Adaptive structure metrics for automated feedback provision in intelligent tutoring systems

Benjamin Paassen *, Bassam Mokbel and Barbara Hammer

CITEC centre of excellence
Bielefeld University - Germany

(This is a preprint of the publication [1], as provided by the authors.)

Abstract

Typical intelligent tutoring systems rely on detailed domain-knowledge which is hard to obtain and difficult to encode. As a data-driven alternative to explicit domain-knowledge, one can present learners with feedback based on similar existing solutions from a set of stored examples. At the heart of such a data-driven approach is the notion of similarity. We present a general-purpose framework to construct structure metrics on sequential data and to adapt those metrics using machine learning techniques. We demonstrate that metric adaptation improves the classification of wrong versus correct learner attempts in a simulated data set from sports training, and the classification of the underlying learner strategy in a real Java programming dataset.

1 Introduction

Intelligent tutoring systems (ITSs) have made great strides in recent years; they offer the promise of individual one-on-one computer based support in the context of scarce human resources, as it is common in massive open online courses (MOOCs), for example [2]. However, researchers have reported 100 - 1,000 hours of authoring time for one hour of instructions in ITSs [3]; in addition, ITSs usually require an underlying domain theory such that their applicability is limited in areas where problems and their solution strategies are not easy to formalize [4, 5]. In such domains, data-driven approaches are possible, providing feedback based on a set of existing examples for (correct) solutions of the underlying task [5, 6]: If the students requires a hint on how to change her attempt to get closer to a correct solution, it can be compared

*Corresponding Author

to a similar example from the set, and the dissimilarities between her attempt and the example can be contrasted or highlighted in order to help the student to improve her own solution [7, 8, 9].

As key ingredients such techniques require data and a suitable metric based on which to compare solutions. More specifically, a suitable metric has to meet at least three requirements in order to be suitable: (A) Solutions are typically non-vectorial. Instead, they are given as *structured data*, that is: as sequences, trees or graphs. Therefore, *structure metrics* have to be used that need to fit the given domain. (B) Feedback should be given based on examples that implement the same underlying strategy. Therefore the metric should emphasize differences in strategy, while being insensitive to differences in style across students. This corresponds to the choice of the metric as well as the choice of parameters for the metric. (C) In order to provide helpful feedback, the metric should be interpretable, in the sense that it should be possible to retrieve the parts of both solutions, which differ from each other.

In this contribution, we focus on alignment metrics for sequential data: Recently it has been shown that such metrics can be expressed in terms of a general framework, called algebraic dynamic programming (ADP) [10], which addresses the first requirement (A). Further, we show in this work that all alignment algorithms expressed in that framework can be systematically adapted, as required (B). Finally, all these alignment algorithms allow to retrieve detailed information which parts of both input solutions are similar and which are not: Alignment algorithms match similar parts of both solutions and identify parts which can not be matched, thereby providing interpretable and actionable knowledge for feedback (requirement C), see e.g. [9].

1.1 Contribution and Overview

The main contributions of this paper are the following: First, we show that the general framework of *algebraic dynamic programming* (ADP) enables us to express a broad class of structure metrics, namely alignment distances. Exemplary, we use ADP to express four alignment algorithms: Global sequence alignment, affine sequence alignment, dynamic time warping and the Sakoe-Chiba approximation of dynamic time warping. Second, we demonstrate that gradients on alignment distances can be calculated efficiently using ADP. Third, we use the calculated gradients for *structure metric learning*: We adapt metric parameters to improve the classification accuracy of a *relational generalized learning vector quantization* (RGLVQ) classifier. Finally, we apply the structure metric learning scheme using four different alignment algorithms on two different datasets from the domain of intelligent tutoring systems.

Note, that the techniques presented in this work are by no means limited to intelligent tutoring systems but can be applied in all settings, where metrics on sequential data are required and should be adapted to optimize some (differentiable) cost function (see e.g. [11] for an example from the biomedical domain).

The outline of this paper is as follows: In Section 2 we discuss related work,

in particular data-driven intelligent tutoring systems (ITSs), similarity-based machine learning, structure metrics and structure metric learning. We also discuss our choice of datasets in the context of existing literature on ITSs. In Section 3, we introduce a simplified version of ADP as generalization of alignment algorithms and use it to express four example metrics, which we evaluate in the experiments later on. We explain metric learning on ADP alignment algorithms using RGLVQ in Section 4. Finally, we report our experiments in Section 5.

2 Related Work

This research connects several, seemingly disconnected fields, such as artificial intelligence in education, educational data mining, classic machine learning, structure metrics, metric learning and formal languages. In this section, we provide an overview of these different connections and also embed our own work in the context of the existing literature.

2.1 Data-Driven Intelligent Tutoring Systems

Intelligent tutoring systems (ITSs) are systems to enhance student learning via artificial intelligence methods. Most of the time, students proceed through a curriculum of different tasks to obtain skills and knowledge. The systems job is to select the next task depending on the current level of knowledge and individual parameters of the student (*outer loop*) and to support her solving the current task (*inner loop*) [5]. Such systems have been successfully applied in many contexts, especially in learning logic and math concepts, and have been proven to lead to positive learning outcomes for students [12, 6].

However, they usually rely on extensive knowledge engineering to formalize domain concepts and explicitly track student knowledge, which is both costly and difficult, especially in domains where explicit and detailed knowledge about the domain can not be obtained (so-called *ill-defined domains*) [3, 4, 5]. To relieve ITS engineers from the burden of knowledge engineering, data-driven approaches have emerged. Such approaches try to replace pre-defined and explicit domain knowledge by inference based on example-data of students interacting with the system [5, 6].

Here, we focus on the *inner loop* mentioned before: To support students in solving a task, utilizing only example solution attempts handed in by other students. An intuitive solution to this problem is to base student support on a notion of *similarity* to existing solutions: We can approximate a student model by considering the similarity of her solution to all solutions in the example set. A hint to improve her solution can be based on the difference between her solution and a similar (but better) solution [7, 8, 9].

Further, a proper similarity measure enables ITS engineers to apply machine learning techniques for further problems: One can try to detect outliers or buggy solutions, one can estimate the quality of solutions based on the known quality

for some examples (regression) and one can cluster or classify solutions into discrete, meaningful sets. In our experiments we focus on the latter and distinguish between correct and wrong executions of a sports exercise (see Section 5.1) and between the underlying algorithms of computer programs (see Section 5.2).

Such an approach requires a proper similarity measure (that is: a metric) as key ingredient. Note, that most common similarity measures, such as the Euclidean distance or the radial basis function kernel, are based on a vectorial data representation. While first approaches exist to transform student data into a vectorial format, most data is still only available as structured data, such as sequences, trees or graphs [13]. Thus, we face a three-fold challenge: Constructing a similarity measure that works on the available data in the first place, adjusting this similarity measure to be apt for the task at hand and utilizing the similarity measure to generate actionable knowledge for an ITS. The latter is the general topic of similarity-based machine learning, the former two refer to structure metrics and (structure) metric learning respectively.

2.2 Similarity-Based Machine Learning

From early on, machine learning methods based on similarity measures have been utilized, starting with simple schemes like k-nearest neighbor classification [14] or k-means clustering [15]. The general rationale is that data which are similar to each other in some respect may be similar in other respects as well. Research on similarity-based machine learning has flourished in recent years, mainly driven by the development of powerful kernel-approaches, and includes such popular methods such as the Support Vector Machine, extended nearest neighbor-schemes and Gaussian process regression [16, 17].

Here, we require a method which lends itself to gradient-based optimization. Gradient-based schemes in similarity-based machine learning have been applied successfully in the case of relational learning vector quantization (RGLVQ) [18], which we describe in more detail in Section 4.

Note, however, that the focus of this work is not so much on demonstrating the capabilities of methods based on an existing similarity measure (here, the interested reader is referred to the literature cited above), but rather how to obtain a proper (structure) similarity measure in the first place.

2.3 Structure Metrics

Over the years, multiple structure metrics have been suggested, reaching from sequential data over trees to graphs, see e.g. [19] for a recent review. Kernel-approaches have been especially popular, such as the diffusion or convolution kernel approach [20, 21]. Unfortunately, most of these approaches can not directly deal with rich data attached to the graph nodes and/or are runtime-inefficient.

In this contribution, we focus on sequential data, where we can rely on the abundant work on *edit* or *alignment distances*. Such methods extend both input sequences, such that similar elements are *aligned*. They have been successfully

applied in diverse domains, such as automatic spell-checking [22, 23], bioinformatics [24, 25, 26] and speech processing [27]. All of those alignment distances can be efficiently calculated using dynamic programming with a worst-case runtime of $\mathcal{O}(M \cdot N)$, with M and N being the number of sequence elements in the first and the second input sequence respectively.

Given the abundance of alignment algorithms in the literature, we can select a suitable one for our data: For motion data, dynamic time warping is a well established technique [27], accompanied even by techniques to make it a linear-time algorithm [28]. For comparing syntactic building blocks, however, classic edit distance approaches like [22, 24, 26] are more common. Of course, it would be tedious (and error-prone) to implement every possible alignment algorithm by hand. Fortunately, Giegerich and colleagues recently proposed *algebraic dynamic programming* (ADP) as a generalization of dynamic programming over sequential data [10]. Thus it is possible to summarize a broad class of alignment algorithms, including the ones mentioned before, in a common framework (see Section 3).

2.4 Structure Metric Learning

In many scenarios, a-priori assumptions regarding the parameters of a structure metric might be unavailable or wrong. In bioinformatics in particular, this topic has been investigated under the term *inverse alignment problem*, which has been addressed by linear programming methods to find metric parameters that generate desired, pre-defined alignments [29]. However, such approaches require detailed knowledge regarding (at least a few) desired optimal alignments, which are unknown in our educational setting.

Inferring metric parameters for classification tasks, using label information only, is the topic of *metric learning* in the machine learning community. Popular and powerful algorithms have been designed for vectorial data, see e.g. [30, 31, 32]. However, only few approaches exist for structure metrics and alignment distances in particular, making it a novel and challenging field of machine learning research [30, 33, 11].

We build on the work of [11] and optimize metric parameters with respect to the cost function of a popular machine learning classifier, *learning vector quantization*. Thereby, metric parameter learning turns into a nonlinear optimization problem, which we solve via gradient descent. Here, the ADP framework enables us to efficiently compute alignment distances and their gradients w.r.t. the metric parameters for arbitrary alignment algorithms, as we demonstrate in Section 4.

2.5 Experimental data

In this paper, we investigate how to efficiently obtain a proper metric for two example datasets (see Section 5): In sports training, students need to learn how to execute a certain motion or exercise. Virtual reality-based systems have been applied in this domain, to contrast a students movement with the correct

execution done by a 3D virtual trainer [34][1]. A natural representation of human motion is a sequence of frames, where each frame contains the position and/or rotation of the body parts (e.g. the position of the hands, the feet, the head, etc. in three-dimensional space)[2]. However, to distinguish between correct and wrong executions of an exercise, only few features matter (e.g. the position of the arms is hardly relevant when executing a squat). We simulate executions of a sports exercise by generating 10-dimensional sequences via *dynamical movement primitives* [35], where the motion in 9 dimension is irrelevant for the distinction between correct and wrong executions, while the first dimension matters (see Section 5.1).

Our second dataset is a benchmark dataset from [11], consisting of 64 Java programs sorting an input array of integers in ascending order. Here, we consider only correct solutions and investigate a different aspect of data-driven tutoring systems: Oftentimes, there are many correct solutions for a given programming problem. If a student applies a certain solution strategy, feedback based on an example with a different underlying strategy might be confusing. Therefore, one would like to obtain a metric that is sensitive do differences in the underlying strategy. In our example dataset, we consider the classification of *BubbleSort* implementations versus *InsertionSort* implementations (see Section 5.2).

3 Algebraic Dynamic Programming

In this section we introduce the notion of sequence, alignment and alignment distance more formally. We formalize the problem an alignment algorithm has to solve and introduce a simplified version of *algebraic dynamic programming* (ADP) as a generalized solution for this problem.

Definition 1. Let $\{\Sigma_\kappa\}_{\kappa=1,\dots,K}, K \in \mathbb{N}$ be arbitrary sets. We define a *sequence* $\bar{x}$ as a succession of elements $x \in \Sigma_\times$, where:

$$\Sigma_\times := \bigtimes_{\kappa=1}^{K} \Sigma_\kappa \tag{1}$$

That means: We consider sequences with multidimensional elements, where the entries in each dimension stem from different, arbitrary sets.

Definition 2. Let $\bar{x}$ and $\bar{y}$ be sequences. An *alignment* is defined as two extensions $\bar{x}^*$ and $\bar{y}^*$ of those sequences that have the same number of elements. Extensions should always contain the elements of the original sequences and may contain additional elements in between. The permitted additional elements differ between alignment algorithms.

Let d be a metric on $\Sigma_\times \cup \{-\}$ and let $(\bar{x}^*, \bar{y}^*)$ be an alignment of the sequences $\bar{x}$, $\bar{y}$. Let $|\bar{x}^*|$ be the number of elements of $\bar{x}^*$. We define the *cost* of

[1] Another project in this regard can be found at `https://www.cit-ec.de/de/content/intelligent-coaching-space`

[2] see e.g. `http://mocap.cs.cmu.edu/`,

$(\bar{x}^*, \bar{y}^*)$ as:

$$\mathcal{F}(\bar{x}^*, \bar{y}^*) := \sum_{i=1}^{|\bar{x}^*|} d(\bar{x}_i^*, \bar{y}_i^*) \tag{2}$$

The alignment $(\bar{x}^*, \bar{y}^*)$ is called *optimal* iff

$$\mathcal{F}(\bar{x}^*, \bar{y}^*) = \min\{\mathcal{F}(\bar{x}^*, \bar{y}^*) | (\bar{x}^*, \bar{y}^*) \text{ is an alignment of } (\bar{x}, \bar{y})\} \tag{3}$$

The cost of an optimal alignment is called the *alignment distance* $D(\bar{x}, \bar{y})$.

As an example, consider the two character sequences $\bar{x} = \text{ac}$ and $\bar{y} = \text{bc}$. Possible alignments include:

$$(\text{ac}, \text{bc}), (-\text{ac}, \text{a}-\text{c}), (\text{ac}-, -\text{bc}) \tag{4}$$

In the first alignment, we do not insert additional elements in any of the sequences. In the second alignment, we add a $-$ in the front of $\bar{x}$ and in the middle of $\bar{y}$ and in the last example we add a ba in front of $\bar{x}$ and a aa in the middle of $\bar{y}$. Assume that we apply a simple metric d of the form:

$$d(a, b) := 1 - \delta(a, b) \tag{5}$$

where δ is the Kronecker-Delta, defined as

$$\delta(a, b) := \begin{cases} 1 & \text{if } a = b \\ 0 & \text{if } a \neq b \end{cases} \tag{6}$$

Then, the costs of the alignments are:

$$\mathcal{F}(\text{ac}, \text{bc}) = 1, \mathcal{F}(-\text{ac}, \text{a}-\text{c}) = 2, \mathcal{F}(\text{ac}-, -\text{bc}) = 3 \tag{7}$$

which would make the first one the *optimal alignment*.

Obviously, the set of possible alignments grows exponentially with increasing length of the input sequences. However, the optimal alignment can still be found in polynomial time via *dynamic programming*. The first algorithm of that kind to be discovered is the *classic edit distance*: If extensions may only contain gap symbols $(-)$ as additional elements, the alignment distance for the two sequences $\bar{x}$ and $\bar{y}$ with lengths M and N respectively can be expressed in terms of a Bellman equation [22, 24, 36]:

$$D(M+1, N+1) := 0 \tag{8}$$

$$D(i, N+1) := M + 1 - i \tag{9}$$

$$D(M+1, j) := N + 1 - j \tag{10}$$

$$\begin{aligned} D(i, j) := \min\{&D(i+1, j+1) + d(x_i, y_j), \\ &D(i+1, j) + d(x_i, -), \\ &D(i, j+1) + d(-, y_j)\} \end{aligned} \tag{11}$$

$$d(a, b) := 1 - \delta(a, b) \tag{12}$$

This can be solved in $\mathcal{O}(M \cdot N)$, if one creates a table for D of size $(M+1) \times (N+1)$ and iteratively calculates the entries in two nested loops starting at $D(M+1, N+1)$ and ending at $D(1,1)$, which then contains the alignment distance.

Accordingly, we define the more general alignment problem as finding the (cost of the) optimal alignment for two input sequences $\bar{x}$ and $\bar{y}$, given the metric d in polynomial time, more specifically in $\mathcal{O}(M \cdot N)$. As history has shown, this problem is not a trivial one and solving it required quite a bit of skill and ingenuity for each variation of the original alignment scheme. Few very successful algorithms were carefully crafted over the past decades, most notably in bioinformatics [24, 25, 26] and speech processing [27].

It was only in 2004, that Giegerich and colleagues proposed a much more general framework for dynamic programming over sequential data, in the form of *algebraic dynamic programming* (ADP) [10]. This framework provides a clear separation of concepts: First, users have to define the *operations* permitted on both input sequences. The set of all combinations of such operations forms the set of possible alignments. By defining a regular tree *grammar* the user can specify the possible combinations of operations in more detail. Finally, the cost of alignments has to be abstractly specified as an *algebra* on some given combination of operations.

Given these three ingredients, ADP automatically provides a dynamic programming scheme to calculate the respective alignment distance for any given input in $\mathcal{O}(M \cdot N)$, thereby solving the general alignment problem.

In the remainder of this chapter we introduce *operations*, combinations of operations (ADP *trees*), *algebrae* and *grammars* more formally. We show how these ingredients imply an alignment distance and we provide a general-purpose algorithm to calculate the respective alignment distance efficiently. Finally, we express four different alignment schemes using ADP: The classic edit distance (also known as global sequence alignment), affine alignment, dynamic time warping and the Sakoe-Chiba approximation of dynamic time warping. We use the classic edit distance as example to explain each ADP concept in more detail.

Definition 3. We call $\mathtt{T} := \{\mathtt{empty}, \mathtt{peek}, \mathtt{read}\}$ the set of *sequence operations*. We define an *alignment operation* as a tuple of two sequence operations and a name, where one of the operations has to be `read`. More formally:

$$(t, \mathtt{x}, \mathtt{y}) \text{ where } \mathtt{x}, \mathtt{y} \in \mathtt{T} \text{ and } \mathtt{x} = \mathtt{read} \text{ or } \mathtt{y} = \mathtt{read} \tag{13}$$

Finally, we define a *signature* as a set of alignment operations.

Note that `empty` is meant to express that the respective operation does not manipulate the respective sequence at all. `peek` means that it does refer to an element of the sequence, but does not remove it and `read` means that the element is removed from the sequence.

As an example consider the signature

$$\mathcal{T}_{\text{edit}} = \{(\text{rep}, \mathtt{read}, \mathtt{read}), (\text{del}, \mathtt{read}, \mathtt{empty}), (\text{ins}, \mathtt{empty}, \mathtt{read})\} \tag{14}$$

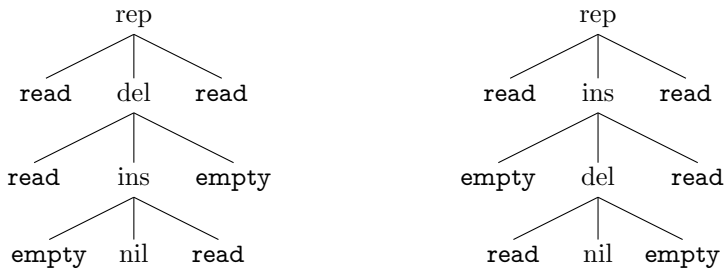


Figure 1: Two example alignment trees for the signature $\mathcal{T}_{\text{edit}}$ and the sequences $\bar{x} = \text{ab}$ and $\bar{y} = \text{ac}$ over elements from $\Sigma_\times := \Sigma_1 := \{\text{'a', 'b', 'c'}\}$. The corresponding alignments are $(\text{ab}-, \text{a}-\text{c})$ and $(\text{a}-\text{b}, \text{ac}-)$ respectively.

for the edit distance. The first operation is commonly called *replacement*, the second *deletion* and the latter *insertion*.

Based on a signature, we can rephrase the notion of an alignment in an ADP sense, as *alignment trees*:

Definition 4. Let $\mathcal{T}$ be an ADP signature and $\Sigma_\times$ be an arbitrary set. We define an ADP *tree* T recursively as either

$$T = \text{nil}() \text{ or} \tag{15}$$
$$T = t(\mathtt{x}, T', \mathtt{y}) \tag{16}$$

where T' is an ADP tree and $(t, \mathtt{x}, \mathtt{y}) \in \mathcal{T}$.

We define the *size* $|T|$ of a tree T recursively as:

$$|T| := \begin{cases} (0,0) & \text{if } T = \text{nil}() \\ \Big(m + \delta(\mathtt{x}, \mathtt{read}), n + \delta(\mathtt{y}, \mathtt{read})\Big) & \text{if } T = t(\mathtt{x}, T', \mathtt{y}) \end{cases} \tag{17}$$

where $(m, n) := |T'|$ and δ is the Kronecker-delta. Note, that the size is in effect just the number of `read`-operations on both sides of the tree.

Finally, Let $\bar{x}$, $\bar{y}$ be sequences of elements from $\Sigma_\times$. Let M and N be the respective lengths of the sequences. Then, T is called an *alignment tree* for $\bar{x}$ and $\bar{y}$, iff

$$|T| = (M, N) \tag{18}$$

Note that the notion of an ADP tree is independent of the specific input sequences. Further, the definition of an alignment tree for some two sequences $\bar{x}$ and $\bar{y}$ is only dependent on the *lengths* of $\bar{x}$ and $\bar{y}$. This is an important observation: The search space of possible alignment trees can be constructed independent of the specific input. ADP exploits this observation by separating the concerns of constructing the search space and finding the optimum in that search space. The former is done by a regular tree grammar, the latter by an algebra on alignment trees.

Two example alignment trees for $\mathcal{T}_{\text{edit}}$ and the input sequences $\bar{x} = \text{ab}$ and $\bar{y} = \text{ac}$ are shown in Figure 1. Note that alignment trees do indeed express alignments: Each `empty`-operation creates a $-$, each `read` introduces exactly one element of the input sequence and each `peek` repeats an element of the input sequence.

We generalize the notion of the *cost* of an alignment using the ADP concept of an *algebra*:

Definition 5. Let X be a set of sequences of elements from $\Sigma_\times := \bigtimes_{\kappa=1}^{K} \Sigma_\kappa$, let $\mathcal{T}$ be an ADP signature and $(t, \mathtt{x}, \mathtt{y}) \in \mathcal{T}$. We define an ADP *algebra* $\mathcal{F}(\mathcal{T}, \Sigma_\times)$ as a set of *comparator functions* $\{c_\kappa^t\}_{\kappa=1,\ldots,K}^{(t,\mathtt{x},\mathtt{y})\in\mathcal{T}}$. A comparator function, in turn, is defined as some function

$$c_\kappa^t : (\Sigma_\kappa)^{\mathcal{I}_\mathtt{x}+\mathcal{I}_\mathtt{y}} \to \mathbb{R}^+ \text{ where} \tag{19}$$

$$\mathcal{I}_\mathtt{x} := \mathbb{1}_{\{\mathtt{peek},\mathtt{read}\}}(\mathtt{x}) := \begin{cases} 1 & \text{, if } \mathtt{x} \in \{\mathtt{peek}, \mathtt{read}\} \\ 0 & \text{, if } \mathtt{x} \notin \{\mathtt{peek}, \mathtt{read}\} \end{cases} \tag{20}$$

$\mathcal{I}_\mathtt{y}$ is defined analogously to $\mathcal{I}_\mathtt{x}$.

Given an algebra, we can define the according *algebra functions*. Let $x, y \in \Sigma_\times \cup \{\epsilon\}$. We define the *algebra function* for an alignment operation $(t, \mathtt{x}, \mathtt{y}) \in \mathcal{T}$ as

$$d^t : (\Sigma_\times)^{\mathcal{I}_\mathtt{x}+\mathcal{I}_\mathtt{y}} \to \mathbb{R}^+ \tag{21}$$

$$d^t(x, y) := \sum_{\kappa=1}^{K} \lambda_\kappa c_\kappa^t(x_\kappa, y_\kappa) \tag{22}$$

where $\{\lambda_\kappa\}_{\kappa=1,\ldots,K}$ are positive, real numbers that sum up to 1. We call these numbers *relevance weights*.

Finally, let T be an alignment tree for the sequences $\bar{x}$, $\bar{y}$. we define the *application* of an algebra $\mathcal{F}$ on an T and $\bar{x}$, $\bar{y}$ as:

$$\mathcal{F}\Big(\bar{x}, \text{nil}(), \bar{y}\Big) := 0 \tag{23}$$

$$\mathcal{F}\Big(\bar{x}, t(\mathtt{x}, T', \mathtt{y}), \bar{y}\Big) := d^t(x', y') + \mathcal{F}(\bar{x}', T, \bar{y}') \text{ where} \tag{24}$$

$$x' := \begin{cases} x_1 & \text{if } \mathtt{x} \in \{\mathtt{read}, \mathtt{peek}\} \\ \epsilon & \text{if } \mathtt{x} = \mathtt{empty} \end{cases} \tag{25}$$

$$\bar{x}' := \begin{cases} (x_2, \ldots, x_M) & \text{if } \mathtt{x} = \mathtt{read} \\ \bar{x} & \text{if } \mathtt{x} \in \{\mathtt{peek}, \mathtt{empty}\} \end{cases} \tag{26}$$

y' and $\bar{y}'$ are defined analogously to x and $\bar{x}$.

As an example, consider the following algebra for the signature $\mathcal{T}_{\text{edit}}$:

$$\mathcal{F}_{\text{edit}} := \left\{ c_\kappa^{\text{rep}}(a, b) := \delta(a, b), c_\kappa^{\text{del}}(a) := 1, c_\kappa^{\text{ins}}(b) := 1 \,\middle|\, \kappa \in \{1, \ldots, K\} \right\} \tag{27}$$

where δ is the Kronecker-Delta. Consider the left example tree in Figure 1. There, we have $K = 1$ with $\lambda_1 = 1$. The corresponding application of $\mathcal{F}_{\text{edit}}$ is given as:

$$\mathcal{F}_{\text{edit}}(\bar{x}, T, \bar{y}) = \mathcal{F}_{\text{edit}}\Big(\text{ab}, \text{rep}(\texttt{read}, \text{del}(\texttt{read}, \text{ins}(\texttt{empty}, \text{nil}(), \texttt{read}), \texttt{empty}), \texttt{read}), \text{ac}\Big) \tag{28}$$

$$= d^{\text{rep}}(\text{a}, \text{a}) + \mathcal{F}_{\text{edit}}\Big(\text{b}, \text{del}(\texttt{read}, \text{ins}(\texttt{empty}, \text{nil}(), \texttt{read}), \texttt{empty}), \text{c}\Big) \tag{29}$$

$$= c_1^{\text{rep}}(\text{a}, \text{a}) + c_1^{\text{del}}(\text{b}) + \mathcal{F}_{\text{edit}}\Big((), \text{ins}(\texttt{empty}, \text{nil}(), \texttt{read}), \text{c}\Big) \tag{30}$$

$$= c_1^{\text{rep}}(\text{a}, \text{a}) + c_1^{\text{del}}(\text{b}) + c_1^{\text{ins}}(\text{c}) + \mathcal{F}_{\text{edit}}\Big((), \text{nil}(), ()\Big) \tag{31}$$

$$= 0 + 1 + 1 + 0 = 2 \tag{32}$$

Finally, we introduce the notion of a *grammar* to limit our search space of possible ADP trees.

Definition 6. Let $\mathcal{T}$ be an ADP signature. Then we define an ADP *grammar* as a tuple

$$\mathcal{G}(\mathcal{T}) = (\Phi, \Phi^*, \text{A}^*, \Delta) \tag{33}$$

where Φ is a finite set of *nonterminal symbols*, Φ^* is a subset of Φ called *accepting nonterminal symbols*, A^* is an element of Φ called the *axiom* and Δ is a set of *production rules* δ of the form:

$$\text{A} \to t(\texttt{x}, \text{B}, \texttt{y}) \tag{34}$$

where $\text{A}, \text{B} \in \Phi$ and $(t, \texttt{x}, \texttt{y}) \in \mathcal{T}$.[3]

Further, we permit the application of the production rule

$$\text{A} \to \text{nil}() \tag{35}$$

for each *accepting* nonterminal symbol. An application of this production rule removes the single existing nonterminal symbol from the expression and makes it a tree. Therefore, we can define the *language* $\mathcal{L}$ of an ADP grammar $\mathcal{G}$ as the set of all trees that can be generated by successive applications of production rules in Δ, starting with the axiom A^*.

[3]Note that we deviate from the standard formalism of regular tree grammars here: Formally, the symbols in $\texttt{T}$ are applied here as nullary elements of the alphabet, while all elements in a signature $\mathcal{T}$ are non-nullary elements. However, these definitions are only to provide a clearer separation of concepts. Each ADP grammar according to this definition is also a standard regular tree grammar if one considers the union of $\texttt{T}$, the names in $\mathcal{T}$ and $\{\text{nil}()\}$ as the alphabet and adds the rule $\text{A} \to \text{nil}()$ to the production rules instead of separately specifying a set of accepting nonterminal symbols.

As an example, consider the ADP grammar $\mathcal{G}_{\text{edit}}$ for signature $\mathcal{T}_{\text{edit}}$, which we define as:

$$\mathcal{G}_{\text{edit}} := \big(\{\text{ALI}\}, \{\text{ALI}\}, \text{ALI}, \Delta_{\text{edit}}\big) \tag{36}$$

$$\Delta_{\text{edit}} := \big\{\text{ALI} \to \text{rep}(\texttt{read}, \text{ALI}, \texttt{read}), \tag{37}$$

$$\text{ALI} \to \text{del}(\texttt{read}, \text{ALI}, \texttt{empty}), \tag{38}$$

$$\text{ALI} \to \text{ins}(\texttt{empty}, \text{ALI}, \texttt{read})\big\} \tag{39}$$

The left example tree in Figure 1 can be produced as follows:

$$\text{ALI} \to \text{rep}(\texttt{read}, \text{ALI}, \texttt{read}) \to \text{rep}(\texttt{read}, \text{del}(\texttt{read}, \text{ALI}, \texttt{empty}), \texttt{read}) \tag{40}$$

$$\to \text{rep}(\texttt{read}, \text{del}(\texttt{read}, \text{ins}(\texttt{empty}, \text{ALI}, \texttt{read}), \texttt{empty}), \texttt{read}) \tag{41}$$

$$\to \text{rep}(\texttt{read}, \text{del}(\texttt{read}, \text{ins}(\texttt{empty}, \text{nil}(), \texttt{read}), \texttt{empty}), \texttt{read}) \tag{42}$$

Using this concept of a grammar, we can find an alternative formulation for the problem of an alignment algorithm:

Definition 7. Let $\bar{x}$ and $\bar{y}$ be two sequences of elements from $\Sigma_\times$ with lengths M and N respectively. Further, let $\mathcal{T}$ be a signature, $\mathcal{F}(\mathcal{T}, \Sigma_\times)$ an algebra and $\mathcal{G}(\mathcal{T}) = (\Phi, \Phi^*, \text{A}^*, \Delta)$ be a grammar. Then the *optimal alignment tree* of $\bar{x}$ and $\bar{y}$ with respect to $\mathcal{T}$, $\mathcal{F}$ and $\mathcal{G}$ is defined as

$$T^* := \underset{T \in \mathcal{L}(\mathcal{G})}{\text{argmin}} \Big\{ \mathcal{F}(\bar{x}, T, \bar{y}) \Big| |T| = (M, N) \Big\} \tag{43}$$

Accordingly, the *alignment distance* of $\bar{x}$ and $\bar{y}$ with respect to $\mathcal{T}$, $\mathcal{F}$ and $\mathcal{G}$ is

$$D(\bar{x}, \bar{y}) = \min_{T \in \mathcal{L}(\mathcal{G})} \Big\{ \mathcal{F}(\bar{x}, T, \bar{y}) \Big| |T| = (M, N) \Big\} \tag{44}$$

As mentioned before, ADP only requires the definition of signature, algebra and grammar. Given these ingredients, ADP provides an efficient dynamic programming scheme for the calculation of the respective alignment distance automatically:

Theorem 1. *Let $\bar{x}$ and $\bar{y}$ be two sequences of elements from $\Sigma_\times$ with lengths M and N respectively. Further, let $\mathcal{T}$ be a signature, $\mathcal{F}(\mathcal{T}, \Sigma_\times)$ an algebra and $\mathcal{G}(\mathcal{T}) = (\Phi, \Phi^*, \text{A}^*, \Delta)$ be a grammar. Then, algorithm 1 calculates the alignment distance $D(\bar{x}, \bar{y})$, and it does so in $\mathcal{O}(M \cdot N)$.*

For the algorithm, we define a production rule $\delta = \text{A} \to t(\texttt{x}, \text{B}, \texttt{y})$ from Δ as applicable *iff*

$$(\texttt{x} \neq \texttt{read} \vee i < M + 1) \wedge (\texttt{y} \neq \texttt{read} \vee j < N + 1) \tag{45}$$

Further work on applicability can be found in [37].

Proof. The general proof has been provided by Giegerich and others in [10]. Here, we provide a sketch of the proof: The algorithms creates a dynamic programming table A for every nonterminal symbol A in the grammar $\mathcal{G}$. An entry

at position (i, j) stores the alignment distance for the subsequences $x_i, \ldots, x_M$ and $y_j, \ldots, y_N$. Each element $\theta_l = d^t(\ldots) + B(\ldots)$ adds the cost of an alignment operation to extend this partial alignment distance to the alignment distance for the subsequences $x_{i'}, \ldots, x_M$ and $y_{j'}, \ldots, y_N$. By applying the minimum, we use the best possible operation in each step. Therefore, $A(i, j) = \min\{\theta_1, \ldots, \theta_L\}$ is indeed a decomposition of the problem of finding an optimal alignment. This is only valid, however, if the entries in all tables A are monotonically increasing for lower i or j. This is guaranteed because $d^t(\ldots)$ is non-negative by definition (see Definition 5), which implies $\theta_l \geq B(\ldots)$ for all l.

Finally, the worst-case runtime is obvious as the number of nonterminal symbols and the number of applicable production rules are constants and the two nested for-loops lead to $\mathcal{O}(M \cdot N)$. □

Algorithm 1 An abstract dynamic programming algorithm to calculate any alignment distance corresponding to a given combination of a signature $\mathcal{T}$, algebra $\mathcal{F}$ and grammar $\mathcal{G}$.

Let $\bar{x}$ and $\bar{y}$ be two input sequences over the set $\Sigma_\times$ with lengths M and N.
Let $\mathcal{T}$ be a signature, $\mathcal{F}(\mathcal{T}, \Sigma_\times)$ an algebra and $\mathcal{G}(\mathcal{T}) = (\Phi, \Phi^*, \mathrm{A}^*, \Delta)$ be a grammar.
for $\mathrm{A} \in \Phi$ **do**
  Initialize A as a table of size $(M + 1) \times (N + 1)$.
end for
for $\mathrm{A} \in \Phi^*$ **do**
  $A(M + 1, N + 1) \leftarrow 0$.
end for
for $i \leftarrow M + 1, \ldots, 1$ **do**
  for $j \leftarrow N + 1, \ldots, 1$ **do**
    for $\mathrm{A} \in \Phi$ **do**
      Let $\{\delta_l\}_{l=1,\ldots,L}$ be the *applicable* production rules for A.
      for $l \in \{1, \ldots, L\}$ **do**
        Let $\delta_l = \mathrm{A} \rightarrow t(\mathtt{x}, \mathrm{B}, \mathtt{y})$.
        $\theta_l \leftarrow d^t(x', y') + B(i', j')$, where
        $x' = \epsilon$ if $\mathtt{x} = \mathtt{empty}$ and $x' = x_i$ otherwise and
        $i' = i + 1$ if $\mathtt{x} = \mathtt{read}$ and $i' = i$ otherwise.
      end for
      $A(i, j) \leftarrow \min\{\theta_1, \ldots, \theta_L\}$.
    end for
  end for
end for
return $D_\lambda(\bar{x}, \bar{y}) = A^*(1, 1)$.

This general scheme allows for easy construction of alignment algorithms for different situations. The simple edit distance mentioned above can be expressed as the combination of $\mathcal{T}_{\text{edit}}$, $\mathcal{F}_{\text{edit}}$ and $\mathcal{G}_{\text{edit}}$ as mentioned before. *Dynamic time*

warping (DTW) can be expressed like this:

$$\mathcal{T}_{\mathrm{DTW}} := \Big\{ \mathrm{rep}(\texttt{read}, \texttt{read}), \mathrm{rep\_del}(\texttt{read}, \texttt{peek}), \mathrm{rep\_ins}(\texttt{peek}, \texttt{read}) \Big\} \tag{46}$$

$$\mathcal{F}_{\mathrm{DTW}} := \Big\{ c^t_\kappa(a,b) := \|a-b\|_2 \Big| \kappa \in \{1,\ldots,K\}, (t, \mathtt{x}, \mathtt{y}) \in \mathcal{T}_{\mathrm{DTW}} \Big\} \tag{47}$$

$$\mathcal{G}_{\mathrm{DTW}} := \Big(\{\mathrm{ALI}\}, \{\mathrm{ALI}\}, \mathrm{ALI}, \Delta_{\mathrm{DTW}} \Big) \tag{48}$$

$$\begin{aligned} \Delta_{\mathrm{DTW}} := \Big\{ & \mathrm{ALI} \to \mathrm{rep}(\texttt{read}, \mathrm{ALI}, \texttt{read}), \\ & \mathrm{ALI} \to \mathrm{rep\_del}(\texttt{read}, \mathrm{ALI}, \texttt{peek}), \\ & \mathrm{ALI} \to \mathrm{rep\_ins}(\texttt{peek}, \mathrm{ALI}, \texttt{read}) \Big\} \end{aligned} \tag{49}$$

DTW can be accelerated by slightly changing Algorithm 1 and permitting the second and third rule only if $|i-j| < W$, where $W \in \mathbb{N}$ is the so-called Sakoe Chiba-bandwidth [28]. This leads to a worst case runtime of $\mathcal{O}(\max\{M,N\} \cdot W) = \mathcal{O}(\max\{M,N\})$. We use these two algorithms in our first experiment on motion data (see Section 5.1).

To compare Java programs, we apply a variant of the edit distance mentioned above: We do not only permit the algorithm to delete or insert, but also to skip consecutive subsequences of both inputs. For this, we define a relatively high initial opening cost s for such a region, while each following skip is relatively cheap with a cost of $\hat{s} < s$. Thereby, the algorithm is enabled to skip subsequences that could only be aligned at high cost, such that those long, consecutive subsequences do not drive the alignment distance as much. Conversely, one can also interpret this form of alignment as identifying those subsequences of both inputs that match best and ignore the rest. This alignment approach is called *affine* alignment in the bioinformatics community and was first introduced by [26]. The according ADP signature, algebra and grammar are given as $\mathcal{T}_{\mathrm{affine}}$, $\mathcal{F}_{\mathrm{affine}}$ and $\mathcal{G}_{\mathrm{affine}}$ below.

$$\mathcal{T}_{\text{affine}} := \{\ \text{rep} := (\texttt{read}, \texttt{read}), \tag{50}$$
$$\text{del} := (\texttt{read}, \texttt{empty}), \text{ins} := (\texttt{empty}, \texttt{read}),$$
$$\text{skip\_del} := (\texttt{read}, \texttt{empty}), \text{skip\_ins} := (\texttt{empty}, \texttt{read}),$$
$$\text{skip\_del\_open} := (\texttt{read}, \texttt{empty}), \text{skip\_ins\_open} := (\texttt{empty}, \texttt{read})\}$$

$$\mathcal{F}_{\text{affine}} := \Bigg\{ c_\kappa^{\text{rep}}(a,b) := \begin{cases} 0 & \text{, if } a = b \\ 1 & \text{, if } a \neq b \end{cases}, \tag{51}$$
$$c_\kappa^{\text{del}}(a) := c_\kappa^{\text{ins}}(b) := 1,$$
$$c_\kappa^{\text{skip\_del\_open}}(a) := c_\kappa^{\text{skip\_ins\_open}}(b) := s,$$
$$c_\kappa^{\text{skip\_del}}(a) := c_\kappa^{\text{skip\_ins}}(b) := \hat{s}, \Big| \kappa \in \{1, \ldots, K\} \Bigg\}$$

$$\mathcal{G}_{\text{affine}} := \big(\ \Phi_{\text{affine}}, \Phi_{\text{affine}}, \text{ALI}, \Delta_{\text{affine}}\big) \tag{52}$$

$$\Phi_{\text{affine}} := \{\ \text{ALI}, \text{SKIPDEL}, \text{SKIPINS}\} \tag{53}$$

$$\Delta_{\text{affine}} := \big\{\ \text{ALI} \to \text{skip\_del\_open}(\texttt{read}, \text{SKIPDEL}, \texttt{empty}), \tag{54}$$
$$\text{ALI} \to \text{skip\_ins\_open}(\texttt{empty}, \text{SKIPINS}, \texttt{read}),$$
$$\text{ALI} \to \text{rep}(\texttt{read}, \text{ALI}, \texttt{read}),$$
$$\text{ALI} \to \text{del}(\texttt{read}, \text{ALI}, \texttt{empty}),$$
$$\text{ALI} \to \text{ins}(\texttt{empty}, \text{ALI}, \texttt{read}),$$
$$\text{SKIPDEL} \to \text{skip\_del}(\texttt{read}, \text{SKIPDEL}, \texttt{empty}),$$
$$\text{SKIPDEL} \to \text{rep}(\texttt{read}, \text{ALI}, \texttt{read}),$$
$$\text{SKIPINS} \to \text{skip\_ins}(\texttt{empty}, \text{SKIPINS}, \texttt{read}),$$
$$\text{SKIPINS} \to \text{rep}(\texttt{read}, \text{ALI}, \texttt{read})\big\}$$

4 Metric Learning using RGLVQ

The definition of algebrae in Section 3 provides us with the parameters λ_κ, which capture the *relevance* (and scaling) of the different dimensions κ in each sequence element. This formulation is inspired by the generalized quadratic form metric in vectorial settings [31, 30, 32]:

$$D_\Lambda(\vec{x}, \vec{y}) := (\vec{x} - \vec{y})^T \Lambda^T \Lambda (\vec{x} - \vec{y}) = (\Lambda\vec{x} - \Lambda\vec{y})^T (\Lambda\vec{x} - \Lambda\vec{y}) \tag{55}$$

If one restricts Λ to diagonal matrices, this is equivalent to a weighting of the dimensions:

$$D_\Lambda(\vec{x}, \vec{y}) = (\vec{x} - \vec{y})^T \text{diag}(\Lambda)^T \text{diag}(\Lambda)(\vec{x} - \vec{y}) = \sum_{\kappa=1}^{K} \lambda_\kappa^2 (x_\kappa - y_\kappa)^2 \tag{56}$$

Metric learning is the challenge of finding optimal parameters Λ for the task at hand. In particular, metric learning has been applied to classification tasks:

If a classification algorithm provides a model based on Λ and/or the distances created with it, one can adjust Λ to optimize classification accuracy. Numerous such metric learning approaches exist for the vectorial setting, as can be seen in the summaries [31, 30].

In this paper, we build in particular on recent work on *learning vector quantization* (LVQ) schemes: Such a scheme classifies a data point x by representing the classes by *prototypes* $\{\vec{w}_s\}_{s=1,\ldots,S}$ and assigning the data point x to its closest prototype in a winner-takes-all fashion. *Learning* an LVQ model means moving the prototypes in the data space, such that the classification accuracy is optimal. Apparently, the classification accuracy of LVQ critically relies on distances between data points and prototypes can profit from metric learning approaches: *Generalized relevance learning vector quantization* (GRLVQ) [32], optimizes a diagonal Λ matrix in the vectorial case. *Relational generalized learning vector quantization* (RGLVQ) extends LVQ to relational data, based solely on the dissimilarities between data points [18]. Thereby, RGLVQ is able to classify structure data as well, if a suitable structure metric is given. Both approaches have been merged in the work of [11], providing a metric learning scheme on edit distances. We extend this approach using algebraic dynamic programming towards more sophisticated alignment schemes.

RGLVQ is based on the insight that data, given only in terms of pairwise, symmetric dissimilarities with vanishing self-dissimilarities, can be embedded as vectors in a pseudo-Euclidean space, in which regular (G)LVQ could be applied [38]. Interestingly though, one does not have to compute this vectorial representation explicitly, but can compute an RGLVQ model based on the pairwise dissimilarities alone. The LVQ prototypes $\{\vec{w}_s\}_{s=1,\ldots,S}$ are assumed to be given as convex combinations of the embedded data points $\{\vec{x}_r\}_{r=1,\ldots,R}$:

$$\vec{w}_s := \sum_{r=1}^{R} \alpha_{sr} \vec{x}_r \text{ with } \sum_{r=1}^{R} \alpha_{sr} = 1 \text{ and } \forall s, r : \alpha_{sr} \geq 0 \tag{57}$$

Therefore, the dissimilarities between prototypes and data points are given in terms of pairwise dissimilarities only:

$$D_\lambda(\bar{x}_r, w_s) := \sum_{r'=1}^{R} \alpha_{sr'} D_\lambda(\bar{x}_r, \bar{x}_{r'}) - \frac{1}{2} \sum_{r'=1}^{R} \sum_{r''=1}^{R} \alpha_{sr'} \alpha_{sr''} D_\lambda(\bar{x}_{r'}, \bar{x}_{r''}) \tag{58}$$

These distances can be plugged into the usual GLVQ cost function:

$$E_{\text{GLVQ}} := \sum_{r=1}^{R} \Phi\left(\frac{D_\lambda(\bar{x}_r, w^+) - D_\lambda(\bar{x}_r, w^-)}{D_\lambda(\bar{x}_r, w^+) + D_\lambda(\bar{x}_r, w^-)} \right) \tag{59}$$

where w^+ is the closest prototype with the same label as $\bar{x}_r$ and w^- is the closest prototype with a different label than $\bar{x}_r$. In usual RGLVQ, the position of the prototypes is adapted to minimize wrong labeling of data points [18]. In [11] the authors suggest to also optimize the metric parameters λ_κ by gradient descent. The detailed formulas can be found there. We want to highlight, that

a gradient descent on the GLVQ cost function with respect to λ_κ depends on a gradient of the pairwise dissimilarities $D_\lambda(\bar{x}, \bar{y})$ with respect to λ_κ. In [11], this gradient has been derived for the simple edit distance case. For the formulas given in Algorithm 1, we arrive at:

$$\frac{\partial}{\partial \lambda_\kappa} A(i,j) = \frac{\partial}{\partial \lambda_\kappa} \min\{\theta_1, \ldots, \theta_L\} \tag{60}$$

As the minimum function is non-differentiable, we replace it by the differentiable approximation

$$\text{softmin}\{\theta_1, \ldots, \theta_L\} := \frac{\sum_{l=1}^{L} \exp(-\beta \cdot \theta_l) \cdot \theta_l}{\sum_{l=1}^{L} \exp(-\beta \cdot \theta_l)} \tag{61}$$

This approximation converges exponentially towards the strict minimum with higher β [11, 37]. This leads us to:

$$\frac{\partial}{\partial \lambda_\kappa} A(i,j) = \sum_{l=1}^{L} \text{softmin}'(\theta_l) \cdot \left(\frac{\partial}{\partial \lambda_\kappa} \theta_l \right) \text{ where} \tag{62}$$

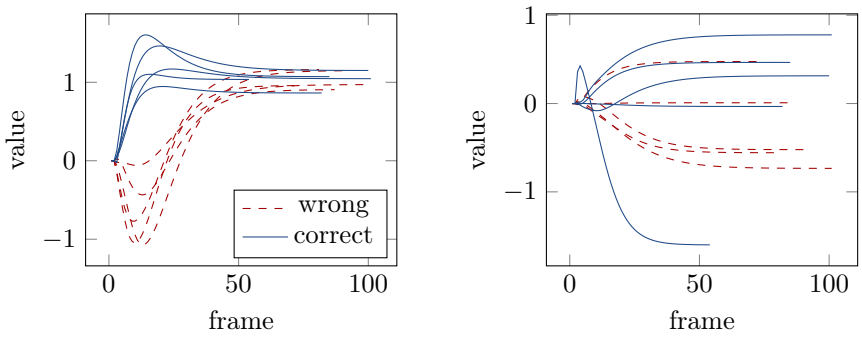
$$\text{softmin}'(\theta_l) := \frac{\exp(-\beta \cdot \theta_l) \cdot \theta_l}{\sum_{l'=1}^{L} \exp(-\beta \cdot \theta_{l'})} \cdot \Big(1 - \beta \cdot (\theta_l - \text{softmin}\{\theta_1, \ldots, \theta_L\}) \Big) \tag{63}$$

$$\frac{\partial}{\partial \lambda_\kappa} \theta_l = \frac{\partial}{\partial \lambda_\kappa} d^t(x'_i, y'_j) + \frac{\partial}{\partial \lambda_\kappa} B(i', j') = c^t_\kappa(x'_i, y'_j) + \frac{\partial}{\partial \lambda_\kappa} B(i', j') \tag{64}$$

This does not only provide a generalized formula for the gradient on alignment distances. We also observe, that this formula follows the same recursive scheme as the original distance calculation. Therefore, we can calculate it using Algorithm 1, using the same ADP signature and grammar, but softmin instead of a strict minimum as well as a different algebra. This also implies that the gradient calculation can be done efficiently in $\mathcal{O}(M \cdot N)$. Gradient calculation with respect to parameters of the comparator functions c^t_κ are possible as well and result in a Hebbian learning scheme for the case of an explicit parameter matrix [11, 37]. Further, one can apply approximations to limit the number of calculated cells in the dynamic programming tables, such that further runtime improvements are possible [11].

5 Experiments

We present experimental results on two example datasets, an artificial one on sports coaching and a real one on Java programming. In both experiments, we adapt the underlying alignment metric parameters λ_κ in order to increase the classification accuracy of an RGLVQ classifier with one prototype per class. The classes are defined as correct and wrong executions of a sports exercise in the first dataset, and as two different programming strategies in the second dataset. Our experimental hypotheses are: (A) metric learning does improve



(a) The movement in the first (and relevant) dimension over time. All executions start at position 0 and end (roughly) at position one. Intermediately, however, the wrong executions go into the wrong direction.

(b) The movement in the second (irrelevant) dimension. All executions start at position 0 and end at a random position around 0. The nonlinear motion in between is also determined by Gaussian random noise.

Figure 2: Some example executions for our fictional sports exercise. The position of a body part is shown over time (in frames). Wrong executions are shown in red (and dashed), correct executions in blue.

the classification accuracy of an RGLVQ classifier, (B) the learned metric leads to better classification accuracy for k-nearest neighbor (KNN) classifiers and support vector machine (SVM) as well, and (C) the resulting relevance profile makes sense with respect to the task.

We initialized the weights as $\lambda_\kappa := \frac{1}{K}$ for all $\kappa \in \{1, \ldots, K\}$. We applied 10 gradient steps with a learning rate of $\eta := \frac{2}{|X| \cdot \langle M \rangle}$ in the first and $\eta := \frac{0.1}{|X| \cdot \langle M \rangle}$ in the second experiment, where $|X|$ is the number of sequences and $\langle M \rangle$ is the mean sequence length. For comparison with SVM, we transformed the dissimilarity matrix into a kernel matrix by double centering and flip-correction of the negative eigenvalues.

5.1 Motion Data

We simulated executions of a sports exercise using *dynamical movement primitives* (DMPs). DMPs are a well-established mean to describe movements by dynamical systems [35]. We use the fixed point attractor version of the DMP framework to generate 60 sample trajectories (30 per class) from the origin of the coordinate system to a goal point $\vec{g}$. Each element in our trajectories simulates a frame in motion tracking, and each dimension simulates the position of a body part or the angle of a joint in the human body. In our fictional exercise, we assume that the movement of only one body part is relevant, while the remaining parts of the body may move freely.

We simulated the movement over 20 seconds using step-wise approximations

Method	Train (Std.)	Test (Std.)	SVM (Std.)	5-NN	Sep.
DTW	0.70 (0.06)	0.55 (0.11)	0.58 (0.16)	0.42	0.94
DTW adapted	0.94 (0.02)	0.94 (0.06)	0.88 (0.11)	0.92	0.57
SC-DTW	0.81 (0.04)	0.65 (0.14)	0.71 (0.16)	0.58	0.93
SC-DTW adapted	0.98 (0.01)	0.98 (0.03)	0.95 (0.06)	1.00	0.35

Table 1: The results for the motion dataset. The rows represent different configurations of the metric (unrestricted dynamic time warping (DTW) without and with metric learning and Sakoe-Chiba approximation (SC-DTW) without and with metric learning). In the first two columns, we show average training and test accuracy for RGLVQ over a 5-fold crossvalidation with 5 repeats (the standard deviation is shown in brackets). The third column shows mean test accuracy across crossvalidation trials for a support vector machine (SVM) as well as the standard deviation in brackets. Column four contains the test accuracy of a 5-nearest neighbor (5-NN) classifier and column 5 the class separation ratio (Sep.; summed intra-class distances divided by summed inter-class distances; a low value is better) respectively.

of the differential equations with a standard Euler approach. To simulate different movement speeds of the learners, the step size was randomly chosen as $\Delta t = \mu + \rho^2$ with $\mu = 0.2s$, where ρ in turn was a random variable drawn from the normal distribution $\mathcal{N}(\mu = 0s, \sigma = 0.2s)$. This leads to an average sequence length of 89 frames (standard deviation: ≈ 15). In our fictional sports exercise, students should move the body part corresponding to the first dimension from the origin to position 1. In our simulation, we assumed that students where generally able to do that (the actual goal of the movement was selected from $\mathcal{N}(\mu = 1, \sigma = 0.1)$), but erroneous executions first moved the arm a little in the wrong direction, before proceeding towards the goal. Thereby, we simulated execution errors that might be harmful to students, e.g. due to hyper-extension. We simulated these errors by utilizing the nonlinear force term of the DMP framework with weights $\vec{w}_r$ chosen from $\mathcal{N}(-1, 1)$ for the erroneous executions and $\mathcal{N}(1, 1)$ for the correct executions. The random noise simulates differences in style. We simulated the movement of other body parts by 9 additional dimensions, which are not related to the exercise and thus do not help for the classification task. The goal position in these dimensions as well as the nonlinear force weights were selected from $\mathcal{N}(0, 1)$. The first two dimensions for some sample trajectories are visualized in Figure 2.

To compare such time series of different lengths, warping techniques are required and dynamic time warping (DTW) is probably the best-known warping technique available [27]. We applied DTW as introduced in Section 3 as well as the Sakoe-Chiba approximation [28].

The experimental results are shown in Table 1. Due to the high noise level, the test classification accuracy without metric learning is close to random guessing (around 55% on average in a 5-fold crossvalidation with 5 repeats), while the accuracy goes to 94% on average with metric learning, which confirms hy-

pothesis (A). Interestingly, the results for the Sakoe-Chiba approximation are even better than unrestricted dynamic time warping (10% better without metric learning and 4% better with metric learning). This might be due to the fact, that the additional degrees of freedom of the unrestricted DTW might underestimate the differences between classes. Our second research hypothesis (B) is supported as well: The adapted metric leads to improvements of 24% or more for SVM- and 5-NN classifiers. Also, the separation ratio between the classes is notably improved by metric learning. Finally, we note that metric learning does indeed identify the first dimension as the only relevant one, with λ_1 going to 1 during learning, while all other weights are reduced to 0, thereby confirming our last hypothesis (C).

5.2 Java Programs

Our Java dataset consists of 64 programs collected from 37 different web sites, which all solve the same underlying task, namely sorting an input array of integers in ascending order. However, the programs differed in their underlying strategy, which was either the *BubbleSort* (35 programs) or the *InsertionSort* (29 programs) algorithm.

In order to apply alignment algorithms on Java programs, we need a sequential representation of said programs as well as metrics on the element-wise features. As proposed in [39], we transfer program code to an abstract syntax tree using the Oracle Java Compiler API. Every vertex of this parse tree is characterized by a feature vector encoding characteristic properties: the vertex type, the scope, the parent vertex, code position, name, class name, return type, external references, and internal references (number of edges). The prefix order of the tree vertices gives us a sequential representation of the program, which corresponds to their natural sequential order when executing the program. We defined straightforward metrics for each feature: For the scope we returned one minus the normalized depth of the common parent scope, for vectorial features (parent vertex, code position and internal references) we applied the Manhattan-distance, for string features (name, class name, return type, external references) we applied a Manhattan-distance on the character frequency vectors and we compared vertex types with a Kronecker-Delta (1 for unequal types and 0 for equal types).

We compared two different alignment algorithms: First, the classic edit distance/global sequence alignment as baseline [22, 36, 24] and second, inspired by bioinformatics, the affine sequence alignment proposed by Gotoh [26]. The latter approach is motivated by the fact, that differences in programming style might lead to long subsequences of irrelevant code, e.g. intermediate results or diagnostic code. Affine sequence alignment permits to skip these parts of the program at low cost and concentrating on more important differences between the programs.

The results are shown in Table 2: Metric learning increases the classification accuracy of RGLVQ notably (6 percent points for global alignment and 11 percent for affine alignment), confirming hypothesis (A). While the initial ac-

Method	Train (Std.)	Test (Std.)	SVM (Std.)	5-NN	Sep.
global	0.75 (0.03)	0.74 (0.10)	0.65 (0.16)	0.77	0.88
global adapted	0.80 (0.02)	0.80 (0.09)	0.63 (0.19)	0.92	0.75
affine	0.77 (0.03)	0.74 (0.12)	0.74 (0.12)	0.62	0.9
affine adapted	0.85 (0.03)	0.85 (0.10)	0.78 (0.17)	1.00	0.74

Table 2: The results for the Java dataset. The rows represent different configurations of the metric (edit distance (global) without and with metric learning and affine alignment (affine) without and with metric learning). In the first two columns, we show average training and test accuracy for RGLVQ over a 5-fold crossvalidation with 5 repeats (the standard deviation is shown in brackets). The third column shows mean test accuracy across crossvalidation trials for a support vector machine (SVM) as well as the standard deviation in brackets. Column four contains the test accuracy of a 5-nearest neighbor (5-NN) classifier and column 5 the class separation ratio (Sep.; intra-class distances divided by inter-class distances; a low value is better) respectively.

curacies for global and affine sequence alignment are the same, metric learning showed a stronger effect for affine sequence alignment. This is reflected as well in the accuracy of a support vector machine and a 5-nearest neighbor classifier on the resulting dissimilarity matrix: In line with hypothesis (B), metric learning improves the result and affine alignment leads to better results (after metric learning) than global alignment. The adapted dissimilarities also notably reduce the class separation ratio (where low values are good).

The resulting relevances λ_κ (normalized by their frequency in the data) are shown in Fig. 3. As predicted in hypothesis (C), a semantically meaningful profile results, which marks the type and scope as most prominent structuring elements to distinguish programming styles. In contrast, exact code position, as well as several other features play only a very minor role, as they tend to be sensitive to mere stylistic differences.

The improvements of the metric for the classification task can also be seen in a t-stochastic neighborhood embedding (t-SNE [40]) of the resulting dissimilarity matrix. Figure 4 shows embeddings for the unadapted and the adapted metric. Apparently, in the embedding for the adapted metric, programs with the same underlying strategy become visible as clusters.

6 Conclusion

We have introduced a simplified version of algebraic dynamic programming (ADP) in order to express four different alignment algorithms (DTW, DTW with Sakoe-Chiba approximation, global alignment and affine alignment) in a common framework. We showed that the ADP formulation is as efficient as the original algorithms and that one can generally compute gradients on these alignment distances with respect to metric parameters.

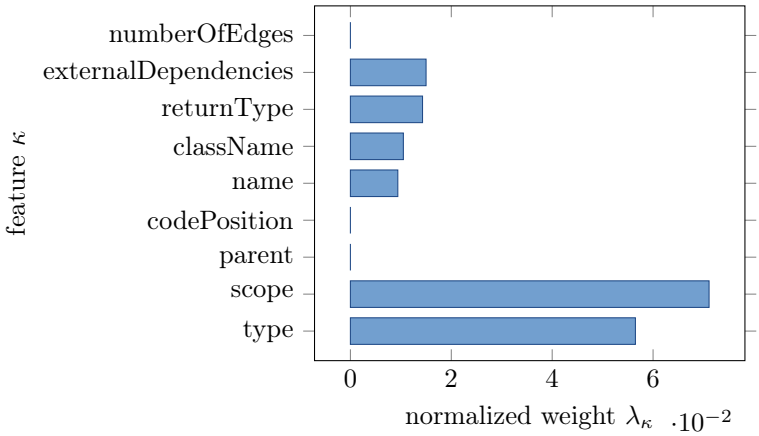


Figure 3: The relevance weights λ_κ after metric learning has been applied. The weights were normalized by their frequency in the dataset.

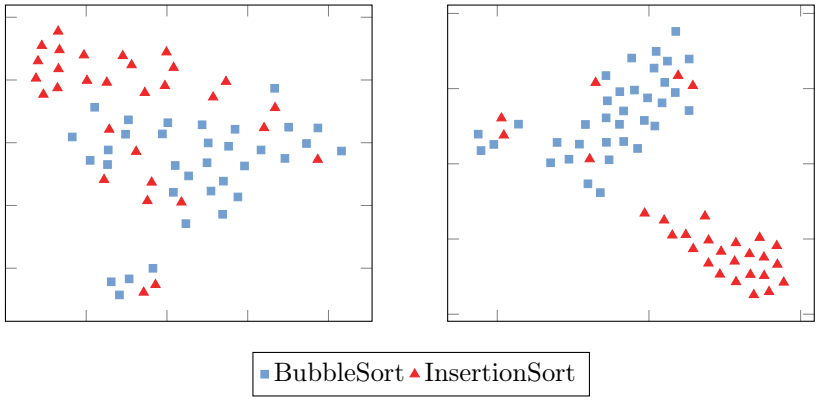


Figure 4: Two-dimensional t-SNE embeddings of the Java dataset without metric learning (left) and with metric learning (right). *BubbleSort* programs are visualized as blue squares, *InsertionSort* programs as red triangles. These visualizations are based on the affine sequence alignment metric.

Based on those gradients, we introduced a learning scheme for alignment metric parameters based on the cost function of the popular learning vector quantization (LVQ) classifier. We demonstrated the effectiveness of this learning scheme in two datasets: classifying right versus wrong executions of a simulated sports exercise, and classifying the underlying algorithm in Java sorting programs. In both cases, metric learning notably increased classification accuracy; whereby this result is independent from the subsequent classifier used (LVQ, kNN, and SVM). In addition, metric parameters allow for semantic insight into the domain (the relevant body parts for sport executions, and the relevant features in Java program strategy distinction).

These results lay the foundation for semantically meaningful distance measures, based on which an intelligent tutoring system can autonomously select examples for feedback. Thereby, we provide further support for data-driven intelligent tutoring approaches, that do not require formalized domain knowledge or a clear definition of correct solutions.

Furthermore, we have extended metric learning schemes for autonomous selection of metric parameters towards sophisticated structure metric schemes like affine sequence alignment, enabling the application of structure metric learning in new situations.

In future work, it would be interesting to transfer these promising results from GLVQ to other cost functions, such as the popular Large Margin Nearest Neighbor approach, and to apply other nonlinear optimization methods than gradient descent.

Acknowledgement

Funding by the DFG under grant numbers HA 2719/6-1 and HA 2719/6-2 is gratefully acknowledged. Additionally, this research/work was supported by the Cluster of Excellence Cognitive Interaction Technology 'CITEC' (EXC 277) at Bielefeld University, which is funded by the German Research Foundation (DFG).

References

References

[1] B. Paassen, B. Mokbel, B. Hammer, Adaptive structure metrics for automated feedback provision in intelligent tutoring systems, Neurocomputing 192 (2016) 3–13. doi:10.1016/j.neucom.2015.12.108.

[2] R. Nkambou, R. Mizoguchi, J. Bourdeau, Advances in Intelligent Tutoring Systems, Springer, 2010.

[3] T. Murray, S. Blessing, S. Ainsworth, Authoring tools for advanced technology learning environments: Toward cost-effective adaptive, interactive and intelligent educational software, Springer, 2003.

[4] C. Lynch, K. D. Ashley, N. Pinkwart, V. Aleven, Concepts, structures, and goals: Redefining ill-definedness, International Journal of Artificial Intelligence in Education 19 (3) (2009) 253–266.

[5] K. R. Koedinger, E. Brunskill, R. S. Baker, E. A. McLaughlin, J. Stamper, New potentials for data-driven intelligent tutoring system development and optimization, AI Magazine 34 (3) (2013) 27–41.

[6] J. C. Stamper, M. Eagle, T. Barnes, M. Croy, Experimental evaluation of automatic hint generation for a logic tutor, in: G. Biswas, S. Bull, J. Kay, A. Mitrovic (Eds.), Artificial Intelligence in Education, Vol. 6738 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2011, pp. 345–352.

[7] S. Gross, B. Mokbel, B. Hammer, N. Pinkwart, How to select an example? A comparison of selection strategies in example-based learning, in: ITS, 2014, pp. 340–347.

[8] S. Gross, B. Mokbel, B. Hammer, N. Pinkwart, Example-bases feedback provision using structured solution spaces, International Journal on Learning Technologies 9 (3) (2014) 248–280.

[9] K. Rivers, K. R. Koedinger, Automatic generation of programming feedback: A data-driven approach, The First Workshop on AI-supported Education for Computer Science (AIEDCS 2013) (2013) 50.

[10] R. Giegerich, C. Meyer, P. Steffen, A discipline of dynamic programming over sequence data, Science of Computer Programming 51 (3) (2004) 215 – 263.

[11] B. Mokbel, B. Paaßen, F.-M. Schleif, B. Hammer, Metric learning for sequences in relational LVQ, Neurocomputing 169 (2015) 306–322.

[12] S. Ritter, J. R. Anderson, K. R. Koedinger, A. Corbett, Cognitive tutor: Applied research in mathematics education, Psychonomic bulletin & review 14 (2) (2007) 249–255.

[13] C. Piech, J. Huang, A. Nguyen, M. Phulsuksombati, M. Sahami, L. J. Guibas, Learning program embeddings to propagate feedback on student code, in: Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015, 2015, pp. 1093–1102.

[14] T. M. Cover, P. E. Hart, Nearest neighbor pattern classification, IEEE Transactions on Information Theory 13 (1) (1967) 21–27.

[15] S. P. Lloyd, Least squares quantization in pcm, IEEE Transactions on Information Theory 28 (2) (1982) 129–136.

[16] Y. Chen, E. K. Garcia, M. R. Gupta, A. Rahimi, L. Cazzanti, Similarity-based classification: Concepts and algorithms, The Journal of Machine Learning Research 10 (2009) 747–776.

[17] C. E. Rasmussen, C. K. I. Williams, Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning), The MIT Press, 2005.

[18] B. Hammer, D. Hofmann, F. Schleif, X. Zhu, Learning vector quantization for (dis-)similarities, Neurocomputing 131 (2014) 43–51.

[19] G. Da San Martino, A. Sperduti, Mining structured data, Computational Intelligence Magazine, IEEE 5 (1) (2010) 42–49.

[20] T. Gärtner, A survey of kernels for structured data, SIGKDD Explor. Newsl. 5 (1) (2003) 49–58.

[21] T. Gärtner, T. Horváth, S. Wrobel, Graph kernels, in: Encyclopedia of Machine Learning, 2010, pp. 467–469.

[22] V. I. Levenshtein, Binary codes capable of correcting deletions, insertions, and reversals, Soviet Physics Doklady 10 (8) (1965) 707–710.

[23] F. J. Damerau, A technique for computer detection and correction of spelling errors, Commun. ACM 7 (3) (1964) 171–176.

[24] S. B. Needleman, C. D. Wunsch, A general method applicable to the search for similarities in the amino acid sequence of two proteins, Journal of Molecular Biology 48 (3) (1970) 443 – 453.

[25] T. Smith, M. Waterman, Identification of common molecular subsequences, Journal of Molecular Biology 147 (1) (1981) 195 – 197.

[26] O. Gotoh, An improved algorithm for matching biological sequences, Journal of molecular biology 162 (3) (1982) 705–708.

[27] T. Vintsyuk, Speech discrimination by dynamic programming, Cybernetics 4 (1) (1968) 52–57.

[28] H. Sakoe, S. Chiba, Readings in speech recognition, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990, Ch. Dynamic Programming Algorithm Optimization for Spoken Word Recognition, pp. 159–165.

[29] J. D. Kececioglu, E. Kim, Simple and fast inverse alignment, in: Research in Computational Molecular Biology, 10th Annual International Conference, RECOMB 2006, Venice, Italy, April 2-5, 2006, Proceedings, 2006, pp. 441–455.

[30] A. Bellet, A. Habrard, M. Sebban, A Survey on Metric Learning for Feature Vectors and Structured Data, ArXiv e-prints (1306.6709).

[31] B. Kulis, Metric learning: A survey, Foundations and Trends in Machine Learning 5 (4) (2013) 287–364.

[32] P. Schneider, M. Biehl, B. Hammer, Adaptive relevance matrices in learning vector quantization, Neural Computation 21 (12) (2009) 3532–3561.

[33] A. Bellet, A. Habrard, M. Sebban, Good edit similarity learning by loss minimization, Machine Learning 89 (1-2) (2012) 5–35.

[34] U. Yang, G. J. Kim, Implementation and evaluation of "just follow me": An immersive, VR-based, motion-training system, Presence: Teleoperators and Virtual Environments 11 (3) (2002) 304–323.

[35] A. J. Ijspeert, J. Nakanishi, H. Hoffmann, P. Pastor, S. Schaal, Dynamical movement primitives: Learning attractor models for motor behaviors, Neural Comput. 25 (2) (2013) 328–373.

[36] R. A. Wagner, M. J. Fischer, The string-to-string correction problem, J. ACM 21 (1) (1974) 168–173.

[37] B. Paaßen, Adaptive affine sequence alignment using algebraic dynamic programming, Master's thesis (2015).

[38] E. Pękalska, The dissimilarity representation for pattern recognition: foundations and applications, Ph.D. thesis (2005).

[39] B. Mokbel, S. Gross, B. Paaßen, N. Pinkwart, B. Hammer, Domain-independent proximity measures in intelligent tutoring systems, in: EDM, 2013, pp. 334–335.

[40] L. van der Maaten, G. E. Hinton, Visualizing high-dimensional data using t-SNE, Journal of Machine Learning Research 9 (2008) 2579–2605.