Similarity Search

The Tree Edit Distance

Nikolaus Augsten
nikolaus.augsten@plus.ac.at
Department of Computer Science
University of Salzburg

https://dbresearch.uni-salzburg.at

WS 2025/26

Version November 6, 2025

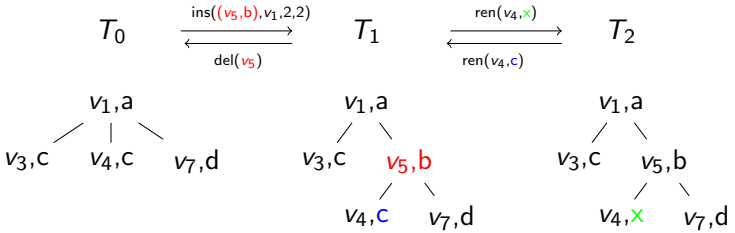
Outline

Outline

Edit Operations

- We assume ordered, labeled trees
- Rename node: $ren(v, l')$
  - change label l of v to $l' \neq l$
- Delete node: $del(v)$ (v is not the root node)
  - remove v
  - connect v's children directly to v's parent node (preserving order)
- Insert node: $ins(v, p, k, m)$
  - remove m consecutive children of p, starting with the child at position k, i.e., the children $c_k, c_{k+1}, \ldots, c_{k+m-1}$
  - insert $c_k, c_{k+1}, \ldots, c_{k+m-1}$ as children of the new node v (preserving order)
  - insert new node v as k-th child of p
- Insert and delete are inverse edit operations (i.e., insert undoes delete and vice versa)

Example: Edit Operations



Edit Cost Function

- Represent edit operation as node pair $(a, b) \neq (\varepsilon, \varepsilon)$ (written also as $a \to b$, ε is the null node)
  - $a \to \varepsilon$: delete a
  - $\varepsilon \to b$: insert b
  - $a \to b$: rename a to b
- Cost function $\alpha(a \to b)$:
  - assign to each edit operation a non-negative real
  - cost can be different for different nodes
  - we use constant costs $\omega_{ins}, \omega_{del}, \omega_{ren}$
- We constrain α to be a distance metric:
  - (i) triangle inequality: $\alpha(a \to b) + \alpha(b \to c) \geq \alpha(a, c)$
  - (ii) symmetry: $\alpha(a \to b) = \alpha(b \to a)$
  - (iii) identity: $\alpha(a \to b) = 0 \Leftrightarrow \lambda(a) = \lambda(b)$

Definition

Definition (Tree Edit Distance)

The tree edit distance between two trees is the minimum cost sequence of node edit operations (node deletion, node insertion, node rename) that transforms one tree into the other.

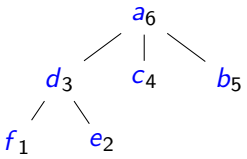
- Cost of a sequence $S = \{s_1, \ldots, s_n\}$ of edit operations:

$$\alpha(S) = \sum_{i=1}^{n} \alpha(s_i)$$

- As the cost function is a metric, also the tree edit distance is a metric.

Postorder Traversal

- Postorder traversal of an ordered tree:
  - traverse subtrees rooted in children of current node (from left to right) in postorder
  - visit current node
- Example: postorder = (f, e, d, c, b, a)



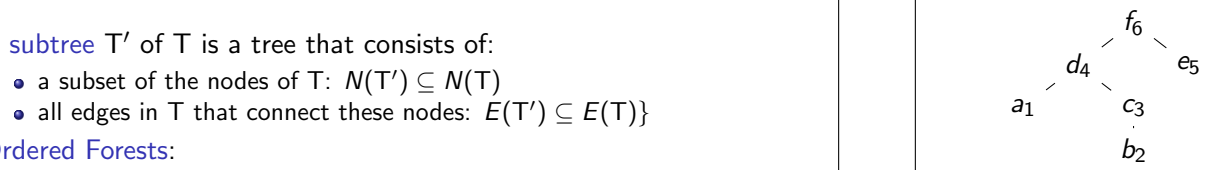
- Observations: The postorder number of a node v is larger than
  - the postorder numbers of all its descendants (excluding node v)
  - the postorder numbers of all its left siblings

Subtrees and Subforests

- A subtree T' of T is a tree that consists of:
  - a subset of the nodes of T: $N(T') \subseteq N(T)$
  - all edges in T that connect these nodes: $E(T') \subseteq E(T)\}$
- Ordered Forests:
  - a forest is a set of trees
  - an *ordered* forest is a sequence of trees
- Ordered Subforests of a tree T:
  - formed by subtrees of T with disjoined nodes
  - subtrees ordered by the postorder number in T of their root

Example: Subtrees and Subforests

- Example tree (postorder numbers are node IDs):
  $T = (\{v_1, v_2, v_3, v_4, v_5, v_6\}, \{(v_6, v_4), (v_6, v_5), (v_4, v_1), (v_4, v_3), (v_3, v_2)\})$



- Two subtrees of T:

$T'_1 = (\{v_3\}, \{\})$ c_3

$T'_2 = (\{v_4, v_1, v_3\}, \{(v_4, v_1), (v_4, v_3)\})$

- Ordered subforest of T:
  $F = ((\{v_2\}, \{\}), (\{v_4, v_1, v_3\}, \{(v_4, v_1), (v_4, v_3)\}), (\{v_5\}, \{\}))$

Notation I/II

- We use the following notation:
  - $T[i]$ is the i-th node of T in postorder (we say: $T[i]$ is node i of T)
  - $T[i..j]$ is the subforest formed by the nodes $T[i]$ to $T[j]$
  - $l(i)$ is the left-most leaf descendant of node $T[i]$
  - $desc(T[i])$ is the set of all descendants of $T[i]$ including $T[i]$ itself (elements of $desc(T[i])$ are usually denoted with d_i)
- Node identifiers:
  - we assume that the node IDs correspond to their postorder number
  - we refer to a node simply by its ID, if the context is clear

Notation II/II

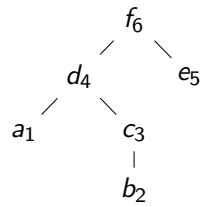
- $T[l(i)..i]$ is the subtree rooted in $T[i]$, i.e., the subtree consisting of node i and all its descendants
- A special subforests of the form

$$T[l(i)..d_i], \qquad (d_i \in desc(T[i]))$$

  is a prefix of the subtree rooted in $T[i]$.
- Observations:
  - If a node k is in $T[l(i)..d_i]$, also all its descendants are in $T[l(i)..d_i]$.
  - A (sub)tree with n nodes has n prefixes.

Example: Subtrees and Subforests

- Example tree:



- Descendants: $desc(\mathrm{T}[4]) = \{\mathrm{T}[1], \mathrm{T}[2], \mathrm{T}[3], \mathrm{T}[4]\}$
- Left-most leaf descendants: $l(1) = l(4) = l(6) = \mathrm{T}[1]$
- Some ordered subforests of the form $\mathrm{T}[l(i)..\mathrm{d}_i]$, $\mathrm{d}_i \in desc(i)$:

$\mathrm{T}[l(4)..3]$	$\mathrm{T}[l(4)..4]$	$\mathrm{T}[l(6)..5]$	$\mathrm{T}[l(5)..5]$
a_1 c_3	d_4	d_4 e_5	e_5
b_2	a_1 c_3	a_1 c_3	
	b_2	b_2	

Edit Mapping

Definition (Edit Mapping)

An edit mapping M between T_1 and T_2 is a set of node pairs that satisfy the following conditions:

(1) $(\mathrm{a}, \mathrm{b}) \in M \Rightarrow \mathrm{a} \in N(\mathrm{T}_1), \mathrm{b} \in N(\mathrm{T}_2)$

(2) for any two pairs (a, b) and (x, y) of M:

- (i) $\mathrm{a} = \mathrm{x} \Leftrightarrow \mathrm{b} = \mathrm{y}$ (one-to-one condition)
- (ii) a is to the left of x[1] $\Leftrightarrow$ b is to the left of y (order condition)
- (iii) a is an ancestor of x $\Leftrightarrow$ b is an ancestor of y (ancestor condition)

[1] i.e., a precedes x in both preorder and postorder

Edit Mapping

- The cost of the mapping is

$$\alpha(M) = \sum_{(a,b)\in M} \alpha(\mathrm{a} \to \mathrm{b}) + \sum_{a\in D} \alpha(\mathrm{a} \to \varepsilon) + \sum_{b\in I} \alpha(\varepsilon \to \mathrm{b}),$$

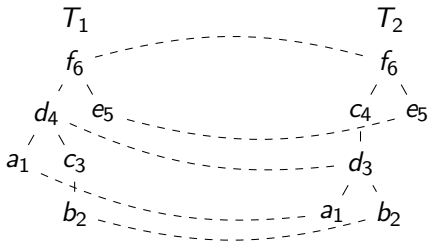
where D resp. I are the nodes of T_1 resp. T_2 that are not in M.

- Alternative definition of the tree edit distance $ted(\mathrm{T}_1, \mathrm{T}_2)$:

$$ted(\mathrm{T}_1, \mathrm{T}_2) = \min\{\alpha(M) \mid M \text{ is an edit mapping from } \mathrm{T}_1 \text{ to } \mathrm{T}_2\}$$

Example: Mapping

- $M = \{(\mathrm{T}_1[6], \mathrm{T}_2[6]), (\mathrm{T}_1[5], \mathrm{T}_2[5]), (\mathrm{T}_1[4], \mathrm{T}_2[3]), (\mathrm{T}_1[1], \mathrm{T}_2[1]), (\mathrm{T}_1[2], \mathrm{T}_2[2])\}$
  - $\mathrm{T}_1[3]$ is deleted
  - $\mathrm{T}_2[4]$ is inserted
  - no proper rename (only rename to the same label with cost 0)



Outline

Forest Distance

Definition (Forest Distance)

The forest distance between two ordered forests is the minimum cost sequence of node edit operations (node deletion, node insertion, node rename) that transforms one forest into the other.

- Edit mapping and edit operations in a forest:
  - Each tree in the forest has a root node.
  - We imagine a dummy node that is the parent of all these root nodes.
  - The sibling order in the imaginary tree is the tree order in the forest.
  - The dummy node connects the forest to become a tree.
  - Then all edit operations and edit mappings valid between two imaginary trees are valid also between the respective forests.
- The tree edit distance is a special case of the forest distance, where the forest has the form $T[l(i)..i]$, i.e., it consists of a single tree.

Recursive Formula: Distance to the Empty Forest

Lemma (Empty Forest [ZS89, AG97])

Given two trees T_1 *and* T_2, $i \in N(T_1)$ *and* $d_i \in desc(i)$, $j \in N(T_2)$ *and* $d_j \in desc(j)$, *then:*

$$
\begin{array}{ll}
(i) & fdist(\emptyset, \emptyset) = 0 \\
(ii) & fdist(T_1[l(i)..d_i], \emptyset) = fdist(T_1[l(i)..d_i - 1], \emptyset) + \omega_{del} \\
(iii) & fdist(\emptyset, T_2[l(j)..d_j]) = fdist(\emptyset, T_2[l(j)..d_j - 1]) + \omega_{ins}
\end{array}
$$

Proof.

Case (i) requires no edit operation. In cases (ii), the distance corresponds to the cost of deleting all nodes in $T_1[l(i)..d_i]$. In cases (iii), the distance corresponds to the cost of inserting all nodes in $T_2[l(j)..d_j]$. □

First Recursive Formula: Forest Distance

Lemma (First Recursive Formula)

Given two trees T_1 *and* T_2, $i \in N(T_1)$ *and* $d_i \in desc(i)$, $j \in N(T_2)$ *and* $d_j \in desc(j)$, *then:*

$$
fdist(T_1[l(i)..d_i], T_2[l(j)..d_j]) = \min \begin{cases}
fdist(T_1[l(i)..d_i - 1], T_2[l(j)..d_j]) + \omega_{del} \\
fdist(T_1[l(i)..d_i], T_2[l(j)..d_j - 1]) + \omega_{ins} \\
fdist(T_1[l(i)..l(d_i) - 1], T_2[l(j)..l(d_j) - 1]) \\
\quad + fdist(T_1[l(d_i)..d_i - 1], T_2[l(d_j)..d_j - 1]) \\
\quad + \omega_{ren}
\end{cases}
$$

Proof

Proof.

Let M be the minimum-cost map between $T_1[l(i)..d_i]$ and $T_2[l(j)..d_j]$, i.e., the map we are looking for. Then for $T_1[d_i]$ and $T_2[d_j]$ there are three possibilities:

(1) $T_1[d_i]$ is not touched by a line in M: $T_1[d_i]$ is deleted and $fdist(T_1[l(i)..d_i], T_2[l(j)..d_j]) = fdist(T_1[l(i)..d_i - 1], T_2[l(j)..d_j]) + \omega_{del}$

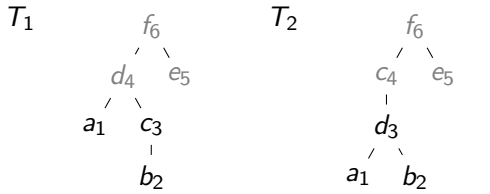
(2) $T_2[d_j]$ is not touched by a line in M: $T_2[d_j]$ is inserted and $fdist(T_1[l(i)..d_i], T_2[l(j)..d_j]) = fdist(T_1[l(i)..d_i], T_2[l(j)..d_j - 1]) + \omega_{ins}$

(3) Both, $T_1[d_i]$ and $T_2[d_j]$ are touched by a line in M: We show (by contradiction) that in this case $(T_1[d_i], T_2[d_j]) \in M$, i.e., $T_1[d_i]$ is renamed to $T_2[d_j]$: Assume $(T_1[d_i], T_2[d'_i]) \in M$ and $(T_1[d'_j], T_2[d_j]) \in M$.

- Case $T_1[d_i]$ is to the right of $T_1[d'_j]$: By sibling condition on M also $T_2[d'_i]$ must be to the right of $T_2[d_j]$. Impossible in $T_2[l(j)..d_j]$.
- Case $T_1[d_i]$ is proper ancestor of $T_1[d'_j]$: By ancestor condition on M also $T_2[d'_i]$ must be ancestor of $T_2[d_j]$. Impossible in $T_2[l(j)..d_j]$.

As these three cases express all possible mappings yielding $fdist(T_1[l(i)..d_i], T_2[l(j)..d_j])$, we take the minimum of these tree costs. □

Example: First Recursive Formula (1/3)



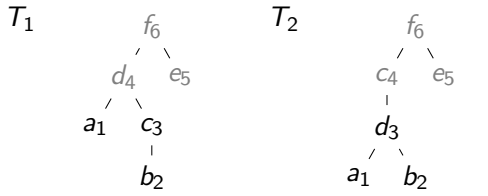
$T_1[l(i)...d_i]$ (i=6, d_i=3) $T_2[l(j)...d_j]$ (j=6, d_j=3)

(1) $fdist(T_1[l(i)..d_i - 1], T_2[l(j)..d_j]) + \omega_{del}$

$T1[l(i)...d_i - 1]$ $T_2[l(j)...d_j]$

- edit script: $ins(d_3), del(c_3)$
- cost: $1 + 1 = 2$

Example: First Recursive Formula (2/3)



$T_1[l(i)...d_i]$ (i=6, d_i=3) $T_2[l(j)...d_j]$ (j=6, d_j=3)

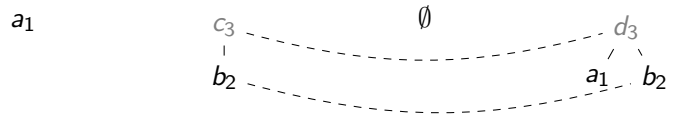
(2) $fdist(T_1[l(i)..d_i], T_2[l(j)..d_j - 1]) + \omega_{ins}$

$T1[l(i)...d_i - 1]$ $T_2[l(j)...d_j]$

- edit script: $del(c_3), ins(d_3)$
- cost: $1 + 1 = 2$

Example: First Recursive Formula (3/3)

(3) $fdist(T_1[l(i)..l(d_i) - 1], T_2[l(j)..l(d_j) - 1])$
$+ fdist(T_1[l(d_i)..d_i - 1], T_2[l(d_j)..d_j - 1])$
$+ \omega_{ren}$



$T_1[l(i)..l(d_i) - 1]$ $T_1[l(d_i)..d_i - 1]$ $T_2[l(j)..l(d_j) - 1]$ $T_2[l(d_j)..d_j - 1]$

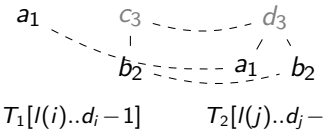
- $T_1[l(i)..l(d_i) - 1] \to T_2[l(j)..l(d_j) - 1]$: $del(a_1)$
- $T_1[l(d_i)..d_i - 1] \to T_2[l(d_j)..d_j - 1]$: $ins(a_1)$
- $c_3 \to d_3$: $ren(c_3, d_3)$
- cost: $1 + 1 + 1 = 3$

Analogy to the String Case

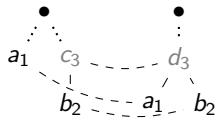
- Why is the third formula not (in analogy to the string case):

$$fdist(T_1[l(i)..d_i - 1], T_2[l(j)..d_j - 1]) + \omega_{ren}$$

- Consider the previous example:



- $ren(c_3, d_3)$ does not transform $T_1[l(i)..d_i]$ to $T_2[l(j)..d_j]$
- In fact the mapping $M = \{(a_1, a_1), (b_2, b_2), (c_3, d_3)\}$ is not valid:
  - Connect all trees in the forest with a dummy node ($\bullet$):
  - As d_3 is an ancestor of a_1, c_3 must be an ancestor of a_1, which is false.



Outline

Observation

$$fdist(T_1[l(i)..d_i], T_2[l(j)..d_j]) = \min \begin{cases} fdist(T_1[l(i)..d_i{-}1], T_2[l(j)..d_j]) + \omega_{del} \\ fdist(T_1[l(i)..d_i], T_2[l(j)..d_j{-}1]) + \omega_{ins} \\ fdist(T_1[l(i)..l(d_i) - 1], T_2[l(j)..l(d_j) - 1]) \\ \quad + fdist(T_1[l(d_i)..d_i{-}1], T_2[l(d_j)..d_j{-}1]) \\ \quad + \omega_{ren} \end{cases}$$

- Observation about the First Recursive Formula:
  - $fdist(T_1[l(d_i)..d_i{-}1], T_2[l(d_j)..d_j{-}1])$ *[D]* compares prefixes of subtrees rooted in d_i resp. d_j
  - all other subforests are prefixes of subtrees rooted in i resp. j
  - *[D]* does not fit the scheme (bad for dynamic programming algorithm)
- We derive the Second Recursive Formula:
  - we distinguish two cases (both forests are trees/one forest is not a tree)
  - in each case we replace term *[D]* by a new term that is easier to handle in a dynamic programming algorithm

Second Recursive Formula: Forest Distance

Lemma (Second Recursive Formula)

Given two trees T_1 and T_2, $i \in N(T_1)$ and $d_i \in desc(i)$, $j \in N(T_2)$ and $d_j \in desc(j)$, then:

(1) *If $l(i) = l(d_i)$ and $l(j) = l(d_j)$, i.e., both forests are trees:*

$$fdist(T_1[l(i)..d_i], T_2[l(j)..d_j]) = \min \begin{cases} fdist(T_1[l(i)..d_i - 1], T_2[l(j)..d_j]) + \omega_{del} \\ fdist(T_1[l(i)..d_i], T_2[l(j)..d_j - 1]) + \omega_{ins} \\ fdist(T_1[l(i)..d_i - 1], T_2[l(j)..d_j - 1]) + \omega_{ren} \end{cases}$$

(2) *If $l(i) \neq l(d_i)$ and/or $l(j) \neq l(d_j)$, i.e., one of the forests is not a tree:*

$$fdist(T_1[l(i)..d_i], T_2[l(j)..d_j]) = \min \begin{cases} fdist(T_1[l(i)..d_i - 1], T_2[l(j)..d_j]) + \omega_{del} \\ fdist(T_1[l(i)..d_i], T_2[l(j)..d_j - 1]) + \omega_{ins} \\ fdist(T_1[l(i)..l(d_i) - 1], T_2[l(j)..l(d_j) - 1]) \\ \quad + fdist(T_1[l(d_i)..d_i], T_2[l(d_j)..d_j]) \end{cases}$$

Proof of the Second Recursive Formula

Proof.

(1) follows from the previous recursive formula for $l(i) = l(\mathrm{d}_i)$ and $l(j) = l(\mathrm{d}_j)$ as the following holds:

$$fdist(\mathrm{T}_1[l(i)..l(\mathrm{d}_i)-1], \mathrm{T}_2[l(j)..l(\mathrm{d}_j)-1]) = fdist(\emptyset, \emptyset) = 0.$$

(2) The following inequation holds:

$$\begin{aligned} [A] \quad fdist(\mathrm{T}_1[l(i)..\mathrm{d}_i], \mathrm{T}_2[l(j)..\mathrm{d}_j]) &\leq fdist(\mathrm{T}_1[l(i)..l(\mathrm{d}_i)-1], \mathrm{T}_2[l(j)..l(\mathrm{d}_j)-1]) && [B] \\ &\quad + fdist(\mathrm{T}_1[l(\mathrm{d}_i)..\mathrm{d}_i], \mathrm{T}_2[l(\mathrm{d}_j)..\mathrm{d}_j]) && [C] \\ &\leq fdist(\mathrm{T}_1[l(i)..l(\mathrm{d}_i)-1], \mathrm{T}_2[l(j)..l(\mathrm{d}_j)-1]) && [B] \\ &\quad + fdist(\mathrm{T}_1[l(\mathrm{d}_i)..\mathrm{d}_i-1], \mathrm{T}_2[l(\mathrm{d}_j)..\mathrm{d}_j-1]) && [D] \\ &\quad + \omega_{ren} \end{aligned}$$

$A \leq B + C$ as the left-hand side is the *minimal* cost mapping, while the right-hand side is a particular case with a possibly sub-optimal mapping. $C \leq D + \omega_{ren}$ holds for the same reason.

As we are looking for the *minimum* distance, we can substitute $D + \omega_{ren}$ by C.

□

Illustration: Proof of the Second Recursive Formula (1/2)

- Case (1): $l(i) = l(\mathrm{d}_i)$ and $l(j) = l(\mathrm{d}_j)$:

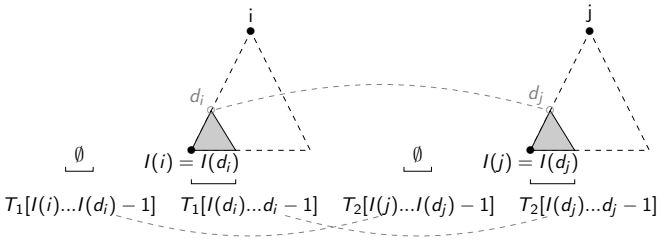
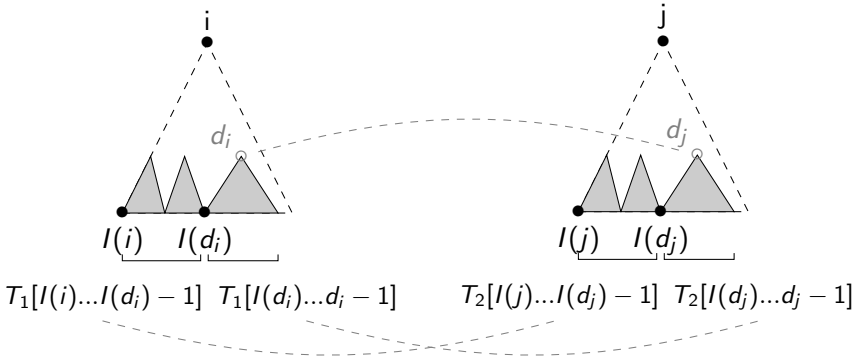


Illustration: Proof of the Second Recursive Formula (2/2)

- Case (2): $l(i) \neq l(\mathrm{d}_i)$ and/or $l(j) \neq l(\mathrm{d}_j)$:



Implications by the Second Recursive Formula

- Note: $fdist(\mathrm{T}_1[l(\mathrm{d}_i)..\mathrm{d}_i], \mathrm{T}_2[l(\mathrm{d}_j)..\mathrm{d}_j]$ is the tree edit distance between the subtrees rooted in $\mathrm{T}[\mathrm{d}_i]$ and $\mathrm{T}[\mathrm{d}_j]$. We use the following notation:

$$treedist(\mathrm{d}_i, \mathrm{d}_j) = fdist(\mathrm{T}_1[l(\mathrm{d}_i)..\mathrm{d}_i], \mathrm{T}_2[l(\mathrm{d}_j)..\mathrm{d}_j])$$

- Dynamic Programming: As the same sub-problem must be solved many times, we use a dynamic programming approach.
- Bottom-Up: As for the computation of the tree distance $treedist(i, j)$ we need almost all values $treedist(\mathrm{d}_i, \mathrm{d}_j)$ ($\mathrm{d}_i \in desc(\mathrm{T}_1[i])$, $\mathrm{d}_j \in desc(\mathrm{T}_1[j])$), we use a bottom-up approach.
- Key Roots: If
  - d_i is on the path from $l(i)$ to $\mathrm{T}_1[i]$ and
  - d_j is on the path from $l(j)$ to $\mathrm{T}_2[j]$,

  then $treedist(\mathrm{d}_i, \mathrm{d}_j)$ is computed as a byproduct of $treedist(i, j)$. We call the nodes that are *not* computed as a byproducts the key roots.

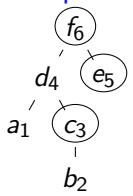
Key Roots

Definition (Key Root)

The set of *key roots* of a tree T is defined as

$$kr(T) = \{k \in N(T) \mid \nexists k' \in N(T) : k' > k \text{ and } l(k) = l(k')\}$$

- Alternative definition: A *key root* is a node of T that either has a left sibling or is the root of T.
- Example: $kr(T) = \{3, 5, 6\}$



- Only subtrees rooted in a key root need a separate computation.
- The number of key roots is equal to the number of leaves in the tree.

Outline

The Edit Distance Algorithm I/II

tree-edit-dist(T_1, T_2)

$td[1..|T_1|, 1..|T_2|]$: empty array for tree distances;
$l_1 = \text{lmld}(root(T_1))$; $kr_1 = \text{kr}(l_1, |leaves(T_1)|)$;
$l_2 = \text{lmld}(root(T_2))$; $kr_2 = \text{kr}(l_2, |leaves(T_2)|)$;
for $x = 1$ to $|kr_1|$ **do**
  for $y = 1$ to $|kr_2|$ **do**
    forest-dist($kr_1[x], kr_2[y], l_1, l_2, td$);

- l_1 is an array of size $|T_1|$, $l_1[i]$ is the leftmost leaf descendant of node i; l_2 is the analog for T_2 (detailed algorithm for lmld(.) follows)
- kr_1 is an array that contains all the key roots of T_1 sorted in ascending order; kr_2 is the analog for T_2 (detailed algorithm kr(.,.) follows)
- Algorithm and lemmas by [ZS89] (see also [AG97])

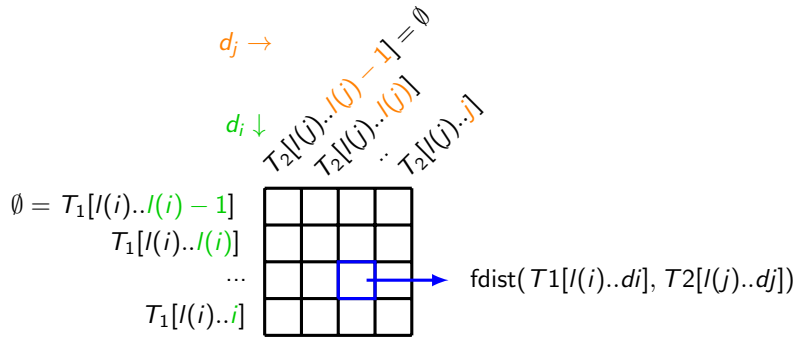
The Edit Distance Algorithm II/II

forest-dist(i, j, l_1, l_2, td)

$fd[l_1[i]-1..i, l_2[j]-1..j]$: empty array;
$fd[l_1[i]-1, l_2[j]-1] = 0$;
for $d_i = l_1[i]$ **to** i **do** $fd[d_i, l_2[j]-1] = fd[d_i-1, l_2[j]-1] + \omega_{del}$;
for $d_j = l_2[j]$ **to** j **do** $fd[l_1[i]-1, d_j] = fd[l_1[i]-1, d_j-1] + \omega_{ins}$;
for $d_i = l_1[i]$ **to** i **do**
  for $d_j = l_2[j]$ **to** j **do**
    if $l_1[d_i] = l_1[i]$ **and** $l_2[d_j] = l_2[j]$ **then**
      $fd[d_i, d_j] = \min(fd[d_i-1, d_j] + \omega_{del},$
        $fd[d_i, d_j-1] + \omega_{ins},$
        $fd[d_i-1, d_j-1] + \omega_{ren})$;
      $td[d_i, d_j] = f[d_i, d_j]$;
    else $fd[d_i, d_j] = \min(fd[d_i-1, d_j] + \omega_{del},$
        $fd[d_i, d_j-1] + \omega_{ins},$
        $fd[l_1[d_i]-1, l_2[d_j]-1] + td[d_i, d_j])$;

The Temporary Forest Distance Matrix

- $fd[d_i, d_j]$ contains the forest distance between
  - $T_1[l(i)..d_i]$, where $d_i \in desc(T_1[i])$ and
  - $T_2[l(j)..d_j]$, where $d_j \in desc(T_2[j])$.



- fd is temporary and exists only in forest-dist()

The Tree Distance Matrix

- $td[i, j]$ stores the tree edit distance between
  - the tree rooted in $T_1[i]$ (*i.e.*, $T_1[l(i)..i]$) and
  - the tree rooted in $T_2[j]$ (*i.e.*, $T_2[l(j)..j]$).
- each call of forest-dist() fills new values into td
- $td[|T_1|, |T_2|]$ stores the tree edit distance between T_1 and T_2

Computing Key Roots and Left-Most Leaf Descendants

The tree edit distance algorithm uses the following functions:

- lmld(i): computes an array with the left-most leaf descendants of all descendants of a node i
- kr(l, lc): given the array $l = \text{lmld}(i)$ of left-most leaf descendants, and the number lc of leaf descendants of i, compute all key roots of the subtree rooted in i

tree-edit-dist(T_1, T_2)

$td[1..|T_1|, 1..|T_2|]$: empty array for tree distances;
$l_1 = \text{lmld}(root(T_1))$; $kr_1 = \text{kr}(l_1, |leaves(T_1)|)$;
$l_2 = \text{lmld}(root(T_2))$; $kr_2 = \text{kr}(l_2, |leaves(T_2)|)$;
for $x = 1$ to $|kr_1|$ **do**
  for $y = 1$ to $|kr_2|$ **do**
    forest-dist($kr_1[x], kr_2[y], l_1, l_2, td$);

Computing the Left-Most Leaf Descendants

lmld(v, l)

foreach child c of v (left to right) **do** $l \leftarrow$ lmld(c, l);
if v is a leaf **then**
  l[id(v)] $\leftarrow$ id(v)
else
  $c_1 \leftarrow$ first child of v;
  l[id(v)] $\leftarrow$ l[id(c_1)];
return l;

- Input: root node v of a tree T, empty array $l[1..|T|]$
- Output: array l, $l[i]$ is the left-most leaf descendent of node T[i]
- lmld($root$(T)) (see tree-edit-dist(.,.)) is implemented as lmld($root$(T), l) with an empty array $l[1..|T|]$.

Computing the Key Roots

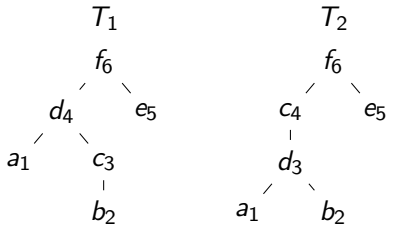
kr(l, lc)

```
kr[1..lc]: empty array;
visited[]: boolean array of size |l|, init with false;
k ← |kr|;   i ← |l|;
while k ≥ 1 do
    if not visited[l[i]] then
        kr[k--] ← i;
        visited[l[i]] ← true;
    i--;
return kr;
```

- Input:
  - l[1..|T|]: $l[i]$ is the left-most leaf descendent of node T[i]
  - $lc = |leaves(T)|$ is the number of leaves in T
- Output: array kr[1..$|leaves(T)|$] with key roots sorted by node ID
- Note: Loop condition is correct due to $k \geq 1 \Rightarrow i \geq 1$
  (the number of key roots is exactly the number of leaves, and kr will always be filled when all nodes are traversed)

Outline

Example Trees and Edit Costs



- Example: Edit distance between T_1 and T_2.
  - $\omega_{ins} = \omega_{del} = 1$
  - $\omega_{ren} = 0$ for identical rename, otherwise $\omega_{ren} = 1$
- Each of the following slide is the result of a call of forest-dist().

Executing the Algorithm (1/9)

	1	2	3	4	5	6
l_1	1	2	2	1	5	1

	1	2	3
kr_1	3	5	6

	1	2	3	4	5	6
l_2	1	2	1	1	5	1

	1	2	3
kr_2	2	5	6

- $i = kr_1[x] = 3 \Rightarrow l_1[i] = 2$
- $j = kr_2[y] = 2 \Rightarrow l_2[j] = 2$
- temporary array fd:

$d_i \downarrow$ \ $d_j \rightarrow 2$		
	0	1
2	1	0
3	2	1

- permanent array td:

	1	2	3	4	5	6
1						
2		0				
3		1				
4						
5						
6						

$l_1[i] = l_1[d_i]$ and $l_2[j] = l_2[d_j]$

Executing the Algorithm (2/9)

	1	2	3	4	5	6
l_1	1	2	2	1	5	1

	1	2	3
kr_1	3	5	6

	1	2	3	4	5	6
l_2	1	2	1	1	5	1

	1	2	3
kr_2	2	5	6

- $i = kr_1[x] = 3 \Rightarrow l_1[i] = 2$
- $j = kr_2[y] = 5 \Rightarrow l_2[j] = 5$
- temporary array fd:

$d_i \downarrow$ / $d_j \rightarrow 5$	0	1
2	1	1
3	2	2

- permanent array td:

	1	2	3	4	5	6
1						
2		0			1	
3		1			2	
4						
5						
6						

$l_1[i] = l_1[d_i]$ and $l_2[j] = l_2[d_j]$

Executing the Algorithm (3/9)

	1	2	3	4	5	6
l_1	1	2	2	1	5	1

	1	2	3
kr_1	3	5	6

	1	2	3	4	5	6
l_2	1	2	1	1	5	1

	1	2	3
kr_2	2	5	6

- $i = kr_1[x] = 3 \Rightarrow l_1[i] = 2$
- $j = kr_2[y] = 6 \Rightarrow l_2[j] = 1$
- temporary array fd:

$d_i \downarrow$ / $d_j \rightarrow 1$	0	1	2	3	4	5	6
2	1	1	1	2	3	4	5
3	2	2	2	2	2	3	4

- permanent array td:

	1	2	3	4	5	6
1						
2	1	0	2	3	1	5
3	2	1	2	2	2	4
4						
5						
6						

$l_1[i] = l_1[d_i]$ and $l_2[j] = l_2[d_j]$

Executing the Algorithm (4/9)

	1	2	3	4	5	6
l_1	1	2	2	1	5	1

	1	2	3
kr_1	3	5	6

	1	2	3	4	5	6
l_2	1	2	1	1	5	1

	1	2	3
kr_2	2	5	6

- $i = kr_1[x] = 5 \Rightarrow l_1[i] = 5$
- $j = kr_2[y] = 2 \Rightarrow l_2[j] = 2$
- temporary array fd:

$d_i \downarrow$ / $d_j \rightarrow 2$	0	1
5	1	1

- permanent array td:

	1	2	3	4	5	6
1						
2	1	0	2	3	1	5
3	2	1	2	2	2	4
4						
5		1				
6						

$l_1[i] = l_1[d_i]$ and $l_2[j] = l_2[d_j]$

Executing the Algorithm (5/9)

	1	2	3	4	5	6
l_1	1	2	2	1	5	1

	1	2	3
kr_1	3	5	6

	1	2	3	4	5	6
l_2	1	2	1	1	5	1

	1	2	3
kr_2	2	5	6

- $i = kr_1[x] = 5 \Rightarrow l_1[i] = 5$
- $j = kr_2[y] = 5 \Rightarrow l_2[j] = 5$
- temporary array fd:

$d_i \downarrow$ / $d_j \rightarrow 5$	0	1
5	1	0

- permanent array td:

	1	2	3	4	5	6
1						
2	1	0	2	3	1	5
3	2	1	2	2	2	4
4						
5		1			0	
6						

$l_1[i] = l_1[d_i]$ and $l_2[j] = l_2[d_j]$

Executing the Algorithm (6/9)

	1	2	3	4	5	6
l_1	1	2	2	1	5	1

	1	2	3
kr_1	3	5	6

	1	2	3	4	5	6
l_2	1	2	1	1	5	1

	1	2	3
kr_2	2	5	6

- $i = kr_1[x] = 5 \Rightarrow l_1[i] = 5$
- $j = kr_2[y] = 6 \Rightarrow l_2[j] = 1$
- temporary array fd:

$d_i \downarrow$ / $d_j \rightarrow 1$	1	2	3	4	5	6
0	1	2	3	4	5	6
5	1	2	3	4	4	5

- permanent array td:

	1	2	3	4	5	6
1						
2	1	0	2	3	1	5
3	2	1	2	2	2	4
4						
5	1	1	3	4	0	5
6						

$l_1[i] = l_1[d_i]$ and $l_2[j] = l_2[d_j]$

Executing the Algorithm (7/9)

	1	2	3	4	5	6
l_1	1	2	2	1	5	1

	1	2	3
kr_1	3	5	6

	1	2	3	4	5	6
l_2	1	2	1	1	5	1

	1	2	3
kr_2	2	5	6

- $i = kr_1[x] = 6 \Rightarrow l_1[i] = 1$
- $j = kr_2[y] = 2 \Rightarrow l_2[j] = 2$
- temporary array fd:

$d_i \downarrow$ / $d_j \rightarrow 2$	0	1
1	1	1
2	2	1
3	3	2
4	4	3
5	5	4
6	6	5

- permanent array td:

	1	2	3	4	5	6
1		1				
2	1	0	2	3	1	5
3	2	1	2	2	2	4
4		3				
5	1	1	3	4	0	5
6		5				

$l_1[i] = l_1[d_i]$ and $l_2[j] = l_2[d_j]$

Executing the Algorithm (8/9)

	1	2	3	4	5	6
l_1	1	2	2	1	5	1

	1	2	3
kr_1	3	5	6

	1	2	3	4	5	6
l_2	1	2	1	1	5	1

	1	2	3
kr_2	2	5	6

- $i = kr_1[x] = 6 \Rightarrow l_1[i] = 1$
- $j = kr_2[y] = 5 \Rightarrow l_2[j] = 5$
- temporary array fd:

$d_i \downarrow$ / $d_j \rightarrow 5$	0	1
1	1	1
2	2	2
3	3	3
4	4	4
5	5	4
6	6	5

- permanent array td:

	1	2	3	4	5	6
1		1			1	
2	1	0	2	3	1	5
3	2	1	2	2	2	4
4		3			4	
5	1	1	3	4	0	5
6		5			5	

$l_1[i] = l_1[d_i]$ and $l_2[j] = l_2[d_j]$

Executing the Algorithm (9/9)

	1	2	3	4	5	6
l_1	1	2	2	1	5	1

	1	2	3
kr_1	3	5	6

	1	2	3	4	5	6
l_2	1	2	1	1	5	1

	1	2	3
kr_2	2	5	6

- $i = kr_1[x] = 6 \Rightarrow l_1[i] = 1$
- $j = kr_2[y] = 6 \Rightarrow l_2[j] = 1$
- temporary array fd:

$d_i \downarrow$ / $d_j \rightarrow 1$	1	2	3	4	5	6
0	1	2	3	4	5	6
1	0	1	2	3	4	5
2	1	0	1	2	3	4
3	2	1	2	3	4	5
4	3	2	1	2	3	4
5	4	3	2	3	2	3
6	5	4	3	3	3	2

- permanent array td:

	1	2	3	4	5	6
1	0	1	2	3	1	5
2	1	0	2	3	1	5
3	2	1	2	2	2	4
4	3	3	1	2	4	4
5	1	1	3	4	0	5
6	5	5	3	3	5	2

$l_1[i] = l_1[d_i]$ and $l_2[j] = l_2[d_j]$

Outline

Notation

- Notation:
  - $|\mathrm{T}|$ is the number of nodes in T
  - $depth(\mathrm{v})$ is the number of ancestors of v (including v itself)
  - $depth(\mathrm{T})$ is the maximum depth of a node in T
  - $leaves(\mathrm{T})$ is the set of leaves of T
  - $t(i)$ is the subtree rooted in node i

forest-dist: Time Complexity

forest-dist(i, j, l_1, l_2, td)

$fd[l_1[i]-1..i, l_2[j]-1..j]$: empty array;
$fd[l_1[i]-1, l_2[j]-1] = 0$;
for $d_i = l_1[i]$ **to** i **do** $fd[d_i, l_2[j]-1] = fd[d_i-1, l_2[j]-1] + \omega_{del}$;
for $d_j = l_2[j]$ **to** j **do** $fd[l_1[i]-1, d_j] = fd[l_1[i]-1, d_j-1] + \omega_{ins}$;
for $d_i = l_1[i]$ **to** i **do**
  for $d_j = l_2[j]$ **to** j **do**
    if $l_1[d_i] = l_1[i]$ **and** $l_2[d_j] = l_2[j]$ **then**
      $fd[d_i, d_j] = \min(\ldots)$;
      $td[d_i, d_j] = f[d_i, d_j]$;
    else $fd[d_i, d_j] = \min(\ldots)$;

- Input nodes are i and j.
- They are root nodes of subtrees $t_1(i)$ and $t_2(j)$.
- The nested loop is executed $|t_1(i)| \times |t_2(j)|$ times.
- $\Rightarrow$ Time complexity $O(|t_1(i)| \times |t_2(j)|)$

tree-edit-dist: Time Complexity

tree-edit-dist$(\mathrm{T}_1, \mathrm{T}_2)$

$td[1..|\mathrm{T}_1|, 1..|\mathrm{T}_2|]$: empty array for tree distances;
$l_1 = \mathrm{lmld}(root(\mathrm{T}_1))$; $kr_1 = \mathrm{kr}(l_1, |leaves(\mathrm{T}_1)|)$;
$l_2 = \mathrm{lmld}(root(\mathrm{T}_2))$; $kr_2 = \mathrm{kr}(l_2, |leaves(\mathrm{T}_2)|)$;
for $x = 1$ to $|kr_1|$ **do**
  for $y = 1$ to $|kr_2|$ **do**
    forest-dist$(kr_1[x], kr_2[y], l_1, l_2, td)$;

- Computing $l_{1/2}$ and $kr_{1/2}$ is linear, $O(|\mathrm{T}_1| + |\mathrm{T}_2|)$
- Main loop executes forest-dist() $|kr_1| \times |kr_2|$ times.
- Complexity:

$$\sum_{i \in kr_1} \sum_{j \in kr_2} |t_1(i)| \times |t_2(j)| = \sum_{i \in kr_1} |t_1(i)| \times \sum_{j \in kr_2} |t_2(j)|$$

- The following lemmas help us to reformulate this expression.

Collapsed Depth

- **Definition:** The collapsed depth of a node v in T is

$$cdepth(\mathrm{v}) = |anc(\mathrm{v}) \cap kr(\mathrm{T})|,$$

i.e., the number of ancestors of v (including v itself) that are key roots.

Collapsed Depth

Lemma (Collapsed Depth)

For a tree T *with key roots* $kr(\mathrm{T})$

$$\sum_{k \in kr(\mathrm{T})} |t(k)| = \sum_{k=1}^{|\mathrm{T}|} cdepth(k)$$

Proof.

- Consider the left-hand formula:
  - A node i of T is counted whenever it appears in a subtree $t(k)$.
  - Node i is in the subtree $t(k)$ iff k is the ancestor of i.
  - Only the subtrees of key roots are considered.
- Thus a node i is counted once for each ancestor key root.
- $cdepth(i)$ is the number of ancestor key roots of i (definition of collapsed depth). □

Collapsed Depth

- Now we can rewrite the complexity formula:

$$\sum_{i \in kr_1} |t_1(i)| \times \sum_{j \in kr_2} |t_2(j)| = \sum_{i=1}^{|\mathrm{T}_1|} cdepth(i) \times \sum_{j=1}^{|\mathrm{T}_2|} cdepth(j)$$

- $cdepth(\mathrm{T}) \geq cdepth(i)$ for a node i of T, thus

$$\sum_{i=1}^{|\mathrm{T}_1|} cdepth(i) \times \sum_{j=1}^{|\mathrm{T}_2|} cdepth(j) \leq |\mathrm{T}_1||\mathrm{T}_2| cdepth(\mathrm{T}_1) cdepth(\mathrm{T}_2)$$

- Two obvious upper bounds for the collapsed depth:
  - the tree depth: $cdepth(\mathrm{T}) \leq depth(\mathrm{T})$
  - the number of key roots: $cdepth(\mathrm{T}) \leq |kr(\mathrm{T})|$
- We show that the number of key roots matches the number of leaves.

Number of Key Roots

Lemma (Number of Key Roots)

The number of key roots of a tree is equal to the number of leaves:

$$|kr(\mathrm{T})| = |leaves(\mathrm{T})|$$

Proof.

We show that $l()$ is a bijection from the key roots $kr(\mathrm{T})$ to the $leaves(\mathrm{T})$:

(a) **Injection** – for any $i, j \in kr(\mathrm{T})$, $i \neq j \Rightarrow l(i) \neq l(j)$:
If $i > j$ and $l(i) = l(j)$, j can not be a key root by definition.
Analogous rational hold for $j > i$.

(b) **Surjection** – Each leaf x has a key root $i \in kr(\mathrm{T})$ such that $l(i) = x$:
If there is no node $i > x$ with $l(i) = l(x)$, then by definition x itself is a key root ($l(x) = x$ is always true). Otherwise i is the key root of x.

□

Complexity of the Tree Edit Distance Algorithm

Theorem (Complexity of the Tree Edit Distance Algorithm)

Let D_1 and D_2 denote the depth, L_1 and L_2 the number of leaf nodes, and N_1 and N_2 the total number of nodes of two trees T_1 and T_2, respectively.

(1) *The runtime of the tree edit distance algorithm is*

$$O(N_1 N_2 \min(D_1, L_1) \min(D_2, L_2)).$$

(2) *Let $N = \max(N_1, N_2)$. For full, balanced, binary trees the runtime is*

$$O(N^2 \log^2 N).$$

(3) *In the worst case $\min(D, L) = O(N)$ and the runtime is $O(N^4)$.*

(4) *The algorithm needs $O(N_1 N_2)$ space.*

Proof of the Complexity Theorem

Proof.

(1) Runtime (general formula): We have shown before, that the complexity is $O(|\mathrm{T}_1||\mathrm{T}_2| cdepth(\mathrm{T}_1)\, cdepth(\mathrm{T}_2))$. As $cdepth(\mathrm{T}) \leq |kr(\mathrm{T})| = |leaves(\mathrm{T})|$ (see definition of $cdepth(\mathrm{T})$ and previous lemma) and $cdepth(\mathrm{T}) \leq depth(\mathrm{T})$ (follows from the definition of $cdepth(\mathrm{T})$), if follows that $cdepth(\mathrm{T}) \leq \min(depth(\mathrm{T}), |leaves(\mathrm{T})|)$.

(2) Full, balanced, binary trees: In this case $depth(\mathrm{T}) = O(\log(|\mathrm{T}|))$.

(3) Worst case: A full binary tree (i.e., each node has zero or two children) where each non-leaf node has at least one leaf child: $\min(depth(\mathrm{T}), |leaves(\mathrm{T})|) = O(|\mathrm{T}|)$.

(4) Space: The size of the tree distance matrix *td* is $|\mathrm{T}_1| \times |\mathrm{T}_2|$. In each call of forest-dist() we need a matrix of size $O(|\mathrm{T}_1| \times |\mathrm{T}_2|)$, which is freed when we exit the subroutine. □

Improvements over the Complexity of the Zhang&Shasha Algorithm

- Klein [Kle98] improves the worst case for the runtime to $O(|\mathrm{T}_1|^2 |\mathrm{T}_2| \log(|\mathrm{T}_2|))$, thus from $O(N^4)$ to $O(N^3 \log(N))$.
- Dulucq and Touzet [DT03] also give an $O(N^3 \log(N))$ algorithm.
- Demaine et al. [DMRW07] give an $O(N^3)$ algorithm. They show that the algorithm is worst case optimal among all *decomposition algorithms* (i.e., algorithms like [ZS89, Kle98, DT03]), but it is not robust, i.e., it runs into the worst case when it could do better.
- Pawlik and Augsten [PA11] introduce the Robust Tree Edit Distance (RTED) algorithm which has optimal $O(N^3)$ worst case complexity and is robust.

Further reading:
`http://tree-edit-distance.dbresearch.uni-salzburg.at`

Summary

- Edit distance for trees
  - Edit scripts and mappings
  - Recursive Formula
  - Dynamic programming algorithm
  - Tree edit distance example
  - Tree Edit Distance Complexity


Alberto Apostolico and Zvi Galill, editors.
Pattern Matching Algorithms, chapter Tree Pattern Matching, pages 341–371.
Oxford University Press, 1997.


Erik D. Demaine, Shay Mozes, Benjamin Rossman, and Oren Weimann.
An optimal decomposition algorithm for tree edit distance.
In *Proceedings of the 34th International Colloquium on Automata, Languages and Programming (ICALP 2007)*, volume 4596 of *Lecture Notes in Computer Science*, pages 146–157, Wroclaw, Poland, July 2007. Springer.


Serge Dulucq and Hélène Touzet.
Analysis of tree edit distance algorithms.
In *Combinatorial Pattern Matching (CPM 2003)*, volume 2676 of *Lecture Notes in Computer Science*, pages 83–95, Berlin, 2003. Springer.


Philip N. Klein.
Computing the edit-distance between unrooted ordered trees.
In *Proceedings of the 6th European Symposium on Algorithms*, volume 1461 of *Lecture Notes in Computer Science*, pages 91–102, Venice, Italy, 1998. Springer.


Mateusz Pawlik and Nikolaus Augsten.
RTED: A robust algorithm for the tree edit distance.
PVLDB, 5(4):334–345, December 2011.
ISSN 2150-8097.


Kaizhong Zhang and Dennis Shasha.
Simple fast algorithms for the editing distance between trees and related problems.
SIAM Journal on Computing, 18(6):1245–1262, 1989.