Learning Probabilistic Models of Tree Edit Distance [⋆]

Marc Bernard [a] Laurent Boyer [a] Amaury Habrard [b] Marc Sebban [a]

[a] *Laboratoire Hubert Curien, Université de Saint-Etienne, 18 rue du Professeur Lauras, 42000 Saint-Etienne (France)*

[b] *Laboratoire d'Informatique Fondamentale, Université de Provence, 39 rue Frédéric Joliot Curie, 13453 Marseille cedex 13 (France)*

Abstract

Nowadays, there is a growing interest in machine learning and pattern recognition for tree-structured data. Trees actually provide a suitable structural representation to deal with complex tasks such as web information extraction, RNA secondary structure prediction, computer music, or conversion of semi-structured data (*e.g.* XML documents). Many applications in these domains require the calculation of similarities over pairs of trees. In this context, the tree edit distance (ED) has been subject of investigations for many years in order to improve its computational efficiency. However, used in its classical form, the tree ED needs *a priori* fixed edit costs which are often difficult to tune, that leaves little room for tackling complex problems. In this paper, to overcome this drawback, we focus on the automatic learning of a non parametric stochastic tree ED. More precisely, we are interested in two kinds of probabilistic approaches. The first one builds a generative model of the tree ED from a joint distribution over the edit operations, while the second works from a conditional distribution providing then a discriminative model. To tackle these tasks, we present an adaptation of the Expectation-Maximization algorithm for learning these distributions over the primitive edit costs. Two experiments are conducted. The first is achieved on artificial data and confirms the interest to learn a tree ED rather than *a priori* imposing edit costs; The second is applied to a pattern recognition task aiming to classify handwritten digits.

⋆ This work is part of the ongoing ARA Marmota research project.
Email addresses: `marc.bernard@univ-st-etienne.fr` (Marc Bernard), `laurent.boyer@univ-st-etienne.fr` (Laurent Boyer), `amaury.habrard@lif.univ-mrs.fr` (Amaury Habrard), `marc.sebban@univ-st-etienne.fr` (Marc Sebban).

1 Introduction

Nowadays, there is a growing interest for tree-structured data due to the potential applications in information extraction from the web, computational biology or phylogeny. Indeed, the hierarchical structure of trees is more suited for modeling web pages (XML, HTML), the RNA secondary structure of a molecule or phylogenetic trees than a flat representation such as strings. In applications, one often needs similarity measures to compare two different instances. This is, for example, useful for defining conversion models for dealing with heterogeneous XML data. In this context, many approaches have extended the well known string edit distance (ED) to trees [1].

The tree ED is defined as the least costly set of basic operations to change one tree to another. These primitive operations usually consist of *substitution*, *deletion* and *insertion* of a node. Tree ED-based methods use, in general, *a priori* fixed costs for these so-called primitive edit operations. However, in many domains, an edit cost can mainly depend on the nature of the symbols used in a given operation. For example, the probability of changing a given symbol in a RNA structure depends on the probability that a genetic mutation occurs on this symbol. Thus, the similarity of two trees can vary a lot according to the specific domain in consideration. A solution could consist of assigning costs according to an expert valuation. However, this strategy may not be efficiently done in domains where the expertise is low. Moreover, even if the expertise level is sufficient, assigning a relevant cost to each edit operation can become a tricky task. Another way to overcome this drawback would consist in automatically learning these edit costs from a learning sample. This is the purpose of this paper. We propose a model of an ED as a stochastic process and we use probabilistic methods to learn the model. Several strategies can be used to learn these edit costs. In this paper, the objective function we are going to minimize is the total distance between training (*input,output*) pairs of trees. In this context, the aim is to learn the optimal edit costs that allow us to transform an *input* tree into an *output* one. From an application point of view, these (*input,output*) pairs can take many forms. For example, the input tree would be a noisy data; and the output one is the corresponding unnoisy tree. The aim may be then to learn the edit costs that allow us to automatically correct a corrupted instance. In another application, the goal could be to learn a document conversion model, where the input tree is a given XML document, and the output tree the corresponding XML conversion satisfying a DTD schema.

Note that in the context of strings, several approaches have already been proposed during the last decade to learn a stochastic ED in the form of stochastic transducers [2,3], conditional random fields [4] or pair-Hidden-Markov-Models (pair-HMM) [5]. A parametric approach has also been presented in [6] in the context of graph ED, where each edit operation is modeled by a Gaussian Mixture Density. With the exception of our preliminary work [7], as far as we know, no method has been proposed to directly learn edit costs for a stochastic tree ED. The aim of this paper is to fill this gap by a non parametric stochastic method specifically adapted to trees.

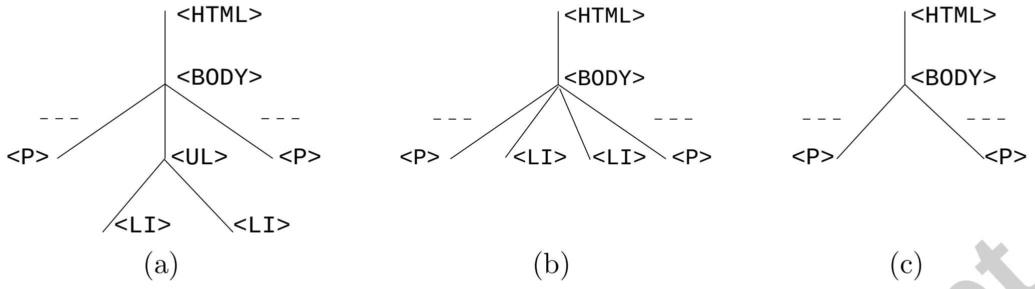


Fig. 1. Strategies to delete a node within a tree.

As we indicated before, the primitive edit operations for the standard tree ED are the substitution, insertion and deletion of a node. The most efficient procedures proposed notably by Zhang and Shasha [8] and Klein [9] have a polynomial complexity of order 4 [10,11]. In these approaches, when a node is deleted within a tree, all its children are then connected to its father. This may not be relevant in some cases, for example in an HTML document: considering a set of items in an unordered list (see Fig. 1.a), it seems clearly irrelevant to delete the `<UL>` node without deleting the `<LI>` items (Fig. 1.b). Thus, to take into account this phenomenon, and particularly to reduce the algorithmic complexity, we decided to use the less costly (with a quadratic complexity) tree ED, initially proposed by Selkow [12], as a base for our stochastic approach. In this case, only a deletion of an entire (sub)tree can occur, and its removal implies the deletion of all its nodes (Fig. 1.c). The insertion of a (sub)tree follows the same principle, *i.e.* requires the iterative insertion of its nodes. In other words, this means that Selkow's approach forbids some edit operations, such as the insertion/deletion of a single node. Using this strategy, the minimization of the distance between tree pairs will be then achieved using a subset of edit operations that allow a reduction of the algorithmic complexity. However, despite this inconvenience, it will be possible to learn edit operations on single nodes by combining operations on subtrees.

Two approaches are offered in this paper for learning, from a sample of (*input,output*) pairs of trees, the costs used for computing a stochastic tree ED. First, we learn a *generative* model in the form of a joint distribution over edit operations inspired by [2] in the case of strings. The advantage of such generative models is to provide an estimate of the unknown joint density with a small variance. However, it has an important drawback: the estimate is biased because it depends on the distribution of the node labels of the *input* trees. In other words, this generative model will work well if the distribution over the labels of the learning input trees follows the (usually unknown) underlying density of the input trees. This constraint justifies our second approach based on the learning of a *discriminative* model in the form of a conditional distribution. This type of model is known [13] to provide an unbiased estimate (despite a higher variance). Such a strategy will be shown to work whatever the input distribution is used.

The rest of the paper is organized as follows: After some notations and definitions about the non-stochastic tree ED in Section 2, our two learning methods are presented in Sections 3 and 4. They are based on an adaptation of the well-known *Expectation-Maximization* algorithm (EM) [14]. The expectation step requires the use of two procedures called *forward* and *backward* that we will adapt for learning a stochastic version of a Selkow's approach-based tree ED. Even if the expectation step actually requires a specific adaptation according to the studied tree ED, we will show that the optimization constraints to satisfy during the maximization stage would remain the same for any stochastic tree ED. In Section 5, we first carry out some series of experiments on artificial datasets showing, from a theoretical standpoint, that our approach is able to learn target distributions. Then, we present results about the behavior of our approach on a pattern recognition task aiming to classify handwritten digits, before concluding.

2 Tree ED

2.1 Notations

We assume we handle ordered labeled trees of arbitrary arity. Since we use only basic concepts on trees, we simply define a tree by $v(a_1, \ldots, a_T)$ where v is a node and $a_1, \ldots, a_T$ are trees[1]. Each node of a tree is labeled by a symbol. We denote by $\mathcal{L}$ the set of labels and by $\lambda \notin \mathcal{L}$ the empty label. For convenience, we will denote the labeled tree $v(a_1, \ldots, a_T)$ by $l(a_1, \ldots, a_T)$, where l is the label of v.

Since we often need to handle pairs of trees, we use the detailed notations $l(a_1, \ldots, a_T)$ and $l'(b_1, \ldots, b_V)$ in this paper when the recursive definition of a tree is necessary. In the opposite case, we use the simple notations x and y.

[1] Formal definitions on trees are provided in [15].

2.2 Edit operations and edit cost functions

The main edit operations are now presented which allow us to change an input tree $l(a_1, \ldots, a_T)$ into an output one $l'(b_1, \ldots, b_V)$. In this paper, we are only concerned by three possible edit operations on trees: Substitution of the label l of a (sub)tree root by l' (denoted (l, l')), deletion of a subtree a_i (denoted (a_i, λ)), and insertion of a subtree a_j (denoted (λ, a_j)) (see Fig. 2).

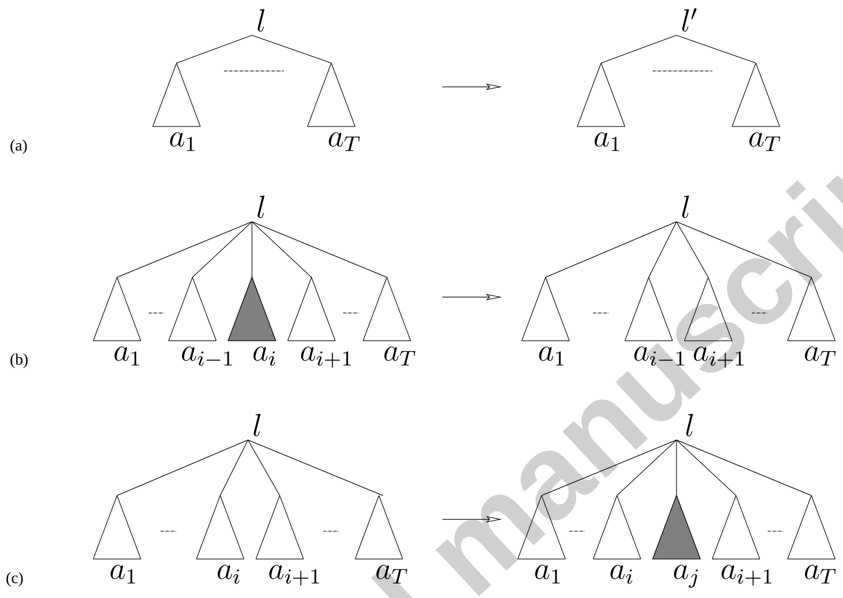


Fig. 2. (a) Substitution of l by l' (b) Deletion of a_i (c) Insertion of a_j.

Let us define a cost function δ_t over these edit operations. Since a deletion or an insertion of a tree are respectively achieved by iteratively removing or inserting a set of nodes, δ_t can be directly defined from a cost function δ of edit operations *on labels* of the nodes. More formally, δ is a function defined from $(\mathcal{L} \cup \{\lambda\}) \times (\mathcal{L} \cup \{\lambda\}) \backslash \{(\lambda, \lambda)\}$ to $[0, 1]$.

The cost of the deletion of a tree can then be recursively computed as follows:

$$\delta_t(l(a_1, \ldots, a_T), \lambda) = \delta(l, \lambda) + \sum_{i=1}^{T} \delta_t(a_i, \lambda).$$

As this paper's introduction indicated, the cost matrix δ is usually *a priori* fixed. For example, consider the cost matrix δ of Fig. 3 and a given tree $b(c, d)$, then $\delta_t(b(c, d), \lambda) = \delta(b, \lambda) + \delta_t(c, \lambda) + \delta_t(d, \lambda) = \delta(b, \lambda) + \delta(c, \lambda) + \delta(d, \lambda) = 1.5$. Based on the same principle, the insertion of a tree requires successive insertions of its nodes, such that:

δ	λ	a	b	c	d
λ	$-$	0.5	0.5	0.5	0.5
a	0.5	0	1	1	1
b	0.5	1	0	1	1
c	0.5	1	1	0	1
d	0.5	1	1	1	0

Fig. 3. A matrix δ.

$$\delta_t(\lambda, l'(b_1, \ldots, b_V)) = \delta(\lambda, l') + \sum_{j=1}^{V} \delta_t(\lambda, b_j).$$

Finally, the substitution of two labels is defined as follows:

$$\delta_t(l, l') = \delta(l, l').$$

2.3 Classic tree ED algorithms

Once the cost function δ_t is established, it is possible to define a tree ED based on the following notion of edit script.

Definition 1 *Let x and y be two trees, an edit script $e = e_1 \cdots e_n$ on these trees, where $e_i = (l_i, l'_i) \in (\mathcal{L} \cup \{\lambda\})^2 \backslash \{(\lambda, \lambda)\}$ is a sequence of edit operations changing x into y. The cost of an edit script is the sum of the costs of its edit operations.*

Since several scripts can exist (as shown in Fig. 4), one can define the tree ED as follows:

Definition 2 *The tree ED $d(l(a_1, \ldots, a_T), l'(b_1, \ldots, b_V))$ between two trees $l(a_1, \ldots, a_T)$ and $l'(b_1, \ldots, b_V)$ is the cost of the minimum cost edit script.*

As described in [12], it can be recursively computed as follows:

$$\begin{aligned} d(\lambda, \lambda) &= 0 \\ d(l(a_1, \ldots, a_T), \lambda) &= \delta_t(l(a_1, \ldots, a_T), \lambda) \\ d(\lambda, l'(b_1, \ldots, b_V)) &= \delta_t(\lambda, l'(b_1, \ldots, b_V)) \\ d(l(a_1, \ldots, a_T), l'(b_1, \ldots, b_V)) &= \delta(l, l') + d'(a_1, \ldots, a_T : b_1, \ldots, b_V) \end{aligned}$$

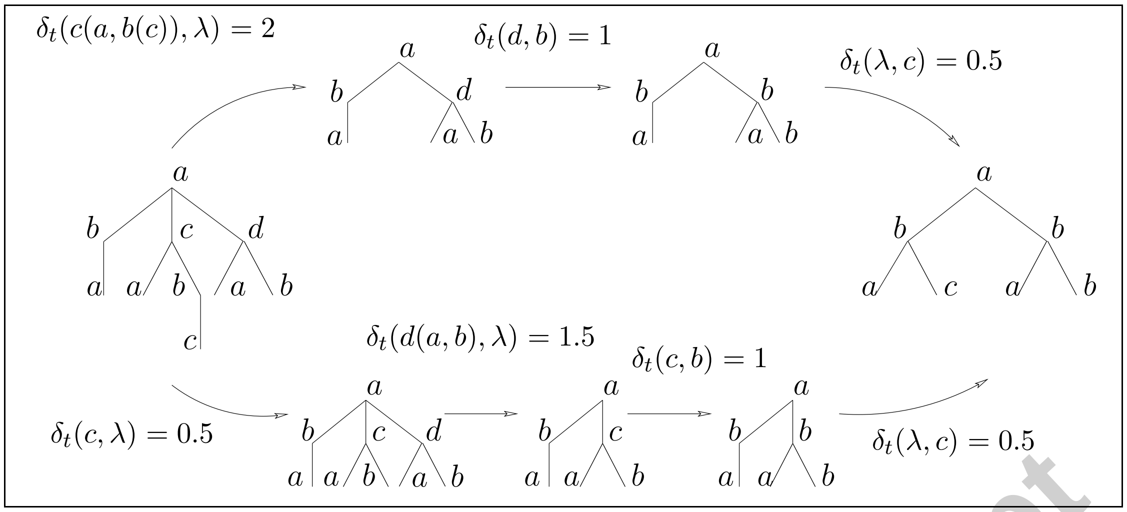

$\delta_t(c(a,b(c)),\lambda) = 2$
$\delta_t(d,b) = 1$
$\delta_t(\lambda,c) = 0.5$
$\delta_t(d(a,b),\lambda) = 1.5$
$\delta_t(c,b) = 1$
$\delta_t(c,\lambda) = 0.5$
$\delta_t(\lambda,c) = 0.5$

Fig. 4. Two possible edit scripts that are applicable between two trees.

where δ_t is defined as shown in Section 2.2 and d' as follows:

$$
\begin{aligned}
d'(\lambda : \lambda) &= 0 \\
d'(a_1,\ldots,a_T : \lambda) &= d'(a_1,\ldots,a_{T-1} : \lambda) + \delta_t(a_T,\lambda) \\
d'(\lambda : b_1,\ldots,b_V) &= d'(\lambda : b_1,\ldots,b_{V-1}) + \delta_t(\lambda,b_V) \\
d'(a_1,\ldots,a_T : b_1,\ldots,b_V) &= \min\begin{cases} d'(a_1,\ldots,a_{T-1} : b_1,\ldots,b_V) + \delta_t(a_T,\lambda) \\ d'(a_1,\ldots,a_T : b_1,\ldots,b_{V-1}) + \delta_t(\lambda,b_V) \\ d'(a_1,\ldots,a_{T-1} : b_1,\ldots,b_{V-1}) + d(a_T,b_V) \end{cases}
\end{aligned}
$$

Note that this distance can be efficiently computed using dynamic programming. However, so far, we assumed that δ was given. While δ is the core of the computation of a tree ED, its valuation can constitute a tricky task in many domains. To overcome this drawback, in the next section, we show how it is possible to automatically learn this matrix δ from a corpus of tree pairs. Our approach is stochastic and based on an adaptation of the EM algorithm [14], which aims at estimating the hidden parameters (the matrix δ) of a probabilistic model from a learning sample. In this new context, the tree ED becomes a stochastic tree ED. Then, we do not use tuned edit costs anymore, but we turn to learn *edit probabilities*. For convenience, we decided to keep unchanged the notation of matrix δ, even if it will represent edit probabilities rather than edit costs. Thus, $\delta(l,l')$ is designated to represent the probability to change the symbol l into l'. In this stochastic context, the probability of the edit operation (l,l) will not always be equal to 1. This is because the symbol l could have a non null probability to be changed into another letter l'.

In the following sections, we illustrate two ways of learning a *stochastic edit distance* between two trees. The first one (Section 3) concerns a *generative* model over the edit scripts changing an input tree into an output one, while the second one (Section 4) learns a so-called *discriminative* model. The main difference between both of them occurs during the maximization step of EM.

Note that throughout the following two sections, an estimated probability from the learned matrix δ will be denoted with a subscript δ.

3 Learning Stochastic Joint Tree Edit Distance

3.1 Definitions

A *probabilistic ED* supposes that edit operations are applied according to an unknown random process. Our goal is to learn this underlying probability distribution $\delta(l, l')$ in order to estimate a joint distribution over sequences of edit operations, the so-called edit scripts. We use a special symbol # to denote the end of an edit script. For the sake of convenience, we also denote the termination probability of a script $\delta(\#)$ by $\delta(\lambda, \lambda)$.

The probability of an edit script $e = e_1 \cdots e_n$ is the product of the probabilities of each edit operation, such that:

$$\pi_\delta(e) = \prod_{i=1}^{n} \delta(e_i) \times \delta(\#).$$

To model the distance between two trees, we propose to compute the probability of all ways to change an input tree x into an output one y (as described in [2] for the case of strings). This probability, denoted $p_\delta(x, y)$, enables us to model a *stochastic tree ED*.

Definition 3 *We define $E(x, y)$ as the set of all possible edit scripts for transforming x into y. The stochastic tree ED between two trees is defined by:*

$$d_s(x, y) = -\log p_\delta(x, y) = -\log \sum_{e \in E(x,y)} \pi_\delta(e).$$

Note that we could have used another strategy to obtain a probabilistic ED consisting of only keeping the most probable script and then computing a so-called Viterbi distance $\mathrm{d}_v(x, y)$, defined as:

$$d_v(x, y) = -\log \max_{e \in E(x,y)} \pi_\delta(e).$$

Achieving this task would only require slight modifications.

3.2 Adapted EM algorithm

To learn the matrix δ and then compute the joint probability $p_\delta(l(a_1, \ldots, a_T), l'(b_1, \ldots, b_V))$, we use an adaptation of the EM algorithm [14]. EM estimates the hidden parameters of a probabilistic model by maximizing the likelihood of a learning sample. In our case, the parameters will correspond to the matrix δ of edit probabilities, and the learning sample will be composed of (*input,output*) pairs of trees.

EM achieves an expectation step followed by a maximization stage. During the first step, EM accumulates, from the training corpus, the expectation of each hidden event (edit operation) for transforming an input tree into an output one. In the maximization step, EM sets the parameter values (edit probabilities) in order to maximize the likelihood. To learn a stochastic tree ED, we adapted EM in the context of trees and more precisely for the learning of tree edit distances that are based on Selkow's algorithm.

3.2.1 Forward and Backward functions

To learn the matrix δ, EM uses two auxiliary functions, so-called *forward* ($\boldsymbol{\alpha}$) and *backward* ($\boldsymbol{\beta}$), that are respectively described in Algorithm 1 and Algorithm 2.

```
Input: Two trees $l(a_1, \ldots, a_i)$ and $l'(b_1, \ldots, b_j)$, $1 \leq i \leq T$ and $1 \leq j \leq V$
Output: $p_\delta(l(a_1, \ldots, a_i), l'(b_1, \ldots, b_j))$
Let $\alpha[0..T, 0..V]$ be a $(T+1) \times (V+1)$ matrix
$\alpha[0,0] \leftarrow \delta(l, l')$
for $t = 0$ to $i$ do
    for $v = 0$ to $j$ do
        if $(t > 0)$ or $(v > 0)$ then $\alpha[t,v] \leftarrow 0$
        if $(t > 0)$ then $\alpha[t,v] \leftarrow \alpha[t,v] + \boldsymbol{\alpha}(a_t, \lambda) \times \alpha[t-1, v]$
        if $(v > 0)$ then $\alpha[t,v] \leftarrow \alpha[t,v] + \boldsymbol{\alpha}(\lambda, b_v) \times \alpha[t, v-1]$
        if $(t > 0)$ and $(v > 0)$ then $\alpha[t,v] \leftarrow \alpha[t,v] + \boldsymbol{\alpha}(a_t, b_v) \times \alpha[t-1, v-1]$
return $\alpha[i][j]$
```

Algorithm 1: Forward function $\boldsymbol{\alpha}(l(a_1, \ldots, a_i), l'(b_1, \ldots, b_j))$

These two functions take a pair of trees as input and compute the probability of the possible edit scripts between these two trees. From an algorithmic

```
Input: Two trees $l(a_i, \ldots, a_T)$ and $l'(b_j, \ldots, b_V)$, $1 \leq i \leq T$ and $1 \leq j \leq V$
Output: $p_\delta(l(a_i, \ldots, a_T), l'(b_j, \ldots, b_V))$
Let $\beta[0..T, 0..V]$ be a $(T+1) \times (V+1)$ matrix
$\beta[T, V] \leftarrow 1$
for $t = T$ down to $i-1$ do
    for $v = V$ down to $j-1$ do
        if $(t < T)$ or $(v < V)$ then $\beta[t, v] \leftarrow 0$
        if $(t < T)$ then $\beta[t, v] \leftarrow \beta[t, v] + \boldsymbol{\beta}(a_{t+1}, \lambda) \times \beta[t+1, v]$
        if $(v < V)$ then $\beta[t, v] \leftarrow \beta[t, v] + \boldsymbol{\beta}(\lambda, b_{v+1}) \times \beta[t, v+1]$
        if $(t < T)$ and $(v < V)$ then
            $\beta[t, v] \leftarrow \beta[t, v] + \boldsymbol{\beta}(a_{t+1}, b_{v+1}) \times \beta[t+1, v+1]$
if $i = 1$ and $j = 1$ then return $\beta[0][0] \times \delta(l, l')$ else return $\beta[i-1][j-1]$
```

Algorithm 2: Backward function $\boldsymbol{\beta}(l(a_1, \ldots, a_i), l'(b_1, \ldots, b_j))$

standpoint, this requires recursively calculating the probability of the edit operations on the pairs of subtrees of the same depth. Although they process differently, these functions are symmetric and provide the same probability:

$$\begin{aligned} p_\delta(l(a_1, \ldots, a_T), l'(b_1, \ldots, b_V)) &= \boldsymbol{\alpha}(l(a_1, \ldots, a_T), l'(b_1, \ldots, b_V)) \times \delta(\#) \\ &= \boldsymbol{\beta}(l(a_1, \ldots, a_T), l'(b_1, \ldots, b_V)) \times \delta(\#). \end{aligned}$$

The *forward* function visits the roots first and then scans the children from left to right, while the backward function processes from right to left and finally visits the roots of the pair of trees. Fig. 5 illustrates these two algorithms that can be computed with a quadratic complexity using dynamic programming techniques. We can note that all the ways for transforming the input tree into the output one are taken into account, that explains why we sum, in both functions, the probabilities of edit scripts. To compute a Viterbi distance, we would have kept the most probable path, that only requires slight modifications such as the use of a max function.

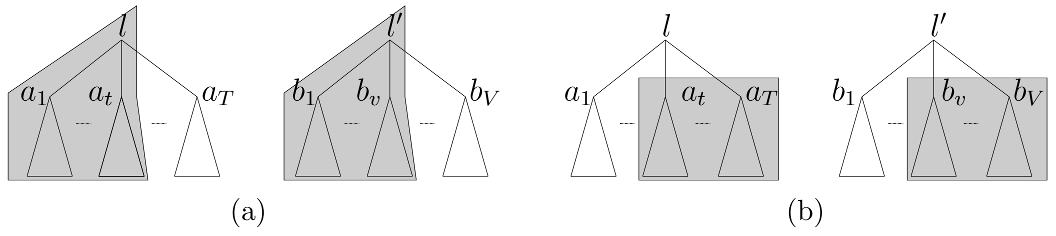


Fig. 5. (a) Evaluation of $\boldsymbol{\alpha}(l(a_1, \ldots, a_t), l'(b_1, \ldots, b_v))$ by the forward algorithm. (b) Evaluation of $\boldsymbol{\beta}(l(a_t, \ldots, a_T), l'(b_v, \ldots, b_V))$ by the backward algorithm.

3.2.2 Expectation Step

During the expectation step, we estimate the expectation of the hidden events, *i.e.* the edit operations used to transform an input tree into an output one. These expectations are stored in an auxiliary matrix γ of dimension $(|\mathcal{L}|+1) \times (|\mathcal{L}|+1)$. This process (see Algorithm 3) takes a training pair of trees (x, y) in input. Then, for all the pairs of subtrees (a_t, b_v), where a_t is a subtree of x and b_v a subtree of y, and where a_t and b_v are located at the same depth in x and y, it accumulates the expectations of the three edit operations by either deleting a_t, or inserting b_v or substituting a_t by b_v. In this algorithm, $l_r(a_t)$ (resp. $l_r(b_v)$) denotes the label of the root of a tree a_t (resp. b_v).

```
Input: Two trees $x$ and $y$
Let $(a_t, b_v)$ be a pair of trees where there are: 2 labels $l$ and $l'$, 2 contexts $C[\ ]$,
$C'[\ ]$ with $depth_c(C[\ ]) = depth_c(C'[\ ])$ and
$(T-1)+(V-1)$ trees $a_1,\dots,a_{t-1},a_{t+1},\dots,a_T,b_1,\dots,b_{v-1},b_{v+1},\dots,b_V$
with $x = C[l(a_1,\dots,a_{t-1},a_t,a_{t+1},\dots,a_T)]$
and $y = C'[l'(b_1,\dots,b_{v-1},b_v,b_{v+1},\dots,b_V)]$

foreach pair $(a_t, b_v)$ in $(x, y)$ do
    if $(a_t \neq \lambda)$ then
        $\gamma(l_r(a_t),\lambda) \leftarrow \gamma(l_r(a_t),\lambda) + \frac{\boldsymbol{\alpha\beta}(C[\ ],C'[\ ])\boldsymbol{\alpha}(l(a_1,\dots,a_{t-1}),l'(b_1,\dots,b_v))\boldsymbol{\alpha}(a_t,\lambda)\boldsymbol{\beta}(l(a_{t+1},\dots,a_T),l'(b_{v+1},\dots,b_V))}{\boldsymbol{\alpha}(x,y)}$
    end
    if $(b_v \neq \lambda)$ then
        $\gamma(\lambda,l_r(b_v)) \leftarrow \gamma(\lambda,l_r(b_v)) + \frac{\boldsymbol{\alpha\beta}(C[\ ],C'[\ ])\boldsymbol{\alpha}(l(a_1,\dots,a_t),l'(b_1,\dots,b_{v-1}))\boldsymbol{\alpha}(\lambda,b_v)\boldsymbol{\beta}(l(a_{t+1},\dots,a_T),l'(b_{v+1},\dots,b_V))}{\boldsymbol{\alpha}(x,y)}$
    end
    if $(a_t \neq \lambda)$ and $(b_v \neq \lambda)$ then
        $\gamma(l_r(a_t),l_r(b_v)) \leftarrow \gamma(l_r(a_t),l_r(b_v)) + \frac{\boldsymbol{\alpha\beta}(C[\ ],C'[\ ])\boldsymbol{\alpha}(l(a_1,\dots,a_{t-1}),l'(b_1,\dots,b_{v-1}))\boldsymbol{\alpha}(a_t,b_v)\boldsymbol{\beta}(l(a_{t+1},\dots,a_T),l'(b_{v+1},\dots,b_V))}{\boldsymbol{\alpha}(x,y)}$
    end
end
```

Algorithm 3: *expectation*$(x,\ y)$

Since a_t and b_v are subtrees, the calculation of these expectations needs to take into account not only their siblings but also the rest of each of the two trees x and y that are not directly concerned by the operations on a_t and b_v, and that can be formally defined as a *context*.

Definition 4 *A context $C[\ ]$ is a non empty tree where exactly one leaf (i.e. a node without any subtree) is labeled by a symbol \$ such that $\$ \notin \mathcal{L} \cup \{\lambda\}$. If $C[\ ]$ is a context and x a tree, $C[x]$ denotes the tree obtained by substituting* *the node labeled by \$ by the tree x (see Figure 6). The depth of a context $C[\ ]$, denoted by $depth_c(C[\ ])$, corresponds to the depth of the node labeled by \$.*

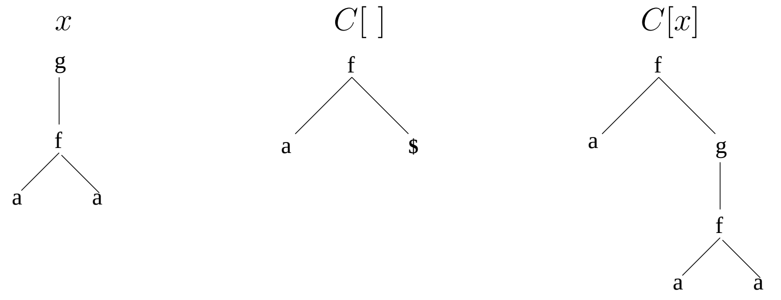
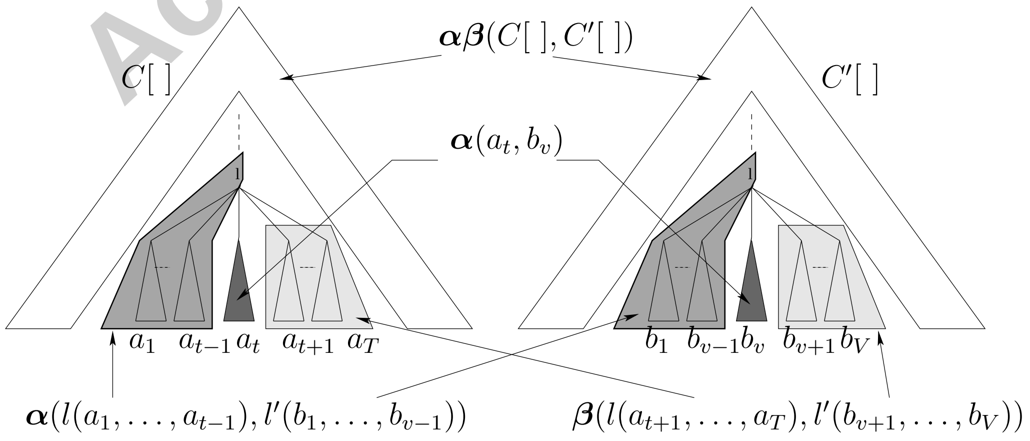


Fig. 6. A tree x, a context $C[\ ]$ of $depth_c(C[\ ]) = 1$ and the resulting tree $C[x]$.

This definition enables us to now explain the calculations done in Algorithm 3 on two subtrees (a_t, b_v). For the sake of simplicity, let us only focus on the substitution case as described in Fig. 7:

- The function $\boldsymbol{\alpha\beta}$, described in Algorithm 4, computes the joint probability of the two contexts $C[\ ]$ of a_t and $C'[\ ]$ of b_v. We call this function $\boldsymbol{\alpha\beta}$ because it uses the forward and backward functions to compute respectively the left and right parts of the contexts.
- The forward and backward functions are used to compute the probability of the left, *i.e.* $\boldsymbol{\alpha}(l(a_1, \dots, a_{t-1}), l'(b_1, \dots, b_{v-1}))$, and right, *i.e.* $\boldsymbol{\beta}(l(a_{t+1}, \dots, a_T), l'(b_{v+1}, \dots, b_V))$ parts of the substitution operation on a_t and b_v.
- The forward function is finally used to calculate the probability of the edit operation itself. Since this edit operation is carried out on a tree, this implies a vertical recursion.

The probabilities obtained by the previous functions are then divided by the whole probability $\alpha(x, y)$ to deduce the required expectations.



$\boldsymbol{\alpha\beta}(C[\ ], C'[\ ])$
$C[\ ]$
$C'[\ ]$
$\boldsymbol{\alpha}(a_t, b_v)$
l
l
a_1 a_{t-1} a_t a_{t+1} a_T
b_1 b_{v-1} b_v b_{v+1} b_V
$\boldsymbol{\alpha}(l(a_1, \dots, a_{t-1}), l'(b_1, \dots, b_{v-1}))$
$\boldsymbol{\beta}(l(a_{t+1}, \dots, a_T), l'(b_{v+1}, \dots, b_V))$

Fig. 7. Graphical explanation of a substitution (a_t, b_v).

```
**Input**: Two contexts $C[\,]$ and $C'[\,]$
**Output**: Probability of pair $(C[\,], C'[\,])$
**if** $C[\,] = \$$ *and* $C'[\,] = \$$ **then**
    **return** $1$
**else**
    Let $l, l'$ be two labels, $C_l[\,]$ and $C_{l'}[\,]$ two contexts and
    $(T-1)+(V-1)$ trees $a_1,\ldots,a_{t-1},a_{t+1},\ldots,a_T,b_1,\ldots,b_{v-1},b_{v+1},\ldots,b_V$ s.t.
    $C[\,] = l(a_1,\ldots,a_{t-1},C_l[\,],a_{t+1},\ldots,a_T)$ and
    $C'[\,] = l'(b_1,\ldots,b_{v-1},C_{l'}[\,],b_{v+1},\ldots,b_V)$

    $A = \boldsymbol{\alpha}(l(a_1,\ldots,a_{t-1}), l'(b_1,\ldots,b_{v-1})) \times \boldsymbol{\beta}(l(a_{t+1},\ldots,a_T), l'(b_{v+1},\ldots,b_V))$
    **return** $A \times \boldsymbol{\alpha\beta}(C_l[\,], C_{l'}[\,])$
```

Algorithm 4: $\boldsymbol{\alpha\beta}(C[\,], C'[\,])$

3.2.3 Maximization Step and EM Algorithm

The maximization step is crucial in the EM algorithm. Actually, it describes the normalization of the expectation values $\gamma(l, l')$, obtained during the expectation step, that we must achieve to maximize the likelihood of the learning set. In our specific case of edit distance learning, this likelihood is computed from all the edit scripts over the training pairs of trees. Beyond the likelihood maximization, this normalization must also provide the δ matrix in the form of a statistical distribution over the edit operations, *i.e.* satisfying the following constraint:

$$\sum_{(l,l')\in(\mathcal{L}\cup\{\lambda\})^2} \delta(l,l') = 1 \quad \text{with} \quad \delta(l,l') \geq 0 \quad \text{and} \quad \delta(\lambda,\lambda) > 0. \tag{1}$$

Note that $\delta(\lambda, \lambda) > 0$ is equivalent to $\delta(\#) > 0$. This means that we must have at least one learning pair of trees.

To ensure this statistical constraint, while maximizing the likelihood of the edit scripts on the learning pairs of trees, we can prove, in the joint case, that the optimal normalization is achieved by the Algorithm 5. It simply consists of dividing each expectation $\gamma(l, l')$ by the total accumulator $TA = \sum_{l\in\mathcal{L}\cup\{\lambda\}} \sum_{l'\in\mathcal{L}\cup\{\lambda\}} \gamma(l, l')$. We don't show here the proof of this optimal normalization because we will detail it, in Section 4, for the conditional case which deserves more technical explanations.

By combining Algorithms 1, 2, 3, 4 and 5, we can now draw the general learning algorithm of a joint stochastic tree ED (see Algorithm 6). Note that the process is repeated until convergence. This is reached either when the probability of each edit operation does not significantly change between two iterations (that can be achieved using a statistical test), or when an *a priori* fixed number of iterations has been run.

```
Input: A matrix of accumulators $\gamma$
Output: A matrix of joint stochastic edit costs $\delta$
$TA \leftarrow 0$
foreach $(l, l') \in (\mathcal{L} \cup \{\lambda\})^2$ do
    $TA \leftarrow TA + \gamma(l, l')$
foreach $(l, l') \in (\mathcal{L} \cup \{\lambda\})^2$ do
    $\delta(l, l') \leftarrow \frac{\gamma(l,l')}{TA}$
```

Algorithm 5: *joint_maximization*(γ)

```
Input: $LS$ a learning set of pairs of trees
repeat
    foreach $(l, l') \in (\mathcal{L} \cup \{\lambda\})^2$ do
        $\gamma(l, l') \leftarrow 0$
    $\gamma(\lambda, \lambda) = |LS|$
    foreach $(l(a_1, \ldots, a_T), l'(b_1, \ldots, b_V)) \in LS$ do
        expectation$(l(a_1, \ldots, a_T), l'(b_1, \ldots, b_V))$
    joint_maximization$(\gamma)$
until convergence
```

Algorithm 6: *expectation* − *maximization*

3.2.4 Example

In order to help the reader to test his own implementation and to improve the understandability of the previous technical sections, we present here the running of our algorithm on a simple example. Consider an alphabet $\mathcal{L} = \{a, b, c\}$ and a training set composed of only one pair of trees $\big(a(b, a(b, c)), a(c, a(c))\big)$ graphically described in Fig.8.

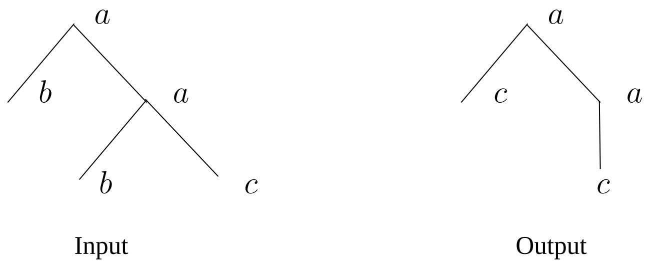


Fig. 8. (*input,output*) pair of trees.

Matrix δ is uniformally initialized with the value $\frac{1}{16}$ for each edit operation.

Matrix γ obtained after the first iteration is described in Fig.9 while the corresponding matrix δ is presented in Fig.10.

γ	λ	a	b	c	$Total$
λ	$-$	0.0053	0.0	0.2045	0.2098
a	0.0053	1.9931	0.0	0.0016	2
b	0.6585	0.0016	0.0	1.339	2
c	0.5460	0.0	0.0	0.4540	1

Fig. 9. Matrix γ at the end of the first iteration. The column $Total$ describes the input tree distribution, *i.e.* the number of times each letter of $\mathcal{L}$ is used in the input tree. The value 0.2098 corresponds to the expected number of insertions, according to the current distribution δ. We omit $\gamma(\lambda, \lambda) = \gamma(\#)$ here which has the value 1.0 after the first iteration.

δ	λ	a	b	c
λ	$-$	0.00086	0.0	0.03294
a	0.00086	0.32096	0.0	0.00025
b	0.10604	0.00025	0.0	0.21577
c	0.0879	0.0	0.0	0.0731

Fig. 10. Matrix δ at the end of the first iteration. Note that $\delta(\#) = \delta(\lambda, \lambda) = .16107$.

After 10 iterations, we obtain an optimum described in matrices γ and δ of Fig.11 and 12. We can note that our algorithm has correctly learned one possible target, *i.e.* a symbol c is always deleted, a letter b is always changed into a c, while the symbol a is kept unchanged.

By analyzing the final values of δ, we can be surprised by the behavior of our algorithm that excludes some possible operations. Actually, we can note that at the end of the process, the edit operation corresponding to the deletion of the symbol b has a null probability ($\delta(b, \lambda) = 0$). Such a value could appear amazing since, considering the deepest leaves of the pair of trees, this edit operation seems to be definitely possible. In fact, we can easily explain this behavior: At the end of the first iteration, this edit operation has a non-null probability (0.10604). However, taking into account all the edit operations achieved between the two trees, the substitution (b, c) is more probable, because it is applicable twice (resulting in a probability of 0.21577). Since the iterative process of EM aims to maximize the likelihood of the edit scripts, it explains why the probability $\delta(b, \lambda)$ converges towards 0.

γ	λ	a	b	c
λ	$-$	0.0	0.0	0.0
a	0.0	2.0	0.0	0.0
b	0.0	0.0	0.0	2.0
c	1.0	0.0	0.0	0.0

Fig. 11. Matrix γ after 10 iterations. Note that $\gamma(\#) = \gamma(\lambda, \lambda) = 1$.

δ	λ	a	b	c
λ	$-$	0.0	0.0	0.0
a	0.0	0.3333	0.0	0.0
b	0.0	0.0	0.0	0.3333
c	0.1667	0.0	0.0	0.0

Fig. 12. Matrix δ after 10 iterations. Note that $\delta(\#) = \delta(\lambda, \lambda) = .1667$, this means that the termination symbol is taken into account as the other symbols of the alphabet and appears with a probability of $\frac{1}{6}$. So the values of δ actually sum to 1.

3.2.5 Discussion

Note that the core of the maximization step is the normalization that allows us to learn a joint distribution over edit operations and to define a joint probability $p_\delta(x, y)$. However, in order to use such a model in a classification task (for example for converting a structured document x into another one y), we would need to have the conditional probability $p_\delta(y|x)$ rather than a joint one. Actually, in such a context, the input tree is known and we are looking for the optimal corresponding output one. A simple solution would consist of computing $p_\delta(y|x)$ from the joint distribution such that $p_\delta(y|x) = \frac{p_\delta(x,y)}{p(x)}$. However, this implies a dependence on the input distribution $p(x)$, that generates a bias.

One solution to overcome this drawback consists of directly learning a conditional distribution $p_\delta(y|x)$, usually called a *discriminative model.* The advantage of this approach is to remove the statistical bias of *generative models.* This is the goal of the next section.

4 Learning Stochastic Conditional Tree Edit Distance

Since we aim to learn a discriminative model, we will consider in this section conditional edit operations $(l'_i|l_i)$, of probabillity $\delta(l'_i|l_i)$, where l'_i is a label of the output tree y and l_i a label of the input one x. Then, an edit script will represent in the following a sequence of conditional edit operations. Except this modification of notations, the *forward*, *backward* and *expectation* functions already presented for the joint case remain unchanged in this conditional context. The only one important modification occurs during the maximization step, for which we suggest another normalization of the accumulators under new statistical constraints. This allows us to directly obtain a conditional probability $p_\delta(y|x)$ at each stage of EM.

4.1 New Statistical Constraints

To achieve the learning of a conditional model, we have to draw the new optimization constraints corresponding to this conditional context $p_\delta(y|x)$. In fact, it is possible to model the edit script distribution conditionally to an input tree x in the form of a non deterministic probabilistic finite state automaton. Let us take a simple example to explain this principle. We assume that the input tree $a(b(b), a)$ is the one described in Fig. 13(a). Since we use a breadth-first scanning for computing the ED, it is possible to model the possible edit scripts in the form of the automaton of Fig. 13(b). By using this graphical representation, note that our objective is not to model the tree language of the possible output trees, but rather to express the statistical constraints of our model.

The cycles of each state correspond to the possible insertions before and after the reading of an input symbol. States with a double circle are final states and correspond to the end of the reading of the input tree. Note that in Fig. 13(b) there are three final states. This amazing fact is actually justified by the fact that we are using a Selkow's tree edit distance-based approach. In this case, the deletion of a node implies the removal of its entire corresponding subtree. This explains the path from state 0 to state 2 which begins with the deletion of the tree root a (implying the deletion of the whole tree) and then admitting only insertions. The path beginning from state 1 to state 4 follows the same principle: the deletion of the symbol b (first child of root a) implies the deletion of its unique child b. Then, it only remains to deal with the leaf a before the end of the reading of the input tree. The rest of the automaton is easy to interpret since it follows the reading of the input tree (by excluding deletions of symbols already taken into account by the previously mentioned paths).

Representing the possible edit scripts in this form of an automaton helps us to define the new constraints. Actually, it is well known that to model a statistical distribution, a probabilistic automaton must notably satisfy the following two conditions:

(1) First, probabilities of the outgoing transitions of each state must sum to

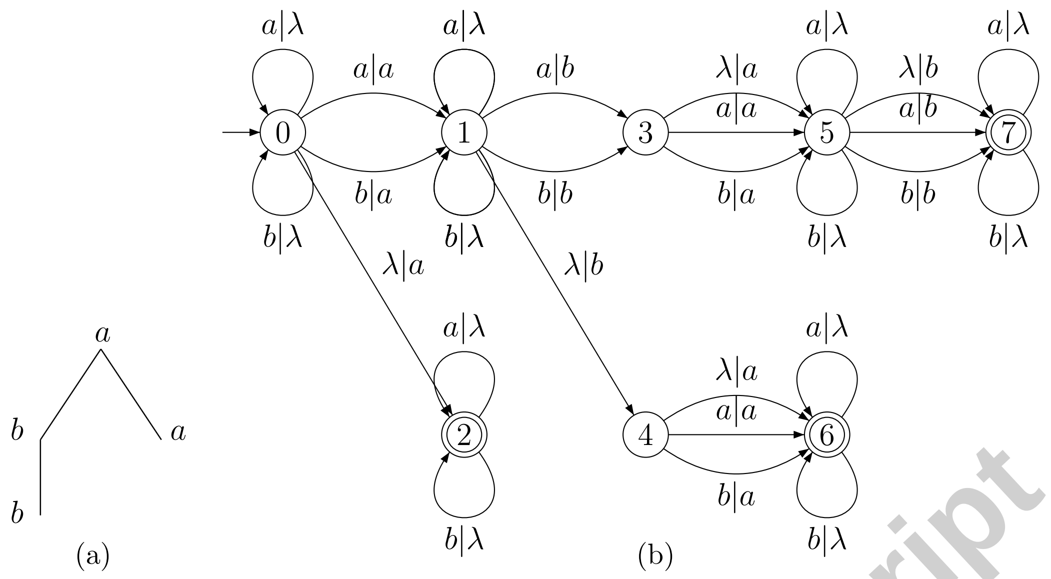


Fig. 13. Output distribution conditionally to an input tree. The transition $\lambda|a$ between states 0 and 2 corresponds to the deletion of the root of the tree $a(b(b), a)$. This deletion implies the removal of all the nodes of the tree. For the sake of clarity, we omit the transitions corresponding to the deletion of the three other nodes. Then, only insertion operations can occur from state 2. The same remark holds for the transition between states 1 and 4 corresponding to the deletion of the two nodes b.

1. More formally,

$$\forall l \in \mathcal{L}, \sum_{l' \in \mathcal{L} \cup \{\lambda\}} \delta(l'|l) + \sum_{l' \in \mathcal{L}} \delta(l'|\lambda) = 1, \qquad (2)$$

(2) Second, probabilities from the final states must also describe a distribution. More formally,

$$\sum_{l' \in \mathcal{L}} \delta(l'|\lambda) + \delta(\#) = 1, \qquad (3)$$

where # is the termination symbol of the tree and $\delta(\#) > 0$.

By fulfilling these two constraints, Appendix A proves that we actually learn a distribution over all the edit scripts definable from a given tree. Despite the fact that this proof is not necessary for showing the optimal normalization that we are going to present, it ensures that we really define a distribution over the possible transformations of an input tree into an output one and so that we have a consistent stochastic model.

It is interesting to note that constraints (2) and (3) remain true whatever the tree edit distance we aim to learn in the form of a stochastic model. Actually, Fig. 13(b) is nothing else but a particular case of Fig. 14 where the deletion of a node does not automatically imply the deletion of its corresponding subtree. In this case, you can note that constraints (2) and (3) remain exactly the same, except that there is only one final state. So, the optimal normalization we are going to present in the next section is directly adaptable to the learning of any tree edit distance. Thus, we claim that the extension of our stochastic approach "only" requires an adaptation of the forward and backward procedures to the specific tree edit distance.

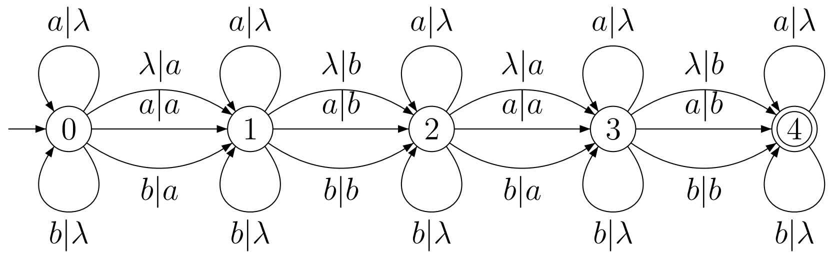


Fig. 14. Output distribution conditionally to an input tree where a node deletion operation is possible without removing the whole subtree.

4.2 Optimal Normalization

The optimal normalization under these new constraints is the solution of an optimization problem as that presented in Dempster et al. [14]. In the following, we show how to obtain it by adapting to trees the principle of the proof presented by Oncina and Sebban in [3] in the case of *string pairs*.

Let $\mathcal{O}$ and $\mathcal{U}$ be two spaces of respectively observable and unobservable data whose distributions are assumed linked by a parameter vector θ. The objective of our algorithm consists of finding the optimal vector θ that maximizes the likelihood function $L(\mathrm{O},\theta) = \ln(p(\mathrm{O}|\theta))$, for a given set $\mathrm{O} \subset \mathcal{O}$ of observed data. Dempster *et al.* [14] showed that it is possible to build, from a given estimate θ_t of θ, a better estimate θ_{t+1} by maximizing the following function:

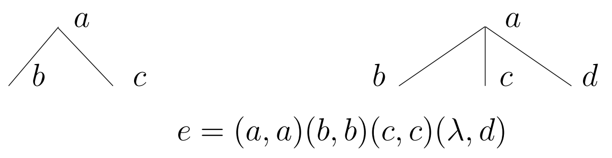
$$Q(\theta_n, \theta_{n+1}) = \mathbf{E}[\ln(p(\mathrm{O}, \mathcal{U}|\theta_{n+1}))|\mathrm{O}, \theta_n]$$

where $\mathbf{E}$ is a conditional expectation over the distribution $\mathcal{U}$.

In our case, the vector θ represents the edit probabilities of the matrix δ. Let us denote $\mathcal{T}(\mathcal{L})$ the set of all labeled trees buildable from the alphabet $\mathcal{L}$. By splitting the learning set of pairs of trees $LS \subset (\mathcal{T}(\mathcal{L}))^2$ into an input set $LS_{in} = \{x : (x, y) \in LS\}$ and an output set $LS_{out} = \{y : (x, y) \in LS\}$, we can rewrite the likelihood function to maximize as follows:

$$L(LS_{out}, \theta, LS_{in}) = \ln(p(LS_{out}|\theta, LS_{in})) = \ln \prod_{(x,y)\in LS} p(y|\theta, x).$$

The transformation of an input tree x into an output one y can be expressed in the form of an edit script $e = e_1...e_n$ of n edit operations. We say that x is the



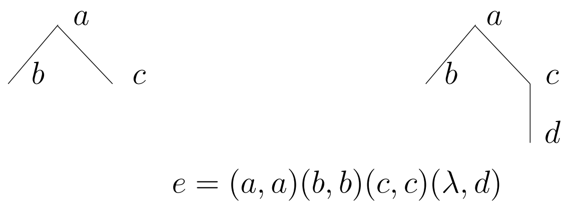


Fig. 15. Example of an edit script e corresponding to two different pairs of trees.

corresponding input tree of e (noted $x = in(e)$) if and only if x is composed of the successive nodes labeled by $l_1 \dots l_n$ according to a breadth-first reading of the tree (where a symbol l_k can be the empty string). Symmetrically, we say that y is the output tree of e (noted $y = out(e)$) if and only if y is composed of the successive nodes labeled by $l'_1 \dots l'_n$.

As we can see in Figure 15, several x and/or y can correspond to a given edit script e. However, since each node could be characterized by its location in the considered tree, we can consider in the following that $in(e)$ and $out(e)$ are in fact unique.

According to an input tree $in(e)$, the probability of a conditional edit script $e = e_1 \cdots e_n$, where $e_i = (l'_i|l_i) \in (\mathcal{L} \cup \{\lambda\})^2 \backslash \{(\lambda, \lambda)\}$, is:

$$\pi_\delta(e|in(e)) = \prod_{i=1}^{n} \delta(l'_i|l_i)\delta(\lambda|\lambda) = \prod_{i=1}^{n} \delta(e_i)\delta(\lambda|\lambda).$$

To each learning pair of trees (x, y), we can associate a set $E(y|x)$ of possible scripts for transforming x into y as:

$$E(y|x) = \left\{ e \in \left((\mathcal{L} \cup \{\lambda\})^2 \right)^* : x = in(e), y = out(e) \right\}.$$

It is easy to see that

$$p(y|x) = \sum_{e \in E(x,y)} \pi_\delta(e|x).$$

Moreover, let us define $E(LS)$ as:

$$E(LS) = \bigcup_{(x,y) \in LS} E(y|x).$$

According to the previous notations, the Q function can be rewritten as follows:

$$\begin{aligned}Q(\theta_n, \theta_{n+1}) &= \mathbf{E}\left[\ln\Big(p(LS_{out}, e|\theta_{t+1}, LS_{in})\Big)|LS_{out}, \theta_t, LS_{in}\right]\\ &= \sum_{e\in\Big((\mathcal{L}\cup\{\lambda\})^2\Big)^*} \pi_\delta(e|LS_{out}, \theta_t, LS_{in}) \ln p(LS_{out}, e|\theta_{t+1}, LS_{in})\end{aligned}$$

as $\pi_\delta(e|y, \theta_t, x) = 0$ if $x \neq in(e)$ or $y \neq out(e)$

$$\begin{aligned}&= \sum_{e\in E(LS)} \pi_\delta\Big(e|out(e), \theta_t, in(e)\Big) \ln \pi_\delta\Big(out(e), e|\theta_{t+1}, in(e)\Big)\\ &= \sum_{e\in E(LS)} \pi_\delta\Big(e|out(e), \theta_t, in(e)\Big) \ln \pi_\delta\Big(e|\theta_{t+1}, in(e)\Big)\\ &= \sum_{e\in E(LS)} \pi_\delta\Big(e|out(e), \theta_t, in(e)\Big)\\ &\quad\left(\sum_{i=0}^{|e|} \ln \delta\Big(out(e_i)|\theta_{t+1}, in(e_i)\Big) + \ln \delta(\lambda|\theta_{t+1}, \lambda)\right)\\ &= \sum_{e_j\in(\mathcal{L}\cup\{\lambda\})^2} \sum_{ee_je'\in E(LS)} \pi_\delta\Big(ee_je'|out(ee_je'), \theta_t, in(ee_je')\Big) \ln \delta(e|\theta_{t+1})\\ &\quad + \sum_{e\in E(LS)} \pi_\delta\Big(e|out(e), \theta_t, in(e)\Big) \ln \delta\Big((\lambda|\lambda)|\theta_{t+1}\Big)\\ &= \sum_{e_j\in(\mathcal{L}\cup\{\lambda\})^2} \gamma(e_j) \ln \delta(e_j|\theta_{t+1}) + |LS| \ln \delta\Big((\lambda|\lambda)|\theta_{t+1}\Big).\end{aligned}$$

Now we have to choose θ_{t+1} that minimizes the $Q(\theta_t, \theta_{t+1})$ function fulfilling the constraints of Equations 2 and 3.

Using the Lagrange Multipliers

$$\begin{aligned}LaMu = &\sum_{e\in(\mathcal{L}\cup\{\lambda\})^2} \gamma(e) \ln \delta(e|\theta_{t+1}) + |LS| \ln \delta\Big((\lambda|\lambda)|\theta_{t+1}\Big)\\ &- \sum_{l\in\mathcal{L}} \mu_l \left(\sum_{l'\in\mathcal{L}} \delta\Big((l'|l)|\theta_{t+1}\Big) + \sum_{l'\in\mathcal{L}} \delta\Big((l'|\lambda)|\theta_{t+1}\Big) + \delta\Big((\lambda|l)|\theta_{t+1}\Big) - 1\right)\\ &- \mu \left(\sum_{l'\in\mathcal{L}} \delta\Big((l'|\lambda)|\theta_{t+1}\Big) + \delta\Big((\lambda|\lambda)|\theta_{t+1}\Big) - 1\right).\end{aligned}$$

Calculating the derivatives and equating to zero results in:

$$\delta\Big((l'|l)|\theta_{t+1}\Big) = \frac{\gamma\Big((l'|l)\Big)}{\mu_l} \qquad \delta\Big((l'|\lambda)|\theta_{t+1}\Big) = \frac{\gamma\Big((l'|\lambda)\Big)}{\sum_l \mu_l + \mu}$$
$$\delta\Big((\lambda|l)|\theta_{t+1}\Big) = \frac{\gamma\Big((\lambda|l)\Big)}{\mu_l} \qquad \delta\Big((\lambda|\lambda)|\theta_{t+1}\Big) = \frac{|LS|}{\mu}.$$

Substituting in the normalization equations (2) and (3) yields:

$$\frac{\sum_{l'} \gamma\Big((l'|\lambda)\Big)}{\sum_l \mu_l + \mu} + \frac{\sum_{l'} \gamma\Big((l'|l)\Big)}{\mu_l} + \frac{\gamma\Big((\lambda|l)\Big)}{\mu_l} = 1, \quad \forall l \in \mathcal{L}$$
$$\frac{\sum_{l'} \gamma\Big((l'|\lambda)\Big)}{\sum_l \mu_l + \mu} + \frac{|LS|}{\mu} = 1$$

where $\delta\Big((\lambda|\lambda)|\theta_{t+1}\Big)$ is equivalent to $\delta(\#)$.

Now, this is a system with $|\mathcal{L}|+1$ equations and $|\mathcal{L}|+1$ unknowns. It is easy to see that

$$\mu = |LS|\frac{N}{N - N(\lambda)} \qquad \mu_l = N(l)\frac{N}{N - N(\lambda)}$$

with

$$N = \sum_{e \in (\mathcal{L})^2} \gamma(e) + |LS| = \sum_{e \in (\mathcal{L}\cup\{\lambda\})^2} \gamma(e)$$

$$N(\lambda) = \sum_{l' \in \mathcal{L}} \gamma\Big((l'|\lambda)\Big) \qquad N(l) = \sum_{l' \in \mathcal{L}\cup\{\lambda\}} \gamma\Big((l'|l)\Big)$$

is a solution to the system. We can now use these parameters in the maximization step presented in Algorithm 7 for learning a conditional distribution.

4.3 Example

As we did for the joint case, we run our algorithm on the simple example already presented in Fig.8. We used the same uniform initialization for matrix δ. Fig.16 and 17 show the results obtained after respectively 1 and 10 iterations. As expected, we can see that the target concept is correctly learned. Actually, when a symbol b is read, it is always changed into a letter c, while when

```
Input: A matrix of accumulators $\gamma$
Output: A matrix of conditional stochastic edit costs $\delta$
$N \leftarrow \sum_{e \in (\mathcal{L} \cup \{\lambda\})^2} \gamma(e)$
$N(\lambda) \leftarrow \sum_{l' \in \mathcal{L}} \gamma(l'|\lambda)$
foreach $l \in \mathcal{L}$ do
    $N(l) \leftarrow \sum_{l' \in \mathcal{L} \cup \{\lambda\}} \gamma(l'|l)$
$\delta(\lambda|\lambda) \leftarrow \frac{N - N(\lambda)}{N}$
foreach $(l, l') \in \mathcal{L}^2$ do
    $\delta(l'|l) \leftarrow \frac{\gamma(l'|l)}{N(l)} \frac{N - N(\lambda)}{N}$
foreach $l \in \mathcal{L}$ do
    $\delta(\lambda|l) \leftarrow \frac{\gamma(\lambda|l)}{N(l)} \frac{N - N(\lambda)}{N}$
foreach $l' \in \mathcal{L}$ do
    $\delta(l'|\lambda) \leftarrow \frac{\gamma(l'|\lambda)}{N}$
```

Algorithm 7: *conditional_maximization*(γ)

the input symbol is a c, it is always removed. The difference with the joint model presented previously is that now, the learned model is independent of the input tree distribution. In other words, whatever the distribution we use to generate the input tree (here, for instance, whatever the number of times the letter b occurs), we will learn the same model. Let us show this behavior in the next experimental section.

δ	λ	a	b	c
λ	–	0.0011	0.0	0.0415
a	0.0028	0.9535	0.0	0.0008
b	0.3198	0.0008	0.0	0.6365
c	0.5255	0.0	0.0	0.4317

Fig. 16. Matrix δ after the first iteration of our EM algorithm. Note that $\delta(\#) = \delta(\lambda|\lambda) = 0.9573$.

δ	λ	a	b	c
λ	–	0.0	0.0	0.0
a	0.0	1.0	0.0	0.0
b	0.0	0.0	0.0	1.0
c	1.0	0.0	0.0	0.0

Fig. 17. Matrix δ after 10 iterations of our EM algorithm. Note that $\delta(\#) = \delta(\lambda|\lambda) = 1.0$.

5 Experiments

5.1 Artificial experiments: Learning a target concept

We carried out experiments to assess the relevance of our two models (joint and conditional) of stochastic tree ED to correctly estimate the parameters of a target model. If we are able to learn them, *i.e.* to converge to the target distribution, this would mean that a learned stochastic tree ED would outperform any classic tree ED with *a priori* hand-tuned costs. Actually, in the best case, these costs would be equivalent to the probabilities of the learned matrix δ.

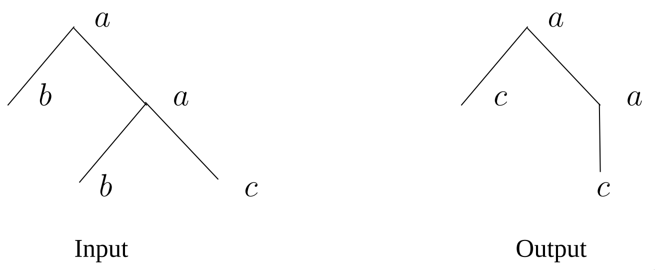
We use the following experimental setup: First, we generate a target distribution defined by a theoretical matrix δ^* (describing either a joint or a conditional distribution). Then, we generate a sample of input trees according to a given input distribution $\mathcal{I}$. This was done by using a target probability for each symbol of the alphabet (here $\mathcal{L} = \{a, b, c\}$), and fixing two parameters limiting the allowed width and depth of the generated trees. To build a learning set LS of pairs of trees, we assign to each input instance an output tree. This one is generated using the input tree and the edit operations described by the target distribution δ^*. Note that such a generation of an output tree according to a target matrix δ^* is a difficult task and probably deserves specific and further investigations from a theoretical standpoint. But this is not the aim of this paper. So, from a practical point of view, to carry out our series of experiments, we simply scan here the input tree with a bottom up analysis, and for each node we apply an edit operation according to the target distribution. Fig.18 shows a simple example explaining the principle of our generation.

Note that in real world applications, these learning pairs of trees could be obtained from a more natural way. For instance, if the aim is to learn a noise model, a pair of trees could represent a couple of (noisy, unnoisy) trees. On the other hand, in a pattern recognition task, such as the one we will tackle in the next section, each pair of trees could be built from an instance and its nearest-neighbor. In this case, the aim would be to learn the possible distortions between two instances of a same class (for example, the different ways to write a digit, or to play a same piece of music).

In this section, the aim is to learn δ^* from LS (constituted from a growing number of pairs of trees) using both of our generative and discriminative models. To assess the effect of the input distribution $\mathcal{I}$ on the learned model, we use different densities to generate the input trees. The performance criterion we use is the normalized distance $dis(\delta, \delta^*)$ between the learned and the target

δ^*	λ	a	b	c
λ	$-$	0.0	0.0	0.0
a	0.0	0.3333	0.0	0.0
b	0.0	0.0	0.0	0.3333
c	0.1667	0.0	0.0	0.0

(a) Matrix δ^* where $\delta(\#) = \delta(\lambda, \lambda) = .1667$



(b) input and output trees

Fig. 18. To build an output tree from an input tree and a joint matrix δ^*, we process the tree bottom up, randomly choosing an edit operation applicable on the considered node. In this simple example, the label b is always changed into a c, while the label c is always removed.

distributions. In the case of a joint model,

$$dis(\delta, \delta^*) = \frac{\sum_{l \in \mathcal{L} \cup \{\lambda\}} \sum_{l' \in \mathcal{L} \cup \{\lambda\}} \left| \delta(l, l') - \delta^*(l, l') \right|}{2}.$$

Normalized in this way, $dis(\delta, \delta^*)$ is a value in the range $[0, 1]$. To do the same thing in the case of a conditional model, we define $dis(\delta, \delta^*)$ as follows:

$$dis(\delta, \delta^*) = \frac{\left(A + B\,|\mathcal{L}|\right)}{2\,|\mathcal{L}|}$$

where

$$A = \sum_{l \in \mathcal{L}} \sum_{l' \in \mathcal{L} \cup \{\lambda\}} \left| \delta(l'|l) - \delta^*(l'|l) \right|$$

and

$$B = \sum_{l' \in \mathcal{L} \cup \{\lambda\}} \left| \delta(l'|\lambda) - \delta^*(l'|\lambda) \right|.$$

In a first series of experiments, we focus on the generative model. In this case, we build two sets of input trees. In order to bring to the fore the bias problem of a generative model, the first one is obtained using the marginal distribution of δ^* which is defined as follows: $\forall l \in \mathcal{L} \cup \{\lambda\}, \delta^*(l) = \sum_{l' \in \mathcal{L} \cup \{\lambda\}} \delta^*(l, l')$. The second one is randomly generated according to a distribution different from the marginal one. The chart of Fig.19 shows the results. As expected, the only way to learn the target requires the use of its marginal distribution to generate the input trees. The use of a non marginal density leads to a bias, characterized by a large distance between the target and the learned model. Does this remark challenge the interest of a generative model? Probably not, even if we can not answer this question in the negative without taking some risks. Actually, we accept the assumption in the Machine Learning community that the learning examples correctly represent the distribution of the unknown theoretical concept. However, the new automatic data acquisition technologies, such as the web, could call nowadays into question this hypothesis [2] . In this new context, directly learning a conditional model could consist of an interesting approach to overcome this problem.

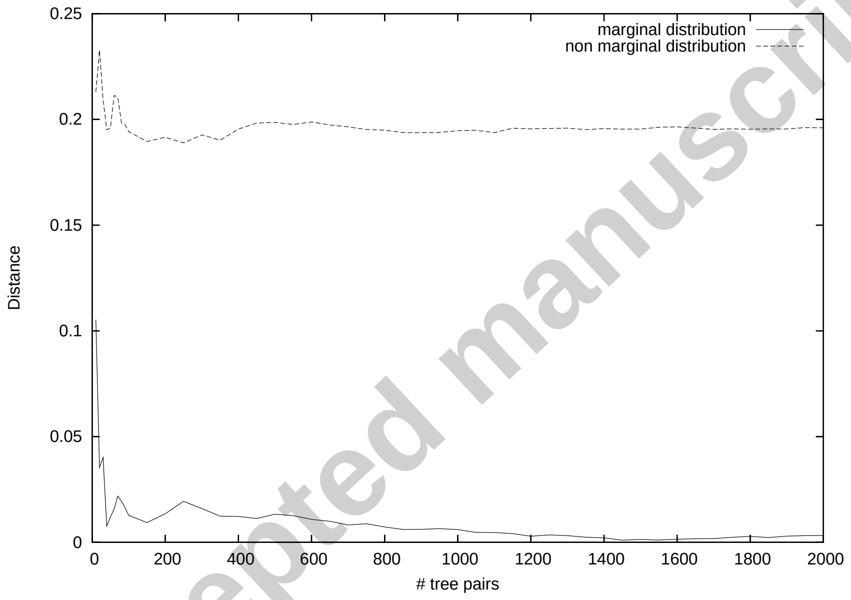


Fig. 19. Results for the joint case.

We use the same experimental setup during the second series of experiments that aims to learn a conditional target model. In this case, we test three different input distributions (among them one is the marginal one). The chart of Fig.20 confirms that whatever the input distribution we use, our discriminative model is able to learn the target model. However, we must make the following remark. It could appear that such a model is requiring more learning pairs of trees to converge. This would be a normal behavior and can be easily explained. The discriminative models are known to have a higher variance than the generative ones [13]. Actually, since they model a conditional distribution, each probability is estimated only on a part of the learning set, leading to a higher variance, which – however – decreases with a growing num-

[2] See for instance the recent Pascal Network Challenge "Learning when test and training inputs have different distributions".

ber of examples. Thus, as mentioned in [13], asymptotically, a discriminative classifier should typically be preferred.

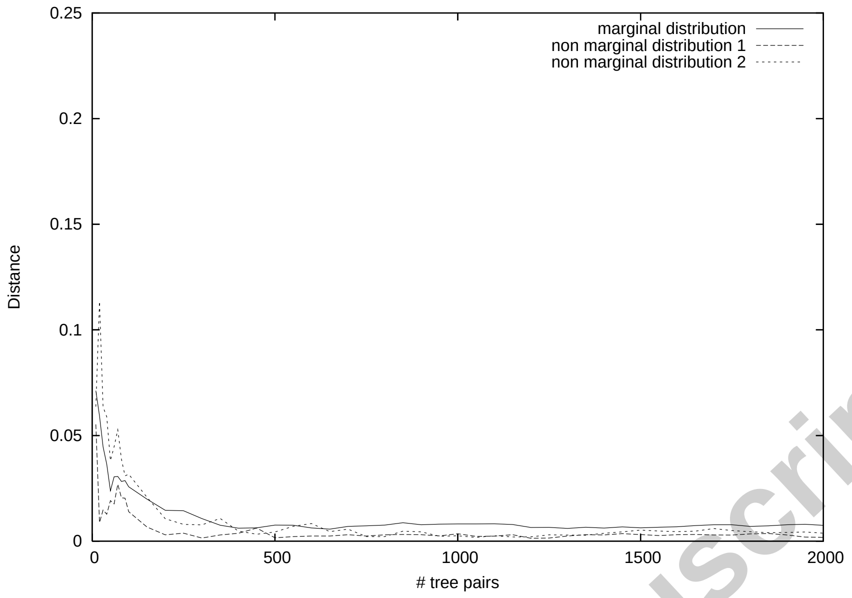


Fig. 20. Results for the conditional case.

5.2 Application on a Handwritten Digit Recognition Task

We showed in the previous section that our discriminative model is able to correctly learn artificial target concepts without any assumption about the input distribution. In this section, we study the behavior of our approach in a real world application and show that a learned ED is always better than a classic tree ED. To achieve this task, we use a part of the well known NIST Special Database 3 of the National Institute of Standards and Technology, describing a set of handwritten characters (letters and digits).

So far, the majority of the learning algorithms that dealt with this pattern recognition problem were numerical approaches. In this series of experiments, a novel way to proceed using a tree-based representation of the handwritten characters is shown. This way enables us to apply our edit distance learning algorithm. However, it is important to underline that our main objective in this experimental study is more to assess the relevance of learning a stochastic tree ED rather than trying to outperform the state of the art methods on this specific task. This explains why we are going to concentrate our efforts particularly on the comparison of our *learned* tree ED with *non learned* tree edit distances. For the reasons mentioned in the previous section, note that we only run in this part our discriminative learning algorithm.

To allow future comparisons of models, we implemented our learning algo-

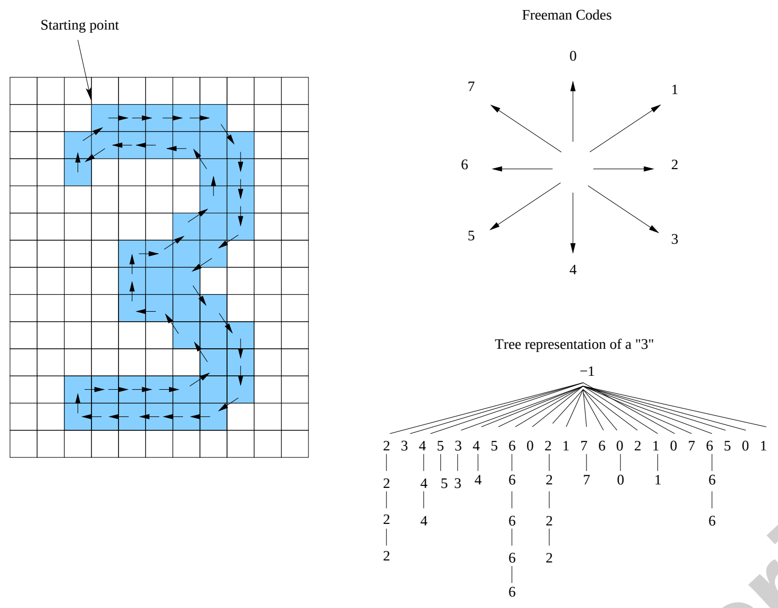


Fig. 21. Example of a tree representation-based digit.

rithms in a JAVA platform, called **SEDiL** for **S**oftware for **E**dit **Di**stance **L**earning[3] . All the results presented in this section have been obtained using this software.

5.2.1 Tree construction from bitmap images

We focus on handwritten digits consisting of 128 $\times$ 128 bitmap images. We use a learning set of about 8000 digits and a test sample of 2000 instances. In order to be able to apply our learning tree ED algorithm, one must handle tree representation-based examples. To do this, we code each bitmap image in the form of a tree built over an octal alphabet. We run the feature extraction algorithm proposed in [16] and that we have adapted to generate trees. To code a given digit, we first build a root labeled by the fictive symbol “-1”. The algorithm then scans the bitmap, left-to-right and starting from the top, until encountering the first point. It follows then the border of the character until it returns to the starting pixel. During this traversal, the algorithm builds the tree using the absolute direction of the next pixel in the border (the so-called Freeman codes) and generating a new child of the root. If the same code is found various times, each repetition becomes a child of the current node. Fig. 21 describes an example on a given “3”. The structured representation of the digit in the form of a tree is presented at the bottom right of the figure.

Even if we are aware that many tree representations are possible for coding a digit, we use this one for the following two reasons:

[3] `http://labh-curien.univ-st-etienne.fr/informatique/SEDiL/`

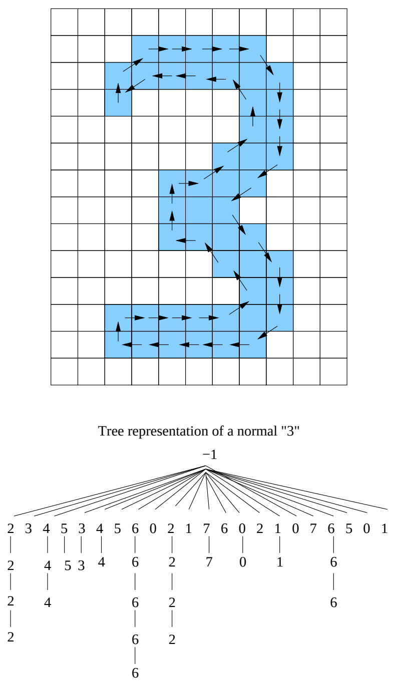


Fig. 22. Interest of a tree representation. To transform the left tree into the right one, the tree edit distance will require only few subtree deletions.

- First, it allows us to keep the structural information of the image.
- Second, the transformation of a digit into a smaller character of the same class only requires simple edit operations with this representation, such as the deletion of a subtree. This remark shows the interest for this pattern recognition task to use a Selkow-based tree edit distance permitting us to remove entire subtrees. For instance, Fig. 22 shows two “3”, a normal and a small one. We can note at the bottom of the figure that they both share the same first level of the tree, and that one needs only some subtree deletions to transform the first tree into the second one.

5.2.2 Experimental comparisons

Our goal is to bring to the fore the interest of learning a tree ED. So, we decided to compare in this series of experiments the behavior of our learned tree ED with the performances of non learned tree EDs. We performed four comparisons:

(1) The first one concerns the comparison with the classic Selkow’s ED algorithm using standard weights for each edit operation. Without any information about the domain, a value of 1 is usually assigned to each basic edit operation. We use such costs in this first comparison.
(2) According to [17], a more relevant strategy would consist of assigning edit operation weights proportionally to the relative angle between the

W_s	0	1	2	3	4	5	6	7
0	0	1	2	3	4	3	2	1
1	1	0	1	2	3	4	3	2
2	2	1	0	1	2	3	4	3
3	3	2	1	0	1	2	3	4
4	4	3	2	1	0	1	2	3
5	3	4	3	2	1	0	1	2
6	2	3	4	3	2	1	0	1
7	1	2	3	4	3	2	1	0

Table 1
Substitution costs W_s according to the relative angle between two directions. The insertion and deletion costs are fixed to 1.

Freeman codes used for describing a digit. To test such a strategy we also used the matrix described in Table 1.

(3) We also performed a comparison with another tree edit distance. Even if our learning algorithm is based on Selkow's approach (for the algorithmic reasons mentioned before), we decided to compare our results with those obtained with Zhang and Shasha's edit distance, that admits the deletion of single nodes. A value of 1 is assigned here to each basic edit operation.
(4) Finally, a last comparison is achieved with the same Zhang and Shasha's algorithm using this time the costs of Table 1.

5.2.3 Constitutions of the set of pairs

As previously mentioned, our method requires a set of (*input,output*) pairs of trees for learning the probabilistic ED. As already illustrated in [18] in the string case, a possible solution consists of building pairs of "similar" examples that describe the possible variations or distortions between instances of each class. Such pairs can be drawn by an expert of the area. In this series of experiments on handwritten digits, we decided rather to automatically build pairs of (*input,output*) trees, where an input is a learning tree of the learning set LS, and the output is a prototype of the input. We used as prototype the corresponding 1-nearest-neighbor in LS of each input. On the one hand, this choice is motivated from an algorithmic standpoint. Actually, with a learning set constituted of $|LS|$ examples, such a strategy does not increase the complexity of the algorithm using $|LS|$ pairs of trees too. On the other hand, by attributing the nearest tree to each character, we ensure that our model is able to learn the main possible distortions between digits in each class.

5.2.4 *Results and Discussion*

The results presented in Fig. 23 have been obtained from different sizes of the learning set, from 50 to 8000 trees. The test accuracy was computed with a test set TS containing always 2000 instances. Each tree of TS has been labeled by the class of its nearest neighbor in LS using one of the considered tree ED algorithms (Selkow, Zhang and Shasha or our EM-based algorithm) and a given matrix of edit costs (for our case, a learned matrix, for the others an a priori fixed one). From Fig. 23, we can make the following remarks.

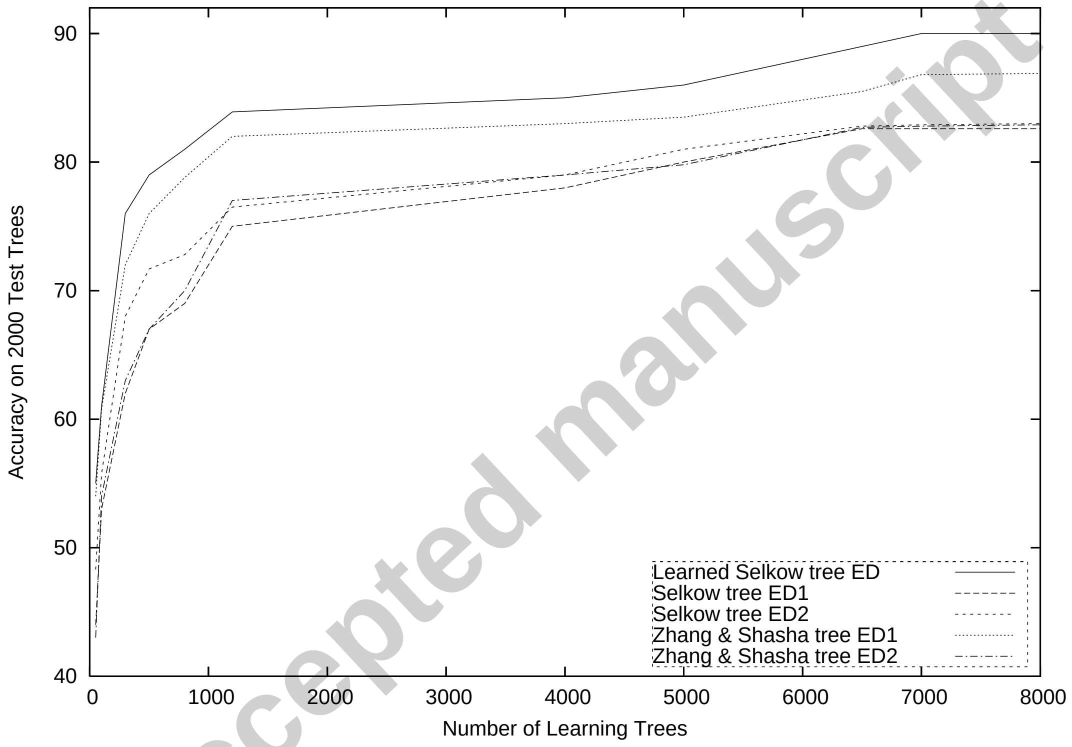


Fig. 23. Results on a handwritten digit recognition task. ED1 uses standard weights (*i.e.* a value of 1 assigned to each operation), while ED2 uses edit costs of Table 1.

First of all, learning a probabilistic tree ED on this pattern recognition task is indisputably more relevant than using standard tree edit distances. Whatever the size of the learning set LS we use, the test accuracy obtained using our learned ED is higher than all the others. Note that using a Student paired t-test, these differences can be shown to be statistically significant.

Second, as already noted in [17], the use of the matrix of costs of Table 1 provides better results than the naive configuration that consists of using the same cost for the three edit operations. This remark is confirmed for both Selkow and Zhang & Shasha distances. However, this is not sufficient to outperform our learned tree ED.

W_s'	0	1	2	3	4	5	6	7
0	0	1.2	2.3	2.9	2.7	2.5	1.8	2.5
1	1	0	2.1	2.8	2.7	2.5	2.2	2.6
2	2.3	2	0	2.5	2.1	2.3	2.3	2.6
3	3.7	2.8	1.7	0	0.2	0.9	2.8	3.9
4	2.6	2.5	2.2	2.5	0	1.2	2	2.9
5	2.7	2.5	2.5	2.7	0.5	0	1.9	2.9
6	2.2	2.3	2.2	2.7	2.2	2.1	0	2.6
7	0.5	0	3.4	5.7	4.7	1.2	0.5	0

Table 2
Substitution costs W_s' learned from 8000 trees by our discriminative algorithm.

Moreover, let us observe the learned edit matrix from the whole set of learning trees (*i.e.* of size 8000) in Table 2. Note that the values presented in this table are edit costs and have been deduced from the edit probabilities learned by our algorithm. We changed them into edit costs to compare this table with edit costs of Table 1 that have been manually tuned to take into account the relative angle between two Freeman codes. Interestingly, we can note that our algorithm has automatically learned almost the same "trend", *i.e.* an edit operation between close codes is less costly than an operation concerning two opposite directions. Despite this similarity between these two matrices, we can note that the test accuracy is significantly in favor of our algorithm. This clearly means that slight modifications on each edit operation can have large consequences on the final performances of the tree ED. This confirms once again the difficulty to manually tune the edit operations in a real world problem and the interest to automatically learn them.

Finally, an interesting remark has to be made by observing the accuracy of Zhang and Shasha's results. Actually, while this algorithm has almost the same performances as Selkow's algorithm using the value 1 for each edit operation, we can note that using the matrix of Table 1, it leads to promising results (even if it does not outperform our learned ED). This clearly means that adapting our EM-based strategy on Shasha's algorithm should lead to better results. However, due to the computational constraints of this algorithm, this requires further investigations.

6 Conclusion

In this paper, we proposed two original approaches for learning a *stochastic tree ED*. This is, as far as we know, the first attempt to learn such a distance specifically adapted to trees. With this paper, we illustrate a first step forward in laying down a theoretical foundation to make tree representation more suitable for real world applications. In the first approach, we modeled this distance as a joint distribution over edit scripts. This model has the advantage of having a small variance but has the drawback to generate a bias on the input distribution. Thus, such a model is suited for dealing with real applications where the instances are not numerous but describing well the underlying distribution. To overcome this drawback, we also proposed to learn a stochastic edit distance from a conditional distribution that allows us to remove this bias. Such a way to proceed is interesting overall when the size of the learning set is sufficiently large, reducing then the variance of such a model. The experimental results on artificial and real world datasets confirm the interest of both approaches.

We think that several perspectives about this work deserve further investigations. First, we plan to extend it to the learning of stochastic models able to take into account edit costs that vary according to the tree context. Actually, the cost of an edit operation can depend on the location where it occurs in the tree, that is not taken into account with our current structures. This implies to learn more complex models, such as stochastic tree transducers or tree conditional random fields. Second, with our proposed method, some edit scripts can not occur, such as those using a single node deletion. This means that we only consider a semi-distribution over the edit scripts, or in other words, a distribution with those scripts having a null probability. This limitation could be overcome by extending our model to other stochastic edit distances, such as the one of Zhang and Shasha which allows the deletion of only one internal node rather than the entire subtree.

Acknowledgments

The authors wish to thank Rémi Gilleron for his many useful remarks that were the basis of a significant improvement of the paper.

A Appendix

We show here that constraints 2 and 3 presented in Section 4.1 enable us to define a conditional distribution over all the possible edit scripts.

Let $e = e_1 \cdots e_n$ an edit script between two trees with n edit operations.

We define the probability of a single edit script as $\pi'_\delta(e) = \delta(e_1) \times \cdots \times \delta(e_n)$. As already defined in Section 3.1, the final probability of the script is evaluated by $\pi_\delta(e) = \pi'_\delta(e) \times \delta(\#)$, where $\delta(\#)$ represents the probability of termination of the script. Note that the probability $\pi'_\delta(\epsilon)$ of the empty edit script ϵ is 1.

Let x be a non empty tree of $\mathcal{T}(\mathcal{L})$ (the set of all labeled trees buildable from the alphabet $\mathcal{L}$) with its sequential representation $l_1 \dots l_n$ corresponding to the labels of its nodes according to a breadth-first reading. In the following, we show that if conditions 2 and 3 are fulfilled one can define a distribution over the whole set $E_{in}(l_1 \dots l_n)$ of edit scripts applicable to transform the input tree x into an output one, such that:

$$\sum_{e \in E_{in}(l_1 \dots l_n)} \pi_\delta(e) = 1.$$

For the sake of convenience, we will also denote $E_{in}(l_1 \dots l_n)$ by $E_{in}(x)$. Let us define $E_{in}(l_1 \dots l_n)^+ = E_{in}(x)^+$ as the set of edit scripts with at least one (non empty) edit operation.

First, let us consider the case when $x = \lambda$

$$\begin{aligned}
\sum_{e \in E_{in}(\lambda)} \pi'_\delta(e) &= 1 + \sum_{e \in E^+_{in}(\lambda)} \pi'_\delta(e) \\
&= 1 + \sum_{e'e \in E^+_{in}(\lambda)} \pi'_\delta(e'e) \\
&= 1 + \sum_{e'e \in E^+_{in}(\lambda)} \delta(l'|\lambda)\pi'_\delta(e) \text{ with } e' = (l'|\lambda) \\
&= 1 + \sum_{l' \in \mathcal{L}} \delta(l'|\lambda) \sum_{e \in E_{in}(\lambda)} \pi'_\delta(e).
\end{aligned}$$

Then,

$$\sum_{e \in E_{in}(\lambda)} \pi'_\delta(e) \left(1 - \sum_{l' \in \mathcal{L}} \delta(l'|\lambda)\right) = 1 \text{ and so}$$

$$\sum_{e \in E_{in}(\lambda)} \pi'_\delta(e) = \left(1 - \sum_{l' \in \mathcal{L}} \delta(l'|\lambda)\right)^{-1}.$$

Let us now consider the complete case. Let x be a non empty tree, with its sequential representation $l_1 \dots l_n$ corresponding to the labels of its nodes according to a breadth-first reading and let x_2 be the sequence of symbols such that $x_2 = l_2 \dots l_n$. Let $E^N_{in}(x)$ be the set of edit scripts such that the output tree can not be the empty tree.

$$\begin{aligned}\sum_{e\in E_{in}(l_1x_2)} \pi'_\delta(e) &= \sum_{e\in E(\lambda|l_1x_2)} \pi'_\delta(e) + \sum_{(l'|l)e\in E^N_{in}(l_1x_2)} \pi'_\delta((l'|l)e) \\ &= \sum_{e\in E(\lambda|l_1x_2)} \pi'_\delta(e) + \sum_{(l'|l)e\in E^N_{in}(l_1x_2)} \delta(l'|l)\pi'_\delta(e) \\ &= \sum_{e\in E(\lambda|l_1x_2)} \pi'_\delta(e) + \sum_{l'\in\mathcal{L}} \delta(l'|l_1) \sum_{e\in E_{in}(x_2)} \pi'_\delta(e) + \\ &\quad \sum_{l'\in\mathcal{L}} \delta(l'|\lambda) \sum_{e\in E_{in}(l_1x_2)} \pi'_\delta(e) + \delta(\lambda|l_1) \sum_{e\in E^N_{in}(x_2)} \pi'_\delta(e) \\ &= \delta(\lambda|l_1) \sum_{e\in E_{in}(x_2)} \pi'_\delta(e) + \sum_{l'\in\mathcal{L}} \delta(l'|l_1) \sum_{e\in E_{in}(x_2)} \pi'_\delta(e) + \sum_{l'\in\mathcal{L}} \delta(l'|\lambda) \sum_{e\in E_{in}(l_1x_2)} \pi'_\delta(e) \\ &= \left(\delta(\lambda|l_1) + \sum_{l'\in\mathcal{L}} \delta(l'|l_1)\right) \sum_{e\in E_{in}(x_2)} \pi'_\delta(e) + \sum_{l'\in\mathcal{L}} \delta(l'|\lambda) \sum_{e\in E_{in}(l_1x_2)} \pi'_\delta(e).\end{aligned}$$

Then

$$\sum_{e\in E_{in}(l_1x_2)} \pi'_\delta(e) \left(1 - \sum_{l'\in\mathcal{L}} \delta(l'|\lambda)\right) = \left(\delta(\lambda|l_1) + \sum_{l'\in\mathcal{L}} \delta(l'|l_1)\right) \sum_{e\in E_{in}(x_2)} \pi'_\delta(e)$$

and

$$\sum_{e\in E_{in}(l_1x_2)} \pi'_\delta(e) = \left(1 - \sum_{l'\in\mathcal{L}} \delta(l'|\lambda)\right)^{-1} \left(\delta(\lambda|l_1) + \sum_{l'\in\mathcal{L}} \delta(l'|l_1)\right) \sum_{e\in E_{in}(x_2)} \pi'_\delta(e).$$

Applying this equation recursively on the size of x and taking into account that the base case is

$$\sum_{e\in E_{in}(\lambda)} \pi'_\delta(e) = \left(1 - \sum_{l'\in\mathcal{L}} \delta(l'|\lambda)\right)^{-1},$$

we have

$$\sum_{e\in E_{in}(l_1\dots l_n)} \pi'_\delta(e) = \prod_{i=1}^{n} \left[\left(1 - \sum_{l'\in\mathcal{L}} \delta(l'|\lambda)\right)^{-1} \left(\delta(\lambda|l_i) + \sum_{l'\in\mathcal{L}} \delta(l'|l_i)\right)\right] \times \left(1 - \sum_{l'\in\mathcal{L}} \delta(l'|\lambda)\right)^{-1}$$

and

$$\sum_{e \in E_{in}(l_1 \ldots l_n)} \pi_\delta(e) = \prod_{i=1}^{n} \left[\left(1 - \sum_{l' \in \mathcal{L}} \delta(l'|\lambda)\right)^{-1} \left(\delta(\lambda|l_i) + \sum_{l' \in \mathcal{L}} \delta(l'|l_i)\right) \right] \times \left(1 - \sum_{l' \in \mathcal{L}} \delta(l'|\lambda)\right)^{-1} \times \delta(\#).$$

A sufficient condition for $\sum_{e \in E_{in}(l_1 \ldots l_n)} \pi_\delta(e) = 1$ is that each element of the terms that appear in the product is equal to 1 and that the final product is also 1. Then, concerning the first part of the right-hand side,

$$\begin{aligned}
&\left(1 - \sum_{l' \in \mathcal{L}} \delta(l'|\lambda)\right)^{-1} \left(\delta(\lambda|l_i) + \sum_{l' \in \mathcal{L}} \delta(l'|l_i)\right) = 1 \\
&1 - \sum_{l' \in \mathcal{L}} \delta(l'|\lambda) = \delta(\lambda|l_i) + \sum_{l' \in \mathcal{L}} \delta(l'|l_i) \\
&\sum_{l' \in \mathcal{L}} \delta(l'|\lambda) + \delta(\lambda|l_i) + \sum_{l' \in \mathcal{L}} \delta(l'|l_i) = 1 \\
&\sum_{l' \in \mathcal{L}} \delta(l'|\lambda) + \sum_{l' \in \mathcal{L} \cup \{\lambda\}} \delta(l'|l_i) = 1 \\
&\sum_{l' \in \mathcal{L}} \delta(l'|\lambda) + \sum_{l' \in \mathcal{L} \cup \{\lambda\}} \delta(l'|l) = 1
\end{aligned}$$

that gives Constraint 2. Now, considering the second part of the right-hand side,

$$\begin{aligned}
&\left(1 - \sum_{l' \in \mathcal{L}} \delta(l'|\lambda)\right)^{-1} \times \delta(\#) = 1 \\
&1 - \sum_{l' \in \mathcal{L}} \delta(l'|\lambda) = \delta(\#) \\
&\sum_{l' \in \mathcal{L}} \delta(l'|\lambda) + \delta(\#) = 1
\end{aligned}$$

and we have Constraint 3.

Note that these equations are not valid if $\sum_{l' \in \mathcal{L}} \delta(l'|\lambda) = 1$, but this is impossible since $\delta(\#) > 0$.

References

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

[2] S. Ristad, P. Yianilos, Learning string-edit distance, IEEE Transactions on Pattern Analysis and Machine Intelligence 20 (5) (1998) 522–532.

[3] J. Oncina, M. Sebban, Learning stochastic edit distance: application in handwritten character recognition, Journal of Pattern Recognition 39 (9) (2006) 1575–1587.

[4] A. McCallum, K. Bellare, P. Pereira, A conditional random field for disciminatively-trained finite-state sting edit distance, in: Proceedings of the 21th Conference on Uncertainty in Artificial Intelligence (UAI'2005), 2005, pp. 388–400.

[5] R. Durbin, S. Eddy, A. Krogh, G. Mitchison, Biological sequence analysis, Cambridge University Press, 1998.

[6] M. Neuhaus, H. Bunke, A probabilistic approach to learning costs for graph edit distance, in: 17th Int. Conf. on Pattern Recognition, IEEE, 2004, pp. 389–393.

[7] M. Bernard, A. Habrard, M. Sebban, Learning stochastic tree edit distance, in: European Conference on Machine Learning (ECML), Vol. 4212 of LNCS, Springer, 2006, pp. 42–53.

[8] K. Zhang, D. Shasha, Simple fast algorithms for the editing distance between trees and related problems, SIAM Journal of Computing (1989) 1245–1262.

[9] P. Klein, Computing the edit-distance between unrooted ordered trees, in: Proc. of the 6th European Symposium on Algorithms (ESA), Springer, 1998, pp. 91–102.

[10] S. Dulucq, L. Tichit, RNA secondary structure comparison: exact analysis of the Zhang-Shasha tree edit algorithm, Theoretical Computer Science 306 (1-3) (2003) 471–484.

[11] S. Dulucq, H. Touzet, Decomposition algorithms for the tree edit distance problem, Journal of Discrete Algorithms 3 (2-4) (2005) 448–471.

[12] S. Selkow, The tree-to-tree editing problem, Information Processing Letters 6 (6) (1977) 184–186.

[13] G. Bouchard, B. Triggs, The trade-off between generative and discriminative classifiers., in: Proc. of the 16th IASC International Symposium on Computational Statistics (COMPSTAT'2004), Springer, 2004, pp. 697–704.

[14] A. Dempster, M. Laird, D. Rubin, Maximum likelihood from incomplete data via the EM algorithm, J. R. Stat. Soc B (39) (1977) 1–38.

[15] P. Kilpelainen, Tree matching problems with applications to structured text databases, Ph.D. thesis, Dept. of Computer Science, University of Helsinki (1992).

[16] E. Gómez, L. Micó, J. Oncina, Testing the linear approximating eliminating search algorithm in handwritten character recognition tasks, in: VI Symposium Nacional de reconocimiento de Formas y Análisis de Imágenes, 1995, pp. 212–217.

[17] L. Micó, J. Oncina, Comparison of fast nearest neighbour classifiers for handwritten character recognition, Pattern Recognition Letters 19 (1998) 351–356.

[18] E. S. Ristad, P. N. Yianilos, Learning string-edit distance, IEEE Trans. Pattern Anal. Mach. Intell. 20 (5) (1998) 522–532.

Author Biography

Marc Bernard was born in France in 1971. He received his Ph.D in Computer Science from the University of Burgundy in 1998. Since 1999, he has been assistant professor within the Faculty of Science at the University of Saint-Etienne. His current research interests include: machine learning and grammatical inference.

Laurent Boyer was born in France in 1984. He is phD student in Computer Science at the University of Saint-Etienne. His current research interests include: machine learning, and more particularly similarity learning.

Amaury Habrard was born in France in 1978. He received his Ph.D in Computer Science from the University of Saint-Etienne in 2004. He has been assistant professor at the University of Marseille for two years. His current research interests include: computational learning theory and grammatical inference.

Marc Sebban was born in Lyon, France in 1969. He received his Ph.D in Computer Science from the University of Lyon I in 1996. Since 2001, he has been full professor within the Faculty of Science at the University of Saint-Etienne, and since 2006, the vice-director of the Hubert Curien laboratory (UMR CNRS 5516). His current research interests include: machine learning, boosting and grammatical inference.