Adaptive Affine Sequence Alignment Using Algebraic Dynamic Programming

Master Thesis (Master of Science)

Benjamin Paaßen

Faculty of Technology
Bielefeld University

Supervisors:
Prof. Dr. Barbara Hammer
Dipl. Inf. Bassam Mokbel

April 2015

Abstract

A core issue in machine learning is the classification of data. However, for data structures that can not easily be summarized in a feature representation, standard vectorial approaches are not suitable. An alternative approach is to represent the data not by features, but by their similarities or disimilarities to each other. In the case of sequential data, dissimilarities can be efficiently calculated by well-established alignment distances. Recently, techniques have been put forward to adapt the parameters of such alignment distances to the specific data set at hand, e.g. using gradient descent on a cost function.

In this thesis we provide a comprehensive theory for gradient descent on alignment distance based on Algebraic Dynamic Programming, enabling us to adapt even sophisticated alignment distances. We focus on Affine Sequence Alignment, which we optimize by gradient descent on the Large Margin Nearest Neighbor cost function. Thereby we directly optimize the classification accuracy of the popular k-Nearest Neighbor classifier.

We present a free software implementation of this theory, the *TCS Alignment Toolbox*, which we use for the subsequent experiments. Our experiments entail alignment distance learning on three diverse data sets (two artificial ones and one real-world example), yielding not only an increase in classification accuracy but also interpretable resulting parameter settings.

Contents

Chapter 1

Introduction

A core issue in machine learning is the classification of data: Given some examples for which the correct class is known, a machine learning algorithm should provide the correct class for every new data point.

Classically in machine learning, one solves this problem by describing the data points with a vector of features and letting a machine learning algorithm detect correlations between features, such as: If feature x_i has a high value this suggests that the data point belongs to class y. An abundance of such methods can be found in the book by Bishop [6], for example. A vectorial description becomes difficult, however, if one deals with rich data structures, like long texts, audio streams or gene sequences. In fact, important information might be lost in the process of describing rich structures in terms of a few features [1, 32].

In this thesis we consider sequential data, where feature-like representations include term frequencies [35, 36] and string kernels [25], which both abstract from the order of elements in the sequence.

An alternative perspective on sequences is offered by sequence alignment algorithms like the classic String Edit Distance [11, 22]. These algorithms do not describe the input sequences in terms of features, but rather by their similarity or dissimilarity to each other and fully account for the order of the sequence elements.

Fortunately, this similarity representation can still be used to solve a classification task, as similarity-based classification methods are well-investigated and one can rely on established and popular methods such as k-Nearest Neighbor (k-NN) and k-means classification [9, 24].

However, classification schemes based on rich structure similarity measures, are bound to fail if the parameters of said measures are not suited for the given task. But what are the right parameters? Ideally, a machine learning algorithm would answer this question autonomously. More formally: One would like to construct a machine learning algorithm that does not only optimize the parameters of a classifier with respect to classification accuracy, but also the parameters of the underlying distance function.

This is the topic of *structure metric learning*[1], which has been identified

[1]It should be noted that the literature does not consistently use the terms "metric" or

as a novel, challenging area of research with high relevance, preceeded by the well-invastigated and very successful field of vectorial metric learning in pattern recognition [1]. Bellet, Habrard, and Sebban [1] report that only a few approaches exist particularly in the context of alignment distances. Consequently, this is the focus of our work: The adaptation of alignment distances in order to optimize classification performance, which we call *alignment distance learning* in the following.

In previous work we have investigated alignment distance learning within the framework of Relational Generalized Learning Vector Quantization (RGLVQ) [17]. We have demonstrated that alignment distance learning is a powerful method to discriminate gene sequences in the biological domain and programming strategies in the educational domain [27, 28, 29, 31]. In particular we have investigated possible approximations [29] and extensions towards more sophisticated alignment techniques [31].

In this thesis we further extend this work in two ways:

1. Based on the general framework of Algebraic Dynamic Programming (ADP) [12] for dynamic programming on sequential data we provide a powerful theory for alignment distance learning.

2. We extend the learning scheme beyond the Relational Generalized Learning Vector Quantization (RGLVQ) framework towards structure metric learning for k-Nearest Neighbor (k-NN) classifiers.

Thereby we demonstrate the broad applicability of alignment distance learning.

1.1 Related Work

1.1.1 Sequence Alignment Algorithms

Sequence alignment algorithms are ubiquitous tools in domains such as bioinformatics (e.g. [30], [38] or [14]), text processing (e.g. Edit- or Levenshtein-distance [11, 22]) and audio processing (e.g. Dynamic Time Warping [41]). The aim of all these algorithms is to *extend* two input sequences $\bar{x}$ and $\bar{y}$ over some common alphabet $\Sigma_\times$, such that matching elements of both sequences are *aligned*.

An extension of some sequence $\bar{x}$ is some longer sequence $\bar{x}^*$, probably over a larger alphabet, that contains all elements of $\bar{x}$ in the same order. Alignment schemes now introduce a cost function d on the alphabet $\Sigma_\times$, such that $d(\bar{x}_i, \bar{y}_j)$ is minimal if and only if $\bar{x}_i = \bar{y}_j$. An alignment algorithm then constructs extensions $\bar{x}^*$ and $\bar{y}^*$ of the input sequences with equal length and minimal cost

$$\sum_{o=1}^{|\bar{x}^*|} d(\bar{x}^*_o, \bar{y}^*_o) \tag{1.1}$$

"distance" in their mathematically strict sense. This leads to a somewhat blurry use of the terms within this thesis as well. We investigate the metric properties of alignment distances in more detail in Section 2.5 on page 32.

This minimal cost is the *alignment distance.*

As an example, consider the simple *String Edit Distance*[11, 22], where the input sequences consist of characters (i.e. $\Sigma_\times := \{\texttt{'A'}, \ldots, \texttt{'z'}\}$. The *String Edit Distance* allows to extend the input sequences by introducing *gaps*, which we denote as "$-$". d is defined as

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

The optimal alignment of two input sequences $\bar{x}$ and $\bar{y}$ with lengths M and N can be calculated using a simple dynamic programming scheme:

$$\begin{aligned} D(0,0) &= 0, \\ D(0,j) &= j, \\ D(i,0) &= i, \\ D(i+1,j) &= \min\{D(i,j) + d(x_{i+1}, y_{j+1}), \\ &\quad D(i+1,j) + 1, \\ &\quad D(i,j+1) + 1\} \end{aligned}$$

where $D(M, N)$ is the *String Edit Distance* between the input sequence $\bar{x}$ and $\bar{y}$.

However, as described by Gotoh [14], such a simple alignment measure is susceptible to length differences of the input sequences: If one input sequence is much longer than the other, many gaps have to be introduced, which might limit the interpretability of the output alignment. In such situations it is often desirable to encourage the alignment algorithm to skip consecutive parts of the longer input sequence and thus identify *regions of interest* that can properly be aligned with the shorter input sequence. Consider the example of the two text sequences

```
The fairy went home.
```

versus

```
After thirty adventures the fairy, overwrought but happy with
 her achievements, went back to her nearly forgotten homestead.
```

According to the *String Edit Distance*, the cheapest extensions of these sequences might look like this:

```
--T----h--------e------ ----fairy- ----w---------------------
After thirty adventures the fairy, overwrought but happy with

--e-----------nt------------ ---h-----------o----------me-----.
 her achievements, went back to her nearly forgotten homestead.
```

However, it would be desirable to produce an alignment like this:

```
----------------------- The fairy----------------------------
After thirty adventures the fairy, overwrought but happy with

----------------- went----------------------------- home-----.
 her achievements, went back to her nearly forgotten homestead.
```

In the literature, several approaches have been presented to encourage alignment algorithms to generate consecutive gaps: Smith and Waterman [38] have suggested an extended algorithm that can skip prefixes and suffixes of both input sequences for a fixed cost, independent of the lengths of these pre- und suffixes. In other words: If the String Edit Distance of the prefixes or suffixes becomes too high, they are skipped/ignored. This way the algorithm can identify the mid parts of both sequences, which allow a relatively cheap alignment (so-called *Local Alignment*).

Building upon this approach Gotoh [14] suggested an affine scoring scheme for gaps, where the first gap that is used is quite expensive while every consecutive gap afterwards has a discounted cost. This approach encourages long gap regions to justify the “initial investment” of the gap opening cost. The affine alignment algorithm presented in this work is based on Gotohs approach.

1.1.2 Algebraic Dynamic Programming

Algebraic Dynamic Programming (ADP) has been introduced by Giegerich, Meyer, and Steffen [12] as “a discipline of dynamic programming over sequence data”. It expresses possible alignments between input sequences as trees, which are produced by an underlying regular tree grammar. The alignment distance is determined by selecting the cheapest tree according to an objective function defined on those trees. Thereby they distinguish between the general alignment scheme (grammar) and the objective function (algebra) in a clean way and allow for simple combinations of alignment schemes with different objective functions and even coupling of different objective criteria (so-called algebra products).

Algebraic Dynamic Programming (ADP) is accompanied by powerful theoretic work, such as an algebra-based formulation of Bellman’s principle of optimality [3]. More recently, Sauthoff [37] provided an ADP programming language and compiler, enabling rapid prototyping of sequence alignment algorithms.

It should be noted that ADP applications are much broader than mere sequence alignment and include sophisticated biological tasks, such as abstract shape analysis [20] and pseudoknot folding [33]. For reasons of simplicity, though, we exclusively focus on sequence alignment restrict the broad definitions of ADP to optimally match this setting.

1.1.3 Similarity-Based Classification

Classifying data based on similarities or dissimilarities is a common technique in machine learning, applied in popular methods like k-Nearest Neighbor (k-NN) or k-means classification [9, 24]. In the past decades the topic has recieved heightened attention in the form of kernel methods [19], most notably the support vector machine [40]. Kernels apply more restrictions on the dissimilarity function, which an alignment distance does not satisfy[2]. Other modern approaches do not require these additional restrictions, such as Relational Generalized Learning Vector Quantization (RGLVQ) [17]. In this work we focus on the well-established k-NN method.

1.1.4 Representation Learning

As Pękalska [32] points out, considering distances or similarities between data points rather than explicit feature vectors, is equivalent to an alternative data representation. From that perspective metric learning is a form of representation learning, as changes of metric parameters induce changes in the implicit representation of the data. Still, representation learning in the strict sense focuses on learning explicit features, as is the case in auto-encoders and deep networks. An overview of techniques for representation learning has been provided by Bengio, Courville, and Vincent [4].

1.1.5 Vectorial Metric Learning

If the data is given in the form of feature vectors, several approaches are available to learn a suitable metric. Recent overview papers on this topic have been provided by Bellet, Habrard, and Sebban [1] and Kulis [21]. Usually such metric learning schemes extend the notion of the euclidian distance metric towards a more general quadratic form:

$$D(\vec{x}, \vec{y}) := \sqrt{(\vec{x} - \vec{y})^T M (\vec{x} - \vec{y})} \tag{1.2}$$

The learning process then has the aim to adapt the (symmetric, positive-semidefinite) matrix M in order to optimize some objective function. An example for such a learning scheme is the work of Weinberger and Saul [43], which provides the cost function we use in this thesis.

1.1.6 Sequence Alignment Distance Learning

Bellet, Habrard, and Sebban [1] list several methods that are more specifically geared towards parameter learning for alignment distances. Interestingly, these methods mostly rely on a stochastic definition of alignment algorithms and expectation maximization schemes (e.g. Habrard et al. [16], Takasu, Fukagawa, and Akutsu [39], and Bernard et al. [5]). While such approaches can be applied

[2]However, there are approaches to derive kernels based on alignment distances, such as [34, 10].

to classification tasks by demanding high intra-class similarity and low inter-class similarity between datapoints, they do not provide a gradient of the alignment distance which could be plugged into other cost functions.

Saigo, Vert, and Akutsu [34] do calculate a gradient over a Local Alignment Kernel, which makes their work quite similar to the one presented within this thesis. However, they do not use the gradient to optimize classification performance and lack the more general framework of ADP.

Bellet, Habrard, and Sebban [2] introduce a gradient-based approach to learn String Edit Distance parameters, which they accompany with strong theoretical results and directly connect to classification performance. Unfortunately they do not support more refined alignment algorithms like local or affine ones.

1.2 Scientific Contribution and Overview

The key contributions of this thesis are

1. extending the theory of alignment distance learning towards a general class of alignment algorithms using ADP,
2. applying this learning scheme on the sophisticated method of Affine Alignment,
3. providing a fast and very flexible implementation of various alignment algorithms and their gradients in form of the *TCS Alignment Toolbox*[3] and
4. extending the approach towards k-NN classification.

We proceed as follows: In Chapter 2 we introduce a simplified version of ADP first and use it to define the subclass of alignment algorithms considered within this thesis. To optimize the parameters of the alignment distance for classification we calculate the gradient of the Large Margin Nearest Neighbor (LMNN) cost function [43]. We achieve a differentiable alignment distance using the soft minimum approximation. The implementation of the *TCS Alignment Toolbox* is described in Chapter 3. Using this implementation we proceed to experiments on three diverse datasets in Chapter 4, after which we conclude the thesis in Chapter 5.

[3]`http://openresearch.cit-ec.de/projects/tcs`

Chapter 2

Theory

Within this work we do not only investigate the specific case of Affine Sequence Alignment (see Section 2.3 on page 25), but rather a more general class of sequence alignment schemes, which can be expressed in Algebraic Dynamic Programming (ADP). For such alignment schemes we provide a generic dynamic programming algorithm to calculate them efficiently (see Section 2.4 on page 28), generic results on metric properties (see Section 2.5 on page 32) and an efficient gradient calculation method (see Section 2.7 on page 42), which enables machine learning on parameters of the alignment scheme.

We begin with a fairly general definition of "sequence" in Section 2.1, which form the input for our alignment algorithms.

Section 2.2 is devoted to Algebraic Dynamic Programming. The theory of ADP has been thoroughly investigated by Giegerich, Meyer, and Steffen [12]. It provides a powerful framework to develop and implement dynamic programming algorithms on sequential data. Due to the scope of this thesis, however, we will not address the full expressive power of ADP, but rather a simplified version of it.

Building upon these definitions we systematically develop Affine Sequence Alignment in Section 2.3. In Section 2.4 we describe how alignment schemes expressed in the ADP framework can be translated to efficient dynamic programming algorithms. We investigate the (metric) properties of the alignment distance extensively in Section 2.5.

Proceeding to the actual adaption of alignment algorithms we further provide a simple cost function for our further experiments (Section 2.6), which we optimize by gradient descent. We calculate the gradient in Section 2.7 and achieve a differentiable alignment distance using the soft minimum approximation described in Section 2.8.

2.1 Sequences

Our input sequences are defined as follows:

Definition 1. An *alphabet* Σ is some arbitrary set. Elements $a, b \in \Sigma$ are called *values.*

α	β	γ	Phase	Comment
30	10	10	pause	Begin of pause before first movement.
30	10	10	pause	
40	10	10	movement	Begin of movement. Quite rapidly at the start.
45	10	10	movement	
50	10	10	movement	
50	10	10	pause	End of movement.

Table 2.1: Some toy data describing the movement of a very simple robotic arm with three joints. The joint angles α, β, γ at a given point in time are listed in the first three columns. The fourth column describes qualitatively whether the arm is moving or not. The last column contains a comment as it might be given by some robotics researcher.

Definition 2. The *keyword set* is an arbitrary, finite set. Every element $\kappa \in K$ is called a *keyword* κ is some arbitrary set.

During our theoretical considerations here a keyword has the function of an index and can be seen as a natural number. In practical applications a string captures the meaning of the respective alphabet better. Each keyword references one alphabet. We denote the alphabet corresponding to keyword κ as Σ_κ.

Definition 3. The *node set* is defined as the Cartesian product of all alphabets:

$$\Sigma_\times = \bigtimes_{\kappa \in K} \Sigma_\kappa \tag{2.1}$$

Each element $x \in \Sigma_\times$ is called a *node*. Each entry in such a vector of values is denoted as x^κ.

Definition 4. A *sequence* $\bar{x}$ is a succession (or a vector) of nodes $\bar{x} = (x_1, \ldots, x_M)$ with $M \geq 0$. We call M the *length* of $\bar{x}$.

In that sense a sequence can be written as a matrix, where each row is a node. Note that the order of columns in that matrix does not matter, while the order of rows is important.

As an example consider movement data from a robotic arm: We might have real numbers for the joint angles of the robotic arm, which might be accompanied by a semantic labeling of the movement phase as well as comments of the robotic researcher. Some toy data for this example is shown in Table 2.1.

2.2 Algebraic Dynamic Programming

Algebraic Dynamic Programming neatly separates concepts, which facilitates loose coupling in implementations and high compatibility. The concepts are:

- The signature (Section 2.2.1 on the facing page) defines function templates and serves as an interface between the algebra and grammar.

- The algebra (see Section 2.2.2 on the next page) implements the functions specified by the signature.
- The grammar (see Section 2.2.4 on page 20) specifies which function can be applied in which situation and thereby provides the search space of possible alignments.

When we combine all these ingredients, we can define the corresponding alignment distance and ADP problem (see Section 2.2.5 on page 24 and 2.2.6 on page 24 respectively).

2.2.1 Signature

Definition 5. We define an *arity* as a tuple

$$(\mathcal{A}, \mathcal{A}') \in \mathbb{N}^2 \tag{2.2}$$

This definition implicitly specifies the input set for a function, as can be seen in the Definitions 8 and 9 on page 19.

Definition 6. The *alignment end* nil or *terminating template* is defined as the tuple

$$(\text{nil}, (0, 0)) \tag{2.3}$$

Definition 7. A *signature* $\mathcal{T}$ is a set of *function templates* t, which we define as a tuple of a name and an arity $(\mathcal{A}, \mathcal{A}')$. Furthermore we require for the aritys of function templates:

$$\mathcal{A}, \mathcal{A}' \in \{0, 1\} \tag{2.4}$$

$$\mathcal{A} + \mathcal{A}' > 0 \tag{2.5}$$

For the purpose of this thesis we mostly use a fixed signature, which we will re-use in the course of this thesis:

- rep (replacement) with the arity $(1, 1)$
- del (deletion) with the arity $(1, 0)$
- ins (insertion) with the arity $(0, 1)$
- skip_del (skip-deletion) with the arity $(1, 0)$
- skip_ins (skip-insertion) with the arity $(0, 1)$

We will refer to this particular signature as $\mathcal{T}^*$ or sig_alignment.

Even though it is not directly expressed in the definition, the function template names already hint at the *semantics* of these function templates. They relate to a different understanding of the general sequence alignment problem: Given two input sequences $\bar{x}$ and $\bar{y}$, how can the first sequence be *transformed* to the second sequence by applying a succession of discrete transformation operations, such that the sum of all operation costs is as low as possible. Taking this point of view the different function templates (or *operators* as they are called by Giegerich, Meyer, and Steffen [12]) have the following semantics:

- An *alignment end* obviously just ends the alignment.
- A *replacement* means replacing one node in the first sequence by one node of the second sequence.
- A *deletion* means deleting one node in the first sequence (denoted as a $-$ in the second sequence).
- An *insertion* means inserting one node from the second sequence into the first sequence (denoted as a $-$ in the first sequence).
- A *skip-deletion* means skipping an irrelevant node in the first sequence (denoted as a _ in the second sequence).
- A *skip-insertion* means skipping an irrelevant node in the second sequence (denoted as a _ in the first sequence).

Consider the example of the String Edit Distance of the two input strings `abc` and `acd`. On optimal alignment looks like this:

```
abc-
a-cd
```

This is equivalent to saying that we

- *replaced* `a` with `a`,
- *deleted* `b`,
- *replaced* `c` with `c` and
- *inserted* `d` into the first sequence.

Algorithms, that explicitly construct such transformation or *edit scripts* are called *edit distance algorithms.* Note that we do not formally proof the equivalence of edit distance algorithms and the alignment schemes discussed in this thesis. However, we find this alternative perspective helpful to better understand the concepts of ADP and occasionally refer to an edit distance interpretation.

2.2.2 Algebra

Let $\mathcal{T}$ be a signature and $(\mathcal{A}_t, \mathcal{A}'_t)$ be the arity of function template t. Let K be the keyword set and κ a keyword with the associated alphabets Σ_κ.

Definition 8. We define the *algebra function input set* $\mathcal{I}(t)$ as:

$$\mathcal{I}(t) := \bigtimes_{\alpha=1}^{\mathcal{A}_t} \Sigma_\times \times \mathbb{R}^+ \times \bigtimes_{\alpha'=1}^{\mathcal{A}'_t} \Sigma_\times \tag{2.6}$$

Definition 9. We further define the *comparator function input set* $\mathcal{I}_c^\kappa(t)$ as:

$$\mathcal{I}_c^\kappa(t) := \bigtimes_{\alpha=1}^{\mathcal{A}_t} \Sigma_\kappa \times \bigtimes_{\alpha'=1}^{\mathcal{A}'_t} \Sigma_\kappa \tag{2.7}$$

Definition 10. Accordingly a *comparator function* c_t^κ is some function

$$c_t^\kappa : \mathcal{I}_c^\kappa(t) \to [0, 1] \tag{2.8}$$

Definition 11. We define an *algebra* $\mathcal{F}(\mathcal{T})$ as a set of comparator functions c_t^κ, one for each tuple $(t, \kappa) \in \mathcal{T} \times K$.

Definition 12. This in turn gives rise to *algebra functions* (one for each function template) as follows:

$$d_t : \mathcal{I}(t) \to \mathbb{R}^+ \tag{2.9}$$

$$d_t(\circ, s, \circ) := s + \sum_{\kappa \in K} g_\kappa c_t^\kappa(\circ, \circ) \tag{2.10}$$

where $\circ$ is a placeholder for the remaining input arguments at that position and $(g_\kappa)_{\kappa \in K}$ are *keyword weights*, one for each keyword κ. Further we restrict keyword weights with the following conditions:

$$\forall \kappa \in K : g_\kappa \in [0, 1] \tag{2.11}$$

$$\sum_{\kappa \in K} g_\kappa = 1 \tag{2.12}$$

These definitions yield an intuitive interpretation: d calculates the accumulation of all previous alignment operations plus the cost of the current alignment operation t. The cost of the current operation in turn is calculated as the weighted sum of all comparator function outputs with the keyword weights $(g_\kappa)_{\kappa \in K}$. Finally, we can interpret the comparator function output as the cost for applying the current alignment operation t on the current values in both sequences.

Again, this is a strong restriction of the algebra definition given by Giegerich, Meyer, and Steffen [12]: They permit arbitrary output sets for algebras, which enables them to frame e.g. printouts of optimal alignments as a special kind of algebra. For the purposes of this thesis, however, the restriction to (positive) real numbers is sufficient, though.

2.2.3 Choice Function

Definition 13. Let $\mathcal{P}(\mathbb{R}^+)$ be the set of all possible multisets of non-negative real numbers. Then we define a *choice function* as a function:

$$h : \mathcal{P}(\mathbb{R}^+) \to \mathbb{R}^+ \tag{2.13}$$

This definition makes the name *choice function* quite intuitive: It chooses one value from a multiset of values.[1] It restricts the definitions given by Giegerich, Meyer, and Steffen [12] once more: They consider functions which return more than one value as well (especially the identity function). However, as we have already fixed the output of an algebra to a real number our intent with the choice function is quite clear: We would like to find the cheapest possible alignment. Therefore, the ideal choice function is:

$$h[\theta_1, \dots, \theta_L] := \min[\theta_1, \dots, \theta_L] \tag{2.14}$$

Note that the definition of h over a multiset[2] of values also makes it invariant regarding changes of order within the input arguments.

2.2.4 Grammar

An ADP *grammar* is a *regular tree grammar*, which expresses the language of all possible alignments. According to Giegerich, Meyer, and Steffen [12] the original definition of such regular tree grammars has been introduced by Brainerd [7] and Giegerich and Schmal [13] have applied some modifications to make such languages specify valid terms.

Definition 14. Very similar to Giegerich, Meyer, and Steffen [12] we define an ADP grammar as a tuple

$$\mathcal{G} = (\Phi, \mathrm{A}^*, K, \{(\kappa, \Sigma_\kappa)\}_{\kappa \in K}, \mathcal{T}, \Delta) \tag{2.15}$$

where

- Φ is a set of *nonterminal symbols*,
- $\mathrm{A}^* \in \Phi$ is the *axiom* or start nonterminal symbol,
- K is a keyword set,
- $(\kappa, \Sigma_\kappa)_{\kappa \in K}$ are tuples mapping the keywords to respective alphabets, thus defining the node set $\Sigma_\times$,
- $\mathcal{T}$ is a signature and
- Δ are *production rules*.

To denote these production rules we use the Bellman's Gap Language (GAP-L) as introduced in Chapter 4 of the PhD thesis by Sauthoff [37]. As in GAP-L we do not regard terminal symbols as part of the grammar but use a fixed global set of terminal symbols, namely:

- `NODE` which is some node from the node set.

[1]Note that we do not require the output value to be part of the input multiset, though. $h[\theta_1, \dots, \theta_L] = 0$ is a valid choice function as well.

[2]In the course of this thesis we will denote multisets with square brackets $[\dots]$ to avoid confusion with simple sets.

- `EMPTY` which is the empty word.

Each production rule is denoted as

```
A = t(<TL,TR>, B);
```

or

```
A = B;
```

or

```
A = nil(<EMPTY,EMPTY>);
```

where A and B are nonterminal symbols and TL and TR are terminal symbols. As before, t refers to some function template in the signature $\mathcal{T}$. The $<$ $>$ syntax means that the left terminal symbol in the enclosed tuple will be applied to the left input sequence, while the right terminal symbol in the enclosed tuple will be applied to the right input sequence[3]. Obviously, the tuple has to match the arity of the enclosing t, so

```
del(<EMPTY,EMPTY>, B);
```

is not valid, because it forwards no input argument to the left side of the del function template.

We introduce two further definitions on grammars for our subsequent theoretical work:

Definition 15. A nonterminal symbol A in grammar $\mathcal{G}$ is called *accepting* if a production rule of the form

```
A = nil(<EMPTY,EMPTY>);
```

exists in Δ.

Definition 16. Let $\mathcal{G}$ be a grammar. A *loop* is defined as a succession of nonterminal symbols $\mathrm{A}_1, \ldots, \mathrm{A}_R \in \Phi$ with $R \geq 1$, such that $\mathrm{A}_1 = \mathrm{A}_R$ and for every $1 \leq r < R$ there is a production rule of the form

$$\mathrm{A}_r = \mathrm{A}_{r+1} \tag{2.16}$$

in Δ.

In this thesis we will assume that all grammars do not contain loops, as they make the calculation of an alignment distance impossible. The existence of loops can be checked efficiently by Algorithm 6 on page 60.

For convenience reasons GAP-L also introduces the | symbol as a shorthand for multiple rules, so

```
A = t(<TL,TR>, B) | t2(<TL2,TR2>, B2);
```

is just a shorthand for

```
A = t(<TL,TR>, B);
A = t2(<TL2,TR2>, B2);
```

[3]This is called *Multi-Track* by Sauthoff [37], as two input sequences are parsed at the same time.

Grammar 2.1: An ADP Grammar $\mathcal{G}_{\text{Glob}}$ for Global Sequence Alignment or String Edit Distance.

```
grammar Glob uses sig_alignment (axiom = ALI){
ALI =    rep(<NODE,NODE>, ALI) |
         del(<NODE,EMPTY>, ALI) |
         ins(<EMPTY,NODE>, ALI) |
         nil(<EMPTY,EMPTY>);
}
```

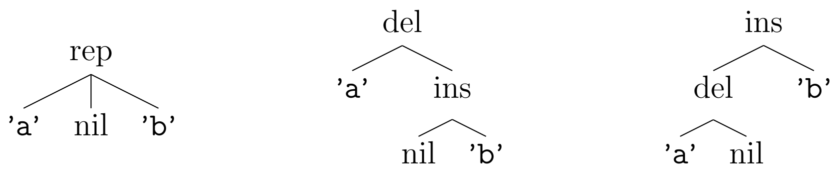


Figure 2.1: Some example trees that can be constructed using the grammar $\mathcal{G}_{\text{Glob}}$.

Note that GAP-L also requires users to explicitly indicate whether a choice function should be applied for the current rule or not. We omit this particular language concept and assume that the choice function should be applied every time. The application of the choice function is discussed in more detail in Sections 2.2.6 on page 24 and 2.4 on page 28

As a first example of such a grammar consider the very simple example of *Global Sequence Alignment* as introduced by Needleman and Wunsch [30]. In the abstract ADP representation this is equivalent to the String Edit Distance or Levenshtein-distance [22]. The production rules are shown in Grammar 2.1.

Lets construct some example trees with these production rules. To make the examples more palpable we assume for the moment only having one keyword κ. As the respective alphabet we define a set of discrete characters:

$$\Sigma_\kappa := \{\texttt{'a'}, \texttt{'b'}\} \tag{2.17}$$

Accordingly, our example grammar is defined as

$$\mathcal{G}_{\text{Glob}} = (\{ALI\}, ALI, \{\kappa\}, (\kappa, \{\texttt{'a'}, \texttt{'b'}\}), \mathcal{T}^*, \Delta_{\text{Glob}}) \tag{2.18}$$

The resulting trees are shown in Figure 2.1. Each of these trees is an element of the *language* $\mathcal{L}(\mathcal{G}_{\text{Glob}})$.

Definition 17. We define a *language* $\mathcal{L}$ over some grammar $\mathcal{G}$ as the set of all trees (without nonterminal symbols) that can be constructed starting at the axiom A^* and iteratively applying production roles from Δ.

This language does not yet depend on any specific input. Each of the trees in $\mathcal{L}(\mathcal{G})$ can be interpreted as *some* alignment, but the connection to input sequences and an algebra has yet to be made. In the course of this thesis we will call the trees *alignments* as well.

We furthermore want to define a particular set of alignments, which will come in handy later on:

Definition 18. Let $x, y \in \Sigma_\times$. An alignment T is called *trivial replacement alignment* if it is of the form:

$$T = \text{rep}(x, T', y) \tag{2.19}$$
$$T = \text{nil}() \tag{2.20}$$

where T' is a trivial replacement alignment itself.

Regarding this class we can conclude:

Lemma 1. *Every alignment that only consists of* rep *and/or* nil *operations is a trivial replacement alignment (independent of the grammar that produced it).*

Proof. Our claim follows from the grammar rules that are possible. Let T be some alignment only consisting of rep and/or nil operations. Now first assume that it does not contain a nil operation. In that case the tree was produced only by rules of the form

```
A = rep(<NODE,NODE>, B);
```

As can be seen this rule necessarily contains a nonterminal symbol on the right-hand side such that only applying these rules never leads to a valid alignment as defined in Definition 17 on the preceding page.

Thus the alignment contains at least one nil operation. Now consider the first nil. As defined above the only production rules producing nil are of the form:

```
A = nil(<EMPTY,EMPTY>);
```

Apparently, there is no nonterminal symbol left on the right hand side. Therefore, no other content can be produced anymore. Thus we can conclude that T contains a succession of rep operations follows by exactly one nil at the end, which makes T a trivial replacement alignment. $\square$

Dynamic Time Warping

Note that our current notion of grammar and algebra does not permit a straightforward definition of Dynamic Time Warping (DTW). It is possible to express Dynamic Time Warping (DTW) in ADP terms, though, if one applies a little trick: If the algebra has access to the input sequences we can define the following algebra $\mathcal{F}_{\text{DTW}}$:

$$c^{\kappa}_{\text{del}}(x_i) := c^{\kappa}_{\text{rep}}(x_i, y_j) \tag{2.21}$$
$$c^{\kappa}_{\text{ins}}(y_j) := c^{\kappa}_{\text{rep}}(x_i, y_j) \tag{2.22}$$
$$\tag{2.23}$$

Then DTW can be expressed using grammar 2.1 on the facing page ($\mathcal{G}_{\text{Glob}}$).

2.2.5 Alignment Distance

Let $\mathcal{G} = (\Phi, \mathrm{A}^*, K, \{(\kappa, \Sigma_\kappa)\}_{\kappa \in K}, \mathcal{T}, \Delta)$ be some grammar.

Definition 19. We define the *yield* $\mathcal{Y}(T)$ of some tree $T \in \mathcal{L}(\mathcal{G})$ as the two sequences that emerge as the concatenation of all terminal leafs on the left side (left sequence) and the right side (right sequence) of the tree. More formally we can define the yield recursively as follows:

$$\mathcal{Y}(\text{nil}) := (\epsilon, \epsilon) \tag{2.24}$$

$$\mathcal{Y}(t(\circ, T', \circ)) := (\circ, \circ) + \!\!\!+\, \mathcal{Y}(T') \tag{2.25}$$

where $+\!\!\!+$ means concatenating the entries in the left-hand tuple to the sequences in the right-hand tuple.

Consider the example trees in Figure 2.1 on page 22. All of them have the yield

$$(\texttt{'a'}, \texttt{'b'}) \tag{2.26}$$

Definition 20. Let $\mathcal{F}(\mathcal{T})$ be some algebra over the signature $\mathcal{T}$. Similarly to the yield we can define the *application* of $\mathcal{F}$ on T as follows:

$$\mathcal{F}(\text{nil}()) := 0 \tag{2.27}$$

$$\mathcal{F}(t(\circ, T', \circ)) := d_t(\circ, \mathcal{F}(T'), \circ) \tag{2.28}$$

Definition 21. Let $\mathcal{F}(\mathcal{T})$ be some algebra over the signature $\mathcal{T}$ and let h be a choice function. Further let $\bar{x}, \bar{y}$ be some input sequences of nodes from the node set $\Sigma_\times = \bigtimes_{\kappa \in K} \Sigma_\kappa$. Then the *alignment distance* D is defined as

$$D(\bar{x}, \bar{y}) := h[\mathcal{F}(T) | T \in \mathcal{L}(\mathcal{G}), \mathcal{Y}(T) = (\bar{x}, \bar{y})] \tag{2.29}$$

In other words: The alignment distance is the algebra application of the best (according to the choice function h) alignment T that can be produced by the grammar, such that T has the input sequences as its yield.

2.2.6 Algebraic Dynamic Programming Problems

Definition 22. Let $\bar{x}, \bar{y}$ be some input sequences. Then the *Algebraic Dynamic Programming problem* is defined as calculating the alignment distance between $\bar{x}$ and $\bar{y}$ in polynomial time with respect to the input sequence lengths M and N.

A naive algorithm to compute the alignment distance is Algorithm 1 on the facing page.

As an example consider the single-keyword setting from above with $\Sigma_\kappa := \{\texttt{'a'}, \texttt{'b'}\}$ and the grammar $\mathcal{G}_{\text{Glob}}$. Additionally we define the trivial algebra:

$$c^\kappa_{\text{rep}}(a, b) := \begin{cases} 0 & \text{if } a = b \\ 1 & \text{if } a \neq b \end{cases} \tag{2.30}$$

$$c^\kappa_{\text{del}}(a) = c^\kappa_{\text{skip\_del}}(a) := 1 \tag{2.31}$$

$$c^\kappa_{\text{ins}}(b) = c^\kappa_{\text{skip\_ins}}(b) := 1 \tag{2.32}$$

Algorithm 1 A naive algorithm to calculate the alignment distance, which does not run in polynomial time.

```
Let $\mathcal{G}$ be our input grammar oder signature $\mathcal{T}$, $\mathcal{F}$ be our algebra over $\mathcal{T}$, $h$ our choice function and $\bar{x}$ and $\bar{y}$ our input sequences.
Calculate $\mathcal{L}(\mathcal{G})$.
Initialize $\Theta$ as an empty multiset.
for $T \in \mathcal{L}(\mathcal{G})$ do
    if $\mathcal{Y}(T) = (\bar{x}, \bar{y})$ then
        Add $\mathcal{F}(T)$ to $\Theta$.
    end if
end for
return $h(\Theta)$.
```

where $a, b \in \Sigma_\kappa$. As choice function we use min and as input sequences we consider $\bar{x} =$ `'a'` and $\bar{y} =$ `'b'`. Figure 2.1 on page 22 shows all trees from $\mathcal{L}(\mathcal{G})$ that have $(\bar{x}, \bar{y})$ as their yield. The algebra results for the trees are (from left to right) 1, 2 and 2. Thus the choice function returns:

$$h[1,2,2] = \min[1,2,2] = 1 \tag{2.33}$$

This is the optimum alignment distance between both input sequences. Obviously, this naive approach does not solve the ADP problem: It does not finish in polynomial time (with respect to the lengths of the input sequences). As the name of ADP suggests, the key to efficiency is decomposing the problem in a dynamic programming fashion. We consider the decomposition in more detail in Section 2.4 on page 28 and the matter of efficiency in Section 2.4.2 on page 32.

2.3 Affine Sequence Alignment

Now lets systematically extend the example grammar $\mathcal{G}_{\text{Glob}}$ to a grammar for Affine Sequence Alignment, which we can use for the experiments later on. The first extension is based on a simple observation: Consider the middle and the right tree shown in Figure 2.1 on page 22. Both trees have the same yield as well as the same algebra result. This is necessarily the case, as can be shown easily:

$$\begin{aligned}
\mathcal{Y}(\text{del}(a, \text{ins}(T', b))) &= (a, \epsilon) +\!\!+ (\epsilon, b) +\!\!+ \mathcal{Y}(T') && (2.34)\\
&= (a, b) +\!\!+ \mathcal{Y}(T') && (2.35)\\
&= (\epsilon, b) +\!\!+ (a, \epsilon) +\!\!+ \mathcal{Y}(T') && (2.36)\\
&= \mathcal{Y}(\text{ins}(\text{del}(a, T'), b)) && (2.37)
\end{aligned}$$

Grammar 2.2: A refined version of Grammar 2.1 on page 22 $\mathcal{G}_{\text{Glob2}}$ enforcing that deletions happen in front of insertions.

```
grammar Glob2 uses sig_alignment (axiom = ALI){
ALI =    rep(<NODE,NODE>, ALI) |
         del(<NODE,EMPTY>, ALI) |
         ins(<EMPTY,NODE>, INS) |
         nil(<EMPTY,EMPTY>);

INS =    ins(<EMPTY,NODE>, INS) |
         rep(<NODE,NODE>, ALI) |
         nil(<EMPTY,EMPTY>);
}
```

and

$$\begin{aligned}
\mathcal{F}(\text{del}(a, \text{ins}(T', b))) &= d_{\text{del}}(a, d_{\text{ins}}(\mathcal{F}(T'), b)) && (2.38)\\
&= \sum_{\kappa \in K} g_\kappa c^\kappa_{\text{del}}(a^\kappa) + \sum_{\kappa \in K} g_\kappa c^\kappa_{\text{ins}}(b^\kappa) + \mathcal{F}(T') && (2.39)\\
&= d_{\text{ins}}(d_{\text{del}}(a, \mathcal{F}(T')), b) && (2.40)\\
&= \mathcal{F}(\text{ins}(\text{del}(a, T'), b)) && (2.41)
\end{aligned}$$

So adjacent deletion and insertion operations can be reversed without changing the ADP problem result (independent of the algebra). Given that observation we can reduce the search space by enforcing that deletions happen in front of insertions. The resulting grammar $\mathcal{G}_{\text{Glob2}}$ is shown in Grammar 2.2. Applied on the example, this grammar does not produce the rightmost tree in Figure 2.1 on page 22 anymore.

Smith and Waterman [38] suggested a *Local Sequence Alignment* in the sense that the beginning and the end of both input sequences can be skipped at low costs such that the algorithm identifies those subsequences of the input sequences that match best and ignores the rest. Grammar 2.3 on the next page is inspired by that approach: It allows to skip the beginning of both sequences (but enforces that skip-deletions happen before skip-insertions) until a first replacement occurs. After a first replacement the grammar is essentially equivalent to $\mathcal{G}_{\text{Glob2}}$ with the sole exception that from the nonterminal symbol ALI we are allowed to switch to SKIPDEL_END. After that switch only skip-deletions and skip-insertions may occur. If the algebra ensures that skip-deletions and skip-insertions are cheap the grammar will lead to the desired behavior. Figure 2.2 on page 28 shows an example tree for this grammar.

Our final extension is motivated by the Affine Alignment algorithm by Gotoh [14]. Here we allow skips in the middle of the alignment as well, if the skip-regions have non-trivial lengths. In Grammar 2.4 on page 29 we expect at least three skip-deletions or skip-insertions. In effect this grammar tries to find those subsequences of both input sequences that match best. To keep the number of such subsequences small, Gotoh [14] introduced the idea of a skip opening cost: If opening skip-regions is relatively expensive it is

Grammar 2.3: An ADP Grammar $\mathcal{G}_{\text{Local}}$ for Local Sequence Alignment as suggested by Smith and Waterman [38].

```
grammar local uses sig_alignment (axiom = SKIPDEL_START){
//skip the start
SKIPDEL_START = skip_del(<NODE,EMPTY>, SKIPDEL_START) |
                SKIPINS_START;

SKIPINS_START = skip_ins(<EMPTY,NODE>, SKIPINS_START) |
                rep(<NODE,NODE>, ALI) |
                nil(<EMPTY,EMPTY>);

//align in the middle
ALI =           del(<NODE,EMPTY>, DEL) |
                ins(<EMPTY,NODE>, INS) |
                rep(<NODE,NODE>, ALI) |
//start skipping the end
                SKIPDEL_END;

DEL =           del(<NODE,EMPTY>, DEL) |
                ins(<EMPTY,NODE>, INS) |
                rep(<NODE,NODE>, ALI);

INS =           ins(<EMPTY,NODE>, INS) |
                rep(<NODE,NODE>, ALI);

//skip the end
SKIPDEL_END =   skip_del(<NODE,EMPTY>, SKIPDEL_END) |
                SKIPINS_END;

SKIPINS_END =   skip_ins(<EMPTY,NODE>, SKIPINS_END) |
//here ends the alignment
                nil(<EMPTY,EMPTY>);
}
```

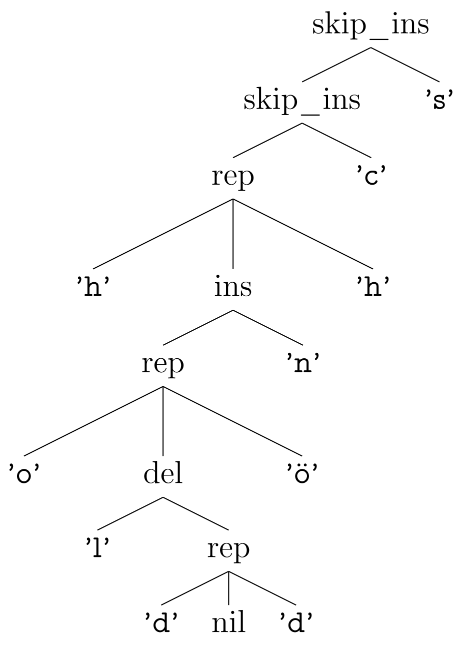


Figure 2.2: An example tree with the yield (`hold`, `schnöd`) using the grammar $\mathcal{G}_{\text{Local}}$.

better to continue an existing region than to open another one somewhere else. We incorporate this idea by enforcing a regular deletion or insertion operation before the production is allowed to continue with skip-deletions or skip-insertions.

2.4 Translation to Dynamic Programming

Our aim is to generically solve the ADP problem for some given grammar, algebra, choice function and input sequences. Fortunately this is quite straightforward given the theory we have introduced so far. Let

$$\mathcal{G} = (\Phi, \mathrm{A}^*, K, \{(\kappa, \Sigma_\kappa)\}_{\kappa \in K}, \mathcal{T}^*, \Delta)$$

be some grammar over the restricted signature $\mathcal{T}^*$. Let further $\bar{x}$ and $\bar{y}$ be our input sequences with lengths M and N. The key points are:

1. Every nonterminal symbol $\mathrm{A} \in \Phi$ is translated to a dynamic programming table A of size $(M+1) \cdot (N+1)$.

2. All tables corresponding to *accepting* nonterminal symbols A (see Definition 15 on page 21) get initialized with

$$A(M+1, N+1) \leftarrow 0 \tag{2.42}$$

3. All production rules in Δ that do not contain nil can be translated according to Table 2.2 on page 31.

Grammar 2.4: The ADP Grammar $\mathcal{G}_{\text{Affine}}$ for the Affine Sequence Alignment algorithm used in this thesis.

```
grammar affine uses sig_alignment (axiom = SKIPDEL_START){
//skip the start
SKIPDEL_START = skip_del(<NODE,EMPTY>, SKIPDEL_START) |
                SKIPINS_START;

SKIPINS_START = skip_ins(<EMPTY,NODE>, SKIPINS_START) |
                rep(<NODE,NODE>, ALI) |
                nil(<EMPTY,EMPTY>);

//align in the middle
ALI =           del(<NODE,EMPTY>, DEL) |
                ins(<EMPTY,NODE>, INS) |
                rep(<NODE,NODE>, ALI) |
//skip to the next interesting region with at least 3 skips
                del(<NODE,EMPTY>,
                skip_del(<NODE,EMPTY>,
                skip_del(<NODE,EMPTY>,
                SKIPDEL_MID))) |
//skip to the next interesting region with at least 3 skips
                ins(<EMPTY,NODE>,
                skip_ins(<EMPTY,NODE>,
                skip_ins(<EMPTY,NODE>,
                SKIPINS_MID))))
//start skipping the end
                SKIPDEL_END;

DEL =           del(<NODE,EMPTY>, DEL) |
                ins(<EMPTY,NODE>, INS) |
                rep(<NODE,NODE>, ALI);

INS =           ins(<EMPTY,NODE>, INS) |
                rep(<NODE,NODE>, ALI);

//continue skipping to an interesting region
SKIPDEL_MID =   skip_del(<NODE,EMPTY>, SKIPDEL_MID) |
                rep(<NODE,NODE>,ALI);

SKIPINS_MID =   skip_ins(<EMPTY,NODE>, SKIPINS_MID) |
                rep(<NODE,NODE>,ALI);

//skip the end
SKIPDEL_END =   skip_del(<NODE,EMPTY>, SKIPDEL_END) |
                SKIPINS_END;

SKIPINS_END =   skip_ins(<EMPTY,NODE>, SKIPINS_END) |
//here ends the alignment
                nil(<EMPTY,EMPTY>);
}
```

4. The value of a cell $A(i, j)$ is calculated as the application of the choice function on the translation of all production rules with the corresponding nonterminal symbol A on the left side.

5. We fill the dynamic programming tables by starting with the indices $i = M + 1$ and $j = N + 1$ for all nonterminal symbols and decreasing i and j iteratively until cell $(1, 1)$ is calculated for all tables.

6. The result can be found in cell $(1, 1)$ of the dynamic programming table representing the axiom.

A formal version is shown in Algorithm 2.

Algorithm 2 A generic dynamic programming algorithm that solves the algebraic dynamic programming problem (see Section 2.2.6 on page 24) corresponding to some given ADP grammar $\mathcal{G}$, algebra $\mathcal{F}$, choice function h and input sequences $\bar{x}$ and $\bar{y}$.

```
Let $\mathcal{G} = (\Phi, \mathrm{A}^*, K, \{(\kappa, \Sigma_\kappa)\}_{\kappa \in K}, \mathcal{T}^*, \Delta)$ be an ADP grammar over signature $\mathcal{T}^*$ and $\mathcal{F}$ be an algebra over the same signature and $\Sigma_\times$. Further let $h$ be a choice function.
Let $\bar{x}, \bar{y}$ be our input sequences over $\Sigma_\times$ with lengths $M$ and $N$.
for $\mathrm{A} \in \Phi$ do
    Initialize $A$ as a table of size $(M + 1) \times (N + 1)$.
    if A is accepting then
        $A(M + 1, N + 1) \leftarrow 0$.
    end if
end for
for $i \in (M + 1, \dots, 1)$ do
    for $j \in (N + 1, \dots, 1)$ do
        for $\mathrm{A} \in \Phi$ do
            Let $A$ be the dynamic programming table corresponding to A.
            Retrieve all applicable production rules from $\mathcal{G}$ that have A on the left-hand side. Let $L$ be the number of such rules. We associate each of these rules with an arbitrary index $l \in \{1, \dots, L\}$.
            for $l \in \{1, \dots, L\}$ do
                Calculate $\theta_l$ as the translation of the production rule with index $l$ as described in Table 2.2 on the next page.
            end for
            $A(i, j) \leftarrow h[\theta_1, \dots, \theta_L]$
        end for
    end for
end for
return $A^*(1, 1)$
```

There are two main technicalities that we have glossed over in Algorithm 2:

1. We are not allowed to iterate over the nonterminal symbols in arbitrary order in the inner loop, as rules of the form

Production Rule	**Translation**
B	$B(i, j)$
del(<NODE,EMPTY>, B)	$d_{\text{del}}(x_i, B(i+1, j))$
skip_del(<NODE,EMPTY>, B)	$d_{\text{skip\_del}}(x_i, B(i+1, j))$
ins(<EMPTY,NODE>, B)	$d_{\text{ins}}(B(i, j+1), y_j)$
skip_ins(<EMPTY,NODE>, B)	$d_{\text{skip\_ins}}(B(i, j+1), y_j)$
rep(<NODE,NODE>, B)	$d_{\text{rep}}(x_i, B(i+1, j+1), y_j)$

Table 2.2: The translation of a right side of a production rule to a term θ_l.

```
A = B;
```

imply a dependency of $A(i, j)$ on $B(i, j)$. Therefore, we have to calculate the latter one first. Fortunately we can detect those dependencies systematically in a simple preprocessing step that only depends on the production rules of the grammar. We provide pseudocode for this preprocessing in Algortihm 6 on page 60. Note that this pseudocode also prevents dependency loops (see Definition 16 on page 21).

2. More critical is the notion of "applicable production rules". Consider the example of Global Sequence Alignment as described in Grammar 2.1 on page 22. Which production rules can be applied depends on the available subsequence: If $i = M$ (that is: there is nothing left of the left input sequence) we can neither apply a replacement nor a deletion because we can not read from the left input sequence anymore. There is no trivial way to determine the applicable production rules in general, given some grammar. We provide rather inefficient pseudocode in Algorithm 3 on page 33. As the applicable production rules have to be retrieved in every calculation step, the efficient retrieval of applicable production rules is critical to the overall efficiency of the algorithm. In Section 3.3 on page 58 we discuss our implementation with respect to this problem.

To provide some more insight to this generic translation algorithm we provide an example calculation for the grammar $\mathcal{G}_{\text{Affine}}$ in Appendix B.

2.4.1 Bellman's Principle of Optimality

We still need to show is that our generic translation algorithm (see Algorithm 2 on the facing page) does indeed compute the alignment distance. Formally, we need to show that for any two sequences $\bar{x}$, $\bar{y}$:

$$A^*(1, 1) \overset{!}{=} D(\bar{x}, \bar{y}) \tag{2.43}$$

That is to say: We need to prove that our problem can be decomposed; that we do not need to construct all possible trees and then find the optimum but can construct an optimal tree for a part of the overall problem and merge it with other optimal subtrees to an overall optimal solution. Giegerich, Meyer, and

Steffen [12] have done extensive work on this topic and found that the decompositionality depends on the combination of algebra $\mathcal{F}$ and choice function h. In reference to the work of Bellman [3] they coined the term "Algebraic Version of Bellman's Principle of Optimality", which a certain tuple $(\mathcal{F}, h)$ must fulfill to be decomposable. In the "standard case" of $h = \min$, Giegerich, Meyer, and Steffen [12] found that algebras just need to adhere to a monotonicity constraint. As we have defined our algebras in such a way that every application of the algebra functions only adds some non-negative term in the interval $[0, 1]$ (see Section 2.2.2 on page 18) this condition is met by definition.

2.4.2 Computational Complexity

Finally, our definition of an ADP problem as stated in 2.2.6 on page 24 requires efficiency, that is: We need Algorithm 2 on page 30 to run in polynomial time with respect to the lengths M and N of the input sequences $\bar{x}$ and $\bar{y}$.

Theorem 2. *Algorithm 2 on page 30 has an asymptotic efficiency of*

$$\mathcal{O}(M \cdot N) \tag{2.44}$$

Proof. As the algorithm creates dynamic programming tables of size $(M + 1) \times (N + 1)$ and fills them iteratively using for-loops it remains to show that the calculation of each entry in these tables can be done in constant time with respect to M and N.

We assume that

- accessing entries of dynamic programming tables,
- calculating algebra functions d and
- calculating the choice function h

is possible in constant time with respect to M and N. Further we assume that the number of nonterminal symbols is (much) smaller than M and N and constant. The only critical operation left is the selection of applicable production rules. Assume that we have already sorted the nonterminal symbols in Φ according to their dependencies. Then we can pre-compute the applicable production rules for every entry $A(i, j)$ in $\mathcal{O}(M \cdot N)$ using Algorithm 3 on the next page. For specific grammars, more efficient calculation schemes can be applied. We discuss this issue in more depth in Section 3.3 on page 58.

With this pre-computation the applicable production rules can be retrieved in constant time as well, proving our claim.

$\square$

2.5 Distance Properties

In this section we investigate different properties of the alignment distance D. In particular we are interested in how far D fulfills the properties of a metric. As a reminder:

Algorithm 3 A generic dynamic programming algorithm that calculates the available production rules of a given grammar and input sequences $\bar{x}$ and $\bar{y}$ for every possible situation in Algorithm 2 on page 30.

```
for A $\in \Phi$ do
    Initialize $A$ as a table of size $(M+1) \times (N+1)$. Every entry is initialized with an empty set.
    if A is accepting then
        Add the production rule A = nil(<EMPTY,EMPTY>) to $A(M+1, N+1)$.
    end if
end for
for $i \in (M+1, \dots, 1)$ do
    for $j \in (N+1, \dots, 1)$ do
        for A $\in \Phi$ do
            Let $A$ be the dynamic programming table corresponding to A.
            for production rules of the form A = B in $\Delta$ do
                Let $B$ be the dynamic programming table corresponding to B.
                if $B(i, j) \neq \emptyset$ then
                    Add A = B; to $A(i, j)$.
                end if
            end for
            for production rules of the form A = t(<TL,TR>, B) in $\Delta$ do
                Let $B$ be the dynamic programming table corresponding to B.
                Let $(\mathcal{A}, \mathcal{A}')$ be the arity of function template $t$.
                if $M+1-i \geq \mathcal{A}$ and $N+1-j \geq \mathcal{A}'$ and $B(i+\mathcal{A}, j+\mathcal{A}') \neq \emptyset$ then
                    Add A = t(<TL,TR>, B) to $A(i, j)$.
                end if
            end for
        end for
    end for
end for
```

Definition 23. Let X be some arbitrary set and $\bar{x}, \bar{y}, \bar{z} \in X$. A metric is some function

$$\mathfrak{d} : X \times X \to \mathbb{R} \tag{2.45}$$

such that:

$$\mathfrak{d}(\bar{x}, \bar{y}) \geq 0 \quad \text{(non-negativity)} \tag{2.46}$$

$$\mathfrak{d}(\bar{x}, \bar{y}) = 0 \Leftrightarrow \bar{x} = \bar{y} \quad \text{(identity of indiscernibles)} \tag{2.47}$$

$$\mathfrak{d}(\bar{x}, \bar{y}) = \mathfrak{d}(\bar{y}, \bar{x}) \quad \text{(symmetry)} \tag{2.48}$$

$$\mathfrak{d}(\bar{x}, \bar{z}) \leq \mathfrak{d}(\bar{x}, \bar{y}) + \mathfrak{d}(\bar{y}, \bar{z}) \quad \text{(triangular inequality)} \tag{2.49}$$

We will investigate the conditions for the former three conditions in the following sections. Under these constraints one can embed the data in a pseudo-euclidian space [32], which is an important precondition for many learning techniques, such as Relational Generalized Learning Vector Quantization [29]. In addition we will provide some notes on scale-invariance in Section 2.5.4 on page 39 and on the interpretation of keyword weights as relevance terms in Section 2.5.5 on page 40.

We can not investigate the triangular inequality in depth due to the scope of this thesis, but rather provide pointers to existing work on the topic: The triangular inequality has been shown to hold for some alignment algorithms by Waterman, Smith, and Beyer [42]. Chen and Ng [8] notes that this does not hold for every alignment algorithm (most notably it does not hold for Dynamic Time Warping). In general the proof is easy for edit distances: If sequence $\bar{x}$ can be transformed to $\bar{y}$ and $\bar{y}$ to $\bar{z}$ using edit operations, then one can also transform $\bar{x}$ to $\bar{z}$ using the concatenation of the edit scripts. If the underlying algebra fulfills the triangular inequality then this holds for the edit distance as well (see e.g. Heun [18]). However, within the scope of this thesis we can not provide a comprehensive equivalence proof between alignments and edit scripts.

2.5.1 Non-Negativity

Theorem 3. *Let $\mathcal{F}$ be some algebra, h be some choice function and $\mathcal{G}$ some grammar. Then for all possible sequences $\bar{x}, \bar{y}$, all nonterminal symbols $\mathrm{A} \in \Phi$ and all $i \in \{1, \dots, M+1\}, j \in \{1, \dots, N+1\}$ it holds:*

$$A(i, j) \geq 0 \tag{2.50}$$

Proof. Consider Algorithm 2 on page 30. For the entries of A we find:

$$A(i, j) = h[\theta_l]_{l=1\dots L} \tag{2.51}$$

This already implies non-negativity as h has a non-negative result per definition (see Section 2.2.3 on page 19). Further we also know that no term θ_l can be negative as per definition all algebra functions add a non-negative term to the input argument (see Section 2.2.2 on page 18). Via induction it follows that no negative term occurs ever in the dynamic programming scheme. □

Note that this directly implies the non-negativity of D, if Bellman's Principle of Optimality holds for the choice function.

2.5.2 Identity of Indiscernibles

Theorem 4. *Let $\mathcal{G}$ be a grammar over the signature $\mathcal{T}^*$ such that trivial replacement alignments are part of its language. Let $\mathcal{F}$ be an algebra such that for all $\kappa \in K$ and for all $a, b \in \Sigma_\kappa$:*

$$a = b \Rightarrow c^{\kappa}_{\text{rep}}(a, b) = 0 \tag{2.52}$$

Using min *as choice function we obtain for all possible sequences $\bar{x}, \bar{y}$:*

$$\bar{x} = \bar{y} \Rightarrow D(\bar{x}, \bar{y}) = 0 \tag{2.53}$$

Proof. Let $\bar{x} = \bar{y}$. Consider the trivial replacement alignment T^*, which has the yield $\mathcal{Y}(T^*) = (\bar{x}, \bar{y})$. Then we obtain:

$$\mathcal{F}(T^*) = \sum_{i=1}^{M} \sum_{\kappa \in K} c^{\kappa}_{\text{rep}}(x^{\kappa}_i, y^{\kappa}_i) + 0 = 0 \tag{2.54}$$

Thus we can conclude using Equation 2.29 on page 24:

$$D(\bar{x}, \bar{y}) := h[\mathcal{F}(T) | T \in \mathcal{L}(\mathcal{G}), \mathcal{Y}(T) = (\bar{x}, \bar{y})] \leq 0 \tag{2.55}$$

As the alignment distance is non-negative according to Theorem 3 on the preceding page, we obtain:

$$D(\bar{x}, \bar{y}) = 0 \tag{2.56}$$

$\square$

Theorem 5. *We use the definitions from Theorem 4, except for the algebra. Instead let $\mathcal{F}$ be an algebra such that for all $\kappa \in K$ and for all $a, b \in \Sigma_\kappa$:*

$$a \neq b \Rightarrow c^{\kappa}_{\text{rep}}(a, b) > 0 \tag{2.57}$$

$$c^{\kappa}_{\text{del}}(a) > 0 \tag{2.58}$$

$$c^{\kappa}_{\text{skip\_del}}(a) > 0 \tag{2.59}$$

$$c^{\kappa}_{\text{ins}}(b) > 0 \tag{2.60}$$

$$c^{\kappa}_{\text{skip\_ins}}(b) > 0 \tag{2.61}$$

Using min *as choice function we obtain for all possible sequences $\bar{x}, \bar{y}$:*

$$\bar{x} \neq \bar{y} \Rightarrow D(\bar{x}, \bar{y}) > 0 \tag{2.62}$$

Proof. Consider the definition of D in Equation 2.29 on page 24. As D is non-negative we can conclude that the only way to construct an alignment T, such that

$$\mathcal{F}(T) \leq 0 \Rightarrow D(\bar{x}, \bar{y}) \leq 0 \tag{2.63}$$

is a tree with $\mathcal{F}(T) = 0$. As our $\mathcal{F}$ implies that all operations other than rep and nil have costs larger than 0 we can infer that T only contains rep and nil operations. Due to Theorem 1 on page 23 we know that T has to be a trivial replacement alignment. Finally our restrictions to $\mathcal{F}$ imply:

$$\text{rep}(a, b) = 0 \Rightarrow a = b \tag{2.64}$$

Therefore, we can infer $\bar{x} = \bar{y}$, which in turn implies:

$$D(\bar{x}, \bar{y}) = 0 \Rightarrow \bar{x} = \bar{y} \tag{2.65}$$

As we know that D is non-negative due to Theorem 3 on page 34 we can conclude:

$$\bar{x} \neq \bar{y} \Rightarrow D(\bar{x}, \bar{y}) \neq 0 \Rightarrow D(\bar{x}, \bar{y}) > 0 \tag{2.66}$$

□

We observe that all introduced grammars allow for trivial replacement alignments, such that these theorems can be applied. Further we note that the combination of both theorems implies the identity of indiscernibles.

2.5.3 Symmetry

We investigate the symmetry of the alignment distance by first introducing the concepts of symmetric algebras and grammars and then proving that combining a symmetric algebra with a symmetric grammar leads to a symmetric alignment distance.

Definition 24. An algebra $\mathcal{F}$ over the signature $\mathcal{T}^*$ is called *symmetric* if and only if for all nodes $x, y \in \Sigma_\times$ and all $s \in \mathbb{R}^+$:

$$d_{\text{rep}}(x, s, y) = d_{\text{rep}}(y, s, x) \tag{2.67}$$

$$d_{\text{del}}(x, s) = d_{\text{ins}}(s, x) \tag{2.68}$$

Lemma 6. *An algebra $\mathcal{F}$ over the signature $\mathcal{T}^*$ is symmetric if for all keywords $\kappa \in K$ and all values $a, b \in \Sigma_\kappa$:*

$$c^\kappa_{\text{rep}}(a, b) = c^\kappa_{\text{rep}}(b, a) \tag{2.69}$$

$$c^\kappa_{\text{del}}(a) = c^\kappa_{\text{ins}}(a) \tag{2.70}$$

$$\tag{2.71}$$

Proof. The proof follows from the definition of an algebra function in Equation 2.10 on page 19:

$$\begin{aligned} d_{\text{rep}}(x, s, y) &= s + \sum_{\kappa \in K} g_\kappa c^\kappa_{\text{rep}}(x^\kappa, y^\kappa) && (2.72) \\ &= s + \sum_{\kappa \in K} g_\kappa c^\kappa_{\text{rep}}(y^\kappa, x^\kappa) && (2.73) \\ &= d_{\text{rep}}(y, s, x) && (2.74) \end{aligned}$$

and

$$d_{\text{del}}(x,s) = s + \sum_{\kappa \in K} g_\kappa c^\kappa_{\text{del}}(x^\kappa) \tag{2.75}$$
$$= s + \sum_{\kappa \in K} g_\kappa c^\kappa_{\text{ins}}(x^\kappa) \tag{2.76}$$
$$= d_{\text{ins}}(s,x) \tag{2.77}$$

□

Definition 25. A grammar $\mathcal{G}$ over the signature $\mathcal{T}^*$ is called *strongly symmetric* if for each production rule in Δ of the form

```
A = del(<NODE,EMPTY>, B);
```

there exists a production rule

```
A = ins(<EMPTY,NODE>, B);
```

Also for each rule of the form

```
A = skip_del(<NODE,EMPTY>, B);
```

there has to be a production rule

```
A = skip_ins(<EMPTY,NODE>, B);
```

This also has to be the case in the other direction (for each insertion rule there is a corresponding deletion rule and for each skip-insertion rule there is a corresponding skip-deletion rule).

We observe that $\mathcal{G}_{\text{Glob}}$ is obviously strongly symmetric, while all subsequently introduced grammars are not strongly symmetric.

Theorem 7. *A strongly symmetric grammar $\mathcal{G}$ in combination with a symmetric algebra $\mathcal{F}$ over the signature $\mathcal{T}^*$ implies symmetric dynamic programming tables independent of the choice function h. In other words: Given two sequences $\bar{x}, \bar{y}$ we can conclude for all nonterminal symbols $\mathrm{A} \in \Phi$ and all $i \in \{1, \ldots, M+1\}, j \in \{1, \ldots, N+1\}$:*

$$A_{\bar{x},\bar{y}}(i,j) = A_{\bar{y},\bar{x}}(j,i) \tag{2.78}$$

where $A_{\bar{x},\bar{y}}$ is the dynamic programming table corresponding to the nonterminal symbol A and given the input $(\bar{x}, \bar{y})$.

Proof. See Appendix A.1.1 on page 99. □

Note that we can directly infer a symmetric alignment distance D if the choice function adheres to Bellman's Principle of Optimality, because:

$$D(\bar{x},\bar{y}) = A^*_{\bar{x},\bar{y}}(1,1) = A^*_{\bar{y},\bar{x}}(1,1) = D(\bar{y},\bar{x}) \tag{2.79}$$

Strong grammar symmetry is usually undesirable because it unnecessarily expands the search space (see Section 2.3 on page 25). However, we can define a weaker form of grammar symmetry, which is much easier to achieve and implies alignment distance symmetry as well:

Definition 26. A grammar $\mathcal{G}$ over the signature $\mathcal{T}^*$ is called *weakly symmetric* if for all sequences and all trees $T \in \mathcal{L}(\mathcal{G})$ with yield $\mathcal{Y}(T) = (\bar{x}, \bar{y})$ there exists at least one alignment $T' \in \mathcal{L}(\mathcal{G})$ with yield $\mathcal{Y}(T) = (\bar{y}, \bar{x})$ such that:

1. If T contains the operation $\mathrm{del}(x_i)$, T' contains $\mathrm{ins}(x_i)$.
2. If T contains the operation $\mathrm{skip\_del}(x_i)$, T' contains $\mathrm{skip\_ins}(x_i)$.
3. If T contains the operation $\mathrm{ins}(y_j)$, T' contains $\mathrm{del}(y_j)$.
4. If T contains the operation $\mathrm{skip\_ins}(y_j)$, T' contains $\mathrm{skip\_del}(y_j)$.
5. If T contains the operation $\mathrm{rep}(x_i, y_j)$, T' contains $\mathrm{rep}(y_j, x_i)$.

Note that both alignments necessarily contain the operation nil exactly once, as they would not be finite otherwise (see also Theorem 1 on page 23).

We observe that strong grammar symmetry implies weak grammar symmetry. We do not proof weak symmetry for every grammar introduced in this thesis. Rather, we do it exemplarily for $\mathcal{G}_{\text{Affine}}$. The technique employed in that proof can be transferred to the other grammars as well.

Theorem 8. *$\mathcal{G}_{\text{Affine}}$ is weakly symmetric.*

Proof. See Appendix A.1.2 on page 101. □

Theorem 9. *A weakly symmetric grammar $\mathcal{G}$ in combination with a symmetric algebra $\mathcal{F}$ over the signature $\mathcal{T}^*$ and* min *as choice function implies a symmetric alignment distance, that means for all possible sequences $\bar{x}, \bar{y}$:*

$$D(\bar{x}, \bar{y}) = D(\bar{y}, \bar{x}) \tag{2.80}$$

Proof. Consider the definition of D in Equation 2.29 on page 24. Now assume

$$D(\bar{x}, \bar{y}) \neq D(\bar{y}, \bar{x}) \tag{2.81}$$

Then one of these statements has to be true:

$$D(\bar{x}, \bar{y}) < D(\bar{y}, \bar{x}) \tag{2.82}$$
$$D(\bar{x}, \bar{y}) > D(\bar{y}, \bar{x}) \tag{2.83}$$

Without loss of generality we assume the former case. Let T be an alignment for which

$$T \in \mathcal{L}(\mathcal{G}) \wedge \mathcal{Y}(T) = (\bar{x}, \bar{y}) \wedge \mathcal{F}(T) = D(\bar{x}, \bar{y}) \tag{2.84}$$

Because $\mathcal{G}$ is weakly symmetric there exists a corresponding alignment $T' \in \mathcal{L}(\mathcal{G})$ with yield $\mathcal{Y}(T') = (\bar{y}, \bar{x})$ that fulfills the conditions listed in Definition 26. Due to algebra symmetry it follows that $\mathcal{F}(T) = \mathcal{F}(T')$, which in turn implies that:

$$D(\bar{y}, \bar{x}) \leq \mathcal{F}(T') = \mathcal{F}(T) = D(\bar{x}, \bar{y}) < D(\bar{y}, \bar{x}) \tag{2.85}$$

This is a contradiction, thus proving our claim. □

2.5.4 Scale-Invariance

Theorem 10. *Let $\mathcal{F}$ be some algebra and $\mathcal{G}$ some grammar, both over the signature $\mathcal{T}^*$. Let further $\alpha \in \mathbb{R}$ and let h be a choice function such that*

$$h[\alpha \cdot \theta_l]_{l=1\ldots L} = \alpha \cdot h[\theta_l]_{l=1\ldots L} \tag{2.86}$$

Finally we define an algebra $\mathcal{F}'$ as follows:

$$\forall \kappa \in K : c_t^{\kappa\prime}(\circ, \circ) := \alpha \cdot c_t^{\kappa}(\circ, \circ) \tag{2.87}$$

Given two sequences $\bar{x}, \bar{y}$ we can conclude for all nonterminal symbols $\mathrm{A} \in \Phi$ and all $i \in \{1, \ldots, M+1\}, j \in \{1, \ldots, N+1\}$:

$$A'(i,j) = \alpha \cdot A(i,j) \tag{2.88}$$

where A is the dynamic programming table corresponding to the nonterminal symbol A given the input $(\bar{x}, \bar{y})$ and using the algebra $\mathcal{F}$, while A' is the dynamic programming table corresponging to the same nonterminal symbol and given the same input, but using the algebra $\mathcal{F}'$.

In other words: If we scale the algebra, all dynamic programming table entries scale accordingly.

Proof. We can proof the theorem by structural induction. Consider Algorithm 2 on page 30 again. The initial cells in the dynamic programming tables have the form

$$A(M+1, N+1) = 0 = \alpha \cdot 0 \tag{2.89}$$

which is the grounding for our induction.

Now let A be a nonterminal symbol from Φ and $i \in \{1, \ldots, M+1\}, j \in \{1, \ldots, N+1\}$. Our induction hypothesis is that our claim holds for all entries of the dynamic programming tables, which we need to calculate $A(i,j)$, according to Algorithm 2. Now we need to show that the calculation of $A(i,j)$ is scale-invariant.

Consider algebra functions:

$$\begin{aligned} d_t'(\circ, \alpha \cdot s, \circ) &= \alpha \cdot s + \sum_{\kappa \in K} g_\kappa \cdot c_t^{\kappa\prime}(\circ, \circ) && (2.90) \\ &= \alpha \cdot s + \sum_{\kappa \in K} g_\kappa \cdot \alpha \cdot c_t^{\kappa}(\circ, \circ) && (2.91) \\ &= \alpha \cdot d_t(\circ, s, \circ) && (2.92) \end{aligned}$$

Consider the choice function:

$$\begin{aligned} A(i,j) &= h[\alpha \cdot \theta_l]_{l=1\ldots L} && (2.93) \\ &= \alpha \cdot h[\theta_l]_{l=1\ldots L} && (2.94) \end{aligned}$$

Thus we have shown that

$$A'(i,j) = \alpha \cdot A(i,j) \tag{2.95}$$

which proofs our claim by structural induction. □

Note that min obviously fulfills the condition imposed by Theorem 10, such that scale-invariance holds for each grammar and algebra that is combined with min. Also note that this implies a scale-invariant alignment distance if the choice function adheres to Bellman's Principle of Optimality, because:

$$D'(\bar{x}, \bar{y}) = A^{*\prime}(1,1) = \alpha \cdot A^*(1,1) = \alpha \cdot D(\bar{x}, \bar{y}) \tag{2.96}$$

The scale-invariance post-hoc legitimizes our restrictive definition of comparator functions to the output interval $[0,1]$. As the alignment distance is scale-invariant one can just scale any desired comparator function output down to that interval without changing the alignment behavior.

2.5.5 Relevance Interpretation

It should be noted that, albeit comparator functions are restricted to the interval $[0,1]$, their values do not have to be distributed equally across that range. In fact it might very well be the case that for some keywords the comparator function values are always more prominent than for others. This can be counteracted by adjusting the keyword weights, e.g. by the inverse average output value of the respective comparator functions. However, such adjustments make it harder to interpret keyword weights in terms of relevance: If a keyword weight is smaller that does not imply a small relevance. In fact the opposite might be the case if the output of the respective comparator function tends to be very large. In our third experiment in Section 4.3 on page 77 we discuss a normalization scheme to still retrieve a meaningful interpretation of keyword weights.

2.6 Cost Function

We optimize the parameters of sequence alignment algorithms by gradient descent on the Large Margin Nearest Neighbor (LMNN) cost function, suggested by Weinberger and Saul [43]. We can intuitively describe this cost function as follows: Try to pull data points closer to their neighbors in the same class and push them away from neighbors from another class.

As with the k-Nearest Neighbor (k-NN) approach [9] we need to set the *number of neighbors* we want to consider. Based on that k we can formally define what we mean by neighbors from the same class and neighbors from another class.

Definition 27. Let X be our data set and $\mathfrak{d}$ be some metric on X. Let $C \subset X$ be the class of $\bar{x}$, that is: $\bar{x} \in C$. The *k target neighbors* $\mathfrak{N}$ of $\bar{x}$ are defined as

$$\mathfrak{N}_0(\bar{x}) := \emptyset \tag{2.97}$$

$$\mathfrak{N}_k(\bar{x}) := \mathfrak{N}_{k-1}(\bar{x}) \cup \{ \operatorname*{argmin}_{\bar{y} \in C \setminus (\mathfrak{N}_{k-1}(\bar{x}) \cup \{\bar{x}\})} \mathfrak{d}(\bar{x}, \bar{y}) \} \tag{2.98}$$

In other words: $\mathfrak{N}_1(\bar{x})$ contains the closest other data point in the same class, $\mathfrak{N}_2(\bar{x})$ the closest and the second-closest and so on. Algorithmically one can calculate $\mathfrak{N}_k(\bar{x})$ more efficiently by making use of a sorted list, which allows for insertion in $\mathcal{O}(\log(k))$ time if k is the length of the list. We provide pseudocode for this in Algorithm 4. As the maximum element of a sorted list can be retrieved in $\mathcal{O}(1)$ this algorithm has an overall asymptotic efficiency of $\mathcal{O}(|X| \cdot \log(k))$.

Algorithm 4 An algorithm to calculate $\mathfrak{N}_k(\bar{x})$ given some $k \in \mathbb{N}, k > 0$, $\bar{x} \in X$ and some metric $\mathfrak{d}$ on X.

```
Let $\bar{x} \in C$ with $C \subset X$.
Initialize $\mathfrak{N}$ as an empty ascendingly sorted list over tuples from $\mathbb{R}^+ \times X$.
**for** $\bar{y} \in C \setminus \{\bar{x}\}$ **do**
    Initialize $d \leftarrow \mathfrak{d}(\bar{x}, \bar{y})$
    **if** $|\mathfrak{N}| < k$ **then**
        Insert $(d, \bar{y})$ into $\mathfrak{N}$.
    **else if** $d < \max_{(d', \bar{z}) \in \mathfrak{N}} d'$ **then**
        Remove the last element from $\mathfrak{N}$.
        Insert $(d, \bar{y})$ into $\mathfrak{N}$.
    **end if**
**end for**
**return** $\mathfrak{N}$
```

Definition 28. Let X be our data set and $\mathfrak{d}$ be some metric on X. Let $\bar{x}$ and $\bar{y}$ be data points from the same class, that is: $\bar{x}, \bar{y} \in C$ and $C \subset X$. Let further γ be a positive real number which we call the *margin*. Then we define the *imposters* of $\bar{x}$ with respect to $\bar{y}$ as follows:

$$\mathfrak{I}_\gamma(\bar{x}, \bar{y}) := \{\bar{z} | \bar{z} \in X \setminus C, \mathfrak{d}(\bar{x}, \bar{z}) < \mathfrak{d}(\bar{x}, \bar{y}) + \gamma\} \tag{2.99}$$

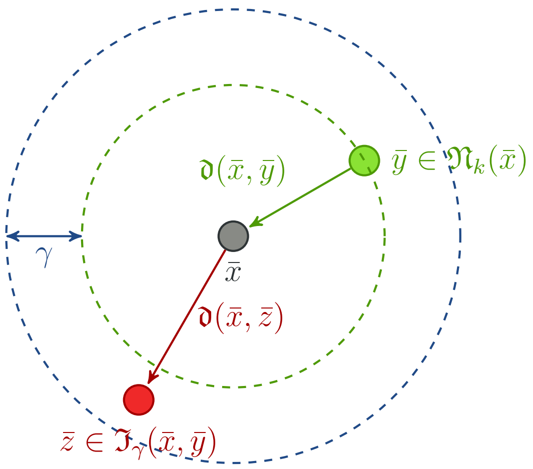
In other words: $\mathfrak{I}_\gamma(\bar{x}, \bar{y})$ contains all data points from another class that are closer to $\bar{x}$ than the neighbor from the same class $\bar{y}$, including a margin of safety. To calculate the imposters one just needs to iterate over all $\bar{z} \in X \setminus C$ and check the distance criterion, which is done in $\mathcal{O}(|X|)$.

Definition 29. Let X be our data set and $\mathfrak{d}$ be some metric on X. Let further γ be our margin. The *LMNN cost function* is defined as:

$$\mathcal{E} := \sum_{\bar{x}} \left(\sum_{\bar{y} \in \mathfrak{N}_k(\bar{x})} (\mathfrak{d}(\bar{x}, \bar{y}))^2 + \sum_{\bar{z} \in \mathfrak{I}_\gamma(\bar{x}, \bar{y})} \gamma^2 + (\mathfrak{d}(\bar{x}, \bar{y}))^2 - (\mathfrak{d}(\bar{x}, \bar{z}))^2 \right) \tag{2.100}$$

Note that in the original definition, Weinberger and Saul [43] define the target neighbors and imposters implicitly using the class label and the hinge loss. As being part of the imposters is equivalent to the hinge loss being bigger than (or equal to) zero, our formulation is equivalent to the original one. We just find a set-based definition to be better understandable.

The LMNN cost function punishes distances between data points and their target neighbors (pull force) and punishes the distance of imposters to the



$\mathfrak{d}(\bar{x}, \bar{y})$
$\bar{y} \in \mathfrak{N}_k(\bar{x})$
γ
$\bar{x}$
$\mathfrak{d}(\bar{x}, \bar{z})$
$\bar{z} \in \mathfrak{I}_\gamma(\bar{x}, \bar{y})$

Figure 2.3: An illustration of the LMNN cost function: The data point $\bar{x}$ is shown in the middle in gray. One of its target neighbors $\bar{y}$ is shown in green. The distance between $\bar{x}$ and $\bar{y}$ is also drawn as green, dashed circle around $\bar{x}$. By adding the margin γ to the radius one gets a bigger circle, shown in blue. All points of another class inside this larger circle are the imposters $\bar{z}$ of $\bar{x}$ with respect to $\bar{y}$. The idea of the LMNN cost function is to pull $\bar{y}$ closer to $\bar{x}$ (that is: make the inner circle smaller) and to push away $\bar{z}$ from $\bar{x}$ (that is: push $\bar{z}$ outside of the outer circle).

margin (seen from the inside; push force). The cost function is also illustrated in Figure 2.3. As Weinberger and Saul [43] explain, this approach has at least two advantages:

1. It directly optimizes the k-NN classification accuracy, as a k-NN classifier will choose the correct classification if the majority of the k-Nearest Neighbors are in the correct class.

2. It optimizes local distances between neighbors instead of global distances between all data points of the same class. This makes the LMNN cost function more robust against non-convex class borders or very cluttered data sets.

2.7 Alignment Gradient

As we intend to learn the parameters of an alignment distance, we require the derivative of the LMNN cost function with respect to some parameter λ of the underlying metric $\mathfrak{d}$:

$$\begin{aligned}\frac{\partial}{\partial \lambda}\mathcal{E} = 2 \cdot \sum_{\bar{x}} \Bigg(& \sum_{\bar{y} \in \mathfrak{N}_k(\bar{x})} \mathfrak{d}(\bar{x}, \bar{y}) \cdot \left(\frac{\partial}{\partial \lambda} \mathfrak{d}(\bar{x}, \bar{y}) \right) + \\ & \sum_{\bar{z} \in \mathfrak{I}_\gamma(\bar{x}, \bar{y})} \mathfrak{d}(\bar{x}, \bar{y}) \cdot \left(\frac{\partial}{\partial \lambda} \mathfrak{d}(\bar{x}, \bar{y}) \right) - \mathfrak{d}(\bar{x}, \bar{z}) \cdot \left(\frac{\partial}{\partial \lambda} \mathfrak{d}(\bar{x}, \bar{z}) \right) \Bigg)\end{aligned} \tag{2.101}$$

Note that this requires the assumption that $\mathfrak{N}_k(\bar{x})$ and $\mathfrak{I}_\gamma(\bar{x}, \bar{y})$ are static, at least for one gradient step. As they depend on $\mathfrak{d}$ and we change $\mathfrak{d}$ using gradient descent this is not necessarily the case. With respect to this problem, Weinberger and Saul [43] note that convergence can still be achieved if the step-size is sufficiently small.

We provide pseudocode for the gradient calculation in Algorithm 5. Note that the same algorithm (with small changes) can be applied to calculate the error itself as well. We obtain an asymptotic efficiency of:

$$\mathcal{O}\Big(|X| \cdot \big(|X| \cdot \log(k) + k \cdot (|X| + |X|)\big)\Big) = \mathcal{O}(|X|^2 \cdot k) \tag{2.102}$$

Algorithm 5 An algorithm to calculate the gradient $\frac{\partial}{\partial \lambda}\mathcal{E}$ given some $k \in \mathbb{N}, k > 0$, the data set X, some metric $\mathfrak{d}$ on X and a partition of X into disjoint sets (classes) C.

```
Initialize $G \leftarrow 0$.
for $\bar{x} \in X$ do
    Let $C$ be the subset for which $\bar{x} \in C$.
    Calculate $\mathfrak{N}_k(\bar{x})$.
    for $\bar{y} \in \mathfrak{N}_k(\bar{x})$ do
        Calculate $\frac{\partial}{\partial \lambda}\mathfrak{d}(\bar{x}, \bar{y})$.
        $G \leftarrow G + \mathfrak{d}(\bar{x}, \bar{y}) \cdot \frac{\partial}{\partial \lambda}\mathfrak{d}(\bar{x}, \bar{y})$.
        Calculate $\mathfrak{I}_\gamma(\bar{x}, \bar{y})$.
        for $\bar{z} \in \mathfrak{I}_\gamma(\bar{x}, \bar{y})$ do
            Calculate $\frac{\partial}{\partial \lambda}\mathfrak{d}(\bar{x}, \bar{z})$.
            $G \leftarrow G + \mathfrak{d}(\bar{x}, \bar{y}) \cdot \frac{\partial}{\partial \lambda}\mathfrak{d}(\bar{x}, \bar{y})$.
            $G \leftarrow G - \mathfrak{d}(\bar{x}, \bar{z}) \cdot \frac{\partial}{\partial \lambda}\mathfrak{d}(\bar{x}, \bar{z})$.
        end for
    end for
end for
return $G$.
```

Apparently, the gradient of the LMNN cost function depends on the gradient of the metric. If we want to use the alignment distance D as the underlying (pseudo-)metric we have to calculate the gradient of the alignment distance:

Theorem 11. *Let $\mathcal{G}$ be some grammar and $\mathcal{F}$ be an algebra. Further, let λ be some parameter of the comparator functions c^{κ^*} for keyword κ^* of $\mathcal{F}$. Finally, let h be a choice function for which Bellman's Principle of Optimality holds.*

We define the alternative algebra $\mathcal{F}'$ as

$$c_t^{\kappa\prime}(\circ, \circ) := \begin{cases} \frac{\partial}{\partial \lambda} c_t^\kappa(\circ, \circ) & \text{, if } \kappa = \kappa^* \\ 0 & \text{otherwise} \end{cases} \tag{2.103}$$

and the alternative choice function h' as

$$h'[\theta_1, \ldots, \theta_L] := \sum_{l=1}^{L} \left(\frac{\partial}{\partial \theta_l} h[\theta_1, \ldots, \theta_L] \right) \cdot \theta_l \tag{2.104}$$

Then the result of Algorithm 2 on page 30 on the same input and grammar, but with the alternative algebra and choice function is the derivative of D with respect to λ. That is to say:

$$\frac{\partial}{\partial\lambda} D(\bar{x}, \bar{y}) = A^*(1,1) \tag{2.105}$$

if A^* is the dynamic programming table corresponding to the axiom of $\mathcal{G}$ for this calculation.

Proof. Consider the derivative of dynamic programming table entries:

$$\frac{\partial}{\partial\lambda} A(i,j) = \frac{\partial}{\partial\lambda} h[\theta_1, \dots, \theta_L] \tag{2.106}$$

$$= \sum_{l=1}^{L} \left(\frac{\partial}{\partial\theta_l} h[\theta_1, \dots, \theta_L] \right) \cdot \left(\frac{\partial}{\partial\lambda} \theta_l \right) \tag{2.107}$$

$$= h' \left[\frac{\partial}{\partial\lambda} \theta_1, \dots, \frac{\partial}{\partial\lambda} \theta_L \right] \tag{2.108}$$

Now consider the inner derivative. Let t be the referenced function template and B be the referenced nonterminal in the respective production rule. Further let B be the dynamic programming table corresponding to B. Then we obtain:

$$\frac{\partial}{\partial\lambda} \theta_l = \frac{\partial}{\partial\lambda} d_t(\circ, B(\circ,\circ), \circ) \tag{2.109}$$

$$= \frac{\partial}{\partial\lambda} B(\circ,\circ) + \frac{\partial}{\partial\lambda} \sum_{\kappa \in K} g_\kappa \cdot c_t^\kappa(\circ,\circ) \tag{2.110}$$

$$= \frac{\partial}{\partial\lambda} B(\circ,\circ) + \sum_{\kappa \in K} g_\kappa \cdot \left(\frac{\partial}{\partial\lambda} c_t^\kappa(\circ,\circ) \right) \tag{2.111}$$

$$= \frac{\partial}{\partial\lambda} B(\circ,\circ) + g_{\kappa^*} \cdot \left(\frac{\partial}{\partial\lambda} c_t^{\kappa^*}(\circ,\circ) \right) \tag{2.112}$$

$$= \frac{\partial}{\partial\lambda} B(\circ,\circ) + \sum_{\kappa \in K} g_\kappa \cdot c_t^{\kappa\prime}(\circ,\circ) \tag{2.113}$$

$$\tag{2.114}$$

□

Theorem 12. *Let $\mathcal{G}$ be some grammar and $\mathcal{F}$ be an algebra. Let g_{κ^*} be the keyword weight for keyword κ^*. Finally let h be a choice function for which Bellman's Principle of Optimality holds.*

We define the alternative algebra $\mathcal{F}'$ with

$$c_t^{\kappa\prime}(\circ,\circ) := \begin{cases} \frac{1}{g_{\kappa^*}} \cdot c_t^{\kappa^*}(\circ,\circ) & , \textit{if } \kappa = \kappa^* \\ 0 & \textit{otherwise} \end{cases} \tag{2.115}$$

and the alternative choice function h' as above. Then the result of Algorithm 2 on page 30 on the same input and grammar, but with the alternative algebra *and choice function is the derivative of D with respect to g_{κ^*}. That is to say:*

$$\frac{\partial}{\partial g_{\kappa^*}} D(\bar{x}, \bar{y}) = A^*(1,1) \tag{2.116}$$

if A^* is the dynamic programming table corresponding to the axiom of $\mathcal{G}$ for this calculation.

Proof. Consider the derivative of dynamic programming table entries:

$$\begin{aligned}
\frac{\partial}{\partial g_{\kappa^*}} A(i,j) &= \frac{\partial}{\partial g_{\kappa^*}} h[\theta_1, \dots, \theta_L] && (2.117)\\
&= \sum_{l=1}^{L} \left(\frac{\partial}{\partial \theta_l} h[\theta_1, \dots, \theta_L]\right) \cdot \left(\frac{\partial}{\partial g_{\kappa^*}} \theta_l\right) && (2.118)\\
&= h' \left[\frac{\partial}{\partial g_{\kappa^*}} \theta_1, \dots, \frac{\partial}{\partial g_{\kappa^*}} \theta_L\right] && (2.119)
\end{aligned}$$

Now consider the inner derivative. Let t be the referenced function template and B be the referenced nonterminal in the respective production rule. Further let B be the dynamic programming table corresponding to B. Then we obtain:

$$\begin{aligned}
\frac{\partial}{\partial g_{\kappa^*}} \theta_l &= \frac{\partial}{\partial g_{\kappa^*}} d_t(\circ, B(\circ, \circ), \circ) && (2.120)\\
&= \frac{\partial}{\partial g_{\kappa^*}} B(\circ, \circ) + \frac{\partial}{\partial g_{\kappa^*}} \sum_{\kappa \in K} g_\kappa \cdot c_t^\kappa(\circ, \circ) && (2.121)\\
&= \frac{\partial}{\partial g_{\kappa^*}} B(\circ, \circ) + c_t^{\kappa^*}(\circ, \circ) && (2.122)\\
&= \frac{\partial}{\partial g_{\kappa^*}} B(\circ, \circ) + \sum_{\kappa \in K} g_\kappa \cdot c_t^{\kappa\prime}(\circ, \circ) && (2.123)
\end{aligned}$$

□

Thus we can directly apply Algorithm 2 on page 30 to calculate the derivative. It should be noted, though, that for comparator functions with many parameters, it can be more efficient not to calculate the gradient for each parameter separately, but rather in parallel. This particular topic, as well as approximations for faster gradient calculation, are discussed in more detail by Mokbel et al. [29]. Similarly it makes more sense to calculate the gradient with respect to all keyword weights at once, instead of applying the calculation scheme for each keyword weight separately.

Disregarding these optimizations, we obtain an overall asymptotic efficiency for one gradient step of

$$\mathcal{O}(|X|^2 \cdot k \cdot M \cdot N) \tag{2.124}$$

where M and N are the lengths of the respective sequences.

2.8 Softmin

To solve the problem of a non-differentiable choice function, we introduce an exponential-based approximation of the strict minimum, namely the soft minimum. Such soft approximations of the strict minimum/maximum are well-established in the literature. They are the inspiration for Global Alignment Kernels, as introduced by Cuturi et al. [10], but are also used in many other domains of machine learning: The book by Bishop [6], for example, lists 6 index entries for the soft maximum. Interestingly, to our knowledge the literature does not contain rigorous investigations regarding the approximation quality and some other properties of the soft minimum, which is why we provide some more details in the course of this section.

Definition 30. Let $\theta_1, \dots, \theta_L$ be real numbers. The *soft minimum* is defined as:

$$\text{softmin}[\theta_1, \dots, \theta_L] := \frac{1}{Z} \sum_{l=1}^{L} e_l \cdot \theta_l \tag{2.125}$$

where

$$e_l := \exp(-\beta \cdot \theta_l) \tag{2.126}$$

$$Z := \sum_{l=1}^{L} e_l \tag{2.127}$$

β is called *crispness*-Parameter and regulates the approximation quality, as shown in Theorem 15 on the facing page.

2.8.1 Derivative

Lemma 13. *Let $\theta_1, \dots, \theta_L$ be real numbers. Then the soft minimum derivative with respect to some* parameter λ *is given as:*

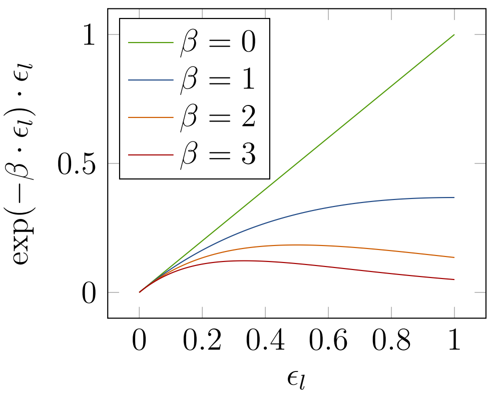
$$\begin{aligned}&\frac{\partial}{\partial\lambda}\text{softmin}[\theta_1, \dots, \theta_L] \\ &\quad = \sum_{l=1}^{L} \frac{e_l}{Z} \cdot \left(\frac{\partial}{\partial\lambda}\theta_l\right) \cdot \Big(1 - \beta \cdot (\theta_l - \text{softmin}[\theta_1, \dots, \theta_L])\Big)\end{aligned} \tag{2.128}$$

Proof. See Appendix A.2.1 on page 107. □

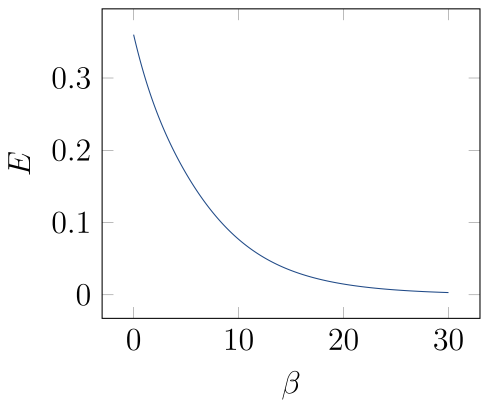
2.8.2 Approximation Error

Definition 31. Let $\theta_1, \dots, \theta_L$ be real numbers. We define the *soft minimum approximation error* E as follows:

$$E := |\text{softmin}[\theta_1, \dots, \theta_L] - \min[\theta_1, \dots, \theta_L]| \tag{2.129}$$



(a) The error contribution of a single non-minimum term in the soft minimum approximation plotted against its deviation ϵ_l for various β.



(b) The soft minimum approximation error E for $\beta \in [0, 10]$ and 10 random numbers as input drawn from a uniform distribution in the interval $[0, 1]$.

Figure 2.4: The behavior of the soft minimum approximation error.

Theorem 14. *Let $\theta_1, \dots, \theta_L$ be real numbers. We further define:*

$$\theta_{\min} := \min[\theta_1, \dots, \theta_L] \tag{2.130}$$

$$\Theta_{\min} := \{l \,|\, \theta_l = \theta_{\min}\} \tag{2.131}$$

$$\epsilon_l := \theta_l - \theta_{\min} \tag{2.132}$$

Then the soft minimum approximation error is given as:

$$E = \text{softmin}[\epsilon_1, \dots, \epsilon_L] = \frac{\sum_{l \notin \Theta_{\min}} \exp(-\beta \cdot \epsilon_l) \cdot \epsilon_l}{|\Theta_{\min}| + \sum_{l \notin \Theta_{\min}} \exp(-\beta \cdot \epsilon_l)} \tag{2.133}$$

Proof. See Appendix A.2.2 on page 108 □

From that we can draw a direct conclusion regarding the convergence of the soft minimum approximation error with respect to β:

Theorem 15. *For a fixed input $[\theta_1, \dots, \theta_L]$ the soft minimum approximation error E is limited by:*

$$E < \sum_{l \notin \Theta_{\min}} \exp(-\beta \cdot \epsilon_l) \cdot \epsilon_l \tag{2.134}$$

Proof. Given that $\Theta_{\min}$ has by definition at least one element we can conclude that

$$|\Theta_{\min}| + \sum_{l \notin \Theta_{\min}} \exp(-\beta \cdot \epsilon_l) > 1 \tag{2.135}$$

Thus the limit follows directly from Theorem 14. □

The soft minimum behavior is quite satisfying: Large deviations ϵ_l have small influence on the result because they get suppressed by the exponential term $\exp(-\beta \cdot \theta_l)$ (see Figure 2.4a). Smaller deviations are not suppressed as much but can not push the result into the wrong direction too much either. For practical purposes even moderate values of $\beta \approx 10$ should be enough to suppress errors sufficiently (see Figure 2.4b on the preceding page).

We can also derive an upper bound for the approximation error with respect to the worst case input:

Theorem 16. *The soft minimum approximation error for some given β and number of inputs L is limited by:*

$$\max_{[\theta_1,\ldots,\theta_L]} E < \frac{L-1}{\beta \cdot e} \tag{2.136}$$

Proof. See Appendix A.2.3 on page 109. □

It is important to note that these worst case inputs have to be close to the minimum input and have to be closer to it for higher values of β. This behavior of the worst case error contribution can also be seen in Figure 2.4a on the preceding page.

2.8.3 Zero Crispness

Another interesting property of the soft minimum is the border case for $\beta = 0$:

$$\Rightarrow \forall l \in \{1,\ldots,L\} : e_l = \exp(-0 \cdot \theta_l) = 1 \tag{2.137}$$

$$\Rightarrow Z = \sum_{l=1}^{L} e_l = \sum_{l=1}^{L} 1 = L \tag{2.138}$$

$$\Rightarrow \mathrm{softmin}[\theta_1,\ldots,\theta_L] = \frac{1}{Z}\sum_{l=1}^{L} e_l \cdot \theta_l = \frac{1}{L}\sum_{l=1}^{L} \theta_l \tag{2.139}$$

In this case softmin is apparently equal to the arithmetic average of all input values.

2.8.4 Scale-Invariance

The soft minimum is scale-invariant if β can be tuned accordingly, that is to say: If we know the scaling factor α we can set $\beta' := \frac{\beta}{\alpha}$ and obtain:

$$\mathrm{softmin}_{\beta'}[\alpha \cdot \theta_1,\ldots,\alpha \cdot \theta_L] = \frac{\sum_{l=1}^{L} \exp(\alpha \cdot \beta' \cdot \theta_l) \cdot \alpha \cdot \theta_l}{\sum_{l=1}^{L} \exp(\alpha \cdot \beta' \cdot \theta_l)} \tag{2.140}$$

$$= \alpha \cdot \frac{\sum_{l=1}^{L} \exp(\beta \cdot \theta_l) \cdot \theta_l}{\sum_{l=1}^{L} \exp(\beta \cdot \theta_l)} \tag{2.141}$$

$$= \alpha \cdot \mathrm{softmin}_{\beta}(\theta_1,\ldots,\theta_L) \tag{2.142}$$

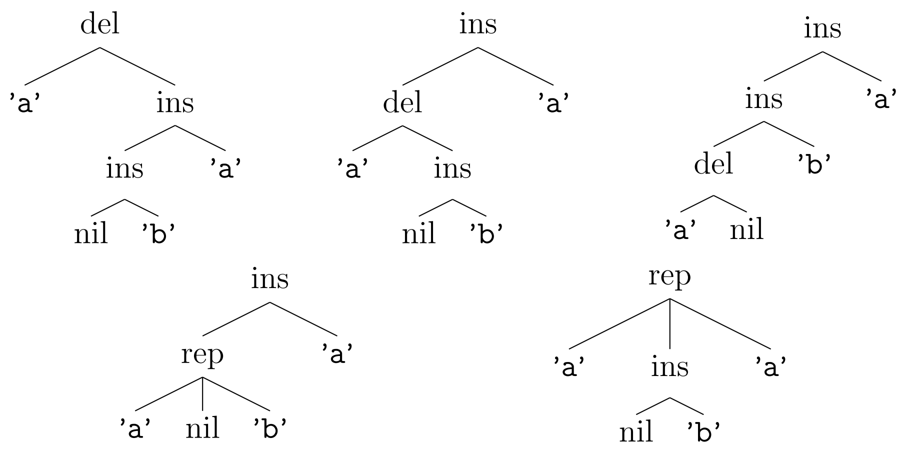


Figure 2.5: All (five) trees in $\mathcal{L}(\mathcal{G}_{\text{Glob}})$ that have the yield $(\texttt{a}, \texttt{ab})$.

$\bar{x}\backslash\bar{y}$	a	b	-
a	1.46	1.42	1
-	2	1	0

Table 2.3: The translated dynamic programming matrix for the nonterminal symbol ALI from $\mathcal{G}_{\text{Glob}}$ for the example input $\bar{x} = \texttt{a}$ and $\bar{y} = \texttt{ab}$ and the choice function softmin.

2.8.5 Relation to Physics

Interestingly, the soft minimum can be interpreted in terms of physics. So-called *Maxwell-Boltzmann statistics* describe the expected number of particles N_i with energy E_i within a gas of non-interacting material particles in thermal equilibrium [44]:

$$N_i = N \cdot \frac{g_i \cdot \exp(-\frac{E_i}{kT})}{\sum_j g_j \cdot \exp(-\frac{E_j}{kT})} \tag{2.143}$$

where k is the Boltzmann constant, T the temperature of the system and g_i the so called degeneracy of energy level E_i. The formula expresses the insight that one expects less particles with high energies, but that high energetic particles are more probable at higher temperatures.

This bears striking resemblance to the soft minimum approximation: softmin can be interpreted as the expected energy of the system given some temperature T. We can interpret β as the inverse system temperature: $\beta = 0$ means an infinite temperature where each state is equally probable independent of its energy, while $\beta \to \infty$ maps to absolute zero, where we only find particles with minimum energy.

2.8.6 Choice Function Properties

While the soft minimum obviously satisfies the definition of a choice function as given in Equation 2.13 on page 19 it does *not* satisfy Bellman's Principle of Optimality, as phrased in Section 2.4.1 on page 31. As proof consider the simple example of grammar $\mathcal{G}_{\text{Glob}}$, the trivial algebra in Equations 2.30 to 2.32 on page 24 and the input sequences $\bar{x} = \texttt{a}$ and $\bar{y} = \texttt{ab}$. All possible trees $T \in \mathcal{L}(\mathcal{G}_{\text{Glob}})$ with $\mathcal{Y}(T) = (\texttt{a}, \texttt{ab})$ are shown in Figure 2.5. The results of $\mathcal{F}(T)$ are (from left to right and top to bottom) 3, 3, 3, 2 and 1. Therefore, the softmin-result for $\beta = 1$ would be:

$$\frac{3 \cdot (3 \cdot e^{-\beta \cdot 3}) + 2 \cdot e^{-\beta \cdot 2} + 1 \cdot e^{-\beta \cdot 1}}{3 \cdot e^{-\beta \cdot 3} + e^{-\beta \cdot 2} + e^{-\beta \cdot 1}} = \frac{9 \cdot e^{-2} + 2 \cdot e^{-1} + 1}{3 \cdot e^{-2} + e^{-1} + 1} \approx 1.67 \tag{2.144}$$

Apparently, this is not equal to the result of our ADP algorithm, which is illustrated in Table 2.3. As we have shown in Theorem 15 on page 47, however, the soft minimum converges towards the true minimum with exponential speed with respect to β. Furthermore we are not interested in the "real" value of softmin anyways, as we use it as an approximation of the strict minimum. However, we need to be sure that the decomposition does not uncontrollably increase the approximation error with respect to the strict minimum. In other words: If softmin is applied multiple times instead of only once, the approximation error should not increase dramatically. This is guaranteed by Theorem 15 on page 47 and Theorem 16 on page 48. Interestingly, the decomposition might actually reduce the original approximation error, if the ϵ terms are sufficiently large compared to β.

Chapter 3

Implementation

Our implementation is the *TCS Alignment Toolbox*[1], a highly flexible Matlab-compatible Java library, which we have released as free software under the AGPLv3[2].

In Section 3.1 on the next page we describe the sequence datastructure, which serves as our input. To process the input we need implementation counterparts of key concepts of Algebraic Dynamic Programming (ADP), as described in Chapter 2, namely signature, algebra, choice function and grammar: The signature is an `Enum` listing the operations of $\mathcal{T}^*$. Algebras take the form of `AlignmentSpecifications`, which we describe in Section 3.2 on page 55. Users can encode grammars by implementing the `Grammar` interface, which we discuss in more detail in Section 3.3 on page 58. These objects then are plugged into an `AlignmentAlgorithm` for calculation. The different `AlignmentAlgorithms` are child-classes of the `AbstractADPAlgorithm`, which implements the generic dynamic programming algorithm shown in Algorithm 2 on page 30. They are different in terms of their implemented choice function and return value. Choice functions had to be hard-coded in the algorithm implementation, because the calculation of the output depends on the specific choice function. For example: An algorithm can only reconstruct an optimal alignment via backtracing if the choice function is the strict minimum rather

[1]`http://opensource.cit-ec.de/projects/tcs`
[2]`http://www.gnu.org/licenses/agpl-3.0.de.html`

Concept	**Implementation**
node set $\Sigma_\times$	`NodeSpecification`
sequence $\bar{x}$	`Sequence`
signature $\mathcal{T}^*$	`OperationType`
algebra $\mathcal{F}$	`AlignmentSpecification`
choice function h	`AlignmentAlgorithm`
grammar $\mathcal{G}$	`Grammar` or `AlignmentAlgorithm`
Algorithm 2	`AbstractADPAlgorithm`

Table 3.1: The correspondence of theoretical concepts from Chapter 2 with Java classes in the *TCS Alignment Toolbox* described in this chapter.

than the soft minimum. Furthermore, optimization techniques in gradient calculation depend on the choice function as well. We describe AlignmentAlgorithms in more detail in Section 3.4 on page 62. The correspondence between concepts from Chapter 2 and Java classes in the implementation is also outlined in Table 3.1 on the preceding page.

The flexibility of the AbstractADPAlgorithm comes at the cost of overhead, slowing down the computation. Therefore, we provide hard-coded combinations of choice function and grammar for some standard cases (Global Alignment, Dynamic Time Warping and a variation of Affine Alignment) as AlignmentAlgorithms. For each case we also provide a version of the AlignmentAlgorithm that is able to calculate the gradient with respect to comparator function parameters or keyword weights, as we describe in Section 3.5 on page 64. We close this chapter with some short notes on parallel processing in Section 3.6 on page 66.

3.1 Node Specification and Sequences

The package de.citec.tcs.alignment.sequence contains datastructures for the specification of the algebra function input set as well as the sequence datastructure itself. When using the toolbox, one starts by listing the keywords with the respective alphabets. Currently we support three different types of alphabet:

- Discrete alphabets, defined by a user-specified list of strings (SymbolicKeywordSpecification).
- $\mathbb{R}^n$ for some dim $\in \mathbb{N}$ chosen by the user (VectorialKeywordSpecification).
- The set of all possible strings (StringKeywordSpecification).

A tuple of a keyword and a alphabet is called a KeywordSpecification. An array of such specifications is a NodeSpecification.

Lets consider the example of robotic movement data from Section 2.1 again. As before, we have three joint angles of the robots arm movement, which can be described as vectorial data. The qualitative descriptor (moving or not moving) is best captured as a discrete/symbolic alphabet, while the researchers comment is a string. The java code for specifying this node set looks like this:

```java
// First we specify the three joint angles.
// The first argument of the constructor is the dimensionality.
// The second argument is the keyword
VectorialKeywordSpecification alpha = new
    VectorialKeywordSpecification(1, "alpha");
VectorialKeywordSpecification beta = new
    VectorialKeywordSpecification(1, "beta");
VectorialKeywordSpecification gamma = new
    VectorialKeywordSpecification(1, "gamma");
// Now we specify the qualitative phase descriptor.
Alphabet phases = new Alphabet(new String[]{"pause", "movement"});
```

```
SymbolicKeywordSpecification phase = new
    SymbolicKeywordSpecification(phases, "phase");
// Finally we specify the comment.
StringKeywordSpecification comment = new
    StringKeywordSpecification("comment");

// The set of these specifications makes up the input set.
NodeSpecification nodeSpec = new NodeSpecification(new
    KeywordSpecification[]{
           alpha, beta, gamma, phase, comment
     });
```

The NodeSpecification defines our node set. We can now build sequences over this node set. Remember that a sequence is defined as a succession of nodes, which in turn is defined as a vector of values. This conceptualization is directly implemented in the Java datastructures.

For the example data from Table 2.1 on page 16, the corresponding Java code would look like this:

```
// we start by creating the sequence object itself.
final Sequence seq = new Sequence(nodeSpec);

// then we successively add nodes.
Node n = new Node(seq);
n.setValue("alpha", new VectorialValue(30));
n.setValue("beta", new VectorialValue(10));
n.setValue("gamma", new VectorialValue(10));
n.setValue("phase", new SymbolicValue(phases, "pause"));
n.setValue("comment", new StringValue("Begin of pause before first
     movement"));
seq.getNodes().add(n);

n = new Node(seq);
n.setValue("alpha", new VectorialValue(30));
n.setValue("beta", new VectorialValue(10));
n.setValue("gamma", new VectorialValue(10));
n.setValue("phase", new SymbolicValue(phases, "pause"));
seq.getNodes().add(n);

n = new Node(seq);
n.setValue("alpha", new VectorialValue(40));
n.setValue("beta", new VectorialValue(10));
n.setValue("gamma", new VectorialValue(10));
n.setValue("phase", new SymbolicValue(phases, "movement"));
n.setValue("comment", new StringValue("Begin of movement. Quite
    rapidly at the start."));
seq.getNodes().add(n);

n = new Node(seq);
n.setValue("alpha", new VectorialValue(45));
n.setValue("beta", new VectorialValue(10));
n.setValue("gamma", new VectorialValue(10));
n.setValue("phase", new SymbolicValue(phases, "movement"));
seq.getNodes().add(n);

n = new Node(seq);
```

Name	Data Type	Description
L1Norm	vectorial	$\sigma(\lvert\vec{a}-\vec{b}\rvert)$
Euclidian	vectorial	$\sigma(\lVert a-b\rVert)$
TrivialEdit	symbolic	Returns a constant value for match, mismatch, deletion, insertion, skip-deletion und skip-insertion.
Replacement	symbolic	Specifies an explicit cost matrix with constant return values for each combination $(a,b) \in (\Sigma \cup \{-,\_\})^2$.
CharStat	string	Calculates character occurences for letters and numbers in both sequences, subtracts those feature vectors and returns the L1 norm of it divided by the added lengths of both sequences.
NCD	string	Calculates the Normalized Compression Distance (NCD) of both strings as described by Li et al. [23]. In essence, one calculates the compressed size of both concatenated strings minus the minimum of the compressed sizes of the separate strings and divides by the maximum of the compressed sizes of the separate strings. Speed and precise return values depend on the compressor.

Table 3.2: The different comparator functions available in the *TCS Alignment Toolbox*. Note that both vectorial comparator functions require a squashing function σ to ensure that their results stay in the range $[0,1]$.

```
n.setValue("alpha", new VectorialValue(50));
n.setValue("beta", new VectorialValue(10));
n.setValue("gamma", new VectorialValue(10));
n.setValue("phase", new SymbolicValue(phases, "movement"));
n.setValue("comment", new StringValue("End of movement."));
seq.getNodes().add(n);

n = new Node(seq);
n.setValue("alpha", new VectorialValue(50));
n.setValue("beta", new VectorialValue(10));
n.setValue("gamma", new VectorialValue(10));
n.setValue("phase", new SymbolicValue(phases, "pause"));
seq.getNodes().add(n);
```

Note that we did not set values for the comment every time. If values are not set they remain `null`.

3.2 Alignment Specification

Next we specify the algebra in terms of the comparator functions we intend to use. We provide a standard library of comparator function implementations in the de.citec.tcs.alignment.comparators package, which we have listed in Table 3.2 on the facing page. Note that a comparator function in this definition is actually a vector of comparator functions, one for each function template in $\mathcal{T}^*$ as described in Section 2.2.1 on page 17.

Of course, it is possible to plug in custom comparator functions. For that purpose the package contains interfaces in order to generically declare the alphabet of the comparator function as well as the function templates it implements. These interfaces are namely:

```
public interface Comparator<X extends Value> {

    public ValueType getType();

    public double compare(X a, X b);
}

public interface GapComparator<X extends Value> extends Comparator
   <X> {

    public double delete(X a);

    public double insert(X b);
}

public interface SkipComparator<X extends Value> extends
   GapComparator<X> {

    public double skipDelete(X a);

    public double skipInsert(X b);
}
```

By implementing the first interface, a Java class identifies itself as a comparator function that is able to handle replacements. The second interface additionally guarantees deletions and insertions. The last one guarantees skip-deletions and skip-insertions.

Many methods to define a distance between two values do not account for other operations like deletions. Take the Euclidian distance for example: What should we define as the result of $c^{\kappa}_{\text{del}}(a)$? The best definitions might very well depend on the data that is handled. We provide the means for two different definitions:

$$c^{\kappa}_{\text{del}}(a) := \lambda_{\text{del}} \tag{3.1}$$

$$c^{\kappa}_{\text{del}}(a) := c^{\kappa}_{\text{rep}}(a, \phi_{\text{del}}) \tag{3.2}$$

where λ_{del} is some constant in the interval $[0, 1]$ and $\phi_{\text{del}} \in \Sigma_{\kappa}$ is some constant value. In the former case we apply a constant deletion cost λ_{del} every time a value is deleted, no matter the input. The latter case introduces a more dynamic deletion cost by comparing a to some constant other value. For the Euclidian distance one could define:

$$c^{\kappa}_{\text{del}}(a) := 1 \tag{3.3}$$

$$c^{\kappa}_{\text{del}}(a) := c^{\kappa}_{\text{rep}}(\vec{a}, \vec{0}) = \|\vec{a}\| \tag{3.4}$$

The same method can be applied for insertions, skip-deletions and skip-insertions as well, of course. We have implemented these methods to handle other operations than replacements in the abstract classes `SkipExtendedComparator` for the former case and `ComparisonBasedSkipExtendedComparator` for the latter case.

Another issue is that both vectorial comparator functions, the L1-distance and the Euclidian distance, do not necessarily return output in the range $[0, 1]$. Therefore, we need to apply some kind of squashing function, which we define as some function of the form

$$\sigma : \mathbb{R}^+ \rightarrow [0, 1] \tag{3.5}$$

In the `comparators` package we provide three different squashing (or normalizer) functions:

- If the `Comparator` (maybe due to the nature of the data at hand) just projects to a larger range $[\min, \max]$, one can apply the `AffineNormalizer`, which is defined as:

  $$\sigma_{\text{aff}}(s) := \frac{s - \min}{\max - \min} \tag{3.6}$$

- A modest squashing behaviour is provided by the `HyperbolicNormalizer`, which is defined as:

  $$\sigma_{\text{hyper}}(s) := 1 - \frac{1}{1 + \alpha \cdot s} \tag{3.7}$$

  For $s = 0$ it returns 0 as well and it goes towards 1 for $s \rightarrow \infty$. α manipulates the convergence speed and is a hyper-parameter. It should be set according to the scaling of the underlying data set. A natural choice is the inverse of the mean for s.

- Much faster convergence is achieved by the `ExponentialNormalizer`, which is defined as:

  $$\sigma_{\text{exp}}(s) := 1 - \exp(-\beta \cdot s) \tag{3.8}$$

  As the symbol suggests, β has the same function as the crispness parameter in the soft minimum: It regulates the convergence speed.

We provide an interface in order to allow users to program their own squashing/normalizer functions. The interface is very straightforward:

```
public interface Normalizer {

    public double normalize(final double distance);
}
```

In our example the joint angles can be compared using a L1NormComparator. As angles necessarily are in the range $[0°, 360°]$ we can apply an AffineNormalizer. For the qualitative descriptor we can use a ReplacementComparator. The default scoring scheme for such a comparator function is:

$$c^{\kappa}_{\text{rep}}(a, b) := \begin{cases} 0 & \text{if } a = b \\ 1 & \text{if } a \neq b \end{cases} \tag{3.9}$$

$$c^{\kappa}_{\text{del}}(a) = c^{\kappa}_{\text{ins}}(b) := 1 \tag{3.10}$$

$$c^{\kappa}_{\text{skip\_del}}(a) = c^{\kappa}_{\text{skip\_ins}}(b) := 1 \tag{3.11}$$

for all $a, b \in \Sigma_\kappa$. We do not want to consider the comment keyword in the alignment distance, which is why we do not define a comparator function for it.

The algebra in combination with keyword weights forms an AlignmentSpecification. For our example data set, we want to put emphasis on the qualitative descriptor. Furthermore, changes in the first joint might have stronger effect on the final position of the robot arm than changes in the second or third joint. So a good weighting might look like this:

$$g_\alpha := 0.3 \tag{3.12}$$

$$g_\beta := 0.2 \tag{3.13}$$

$$g_\gamma := 0.1 \tag{3.14}$$

$$g_{\text{phase}} := 0.4 \tag{3.15}$$

The corresponding Java code is as follows:

```
// We start by constructing the normalizer object.
AffineNormalizer angleSquash = new AffineNormalizer(0, 360);
// The comparator function is defined by the dimensionaliy
// and the normalizer.
L1NormComparator angleComp = new L1NormComparator(1, angleSquash);
// Further we construct the ReplacementComparator
// We specify the cost matrix first and use default values.
ReplacementCosts costMatrix = new ReplacementCosts(phases, true,
    true);
ReplacementComparator phaseComp = new ReplacementComparator(
    costMatrix);

// Finally we create the AlignmentSpecification object
AlignmentSpecification alignSpec = new AlignmentSpecification(
    nodeSpec, new String[]{"alpha", "beta", "gamma", "phase"},
    new Comparator[]{angleComp, angleComp, angleComp, phaseComp},
    new double[]{0.3, 0.2, 0.1, 0.4});
```

3.3 Grammar

The third key ingredient to specify an alignment algorithm is an ADP grammar. In the *TCS Alignment Toolbox*, a grammar is some Java object implementing the Grammar interface, which can be found in the de.citec.tcs.alignment.adp package:

```
public interface Grammar<N extends Enum<N>> {

    public Class<N> getNonterminalClass();

    public N[] dependencySort();

    public N getAxiom();

    public EnumSet<N> getAccepting();

    public List<ProductionRule<N>> getPossibleRules(N nonterminal,
        int leftSize, int rightSize);

    public boolean containsGaps();

    public boolean containsSkips();
}
```

Note that N is a Java generic class, where implementing classes can specify an Enum that lists the nonterminal symbols of the respective grammar. Further note that ProductionRule is a Java class describing production rules in terms of the operations they apply and the nonterminal symbol they produce. Given these basics, most of the methods of the Grammar interface are straightforward and easy to implement. Consider the example of Affine Alignment (Grammar 2.4 on page 29). The implementation might look like this:

```
public class AffineGrammar implements Grammar<Nonterminal> {

    public Nonterminal[] dependencySort() {
        ...
    }

    public List<ProductionRule<Nonterminal>> getPossibleRules(N
       nonterminal, int leftSize, int rightSize) {
        ...
    }

    public boolean containsGaps() {
        return true;
    }

    public boolean containsSkips() {
        return true;
    }

    public Class<Nonterminal> getNonterminalClass() {
        return Nonterminal.class;
    }

    public Nonterminal getAxiom() {
        return Nonterminal.SKIPDEL_START;
    }

    private final EnumSet<Nonterminal> accepting = EnumSet.of(
        Nonterminal.SKIPINS_START, Nonterminal.SKIPINS_END);

    public EnumSet<Nonterminal> getAccepting() {
        return accepting;
    }

    /**
     * These are the nonterminal symbols for the affine grammar.
     */
    public static enum Nonterminal {
        SKIPINS_END, SKIPDEL_END,
        ALI, DEL, INS,
        SKIPDEL_MID, SKIPINS_MID,
        SKIPINS_START, SKIPDEL_START,
    }
}
```

We want to talk in more detail about two more difficult methods already mentioned in Section 2.4 on page 28: `dependencySort` and `getPossibleRules`:

`dependencySort` is supposed to return the nonterminal symbols of the grammar as an array that is sorted according to the following order: If a production rule in Δ exists that has the form

```
A = B;
```

then B is in front of A in the array. Ideally, users sort the nonterminal symbols in the `Enum` directly in a way that respects this order. Then the implementation is just

```
public Nonterminal[] dependencySort() {
    return Nonterminal.values();
}
```

A generic way to determine a proper ordering of nonterminal symbols is depicted in Algorithm 6 on the following page. The basic idea is to just serialize the dependency trees in an inverse depth-first-search manner, that is: For some given nonterminal symbol A, we first add the nonterminal symbols it depends on and then A itself.

`getPossibleRules` is also fairly simple to implement for any specific given grammar. As an example consider $\mathcal{G}_{\text{Glob}}$:

```
private final ProductionRule<Nonterminal> del = new ProductionRule
    <>(OperationType.DELETION, Nonterminal.ALI);
private final ProductionRule<Nonterminal> ins = new ProductionRule
    <>(OperationType.INSERTION, Nonterminal.ALI);
private final ProductionRule<Nonterminal> rep = new ProductionRule
    <>(OperationType.REPLACEMENT, Nonterminal.ALI);
```

Algorithm 6 A generic implementation for the `dependencySort` function of the `Grammar` interface. Given the production rules Δ of some grammar $\mathcal{G}$ we can sort the nonterminal symbols of $\mathcal{G}$ in such a way that a production rule of the form $\text{A} = \text{B};$ implies that B is before A in the output.

```
Let $\mathcal{G} = (\Phi, \mathrm{A}^*, K, \{(\kappa, \Sigma_\kappa)\}_{\kappa \in K}, \mathcal{T}^*, \Delta)$ be an ADP grammar.
Initialize $V \leftarrow \emptyset$, which is the set of nonterminal symbols that we have already processed.
Initialize $L$ as an empty list, which will be our output.
for $\mathrm{A} \in \Phi$ do
    if $\mathrm{A} \in V$ then
        Continue with the next nonterminal symbol.
    end if
    SERIALIZE(A, $V$, $L$)
end for
return $L$.

function SERIALIZE(A, $V$, $L$)
    if $\mathrm{A} \in V$ then
        $\Delta$ contains loops (see Definition 16 on page 21).
    end if
    Add A to $V$.
    for production rules of the form $\mathrm{A} = \mathrm{B};$ in $\Delta$ do
        SERIALIZE(B, $V$, $L$)
    end for
    Append A to $L$.
end function
```

```
public List<ProductionRule<Nonterminal>> getPossibleRules(
    Nonterminal nonterminal, int leftSize, int rightSize) {
    final ArrayList<ProductionRule<Nonterminal>> rules = new
        ArrayList<>();
    if (leftSize > 0) {
        rules.add(del);
    }
    if (rightSize > 0) {
        rules.add(ins);
    }
    if (leftSize > 0 && rightSize > 0) {
        rules.add(rep);
    }
    return rules;
}
```

This expresses very simple rules: If there is something left in the left input sequence, we can apply a deletion. If there is something left in the right input sequence, we can apply an insertion. If both is the case, we can apply either a deletion, an insertion or a replacement. While this seems very straightforward, there is no trivial algorithm to generically determine the implementation of this function given some arbitrary grammar. This is due to the fact that the production rules we can apply depend on the nonterminal symbol this production rule produces. Consider the example of $\mathcal{G}_{\text{Affine}}$ (Grammar 2.4 on page 29). More specifically, consider the production rule

```
INS = ins(<EMPTY, NODE>, INS);
```

Naively one would argue that this has to be applicable if at least one node is available in the right input sequence. But assume we consume the last node in the right input sequence with an insertion. Then we have no production rules left to apply, because both

```
INS =           ins(<EMPTY,NODE>, INS) |
                rep(<NODE,NODE>, ALI);
```

require another node in the right input sequence. Applying this rule leads us to a dead end. Even worse: Even if the nonterminal symbol we obtain after applying a production rule is no dead end in itself, it becomes a dead end if all of its production rules lead to a dead end, and so forth. Due to this recursive problem structure, one essentially needs a dynamic programming algorithm in itself to generically determine the applicable production rules (see Algorithm 3 on page 33), which produces considerable overhead.

However, we still implemented both generic algorithms in the FlexibleGrammar class, which allows users to define new grammars at runtime by just instantiating a new FlexibleGrammar object and adding ProductionRule objects to it. The code to create the global alignment grammar $\mathcal{G}_{\text{Glob}}$, for example, looks like this:

```
FlexibleGrammar global = new FlexibleGrammar<>(GlobalNonterminals.
    class, GlobalNonterminals.ALI);
global.getAccepting().add(GlobalNonterminals.ALI);
global.addRule(GlobalNonterminals.ALI, new ProductionRule<>(
    OperationType.REPLACEMENT, GlobalNonterminals.ALI));
```

```
global.addRule(GlobalNonterminals.ALI, new ProductionRule<>(
    OperationType.DELETION, GlobalNonterminals.ALI));
global.addRule(GlobalNonterminals.ALI, new ProductionRule<>(
    OperationType.INSERTION, GlobalNonterminals.ALI));
```

Note that this still requires an `Enum` listing all nonterminal symbols.

3.4 Alignment Algorithms

The generic dynamic programming algorithm for the alignment distance (Algorithm 2 on page 30) is implemented in the class `AbstractADPAlgorithm`, independent of the choice function. It does, however, require of subclasses to provide a choice function implementation as well as an implementation of the transformation to the desired output format:

```
public abstract class AbstractADPAlgorithm<R, N extends Enum<N>>
    implements SkipAlignmentAlgorithm<R> {

     ...

     public abstract double choice(double[] choices);

     public abstract R transformToResult(...);
}
```

Note that the subclasses are also required to implement the `AlignmentAlgorithm` interface, which is very basic:

```
public interface AlignmentAlgorithm<R> {

     public AlignmentSpecification getSpecification();

     public R calculateAlignment(Sequence a, Sequence b);

     public Class<R> getResultClass();
}
```

The former two methods are actually already implemented by the `AbstractADPAlgorithm`. Still, the `AlignmentAlgorithm` is expected to take care of the choice function h and the output. We distinguish four main types, which correspond to subclasses of `AbstractADPAlgorithm` provided in the `de.citec.tcs.alignment.adp` package:

- $h = \min$ and the output is just the alignment distance itself. This is implemented in the `StrictADPScoreAlgorithm`.
- $h = \min$ and the output is an optimal alignment, retrieved via backtracing. This is implemented in the `StrictADPFullAlgorithm`.
- $h = \text{softmin}$ and the output is just the alignment distance itself. This is implemented in the `SoftADPScoreAlgorithm`.
- $h = \text{softmin}$ and the output are all dynamic programming tables as basis for further gradient calculation. We call this return value a `SoftPathModel`. This is implemented in the `SoftADPFullAlgorithm`.

Choice	Grammar	Return Value	Name
min	$\mathcal{G}_{\text{Glob}}$	$D(\bar{x}, \bar{y})$	StrictAlignmentScore
min	$\mathcal{G}_{\text{Glob}}$	$\text{argmin}_T\, \mathcal{F}(T)$	StrictAlignmentFull
min	$\mathcal{G}_{\text{Glob}}$	$\{\text{argmin}_T\, \mathcal{F}(T)\}$	StrictAlignmentAllOptimal
min	$\mathcal{G}_{\text{Glob}}$	$\{\text{argmin}_T^K\, \mathcal{F}(T)\}$	StrictAlignmentKPath
softmin	$\mathcal{G}_{\text{Glob}}$	$D(\bar{x}, \bar{y})$	SoftAlignmentScore
softmin	$\mathcal{G}_{\text{Glob}}$	$A, B, \ldots$	SoftAlignmentFull
softmin	$\mathcal{G}_{\text{Glob}}$	$T \sim P_{\text{softmin}}(T)$	StrictAlignmentSampling
min	$\mathcal{G}_{\text{Glob}}$	$D(\bar{x}, \bar{y})$	StrictDTWScore
min	$\mathcal{G}_{\text{Glob}}$	$\text{argmin}_T\, \mathcal{F}(T)$	StrictDTWFull
min	$\mathcal{G}'_{\text{Affine}}$	$D(\bar{x}, \bar{y})$	StrictAffineAlignmentScore
min	$\mathcal{G}'_{\text{Affine}}$	$\text{argmin}_T\, \mathcal{F}(T)$	StrictAffineAlignmentFull
softmin	$\mathcal{G}'_{\text{Affine}}$	$D(\bar{x}, \bar{y})$	SoftAffineAlignmentScore
softmin	$\mathcal{G}'_{\text{Affine}}$	$A, B, \ldots$	SoftAffineAlignmentFull

Table 3.3: The different hard-coded AlignmentAlgorithms available in the *TCS Alignment Toolbox*. The name of the corresponding Java class is generated as choice function + grammar + return value + Algorithm. So the Java class implementing the strict minimum as choice function, using $\mathcal{G}_{\text{Glob}}$ and returning just the score is called StrictAlignmentScoreAlgorithm. Note that the $\mathcal{G}'_{\text{Affine}}$ is not exactly the same as Grammar 2.4 on page 29, as it does not use a proper gap opening cost but allows for skips directly.

Each of those subclasses takes an AlignmentSpecification and a Grammar as arguments in the constructor and can subsequently return the output for any two given input sequences. The alignment distance itself is normalized to be in the range $[0, 1]$ by dividing the output by the added length of both sequences, $M + N$.

As an example imagine we want to use the previously created grammar $\mathcal{G}_{\text{Glob}}$ to align two sequences of robot movement data. The Java code for that would look like this:

```java
// We create the algorithm object.
SoftADPFullAlgorithm algo = new SoftADPFullAlgorithm(alignSpec);
// We set the crispness value.
algo.setBeta(5);
// We calculate the Alignment.
SoftADPPathModel dpTables = algo.calculateAlignment(seq, seq2);
```

Unfortunately, the generic implementation in the AbstractADPAlgorithm comes at the cost of producing some overhead (e.g. due to the getPossibleRules function). Therefore, we also provide some hard-coded AlignmentAlgorithms for some standard grammars in the de.citec.tcs.alignment package, which are listed in Table 3.3. We have named them according to their properties:

- The prefix Strict marks algorithms with the choice function being the strict minimum, while the prefix Soft marks algorithms with softmin as the choice function.

- The mid part of the name marks the grammar: Either `Alignment` for $\mathcal{G}_{\text{Glob}}$ (Global Sequence Alignment, see Grammar 2.1 on page 22), `DTW` for $\mathcal{G}_{\text{Glob}}$ with a tweaked algebra (Dynamic Time Warping, see Section 2.2.4 on page 23) or `AffineAlignment` for $\mathcal{G}_{\text{Affine}}$ (Affine Sequence Alignment, see Grammar 2.4 on page 29).

- The suffix denotes the return value of the algorithm. `Score` means that the algorithm just computes the alignment distance itself. `Full` means that either one optimal alignment is returned using backtracing (in case of `Strict` algorithms) or the dynamic programming tables themselves to facilitate gradient calculation.

  Note that the backtracing is somewhat problematic as multiple optimal alignments may exist. Formally, the following set might have more than one element:

  $$\{T|T \in \mathcal{L}(\mathcal{G}), \mathcal{Y}(T) = (\bar{x}, \bar{y}), \mathcal{F}(T) = D(\bar{x}, \bar{y})\} \tag{3.16}$$

  A `Full` algorithm actually draws one (random) alignment from that set. For the case of $\mathcal{G}_{\text{Glob}}$ we also provide an `AllOptimal` version returning this whole set, a `KPath` version returning the `K` best alignment trees and a `Sampling` version which samples `K` alignments from $\mathcal{L}(\mathcal{G}_{\text{Glob}})$ according to their softmin probability. The softmin probability is given as

  $$P_{\text{softmin}}(T) := \frac{\exp(-\beta \cdot \mathcal{F}(T)) \cdot \mathcal{F}(T)}{\sum_{T':T' \in \mathcal{L}(\mathcal{G}), \mathcal{Y}(T')=(\bar{x},\bar{y})} \exp(-\beta \cdot \mathcal{F}(T')) \cdot \mathcal{F}(T')} \tag{3.17}$$

One exception to that scheme is the `KernelDTWFullAlgorithm` programmed by Georg Zentgraf, which calculates a kernel rather than a distance and returns the dynamic programming tables for it.

3.5 Gradient Calculation

The return values of `Full`, `KPath`, `AllOptimal` and `Sampling` algorithms implement a common interface, which provides the means to calculate the gradient:

```java
public interface AlignmentDerivativeAlgorithm {

    public <X extends Value, Y> double[]
        calculateRawParameterDerivative(DerivableComparator<X, Y>
        comp, String keyword);

    public <X extends Value, Y> Y calculateParameterDerivative(
        DerivableComparator<X, Y> comp, String keyword);

    public double[] calculateWeightDerivative();
}
```

The first two functions require an implementation of the gradient calculation as shown in Theorem 11 on page 43, while the last one refers to Theorem 12 on page 44. Note that the interface requires us to provide the keyword κ as well as the comparator function c^κ for which all parameter gradients shall be calculated. Comparators that have parameters for which a gradient can be calculated are required to implement the DerivableComparator interface:

```java
public interface DerivableComparator<X extends Value, Y> {

    public double calculateLocalDerivative(int paramIdx, X a, X b,
         OperationType type);

    public Y transformToResult(double[] derivatives);

    public int getNumberOfParameters();

    public Class<Y> getResultClass();
}
```

The indexing scheme for the parameters is determined by the Comparator. The method calculateLocalDerivative is supposed to return the value of

$$\frac{\partial}{\partial \lambda_p} c_t(a, b) \tag{3.18}$$

where type specifies the function template from $\mathcal{T}^*$ in Section 2.2.1 on page 17. transformToResult is supposed to transform the finished vector of parameter gradients to some user-readable format that depends on the comparator function. For the ReplacementComparator, for example, it is intuitive to return the cost parameters as a matrix, while other comparator function just return the input vector as it is. Using this interface the implementation of calculateParameterDerivative of the AlignmentDerivativeAlgorithm becomes trivial:

```java
@Override
public <X extends Value, Y> Y calculateParameterDerivative(
    DerivableComparator<X, Y> comp, String keyword) {
    return comp.transformToResult(calculateRawParameterDerivative(
        comp, keyword));
}
```

Note that these interfaces permit users to calculate gradients for their own comparator functions as long as they provide the necessary interface implementations.

If the algorithm returns an alignment T the gradient is not calculated exactly but rather by an approximation described by Mokbel et al. [29]: We count the operations used within the alignment and calculate the alignment based on those operations.

If dynamic programming tables are returned the gradient calculation is done as described in Theorem 11 and 12 on page 44. However, we do employ heuristics to limit the search space somewhat more. These heuristics are described by Mokbel et al. [29].

In our example (and within Chapter 4) the only two gradients that matter are the gradient with respect to the parameters of the ReplacementComparator as well as the gradient with respect to the keyword weights. The respective Java code looks like this:

```
// calculate the gradient for the cost matrix of phaseComp.
double[][] phaseGradient = dpTables.calculateParameterDerivative(
    phaseComp, "phase");
// calculate the gradient for the weights.
double[] weightGradient = dpTables.calculateWeightDerivative();
```

3.6 Parallel Processing

In order to make the most of modern mutli-CPU hardware the *TCS Alignment Toolbox* also has the capacity to calculate the alignment distance and the gradient thereof in parallel. The respective Java classes are ParallelProcessingEngine and ParallelDerivativeEngine. The parallel computing is implemented in a straightforward way: Given a data set X all terms:

$$D(\bar{x}, \bar{y}), \bar{x}, \bar{y} \in X \tag{3.19}$$

are computed in parallel. This does apply to all possible return values of course, not only to the plain distance. In Java terms, we have the following setup:

```
// We assume to already have set up our sequences.
Sequence[] seqs = ...;
// As well as our AlignmentAlgorithm.
AlignmentAlgorithm<R> algo = ...;
// Now we set up the ParallelProcessingEngine.
ParallelProcessingEngine<R> engine = new ParallelProcessingEngine(
    algo, seqs);
// We can set additional parameters like the number of threads.
engine.setNumberOfThreads(64);
// Finally we give the calculation command.
engine.calculate();
// And we retrieve the result.
R[][] result = engine.getResultMatrix();
```

The ParallelDerivativeEngine expects instances of the AlignmentDerivativeAlgorithm interface as input and calculates either a parameter or keyword weight gradient for them in parallel. The setup is quite similar:

```
// We assume to already have a vector (or matrix) of
// AlignmentDerivativeAlgorithms.
AlignmentDerivativeAlgorithm[] fullResults = ...;
// And a DerivableComparator
DerivableComparator<X,Y> comp = ...;
// Then we set up the ParallelDerivativeEngine.
ParallelDerivativeEngine engine = new ParallelDerivativeEngine(
    fullResults);
// We can set additional parameters like the number of threads.
engine.setNumberOfThreads(64);
// Finally we can calculate the gradients in parallel
double[][] weightGradients = engine.calculateWeightDerivatives();
Y[] paramGradients = engine.calculateParameterDerivatives(comp,
    keyword);
```

Chapter 4

Experiments

In this chapter we apply our learning approach in experimental setups. In all experiments we try to optimize the classification accuracy of a k-Nearest Neighbor (k-NN) classifier by performing a gradient descent on the LMNN cost function as described in Section 2.6 on page 40. The alignment distances itself, as well as the gradient calculation on them, is implemented in the *TCS Alignment Toolbox* described in Chapter 3. More specifically, we use the `SoftAlignmentFullAlgorithm` as implementation of Global Alignment (see Grammar 2.1 on page 22) and the `SoftADPFullAlgorithm` with the `AffineGrammar` as implementation of Affine Alignment (see Grammar 2.4 on page 29). In the experiments we compare the performance of both algorithms.

We experiment on three different data sets, the first two being artificically constructed, while the latter one stems from a practical domain.

1. In the string data set described in Section 4.1 on the following page, we learn the optimal parameter matrix of a `ReplacementComparator` to obtain an alignment distance that can distinguish between two classes of strings under conditions of noise.

2. In the time series data set described in Section 4.2 on page 73, we identify the data dimension that helps to distinguish two classes of multi-dimensional time series and to ignore all other dimensions only containing noise.

3. The last data set described in Section 4.3 on page 77 consists of multimodal sequences that model Java programs. We train an alignment distance in order to distinguish between implementations of the algorithms *InsertionSort* and *BubbleSort*.

All sections have a common structure: We first describe the data set itself, then our experimental hypotheses, our parameters and further methods and finally the results of the experiment as well as a discussion of these results.

4.1 String Data

4.1.1 Data Set Description

The data set is inspired by the replacement data set used in [27, 29]: We use only one keyword κ with the discrete alphabet

$$\Sigma_\kappa := \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{z}\} \tag{4.1}$$

Thus our sequences are essentially strings of characters.

We created 30 strings of 24 characters each for both classes, using the following creation procedure:

1. We start with a section of length 10, consisting of `a`s and `b`s chosen at random.

2. For the first class, two more random characters from the set $\{\mathtt{a}, \mathtt{b}\}$ follow, while in the second class we add two randomly chosen characters from the set $\{\mathtt{c}, \mathtt{d}\}$.

3. Then we switch around: For the first class we add two randomly chosen characters from the set $\{\mathtt{c}, \mathtt{d}\}$, while for the second class we add two randomly chosen characters from the set $\{\mathtt{a}, \mathtt{b}\}$.

4. Finally there is another section of 10 `a`s or `b`s for both classes.

Due to this creation process, we ensure that the created strings adhere to one of these regular expressions:

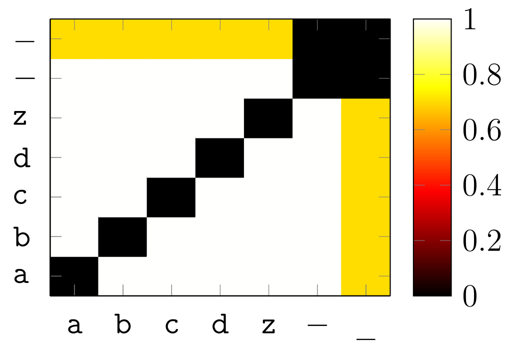
$$\text{first class} : (\mathtt{a}|\mathtt{b})^{10}(\mathtt{a}|\mathtt{b})^{2}(\mathtt{c}|\mathtt{d})^{2}(\mathtt{a}|\mathtt{b})^{10} \tag{4.2}$$
$$\text{second class} : (\mathtt{a}|\mathtt{b})^{10}(\mathtt{c}|\mathtt{d})^{2}(\mathtt{a}|\mathtt{b})^{2}(\mathtt{a}|\mathtt{b})^{10} \tag{4.3}$$

Apparently, the significant difference between both classes lies in the position of the $\mathtt{c}|\mathtt{d}$ section in the strings. The frequency of characters is equal for both classes. In order to distinguish between the classes we need to take the order of characters into account, which is why sequence alignment algorithms are a natural choice to analyze the data.

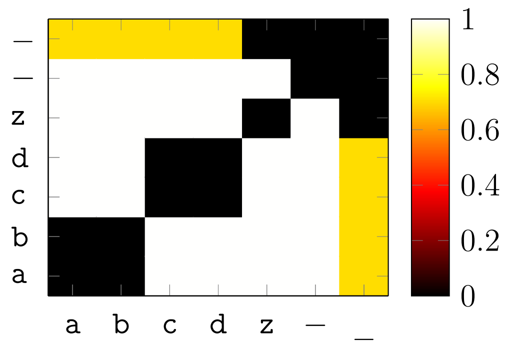
As comparator function we use a `ReplacementComparator`. Thus we can describe the parameters of our alignment distance as a matrix λ with one entry for each pair $a, b \in \Sigma_\kappa \cup \{-, \_\}$, which we denote as $\lambda(a, b)$. The initial parameters are:

$$\lambda(a,b) := \lambda(b,a) := \begin{cases} 0 & \text{, if } a = b \\ 1 & \text{, if } a \in \Sigma_\kappa \wedge b \in (\Sigma_\kappa \cup \{-\}) \setminus \{a\} \\ 0.7 & \text{, if } a \in \Sigma_\kappa \wedge b = \_ \end{cases} \tag{4.4}$$

A visualization of the parameter matrix is shown in Figure 4.1a on the next page. These initial parameters are the typical choice for a String Edit Distance: replacements cost nothing if the two characters match, but they have maximum



(a) The initial parameter matrix λ for a `ReplacementComparator`. The diagonal is zero, which means that self-replacements do not cost anything. Skip-deletions and skip-insertions cost 0.7 and all other operations cost 1. This is a classic String Edit Distance cost scheme.



(b) One possible parameter matrix λ for a `ReplacementComparator` to achieve perfect class separation on the string data set. The replacement costs between `a` and `b`, as well as between `c` and `d` are zero. To counteract noise, the costs of skip-insertions and skip-deletions of `z`s are zero as well.

Figure 4.1: The initial (left) and optimal (right) parameters for a `ReplacementComparator` in the string data set. Dark colors in the matrix cell indicate low values, while bright colors indicate high values.

cost (1) if the characters do not match. The same cost applies for deletions and insertions, while skip-deletions and skip-insertions are cheaper with a cost of 0.7.

The alignment distance induced by this initial setting, however, is not helpful in distinguishing between both classes: As the characters at each position in the strings are randomly chosen from two possible characters, the alignment distance with default parameters overestimates the dissimilarity inside the classes, such that strings from the same class and from different classes are about equally far away.

In order to obtain better classification performance, one has to acknowledge that `a`s and `b`s, as well as `c`s and `d`s are essentially the same. What really matters is the difference between these two pairs of characters. Therefore, an optimal parameter matrix would look like this:

$$\lambda(a, b) := \lambda(b, a) := \begin{cases} 0 & , \text{if } a, b \in \{\texttt{a}, \texttt{b}\} \vee a, b \in \{\texttt{c}, \texttt{d}\} \\ 1 & , \text{if } a \in \{\texttt{a}, \texttt{b}\} \wedge b \in \{\texttt{c}, \texttt{d}\} \end{cases} \tag{4.5}$$

The costs for deletion, insertion, skip-deletion and skip-insertion have to be set to some value other than 0. The alignment distance induced by these parameters collapses all data points in the same class to a single point while the inter-class distances remain positive. In previous work, we could show that these parameters can indeed be found using a similar learning approach in the Relational Generalized Learning Vector Quantization framework [27, 29].

As we want to analyze the performance of Affine Alignment versus Global Alignment, we need to add one further twist: We added noise to the strings in both classes. After creating the strings in the aforementioned fashion, we inserted a substrings of length 40, consisting only of `z`s, at a random position in 10% of the strings, independent of the class. As these noisy substrings are longer than the original string, a alignment distance with default parameters would deem them very dissimilar from the other strings in the same class. In order to counteract the noise, the costs for skip-deletions and skip-insertions of the letter `z` must be low or zero. The optimal parameter matrix for this noisy setting is shown in Figure 4.1b on the preceding page.

4.1.2 Hypotheses

Regarding this data set we have the following predictions:

Hypothesis 1. Due to the construction of the data set with a high level of noise, we expect a bad initial classification accuracy of about 50% (random classifier) on the test set.

Hypothesis 2. The classification accuracy should increase for both algorithms.

Hypothesis 3. Over time both alignment algorithms should find the correct replacement cost settings as specified in Equation 4.5 on the previous page.

Hypothesis 4. We expect Affine Alignment to lower the cost for skip-deletions and skip-insertions of `z`s.

Hypothesis 5. As Global Alignment can not employ skip-deletions or skip-insertions, we expect a different learning behavior to counteract the `z`-noise.

Hypothesis 6. As the skip-deletion and skip-insertion operations in Affine Alignment are geared towards noise, we expect that it performs better than Global Alignment regarding classification accuracy.

4.1.3 Methods

We applied gradient descent as discussed in Chapter 2. However, we needed to apply several normalization techniques to ensure a well-formed alignment distance: Let ∇_λ be the matrix of derivatives of the LMNN cost function with respect to the parameter matrix of the `ReplacementComparator`, that is:

$$\nabla_\lambda := \left(\frac{\partial}{\partial \lambda} \mathcal{E} \right)_{a,b \in \Sigma_\kappa \cup \{-,\_\}} \tag{4.6}$$

1. We enforced a symmetric alignment distance by preprocessing ∇_λ as follows:
$$\nabla_\lambda \leftarrow \nabla_\lambda + \nabla_\lambda^T \tag{4.7}$$

2. We set the diagonal of ∇_λ to 0, in order to enforce zero self-distance.

3. Then we applied the gradient step
$$\lambda \leftarrow \lambda - \eta \cdot \nabla_\lambda \tag{4.8}$$

4. We enforced non-negativity by setting
$$\lambda \leftarrow \lambda - \min_{a,b \in \Sigma_\kappa \cup \{-,\_\}} \lambda(a,b) \tag{4.9}$$

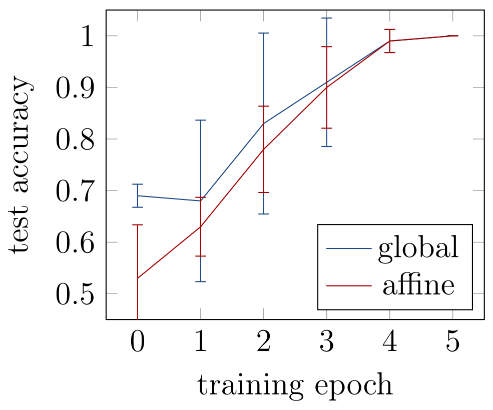
5. We enforced that the comparator function output does not exceed 1 by setting
$$\lambda \leftarrow \frac{\lambda}{\max_{a,b \in \Sigma_\kappa \cup \{-,\_\}} \lambda(a,b)} \tag{4.10}$$

As described in the previous section, we have $|X| = 60$ input sequences of length $M = 24$ for 90% of the strings. With probability $p_{\text{Noise}} = 0.1$ a noisy substring of length $N = 40$ was added. We set the k-parameter for LMNN as well as the k-NN classifier to 5. As learning rate we chose
$$\eta := \frac{10}{|X| \cdot k \cdot (M + p_{\text{Noise}} \cdot N)} \approx 0.0012 \tag{4.11}$$
We set the crispness to a moderate value of $\beta = 10$. We trained using 5 gradient steps. Results were obtained in a crossvalidation with 3 folds that we repeated 5 times to gather some statistics. We tracked the accuracy of the k-NN classifier on the test set as well as the L1 norm of the gradient
$$|\nabla_\lambda| = \sum_{a \in \Sigma_\kappa \cup \{-,\_\}} \sum_{b \in \Sigma_\kappa \cup \{-,\_\}} \left| \frac{\partial}{\partial \lambda} \mathcal{E} \right| \tag{4.12}$$
to assess convergence.

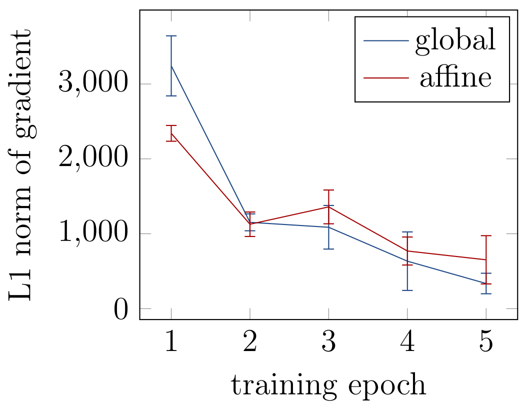
4.1.4 Results

Both alignment algorithms start out with a bad classification accuracy, with Affine Alignment around 55% and Global Alignment around 70% on average. This at least partly confirms H1. Both algorithms achieved 100% classification accuracy after the 5 training epochs in all repeats (see Figure 4.2a on the following page). This confirms H2. However, our experimental data is contrary to our expectations with respect to H6: Global alignment seems to show faster learning behavior and has a better initial classification performance. Note that the L1 norm of the gradient drops after the first learning epoch, suggesting convergence, but does not decay to zero (see Figure 4.2b on the next page).

The learned matrices for both algorithms are visualized in Figure 4.3 on the following page. In both cases the costs for replacements of `a`s and `b`s have become low, while the costs for replacements of `c`s and `d`s remain fairly high. The other replacement costs remain unchanged. This behavior is mostly consistent with our expectations in H3.



(a) The mean classification accuracy of a k-NN classifier with $k = 5$ on the test set plotted over the training epochs. The 0th training epoch is the initial test accuracy before any training was applied. The error bars show the standard deviation between the crossvalidation repeats.



(b) The mean L1 norm of the gradient plotted over the training epochs. The error bars show the standard deviation between the crossvalidation repeats.

Figure 4.2: The results of the string data experiment.

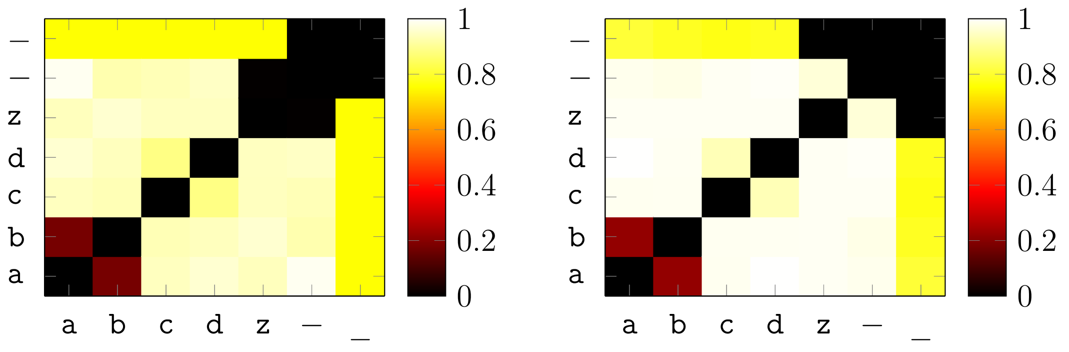


Figure 4.3: The mean parameter matrix λ after training. Dark colors in the matrix cell indicate low values, while bright colors indicate high values. The parameter matrix for Global Alignment is shown on the left side, the parameter matrix for Affine Alignment on the right. In both cases the replacement costs of `a` versus `b` have been reduced to near zero. Global alignment has reduced the deletion and insertion costs for `z` while Affine Alignment has reduced the skip-deletion and skip-insertion costs. The standard deviation of the parameter matrix between the crossvalidation repeats was below 0.1.

Even though both algorithms achieve the same classification accuracy in the end, they do so by different means: In Global Alignment the noise by z-substrings is counteracted by low deletion and insertion costs for z, while Affine Alignment uses low skip-deletion and skip-insertion and does not change the costs for deletions and insertions. This confirms our hypotheses 4 and 5.

4.1.5 Discussion

Both algorithms perform very well on the string data set, with a consistent 100% classification accuracy after training, while the initial accuracy is close to a random classifier. However, we underestimated the capabilities of Global Alignment: Making use of deletions and insertions, Global Alignment is capable to counteract noise sufficiently. Despite the high noise level it even shows better initial classification performance and faster learning behavior than Affine Alignment. We can explain this by the additional degrees of freedom that come with Affine Alignment: These make the behavior of the algorithm more difficult to predict and require longer learning times.

Furthermore, the replacement costs between cs and ds did not decay to zero. However, as this was apparently not necessary to achieve perfect classification, no imposters are left to drive the LMNN cost function. Thus, the gradient gets much weaker and learning this additional detail would require significantly longer learning time.

4.2 Time Series Data

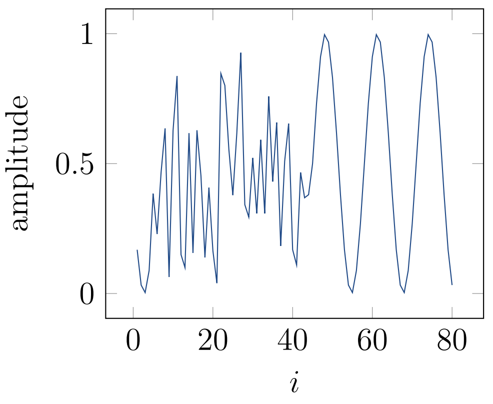
4.2.1 Data Set Description

The purpose of the time series data set is to test learning of keyword weight under conditions of noise. The task for the learning scheme is to identify one dimension of the multidimensional sequences that actually helps to distinguish between the classes, while all other dimensions do not help to do that. To that end we constructed 60 sequences (30 per class) of length 40 with 10 keywords according to the following scheme:

1. The first keyword contains the amplitude of a sine wave in the range $[0, 1]$. The frequency of the wave depends on the class. Therefore, this is the dimension that helps to distinguish between the classes. For the first class we used a period length of 13 nodes and for the second one a period length of 17 nodes. To make the task harder we introduced a random phase shift for each sequence.

2. With a probability $p_{\text{Noise}} = 0.3$ we additionally introduced noise in the first dimension by inserting a subsequence of length 40 of uniform random noise at a uniform random position of the sequence. The first dimension of two such sequences are shown in Figure 4.4 on the next page.

3. The remaining nine dimensions only contain uniform random noise in the range $[0, 1]$ independent of the class.



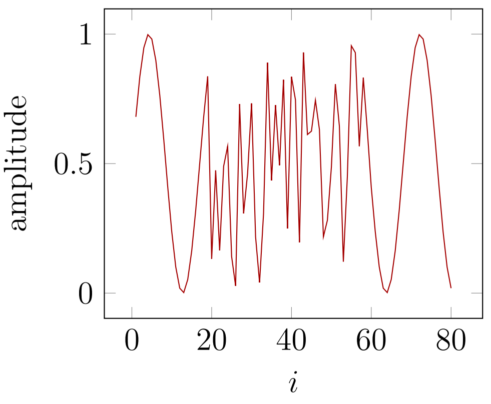


Figure 4.4: The values for the first keyword for two example sequences. The left figure shows a sequence from the first class, while the right figure shows a sequence from the second class. The values are the amplitude of a randomly phase-shifted sine wave. The frequency/period length is different depending on the class (a period length of 13 nodes for the first class and 17 nodes for the second class). Regularly these sequences have length 40. In 30% of the cases, however, we also insert 40 nodes of uniform random noise at a random position in the sequence. This is also the case in both sequences shown here.

4.2.2 Hypotheses

Regarding this data set we have the following predictions:

Hypothesis 1. Due to the very high noise level, we expect a bad initial classification accuracy of about 50% (random classifier) on the test set.

Hypothesis 2. The classification accuracy should increase for both algorithms.

Hypothesis 3. We predict that both algorithms should be able to increase the keyword weight for the first keyword to 1 while all other keyword weights decay to 0.

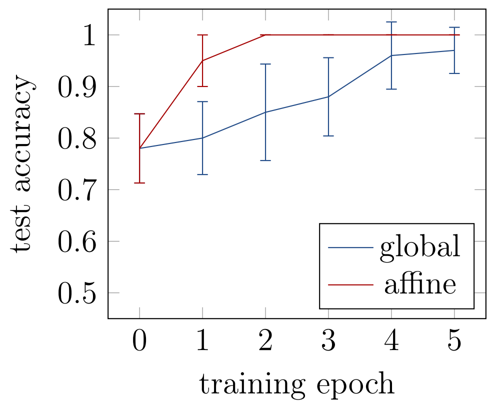
Hypothesis 4. Due to the noise in the first dimension, we expect Affine Alignment to perform better than Global Alignment, because it can skip the noisy sequence parts.

4.2.3 Methods

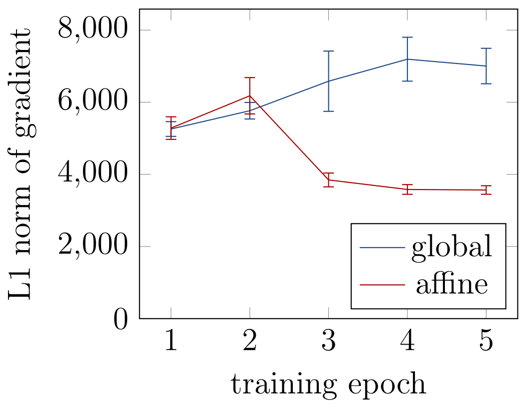
As comparator function we used an `L1NormComparator` for every keyword without applying extra normalization, as our data ensures that all comparator function distances stay in the range $[0, 1]$. We initialized the keyword weights as

$$\forall \kappa \in K : g_\kappa \leftarrow \frac{1}{|K|} \tag{4.13}$$

As in the string data set, we had to apply normalization measures in the learning scheme:



(a) The mean classification accuracy of a k-NN classifier with $k = 5$ on the test set plotted over the training epochs. The 0th training epoch is the initial test accuracy before any training was applied. The error bars show the standard deviation between the crossvalidation repeats.



(b) The mean L1 norm of the gradient plotted over the training epochs. The error bars show the standard deviation between the crossvalidation repeats.

Figure 4.5: The results of the time series data experiment.

1. After applying the gradient step, we enforced non-negative keyword weights by setting negative keyword weights to zero.
2. After that, we ensured that all keyword weights sum up to 1 by setting

$$\vec{g} \leftarrow \frac{\vec{g}}{\sum_{\kappa \in K} g_\kappa} \tag{4.14}$$

   where $\vec{g}$ is the vector of all keyword weights.

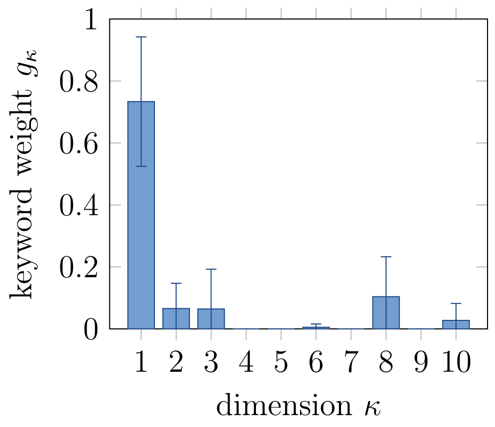
We have $|X| = 60$ input sequences of length $M = 24$ for 70% of the strings. With probability $p_{\text{Noise}} = 0.3$ we added a noisy substring of length $N = 40$. We set the k-parameter for LMNN as well as the k-NN classifier to 5. As learning rate we chose

$$\eta := \frac{2}{|X| \cdot k \cdot (M + p_{\text{Noise}} \cdot N)} \approx 1.28 \cdot 10^{-4} \tag{4.15}$$

We set the crispness to a moderate value of $\beta = 10$. We trained using 5 gradient steps. Results were obtained in a crossvalidation with 3 folds that we repeated 5 times to obtain some statistics. We tracked the classification accuracy on the test set of the k-NN classifier as well as the L1 norm of the gradient to assess convergence.

4.2.4 Results

Apparently, we once again underestimated the capabilities of alignment algorithms: Despite the high noise level, the initial classification accuracy on the



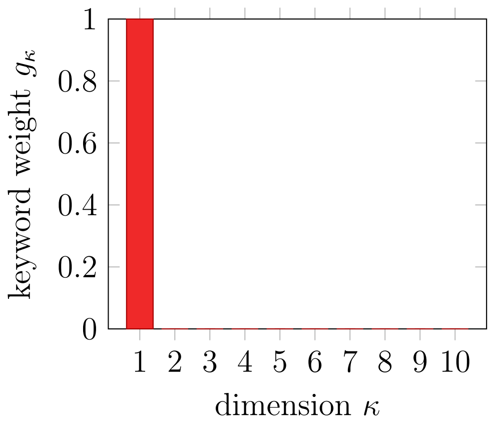


Figure 4.6: The average keyword weights after training for Global Alignment (left) and for Affine Alignment (right) respectively. The first dimension contains the sine waves of different frequencies while all other dimensions contain only random noise. Error bars mark the standard deviation between the cross-validation repititions.

test set was around 78% in both cases, contrary to H1. Our expectations were met during training, though: While training increased classification accuracy for both algorithms, confirming H2, the performance is vastly better for Affine Alignment: After two learning steps Affine Alignment achieves 100% classification accuracy consistently, while Global Alignment only arrives at 95% with 4% standard deviation (see Figure 4.5a on the preceding page), which confirms H4.

The advantage of Affine Alignment is also reflected in the dynamics of the gradient: The L1 norm of the gradient decreases fast for Affine Sequence Alignment and then stays at a low level. It does not decay entirely, because it tries to decrease the keyword weights for the noisy dimensions even further beyond zero. The L1 norm of the gradient for Global Alignment, however, shows no decreasing tendency (see Figure 4.5b on the previous page).

Finally our hypothesis 3 is confirmed as well: In both settings the first dimension was identified as the most important one by the learning procedure. However, Affine Alignment achieved absolute dominance of the first keyword weight consistently while residual keyword weights for the noisy dimensions can be observed in Global Alignment (see Figure 4.6).

4.2.5 Discussion

In this data set, Affine Alignment vastly outperforms Global Alignment, which is most likely due to the high noise level, which can not entirely be counteracted by the parameters we learn: Even with the optimal keyword weights, significant noise remains in the first dimension, which has to be skipped.

4.3 Java Programs

4.3.1 Data Set Description

Our final data set is motivated by intelligent tutoring systems (ITSs): Assume we had asked students to write a Java program that sorts an array of integers in ascending order, using either the *BubbleSort* or the *InsertionSort* algorithm. We provide pseudocode in Algorithm 7 for *BubbleSort* and in Algorithm 8 on the following page for *InsertionSort* respectively. Both algorithms are conceptionally similar, as they sort in place and start with a trivial sorted list that is iteratively extended, until the whole input list is sorted. The only difference is that *BubbleSort* takes the next element from the unsorted area and appends it to the sorted list, while *InsertionSort* takes the first unsorted element after the sorted list and inserts it at its right place in the sorted list. Both algorithms have a worst case runtime of $\mathcal{O}(N^2)$, with N being the length of the input array, and a best case runtime of $\Omega(N)$, if the input list is already sorted.[1]

Algorithm 7 A pseudocode illustration of the *BubbleSort* algorithm. The conceptual idea is to ensure that the list is sorted from index n to the end of the list. We achieve this by letting the largest element in the list from $1\ldots,n$ "bubble up" to index n. If one starts with $n = M$ and iteratively decreases n to 1, one achieves a sorted list. Note that we can stop the sorting process if one "bubbling" run does not change the list anymore.

```
Let $L$ be some (unsorted) list of integers.
Let $N$ be the length of $L$.
for $n \leftarrow N, \ldots, 2$ do
    sorted $\leftarrow$ true
    for $j \leftarrow 1, \ldots, n-1$ do
        if $L[j] > L[j+1]$ then
            Swap $L[j]$ and $L[j+1]$.
            sortiert $\leftarrow$ false
        end if
    end for
    if sorted then
        return $L$.
    end if
end for
return $L$.
```

Our aim in this fictional setting is to provide automated, example-based feedback to the students: If a student hands in a (probably erroneous) program, we would like to select an example from a database of existing programs for the same task, such that the sorting algorithm is the same, but the remaining dissimilarities between the programs might provide valuable information to the

[1]This statement about the best case runtime only holds, if the algorithms are conceptualized in the way we present them here. The easiest version of *BubbleSort* has $\Omega(N^2)$.

Algorithm 8 A pseudocode illustration of the *InsertionSort* algorithm. As with *BubbleSort*, the conceptual idea is to ensure that the list is sorted from index n to the end. But the perspective is somewhat reversed: If we start with a one-element sorted list ($n = N$), this list is sorted by definition. Then we take the element at position $n-1$ and insert it in the sorted list from $n, \dots, N$ at its right position. We can then iteratively decrease n until the whole list is sorted.

```
Let $L$ be some (unsorted) list of integers.
Let $N$ be the length of $L$.
for $n \leftarrow N, \dots, 2$ do
    for $j \leftarrow n-1, \dots, N-1$ do
        if $L[j] > L[j+1]$ then
            Swap $L[j]$ and $L[j+1]$.
        else
            Leave the inner loop directly.
        end if
    end for
end for
return $L$.
```

student to improve her own program. In this regard we follow the approaches suggested by Mokbel et al. [26] and Gross et al. [15].

Similarly to [29, 31], however, we focus on one more specific subquestion: Is it possible to distinguish between implementations of the two sorting algorithms using an alignment distance on the Java programs and a k-NN classifier? This can be seen as a necessary precondition for successful feedback provision, as feedback based on an example from a different solution strategy might be unhelpful.

This is a challenging task, considering the strong structural similarity between both algorithms. It is made even harder by the fact that programs within the same cluster might be quite dissimilar on a superficial level:

- One can implement *BubbleSort* without exploiting the fact that a sorted input list does not need to be sorted anymore.
- One can implement the algorithms in a recursive rather than iterative fashion.
- One can use sub-routines for the “bubbling” or inserting.
- One can store intermediate results or go for a compact solution with the least lines.
- One can switch the order of some statements, especially in the swapping.
- One can change the direction of the `for`-loops.
- One can use `while`-loops instead of `for`-loops.

All these differences, which we would regard as differences in programming *style* rather than differences in the overall solution *strategy*, can lead to errors in classification. In fact, all of them occur in our data set.

As a data set we used only correct implementations of both algorithms, which we took from 37 different web sites. We gathered 35 *BubbleSort* implementations and 29 *InsertionSort* implementations. 4 of the *BubbleSort*s and 2 of the *InsertionSort*s were recursive versions.

We considered the abstract abstract syntax trees (ASTs) of the programs, the nodes of which represent syntactic building blocks. The ASTs were extracted using the Oracle Java Compiler API. For each of those building blocks we considered nine features, which form our alphabets for the alignment:

1. The overall `type` of the node, e.g. a variable declaration or a `for`-loop. Overall there are 24 discrete types given by Java Compiler API in this data set.

2. The `scope` of the node, which consisted of a path-specification along the AST: If the node is at the root level of the program file, the `scope` would be empty (`"[]"`). If it is inside the second method of the class declared within this program file, the `scope` would be `"[0, 1]"`. The term `scope` is taken from the Java structure itself: Scopes are regions of visibility. Objects declared in a child scope are not visible in parent scopes. Also one does not have access the objects declared in siblings in the scope hierarchy. Note that the scope hierarchy also has a tree structure, which is an abstraction of the full-fleshed ASTs.

3. The `parent`, referring to the index of the parent node in the AST.

4. The `codePosition`, meaning the location of the code this node represents in the original Java code file. The position is encoded as a 4-tuple with the entries:

   (a) line index of the start position,

   (b) column index of the start position,

   (c) line index of the end position and

   (d) column index of the end position.

5. The `name` of the variable, method or class declared by this node. If the node is neither of those declarations, the value is `null`.

6. The `className`, which is only set for variable declarations and refers to the name of the class of that variable. Consider the following variable declaration:

   ```java
   int i = 4;
   ```

   Then the `className` would be `int`.

7. The `returnType`, which is only set for method declarations. It refers to the class of the object returned by the method.

keyword	value
type	variable
scope	"[0, 1]"
parent	16
codePosition	[12, 38, 12, 45]
name	"x"
className	"int[]"
numberOfEdges	4

Table 4.1: An example for a node in the AST of a *BubbleSort* program. This node represents a variable declaration in the second method of a class (the `scope` is `"[0, 1]"`). The index of the parent node is 16, which is the node representing the method declaration. The variable declaration can be found at line 12 from column $38 - 45$ in the code. The name of the declared variable is `x` and it is of class `int[]`. It has four edges to other nodes, one representing its array type, the other three representing assignments, where entries in the array get set to another value. Note that the other keywords, namely `returnType` and `externalDependencies`, have no value.

8. The `externalDependencies` of the node, which means a list of named entities that were referenced but could not be resolved within the program. A typical example for our sample programs is the `length` property of an array: It is declared in the Java standard library but can not be resolved to a node within the program itself.

9. The `numberOfEdges`, which refers to the number of children of this node in the AST, as well as additional connections it might have within the AST. For example, we enrich the AST by additional references encoding the usage of variables. Consider the following code snippet:

   ```
           int i;
           i = 4;
   ```

   Then the node representing the variable declaration of `i` has a connection to the node representing the assignment of `4` to `i` and the other way around. Thus the `numberOfEdges` feature represents the degree of connectedness of this node in the program.

We show an example for such a node from one *BubbleSort* program in Table 4.1.

4.3.2 Hypotheses

The aim of this experiment is to learn the keyword weights of the nine features presented above. Regarding this data set we have the following hypotheses:

Hypothesis 1. We expect an overall positive effect of learning regarding test classification accuracy for both algorithms.

Hypothesis 2. We expect that our experiment roughly reproduces the weighting achieved in [29, 31], that is: We expect zero keyword weights for `codePosition`, `parent` and `numberOfEdges`, high keyword weights for `type` and `scope` and intermediate keyword weights for the remaining keywords.

Hypothesis 3. We expect that, similar to the findings in [31], Affine Alignment outperforms Global Alignment with respect to classification accuracy.

4.3.3 Methods

In order to transform the AST into a sequence we used prefix notation as in [29, 31]. Further, we used the same comparator functions as in [31], namely:

1. For `type` we used a `ReplacementComparator` with the default parameters as in the string experiment (see Section 4.1 on page 68).

2. For `scope` we used a custom comparator function that returns 1 minus the relative length of the longest shared prefix of both input strings. For example, consider the two `scope` descriptors `"[0, 1, 0]"` and `"[0, 1, 2, 0]"`. They have the common prefix `"[0, 1, "`, which is 7 characters long. The longer input string is `"[0, 1, 2, 0]"` with a length of 12 characters. So the returned value is

$$\begin{aligned} & c_{\text{rep}}^{\texttt{scope}}(\texttt{"[0, 1, 0]"}, \texttt{"[0, 1, 2, 0]"}) \\ & := 1 - \frac{|\texttt{"[0, 1, "}|}{\max\{|\texttt{"[0, 1, 0]"}|, |\texttt{"[0, 1, 2, 0]"}|\}} \\ & = 1 - \frac{7}{12} \approx 0,583 \end{aligned} \quad (4.16)$$

3. For `parent` and `numberOfEdges` we applied a `L1NormComparator` with a `HyperbolicNormalizer` with $\alpha = 1$.

4. For `codePosition` we estimated the distance of two code positions as the character distance between both start positions. We estimated the average line length by 80. Consider the two `codePosition` descriptors `[12, 35, 12, 38]` and `[15, 2, 17, 6]`. Then the character distance would be

$$|12 \cdot 80 + 35 - (15 \cdot 80 + 2)| = 207 \quad (4.17)$$

   We apply a `HyperbolicNormalizer` afterwards with $\alpha = 880$, meaning that a moderate code distance of about 10 lines leads to a cost of 0.5.

5. For the remaining keywords we applied a plain `CharStatComparator`.

As normalization of the gradient we applied the same techniques as in the time series data set. We also used the same initial keyword weights, namely:

$$\forall \kappa \in K : g_\kappa \leftarrow \frac{1}{|K|} \quad (4.18)$$

We have $|X| = 64$ input sequences with an average length of $\hat{M} = 95$. We set the k-parameter for LMNN as well as the k-NN classifier to 5. For the learning rate we chose

$$\eta := \frac{1}{|X| \cdot k \cdot \hat{M}} \approx 3.29 \cdot 10^{-5} \tag{4.19}$$

Similar to our previous work, we set the crispness to a higher value of $\beta = 100$ to boost performance. We trained using 5 gradient steps. Results were obtained in a crossvalidation with 4 folds that we repeated 5 times to obtain some statistics. We tracked the classification accuracy on the test set of the k-NN classifier as well as the L1 norm of the gradient to assess convergence.

We note that the scaling of the different comparator functions is not equal, that is: The average returned values for this data set were not equal. This relates to the interpretation problem described in Section 2.5.5 on page 40: The learning has to counteract the scaling effects as well as emphasize discriminative dimensions using the keyword weights. What we want to interpret, however, is only the emphasis the learning puts on the single dimensions with respect to class discrimination. In order to do that, we apply a post-processing step after learning: We multiply the keyword weights with the average value returned by the respective comparator function. To obtain this average, we calculate the optimal alignments for all pairs of sequences in the data set after training, add all contributions by the respective comparator function in all alignments, and divide by the respective alignment length as well as the number of alignments. After that, we renormalize the postprocessed keyword weights to a sum of 1.

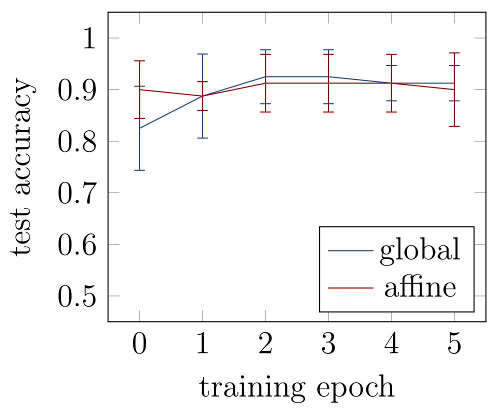
4.3.4 Results

For both algorithms the test classification accuracy starts at a fairly high level (82.5% for Global and 90% for Affine Alignment). While we can observe an increase for Global Alignment during training, this is not the case for Affine Alignment, only partly confirming H1. In our experimental data we do not see any significant difference between Global and Affine Alignment regarding the classification accuracy after training, which speaks against hypothesis 3. We do find a consistently lower L1 norm of the gradient for Affine Alignment, however, suggesting less need for learning (see Figure 4.7b on the facing page).

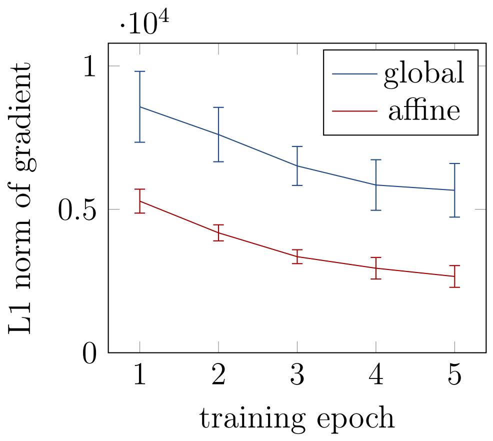
However, we do see the expected keyword weight profile of hypothesis 2: For both algorithms the learning puts special emphasis on `type` and `scope` while the keyword weights for `codePosition` and `parent` decay to zero. Interestingly, our prediction for `numberOfEdges` was incorrect: Contrary to our expectations the learning rather emphasized this keyword.

4.3.5 Discussion

We were not able to reproduce the results of Paaßen, Mokbel, and Hammer [31] stating that Affine Alignment outperforms Global Alignment on this data

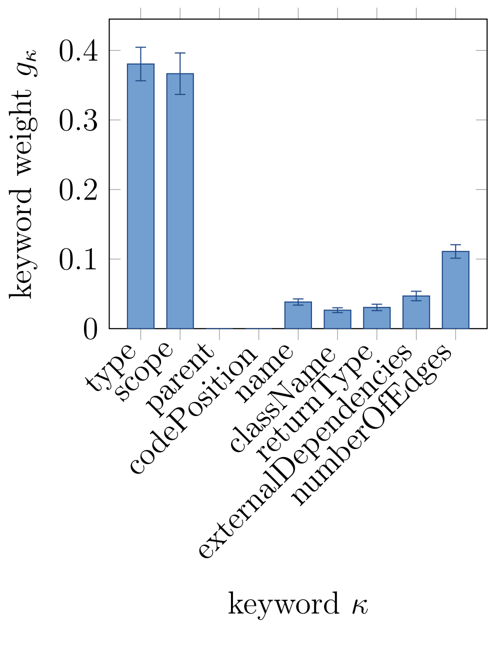


(a) The mean classification accuracy of a k-NN classifier with $k = 5$ on the test set plotted over the training epochs. The 0th training epoch is the initial test accuracy before any training was applied. The error bars show the standard deviation between the crossvalidation repeats.



(b) The mean L1 norm of the gradient plotted over the training epochs. The error bars show the standard deviation between the crossvalidation repeats.

Figure 4.7: The results of the Java program experiment.



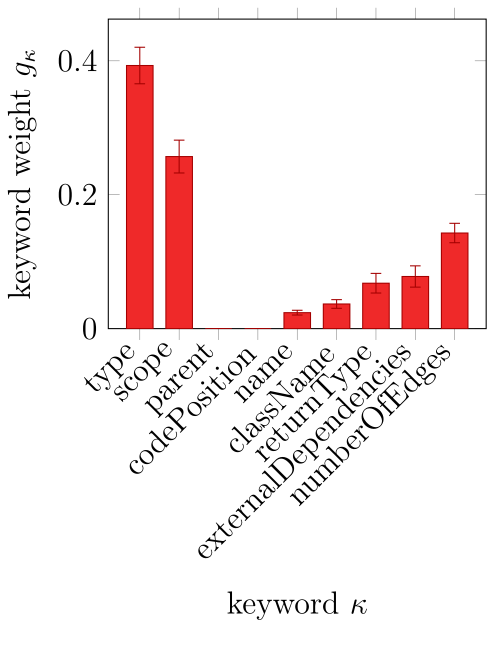


Figure 4.8: The average keyword weights after training for Global Alignment (left) and for Affine Alignment (right) respectively. Error bars mark the standard deviation between the crossvalidation repititions. Note that these keyword weights have been postprocessed to be interpretable as relevance terms (see Section 4.3.3 on page 81).

set, which might be connected to the fact that we also find a slightly different keyword weight profile: In particular our learning scheme in this thesis emphasized the `numberOfEdges` keyword, while the very same keyword was suppressed when learning with the RGLVQ framework. From a semantic perspective we are inclined to agree with the former strategy, as this particular feature does highlight nodes with many connections, which is an important structural cue. However, this feature is certainly highly correlated with other features (e.g. the `type: variable` declarations are much more likely to be highly connected than `if` statements), which is why it is plausible that the remaining keywords are sufficient to properly distinguish between the two classes.

Still, the classification accuracy on this challenging task is impressive and outperforms even RGLVQ. Also, we could reproduce most of the keyword weight profile, which still remains semantically sound.

Chapter 5

Conclusion

Our work in Chapter 2 provides a theoretical framework for alignment distance learning based on Algebraic Dynamic Programming (ADP). We have shown that alignments can be viewed as trees in the language of a regular tree grammar and that the alignment distance is just the application of an algebra and a choice function on this language. Furthermore, we have shown how one can compute the alignment distance generically in $\mathcal{O}(M \cdot N)$ using dynamic programming, given some grammar, algebra, choice function and two input sequences. These are all direct conclusions from the brilliant work on ADP by Giegerich, Meyer, and Steffen [12].

We have shown that, by enforcing certain conditions on grammar, algebra and choice function, the resulting alignment distance fulfills most conditions of a metric (see Section 2.5 on page 32), which enables a pseudo-euclidian embedding of the data set [32].

We have further extended the work on ADP by calculating a gradient on the alignment distance, which is equivalent to applying an alternative algebra and choice function on the same grammar (see Section 2.7 on page 42). To make the alignment distance differentiable we used the soft minimum approximation, for which we also have provided estimations of the approximation error (see Section 2.8 on page 46).

We provide an extensible Java implementation of this work under a free software license in form of the *TCS Alignment Toolbox*[1] (see Chapter 3). This implementation also comes with means for parallel computation of alignment distances and gradients thereof.

After first work on gradient-based alignment distance learning based on the Relational Generalized Learning Vector Quantization (RGLVQ) framework [27, 28, 29, 31], we have extended this approach to the realm of the popular k-Nearest Neighbor (k-NN) classifier by plugging the alignment distance gradient into the Large Margin Nearest Neighbor (LMNN) cost function introduced by Weinberger and Saul [43].

In our experiments in Chapter 4 we have shown that, using Affine Sequence Alignment, we can infer a helpful scoring scheme for the alignment distance even under conditions of heavy noise, which a global alignment algorithm can

[1] `http://openresearch.cit-ec.de/projects/tcs`

not achieve to the same degree. This is not only the case for artificial data sets but for a real-world example of Java programs as well.

Overall we have demonstrated that gradient-based alignment distance learning is an effective tool from multiple perspectives:

- Using the theory presented in this work one can easily extend the learning techniques to other alignment schemes, by plugging in other grammars.
- As we have shown in the Java programming experiment, the approach works even for very rich sequential datasets, wich are not only multidimensional but also multimodal, by defining proper comparator functions over the single alphabets.
- For many (if not all) sequential data sets the definition of comparator functions over the features contained in single nodes is more intuitive and straightforward than trying to find a feature representation of the sequence as a whole. Even more, one does not have to abstract from the structural richness of the underlying data. Here we follow the argument of Pękalska [32]: Viewing data points in terms of their relations to each other opens up a valuable new perspective.
- As we have shown, the gradient over the alignment distance can be effectively calculated with the same asymptotic efficiency as the alignment distance itself. Furthermore, it can be understood as just an application of a different choice function and algebra on the same grammar. This also provides a nice guideline for implementations. Further work on approximations and speedup techniques is provided in [29].
- The gradient of the alignment distance can be plugged into arbitrary cost functions, and indeed we have shown that the approach does work not only in the RGLVQ framework but for k-NN classifiers as well.
- Using the right cost function, one can directly optimize the underlying alignment distance for better classification performance, as we have done here and in our previous work.
- Even though the cost function we used in this work is only partly differentiable and non-convex, such that local optima instead of global ones are probable, we achieved consistent and satisfying learning results[2].

Further work in the area is of course possible, for example:

- It could be possible to describe existing kernel approaches like the ones presented by Saigo, Vert, and Akutsu [34] and Cuturi et al. [10] in terms of ADP algebras and choice functions. Based on that one could easily construct kernels for more complex alignment approaches, like Affine Alignment, and extend the gradient-based learning approach to them.

[2]some further notes on that topic in the context of the RGLVQ framework are provided in [29]

- One could extend the learning scheme to other cost functions, e.g. to optimize other classifiers, or to an unsupervised learning task like clustering.
- One could apply more sophisticated nonlinear optimization techniques than a simple gradient-descend.
- We have omitted a proof for the triangular inequality of the alignment distance. It should be possible to provide this proof if one is able to connect the class of ADP grammars we presented here to edit distances (see e.g. Heun [18]).

However, even without these further extensions, we have provided strong arguments for an extended use of gradient-based alignment distance learning in the research community and in broader application domains.

5.1 Acknowledgement

This thesis is largely based on research within the DFG priority programme *Autonomous Learning*, more specifically the research project *Learning Feedback for Intelligent Tutoring Systems (FIT)*. Funding for this project by the DFG under grant number HA 2719/6-1 is gratefully acknowledged.

Bibliography

[1] Aurélien Bellet, Amaury Habrard, and Marc Sebban. "A Survey on Metric Learning for Feature Vectors and Structured Data". In: *ArXiv e-prints* (2013). URL: `http://arxiv.org/abs/1306.6709`.

[2] Aurélien Bellet, Amaury Habrard, and Marc Sebban. "Good edit similarity learning by loss minimization". English. In: *Machine Learning* 89.1-2 (2012), pp. 5–35. DOI: `10.1007/s10994-012-5293-8`.

[3] Richard Bellman. *Dynamic Programming.* 1st ed. Princeton, NJ, USA: Princeton University Press, 1957.

[4] Yoshua Bengio, Aaron C. Courville, and Pascal Vincent. "Unsupervised Feature Learning and Deep Learning: A Review and New Perspectives". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35.8 (2013), pp. 1798–1828.

[5] Marc Bernard, Laurent Boyer, Amaury Habrard, and Marc Sebban. "Learning probabilistic models of tree edit distance". In: *Pattern Recognition* 41.8 (2008), pp. 2611–2629. DOI: `10.1016/j.patcog.2008.01.011`.

[6] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics).* Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006. ISBN: 0387310738.

[7] Walter S. Brainerd. "Tree generating regular systems". In: *Information and Control* 14.2 (1969), pp. 217–231.

[8] Lei Chen and Raymond Ng. "On the Marriage of Lp-norms and Edit Distance". In: *Proceedings of the Thirtieth International Conference on Very Large Data Bases (VLDB).* Vol. 30. Toronto, Canada: VLDB Endowment, 2004, pp. 792–803.

[9] Thomas M. Cover and Peter E. Hart. "Nearest neighbor pattern classification". In: *IEEE Transactions on Information Theory* 13.1 (1967), pp. 21–27. DOI: `10.1109/TIT.1967.1053964`.

[10] Marco Cuturi, J Vert, Oystein Birkenes, and Tomoko Matsui. "A kernel for time series based on global alignments". In: *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP).* Vol. 2. IEEE. 2007, pp. II–413.

[11] Fred J. Damerau. "A Technique for Computer Detection and Correction of Spelling Errors". In: *Communications of the ACM* 7.3 (Mar. 1964), pp. 171–176.

[12] Robert Giegerich, Carsten Meyer, and Peter Steffen. “A discipline of dynamic programming over sequence data”. In: *Science of Computer Programming* 51.3 (2004), pp. 215–263.

[13] Robert Giegerich and Karl Schmal. “Code selection techniques: Pattern matching, tree parsing, and inversion of derivors”. English. In: *European Symposium on Programming (ESOP)*. Ed. by H. Ganzinger. Vol. 300. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1988, pp. 247–268.

[14] Osamu Gotoh. “An improved algorithm for matching biological sequences”. In: *Journal of molecular biology* 162.3 (1982), pp. 705–708.

[15] Sebastian Gross, Bassam Mokbel, Barbara Hammer, and Niels Pinkwart. “How to Select an Example? A Comparison of Selection Strategies in Example-Based Learning”. In: *Intelligent Tutoring Systems (ITS)*. Ed. by Stefan Trausan-Matu, Kristy Elizabeth Boyer, Martha Crosby, and Kitty Panourgia. Vol. 8474. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 340–347. DOI: `10.1007/978-3-319-07221-0_42`.

[16] Amaury Habrard, José Manuel Iñesta, David Rizo, and Marc Sebban. “Melody recognition with learned edit distances”. In: Lecture Notes in Computer Science 5342 LNCS (2008), pp. 86–96. DOI: `10.1007/978-3-540-89689-0_13`.

[17] Barbara Hammer, Daniela Hofmann, Frank-Michael Schleif, and Xibin Zhu. “Learning vector quantization for (dis-)similarities”. English. In: *Neurocomputing* 131 (2014), pp. 43–51. DOI: `10.1016/j.neucom.2013.05.054`.

[18] Volker Heun. *Skriptum zur Vorlesung Algorithmische Bioinformatik I & II*. 5.100. Munich: Ludwig-Maximilians-University, Jan. 27, 2015. URL: `http://www.bio.ifi.lmu.de/~heun/lecturenotes/` (visited on 03/23/2015).

[19] Thomas Hofmann, Bernhard Schölkopf, and Alexander J. Smola. “Kernel methods in machine learning”. In: *The annals of statistics* (2008), pp. 1171–1220.

[20] Stefan Janssen and Robert Giegerich. “The RNA shapes studio”. In: *Bioinformatics* 31.3 (2015), pp. 423–425. DOI: `10.1093/bioinformatics/btu649`.

[21] Brian Kulis. “Metric Learning: A Survey”. In: *Foundations and Trends in Machine Learning* 5.4 (2013), pp. 287–364. DOI: `10.1561/2200000019`.

[22] Vladimir I. Levenshtein. “Binary codes capable of correcting deletions, insertions, and reversals”. In: *Soviet Physics Doklady* 10.8 (1965), pp. 707–710.

[23] Ming Li, Xin Chen, Xin Li, Bin Ma, and P.M.B. Vitanyi. “The similarity metric”. In: *IEEE Transactions on Information Theory* 50.12 (2004), pp. 3250–3264. DOI: `10.1109/TIT.2004.838101`.

[24] Stuart P. Lloyd. “Least squares quantization in PCM”. In: *IEEE Transactions on Information Theory* 28.2 (1982), pp. 129–136. DOI: `10.1109/TIT.1982.1056489`.

[25] Huma Lodhi, Craig Saunders, John Shawe-Taylor, Nello Cristianini, and Chris Watkins. “Text classification using string kernels”. In: *The Journal of Machine Learning Research (JMLR)* 2 (2002), pp. 419–444.

[26] Bassam Mokbel, Sebastian Gross, Benjamin Paaßen, Niels Pinkwart, and Barbara Hammer. “Domain-Independent Proximity Measures in Intelligent Tutoring Systems”. In: *Educational Datamining (EDM)*. 2013, pp. 334–335.

[27] Bassam Mokbel, Benjamin Paassen, and Barbara Hammer. “Adaptive distance measures for sequential data”. English. In: *European Symposium on Artificial Neural Networks (ESANN)*. Ed. by Michel Verleysen. Bruges, Belgium: i6doc.com, 2014, pp. 265–270.

[28] Bassam Mokbel, Benjamin Paassen, and Barbara Hammer. “Efficient Adaptation of Structure Metrics in Prototype-Based Classification”. English. In: *International Conference on Artificial Neural Networks and Machine Learning (ICANN)*. Vol. 8681. Lecture Notes in Computer Science. Springer, 2014, pp. 571–578. DOI: `10.1007/978-3-319-11179-7_72`.

[29] Bassam Mokbel, Benjamin Paassen, Frank-Michael Schleif, and Barbara Hammer. “Metric learning for sequences in relational LVQ”. English. In: *Neurocomputing* (2015). (accepted/in press).

[30] Saul B. Needleman and Christian D. Wunsch. “A general method applicable to the search for similarities in the amino acid sequence of two proteins”. In: *Journal of Molecular Biology* 48.3 (1970), pp. 443–453.

[31] Benjamin Paaßen, Bassam Mokbel, and Barbara Hammer. “Adaptive structure metrics for automated feedback provision in Java programming”. English. In: *European Symposium on Artificial Neural Networks (ESANN)*. Ed. by Michel Verleysen. (accepted/in press). Bruges, Belgium, 2015.

[32] Elżbieta Pękalska. “The dissimilarity representation for pattern recognition: foundations and applications”. PhD thesis. Delft University of Technology, 2005.

[33] Jens Reeder, Peter Steffen, and Robert Giegerich. “pknotsRG: RNA pseudoknot folding including near-optimal structures and sliding windows”. English. In: *Nucleids Acid Research* 35 (2007), W320–W324. DOI: `10.1093/nar/gkm258`.

[34] Hiroto Saigo, Jean-Philippe Vert, and Tatsuya Akutsu. “Optimizing amino acid substitution matrices with a local alignment kernel”. In: *BMC bioinformatics* 7.1 (2006), p. 246.

[35] G. Salton, A. Wong, and C. S. Yang. “A Vector Space Model for Automatic Indexing”. In: *Communications of the ACM* 18.11 (1975), pp. 613–620. DOI: `10.1145/361219.361220`.

[36] Gerard Salton and Christopher Buckley. “Term-weighting approaches in automatic text retrieval”. In: *Information processing & management* 24.5 (1988), pp. 513–523.

[37] Georg Sauthoff. “Bellman’s GAP: A 2nd Generation Language and System for Algebraic Dynamic Programming”. PhD thesis. Bielefeld University, 2011.

[38] Temple F. Smith and Michael S. Waterman. “Identification of common molecular subsequences”. In: *Journal of Molecular Biology* 147.1 (1981), pp. 195–197.

[39] A. Takasu, D. Fukagawa, and T. Akutsu. “Statistical learning algorithm for tree similarity”. In: *IEEE International Conference on Data Mining (ICDM).* 2007, pp. 667–672.

[40] Vladimir Vapnik, Steven E. Golowich, and Alexander J. Smola. “Support vector method for function approximation, regression estimation, and signal processing”. In: *Advances in neural information processing systems (NIPS)* (1997), pp. 281–287.

[41] T.K. Vintsyuk. “Speech discrimination by dynamic programming”. English. In: *Cybernetics* 4.1 (1968), pp. 52–57.

[42] Michael S. Waterman, Temple F. Smith, and William A. Beyer. “Some biological sequence metrics”. In: *Advances in Mathematics* 20.3 (1976), pp. 367–387.

[43] Kilian Q. Weinberger and Lawrence K. Saul. “Distance Metric Learning for Large Margin Nearest Neighbor Classification”. In: *Journal of Machine Learning Research* 10 (2009), pp. 207–244. URL: `http://dl.acm.org/citation.cfm?id=1577069.1577078`.

[44] Wikipedia. *Maxwell-Boltzmann-Statistics.* URL: `http://en.wikipedia.org/wiki/Maxwell%E2%80%93Boltzmann_statistics` (visited on 03/20/2015).

Acronyms

***k*-NN** *k*-Nearest Neighbor. 3, 9, 10, 13, 14, 40, 42, 67, 71, 72, 75, 78, 82, 83, 85, 86, 96, 97

ADP Algebraic Dynamic Programming. 3, 10, 12, 14–18, 20, 22–30, 32, 50, 51, 58, 60, 85–87

AST abstract syntax tree. 79–81

DTW Dynamic Time Warping. 10, 23, 34, 52, 64

GAP-L Bellman's Gap Language. 20–22

ITS intelligent tutoring system. 77

LMNN Large Margin Nearest Neighbor. 3, 14, 40, 71, 75, 82, 85, 96

NCD Normalized Compression Distance. 54

RGLVQ Relational Generalized Learning Vector Quantization. 10, 13, 34, 69, 84–86

Glossary

algebra ($\mathcal{F}$) a set of comparator functions, one for each keyword and one for each alignment operation in the respective signature; see Definition 11 on page 19. 12, 16, 17, 19, 20, 22–26, 28, 30, 32, 34–40, 43, 44, 50, 51, 55, 57, 64, 85, 86, 100, 101, 113

algebra function (d) a function calculating the cost for one alignment operation; see Definition 12 on page 19. 19, 32, 34, 36, 39, 95

algebra function input set ($\mathcal{I}(t)$) the input set for an algebra function according to function template t; see Definition 8 on page 18. 18, 52

alignment distance (D) the alignment distance between two input sequences $\bar{x}$ and $\bar{y}$; see Definition 21 on page 24. 3, 10–15, 17, 21, 24, 25, 31, 32, 35–38, 40, 42, 43, 57, 62–64, 66–70, 78, 85–87

alignment end (nil) empty end operator. 17, 18

alphabet (Σ_κ) the set of possible values for keyword κ; see Definition 1 on page 15. 15, 16, 18, 20, 22, 52, 55, 68, 79, 86, 95–97, 113

arity ($(\mathcal{A}, \mathcal{A}')$) a tuple of two natural numbers specifying the number of input arguments of an algebra function or a comparator function on the left and on the right side. Within this thesis the arity just specifies whether some function template takes inputs from the left sequence only, the right sequence only or both sequences; see Definition 5 on page 17. 17, 18, 21, 33, 96

axiom (A^*) the start nonterminal symbol (or axiom) of a grammar. 20, 22, 30, 44, 45

choice function (h) a function calculating one (optimum) value from a set of input values; also called *objective function*; see Definition 13 on page 19. 19, 20, 22, 24, 25, 28, 30, 32, 34, 35, 37–40, 43–46, 49, 51, 52, 62, 63, 85, 86, 113, 124

comparator function (c^κ) a function calculating the distance between two values from the alphabet for keyword κ; see Definition 10 on page 19. 19, 40, 43, 45, 52, 54–57, 65, 68, 71, 74, 81, 82, 86, 95, 96, 124

comparator function input set ($\mathcal{I}_c(t)$) the input set for a comparator function according to function template t; see Definition 9 on page 19. 19

crispness (β) A parameter to tune the soft minimum approximation: For a β of 0 softmin is equal to the arithmetic average. For $\beta \to \infty$ softmin approaches min. 46, 56, 71, 75, 82

data set (X) the set of all sequences. 66, 82, 85

deletion (del) deleting one node in the first sequence. 17, 18, 26, 28, 31, 37, 54–56, 61, 69, 72, 73, 116

function template (t) a tuple of a name and an arity; see Definition 7 on page 17. 16–19, 21, 33, 44, 45, 55, 65, 95

grammar ($\mathcal{G}$) a regular tree grammar that expresses the language of all possible alignments; see Definition 14 on page 20. 12, 16, 17, 20–26, 28, 30–40, 43, 44, 50–52, 58–61, 63, 64, 85–87, 95, 96, 100, 105, 113

imposters ($\mathfrak{I}(\bar{x}, \bar{y})$) The set of data points from the other class, which are closer to $\bar{x}$ than $\bar{y}$; see Definition 28 on page 41. 41, 42, 73

insertion (ins) inserting one node from the second sequence into the first sequence. 17, 18, 26, 28, 37, 54–56, 61, 69, 72, 73, 116

keyword (κ) a keyword; see Definition 2 on page 16. 16, 18–20, 22, 24, 36, 40, 43, 44, 52, 57, 64, 68, 73, 74, 80–84, 95–97, 113

keyword set (K) the set of all keywords; see Definition 2 on page 16. 16, 18, 20

keyword weight (g_κ) the weight (or relevance) of keyword κ; see Definition 12 on page 19. 19, 34, 40, 44, 45, 52, 57, 65, 66, 73–76, 80–84, 96

language ($\mathcal{L}(\mathcal{G})$) the language defined by the grammar $\mathcal{G}$ which means the set of all trees that can be constructed using the grammar $\mathcal{G}$. 22, 35, 85

length (M, N) the number of nodes in sequence $\bar{x}$ and $\bar{y}$ respectively ($M := |\bar{x}|$, $N := |\bar{y}|$). 11, 16, 24, 25, 28, 30, 32, 45

LMNN cost function ($\mathcal{E}$) The Large Margin Nearest Neighbor cost function; see Definition 29 on page 41. 41–43, 67, 70, 73

margin (γ) The minimum margin we enforce between data points from different classes in the Large Margin Nearest Neighbor metric learning scheme. 41, 42

node (x_i, y_j) the ith node in sequence $\bar{x}$ or the jth node in sequence $\bar{y}$, in general a (multimodal) vector; see Definition 3 on page 16. 16, 20, 24, 36, 53, 61, 73, 74, 79, 80, 84, 86, 96, 97

node set ($\Sigma_\times$) the set of possible nodes, equivalent to the Cartesian product of all alphabets; see Definition 3 on page 16. 16, 20, 24, 51–53

nonterminal symbol (A, B) nonterminal symbol in a grammar. 20–23, 26, 28, 30, 32, 34, 37, 39, 49, 58–62, 99, 102–104, 113–116, 118–122

number of neighbors (k) The number of neighbors in a k-NN algorithm. 40

parameter (λ) some parameter of the the metric that shall be optimized. In the context of this work either a specific comparator function result for two input values a and b denoted as $\lambda(a, b)$ or some keyword weight g. 3, 9, 13–15, 40, 42, 43, 45, 46, 65–70, 72, 76, 81

replacement (rep) replacing one node in the first sequence by one node of the second sequence. 17, 18, 23, 26, 31, 35, 36, 55, 56, 61, 68–73, 116, 119

sequence ($\bar{x}$, $\bar{y}$) a sequence of nodes; see Definition 4 on page 16. 9–12, 15–19, 21–26, 28, 30–35, 37–40, 45, 50–54, 61, 63, 67, 68, 71, 73–75, 81, 82, 85, 86, 95–97, 113, 124

signature ($\mathcal{T}$) a set of permitted alignment operations; see Definition 7 on page 17. 16–18, 20, 21, 24, 25, 28, 30, 35–39, 51, 95
skip-deletion (skip_del) skipping an irrelevant node in the first sequence. 17, 18, 26, 28, 37, 54–56, 69, 70, 72, 73, 117
skip-insertion (skip_ins) skipping an irrelevant node in the second sequence. 17, 18, 26, 28, 37, 54–56, 69, 70, 72, 73
soft minimum (softmin) the soft approximation of the strict minimum function; see Definition 30 on page 46. 14, 15, 46–50, 52, 56, 85, 95, 109, 110

target neighbors ($\mathfrak{N}_k(\bar{x})$) The k-NNs of $\bar{x}$ in the same class; see Definition 27 on page 40. 40–42

value (x^κ, a, b) the value for keyword κ in node x or some arbitrary value from the alphabet Σ_κ; see Definition 1 on page 15. 15, 16, 19, 36, 53, 55, 56, 74, 79, 80, 96, 97

yield ($\mathcal{Y}(T)$) the concatenation of all terminal symbols that are leafs of tree T; see Definition 19 on page 24. 24, 25, 28, 35, 38, 49

Appendix A

Proofs

A.1 Grammar Symmetry

A.1.1 Strong Symmetry

Have a look at Theorem 7 on page 37 for definitions and claim. We do a proof by structural induction. Our base case are all nonterminal symbols A $\in \Phi$ for which at least one production rule of the following form exists:

```
A = nil(<EMPTY,EMPTY>);
```

For those we obtain:

$$A_{\bar{x},\bar{y}}(M+1,N+1) = 0 = A_{\bar{y},\bar{x}}(N+1,M+1) \tag{A.1}$$

Now let A be a nonterminal symbol from Φ and $i \in \{1,\dots,M+1\}, j \in \{1,\dots,N+1\}$. Our induction hypothesis is that our claim holds for all entries of the dynamic programming tables, which we need to calculate $A(i,j)$, according to Algorithm 2. In other words: For every entry $B(i',j')$, that is required to calculate $A_{\bar{x},\bar{y}}(M+1,N+1)$, it holds:

$$B_{\bar{x},\bar{y}}(i',j') = B_{\bar{y},\bar{x}}(j',i') \tag{A.2}$$

Using the equation for dynamic programming table entries shown in Algorithm 2 on page 30, we obtain:

$$A_{\bar{x},\bar{y}}(i,j) = h[\theta_l^{\bar{x},\bar{y}}]_{l=1\dots L} \tag{A.3}$$

$$A_{\bar{y},\bar{x}}(j,i) = h[\theta_l^{\bar{y},\bar{x}}]_{l=1\dots L} \tag{A.4}$$

Obviously, the result of h has to be equal if the multiset of input arguments is equal. Now consider the different possible production rules that might correspond to $\theta_l^{\bar{x},\bar{y}}$:

1.
   ```
   A = B;
   ```

   This production rule can be applied for $A_{\bar{y},\bar{x}}(j,i)$ as well. Due to our induction hypothesis, we further know that

   $$B_{\bar{x},\bar{y}}(i,j) = B_{\bar{y},\bar{x}}(j,i) \tag{A.5}$$

   Thus an equal term $\theta_{l'}^{\bar{y},\bar{x}}$ has to exist for some l'.

2.
```
    A = del(<NODE, EMPTY>, B);
```

   Due to strong grammar symmetry, we know that a production rule of the form

```
    A = ins(<EMPTY, NODE>, B);
```

   exists as well. Due to our induction hypothesis, we know that

$$B_{\bar{x},\bar{y}}(i+1,j) = B_{\bar{y},\bar{x}}(j,i+1) \tag{A.6}$$

   Finally, we can infer using algebra symmetry:

$$d_{\text{del}}(x_i, B_{\bar{x},\bar{y}}(i+1,j)) = d_{\text{ins}}(B_{\bar{y},\bar{x}}(j,i+1), x_i) \tag{A.7}$$

   Thus an equal term $\theta_{l'}^{\bar{y},\bar{x}}$ has to exist for some l'.

3.
```
    A = skip_del(<NODE, EMPTY>, B);
```

   Due to strong grammar symmetry, we know that a production rule of the form

```
    A = skip_ins(<EMPTY, NODE>, B);
```

   exists as well. Due to our induction hypothesis, we know that

$$B_{\bar{x},\bar{y}}(i+1,j) = B_{\bar{y},\bar{x}}(j,i+1) \tag{A.8}$$

   Finally, we can infer using algebra symmetry:

$$d_{\text{skip\_del}}(x_i, B_{\bar{x},\bar{y}}(i+1,j)) = d_{\text{skip\_ins}}(B_{\bar{y},\bar{x}}(j,i+1), x_i) \tag{A.9}$$

   Thus an equal term $\theta_{l'}^{\bar{y},\bar{x}}$ has to exist for some l'.

4.
```
    A = ins(<EMPTY, NODE>, B);
```

   Due to strong grammar symmetry, we know that a production rule of the form

```
    A = del(<NODE, EMPTY>, B);
```

   exists as well. Due to our induction hypothesis, we know that

$$B_{\bar{x},\bar{y}}(i,j+1) = B_{\bar{y},\bar{x}}(j+1,i) \tag{A.10}$$

   Finally, we can infer using algebra symmetry:

$$d_{\text{ins}}(B_{\bar{x},\bar{y}}(i,j+1), y_j) = d_{\text{del}}(y_j, B_{\bar{y},\bar{x}}(j+1,i)) \tag{A.11}$$

   Thus an equal term $\theta_{l'}^{\bar{y},\bar{x}}$ has to exist for some l'.

5.
```
    A = skip_ins(<EMPTY, NODE>, B);
```

   Due to strong grammar symmetry, we know that a production rule of the form

```
    A = skip_del(<NODE, EMPTY>, B);
```

exists as well. Due to our induction hypothesis, we know that

$$B_{\bar{x},\bar{y}}(i, j+1) = B_{\bar{y},\bar{x}}(j+1, i) \tag{A.12}$$

Finally, we can infer using algebra symmetry:

$$d_{\text{skip\_ins}}(B_{\bar{x},\bar{y}}(i, j+1), y_j) = d_{\text{skip\_del}}(y_j, B_{\bar{y},\bar{x}}(j+1, i)) \tag{A.13}$$

Thus an equal term $\theta_{l'}^{\bar{y},\bar{x}}$ has to exist for some l'.

6.
```
    A = rep(<NODE, NODE>, B);
```

In that case the same production can be applied for $A_{\bar{y},\bar{x}}(j, i)$. Due to our induction hypothesis, we further know that

$$B_{\bar{x},\bar{y}}(i+1, j+1) = B_{\bar{y},\bar{x}}(j+1, i+1) \tag{A.14}$$

Finally, we can infer using algebra symmetry:

$$d_{\text{rep}}(x_i, B_{\bar{x},\bar{y}}(i+1, j+1), y_j) = d_{\text{rep}}(y_j, B_{\bar{y},\bar{x}}(i+1, j+1), x_i) \tag{A.15}$$

Thus an equal term $\theta_{l'}^{\bar{y},\bar{x}}$ has to exist for some l'.

Altogether, this proofs that for each $\theta_l^{\bar{x},\bar{y}}$ there is a $\theta_{l'}^{\bar{y},\bar{x}}$, such that

$$\theta_l^{\bar{x},\bar{y}} = \theta_{l'}^{\bar{y},\bar{x}} \tag{A.16}$$

As multisets are order-invariant, this implies that the input for h is equal in both cases, which in turn implies that

$$A_{\bar{x},\bar{y}}(i, j) = A_{\bar{y},\bar{x}}(j, i) \tag{A.17}$$

This concludes the proof by structural induction.

A.1.2 Weak Symmetry of $\mathcal{G}_{\text{Affine}}$

Have a look at Theorem 8 on page 38 for definitions and claim. We proof our claim by simply looking at all possible productions that are permitted by $\mathcal{G}_{\text{Affine}}$ (see Grammar 2.4 on page 29). In particular we are concerned with the subtree T that is constructed in the production so far and whether a symmetric subtree T' according to Definition 26 on page 38 can be constructed for it.

We differentiate between several types of subtrees.

Front: This kind of subtree is constructed using the production rule

```
SKIPDEL_START = skip_del(<NODE,EMPTY>, SKIPDEL_START);
```

m times, then the rule

```
SKIPDEL_START = SKIPINS_START;
```

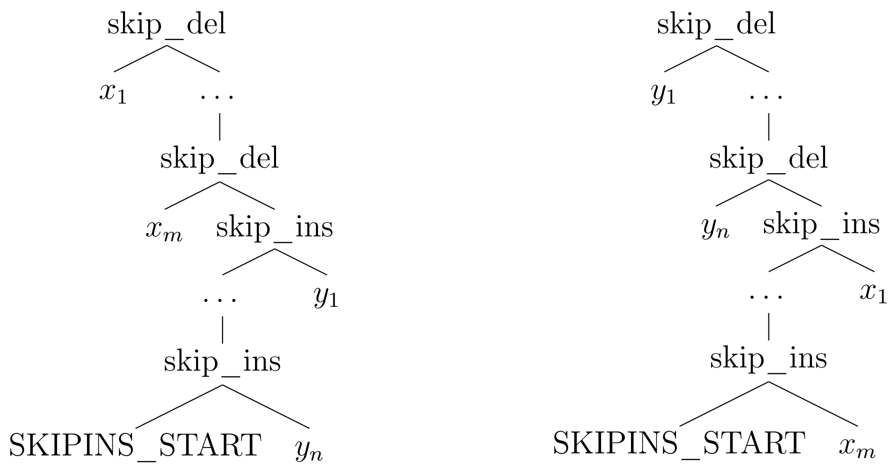


Figure A.1: A front-type subtree (left) and its symmetric counterpart (right).

and finally the production rule

```
SKIPINS_START = skip_ins(<EMPTY,NODE>, SKIPINS_START);
```

n times (see Figure A.1). The symmetric production for this type of subtree is very straightforward: Just apply the first rule n times, then the second one and the third one m times (see Figure A.1). Note that we end up not only with a symmetric subtree, but also with the same nonterminal symbol, namely SKIPINS_START.

If the next production rule is

```
SKIPINS_START = nil(<EMPTY,EMPTY>);
```

we can apply it for the symmetric tree as well and obtain a full $T \in \mathcal{L}(\mathcal{G}_{\text{Affine}})$, as well as its symmetric counterpart $T' \in \mathcal{L}(\mathcal{G}_{\text{Affine}})$.

If we have a front-subtree and apply the production rule

```
SKIPINS_START = rep(<NODE,NODE>, ALI);
```

we do not get back to the nonterminal symbols SKIPDEL_START or SKIPINS _START. However, we arive at the nonterminal symbol ALI. Now we consider the different productions that might follow from here.

Rep: Consider only the production rule:

```
ALI = rep(<NODE,NODE>, ALI);
```

Here the symmetric production is obvious: We just apply the same production rule.

Del: Now we consider productions that start with an application of the rule

```
ALI = del(<NODE,EMPTY>, DEL);
```

then apply

```
DEL = del(<NODE,EMPTY>, DEL);
```

$m - 1$ times, then apply either directly

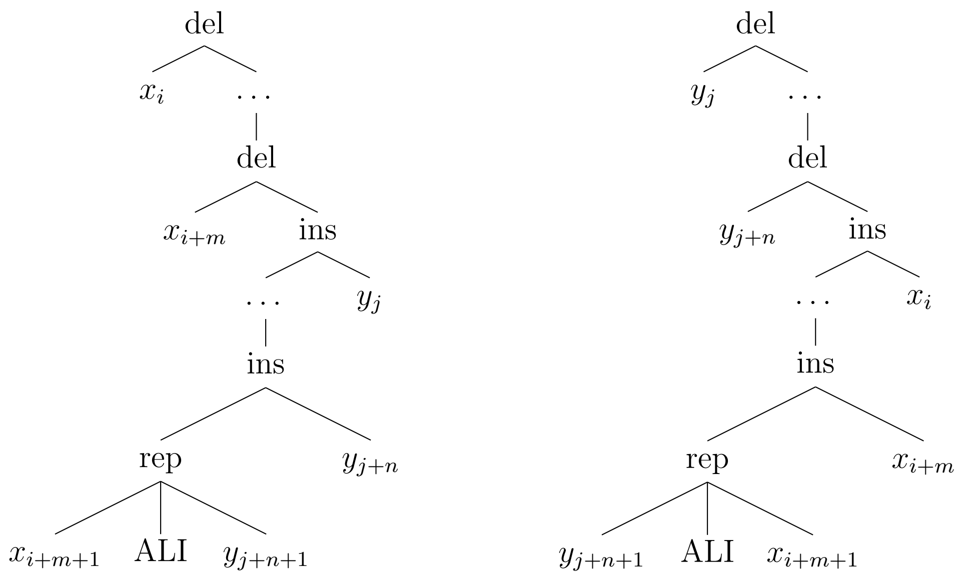


Figure A.2: A del-type subtree (left) and its symmetric counterpart (right).

```
DEL = rep(<NODE,NODE>, ALI);
```

or apply

```
DEL = ins(<EMPTY,NODE>, INS);
```

and thereafter

```
INS = ins(<EMPTY,NODE>, INS);
```

$n-1$ times until finally the rule

```
INS = rep(<NODE,NODE>, ALI);
```

is applied. A subtree created in this way is visualized in Figure A.2. The symmetric construction is basically the same: If $n > 0$ we apply

```
ALI = del(<NODE,EMPTY>, DEL);
```

Then we apply

```
DEL = del(<NODE,EMPTY>, DEL);
```

$n-1$ times. If $m = 0$ we directly apply

```
DEL = rep(<NODE,NODE>, ALI);
```

Otherwise we first apply

```
DEL = ins(<EMPTY,NODE>, INS);
```

and then

```
INS = ins(<EMPTY,NODE>, INS);
```

$m-1$ times. Finally we apply

```
INS = rep(<NODE,NODE>, ALI);
```

The corresponding symmetric subtree is visualized in Figure A.2. Note that we arrive at the same nonterminal symbol ALI.

Ins: Now we consider the subtrees constructed if we first apply the production rule

```
ALI = ins(<EMPTY,NODE>, INS);
```

then apply

```
INS = ins(<EMPTY,NODE>, INS);
```

$n-1$ times and finally apply

```
INS = rep(<NODE,NODE>, ALI);
```

Obviously we can apply the same strategy here as for del type subtrees to construct the symmetric version: If $n > 0$ we apply the rule

```
ALI = del(<NODE,EMPTY>, DEL);
```

Then we apply

```
DEL = del(<NODE,EMPTY>, DEL);
```

$n-1$ times and finally we apply

```
DEL = rep(<NODE,NODE>, ALI);
```

Once again note that we arrive at the nonterminal symbol ALI.

Skipdel-Mid: Consider the subtrees constructed like this: Starting at ALI we first apply the production rule

```
ALI =           del(<NODE,EMPTY>,
                skip_del(<NODE,EMPTY>,
                skip_del(<NODE,EMPTY>,
                SKIPDEL_MID)));
```

Then we apply

```
SKIPDEL_MID =   skip_del(<NODE,EMPTY>, SKIPDEL_MID);
```

m times and finally we apply

```
SKIPDEL_MID =   rep(<NODE,NODE>, ALI);
```

The symmetric subtree can be constructed accordingly by first applying

```
ALI =           ins(<EMPTY,NODE>,
                skip_ins(<EMPTY,NODE>,
                skip_ins(<EMPTY,NODE>,
                SKIPINS_MID))));
```

Then

```
SKIPINS_MID =   skip_ins(<EMPTY,NODE>, SKIPINS_MID);
```

m times and finally

```
SKIPINS_MID =   rep(<NODE,NODE>,ALI);
```

Skipins-Mid: This is just the inverse case of the previous one.

Tail: At last we consider subtrees starting at ALI where we first apply

```
ALI = SKIPDEL_END;
```

Then we apply

```
SKIPDEL_END = skip_del(<NODE,EMPTY>, SKIPDEL_END);
```

m times. Further we apply

```
SKIPDEL_END = SKIPINS_END;
```

We proceed by using

```
SKIPINS_END =    skip_ins(<EMPTY,NODE>, SKIPINS_END);
```

n times and end by applying

```
SKIPINS_END = nil(<EMPTY,EMPTY>);
```

As with the front-type subtrees we construct the symmetric subtree by first applying

```
ALI = SKIPDEL_END;
```

Then n times

```
SKIPDEL_END = skip_del(<NODE,EMPTY>, SKIPDEL_END);
```

Then

```
SKIPDEL_END = SKIPINS_END;
```

Then m times

```
SKIPINS_END =    skip_ins(<EMPTY,NODE>, SKIPINS_END);
```

And finally

```
SKIPINS_END = nil(<EMPTY,EMPTY>);
```

One can also imagine the grammar $\mathcal{G}_{\text{Affine}}$ in a quasi-automaton/dependency graph format, like depicted in Figure A.3 on the next page. Then the meaning of the different subtrees becomes quite clear: They form different paths within the graph, such that every possible production permitted by $\mathcal{G}_{\text{Affine}}$ can be seen as composed of these paths. Using our definitions, we can now apply a simple induction argument over the size of some tree $T \in \mathcal{L}(\mathcal{G}_{\text{Affine}})$: If it just consists of one nil operation it is obviously symmetric. If it is larger than that, look at the last subtree before the nil operation was used. For the remainder of the tree the induction hypothesis applies. For the subtree itself apply the construction of the symmetric subtree as stated above.

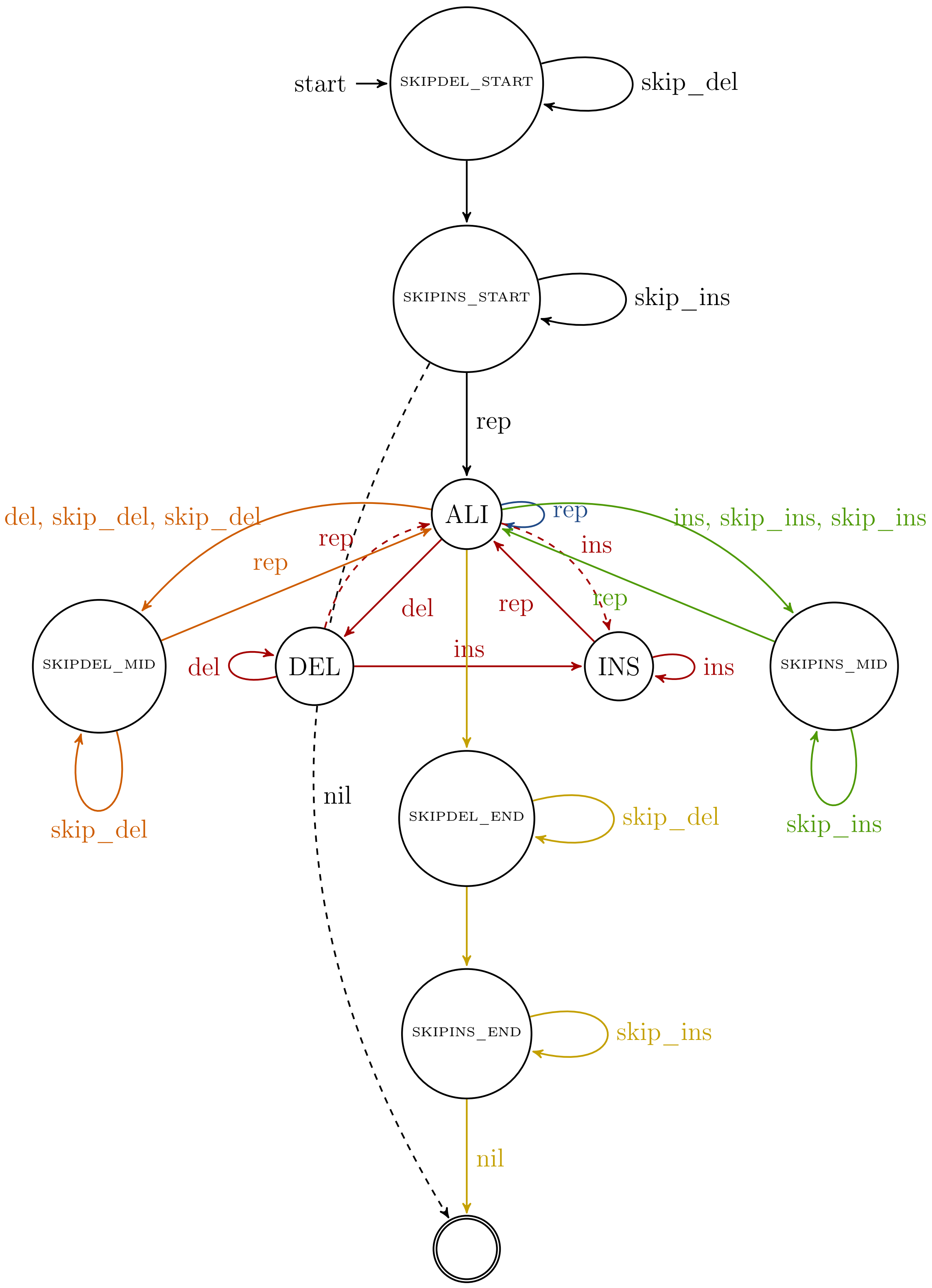


Figure A.3: An automaton-style dependency graph representation of $\mathcal{G}_{\text{Affine}}$, visualizing the different productions for subtrees of the different types discussed in Section A.1.2 on page 101. Black stands for the front-type subtree, blue for the rep-type, red for del- and ins-type subtrees, orange for skipdel-mid, green for skipins-mid and yellow for tail. Shortcuts are displayed with dashed lines.

A.2 Softmin

A.2.1 Derivative

Have a look at Theorem 13 on page 46 for definitions and claim. This derivation is taken from Mokbel et al. [29]:

$$\begin{aligned}
&\frac{\partial}{\partial\lambda}\mathrm{softmin}[\theta_1,\ldots,\theta_L] \\
&\quad= \frac{\partial}{\partial\lambda}\frac{1}{Z}\sum_{l=1}^{L} e_l\cdot\theta_l && \text{(A.18)}\\
&\quad= \frac{1}{Z^2}\left(Z\cdot\sum_{l=1}^{L}\frac{\partial}{\partial\lambda}(e_l\cdot\theta_l) - \left(\sum_{l=1}^{L} e_l\cdot\theta_l\right)\cdot\frac{\partial}{\partial\lambda}Z\right) && \text{(A.19)}\\
&\quad= \frac{1}{Z}\left(\sum_{l=1}^{L}\frac{\partial}{\partial\lambda}(e_l\cdot\theta_l) - \mathrm{softmin}[\theta_1,\ldots,\theta_L]\cdot\frac{\partial}{\partial\lambda}Z\right) && \text{(A.20)}
\end{aligned}$$

For the derivative of $e_l\cdot\theta_l$ we obtain:

$$\begin{aligned}
\frac{\partial}{\partial\lambda}(e_l\cdot\theta_l) &= \left(\frac{\partial}{\partial\lambda}e_l\right)\cdot\theta_l + e_l\cdot\left(\frac{\partial}{\partial\lambda}\theta_l\right) && \text{(A.21)}\\
&= -\beta\cdot e_l\cdot\left(\frac{\partial}{\partial\lambda}\theta_l\right)\cdot\theta_l + e_l\cdot\left(\frac{\partial}{\partial\lambda}\theta_l\right) && \text{(A.22)}\\
&= e_l\cdot\left(\frac{\partial}{\partial\lambda}\theta_l\right)\cdot(-\beta\cdot\theta_l+1) && \text{(A.23)}
\end{aligned}$$

And for the derivative of Z:

$$\begin{aligned}
\frac{\partial}{\partial\lambda}Z &= \sum_{l=1}^{L}\frac{\partial}{\partial\lambda}e_l && \text{(A.24)}\\
&= -\beta\cdot\sum_{l=1}^{L} e_l\cdot\left(\frac{\partial}{\partial\lambda}\theta_l\right) && \text{(A.25)}
\end{aligned}$$

If we plug that into Equation A.20 we obtain:

$$\begin{aligned}
&\frac{\partial}{\partial\lambda}\mathrm{softmin}[\theta_1,\ldots,\theta_L] \\
&\quad= \frac{1}{Z}\left(\sum_{l=1}^{L} e_l \cdot \left(\frac{\partial}{\partial\lambda}\theta_l\right) \cdot (-\beta\cdot\theta_l + 1)\right. \\
&\quad\left. - \mathrm{softmin}[\theta_1,\ldots,\theta_L]\cdot(-\beta)\cdot e_l \cdot \left(\frac{\partial}{\partial\lambda}\theta_l\right)\right) && \text{(A.26)} \\
&\quad= \frac{1}{Z}\left(\sum_{l=1}^{L} e_l \cdot \left(\frac{\partial}{\partial\lambda}\theta_l\right) \cdot (-\beta\cdot\theta_l + 1 + \beta\cdot\mathrm{softmin}[\theta_1,\ldots,\theta_L])\right) && \text{(A.27)} \\
&\quad= \sum_{l=1}^{L} \frac{e_l}{Z} \cdot \left(\frac{\partial}{\partial\lambda}\theta_l\right) \cdot \Big(1 - \beta\cdot(\theta_l - \mathrm{softmin}[\theta_1,\ldots,\theta_L])\Big) && \text{(A.28)}
\end{aligned}$$

A.2.2 Approximation Error

Have a look at Theorem 14 on page 47 for definitions and claim. The proof works as follows:

$$\begin{aligned}
E &= \mathrm{softmin}[\theta_1,\ldots,\theta_L] - \theta_{\min} && \text{(A.29)} \\
&= \frac{\sum_{l=1}^{L} e_l\cdot\theta_l}{\sum_{l=1}^{L} e_l} - \theta_{\min} && \text{(A.30)} \\
&= \frac{\sum_{l=1}^{L} e_l\cdot\exp(-\beta\cdot\theta_{\min})\cdot\theta_l}{\sum_{l=1}^{L} e_l\cdot\exp(-\beta\cdot\theta_{\min})} - \theta_{\min} && \text{(A.31)} \\
&= \frac{\sum_{l=1}^{L} \exp(-\beta\cdot\epsilon_l)\cdot\theta_l}{\sum_{l=1}^{L} \exp(-\beta\cdot\epsilon_l)} - \theta_{\min} && \text{(A.32)} \\
&= \frac{\sum_{l=1}^{L} \exp(-\beta\cdot\epsilon_l)\cdot\theta_l - \theta_{\min}\cdot\Big(\sum_{l=1}^{L}\exp(-\beta\cdot\epsilon_l)\Big)}{\sum_{l=1}^{L} \exp(-\beta\cdot\epsilon_l)} && \text{(A.33)} \\
&= \frac{\sum_{l=1}^{L} \exp(-\beta\cdot\epsilon_l)\cdot\theta_l - \exp(-\beta\cdot\epsilon_l)\cdot\theta_{\min}}{\sum_{l=1}^{L} \exp(-\beta\cdot\epsilon_l)} && \text{(A.34)} \\
&= \frac{\sum_{l=1}^{L} \exp(-\beta\cdot\epsilon_l)\cdot\epsilon_l}{\sum_{l=1}^{L} \exp(-\beta\cdot\epsilon_l)} && \text{(A.35)} \\
&= \mathrm{softmin}[\epsilon_1,\ldots,\epsilon_L] && \text{(A.36)}
\end{aligned}$$

The second part of the proof is similarly simple:

$$E = \frac{\sum_{l=1}^{L} \exp(-\beta \cdot \epsilon_l) \cdot \epsilon_l}{\sum_{l=1}^{L} \exp(-\beta \cdot \epsilon_l)} \tag{A.37}$$

$$= \frac{\sum_{l \in \Theta_{\min}} \exp(-\beta \cdot \epsilon_l) \cdot \epsilon_l + \sum_{l \notin \Theta_{\min}} \exp(-\beta \cdot \epsilon_l) \cdot \epsilon_l}{\sum_{l \in \Theta_{\min}} \exp(-\beta \cdot \epsilon_l) + \sum_{l \notin \Theta_{\min}} \exp(-\beta \cdot \epsilon_l)} \tag{A.38}$$

$$= \frac{\sum_{l \in \Theta_{\min}} \exp(-\beta \cdot 0) \cdot 0 + \sum_{l \notin \Theta_{\min}} \exp(-\beta \cdot \epsilon_l) \cdot \epsilon_l}{\sum_{l \in \Theta_{\min}} \exp(-\beta \cdot 0) + \sum_{l \notin \Theta_{\min}} \exp(-\beta \cdot \epsilon_l)} \tag{A.39}$$

$$= \frac{\sum_{l \notin \Theta_{\min}} \exp(-\beta \cdot \epsilon_l) \cdot \epsilon_l}{|\Theta_{\min}| + \sum_{l \notin \Theta_{\min}} \exp(-\beta \cdot \epsilon_l)} \tag{A.40}$$

A.2.3 Error Limit

Have a look at Theorem 16 on page 48 for definitions and claim. First we intend to fix the form of the worst case softmin inputs. Consider Equation 2.133 on page 47 from Theorem 14:

$$E = \frac{\sum_{l \notin \Theta_{\min}} \exp(-\beta \cdot \epsilon_l) \cdot \epsilon_l}{|\Theta_{\min}| + \sum_{l \notin \Theta_{\min}} \exp(-\beta \cdot \epsilon_l)} \tag{A.41}$$

If we reduce the number of minimum elements, that is the size of $\Theta_{\min}$, the numerator increases and the denominator decreases. Therefore, the error E increases and we can conclude that the worst-case ϵ-terms are of the form $[0, \epsilon_2, \ldots, \epsilon_L]$ with

$$\forall l \in \{2, \ldots, L\} : \epsilon_l > 0 \tag{A.42}$$

Let $[0, \epsilon_2, \ldots, \epsilon_L]$ be such a multiset. Now we proof that there is some (unique) $\epsilon^* \in \mathbb{R}^+$ such that:

$$\epsilon^* = \operatorname*{argmax}_{\epsilon_l} \operatorname{softmin}[0, \epsilon_2, \epsilon_3, \ldots, \epsilon_L] = \operatorname*{argmax}_{\epsilon_l} E \tag{A.43}$$

Consider the soft minimum derivative specified in Equation 2.128 on page 46:

$$\frac{\partial}{\partial \epsilon_l} \operatorname{softmin}[0, \epsilon_2, \ldots, \epsilon_L]$$

$$= \sum_{l'=2}^{L} \frac{e_{l'}}{Z} \cdot \left(\frac{\partial}{\partial \epsilon_l} \epsilon_{l'} \right) \cdot \Big(1 - \beta \cdot (\epsilon_{l'} - \operatorname{softmin}[0, \epsilon_2, \ldots, \epsilon_L]) \Big) \tag{A.44}$$

$$= \frac{e_l}{Z} \cdot \Big(1 - \beta \cdot (\epsilon_l - \operatorname{softmin}[0, \epsilon_2, \ldots, \epsilon_L]) \Big) \overset{!}{=} 0 \tag{A.45}$$

$$\Leftrightarrow 0 \overset{!}{=} 1 - \beta \cdot (\epsilon_l - \operatorname{softmin}[0, \epsilon_2, \ldots, \epsilon_L]) \tag{A.46}$$

$$\Leftrightarrow \epsilon_l \overset{!}{=} \operatorname{softmin}[0, \epsilon_2, \ldots, \epsilon_L] + \frac{1}{\beta} \tag{A.47}$$

This equation has no closed-form solution, but it certainly has *some* (unique) solution. We call the ϵ_l, for which that equation is fulfilled, ϵ^*. It remains to be shown that the approximation error has a maximum (and not a minimum) at

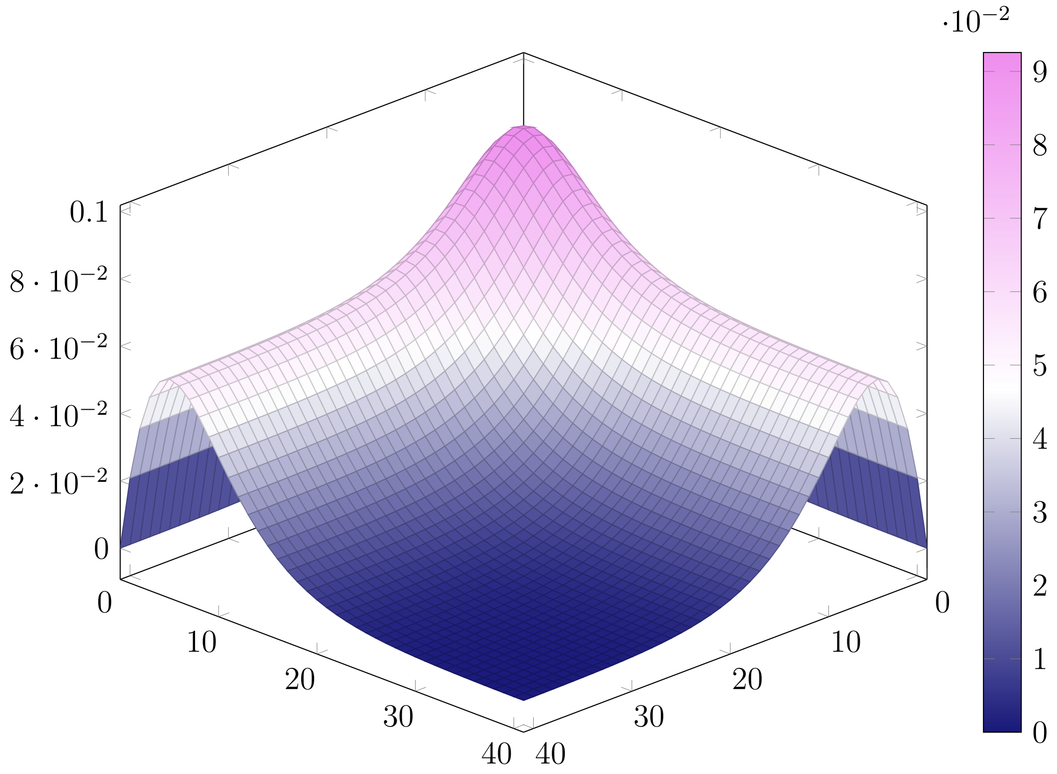


Figure A.4: The soft minimum approximation error E for $\beta = 5$ plotted versus the error contributions ϵ_2, ϵ_3 with $\epsilon_1 = 0$. As shown in Theorem 14 on page 47 this is equivalent to $\text{softmin}[0, \epsilon_2, \epsilon_3]$. As can be seen the worst case approximation error can be found on the diagonal with $\epsilon_2 = \epsilon_3$.

ϵ^*. We check this, by inspecting the behavior of the derivative in a δ-interval around ϵ^*:

$$\begin{aligned}
&\left(\frac{\partial}{\partial \epsilon_l}\text{softmin}[0, \epsilon_2, \dots, \epsilon_L]\right)(\epsilon^* + \delta) \\
=&\frac{e_l}{Z} \cdot \Big(1 - \beta \cdot (\epsilon^* - \text{softmin}[0, \epsilon_2, \dots, \epsilon_L])\Big) && \text{(A.48)} \\
=&\frac{e_l}{Z} \cdot \Big(1 - \beta \cdot (\frac{1}{\beta} + \delta)\Big) && \text{(A.49)} \\
=&\frac{e_l}{Z} \cdot \Big(1 - 1 - \beta \cdot \delta\Big) && \text{(A.50)} \\
=&-\frac{e_l}{Z} \cdot \beta \cdot \delta && \text{(A.51)}
\end{aligned}$$

As $e_l > 0$, $Z > 0$ and $\beta \geq 0$, we can conclude that the derivative is positive for a small $\delta < 0$ and negative for a small $\delta > 0$, which in turn implies a maximum.

Due to the order-invariance of multisets, we can transfer this result to the other variables as well. We obtain:

$$\operatorname*{argmax}_{[\epsilon_2,\dots,\epsilon_L]} \text{softmin}[0, \epsilon_2, \dots, \epsilon_L] = [\epsilon^*, \dots, \epsilon^*] \tag{A.52}$$

A visualization of the maximum for a simple two-dimensional case is shown in Figure A.4 on the preceding page.

Now we apply Equation 2.133 on page 47:

$$\max_{[\theta_1,\dots,\theta_L]} E = \frac{\sum_{l=2}^{L} \exp(-\beta \cdot \epsilon^*) \cdot \epsilon^*}{1 + \sum_{l=2}^{L} \exp(-\beta \cdot \epsilon^*)} \tag{A.53}$$

$$= \frac{(L-1) \cdot \exp(-\beta \cdot \epsilon^*) \cdot \epsilon^*}{1 + (L-1) \cdot \exp(-\beta \cdot \epsilon^*)} \tag{A.54}$$

$$= \frac{(L-1) \cdot \epsilon^*}{\exp(\beta \cdot \epsilon^*) + (L-1)} \tag{A.55}$$

$$\leq (L-1) \cdot \frac{\epsilon^*}{\exp(\beta \cdot \epsilon^*)} \tag{A.56}$$

We can obtain an upper bound for this formula by finding a maximum of this new function:

$$\frac{\partial}{\partial \hat{\epsilon}^*} (L-1) \cdot \frac{\hat{\epsilon}^*}{\exp(\beta \cdot \hat{\epsilon}^*)} \overset{!}{=} 0 \tag{A.57}$$

$$\Leftrightarrow \frac{\exp(\beta \cdot \hat{\epsilon}^*) - \hat{\epsilon}^* \cdot \beta \cdot \exp(\beta \cdot \hat{\epsilon}^*)}{\exp(\beta \cdot \hat{\epsilon}^*)^2} \overset{!}{=} 0 \tag{A.58}$$

$$\Leftrightarrow \frac{1 - \hat{\epsilon}^* \cdot \beta}{\exp(\beta \cdot \hat{\epsilon}^*)} \overset{!}{=} 0 \tag{A.59}$$

$$\Leftrightarrow 1 - \hat{\epsilon}^* \cdot \beta \overset{!}{=} 0 \tag{A.60}$$

$$\Leftrightarrow \hat{\epsilon}^* \overset{!}{=} \frac{1}{\beta} \tag{A.61}$$

To proof that this extremum is indeed a maximum we once again inspect a δ-interval around $\frac{1}{\beta}$:

$$\left(\frac{\partial}{\partial \hat{\epsilon}^*} (L-1) \cdot \frac{\hat{\epsilon}^*}{\exp(\beta \cdot \hat{\epsilon}^*)} \right) (\frac{1}{\beta} + \delta) \tag{A.62}$$

$$= \frac{1 - (\frac{1}{\beta} + \delta) \cdot \beta}{\exp\Big(\beta \cdot (\frac{1}{\beta} + \delta)\Big)} \tag{A.63}$$

$$= \frac{1 - 1 - \delta \cdot \beta}{\exp(1 + \beta \cdot \delta)} \tag{A.64}$$

$$= -\frac{\delta \cdot \beta}{\exp(1 + \beta \cdot \delta)} \tag{A.65}$$

$$\tag{A.66}$$

As $\beta \geq 0$ and $\exp(1 + \beta \cdot \delta) > 0$ we can conclude that the derivative is positive for a small $\delta < 0$ and negative for a small $\delta > 0$, which in turn implies a maximum.

We plug this result into Equation A.56 on the previous page:

$$\max_{[\theta_1,\ldots,\theta_L]} E \leq \frac{(L-1)\cdot\frac{1}{\beta}}{\exp(\beta\cdot\frac{1}{\beta})} = \frac{L-1}{\beta\cdot e} \tag{A.67}$$

This concludes our proof.

Appendix B

Affine grammar translation

To illustrate the translation process, we consider the example of grammar $\mathcal{G}_{\text{Affine}}$ in more detail in this section. We use only one keyword κ and define the respective alphabet as:

$$\Sigma_\kappa := \{\texttt{'a'}, \texttt{'b'}, \texttt{'c'}, \texttt{'d'}, \texttt{'e'}, \texttt{'y'}, \texttt{'z'}\} \tag{B.1}$$

As example input sequences we use:

$$\bar{x} = \texttt{cazzzaad} \qquad (M = 8) \tag{B.2}$$

$$\bar{y} = \texttt{baayyyae} \qquad (N = 8) \tag{B.3}$$

As algebra we define:

$$c^\kappa_{\text{nil}}() := 0 \tag{B.4}$$

$$c^\kappa_{\text{rep}}(a, b) := \begin{cases} 0 & \text{if } a = b \\ 1 & \text{if } a \neq b \end{cases} \tag{B.5}$$

$$c^\kappa_{\text{del}}(a) := 0.7 \tag{B.6}$$

$$c^\kappa_{\text{skip\_del}}(a) := 0.4 \tag{B.7}$$

$$c^\kappa_{\text{ins}}(b) := 0.7 \tag{B.8}$$

$$c^\kappa_{\text{skip\_ins}}(b) := 0.4 \tag{B.9}$$

As choice function we use the strict minimum.

B.1 SKIPINS_END

The production rules for SKIPINS_END are:

```
SKIPINS_END =    skip_ins(<EMPTY,NODE>, SKIPINS_END) |
                 nil(<EMPTY,EMPTY>);
```

As SKIPINS_END is a nonterminal symbol with a nil production rule we can initialize

$$\text{SKIPINS\_END}(M + 1, N + 1) = 0 \tag{B.10}$$

$\bar{x}\backslash\bar{y}$	b	a	a	y	y	y	a	e	-
c									
a									
z									
z									
z									
z									
a									
a									
d									
-	3.2	2.8	2.4	2.0	1.6	1.2	0.8	**0.4**	**0.0**

Table B.1: The dynamic programming table for the nonterminal symbol SKIP-INS_END. Cells that are part of the optimal path are highlighted.

As only one rule remains the rest of the cell filling for this table is trivial:

$$\begin{aligned}
&\text{SKIPINS\_END}(M+1,j) \\
&\qquad = \min\{d_{\text{skip\_ins}}(\text{SKIPINS\_END}(M+1,j+1),y_j)\} && \text{(B.11)} \\
&\qquad = \text{SKIPINS\_END}(M+1,j+1) + c^{\kappa}_{\text{skip\_ins}}(y^{\kappa}_j) && \text{(B.12)} \\
&\qquad = \sum_{j'=j}^{N} c^{\kappa}_{\text{skip\_ins}}(y^{\kappa}_{j'}) && \text{(B.13)}
\end{aligned}$$

All other rows of this table remain empty. The table for our example input is shown in Table B.1.

B.2 SKIPDEL_END

The production rules for SKIPDEL_END are:

```
SKIPDEL_END =     skip_del(<NODE,EMPTY>, SKIPDEL_END) |
                  SKIPINS_END;
```

In the last row of this table we can not yet apply the translation of the first production rule. Thus we just copy the value of SKIPINS_END there. As mentioned before, however, all rows but the last one remain empty for SKIPINS _END such that we can not apply the translation of the second production rule there. Thus the translation becomes fairly easy:

$\bar{x}\backslash\bar{y}$	b	a	a	y	y	y	a	e	-
c	6.4	6.0	5.6	5.2	4.8	4.4	4.0	3.6	3.2
a	6.0	5.6	5.2	4.8	4.4	4.0	3.6	3.2	2.8
z	5.6	5.2	4.8	4.4	4.0	3.6	3.2	2.8	2.4
z	5.2	4.8	4.4	4.0	3.6	3.2	2.8	2.4	2.0
z	4.8	4.4	4.0	3.6	3.2	2.8	2.4	2.0	1.6
a	4.4	4.0	3.6	3.2	2.8	2.4	2.0	1.6	1.2
a	4.0	3.6	3.2	2.8	2.4	2.0	1.6	1.2	0.8
d	3.6	3.2	2.8	2.4	2.0	1.6	1.2	**0.8**	0.4
-	3.2	2.8	2.4	2.0	1.6	1.2	0.8	**0.4**	0.0

Table B.2: The dynamic programming table for the nonterminal symbol SKIPDEL_END. Cells that are part of the optimal path are highlighted. We separated parts of the table with extra spacing that are calculated using different equations. The equations are listed in the text.

$$\begin{aligned}
&\text{SKIPDEL\_END}(M+1, j)\\
&\qquad = \text{SKIPINS\_END}(M+1, j) && \text{(B.14)}\\
&\text{SKIPDEL\_END}(i, j)\\
&\qquad = d_{\text{skip\_del}}(x_i, \text{SKIPDEL\_END}(i+1, j)) && \text{(B.15)}\\
&\qquad = \text{SKIPDEL\_END}(i+1, j) + c^{\kappa}_{\text{skip\_del}}(x^{\kappa}_i) && \text{(B.16)}\\
&\qquad = \text{SKIPINS\_END}(M+1, j) + \sum_{i'=i}^{M} c^{\kappa}_{\text{skip\_del}}(x^{\kappa}_{i'}) && \text{(B.17)}\\
&\qquad = \sum_{j'=j}^{N} c^{\kappa}_{\text{skip\_ins}}(y^{\kappa}_{j'}) + \sum_{i'=i}^{M} c^{\kappa}_{\text{skip\_del}}(x^{\kappa}_{i'}) && \text{(B.18)}
\end{aligned}$$

The table for our example input is shown in Table B.2.

B.3 ALI

ALI is by far the most complicated nonterminal symbol. Its production rules are:

```
ALI =          del(<NODE,EMPTY>, DEL) |
               ins(<EMPTY,NODE>, INS) |
               rep(<NODE,NODE>, ALI) |
               del(<NODE,EMPTY>,
               skip_del(<NODE,EMPTY>,
               skip_del(<NODE,EMPTY>,
               SKIPDEL_MID))) |
               ins(<EMPTY,NODE>,
               skip_ins(<EMPTY,NODE>,
               skip_ins(<EMPTY,NODE>,
```

$\bar{x} \backslash \bar{y}$	b	a	a	y	y	y	a	e	-
c	4.8	4.5	4.6	5.2	4.5	3.8	3.5	3.6	3.2
a	4.5	3.8	4.2	4.3	4.2	3.5	2.8	3.2	2.8
z	4.8	3.9	**3.8**	4.2	3.9	3.3	2.7	2.8	2.4
z	4.1	3.8	3.7	3.8	3.2	2.9	2.3	2.4	2.0
z	3.4	3.1	3.0	3.5	2.8	2.2	1.9	2.0	1.6
a	3.1	2.4	2.3	2.7	2.5	1.8	1.2	1.6	1.2
a	3.5	2.8	2.4	**2.3**	2.2	1.5	.8	1.2	0.8
d	3.6	3.2	2.8	2.4	2.0	1.6	1.2	**0.8**	0.4
-	3.2	2.8	2.4	2.0	1.6	1.2	0.8	0.4	0.0

Table B.3: The dynamic programming table for the nonterminal symbol ALI. Cells that are part of the optimal path are highlighted. We separated parts of the table with extra spacing that are calculated using different equations. The equations are listed in the text.

```
                SKIPINS_MID)))
                SKIPDEL_END;
```

We did not discuss the proper translation of nested operations yet, but it is fairly straightforward:

```
del(<NODE,EMPTY>,
skip_del(<NODE,EMPTY>,
skip_del(<NODE,EMPTY>,
SKIPDEL_MID)))
```

becomes:

$$\mathrm{del}(x_i, \mathrm{skip\_del}(x_{i+1}, \mathrm{skip\_del}(x_{i+2}, \mathrm{SKIPDEL\_START}(i+3, j)))) \quad \text{(B.19)}$$

In the last column and last row we can not apply any operations yet, thus the content of SKIPDEL_END at that point is copied:

$$\mathrm{ALI}(M+1, j) = \mathrm{SKIPDEL\_END}(M+1, j) \quad \text{(B.20)}$$
$$\mathrm{ALI}(i, N+1) = \mathrm{SKIPDEL\_END}(i, N+1) \quad \text{(B.21)}$$

In the cell (M, N) we can only apply replacements. Thus we calculate:

$$\begin{aligned}\mathrm{ALI}(M, N) = \min\{&\mathrm{ALI}(M+1, N+1) + c^{\kappa}_{\mathrm{rep}}(x^{\kappa}_M, y^{\kappa}_N),\\ &\mathrm{SKIPDEL\_END}(M, N)\}\end{aligned} \quad \text{(B.22)}$$

The 2 cells above it additionally allow deletions and the 2 cells left of it additionally allow insertions. Thus we obtain for $M - 3 < i < M$:

$$\begin{aligned}\mathrm{ALI}(i, N) = \min\{&\mathrm{ALI}(i+1, N+1) + c^{\kappa}_{\mathrm{rep}}(x^{\kappa}_i, y^{\kappa}_N),\\ &\mathrm{DEL}(i+1, N) + c^{\kappa}_{\mathrm{del}}(x^{\kappa}_i),\\ &\mathrm{SKIPDEL\_END}(i, N)\}\end{aligned} \quad \text{(B.23)}$$

and for $N-3 < j < N$:

$$\begin{aligned}\text{ALI}(M,j) = \min\{&\text{ALI}(M+1,j+1) + c^{\kappa}_{\text{rep}}(x^{\kappa}_M, y^{\kappa}_j),\\ &\text{INS}(M,j+1) + c^{\kappa}_{\text{ins}}(y^{\kappa}_j),\\ &\text{SKIPDEL\_END}(M,j)\}\end{aligned} \tag{B.24}$$

Both of these obviously depend on DEL and INS being partly filled already.

We introduce further dependencies in the rest of the last row and column: For $i \leq M-3$ the cells (i,N) allow skip-deletions in the middle as well. We obtain:

$$\begin{aligned}\text{ALI}(i,N) = \min\{&\text{ALI}(i+1,N+1) + c^{\kappa}_{\text{rep}}(x^{\kappa}_i, y^{\kappa}_N),\\ &\text{DEL}(i+1,N) + c^{\kappa}_{\text{del}}(x^{\kappa}_i),\\ &\text{SKIPDEL\_MID}(i+3,N)\\ &+ c^{\kappa}_{\text{del}}(x^{\kappa}_i) + c^{\kappa}_{\text{skip\_del}}(x^{\kappa}_{i+1}) + c^{\kappa}_{\text{skip\_del}}(x^{\kappa}_{i+2}),\\ &\text{SKIPDEL\_END}(i,N)\}\end{aligned} \tag{B.25}$$

Accordingly we obtain a similar equation for $j \leq N-3$:

$$\begin{aligned}\text{ALI}(M,j) = \min\{&\text{ALI}(M+1,j+1) + c^{\kappa}_{\text{rep}}(x^{\kappa}_M, y^{\kappa}_j),\\ &\text{INS}(M,j+1) + c^{\kappa}_{\text{ins}}(y^{\kappa}_j),\\ &\text{SKIPINS\_MID}(M,j+3)\\ &+ c^{\kappa}_{\text{ins}}(y^{\kappa}_j) + c^{\kappa}_{\text{skip\_ins}}(y^{\kappa}_{j+1}) + c^{\kappa}_{\text{skip\_ins}}(y^{\kappa}_{j+2}),\\ &\text{SKIPDEL\_END}(M,j)\}\end{aligned} \tag{B.26}$$

The cells $\{(i,j) | M-3 < i < M \wedge N-3 < j < N\}$ allow the following operations:

```
ALI =           del(<NODE,EMPTY>, DEL)  |
                ins(<EMPTY,NODE>, INS)  |
                rep(<NODE,NODE>, ALI)  |
                SKIPDEL_END;
```

The translation is:

$$\begin{aligned}\text{ALI}(i,j) = \min\{&\text{ALI}(i+1,j+1) + c^{\kappa}_{\text{rep}}(x^{\kappa}_i, y^{\kappa}_j),\\ &\text{DEL}(i+1,j) + c^{\kappa}_{\text{del}}(x^{\kappa}_i),\\ &\text{INS}(i,j+1) + c^{\kappa}_{\text{ins}}(y^{\kappa}_j),\\ &\text{SKIPDEL\_END}(i,j)\}\end{aligned} \tag{B.27}$$

The cells $\{(i,j) | i \leq M-3 \wedge N-3 < j < N\}$ allow skip-deletions in the middle as well:

$$\begin{aligned}\text{ALI}(i,j) = \min\{&\text{ALI}(i+1,j+1) + c^{\kappa}_{\text{rep}}(x^{\kappa}_i, y^{\kappa}_j),\\ &\text{DEL}(i+1,j) + c^{\kappa}_{\text{del}}(x^{\kappa}_i),\\ &\text{INS}(i,j+1) + c^{\kappa}_{\text{ins}}(y^{\kappa}_j),\\ &\text{SKIPDEL\_MID}(i+3,j)\\ &+ c^{\kappa}_{\text{del}}(x^{\kappa}_i) + c^{\kappa}_{\text{skip\_del}}(x^{\kappa}_{i+1}) + c^{\kappa}_{\text{skip\_del}}(x^{\kappa}_{i+2}),\\ &\text{SKIPDEL\_END}(i,j)\}\end{aligned} \tag{B.28}$$

$\bar{x}\backslash\bar{y}$	b	a	a	y	y	y	a	e
c	4.8	5.2	5.3	5.2	4.5	3.8	4.2	3.8
a	4.5	3.8	4.2	4.9	4.2	3.5	2.8	3.4
z	4.8	4.7	4.8	4.2	3.9	3.3	3.4	3.0
z	4.1	4.0	4.5	3.8	3.2	2.9	3.0	2.6
z	3.4	3.3	3.7	3.5	2.8	2.2	2.6	2.2
a	3.1	2.4	2.3	3.2	2.5	1.8	1.2	1.8
a	3.5	2.8	2.4	2.9	2.2	1.5	0.8	1.4
d	3.8	3.4	3.0	2.6	2.2	1.8	1.4	1.0

Table B.4: The dynamic programming table for the nonterminal symbol INS. Cells that are part of the optimal path are highlighted. We separated parts of the table with extra spacing that are calculated using different equations. The equations are listed in the text.

The equation for the cells $\{(i,j)|M-3 < i < M \wedge j \leq N-3\}$ can be translated accordingly:

$$\begin{aligned}\text{ALI}(i,j) = \min\{&\text{ALI}(i+1,j+1) + c^{\kappa}_{\text{rep}}(x^{\kappa}_i, y^{\kappa}_j),\\ &\text{DEL}(i+1,j) + c^{\kappa}_{\text{del}}(x^{\kappa}_i),\\ &\text{INS}(i,j+1) + c^{\kappa}_{\text{ins}}(y^{\kappa}_j),\\ &\text{SKIPINS\_MID}(i,j+3)\\ &\ + c^{\kappa}_{\text{ins}}(y^{\kappa}_j) + c^{\kappa}_{\text{skip\_ins}}(y^{\kappa}_{j+1}) + c^{\kappa}_{\text{skip\_ins}}(y^{\kappa}_{j+2}),\\ &\text{SKIPDEL\_END}(i,j)\}\end{aligned} \tag{B.29}$$

Finally we can calculate the last region of the table, namely the cells $\{(i,j)|i \leq M-3 \wedge j \leq N-3\}$ where all operations are allowed:

$$\begin{aligned}\text{ALI}(i,j) = \min\{&\text{ALI}(i+1,j+1) + c^{\kappa}_{\text{rep}}(x^{\kappa}_i, y^{\kappa}_j),\\ &\text{DEL}(i+1,j) + c^{\kappa}_{\text{del}}(x^{\kappa}_i),\\ &\text{INS}(i,j+1) + c^{\kappa}_{\text{ins}}(y^{\kappa}_j),\\ &\text{SKIPDEL\_MID}(i+3,j)\\ &\ + c^{\kappa}_{\text{del}}(x^{\kappa}_i) + c^{\kappa}_{\text{skip\_del}}(x^{\kappa}_{i+1}) + c^{\kappa}_{\text{skip\_del}}(x^{\kappa}_{i+2}),\\ &\text{SKIPINS\_MID}(i,j+3)\\ &\ + c^{\kappa}_{\text{ins}}(y^{\kappa}_j) + c^{\kappa}_{\text{skip\_ins}}(y^{\kappa}_{j+1}) + c^{\kappa}_{\text{skip\_ins}}(y^{\kappa}_{j+2}),\\ &\text{SKIPDEL\_END}(i,j)\}\end{aligned} \tag{B.30}$$

The table for our example input is shown in Table B.3 on page 116.

B.4 INS

The production rules for INS are:

```
INS =           ins(<EMPTY,NODE>, INS) |
                rep(<NODE,NODE>, ALI);
```

$\bar{x}\backslash\bar{y}$	b	a	a	y	y	y	a	e
c	4.8	4.5	4.9	5.2	4.5	3.8	3.5	3.8
a	4.5	3.8	4.2	4.9	4.2	3.5	2.8	3.4
z	4.8	4.5	4.4	4.2	3.9	3.3	3.3	3.0
z	4.1	3.8	3.7	3.8	3.2	2.9	2.6	2.6
z	3.4	3.1	3.0	3.5	2.8	2.2	1.9	2.2
a	3.1	2.4	2.3	3.2	2.5	1.8	1.2	1.8
a	3.5	2.8	2.4	2.9	2.2	1.5	0.8	1.4
d	3.8	3.4	3.0	2.6	2.2	1.8	1.4	1.0

Table B.5: The dynamic programming table for the nonterminal symbol DEL. Cells that are part of the optimal path are highlighted. We separated parts of the table with extra spacing that are calculated using different equations. The equations are listed in the text.

As no operations are possible in the last row and column, these cells remain unused. In the next column replacements are possible. For all $i < M$ all operations in the production rules are possible. Thus we obtain:

$$\text{INS}(M, j) = \text{ALI}(M+1, j+1) + c^{\kappa}_{\text{rep}}(x^{\kappa}_M, y^{\kappa}_j) \tag{B.31}$$

$$\begin{aligned}\text{INS}(i, j) = \min\{&\text{INS}(i, j+1) + c^{\kappa}_{\text{ins}}(y^{\kappa}_j),\\ &\text{ALI}(i+1, j+1) + c^{\kappa}_{\text{rep}}(x^{\kappa}_i, y^{\kappa}_j)\}\end{aligned} \tag{B.32}$$

The table for our example input is shown in Table B.4 on the preceding page.

B.5 DEL

The production rules for DEL are:

```
DEL =           del(<NODE,EMPTY>, DEL) |
                ins(<EMPTY,NODE>, INS) |
                rep(<NODE,NODE>, ALI);
```

Taking possible operations into account we obtain the following equations:

$$\text{DEL}(M, N) = \text{ALI}(M+1, N+1) + c^{\kappa}_{\text{rep}}(x^{\kappa}_M, y^{\kappa}_N) \tag{B.33}$$

$$\begin{aligned}\text{DEL}(M, j) = \min\{&\text{INS}(M, j+1) + c^{\kappa}_{\text{ins}}(y^{\kappa}_j),\\ &\text{ALI}(M+1, j+1) + c^{\kappa}_{\text{rep}}(x^{\kappa}_M, y^{\kappa}_j)\}\end{aligned} \tag{B.34}$$

$$\begin{aligned}\text{DEL}(i, N) = \min\{&\text{DEL}(i+1, N) + c^{\kappa}_{\text{del}}(x^{\kappa}_i),\\ &\text{ALI}(i+1, N+1) + c^{\kappa}_{\text{rep}}(x^{\kappa}_i, y^{\kappa}_N)\}\end{aligned} \tag{B.35}$$

$$\begin{aligned}\text{DEL}(i, j) = \min\{&\text{DEL}(i+1, j) + c^{\kappa}_{\text{del}}(x^{\kappa}_i),\\ &\text{INS}(i, j+1) + c^{\kappa}_{\text{ins}}(y^{\kappa}_j),\\ &\text{ALI}(i+1, j+1) + c^{\kappa}_{\text{rep}}(x^{\kappa}_i, y^{\kappa}_j)\}\end{aligned} \tag{B.36}$$

The latter equation holds for all cells in $\{(i, j)|i < M \wedge j < N\}$. The table for our example input is shown in Table B.5.

$\bar{x}\backslash\bar{y}$	b	a	a	y	y	y	a	e
c	4.8	5.2	5.0	4.6	4.2	3.8	4.2	3.8
a	4.2	3.8	4.2	4.0	3.6	3.2	2.8	3.4
z	4.8	4.7	4.5	4.1	3.7	3.3	3.4	3.0
z	4.1	4.0	4.0	3.6	3.2	2.9	3.0	2.6
z	3.4	3.3	3.4	3.0	2.6	2.2	2.6	2.2
a	2.8	2.4	2.3	2.4	2.0	1.6	1.2	1.8
a	3.2	2.8	2.4	2.0	1.6	1.2	**0.8**	1.4
d	3.8	3.4	3.0	2.6	2.2	1.8	1.4	1.0

Table B.6: The dynamic programming table for the nonterminal symbol SKIPINS_MID. Cells that are part of the optimal path are highlighted. We separated parts of the table with extra spacing that are calculated using different equations. The equations are listed in the text.

B.6 SKIPINS_MID

The production rules for SKIPINS_MID are:

```
SKIPINS_MID =    skip_ins(<EMPTY,NODE>, SKIPINS_MID) |
                 rep(<NODE,NODE>,ALI);
```

Taking possible operations into account we obtain the following equations:

$$\text{SKIPINS\_MID}(i,N) = \text{ALI}(i+1,N+1) + c^{\kappa}_{\text{rep}}(x^{\kappa}_i, y^{\kappa}_N) \tag{B.37}$$

$$\text{SKIPINS\_MID}(i,j) = \min\{ \text{SKIPINS\_MID}(i,j+1) + c^{\kappa}_{\text{skip\_ins}}(y^{\kappa}_j), \text{ALI}(i+1,j+1) + c^{\kappa}_{\text{rep}}(x^{\kappa}_i, y^{\kappa}_j)\} \tag{B.38}$$

The latter equation holds for all cells in $\{(i,j)|i \leq M \wedge j < N\}$. The table for our example input is shown in Table B.6.

B.7 SKIPDEL_MID

The production rules for SKIPDEL_MID are:

```
SKIPDEL_MID =    skip_del(<NODE,EMPTY>, SKIPDEL_MID) |
                 rep(<NODE,NODE>,ALI);
```

Taking possible operations into account we obtain the following equations:

$$\text{SKIPDEL\_MID}(M,j) = \text{ALI}(M+1,j+1) + c^{\kappa}_{\text{rep}}(x^{\kappa}_M, y^{\kappa}_j) \tag{B.39}$$

$$\text{SKIPDEL\_MID}(i,j) = \min\{ \text{SKIPDEL\_MID}(i+1,j) + c^{\kappa}_{\text{skip\_del}}(x^{\kappa}_i), \text{ALI}(i+1,j+1) + c^{\kappa}_{\text{rep}}(x^{\kappa}_i, y^{\kappa}_j)\} \tag{B.40}$$

$\bar{x}\backslash\bar{y}$	b	a	a	y	y	y	a	e
c	4.8	4.2	4.3	5.0	4.4	3.8	3.2	3.8
a	4.6	3.8	3.9	4.6	4.0	3.4	2.8	3.4
z	4.2	3.6	3.5	4.2	3.6	3.0	2.4	3.0
z	3.8	3.2	3.1	3.8	3.2	2.6	2.0	2.6
z	3.4	2.8	2.7	3.5	2.8	2.2	1.6	2.2
a	3.8	2.4	**2.3**	3.2	2.5	1.8	1.2	1.8
a	4.2	2.8	2.4	3.0	2.6	2.2	0.8	1.4
d	3.8	3.4	3.0	2.6	2.2	1.8	1.4	1.0

Table B.7: The dynamic programming table for the nonterminal symbol SKIPDEL_MID. Cells that are part of the optimal path are highlighted. We separated parts of the table with extra spacing that are calculated using different equations. The equations are listed in the text.

$\bar{x}\backslash\bar{y}$	b	a	a	y	y	y	a	e	-
c	4.8	5.2	5.0	4.6	4.2	3.8	4.2	3.8	
a	**4.2**	**3.8**	4.2	4.0	3.6	3.2	2.8	3.4	
z	4.8	4.7	4.5	4.1	3.7	3.3	3.4	3.0	
z	4.1	4.0	4.1	3.7	3.3	2.9	3.0	2.6	
z	4.2	3.8	3.4	3.0	2.6	2.2	2.6	2.2	
a	2.8	2.4	2.3	2.4	2.0	1.6	1.2	1.8	
a	3.2	2.8	2.4	2.0	1.6	1.2	0.8	1.4	
d	3.8	3.4	3.0	2.6	2.2	1.8	1.4	1.0	
-	3.2	2.8	2.4	2.0	1.6	1.2	0.8	0.4	0.0

Table B.8: The dynamic programming table for the nonterminal symbol SKIPINS_START. Cells that are part of the optimal path are highlighted. We separated parts of the table with extra spacing that are calculated using different equations. The equations are listed in the text.

The latter equation holds for all cells in $\{(i,j)|i < M \wedge j \leq N\}$. The table for our example input is shown in Table B.7.

B.8 SKIPINS_START

The production rules for SKIPINS_START are:

```
SKIPINS_START = skip_ins(<EMPTY,NODE>, SKIPINS_START) |
                rep(<NODE,NODE>, ALI) |
                nil(<EMPTY,EMPTY>);
```

$\bar{x}\backslash\bar{y}$	b	a	a	y	y	y	a	e	-
c	**4.6**	4.2	4.3	4.4	4.0	3.6	3.2	3.6	3.2
a	**4.2**	3.8	3.9	4.0	3.6	3.2	2.8	3.2	2.8
z	4.0	3.6	3.5	3.6	3.2	2.8	2.4	2.8	2.4
z	3.6	3.2	3.1	3.2	2.8	2.4	2.0	2.4	2.0
z	3.2	2.8	2.7	2.8	2.4	2.0	1.6	2.0	1.6
a	2.8	2.4	2.3	2.4	2.0	1.6	1.2	1.6	1.2
a	3.2	2.8	2.4	2.0	1.6	1.2	0.8	1.2	0.8
d	3.6	3.2	2.8	2.4	2.0	1.6	1.2	0.8	0.4
-	3.2	2.8	2.4	2.0	1.6	1.2	0.8	0.4	0.0

Table B.9: The dynamic programming table for the nonterminal symbol SKIPDEL_START. Cells that are part of the optimal path are highlighted. We separated parts of the table with extra spacing that are calculated using different equations. The equations are listed in the text.

Taking possible operations into account we obtain the following equations:

$$\text{SKIPINS\_START}(M+1, N+1) = 0 \tag{B.41}$$

$$\text{SKIPINS\_START}(M+1, j) = \text{SKIPINS\_START}(M+1, j+1) + c^{\kappa}_{\text{skip\_ins}}(y^{\kappa}_j) \tag{B.42}$$

$$\text{SKIPINS\_START}(i, N) = \text{ALI}(i+1, N+1) + c^{\kappa}_{\text{rep}}(x^{\kappa}_i, y^{\kappa}_N)\} \tag{B.43}$$

$$\begin{aligned}\text{SKIPINS\_START}(i, j) = \min\{ & \\ & \text{SKIPINS\_START}(i, j+1) + c^{\kappa}_{\text{skip\_ins}}(y^{\kappa}_j), \\ & \text{ALI}(i+1, j+1) + c^{\kappa}_{\text{rep}}(x^{\kappa}_i, y^{\kappa}_j)\}\end{aligned} \tag{B.44}$$

where

- Equation B.42 holds for all cells in $\{(M+1, j) | j \leq N\}$,
- Equation B.43 holds for all cells in $\{(i, N) | i \leq M\}$ and
- Equation B.44 holds for all cells in $\{((i, j) | i \leq M \wedge j < N\}$,

The table for our example input is shown in Table B.8 on the previous page.

B.9 SKIPDEL_START

The production rules for SKIPDEL_START are:

```
SKIPDEL_START = skip_del(<NODE,EMPTY>, SKIPDEL_START) |
                SKIPINS_START;
```

$$\begin{aligned}
&\text{SKIPDEL\_START}(M+1, j) = && \text{(B.45)}\\
&\qquad \text{SKIPINS\_START}(M+1, j) \\
&\text{SKIPDEL\_START}(i, N+1) = && \text{(B.46)}\\
&\qquad \text{SKIPDEL\_START}(i+1, N+1) + c^{\kappa}_{\text{skip\_del}}(x^{\kappa}_i) \\
&\text{SKIPDEL\_START}(i, j) = \min\{ && \text{(B.47)}\\
&\qquad \text{SKIPDEL\_START}(i+1, j) + c^{\kappa}_{\text{skip\_del}}(x^{\kappa}_i), \\
&\qquad \text{SKIPINS\_START}(i, j)\}
\end{aligned}$$

where

- Equation B.45 holds for all cells in $\{(M+1, j) | j \leq N+1\}$,
- Equation B.46 holds for all cells in $\{(i, N+1) | i \leq M\}$ and
- Equation B.47 holds for all cells in $\{((i, j) | i \leq M \wedge j \leq N\}$,

The table for our example input is shown in Table B.9 on the facing page.

Example Result

We find the optimum alignment distance according to $\mathcal{G}_{\text{Affine}}$ in the cell

$$\text{SKIPDEL\_START}(1, 1) = 4.6$$

The optimum tree can be constructed via backtracing. We highlighted the optimum backtrace in all previous tables. The tree is visualized in Figure B.1 on the next page.

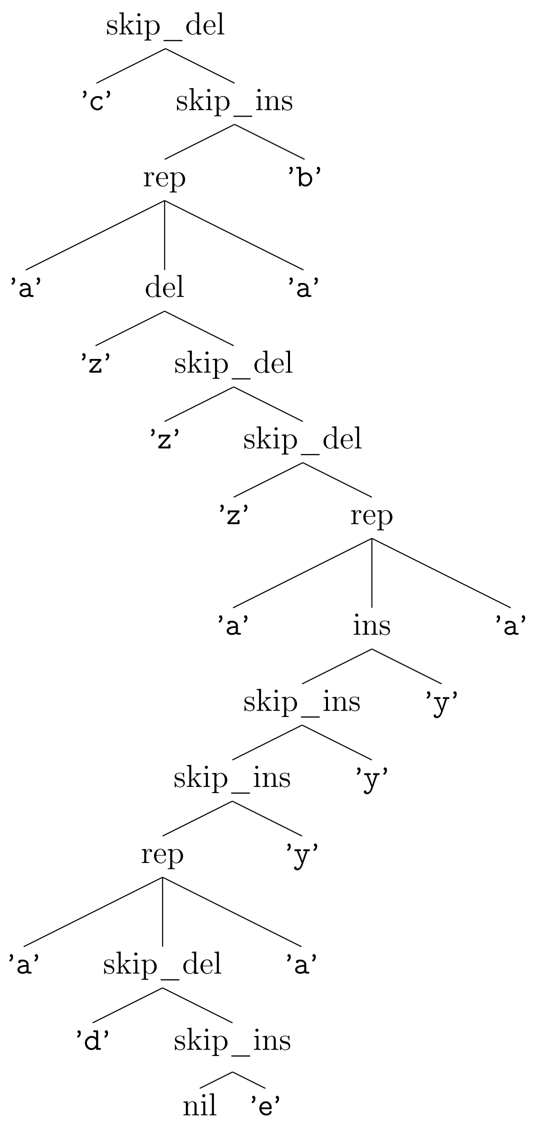


Figure B.1: The optimum tree for an Affine Alignment of the input sequences $\bar{x}$ = cazzzaad and $\bar{y}$ = baayyyae using the comparator function $c^{\kappa}_{\text{skip\_ins}}(b) = c^{\kappa}_{\text{skip\_del}}(a) = 0.4$, $c^{\kappa}_{\text{ins}}(b) = c^{\kappa}_{\text{del}}(a) = 0.7$, $c^{\kappa}_{\text{rep}}(a,b) = 1$ (for $a \neq b$) and $c^{\kappa}_{\text{rep}}(a,b) = 0$ (for $a = b$). As choice function we used min.

Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Masterarbeit selbständig verfasst und gelieferte Datensätze, Zeichnungen, Skizzen und graphische Darstellungen selbständig erstellt habe. Ich habe keine anderen Quellen als die angegebenen benutzt und habe die Stellen der Arbeit, die anderen Werken entnommen sind - einschließlich verwendeter Tabellen und Abbildungen - in jedem einzelnen Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht.

Bielefeld, den

. . . .
Benjamin Paaßen