

Lemma 5 *For $i_1 \in anc(i)$ and $j_1 \in anc(j)$ the distance between the two forests $T_1[l(i_1)..i]$ and $T_2[l(j_1)..j]$ can be computed by*

- *If $l(i) = l(i_1)$ and $l(j) = l(j_1)$*

$$\begin{aligned} fdist(l(i_1)..i, l(j_1)..j) &= \min \begin{cases} fdist(l(i_1)..i-1, l(j_1)..j) + \gamma(T_1[i], \Lambda) \\ fdist(l(i_1)..i, l(j_1)..j-1) + \gamma(\Lambda, T_2[j]) \\ fdist(l(i_1)..i-1, l(j_1)..j-1) + \gamma(T_1[i], T_2[j]) \end{cases} \\ tdist(i, j) &= fdist(l(i_1)..i, l(j_1)..j) \end{aligned}$$

- *If $l(i) \neq l(i_1)$ or $l(j) \neq l(j_1)$*

$$fdist(l(i_1)..i, l(j_1)..j) = \min \begin{cases} fdist(l(i_1)..i-1, l(j_1)..j) + \gamma(T_1[i], \Lambda) \\ fdist(l(i_1)..i, l(j_1)..j-1) + \gamma(\Lambda, T_2[j]) \\ fdist(l(i_1)..i-1, l(j_1)..j-1) + tdist(i, j) \end{cases}$$

Proof. The first case follows immediately from Lemma 4 when we insert the respective values of the indices. The first term in the third case of that Lemma becomes: $fdist(l(i_1)..l(i)-1, l(j_1)..l(j)-1) = fdist(l(i)..l(i)-1, l(j)..l(j)-1) = fdist(\emptyset, \emptyset) = 0$. By the condition $l(i_1) = l(i)$ we know that $T_1[l(i_1)..i]$ is a proper tree and the distance computed in $fdist$ is really a tree distance. The same holds for T_2.

To proof the second case we need two additional inequalities. $fdist(l(i_1)..i, l(j_1)..j) \leq fdist(l(i_1)..l(i)-1, l(j_1)..l(j)-1) + tdist(i, j)$ is true because the right side represents a particular mapping and is not necessarily optimal. The same argumentation can be applied to proof $tdist(i, j) \leq fdist(l(i)..i-1, l(j)..j-1) + \gamma(T_1[i], T_2[k])$ correct. The former inequality tells us that the distance between the two forest must be actually less than the value given in the third term of the second case. However, inserting the latter inequality yields exactly the result from Lemma 4, so our replacement of $tdist$ is correct.

Additionally, we can see from case one of Lemma 5 that we can avoid computing certain subtree values of $tdist$ as they are not needed in the computation process. These values have the property that i is in the path

from $l(i_1)$ to i_1 and j is in the path from $l(j_1)$ to j_1 and are obtained by the forest calculations. However, they must also be stored in the permanent *tdist* array. This implication leads to the following definition of the sets of nodes which need separate computations.
Definition. The set $LR\_keyroots(T)$ is defined to be

$$LR\_keyroots(T) = \{\, k \mid \text{there exists no } k' > k \text{ such that } l(k) = l(k')\}.$$

This set contains the root and all nodes with left siblings. For the example trees in Figure 5 these sets are $LR\_keyroots(T_1) = \{3, 4, 6\}$ and $LR\_keyroots(T_2) = \{4, 5, 6\}$.

With the results from Lemma 5 and the set of nodes which need separate computations it is straightforward to implement an efficient dynamic programming algorithm that computes the edit distance between two given trees. The main loop does all preprocessing and initializations and makes the necessary calls to the function *treedist*. Results are stored in the permanent array *tdist* which in the end contains the distances between all possible pairs of subtrees and at position $(|T_1|, |T_2|)$ the distance between the two roots, i.e. the desired distance between the two trees. Note that *LR_keyroots* is a boolean array with a value of true indicates that a particular element is in that set. The pseudo C code for the main loop is as follows:

```
preprocessing();   // left1/2, LR_keyroots1/2
                   // init tdist array, costs

for (i=1 ; i <= |T1| ; i++)
  if (LR_keyroots1[i])
    for (j=1 ; j <= |T2| ; j++)
      if (LR_keyroots2[j])
        treedist(i,j);

return (tdist[i][j]);
```

The function $treedist(pos1, pos2)$ computes the distances between the two forests $l(pos_1)..pos_1$ and $l(pos_2)..pos_2$ by using previously computed tree values and costs for operations. It stores these results in the temporary array *fdist* and also fills in values for distances between subtrees into the array *tdist*. The pseudo C code for *treedist* loop is as follows:

```
treedist(pos1, pos2)
{
  bound1 = pos1 - left1[pos1] + 2;
  bound2 = pos2 - left2[pos2] + 2;

  // init fdist array to size [bound1][bound2]

  bound1 = pos1 - left1[pos1] + 2;
  bound2 = pos2 - left2[pos2] + 2;

  // init fdist array to size [bound1][bound2]

  fdist[0][0] = 0;
  for (i=1, k=left1[pos1] ; i<bound1 ; i++, k++)
    fdist[i][0] = fdist[i-1][0] + c[k][0];
  for (j=1, l=left2[pos2] ; j<bound2 ; j++, l++)
    fdist[0][j] = fdist[0][j-1] + c[0][l];

  for (k=left1[pos1], i=1 ; k<=pos1 ; k++, i++)
    for (l=left2[pos2], j=1  ; l<=pos2 ; l++, j++)
      if ((left1[k]==left1[pos1]) && (left2[l]==left2[pos2])) {
        fdist[i][j] = MIN(fdist[i-1][j] + c[k][0],
```

```
                        fdist[i][j-1] + c[0][l],
                        fdist[i-1][j-1] + c[k][l]);
          tdist[k][l] = fdist[i][j];
        }
        else
          m = left1[k] - left1[pos1]; n = left2[l] -left2[pos2];
          if (m<0) m=0; if (n<0) n=0;
          fdist[i][j] = MIN(fdist[i-1][j] + c[k][0],
                        fdist[i][j-1] + c[0][l],
                        fdist[m][n] + tdist[k][l]);
}
```

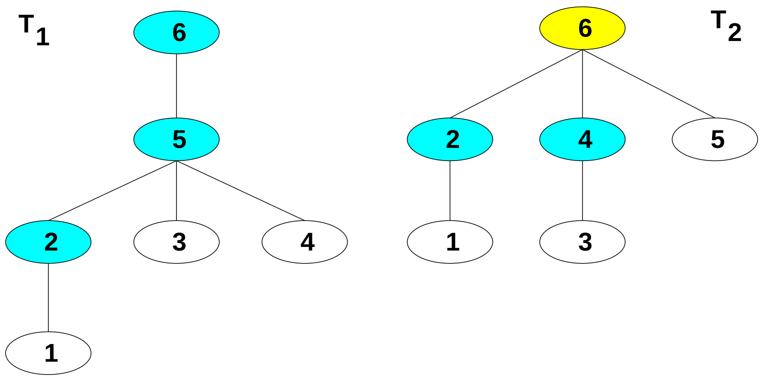

Figure 8: Example Tree

An actual implementation of this code can be found at [1]. Figure 8 depicts an example problem which we will observe a little bit closer in the following. The partial output for a sample run comparing the two trees in Figure 8 with a unit cost model looks like this:

```
Left1 array: 1 1 3 4 1 1
Left2 array: 1 1 3 3 5 1
LR_keyroots: 0 0 1 1 0 1   (i.e. {3,4,6})
LR_keyroots: 0 0 0 1 1 1   (i.e. {4,5,6})
```

Left array gives the values for $l(i)$ in the postorder numbering and a one in the *LR_keyroots* output indicates the occurrence of this element in the respective set. Some distances and the solution to this problem are given in the following:

```
Forestdist for 3 4
 0  1  2
 1  0  1

Treedist after 3 4
 0  0  0  0  0  0
 0  0  0  0  0  0
 0  0  0  1  0  0
 0  0  0  0  0  0
 0  0  0  0  0  0
 0  0  0  0  0  0
```

As we compared a subtree with one node with a subtree with two nodes and one matching label, the edit distance is one and is also entered in the *tdist* at position $(3, 4)$ array because both form proper subtrees.

The solutions for *tdist* can also be found at the lower right corner of the forest array. Other values in the forest array represent the distances between forests numbered from their respective positions plus the $l(i)$ and $l(j)$ values. Please note that the forest arrays contain one additional column and row because empty sets are also possible here.

```
Forestdist for 6 4          Treedist after 6 4
 0  1  2                     0  0  0  1  0  0
 1  0  1                     0  0  1  0  0  0
 2  1  0                     0  1  0  1  0  5
 3  2  1                     0  1  0  1  0  5
 4  3  2                     0  0  4  3  0  0
 5  4  3                     0  0  5  4  0  0
 6  5  4
```

This step of the algorithm matches the subtree with the blue and white labels (nodes 3 and 4 in T_2) correctly into the complete tree T_1 and therefore returns an editing distance of 4 which corresponds to a deletion of all other nodes. The array *tdist* has been filled up to a large degree in the intermediate steps. One can visualize the operation of the algorithm in the forest array. First, row 0 and column 0 are initialized with the costs for insertion of the respective nodes due to Lemma 3. Each interior value is computed from the values at the elements above, to the left, and above left plus the costs for the required operation or subtree.

```
Forestdist for 6 6               Treedist after 6 6
 0  1  2  3  4  5  6              0  1  0  1  0  5
 1  0  1  2  3  4  5              1  0  1  0  1  4
 2  1  0  1  2  3  4              0  1  0  1  0  5
 3  2  1  0  1  2  3              0  1  0  1  0  5
 4  3  2  1  2  1  2              4  3  4  3  4  2
 5  4  3  2  3  2  2              5  4  5  4  5  3
 6  5  4  3  4  3  3

Edit distance between trees is: 3
```

The final step is the calculation of the distance between the two roots. The values in the tree distance array give the distances between all possible pairs of subtrees. The value at the lower right corner is the final result. The resulting mapping was already presented in Figure 5 and the required edit operations are the deletion of node 5, insertion of node 4, and relabeling of node 6.

5 Complexity

To compute the time and space complexity of the algorithm presented in Section 4 several new definitions and Lemmata must be provided.

Lemma 6 $|LR\_keyroots(T)| \leq |leaves(T)|$

Proof. By showing that for any $i, j \in LR\_keyroots(T)$ and $i \neq j$, the sons of i and j are different, i.e. $l(i) \neq l(j)$ it is clear that there must be more leaves in the tree than elements in $LR\_keyroots$. Therefore, let $i, j \in LR\_keyroots(T)$ and without loss of generality $i < j$. Assume that $l(i) = l(j)$. Then i must be on the path from $l(j)$ to j because they have the same son and $i < j$. By the definition of $l(j)$, i has no left sibling, contradicting the assertion that $i \in LR\_keyroots(T)$. Each leaf is the leftmost descandant of at most one member of $LR\_keyroots$.

Definition. The collapsed depth of node i is defined as $LR\_colldepth(i) = |anc(i) \cap LR\_keyroots(T)|$ and the collapsed depth of tree T as $LR\_colldepth(T) = \max\{LR\_colldepth(i)\}$ and corresponds to the number of computations a node participates in. It is immediately clear from this definition and Lemma 6 that $LR\_colldepth(T) \leq \min(depth(T), leaves(T))$.

Lemma 7

$$\sum_{i \in LR\_keyroots(T)} Size(i) = \sum_{j=1}^{|T|} |LR\_colldepth(j)|$$

where $Size(i)$ is defined to be the number of nodes in $T[l(i)..i]$.

Proof. A particular node j is counted in the first summation whenever one of its ancestors is an element of $LR\_keyroots$. As this is exactly the definition of the collapsed depth, it is counted $LR\_colldepth(j)$ times.

Lemma 8 *The space complexity of the dynamic programming algorithm presented in Section 4 is $O(|T_1| \times |T_2|)$ and the time complexity is $O(|T_1| \times |T_2| \times \min(depth(T_1), leaves(T_1)) \times \min(depth(T_2), leaves(T_2)))$*

Proof. The space complexity of our algorithm is obvious because we use the two arrays $tdist$ and $fdist$, both of size $O(|T_1| \times |T_2|)$ and for other data structures linear or constant space is required. The analysis of time complexity is more sophisticated. Preprocessing (i.e. generating the $l()$ and $LR\_keyroots$ arrays) takes linear time in $|T_1|$ or $|T_2|$. To compute the subtree distance takes $Size(i) \times Size(j)$ and the main loop calls this subroutine for all members of $LR\_keyroots$. Solving this yields:

$$\begin{aligned}
\#\text{ops} &= \sum_{i \in LR_keyroots(T_1)} \sum_{j \in LR_keyroots(T_2)} Size(i) \times Size(j) \\
&= \sum_{i \in LR_keyroots(T_1)} Size(i) \times \sum_{j \in LR_keyroots(T_2)} Size(j) \\
&= \sum_{i=1}^{|T_1|} |LR_colldepth(i)| \times \sum_{j=1}^{|T_2|} |LR_colldepth(j)| \\
&\leq |T_1| \times |T_2| \times LR_colldepth(T_1) \times LR_colldepth(T_2) \\
&\quad \Box
\end{aligned}$$

With the inequality $LR\_colldepth(T) \leq \min(depth(T), leaves(T))$ the time complexity is given by

$$O(|T_1| \times |T_2| \times \min(depth(T_1), leaves(T_1)) \times \min(depth(T_2), leaves(T_2)))$$

6 Extensions

Often it is not only necessary to know the distance between the two trees, but it is also required to know how to transform one tree into the other one, i.e. the mapping must be explicitly constructed. This can be achieved by saving which configuration lead to the optimal value for all nodes in the *tdist* array and which operation was applied. Time and space complexity are not changed as the additional operations introduces only a constant factor.

Applying the tree distance algorithm to strings — i.e. degenerated trees — we observe that a string has only one leaf, $LR\_keyroots(S) = \{root\}$, and $l(i) = 1$. Therefore, only the first case of Lemma 5 is executed in our computation. This leads to a reduction of the tree algorithm to the general string algorithm with time complexity $O(|T_1| \times |T_2|)$, i.e. the string problem is contained as a special case in our algorithm.

In *approximate string matching* we compute the minimum number of edit operations between a pattern and a text where any prefix can be removed from the text. This means that the algorithm should find the occurrence in the text that most closely matches the pattern. When used with trees, two different modes of deletion can be applied

- Removing at node $T[i]$: remove subtree rooted at $T[i]$
- Pruning at node $T[i]$: remove all descendants of $T[i]$

The algorithms for these two problems follow the same algorithmic template as the normal tree edit algorithm from Section 4, i.e. they have the same main loop and a slightly modified subroutine for the forest distances which includes a fourth alternative for removing/pruning.

When we restrict the problem to unit costs, several optimizations can be applied. Namely, we can check if two trees are within a certain distance k and then increase this distance until we have the result with the following algorithm

```
for (j=0 ; j<=log2(|T1|+|T2|) ; j++)
  if ((sol = treedist(T1, T2, 2^j)) <= 2^j)
    return (sol);
```

To compute `treedist(T1, T2, k)` we use:

```
for (i=1 ; i<=|T1| ; i++)
  for (j=MAX(i-k, 1) ; j<=MIN(|T2|, i+k) ; j++)
    if (|left1[i]-left2[j]| <= k)
      _treedist(i, j, k-|left1[i] - left2[j]|);
```

The computation in `_treedist(i, j, k)` computes only *relevant* values and the running time of the whole algorithm is $O(k^2 \times \min(|T_1|, |T_2|) \times \min(L_1, L_2))$ where L is number of levels in the tree.

Further optimizations can be achieved when we check if subtrees are equal. This can be done in an efficient way by using suffix trees. The algorithm is constructed similar to the Landau-Vishkin string algorithm and is a parallel implementation with $k^2 \times N$ processors and a running time of $O((k^2 \times \log(k)) + \log(N))$.

7 Tree Alignment

Another metric to compare the similarity of trees is tree alignment. To align two trees we first insert empty nodes, called *spaces* into T_1 and T_2 such that the resulting trees T_1' and T_2' have the same structure. However, the labels at corresponding nodes can be different. Then we *overlay* T_1' and T_2' and assign scores for each pair of labels. The value of an alignment $\mathcal{A}$ is the sum of the scores. An *optimal* alignment minimizes this sum. Figure 9 depicts an example.

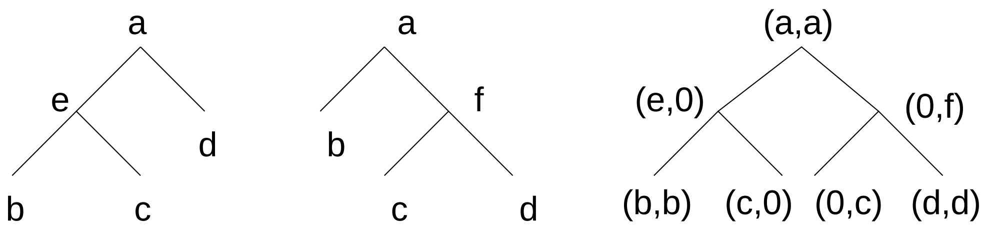

Figure 9: Alignment

The algorithm is very similar to the one we presented in Section 4 and also uses a dynamic programming approach, but runs in time $O(|T_1| \times |T_2| \times (deg(T_1) + deg(T_2))^2)$. $deg(T)$ denotes the degree of a tree, i.e. the maximum number of children of a single node.

Alignment and edit distance are the same for strings because the structure generated by the insertions and deletions of both methods are always the same. However, applied to trees both metrics produce different results with tree alignment seeming to charge more for operations at the top of the tree. The following inequality between the two metrics holds:

$$\text{tree edit distance} \;\leq\; \text{tree alignment}.$$

Naturally, the cost functions must be the same in both cases. This inequality is true because tree alignment is a restricted version of tree edit where insertions precede deletions.

8 Summary and Conclusion

The problem of the similarity of trees can be solved in different ways. The standard approach is tree edit distance with the operations modify, insert, and delete. The algorithm employs a dynamic programming approach and runs in polynomial time. Parallelizations and further optimizations are possible. Another interesting approach is tree alignment. This document, the slides which are the basis for this document, and the actual C source code of the tree edit distance algorithm can be found online at [1].

References

[1] C. Isert. Document and C source code. http://www.in.tum.de/~isert.

[2] T. Jiang, L. Wang, and K. Zhang. Alignment of trees — an alternative to tree edit. *Theor. Comput. Sci.*, 143(1):143–148, 1995.

[3] D. Shasha and K. Zhang. Fast algorithms for the unit cost editing distance between trees. *J. Algorithms*, 11:581–621, 1990.

[4] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18:1245–1262, 1989.

[5] M. Zuker. Lecture notes. http://www.ibc.wustl.edu/~zucker.