DISSERTATION
INTELLIGENT SYSTEMS

METRIC LEARNING FOR STRUCTURED DATA

BENJAMIN PAASSEN

Bielefeld University,
Faculty of Technology,
Machine Learning Research Group

SUPERVISED BY
PROF. DR. BARBARA HAMMER

REVIEWED BY
PROF. DR. BARBARA HAMMER,
PROF. DR. ALESSANDRO SPERDUTI,
PROF. DR. LARS SCHMIDT-THIEME

MAY 10TH, 2019

ACKNOWLEDGEMENTS

This work would not not have been possible without help from many people, both within and beyond the work group. First and foremost, I wish to thank my supervisor, Barbara Hammer, who has been a tremendous inspiration and role model throughout my time in Bielefeld. Additionally, my reviewers, Alessandro Sperduti and Lars Schmidt-Thieme, deserve thanks for their careful and in-depth reading as well as very helpful comments for this revised version of the manuscript.

I also extend my thanks to all my co-workers, especially Bassam Mokbel, who has been a brilliant and kind mentor, Alexander Schulz, who has supported and shared my passion for prosthetic research beyond our own regular projects, and Christina Göpfert, whose sharp mind has enabled all of our shared projects.

Beyond that I owe gratitude to all my collaborators, who have kindly contributed their skill, knowledge, and time to this research, namely Sebastian Groß and Niels Pinkwart at the Humboldt-University of Berlin, Thomas Price and Tiffany Barnes at the North Carolina State University, Thekla Morgenroth at the University of Exeter, Michelle Statemeyer at the University of Melbourne, Cosima Prahm at the Medical University of Vienna, Janne Hahne at the Medical University of Göttingen, and Claudio Gallicchio as well as Alessio Micheli at the University of Pisa.

I also wish to thank my parents for their patience, trust, and support over all these years, my brother for putting up with me, my many on-line and off-line friends for their kind support and believing in me beyond reasonable degrees of confidence, especially Thekla for being both an inspiring example and challenging competition like an ideal big sibling should be, and finally my partner, who has helped me up and kept me grounded whenever needed and who has taught me a strong will, a sharp wit, and a kind heart when interacting with the world.

Finally, I would be remiss to thank the German Research Foundation who have supported this research in the project "Learning Dynamic Feedback for Intelligent Tutoring Systems" (DynaFIT, grant number HA 2719/6-2), and the center of excellence "Cognitive Interaction Technology" (CITEC, grant number EXC 277).

ABSTRACT

Distance measures form a backbone of machine learning and information retrieval in many application fields such as computer vision, natural language processing, and biology. However, general-purpose distances may fail to capture semantic particularities of a domain, leading to wrong inferences downstream. Motivated by such failures, the field of *metric learning* has emerged. Metric learning is concerned with learning a distance measure from data which pulls semantically similar data closer together and pushes semantically dissimilar data further apart. Over the past decades, metric learning approaches have yielded state-of-the-art results in many applications. Unfortunately, these successes are mostly limited to vectorial data, while metric learning for structured data remains a challenge.

In this thesis, I present a metric learning scheme for a broad class of sequence edit distances which is compatible with any differentiable cost function, and a scalable, interpretable, and effective tree edit distance learning scheme, thus pushing the boundaries of metric learning for structured data.

Furthermore, I make learned distances more useful by providing a novel algorithm to perform time series prediction solely based on distances, a novel algorithm to infer a structured datum from edit distances, and a novel algorithm to transfer a learned distance to a new domain using only little data and computation time.

Finally, I apply these novel algorithms to two challenging application domains. First, I support students in intelligent tutoring systems. If a student gets stuck before completing a learning task, I predict how capable students would proceed in their situation and guide the student in that direction via edit hints. Second, I use transfer learning to counteract disturbances for bionic hand prostheses to make these prostheses more robust in patients' everyday lives.

CONTENTS

Contents v

1 Introduction 1

2 Background and Related Work 7
2.1 Kernels and Distances . . . 7
2.2 Kernels for Structured Data . . . 11
2.3 Edit Distances . . . 12
2.3.1 Sequence Edit Distance . . . 12
2.3.2 Algebraic Dynamic Programming . . . 14
2.3.3 Tree Edit Distance . . . 21
2.3.4 Graph Edit Distance . . . 27
2.4 Metric Learning for Edit Distances . . . 29
2.4.1 Good Edit Similarity Learning . . . 29
2.5 Learning Vector Quantization . . . 31
2.5.1 Generalized Matrix Learning Vector Quantization . . . 32
2.5.2 Labeled Gaussian Mixture Models . . . 33
2.5.3 Relational Generalized Learning Vector Quantization . . . 36
2.5.4 Median Generalized Learning Vector Quantization . . . 36
2.6 Distance-based Time Series Prediction . . . 38

3 Sequence Edit Distance Learning 43
3.1 Method . . . 44
3.2 Experiments . . . 51
3.3 Conclusion and Limitations . . . 56

4 Tree Edit Distance Learning 59
4.1 Method . . . 60
4.2 Experiments . . . 68
4.3 Conclusion . . . 73

5 Time Series Prediction for Structured Data 77
5.1 Background and Related Work . . . 79
5.2 Method . . . 81
5.3 Experiments . . . 85
5.4 Discussion and Conclusion . . . 90

6 Application to Intelligent Tutoring Systems 91
6.1 An integrated view of edit-based hint policies . . . 92
6.2 Method . . . 103
6.3 Experiments . . . 106
6.4 Conclusion . . . 113

7 Supervised Transfer Learning 115
7.1 Related Work . . . 116
7.2 Method . . . 119

7.3 Experiments . . . 124
7.4 Conclusion . . . 128

8 Application to Bionic Hand Prostheses 129
8.1 Experiments . . . 131
8.2 Conclusion . . . 137

9 Conclusions and Outlook 139

Publications in the Context of this Thesis 143

References 145

Glossary 163

Acronyms 167

A Proofs 169
A.1 Proof of Theorem 2.1 . . . 169
A.2 Proof of Theorem 2.2 . . . 171
A.3 Proof of Theorem 2.3 . . . 172
A.4 Proof of Theorem 2.4 . . . 174
A.5 Proof of Theorem 2.5 . . . 178
A.6 Proof of Theorem 2.7 . . . 189
A.7 Proof of Theorem 2.8 . . . 193
A.8 Proof of Theorem 3.1 . . . 195
A.9 Proof of Theorem 3.2 . . . 198
A.10 Proof of Theorem 3.3 . . . 200
A.11 Proof of Theorem 3.4 . . . 202
A.12 Proof of Theorem 3.5 . . . 205
A.13 Proof of Theorem 4.2 . . . 207
A.14 Proof of Theorem 4.3 . . . 228
A.15 Proof of Theorem 5.1 . . . 231
A.16 Proof of Theorem 6.2 . . . 233
A.17 Kernelized Othogonal Matching Pursuit . . . 234
A.18 Proof of Theorem 7.1 . . . 236

1 INTRODUCTION

> *The notion of nearness or proximity, which is objectively defined only for pairs of objects in physical space, tends to be carried over to very different situations where the space in which entities can be closer together or further apart is not at all evident.*
>
> — ROGER SHEPARD, 1962

According to foundational works in cognitive science, proximity and distance are key concepts in our understanding of the world (Gentner and Markman 1997; Hodgetts, Hahn, and Chater 2009; Medin, Goldstone, and Genter 1993; Nosofsky 1992; Shepard 1962; Tversky 1977). For example, we estimate the properties of an individual based on our experience with similar individuals in the past (Mussweiler 2003; Eliot Smith and Zarate 1992); we form mental categories based on similarities to exemplars (Edward Smith and Medin 1981; Markman 1998); and we try to transfer solutions from known problems to new but similar problems (Barnett and Ceci 2002).

These cognitive behaviors have inspired various machine learning algorithms. In particular, one-nearest-neighbor or learning vector quantization classify data by assigning the label of the closest exemplar in a data base (Cover and Hart 1967; Kohonen 1995), k-means or relational neural gas cluster data based on their distance to cluster means (MacQueen 1967; Hammer and Hasenfuss 2007), and multiple transfer learning algorithms optimize the similarity between data from related domain to transfer knowledge between these domains (Duan, Xu, and I. Tsang 2012; Kulis, Saenko, and Darrell 2011; Weiss, Khoshgoftaar, and D. Wang 2016). Key to all these approaches is that we have a sufficient understanding of what it means for objects to be *similar* or *different* (Medin, Goldstone, and Genter 1993). In other words, we require a measure of distance that is reasonable for our task at hand.

In most machine learning applications, we utilize general-purpose metrics, such as the Euclidean distance (Bellet, Habrard, and Sebban 2014). However, because these metrics do not take particularities of a domain into account, they may lead to incorrect inferences. For example, when classifying the control signal for a prosthesis, some channels of the signal may be more predictive than others (Paaßen et al. 2018). More generally, default metrics may fail to regard semantically similarly objects as similar because their data representation appears different, and may fail to regard semantically different objects as different, because their data representation appears similar. Therefore, any subsequent inferences based on apparent similarity or difference may be semantically flawed.

This begs the question, how can we learn a metric better takes domain-specific semantics into account? This very question is at the heart of *metric learning* (Bellet, Habrard, and Sebban 2014; Kulis 2013). Generally speaking, a metric learning approach takes as input a set N^+ of semantically close pairs (x, y) as well as a set N^- of semantically distant pairs (x, y) and attempts to learn parameters $\mathbf{\Lambda}$ of a metric $d_{\mathbf{\Lambda}}$ such that $d_{\mathbf{\Lambda}}(x, y)$ is small for all $(x, y) \in N^+$ and $d_{\mathbf{\Lambda}}(x, y)$ is large for all $(x, y) \in N^-$ (Bellet, Habrard, and Sebban 2014).

Most metric learning approaches to date learn a generalization of the Euclidean distance d, namely the generalized quadratic form

$$d_{\boldsymbol{\Lambda}}(\vec{x}, \vec{y}) = \sqrt{(\vec{x} - \vec{y})^\top \cdot \boldsymbol{\Lambda} \cdot (\vec{x} - \vec{y})}$$

where $\vec{x}$ and $\vec{y}$ are m-dimensional real vectors and $\boldsymbol{\Lambda}$ is a symmetric, positive semi-definite $m \times m$-matrix, which constitutes the parameters to be learned (Bellet, Habrard, and Sebban 2014; Kulis 2013; Schneider, Biehl, and Hammer 2009a). This kind of metric learning has been widely successful and has achieved state-of-the-art performance in various information retrieval tasks, especially in computer vision (Bellet, Habrard, and Sebban 2014; Davis et al. 2007; Köstinger et al. 2012; Liao et al. 2015; Lim, Lanckriet, and McFee 2013; Davis et al. 2010). A key appeal is that a learned generalized Euclidean distance retains intuitive properties of the data, such as symmetry, non-negativity, shift-invariance, and the triangular inequality. Furthermore, Euclidean metric learning is flexible enough to support a broad range of cost functions, as well as various architectures and parametrizations, such as deep learning models (De Vries, Memisevic, and Courville 2016; Hu, Lu, and Tan 2014; Oh Song et al. 2016).

However, not all problems involving distances can be tackled with a generalized Euclidean distance. As Hodgetts, Hahn, and Chater (2009) point out: "Real-world objects are not merely represented as lists of features or dimensions but represented in a structured way that considers not only the composite elements but the relations between these different elements." Examples of such *structured data* include chemical processes, human and animal motion data, electrocardiography readings, financial time series, natural language sentences and syntax trees, abstract syntax trees of source code, RNA, DNA, and protein sequences, phylogenetic trees, RNA secondary structures, and glycan molecules (Akutsu 2010; Bellet, Habrard, and Sebban 2014; S. Henikoff and J. G. Henikoff 1992; Keogh and Ratanamahatana 2005; McKenna et al. 2010; Mikolov et al. 2013; Pawlik and Augsten 2011; Rivers and Koedinger 2015; T. F. Smith and Waterman 1981; Snover et al. 2006). For such structured data, we require *structure metrics*, such as the Levenshtein distance, dynamic time warping, or the tree edit distance (Levenshtein 1965; Vintsyuk 1968; Zhang and Shasha 1989).

As with the Euclidean distance, these structure metrics may not correspond to domain-specific semantics. For example, when analyzing protein sequences, the standard string edit distance assumes that all amino acids have the same pairwise distance which does not correspond to biological reality (S. Henikoff and J. G. Henikoff 1992; Hourai, Akutsu, and Akiyama 2004; Kann, Qian, and Goldstein 2000; Saigo, Vert, and Akutsu 2006). Similarly, when considering abstract syntax trees of source code, the standard tree edit distance assumes that all syntactic building blocks of computer programs have the same distance, which does not accurately reflect the function of these building blocks (Paaßen, Mokbel, and Hammer 2016; Paaßen, Gallicchio, et al. 2018). As such, metric learning approaches for structured data are sorely needed. Unfortunately, present approaches for metric learning on structured data are almost exclusively limited to pulling semantically similar data closer together but can not push semantically dissimilar data away, are limited to the string edit distance in particular, and do not scale well to bigger datasets or bigger structures (Bellet, Habrard, and Sebban 2014). This leads us to the first two research questions I wish to tackle in this work.

RQ1: Can we apply metric learning to a broader class of structure metrics?

RQ2: Can we perform metric learning on structured data efficiently and at scale?

I investigate these questions in detail in Chapters 3 and 4. In particular, I use the framework of algebraic dynamic programming (Giegerich, Meyer, and Steffen 2004) to derive general-purpose algorithms that compute a broad class of sequence metrics as well as their gradients in quadratic time. Using these gradients, it is possible to perform metric learning using any differentiable and distance-based cost function.

Further, in Chapter 4, I extend this approach in several ways to make it faster, by learning a sparse classification model and by optimizing the gradient computation, more interpretable by learning symbol embeddings instead of cost parameters, and more general by learning extending it to the tree edit distance. By virtue of these changes I can scale metric learning to larger datasets, such as natural language data with thousands of trees and hundreds of thousand of nodes, and can achieve competitive results on datasets of computer programs and glycan molecules, outperforming one of the best tree edit distance learning algorithms to date.

Beyond these research questions, I am also interested in downstream applications of a learned metric. There is a rich history of machine learning approaches using distances and similarities to address a broad range of machine learning tasks, especially dimensionality reduction (Gisbrecht, Mokbel, and Hammer 2010; Gisbrecht, Schulz, and Hammer 2015; Sammon 1969; Van der Maaten and Hinton 2008), clustering (Gordon 1987; Hammer and Hasenfuss 2007; Hammer and Hasenfuss 2010; S. Johnson 1967), classification (Balcan, Blum, and Srebro 2008; Cover and Hart 1967; Hammer, D. Hofmann, et al. 2014; Nebel, Kaden, et al. 2017), and regression (Nadaraya 1964; Rasmussen and Williams 2005). However, these approaches only consider distance data as *input* and return vectorial data as output. This begs the question:

RQ3: Can we perform predictive tasks with a distance representation *as output*?

In Chapter 5, I explore this question exemplarily for the task of time series prediction, that is, the task of predicting the state of a structured datum x_{t+1} given the previous states $x_1, \ldots, x_t$. I find that the data point x_{t+1} can be represented in terms of its distances to previous points in a data set. In an experimental evaluation I further demonstrate that my predictive scheme outperforms baselines, both for classical theoretical models of structured data evolution, and for practical datasets.

An apparent limitation of my predictive scheme is that it does only provide a distance representation output, *not* a structured output. In other words, we only know the distances of the predicted point to our remaining data, but we do not know what the predicted point actually looks like. Inferring the primal form of a predicted point requires an inversion of the distance representation, which is challenging even for vectorial data, and may be impossible in general for structured data (Bakır, Weston, and Schölkopf 2003; Bakır, Zien, and Tsuda 2004; Kwok and I. W.-H. Tsang 2004). Therefore, my fourth research question is as follows.

RQ4: Can we invert the distance representation of edit distances?

In Chapter 6, I provide a novel algorithm to invert the distance representation of edit distances and use this inversion mechanism for an application in intelligent tutoring systems for computer programming. In particular, I consider the scenario of a student trying to write a computer program but getting stuck before completion. In such a case, I can use the time series prediction mechanism from Chapter 5 to predict how capable students would have continued their program, and then use my inversion mechanism to infer an edit the stuck student could apply to continue in the same direction as capable students. I find experimentally that my hint generation scheme is competitive with other state-of-the-art approaches on real-world data from intelligent tutoring systems.

A final challenge in unlocking the full potential of distance representations is to make a learned metric usable in scenarios beyond its original scope, i.e.:

RQ5: Can we transfer a learned metric to a different, but related domain?

In general, transferring knowledge from a source domain to a target domain is the topic of *transfer learning* or *domain adaptation* (S. J. Pan and Q. Yang 2010; Weiss, Khoshgoftaar, and D. Wang 2016). In Chapter 7, I provide a novel framework to formalize supervised transfer learning by explicitly learning a function that maps from the target to the source domain. By applying this learned mapping to target domain data, we can then re-use our source domain model without changes. The key advantage of my scheme is that it is agnostic regarding the downstream processing pipeline. No matter how complicated a processing pipeline may be, if the relationship between target and source domain is sufficiently simple, we can learn it time- and data-efficiently.

In Chapter 8, I demonstrate the viability of my approach for the example domain of bionic hand prostheses. For decades, researchers have developed machine learning systems, which can infer the desired motion of a hand prostheses from the muscle signals in the patient's stump (Farina et al. 2014). However, while these systems tend to work well under lab conditions, they break down under everyday disturbances, such as shifts of the recording electrodes around the stump (Farina et al. 2014; Khushaba et al. 2014). In such everyday situations, recording large amounts of training data to learn a new model is not a viable option due to time constraints, making electrode shifts an ideal scenario for transfer learning. I demonstrate experimentally that my transfer learning can considerably enhance the accuracy of a disturbed model using less data and less time compared to multiple existing baselines.

In summary, my work contributes

- a general-purpose framework for gradient-based metric learning on sequence edit distances in quadratic time,
- a scalable approach for gradient-based metric learning for the tree edit distance, which yields state-of-the-art results in tree edit distance learning,
- a novel time series prediction algorithm for time series of structured data to date,
- a novel inversion mechanism for edit distance representations to date, and
- an extremely data- and time-efficient transfer learning algorithm for distance-based classification models.

Beyond developing these algorithms, I utilize them to address difficult challenges in contemporary research, namely to generate hints in intelligent tutoring systems, and to counteract electrode shifts in bionic hand prostheses.

I am grateful to have been given the opportunity to present this work in journals as well as renown international conferences, and to have received several awards for these presentations. In particular, this thesis covers work presented in the following journal and conference publications.

Conference Publications:

- Paaßen, Benjamin, Bassam Mokbel, and Barbara Hammer (2015a). "A Toolbox for Adaptive Sequence Dissimilarity Measures for Intelligent Tutoring Systems". In: *Proceedings of the 8th International Conference on Educational Data Mining (EDM 2015)*. (Madrid, Spain). Ed. by Olga Christina Santos et al. International Educational Datamining Society, pp. 632–632. URL: http://www.educationaldatamining.org/EDM2015/uploads/papers/paper_257.pdf.

- — (2015b). "Adaptive structure metrics for automated feedback provision in Java programming". English. In: *Proceedins of the 23rd European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN 2015)*. (Bruges, Belgium). Ed. by Michel Verleysen. **Best student paper award**. i6doc.com, pp. 307–312. URL: http://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2015-43.pdf.

- Göpfert, Christina, Benjamin Paaßen, and Barbara Hammer (2016). "Convergence of Multi-pass Large Margin Nearest Neighbor Metric Learning". In: *Proceedings of the 25th International Conference on Artificial Neural Networks (ICANN 2016)*. (Barcelona, Spain). Ed. by Alessandro E.P. Villa, Paolo Masulli, and Antonio Javier Pons Rivero. Vol. 9886. Lecture Notes in Computer Science. Springer, pp. 510–517. DOI: 10.1007/978-3-319-44778-0_60.

- Paaßen, Benjamin, Christina Göpfert, and Barbara Hammer (2016). "Gaussian process prediction for time series of structured data". In: *Proceedings of the 24th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN 2016)*. (Bruges, Belgium). Ed. by Michel Verleysen. i6doc.com, pp. 41–46. URL: http://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2016-109.pdf.

- Paaßen, Benjamin, Joris Jensen, and Barbara Hammer (2016). "Execution Traces as a Powerful Data Representation for Intelligent Tutoring Systems for Programming". English. In: *Proceedings of the 9th International Conference on Educational Data Mining (EDM 2016)*. (Raleigh, North Carolina, USA). Ed. by Tiffany Barnes, Min Chi, and Mingyu Feng. **Exemplary Paper**. International Educational Datamining Society, pp. 183–190. URL: http://www.educationaldatamining.org/EDM2016/proceedings/paper_17.pdf.

- Paaßen, Benjamin, Alexander Schulz, and Barbara Hammer (2016). "Linear Supervised Transfer Learning for Generalized Matrix LVQ". In: *Proceedings of the Workshop New Challenges in Neural Computation (NC^2 2016)*. (Hannover, Germany). Ed. by Barbara Hammer, Thomas Martinetz, and Thomas Villmann. **Best presentation award**, pp. 11–18. URL: https://www.techfak.uni-bielefeld.de/~fschleif/mlr/mlr_04_2016.pdf#page=14.

- Prahm, Cosima et al. (2016). "Transfer Learning for Rapid Re-calibration of a Myoelectric Prosthesis after Electrode Shift". In: *Proceedings of the 3rd International Conference on NeuroRehabilitation (ICNR 2016)*. (Segovia, Spain). Ed. by Jaime Ibáñez et al. Vol. 15. Converging Clinical and Engineering Research on Neurorehabilitation II. Biosystems & Biorobotics. **Runner-Up for Best Student Paper Award**. Springer, pp. 153–157. DOI: 10.1007/978-3-319-46669-9_28.
- Paaßen, Benjamin et al. (2017). "An EM transfer learning algorithm with applications in bionic hand prostheses". In: *Proceedings of the 25th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN 2017)*. (Bruges, Belgium). Ed. by Michel Verleysen. i6doc.com, pp. 129–134. URL: http://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2017-57.pdf.
- Paaßen, Benjamin, Claudio Gallicchio, et al. (2018). "Tree Edit Distance Learning via Adaptive Symbol Embeddings". In: *Proceedings of the 35th International Conference on Machine Learning (ICML 2018)*. (Stockholm, Sweden). Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research, pp. 3973–3982. URL: http://proceedings.mlr.press/v80/paassen18a.html.

Journal Publications:

- Mokbel, Bassam, Benjamin Paaßen, et al. (2015). "Metric learning for sequences in relational LVQ". English. In: *Neurocomputing* 169, pp. 306–322. DOI: 10.1016/j.neucom.2014.11.082.
- Paaßen, Benjamin, Bassam Mokbel, and Barbara Hammer (2016). "Adaptive structure metrics for automated feedback provision in intelligent tutoring systems". In: *Neurocomputing* 192, pp. 3–13. DOI: 10.1016/j.neucom.2015.12.108.
- Paaßen, Benjamin, Christina Göpfert, and Barbara Hammer (2018). "Time Series Prediction for Graphs in Kernel and Dissimilarity Spaces". In: *Neural Processing Letters* 48.2, pp. 669–689. DOI: 10.1007/s11063-017-9684-5.
- Paaßen, Benjamin, Barbara Hammer, et al. (2018). "The Continuous Hint Factory - Providing Hints in Vast and Sparsely Populated Edit Distance Spaces". In: *Journal of Educational Datamining* 10.1, pp. 1–35. URL: https://jedm.educationaldatamining.org/index.php/JEDM/article/view/158.
- Paaßen, Benjamin et al. (2018). "Expectation maximization transfer learning and its application for bionic hand prostheses". In: *Neurocomputing* 298, pp. 122–133. DOI: 10.1016/j.neucom.2017.11.072.

My thesis has the following structure. First, Chapter 2 covers background knowledge and related work for the remaining chapters. In Chapter 3, I describe a general-purpose learning approach for sequence edit distances, followed by a scalable state-of-the-art metric learning approach for the tree edit distance in Chapter 4. Chapter 5 details an algorithm for time series prediction on structured data, and in Chapter 6, I apply this algorithm for intelligent tutoring systems. Further, Chapter 7 describes a transfer learning algorithm for distance-based models, which I apply to counteract electrode shifts in bionic hand prostheses in Chapter 8. Finally, Chapter 9 provides conclusions and outlook.

2 BACKGROUND AND RELATED WORK

Summary: This chapter covers background knowledge upon which we build in the following chapters. In particular, we revisit basics regarding distances and kernels and go on to cover specific kernels and distances for structured data, with a focus on edit distances, which we adapt via metric learning later on. Further, we review existing metric learning approaches for structured data and position our own work in that context. We close this chapter by covering some distance-based machine learning methods, namely learning learning vector quantization models, Gaussian mixture models, and Gaussian processes.

2.1 KERNELS AND DISTANCES

This entire work is centered around notions of distance. Intuitively, distance is a spatial concept, referring to the length of the shortest path connecting two points in space. However, distance also serves as a more general tool in human cognition, referring to any kind of quantitative measure of dissimilarity between objects (Shepard 1962; Tversky 1977; Nosofsky 1992; Medin, Goldstone, and Genter 1993; Gentner and Markman 1997; Hodgetts, Hahn, and Chater 2009). Accordingly, distances have become a flexible and powerful tool in machine learning, far beyond a strict spatial interpretation (Pekalska and Duin 2005). In this thesis, we define a distance in the general, mathematical sense as follows.

Definition 2.1 (Distance)**.** Let $\mathcal{X}$ be an arbitrary set. A function $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is called a *distance* or a *metric* if and only if for all $x, y, z \in \mathcal{X}$ it holds:

$$d(x,y) \geq 0 \quad \text{(non-negativity)} \tag{2.1}$$

$$d(x,x) = 0 \quad \text{(self-equality)} \tag{2.2}$$

$$x \neq y \Rightarrow d(x,y) > 0 \quad \text{(discernibility)} \tag{2.3}$$

$$d(x,y) = d(y,x) \quad \text{(symmetry)} \tag{2.4}$$

$$d(x,z) + d(z,x) \geq d(x,y) \quad \text{(triangular inequality)} \tag{2.5}$$

We also call these five conditions the *metric axioms*.

Following Nebel, Kaden, et al. (2017), we call d a *semi-* or *pseudo-metric* if all axioms except for discernibility are fulfilled.

Note that all the metric axioms conform to our intuitions about physical distance, namely that there are no negative distances, that any object has no distance to itself, that no two different objects can occupy the same location, that we travel the same length from x to y as from y to x, and that the shortest connection between two points is always the direct path instead of making detours (Shepard 1962).

Spatial distances are a special case of this general notion of distance. In particular, we call such a distance Euclidean.

Definition 2.2 (Euclidean Distance)**.** Let $\mathcal{X}$ be some arbitrary set and let $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$. We call d an *Euclidean* distance, if there exists some function $\phi : \mathcal{X} \to \mathbb{R}^m$ for some $m \in \mathbb{N}$[1], such that for all $x, y \in \mathcal{X}$ it holds:

$$d(x, y) = \|\phi(x) - \phi(y)\|, \quad \text{where} \quad \|\vec{x}\| := \sqrt{\vec{x}^\top \cdot \vec{x}}$$

We call ϕ the *spatial mapping* for d.

In other words, we call a distance Euclidean if it is equivalent to the standard Euclidean distance in $\mathbb{R}^m$ for all points in the image of ϕ on $\mathcal{X}$. This spatial interpretation is key to so-called *relational* machine learning approaches, which perform learning in the image of ϕ (Hammer and Hasenfuss 2010; Hammer, D. Hofmann, et al. 2014). In this thesis for example, we apply relational generalized learning vector quantization (RGLVQ) (refer to Section 2.5.3) for metric learning purposes (refer to Chapter 3).

Euclidean distances are intuitively related to kernel approaches in machine learning which also map implicitly to a m-dimensional space, albeit in terms of an inner product instead of a standard Euclidean distance. More precisely, we define a kernel as follows.

Definition 2.3 (Kernel)**.** Let $\mathcal{X}$ be some arbitrary set. A function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is called a *kernel* if there exists some function $\phi : \mathcal{X} \to \mathbb{R}^m$ for some $m \in \mathbb{N}$ such that for all $x, y \in \mathcal{X}$ it holds:

$$k(x, y) = \phi(x)^\top \cdot \phi(y)$$

We call ϕ the *spatial mapping* for k.

In other words, a k is a function that corresponds to a standard inner product in $\mathbb{R}^m$. As with relational methods, kernel-based methods perform machine learning in the image of ϕ, even though the data are only represented in terms of their pairwise kernel values (T. Hofmann, Schölkopf, and Smola 2008). In this thesis, we use kernels for structured data (refer to Section 2.2) and Gaussian process regression (GPR) as a kernel-based method (refer to Section 2.6). In Chapters 5 and 6, we utilize GPR to predict time series of structured data.

Note that Euclidean distances and kernels are strongly related because they both rely on a spatial mapping ϕ. More precisely, the following theorem from the literature accumulates the most important formal relations between both concepts.

Theorem 2.1. *Let $\mathcal{X}$ be some arbitrary set and let $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$. Then it holds: d is Euclidean if and only if there exists a kernel $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$, such that for all $x, y \in \mathcal{X}$: $d(x, y)^2 = k(x, x) - 2 \cdot k(x, y) + k(y, y)$.*

Now, let $\mathcal{X} = \{x_1, \ldots, x_M\}$ be a finite set and let $s : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$. It holds: s is a kernel if and only if the matrix $\mathbf{S} \in \mathbb{R}^{M \times M}$ with entries $\mathbf{S}_{i,j} = s(x_i, x_j)$ is symmetric and positive semi-definite.

Further, let $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ be a self-equal and symmetric function on $\mathcal{X}$, and let s_d be defined as follows.

$$s_d(x_i, x_j) := \frac{1}{2}\Big(- d(x_i, x_j)^2 + \frac{1}{M}\sum_{k=1}^{M} d(x_i, x_k)^2 + d(x_k, x_j)^2 - \frac{1}{M}\sum_{l=1}^{M} d(x_k, x_l)^2 \Big) \tag{2.6}$$

1 We note that, for infinite $\mathcal{X}$, m can become infinite as well, but we will refrain from a detailed discussion of this issue for simplicity. In our case, we implicitly assume that m is finite either intrinsically or due to the fact that datasets are finite.

Table 2.1: The pairwise string edit distances $d(x, y)$ (top left), the corresponding kernel values $s(x, y)$ (top right), the eigenvalues of $\boldsymbol{S}$ (bottom right), and the vectorial embeddings $\phi(x)$ for the strings ϵ, `a`, and `ab`.

$d(x,y)$	ϵ	`a`	`ab`	$s_d(x,y)$	ϵ	`a`	`ab`	eigenvalues	$\phi(\epsilon)$	$\phi(\texttt{a})$	$\phi(\texttt{ab})$
ϵ	0	1	2	ϵ	1	0	−1	2	$\begin{pmatrix}-1\\0\\0\end{pmatrix}$	$\begin{pmatrix}0\\0\\0\end{pmatrix}$	$\begin{pmatrix}1\\0\\0\end{pmatrix}$
`a`	1	0	1	`a`	0	0	0	0			
`ab`	2	1	0	`ab`	−1	0	1	0			

Then it holds for all $i, j \in \{1, \ldots, M\}$*:* $d(x_i, x_j)^2 = s_d(x_i, x_i) - 2 \cdot s_d(x_i, x_j) + s_d(x_j, x_j)$.

Finally, it holds: d *is Euclidean if and only if the matrix* $\boldsymbol{S} \in \mathbb{R}^{M \times M}$ *with entries* $\boldsymbol{S}_{i,j} = s_d(x_i, x_j)$ *is positive semi-definite.*

Proof. The proofs of these claims have been done by Torgerson (1952) and Pekalska and Duin (2005, pp. 108, 118-124). For a version adjusted to our notation, refer to Appendix A.1. □

As an example, consider the dataset $\mathcal{X} = \{\epsilon, \texttt{a}, \texttt{ab}\}$ with the standard string edit distance of Levenshtein (1965). The corresponding distance values $d(x, y)$, the values $s_d(x, y)$, and the embedded vectors $\phi(x)$, and the eigenvalues of $\boldsymbol{S}$ are shown in Table 2.1. Because all eigenvalues of $\boldsymbol{S}$ are non-negative, $\boldsymbol{S}$ is a kernel matrix and thus the distance d is Euclidean on this dataset. In other words, the standard Euclidean distance between $\phi(x)$ and $\phi(y)$ corresponds exactly to $d(x, y)$. Indeed, because two eigenvalues are zero, the embedding has effectively only one dimension with $\phi(\epsilon) = -1$, $\phi(\texttt{a}) = 0$, and $\phi(\texttt{ab}) = 1$. Note that this embedding is equivalent to *metric multi-dimensional scaling* as described by Torgerson (1952).

Also note that all Euclidean distances are metrics in the sense of Definition 2.1, but that not all metrics are Euclidean. For example, if we extend the dataset in Table 2.1 by the string `b`, the corresponding similarity matrix $\boldsymbol{S}$ has a negative eigenvalue of -0.25, which in turn means that it is not a kernel matrix, which finally implies that the original distance is not Euclidean.

This limitation has severe practical implications, because it means that we explicitly need to ensure that the matrix $\boldsymbol{S}$ for our distance d is positive semi-definite. The canonical way to do so is to compute the eigenvalue decomposition of $\boldsymbol{S}$ and either set negative eigenvalues to zero (clip eigenvalue correction), set all eigenvalues to their absolute value (flip eigenvalue correction), or subtract the smallest eigenvalue from all others (shift eigenvalue correction) (Gisbrecht and Schleif 2015; Nebel, Kaden, et al. 2017). Note that all these techniques have two drawbacks. First, the Eigenvalue decomposition requires $\mathcal{O}(M^3)$ time to compute, which may be infeasible in practice. Fortunately, linear-time approximations via the Nyström-technique do exist (Gisbrecht and Schleif 2015). Second, manipulating the eigenvalues distorts the original distance values, which may result in rank-differences and invalid inferences downstream (Nebel, Kaden, et al. 2017). Accordingly, we attempt to avoid eigenvalue correction whenever possible, and make explicit where it can not be avoided.

Fortunately, Pekalska and Duin (2005) have established the notion of pseudo-Euclidean distances, which still permit spatial reasoning in a weaker form but do not require eigenvalue correction.

Definition 2.4 (Pseudo-Euclidean Distance)**.** Let $\mathcal{X}$ be some arbitrary set and let $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$. We call d an *pseudo-Euclidean* distance, if there exist two functions $\phi^+ : \mathcal{X} \to \mathbb{R}^m$ and $\phi^- : \mathcal{X} \to \mathbb{R}^n$ for some $m, n \in \mathbb{N}$, such that for all $x, y \in \mathcal{X}$ it holds:

$$d(x,y)^2 = \|\phi^+(x) - \phi^+(y)\|^2 - \|\phi^-(x) - \phi^-(y)\|^2 \tag{2.7}$$

We call ϕ^+ the *positive spatial mapping* and ϕ^- the *negative spatial mapping* for d.

The reason we do not require an eigenvalue correction to construct a pseudo-Euclidean distance is the following theorem by Pekalska and Duin (2005), which guarantees that any function that is symmetric and self-equal is a pseudo-Euclidean distance.

Theorem 2.2. *Let $\mathcal{X}$ be some arbitrary set and let $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$. If d is Euclidean with spatial map $\phi : \mathcal{X} \to \mathbb{R}^m$, it is also pseudo-Euclidean with positive spatial map $\phi^+(x) := \phi(x)$ and $\phi^-(x) := 0$.*

Now, let $\mathcal{X} = \{x_1, \ldots, x_M\}$ be a finite set and let $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$. It holds: d is pseudo-Euclidean if and only if d is symmetric and self-equal.

Proof. The first claim follows trivially from the definitions of Euclidean and pseudo-Euclidean distances.

With respect to the second claim, refer to Pekalska and Duin (2005, p. 122-124). A version of the proof adapted to our notation here is shown in Appendix A.2. □

In Chapters 5 and 6, we utilize the notion of pseudo-Euclidean distances to perform time series prediction for structured data. An issue with learning in the (pseudo-)Euclidean space is that we need to compute an eigenvalue decomposition of the similarity matrix $\boldsymbol{S}$ in order to construct the space explicitly. Fortunately, an *implicit* representation is sufficient if we restrict ourselves to the affine hull of a training data set. Within this affine hull, we can compute any pairwise distance relying only on the pairwise distances in the training data and affine coefficients, as Hammer and Hasenfuss (2010) have shown.

Theorem 2.3. *Let $\mathcal{X}$ be some arbitrary set and let $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ be a pseudo-Euclidean distance on $\mathcal{X}$ with the spatial mappings $\phi^+ : \mathcal{X} \to \mathbb{R}^m$ and $\phi^- : \mathcal{X} \to \mathbb{R}^n$. Further, let $\{x_1, \ldots, x_M\} \subseteq \mathcal{X}$ be a finite subset of $\mathcal{X}$, and let $\vec{\alpha}, \vec{\beta} \in \mathbb{R}^M$ such that $\sum_{i=1}^M \alpha_i = \sum_{i=1}^M \beta_i = 1$. Finally, let $\boldsymbol{X}^+ = \big(\phi^+(x_1), \ldots, \phi^+(x_M)\big) \in \mathbb{R}^{M \times m}$ and $\boldsymbol{X}^- = \big(\phi^-(x_1), \ldots, \phi^-(x_M)\big) \in \mathbb{R}^{M \times n}$ be the matrices of positive and negative spatial representations for all x_i, and let $\boldsymbol{D}^2$ be the $M \times M$ matrix with the entries $\boldsymbol{D}^2_{i,j} = d(x_i, x_j)^2$. Then, it holds:*

$$\|\boldsymbol{X}^+ \cdot \vec{\alpha} - \boldsymbol{X}^+ \cdot \vec{\beta}\|^2 - \|\boldsymbol{X}^- \cdot \vec{\alpha} - \boldsymbol{X}^- \cdot \vec{\beta}\|^2 = \vec{\alpha}^\top \cdot \boldsymbol{D}^2 \cdot \vec{\beta} - \frac{1}{2}\vec{\alpha}^\top \cdot \boldsymbol{D}^2 \cdot \vec{\alpha} - \frac{1}{2}\vec{\beta}^\top \cdot \boldsymbol{D}^2 \cdot \vec{\beta} \tag{2.8}$$

Further, for any $x \in \mathcal{X}$ it holds:

$$\|\phi^+(x) - \boldsymbol{X}^+ \cdot \vec{\alpha}\|^2 - \|\phi^-(x) - \boldsymbol{X}^- \cdot \vec{\alpha}\|^2 = \sum_{i=1}^M \alpha_i \cdot d(x, x_i)^2 - \frac{1}{2}\vec{\alpha}^\top \cdot \boldsymbol{D}^2 \cdot \vec{\alpha} \tag{2.9}$$

If d is Euclidean with spatial mapping $\phi : \mathcal{X} \to \mathbb{R}^m$, then let $\boldsymbol{X} := \big(\phi(x_1), \ldots, \phi(x_M)\big) \in \mathbb{R}^{m \times M}$. It holds:

$$\|\boldsymbol{X} \cdot \vec{\alpha} - \boldsymbol{X} \cdot \vec{\beta}\|^2 = \vec{\alpha}^\top \cdot \boldsymbol{D}^2 \cdot \vec{\beta} - \frac{1}{2}\vec{\alpha}^\top \cdot \boldsymbol{D}^2 \cdot \vec{\alpha} - \frac{1}{2}\vec{\beta}^\top \cdot \boldsymbol{D}^2 \cdot \vec{\beta} \tag{2.10}$$

Further, for any $x \in \mathcal{X}$ it holds:

$$\|\phi(x) - \boldsymbol{X} \cdot \vec{\alpha}\|^2 = \sum_{i=1}^{M} \alpha_i \cdot d(x, x_i)^2 - \frac{1}{2}\vec{\alpha}^\top \cdot \boldsymbol{D}^2 \cdot \vec{\alpha} \tag{2.11}$$

Proof. Refer to Theorem 1 by Hammer and Hasenfuss (2010). A version of the proof adapted to our notation here is shown in Appendix A.3. □

Via this trick, one can construct machine learning methods that perform inferences solely based on a given pseudo-Euclidean or Euclidean distance such as relational neural gas (Hammer and Hasenfuss 2007; Hammer and Hasenfuss 2010), relational generative topographic mapping (Gisbrecht, Mokbel, and Hammer 2010), or relational generalized learning vector quantization (RGLVQ, Hammer, D. Hofmann, et al. 2014, also refer to Section 2.5.3). In our work, we extend this branch of machine learning by providing a novel time series prediction mechanism based on pseudo-Euclidean distances in Chapter 5, and a edit distance inversion scheme in Chapter 6.

Now that we have covered the basic notions of kernels, distances, and their relations, we go into more detail regarding kernels and distances for structured data. We first cover structure kernels and then go on to edit distances on structured data.

2.2 KERNELS FOR STRUCTURED DATA

One can construct kernels for structured data in two ways, either by explicitly constructing the spatial mapping ϕ, or by leaving that mapping implicit (Da San Martino and Sperduti 2010; T. Hofmann, Schölkopf, and Smola 2008). The most straightforward form of explicit kernels are histogram kernels, which build a histogram over features of the structured datum x and use those as vectorial representation $\phi(x)$. Examples include histograms over the lengths of shortest paths in a graph (Borgwardt and Kriegel 2005), histograms over subtree types and their position (Aiolli, Martino, and Sperduti 2015), and histograms over hidden states of a Markov model trained on the structured datum (Bacciu, Errica, and Micheli 2018).

Another approach to explicit kernels relies on learning the spatial mapping ϕ, for example via a neural network (Bacciu, Gallicchio, and Micheli 2016; W.-b. Huang et al. 2015; Mehrkanoon and Suykens 2018; Yanardag and Vishwanathan 2015; Z. Yang et al. 2015). This relates kernels to the field of *representation learning* for structured data, which has received heightened attention in recent years (Bengio, Courville, and Vincent 2013; LeCun, Bengio, and Hinton 2015). For example, we can learn vectorial representations of sequential data via recurrent neural networks (Cho et al. 2014; Chung et al. 2015; Hochreiter and Schmidhuber 1997; Jaeger and Haas 2004; Sutskever, Vinyals, and Q. V. Le 2014), we can learn tree representations via recursive neural networks (Gallicchio and Micheli 2013; Irsoy and Cardie 2014; Pollack 1990; Socher, Perelygin, et al. 2013; Sperduti and Starita 1997), and we can learn graph representations via recurrent, recursive, or convolutional networks on graphs (Bacciu, Errica, and Micheli 2018; Gallicchio and Micheli 2010; Garcia Duran and Niepert 2017; Hamilton, Ying, and Leskovec 2017).

In terms of implicit kernels, a popular strategy involves representing a structured datum in terms of constituent parts and constructing an overall kernel as a sum over kernels between the constituents, such as path and walk kernels (Borgwardt and Kriegel 2005; Da San Martino and Sperduti 2010; Feragen et al. 2013) or Weisfeiler-Lehman kernels (Shervashidze et al. 2011). Once a structure kernel has been constructed, it is also possible to combine multiple kernels in linear combinations with non-negative weights, which has been dubbed *multiple kernel learning* (Aiolli and Donini 2015; Gönen and Alpaydın 2011). Alternatively, one can perform an approach similar to metric learning by adjusting the parameters of a kernel to the data at hand, as has been done for biological sequence alignment kernels (Saigo, Vert, and Akutsu 2006).

Note that only few kernels permit intuitive interpretation. In particular, the subtree kernels of Aiolli, Martino, and Sperduti (2015) could be interpreted in terms of the subtrees that are contained in both input trees, and the alignment kernel of Saigo, Vert, and Akutsu (2006) permits to pinpoint the elements that are different and equal in both input sequences. However, the latter is only possible because the kernel is constructed based on an edit distance. Indeed, edit distances have the distinct advantage that they are not only interpretable, but *actionable*, in the sense that an edit distance tells us precisely what we need to change to reduce the edit distance between two structured data. Therefore, we focus on edit distances in our work.

2.3 EDIT DISTANCES

An edit distance between two structured data $\bar{x}$ and $\bar{y}$ is defined as the effort needed to transform $\bar{x}$ into $\bar{y}$. More precisely, an edit distance is defined as the cost of the cheapest edit script which transforms $\bar{x}$ into $\bar{y}$, and different notions of edit scripts yield different kinds of edit distances. The first works on edit distances are by Levenshtein (1965) as well as Damerau (1964) who independently devised a simple distance measure to count the number of spelling mistakes in a written sentence by defining it as the number of characters that have to be deleted, inserted, or changed to arrive at the correct version. Later, multiple researchers discovered dynamic programming algorithms to compute these notions of distance efficiently (Navarro 2001). T. F. Smith and Waterman (1981) and Gotoh (1982) have later extended this basic work to compute edit distances between RNA, DNA, and protein sequences in terms of their amino acid notation. Further, Tai (1979) and Zhang and Shasha (1989) have provided edit distance versions for trees. Indeed, ordered trees and ordered directed acyclic graphs are the most complex data structure for which we can compute the edit distance efficiently as the edit distance for unordered trees and for graphs with cycles are provably NP-hard (Zhang, Statman, and Shasha 1992; Zeng et al. 2009).

In this section, we describe edit distance approaches for sequences, trees, and graphs, with a special focus on edit distances on sequences and trees, because these are efficiently computable.

2.3.1 *Sequence Edit Distance*

We begin our description of sequence edit distances by formally defining sequences, edits over sequences, cost functions, and finally edit distances. While these definitions capture the general intuition behind sequence edit distances, they are insufficient to

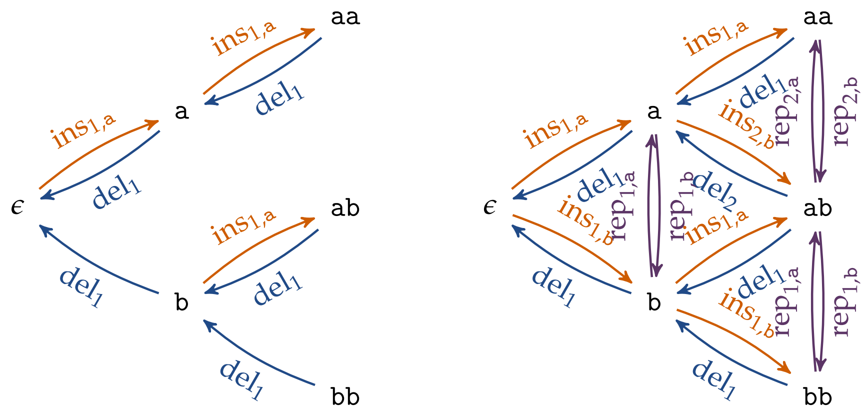


Figure 2.1: Left: A graphical representation of the edit set $\Delta = \{\text{del}_1, \text{ins}_{1,\mathtt{a}}\}$ over the alphabet $\mathcal{A} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. All possible sequences over $\mathcal{A}$ are nodes of the graph, and two sequences are connected if a sequence edit in Δ exists that transforms the first sequence into the second. Right: A similar graphical representation for the edit set over the signature $\mathcal{S}_{\text{ALI}} = (\{\text{del}\}, \{\text{rep}\}, \{\text{ins}\})$.

derive efficient algorithms. Therefore, we introduce algebraic dynamic programming (Giegerich, Meyer, and Steffen 2004) as an alternative formalism to describe sequence edit distances that is strong enough to yield results regarding metric properties and efficient computations. Note that we go into some detail regarding sequence edit distances at this point because we later build upon our concepts and notations to *learn* these edit distances in Chapter 3.

Definition 2.5 (Alphabets, Sequences, Sequence Edits, Edit Sets, Edit Scripts)**.** Let $\mathcal{A}$ be some arbitrary set. We call such a set an *alphabet*. We define a *sequence* over $\mathcal{A}$ as a finite list of elements $\bar{x} = x_1 \dots x_m$ from $\mathcal{A}$. We call m the *length* of $\bar{x}$, denoted as $|\bar{x}|$. We denote the *empty list* as ϵ. We denote the set of all possible sequences over alphabet $\mathcal{A}$ as $\mathcal{A}^*$.

We define a *sequence edit* as a function $\delta : \mathcal{A}^* \to \mathcal{A}^*$. We call a set Δ of sequence edits an *edit set*. We define an *edit script* over Δ as a sequence over Δ. We define the application $\bar{\delta}(\bar{x})$ of an edit script $\bar{\delta} = \delta_1 \dots \delta_T \in \Delta^*$ to a sequence $\bar{x}$ as the function composition $\delta_T \circ \dots \circ \delta_1(\bar{x})$, where $\delta \circ \delta'(\bar{x}) := \delta(\delta'(\bar{x}))$. If $\bar{\delta} = \epsilon$, we define $\bar{\delta}(\bar{x}) := \bar{x}$.

As an example, consider the alphabet $\mathcal{A} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. Then, ϵ, a, abc, and aaa are all sequences over $\mathcal{A}$. Now, consider the sequence edit $\text{del}_1 : \mathcal{A}^* \to \mathcal{A}^*$, which we define as $\text{del}_1(x_1 \dots x_m) := x_2 \dots x_m$, and $\text{del}_1(\epsilon) = \epsilon$. Applying del_1 to the sequence abc results in $\text{del}_1(\mathtt{abc}) = \mathtt{bc}$. Accordingly, the edit script del_1del_1 results in $\text{del}_1\text{del}_1(\mathtt{abc}) = \mathtt{c}$. Conversely, consider the sequence edit $\text{ins}_{1,\mathtt{a}} : \mathcal{A}^* \to \mathcal{A}^*$, which we define as $\text{ins}_{1,\mathtt{a}}(\bar{x}) = \mathtt{a}\bar{x}$. Applying $\text{ins}_{1,\mathtt{a}}$ the sequence abc results in $\text{ins}_{1,\mathtt{a}}(\mathtt{abc}) = \mathtt{aabc}$. The set $\Delta = \{\text{del}_1, \text{ins}_{1,\mathtt{a}}\}$ is an edit set.

We can interpret an edit set over some alphabet $\mathcal{A}$ in terms of a graph $\mathcal{G} = (V, E)$ by setting the nodes as $V = \mathcal{A}^*$ and constructing an edge $(\bar{x}, \bar{y}) \in E$ if and only if there exists a sequence edit $\delta \in \Delta$ such that $\delta(\bar{x}) = \bar{y}$. This graphical view is particularly insightful in intelligent tutoring systems, where we can interpret the graph as the space of all possible states a student could visit on their way to a solution of a learning task. We therefore cover this interpretation in more detail in Chapter 6 (in particular, refer to Definition 6.2).

Figure 2.1 (left) shows an excerpt of this graph for our example above. The sequence edit distance is the shortest path distance in this graph if we set the length of all edges to the values of a cost function, which we define as follows.

Definition 2.6 (Cost Function, Sequence Edit Distance)**.** Let $\mathcal{A}$ be an alphabet and let Δ be an edit set over $\mathcal{A}$. Then, we define a *cost function* over Δ as a function $c : \Delta \times \mathcal{A}^* \to \mathbb{R}$. We call $c(\delta, \bar{x})$ the *cost* of applying δ to $\bar{x}$. Accordingly, we define the cost of applying an edit script $\bar{\delta} = \delta_1 \ldots \delta_T$ to $\bar{x}$ recursively as $c(\bar{\delta}, \bar{x}) := c(\delta_1, \bar{x}) + c(\delta_2 \ldots \delta_T, \delta_1(\bar{x}))$ with the base case $c(\epsilon, \bar{x}) = 0$.

We define the edit distance $d_{\Delta,c}$ according to Δ and c as the following function.

$$d_{\Delta,c} : \mathcal{A}^* \times \mathcal{A}^* \to \mathbb{R}$$

$$d_{\Delta,c}(\bar{x}, \bar{y}) := \min_{\bar{\delta} \in \Delta^*} \left\{ c(\bar{\delta}, \bar{x}) \middle| \bar{\delta}(\bar{x}) = \bar{y} \right\} \tag{2.12}$$

In other words: The edit distance between $\bar{x}$ and $\bar{y}$ is the cost of the cheapest edit script transforming $\bar{x}$ to $\bar{y}$. Consider the edit set $\Delta = \{\text{del}_1, \text{ins}_{1,\texttt{a}}\}$ above in combination with the cost function $c(\delta, \bar{x}) = 1$, independent of the input. Then, we obtain $d_{\Delta,c}(\epsilon, \texttt{a}) = 1$, $d_{\Delta,c}(\epsilon, \texttt{aa}) = 2$, and $d_{\Delta,c}(\texttt{bb}, \texttt{ab}) = 2$.

While conceptually insightful, the definition of edit distances via an edit set and a cost function has severe practical limitations. First, an edit set needs to be infinitely large if we wish to address arbitrarily long sequences, which poses a challenge in definition. Second, the concepts of an edit set and a cost function are too general to permit conclusions regarding metric properties. For example, our edit distance above is not metric because it is not symmetric. However, we require certainty about self-equality and symmetry in order to ensure that a edit distance is pseudo-Euclidean. Third, the space of all possible edit scripts over an infinite edit set is not efficiently searchable, preventing us from computing the edit distance in polynomial time.

As such, we sorely need an alternative formalism to express a subclass of edit distances that are efficiently computable, and this subclass needs to be expressive enough to include all edit distances that are interesting to us. As it turns out, the framework of *algebraic dynamic programming* is exactly what we need.

2.3.2 *Algebraic Dynamic Programming*

Algebraic dynamic programming (ADP) has been introduced by Giegerich, Meyer, and Steffen (2004) as a *discipline of dynamic programming over sequence data*. In particular, the authors suggest to formalize potential solutions for a problem over sequential data as trees, generated by a regular tree grammar, and to find an optimal solution by essentially parsing the problem input via the grammar (Giegerich, Meyer, and Steffen 2004). Note that this approach is highly general and encompasses diverse sequential problems far beyond edit distance computation, such as optimal RNA folding, hidden Markov model inference or scoring of phylogenetic trees (Siederdissen, Prohaska, and Stadler 2015). In this section, we focus particularly on edit distances and simplify the general ADP theory for this purpose. Still, all our definitions follow strictly from the general case as described by Giegerich, Meyer, and Steffen (2004).

Note that we utilize the ADP formulation as basis for sequence edit distance learning later in Chapter 3. We also show metric properties and efficient computability there.

In this section, we focus on definitions. In particular, we introduce three ingredients which suffice to specify any typical sequence edit distance in the literature, namely signatures, which capture the kinds of edits that can be applied, algebrae, which capture how expensive these kinds of edits are, and edit tree grammars, which specify how edits can be combined into edit scripts. First, we begin with signatures.

Definition 2.7 (Signature, Signature Edit Set)**.** We define a *signature* $\mathcal{S}$ as a triple of finite sets $\mathcal{S} = (\text{Del}, \text{Rep}, \text{Ins})$, which are pairwise disjoint, that is $\text{Del} \cap \text{Rep} = \text{Del} \cap \text{Ins} = \text{Rep} \cap \text{Ins} = \emptyset$. We call $\mathcal{S}$ *non-trivial* if neither Del nor Ins are empty.

Let $\mathcal{A}$ be an alphabet and $\mathcal{S} = (\text{Del}, \text{Rep}, \text{Ins})$ be a signature. Then, we define for each element $\text{del} \in \text{Del}$ and each natural number $i \in \mathbb{N}$ the function $\text{del}_i : \mathcal{A}^* \to \mathcal{A}^*$ as $\text{del}_i(x_1 \ldots x_m) := x_1 \ldots x_{i-1} x_{i+1} \ldots x_m$ if $i \leq m$, and $\text{del}_i(\bar{x}) := \bar{x}$ if $i > m$.

For each element $\text{rep} \in \text{Rep}$, each element $y \in \mathcal{A}$, and each natural number $i \in \mathbb{N}$, we define the function $\text{rep}_{i,y} : \mathcal{A}^* \to \mathcal{A}^*$ as $\text{rep}_{i,y}(x_1 \ldots x_m) := x_1 \ldots x_{i-1} y x_{i+1} \ldots x_m$ if $i \leq m$, and $\text{rep}_{i,y}(\bar{x}) := \bar{x}$ if $i > m$.

Finally, for each element $\text{ins} \in \text{Ins}$, each element $y \in \mathcal{A}$, and each natural number $i \in \mathbb{N}$, we define the function $\text{ins}_{i,y} : \mathcal{A}^* \to \mathcal{A}^*$ as $\text{ins}_{i,y}(x_1 \ldots x_m) := x_1 \ldots x_{i-1} y x_i \ldots x_m$ if $i \leq m+1$, and $\text{ins}_{i,y}(\bar{x}) := \bar{x}$ if $i > m+1$.

We define the edit set $\Delta_{\mathcal{S},\mathcal{A}}$ with respect to the signature $\mathcal{S} = (\text{Del}, \text{Rep}, \text{Ins})$, and the alphabet $\mathcal{A}$ as follows.

$$\begin{aligned}\Delta_{\mathcal{S},\mathcal{A}} =& \{\text{del}_i | \text{del} \in \text{Del}, i \in \mathbb{N}\} \cup \\ & \{\text{rep}_{i,y} | \text{rep} \in \text{Rep}, i \in \mathbb{N}, y \in \mathcal{A}\} \cup \\ & \{\text{ins}_{i,y} | \text{ins} \in \text{Rep}, i \in \mathbb{N}, y \in \mathcal{A}\}\end{aligned} \tag{2.13}$$

As an example, consider one of the simplest non-trivial signatures, $\text{ALI} = (\{\text{del}\}, \{\text{rep}\}, \{\text{ins}\})$, which contains one kind of deletion, replacement, and insertion respectively and corresponds to the string edit distance of Levenshtein (1965). An excerpt of the graphical representation of the edit set $\Delta_{\text{ALI},\{\mathsf{a},\mathsf{b},\mathsf{c}\}}$ is shown in Figure 2.1 (right). Note that, as a user of the framework, we only need to specify a small signature $\mathcal{S}$, and the infinitely large edit set $\Delta_{\mathcal{S},\mathcal{A}}$ follows automatically. Also note that we can re-use the same signature for arbitrary alphabets, which simplifies specification.

Now that we have specified an edit set, we only need a cost function to obtain an edit distance. Following the ADP framework, we generate a cost function based on the signature via the vehicle of an algebra.

Definition 2.8 (Algebra, Algebra Cost Function)**.** Let $\mathcal{A}$ be an alphabet, let $\mathcal{S} = (\text{Del}, \text{Rep}, \text{Ins})$ be a signature, let $(\mathcal{A} \to \mathbb{R})$ denote the set of functions mapping from $\mathcal{A}$ to the real numbers $\mathbb{R}$, and let $(\mathcal{A} \times \mathcal{A} \to \mathbb{R})$ denote the set of functions mapping from $\mathcal{A} \times \mathcal{A}$ to the real numbers $\mathbb{R}$.

Then, we define an *algebra* $\mathcal{F}$ over $\mathcal{S}$ and $\mathcal{A}$ as a triple of functions $\mathcal{F}_{\mathcal{S},\mathcal{A}} = (\mathcal{F}_{\text{Del}}, \mathcal{F}_{\text{Rep}}, \mathcal{F}_{\text{Ins}})$, where $\mathcal{F}_{\text{Del}} : \text{Del} \to (\mathcal{A} \to \mathbb{R})$, $\mathcal{F}_{\text{Rep}} : \text{Rep} \to (\mathcal{A} \times \mathcal{A} \to \mathbb{R})$, and $\mathcal{F}_{\text{Ins}} : \text{Ins} \to (\mathcal{A} \to \mathbb{R})$.

As a shorthand, we denote the function $\mathcal{F}_{\text{Del}}(\text{del})$ as c_{del} for all $\text{del} \in \text{Del}$, we denote $\mathcal{F}_{\text{Rep}}(\text{rep})$ as c_{rep} for all $\text{rep} \in \text{Rep}$, and we denote $\mathcal{F}_{\text{Ins}}(\text{ins})$ as c_{ins} for all $\text{ins} \in \text{Ins}$.

We define the cost function $c_{\mathcal{F}}$ with respect to an algebra $\mathcal{F} = (\mathcal{F}_{\text{Del}}, \mathcal{F}_{\text{Rep}}, \mathcal{F}_{\text{Ins}})$ as the following cost function over $\Delta_{\mathcal{S},\mathcal{A}}$.

$$c_{\mathcal{F}}(\delta, \bar{x}) := \begin{cases} c_{\text{del}}(x_i) & \text{if} \quad \delta = \text{del}_i, i \leq |\bar{x}| \\ c_{\text{rep}}(x_i, y) & \text{if} \quad \delta = \text{rep}_{i,y}, i \leq |\bar{x}| \\ c_{\text{ins}}(y) & \text{if} \quad \delta = \text{ins}_{i,y}, i \leq |\bar{x}| + 1 \\ 0 & \text{otherwise} \end{cases} \tag{2.14}$$

As a notational shorthand, we denote the edit distance $d_{\Delta_{\mathcal{S},\mathcal{A}}, c_{\mathcal{F}}}$ as $d_{\mathcal{S},\mathcal{F}}$.

As an example, consider the standard string edit distance of Levenshtein (1965), which corresponds to the signature $\mathcal{S}_{\text{ALI}} = (\{\text{del}\}, \{\text{rep}\}, \{\text{ins}\})$, and the algebra $\mathcal{F}_{\text{ALI}}$ with the functions

$$c_{\text{del}}(x) := c_{\text{ins}}(x) := 1 \quad \text{and} \quad c_{\text{rep}}(x, y) := \begin{cases} 1 & \text{if } x \neq y \\ 0 & \text{if } x = y \end{cases} \quad \forall x, y \in \mathcal{A} \tag{2.15}$$

An advantage in specifying a cost function via an algebra is that we only need to specify the cost of edit types, which then automatically generalizes over the entire, infinitely large edit set $\Delta_{\mathcal{S},\mathcal{A}}$.

An issue with the formalism of edit scripts is that it is highly redundant, that is, edit scripts can make detours before arriving at their final result. For example, the two edit scripts del_1 and $\text{ins}_{1,\text{a}}\text{rep}_{1,\text{b}}\text{del}_2\text{del}_1$ have exactly the same output for every possible input sequence, but the latter makes detours, namely inserting the letter a, replacing it with b, and deleting it again, in addition to performing the actual action, namely deleting the first character in the input sequence.

To express only those edit scripts that avoid such detours, we introduce the script tree concept.

Definition 2.9 (Script Trees, Yield, Tree Cost)**.** Let $\mathcal{A}$ be an alphabet with $\$, \text{match} \notin \mathcal{A}$, and let $\mathcal{S} = (\text{Del}, \text{Rep}, \text{Ins})$ be a signature with $\$, \text{match} \notin \text{Del} \cup \text{Rep} \cup \text{Ins}$. Then, we define a *script tree* $\tilde{\delta}$ over $\mathcal{S}$ and $\mathcal{A}$ as one of the following.

$$\begin{aligned} &\tilde{\delta} = \$, \\ &\tilde{\delta} = \text{match}(x, \tilde{\delta}', x) && \text{where } x \in \mathcal{A}, \text{ and } \tilde{\delta}' \text{ is a script tree,} \\ &\tilde{\delta} = \text{del}(x, \tilde{\delta}') && \text{where del} \in \text{Del}, x \in \mathcal{A}, \text{ and } \tilde{\delta}' \text{ is a script tree,} \\ &\tilde{\delta} = \text{rep}(x, \tilde{\delta}', y) && \text{where rep} \in \text{Rep}, x, y \in \mathcal{A}, \text{ and } \tilde{\delta}' \text{ is a script tree, or} \\ &\tilde{\delta} = \text{ins}(\tilde{\delta}', y) && \text{where ins} \in \text{Ins}, y \in \mathcal{A}, \text{ and } \tilde{\delta}' \text{ is a script tree.} \end{aligned}$$

We define the set of all possible script trees over $\mathcal{S}$ and $\mathcal{A}$ as $\mathcal{T}(\mathcal{S}, \mathcal{A})$.

Let $\tilde{\delta}$ be a script tree over $\mathcal{S}$ and $\mathcal{A}$. Then, we define the yield $\mathcal{Y}(\tilde{\delta}) \in \mathcal{A}^* \times \mathcal{A}^*$ of $\tilde{\delta}$ as follows.

$$\mathcal{Y}(\tilde{\delta}) := \begin{cases} (\epsilon, \epsilon) & \text{if} \quad \tilde{\delta} = \$ \\ (x\bar{x}, x\bar{y}) & \text{if} \quad \tilde{\delta} = \text{match}(x, \tilde{\delta}', x) \quad \text{and } (\bar{x}, \bar{y}) = \mathcal{Y}(\tilde{\delta}') \\ (x\bar{x}, \bar{y}) & \text{if} \quad \tilde{\delta} = \text{del}(x, \tilde{\delta}') \quad \text{and } (\bar{x}, \bar{y}) = \mathcal{Y}(\tilde{\delta}') \\ (x\bar{x}, y\bar{y}) & \text{if} \quad \tilde{\delta} = \text{rep}(x, \tilde{\delta}', y) \quad \text{and } (\bar{x}, \bar{y}) = \mathcal{Y}(\tilde{\delta}') \\ (\bar{x}, y\bar{y}) & \text{if} \quad \tilde{\delta} = \text{ins}(\tilde{\delta}', y) \quad \text{and } (\bar{x}, \bar{y}) = \mathcal{Y}(\tilde{\delta}') \end{cases}$$

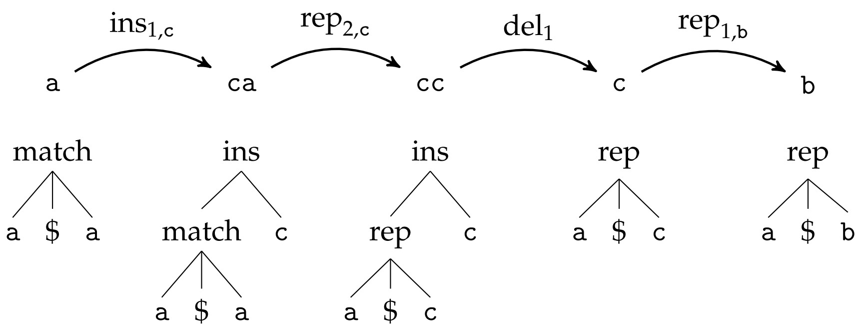


Figure 2.2: An example for the translation of an edit script to a script tree. The top row displays the sequence edits in the edit script which successively convert the sequence $\bar{x} = \texttt{a}$ into the sequence $\bar{y} = \texttt{b}$. The bottom row shows the script tree corresponding to the partial edit script up to the point of the sequence at the top.

Further, we define the *size* $|\tilde{\delta}| \in \mathbb{N}_0$ of $\tilde{\delta}$ as follows.

$$|\tilde{\delta}| := \begin{cases} 0 & \text{if } \tilde{\delta} = \$ \\ 1 + |\tilde{\delta}'| & \text{if } \exists \tilde{\delta}' : \tilde{\delta} \in \{\text{match}(x, \tilde{\delta}', x), \text{del}(x, \tilde{\delta}'), \text{rep}(x, \tilde{\delta}', y), \text{ins}(\tilde{\delta}', y)\} \end{cases}$$

Let $\mathcal{F}$ be an algebra over $\mathcal{S}$ and $\mathcal{A}$. Then, we define the *cost* $c_{\mathcal{F}}(\tilde{\delta})$ of $\tilde{\delta}$ according to $\mathcal{F}$ as follows.

$$c_{\mathcal{F}}(\tilde{\delta}) := \begin{cases} 0 & \text{if } \tilde{\delta} = \$ \\ c_{\mathcal{F}}(\tilde{\delta}') & \text{if } \tilde{\delta} = \text{match}(x, \tilde{\delta}', x) \\ c_{\text{del}}(x) + c_{\mathcal{F}}(\tilde{\delta}') & \text{if } \tilde{\delta} = \text{del}(x, \tilde{\delta}') \\ c_{\text{rep}}(x, y) + c_{\mathcal{F}}(\tilde{\delta}') & \text{if } \tilde{\delta} = \text{rep}(x, \tilde{\delta}', y) \\ c_{\text{ins}}(y) + c_{\mathcal{F}}(\tilde{\delta}') & \text{if } \tilde{\delta} = \text{ins}(\tilde{\delta}', y) \end{cases}$$

As an example, consider the script tree $\tilde{\delta} = \text{del}(\texttt{a}, \text{ins}(\$, \texttt{b}))$ over the signature $\mathcal{S}_{\text{ALI}} = (\{\text{del}\}, \{\text{rep}\}, \{\text{ins}\})$ and the alphabet $\mathcal{A} = \{\texttt{a}, \texttt{b}\}$. The yield of this tree is

$$\mathcal{Y}(\tilde{\delta}) = \Big(\texttt{a}\mathcal{Y}_1\big(\text{ins}(\$, \texttt{b})\big), \mathcal{Y}_2\big(\text{ins}(\$, \texttt{b})\big)\Big) = \Big(\texttt{a}\mathcal{Y}_1(\$), \texttt{b}\mathcal{Y}_2(\$)\Big) = (\texttt{a}, \texttt{b}).$$

The size of the tree is $|\tilde{\delta}| = 1 + |\text{ins}(\$, \texttt{b})| = 2 + |\$| = 2$. Finally, the cost of the tree according to algebra $\mathcal{F}_{\text{ALI}}$ from above is given as

$$c_{\mathcal{F}}(\tilde{\delta}) = c_{\text{del}}(\texttt{a}) + c_{\mathcal{F}}\big(\text{ins}(\$, \texttt{b})\big) = c_{\text{del}}(\texttt{a}) + c_{\text{ins}}(\texttt{b}) + c_{\mathcal{F}}(\$) = c_{\text{del}}(\texttt{a}) + c_{\text{ins}}(\texttt{b}).$$

Intuitively, the script tree has the purpose to jointly represent some sequence $\bar{x}$, some edit script $\bar{\delta}$, and the resulting sequence $\bar{\delta}(\bar{x})$. As mentioned above, however, the relation between edit scripts and script trees is not one-to-one, because script trees represent only edit scripts which avoid detours - at least extreme detours where we insert symbols that are not present in the target sequence. Indeed, omitting such detours ensures that for any two sequences $\bar{x}$ and $\bar{y}$ the search space of possible script trees $\tilde{\delta}$ with the yield $\mathcal{Y}(\tilde{\delta}) = (\bar{x}, \bar{y})$ is guaranteed to be finite, even though the set of edit scripts which transform $\bar{x}$ to $\bar{y}$ may well be infinite. This drastic limitation in the search space also enables us to compute the resulting edit distances efficiently (refer to Chapter 3).

A final limitation of our framework until now is that we can not express additional contraints on the edit distance. Such constraints occur in extensions of the standard edit distance, such as the local alignment distance of T. F. Smith and Waterman (1981), which permits cheaper deletions or insertions at the end of the input sequences, but not before, or the affine alignment distance of Gotoh (1982), which permits cheaper deletions or insertions if they occur in bulk. We can incorporate such constraints in form of an edit tree grammar.

Definition 2.10 (Edit Tree Grammar, Tree Language, Grammar Edit Distance)**.** Let $\mathcal{S} = (\mathrm{Del}, \mathrm{Rep}, \mathrm{Ins})$ be a signature with $\$, \mathrm{match} \notin \mathrm{Del} \cup \mathrm{Rep} \cup \mathrm{Ins}$. Then, we define an *edit tree grammar* $\mathcal{G}$ as a quartuple $\mathcal{G} = (\Phi, \mathcal{S}, \mathcal{R}, \mathrm{S})$, where Φ is a finite set, which we call *nonterminal symbols*, such that $\Phi \cap (\mathrm{Del} \cup \mathrm{Rep} \cup \mathrm{Ins} \cup \{\mathrm{match}, \$\}) = \emptyset$, $\mathrm{S} \in \Phi$, and $\mathcal{R}$ is a finite set of so-called *production rules* of the form $A ::= \delta B$ or the form $A ::= \$$, where $A, B \in \Phi$ and $\delta \in \mathrm{Del} \cup \mathrm{Rep} \cup \mathrm{Ins} \cup \{\mathrm{match}\}$.

Per convention, we denote multiple production rules $A ::= \delta_1 B_1$, ..., $A ::= \delta_T B_T$, $A ::= \$$ with the same left-hand side A as $A ::= \delta_1 B_1 | \ldots | \delta_T B_T | \$$.

Let $\mathcal{A}$ be an alphabet, let $A, B \in \Phi$, $x, y \in \mathcal{A}$, $\mathrm{del} \in \mathrm{Del}$, $\mathrm{rep} \in \mathrm{Rep}$, and $\mathrm{ins} \in \mathrm{Ins}$. We say that $\$$ can be *derived in one step* from A via $\mathcal{G}$, denoted as $A \rightarrow^1_{\mathcal{G}} \$$, if the production rule $A ::= \$$ is in $\mathcal{R}$. Similarly, we say that $A \rightarrow^1_{\mathcal{G}} \mathrm{match}(x, B, x)$ if $A ::= \mathrm{match}B \in \mathcal{R}$, we say that $A \rightarrow^1_{\mathcal{G}} \mathrm{del}(x, B)$ if $A ::= \mathrm{del}B \in \mathcal{R}$, we say that $A \rightarrow^1_{\mathcal{G}} \mathrm{rep}(x, B, y)$ if $A ::= \mathrm{rep}(x, B, y) \in \mathcal{R}$, and we say that $A \rightarrow^1_{\mathcal{G}} \mathrm{ins}(B, y)$ if $A ::= \mathrm{ins}B \in \mathcal{R}$.

We say that an expression $\tilde{\delta}$ can be *derived in* $T+1$ *steps* from $A \in \Phi$ for $T \in \mathbb{N}$, denoted as $A \rightarrow^{T+1}_{\mathcal{G}} \tilde{\delta}$ if one of the following cases holds.

$\tilde{\delta} = \mathrm{match}(x, \tilde{\delta}', x)$ for some expression $\tilde{\delta}'$, and there exists a $B \in \Phi$ such that $A \rightarrow^1_{\mathcal{G}} \mathrm{match}(x, B, x)$, as well as $B \rightarrow^T_{\mathcal{G}} \tilde{\delta}'$.

$\tilde{\delta} = \mathrm{del}(x, \tilde{\delta}')$ for some expression $\tilde{\delta}'$, and there exists a $B \in \Phi$ such that $A \rightarrow^1_{\mathcal{G}} \mathrm{del}(x, B)$, as well as $B \rightarrow^T_{\mathcal{G}} \tilde{\delta}'$.

$\tilde{\delta} = \mathrm{rep}(x, \tilde{\delta}', y)$ for some expression $\tilde{\delta}'$, and there exists a $B \in \Phi$ such that $A \rightarrow^1_{\mathcal{G}} \mathrm{rep}(x, B, y)$, as well as $B \rightarrow^T_{\mathcal{G}} \tilde{\delta}'$.

$\tilde{\delta} = \mathrm{ins}(\tilde{\delta}', y)$ for some expression $\tilde{\delta}'$, and there exists a $B \in \Phi$ such that $A \rightarrow^1_{\mathcal{G}} \mathrm{ins}(B, y)$, as well as $B \rightarrow^T_{\mathcal{G}} \tilde{\delta}'$.

We say that $\tilde{\delta}$ can be *derived in arbitrarily many steps* from A, denoted as $A \rightarrow^*_{\mathcal{G}} \tilde{\delta}$, if there exists any $T \in \mathbb{N}$ such that $A \rightarrow^T_{\mathcal{G}} \tilde{\delta}$. We define the *tree language* of $\mathcal{G}$ with respect to $\mathcal{A}$ as follows.

$$\mathcal{L}(\mathcal{G}, \mathcal{A}) := \{\tilde{\delta} \in \mathcal{T}(\mathcal{S}, \mathcal{A}) | \mathrm{S} \rightarrow^*_{\mathcal{G}} \tilde{\delta}\}$$

Let $\mathcal{F}$ be an algebra over $\mathcal{S}$ and $\mathcal{A}$, and let $\bar{x}, \bar{y} \in \mathcal{A}^*$. Then, we define the edit distance between $\bar{x}$ and $\bar{y}$ with respect to $\mathcal{G}$ and $\mathcal{F}$ as follows.

$$d_{\mathcal{G},\mathcal{F}}(\bar{x}, \bar{y}) := \min_{\tilde{\delta} \in \mathcal{L}(\mathcal{G}, \mathcal{A})} \{c_{\mathcal{F}}(\tilde{\delta}) | \mathcal{Y}(\tilde{\delta}) = (\bar{x}, \bar{y})\}$$

Note that including edit tree grammars into the formalism does not restrict expressivity. For every signature $\mathcal{S} = (\text{Del}, \text{Rep}, \text{Ins})$ and every alphabet $\mathcal{A}$ we can recover the set $\mathcal{T}(\mathcal{A}, \mathcal{S})$ as the tree language of the trivial edit tree grammar

$$\mathcal{G}_{\mathcal{S}} = (\{\text{S}\}, \mathcal{S}, \{\text{S} ::= \$\} \cup \{\text{S} ::= \delta\text{S} | \delta \in \text{Del} \cup \text{Rep} \cup \text{Ins} \cup \{\text{match}\}\}, \text{S}).$$

As an example, consider the signature $\mathcal{S}_{\text{ALI}} = (\{\text{del}\}, \{\text{rep}\}, \{\text{ins}\})$ and the following edit tree grammar $\mathcal{G}_{\text{ALI}}$.

$$\mathcal{G}_{\text{ALI}} = (\{\text{A}\}, \mathcal{S}_{\text{ALI}}, \mathcal{R}, \text{A}) \quad \text{where} \quad \mathcal{R} = \{\text{A} ::= \text{matchA}|\text{delA}|\text{repA}|\text{insA}|\$\} \tag{2.16}$$

For this edit tree grammar and any alphabet $\mathcal{A}$ it holds: $\mathcal{L}(\mathcal{G}_{\text{ALI}}, \mathcal{A}) = \mathcal{T}(\mathcal{A}, \mathcal{S}_{\text{ALI}})$.

Now, consider the two sequences $\bar{x} = \texttt{a}$ and $\bar{y} = \texttt{b}$ over the alphabet $\mathcal{A} = \{\texttt{a}, \texttt{b}\}$ and consider the algebra $\mathcal{F}_{\text{ALI}}$ from Equation 2.16. To compute the edit distance between $\bar{x}$ and $\bar{y}$ with respect to $\mathcal{G}_{\text{ALI}}$ and $\mathcal{F}_{\text{ALI}}$, we need to consider all script trees $\tilde{\delta}$ that can be generated via $\mathcal{G}_{\text{ALI}}$ and have the yield $\mathcal{Y}(\tilde{\delta}) = (\bar{x}, \bar{y}) = (\texttt{a}, \texttt{b})$. These are only the script trees $\text{del}(\texttt{a}, \text{ins}(\$, \texttt{b}))$, $\text{rep}(\texttt{a}, \$, \texttt{b})$, and $\text{ins}(\text{del}(\texttt{a}, \$), \texttt{b})$, which can be derived from A as follows.

$$\begin{aligned}
&\text{A} \rightarrow^1_{\mathcal{G}} \text{del}(\texttt{a}, \text{A}) \rightarrow^1_{\mathcal{G}} \text{del}(\texttt{a}, \text{ins}(\text{A}, \texttt{b})) \rightarrow^1_{\mathcal{G}} \text{del}(\texttt{a}, \text{ins}(\$, \texttt{b})),\\
&\text{A} \rightarrow^1_{\mathcal{G}} \text{rep}(\texttt{a}, \text{A}, \texttt{b}) \rightarrow^1_{\mathcal{G}} \text{rep}(\texttt{a}, \$, \texttt{b}), \qquad\qquad\qquad\qquad \text{and}\\
&\text{A} \rightarrow^1_{\mathcal{G}} \text{ins}(\text{A}, \texttt{b}) \rightarrow^1_{\mathcal{G}} \text{ins}(\text{del}(\texttt{a}, \text{A}), \texttt{b}) \rightarrow^1_{\mathcal{G}} \text{ins}(\text{del}(\texttt{a}, \$), \texttt{b})
\end{aligned}$$

The cheapest of these script trees is $\text{rep}(\texttt{a}, \$, \texttt{b})$ with a cost of 1, whereas both other script trees have a cost of 2. Therefore, we obtain $d_{\mathcal{G},\mathcal{F}}(\texttt{a}, \texttt{b}) = 1$. Note that this is equal to the edit distance $d_{\mathcal{S},\mathcal{F}}(\texttt{a}, \texttt{b})$. This is no coincidence. Indeed, we show in Chapter 3 that any edit distance $d_{\mathcal{S},\mathcal{F}}$ is equivalent to the edit distance over its trivial edit tree grammar $d_{\mathcal{G}_{\mathcal{S}},\mathcal{F}}$ if the algebra $\mathcal{F}$ ensures that edit scripts which make detours can never be cheaper than edit scripts which do not. We also show that any such edit distance adheres to metric axioms if the algebra does as well.

Now, let us return to our original motivation for edit tree grammars, namely to incorporate additional constraints. As an example, consider the local alignment distance of T. F. Smith and Waterman (1981), where the suffices of both sequences are considered irrelevant if the edit distance between them exceeds a constant. We can model this behavior by introducing new symbols in our signature called skip^l, $\text{skip}^{l,o}$, skip^r, and $\text{skip}^{r,o}$ as follows.

$$\mathcal{S}_{\text{LOCAL}} := (\{\text{del}, \text{skip}^l, \text{skip}^{l,o}\}, \{\text{rep}\}, \{\text{ins}, \text{skip}^r, \text{skip}^{r,o}\}) \tag{2.17}$$

We extend the edit tree grammar $\mathcal{G}_{\text{ALI}}$ as follows.

$$\begin{aligned}
\mathcal{G}_{\text{LOCAL}} :=& (\{\text{A}, \text{S}\}, \mathcal{S}_{\text{LOCAL}}, \mathcal{R}, \text{A}), \qquad \text{where}\\
\mathcal{R} =& \{\text{A} ::= \text{matchA}|\text{repA}|\text{delA}|\text{insA}|\$\} \cup\\
& \{\text{A} ::= \text{skip}^{l,o}\text{S}|\text{skip}^{r,o}\text{S}\} \cup\\
& \{\text{S} ::= \text{skip}^l\text{S}|\text{skip}^r\text{S}|\$\}
\end{aligned}$$

To ensure a constant cost for ignoring the suffices of both input sequences, the algebra has to assign some constant cost to $\text{skip}^{l,o}$ and $\text{skip}^{r,o}$ and zero costs to skip^l and skip^r.

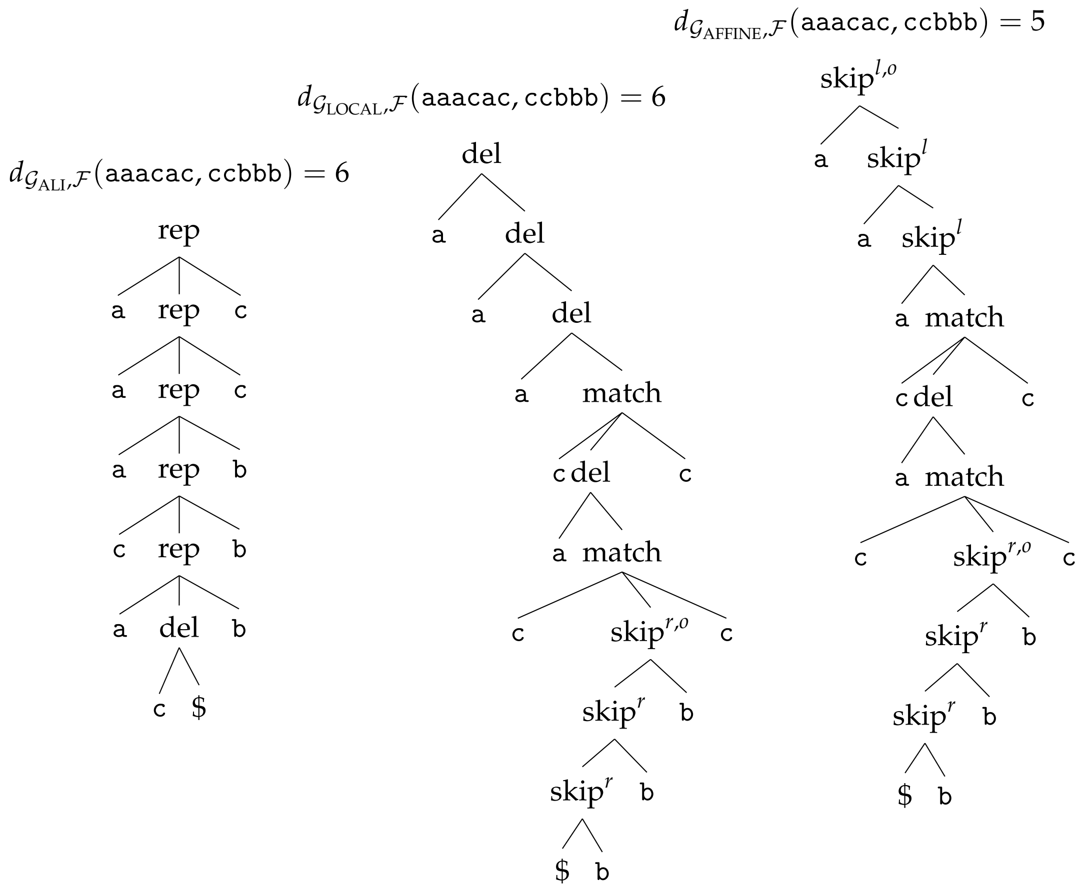


Figure 2.3: The cheapest script trees $\tilde{\delta}$ with yield $\mathcal{Y}(\tilde{\delta}) = (\texttt{aaaacac}, \texttt{ccbbb})$ according to the edit tree grammars $\mathcal{G}_{\text{ALI}}$ (left), $\mathcal{G}_{\text{LOCAL}}$ (middle), and $\mathcal{G}_{\text{AFFINE}}$ (right), respectively. In all cases, we use the algebra $\mathcal{F}_{\text{AFFINE}}$ in Equation 2.19.

If it is possible to use skip^l and skip^r not only in the end but at any point during the edit process, we obtain a scheme, which follows the affine gap cost logic of Gotoh (1982).

$$\begin{aligned} \mathcal{G}_{\text{AFFINE}} :=& (\{A, S\}, \mathcal{S}_{\text{LOCAL}}, \mathcal{R}, A), \qquad \text{where} \\ \mathcal{R} =& \{A ::= \text{match}A|\text{rep}A|\text{del}A|\text{ins}A|\$\} \cup \\ & \{A ::= \text{skip}^{l,o}S|\text{skip}^{r,o}S\} \cup \\ & \{S ::= \text{skip}^l S|\text{skip}^r S|\text{match}A|\text{rep}A|\$\} \end{aligned} \tag{2.18}$$

A comparison of $\mathcal{G}_{\text{ALI}}$, $\mathcal{G}_{\text{LOCAl}}$, and $\mathcal{G}_{\text{AFFINE}}$ is shown in Figure 2.3 for the example sequences $\bar{x} = \texttt{aaacac}$ and $\bar{y} = \texttt{ccbbb}$, using the following algebra $\mathcal{F}_{\text{AFFINE}}$.

$$\begin{aligned} c_{\text{del}}(x) := c^{l,o}_{\text{skip}}(x) := c_{\text{ins}}(x) := c^{r,o}_{\text{skip}}(x) &:= 1 && \forall x \in \mathcal{A} \\ c^l_{\text{skip}}(x) := c^r_{\text{skip}}(x) &= 0.5 && \forall x \in \mathcal{A} \\ c_{\text{rep}}(x, y) := c_{\text{rep}}(y, x) &:= \begin{cases} 1 & \text{if } x \neq y \\ 0 & \text{if } x = y \end{cases} && \forall x, y \in \mathcal{A} \end{aligned} \tag{2.19}$$

This concludes our characterization of sequence edit distance via ADP. In Chapter 3 we build upon this ADP representation and show that the edit distances defined via ADP are indeed pseudo-metrics, and that they are efficiently computable.

Now, that we have covered edit distances over sequences, we can turn towards more complicated data structures, namely trees.

2.3.3 *Tree Edit Distance*

The first edit distance on trees has been suggested by Tai (1979) as a straightforward extension of the standard string edit distance of Levenshtein (1965). In particular, Tai (1979) used the same edit set as the standard string edit distance, namely deletions, insertions, and replacements, and defined the overall edit distance between two trees $\tilde{x}$ and $\tilde{y}$ as the cost of the cheapest edit script $\bar{\delta}$, which transforms $\tilde{x}$ to $\tilde{y}$. To compute the tree edit distance between two trees of size m, Tai (1979) proposed a $\mathcal{O}(m^6)$ dynamic programming algorithm, which was later improved by Zhang and Shasha (1989) to $\mathcal{O}(m^4)$, and by Demaine et al. (2009), Pawlik and Augsten (2011), and Pawlik and Augsten (2016) to $\mathcal{O}(m^3)$, which is provably optimal for this edit set (Demaine et al. 2009).

By constraining the edit set, we can further improve the worst-case bound (Bille 2005). For example, the tree edit distance of Selkow (1977) permits only deletions or insertions of entire subtrees, which reduces the computational complexity to $\mathcal{O}(m^2)$.

In this section, we focus on the classic tree edit distance of Zhang and Shasha (1989) for multiple reasons. First, it provides a proper generalization over the standard string edit distance, and indeed the algorithm of Zhang and Shasha (1989) gracefully degrades to the algorithm of Levenshtein (1965) for the special case of sequential input. Second, it can be seen as the upper limit of structural complexity that can be handled by polynomial-time algorithms, given that both the extensions to graphs, as well as the extension to unordered trees are provably NP-hard (Zhang, Statman, and Shasha 1992; Zeng et al. 2009). Finally, the edits, namely single-node replacements, deletions, and insertions, are simple enough to be intuitive and actionable to humans, e.g. students in intelligent tutoring systems (Rivers and Koedinger 2015; Paaßen, Hammer, et al. 2018). We describe the tree edit distance in detail here because we build upon the notation and concepts introduced in this chapter to learn tree edit distance parameters in Chapter 4.

We first introduce trees and forests as central objects of study for this section, then go on to introduce tree edits and cost functions for such tree edits, and finally define auxiliary concepts on trees, which will enable us to derive the tree edit distance algorithm of Zhang and Shasha (1989).

Definition 2.11 (Tree, Forest, Pre-Order)**.** Let $\mathcal{A}$ be an alphabet. We define a *tree* $\tilde{x}$ over $\mathcal{A}$ recursively as $\tilde{x} = x(\tilde{x}_1, \ldots, \tilde{x}_R)$, where $x \in \mathcal{A}$ and $\tilde{x}_1, \ldots, \tilde{x}_R$ is a (possibly empty) list of trees over $\mathcal{A}$. We denote the set of all trees over $\mathcal{A}$ as $\mathcal{T}(\mathcal{A})$.

We call x the label of $\tilde{x}$, also denoted as $\nu(\tilde{x})$, and we call $\tilde{x}_1, \ldots, \tilde{x}_R$ the children of $\tilde{x}$, also denoted as $\bar{\varrho}(\tilde{x})$. If a tree has no children (i.e. $R = 0$), we call it a *leaf*. In terms of notation, we will generally omit the brackets for leaves, i.e. x is a notational shorthand for $x()$.

We call a list of trees $X = \tilde{x}_1, \ldots, \tilde{x}_R$ from $\mathcal{T}(\mathcal{A})$ a *forest* over $\mathcal{A}$, and we denote the set of all possible forests over $\mathcal{A}$ as $\mathcal{T}(\mathcal{A})^*$. We denote the empty forest as ϵ.

We define the *size* $|X|$ of a forest $X = \tilde{x}_1, \ldots, \tilde{x}_R$ recursively as $|X| = 1 + |\bar{\varrho}(\tilde{x}_1)| + |\tilde{x}_2, \ldots, \tilde{x}_R|$ if $X \neq \epsilon$ and as $|X| = 0$ if $X = \epsilon$.

We define the *pre-order* $\pi(X)$ of a forest $X = \tilde{x}_1, \ldots, \tilde{x}_R$ recursively as the list $\pi(\tilde{x}) := \tilde{x}_1, \pi(\bar{\varrho}(\tilde{x}_1)), \pi(\tilde{x}_2, \ldots, \tilde{x}_R)$ if $X \neq \epsilon$ and as $\pi(X) = \epsilon$ if $X = \epsilon$. We denote the ith tree in the pre-order of X as $\tilde{x}^i$, and the label $\nu(\tilde{x}^i)$ of $\tilde{x}^i$ as x_i.

Regarding this definition, note that any tree is a special case of a forest, that the children of a tree also form a forest, and that any single element from $\mathcal{A}$ is a trivial case of a tree.

As an example, consider the alphabet $\mathcal{A} = \{\mathtt{a}, \mathtt{b}\}$. Some trees over $\mathcal{A}$ are $\mathtt{a}$, $\mathtt{b}$, $\mathtt{a}(\mathtt{a})$, $\mathtt{a}(\mathtt{b})$, $\mathtt{b}(\mathtt{a}, \mathtt{b})$, and $\mathtt{a}(\mathtt{b}(\mathtt{a}, \mathtt{b}), \mathtt{b})$. An example forest over this alphabet is $(\mathtt{a}, \mathtt{b}, \mathtt{b}(\mathtt{a}, \mathtt{b}))$. Now, consider the example tree $\tilde{x} = \mathtt{a}(\mathtt{b}(\mathtt{c}, \mathtt{d}), \mathtt{e})$ The label of $\tilde{x}$ is $\nu(\tilde{x}) = \mathtt{a}$, and the children are $\bar{\varrho}(\tilde{x}) = \mathtt{b}(\mathtt{c}, \mathtt{d})$ and $\mathtt{e}$. The size of $\tilde{x}$ is

$$\begin{aligned}
|\tilde{x}| &= 1 + |\mathtt{b}(\mathtt{c}, \mathtt{d}), \mathtt{e}| + |\epsilon| \\
&= 1 + (1 + |\mathtt{c}, \mathtt{d}| + |\mathtt{e}|) + 0 \\
&= 2 + (1 + |\epsilon| + |\mathtt{d}|) + (1 + |\epsilon| + |\epsilon|) \\
&= 3 + (1 + |\epsilon| + |\epsilon|) + 1 \\
&= 3 + 1 + 1 = 5.
\end{aligned}$$

Intuitively, the pre-order of $\tilde{x}$ is the list of all subtrees of $\tilde{x}$ according to depth-first search. Strictly using the definition, we obtain:

$$\begin{aligned}
\pi(\tilde{x}) &= \tilde{x}, \pi(\mathtt{b}(\mathtt{c}, \mathtt{d}), \mathtt{e}), \pi(\epsilon) \\
&= \tilde{x}, \mathtt{b}(\mathtt{c}, \mathtt{d}), \pi(\mathtt{c}, \mathtt{d}), \pi(\mathtt{e}), \pi(\epsilon) \\
&= \tilde{x}, \mathtt{b}(\mathtt{c}, \mathtt{d}), \mathtt{c}, \pi(\epsilon), \pi(\mathtt{d}), \mathtt{e}, \pi(\epsilon), \pi(\epsilon), \\
&= \tilde{x}, \mathtt{b}(\mathtt{c}, \mathtt{d}), \mathtt{c}, \mathtt{d}, \pi(\epsilon), \pi(\epsilon), \mathtt{e}, \\
&= \tilde{x}, \mathtt{b}(\mathtt{c}, \mathtt{d}), \mathtt{c}, \mathtt{d}, \mathtt{e}.
\end{aligned}$$

Next, we define how to manipulate trees via tree edits. Note that tree edits are analogous to sequence edits in Definition 2.5.

Definition 2.12 (Tree Edits)**.** We define a *tree edit* δ as a function $\delta : \mathcal{T}(\mathcal{A})^* \to \mathcal{T}(\mathcal{A})^*$. We call a set Δ of tree edits an *edit set*. We define an *edit script* over Δ as a sequence over Δ. We denote the set of all possible edit scripts over edit set as Δ^*. We define the application $\bar{\delta}(X)$ of an edit script $\bar{\delta} = \delta_1 \ldots \delta_T$ to a forest X as the function composition $\delta_T \circ \ldots \delta_1(X)$, where $\delta \circ \delta'(X) := \delta(\delta'(X))$. If $\bar{\delta} = \epsilon$, we define $\bar{\delta}(X) = X$.

Further, we define three special tree edits, which will become important for the tree edit distance. In particular, we define a *deletion* as the following function del.

$$\begin{aligned}
\mathrm{del}(\epsilon) &:= \epsilon \\
\mathrm{del}(\tilde{x}_1, \ldots, \tilde{x}_R) &:= \bar{\varrho}(\tilde{x}_1), \tilde{x}_2, \ldots, \tilde{x}_R
\end{aligned}$$

We define a *replacement* with node $y \in \mathcal{A}$ as the following function rep_y.

$$\begin{aligned}
\mathrm{rep}_y(\epsilon) &:= \epsilon \\
\mathrm{rep}_y(\tilde{x}_1, \ldots, \tilde{x}_R) &:= y(\bar{\varrho}(\tilde{x}_1)), \tilde{x}_2, \ldots, \tilde{x}_R
\end{aligned}$$

And we define an *insertion* of node $y \in \mathcal{A}$ as parent of the trees l to $r-1$ as the following function $\text{ins}_{y,l,r}$.

$$\text{ins}_{y,l,r}(\tilde{x}_1, \ldots, \tilde{x}_R) := \begin{cases} \tilde{x}_1, \ldots, \tilde{x}_R & \text{if } r > R+1, l > r, \text{ or } l < 1 \\ \tilde{x}_1, \ldots, \tilde{x}_{l-1}, y, \tilde{x}_l, \ldots, \tilde{x}_R & \text{if } 1 \leq l = r \leq R+1 \\ \tilde{x}_1, \ldots, \tilde{x}_{l-1}, y(\tilde{x}_l, \ldots, \tilde{x}_{r-1}), \tilde{x}_r, \ldots, \tilde{x}_R & \text{if } 1 \leq l < r \leq R+1 \end{cases}$$

We further define variants of these tree edits to be applied to any specific location in the input forest. In particular, let δ be either a deletion or replacement. Then, we define a deletion/replacement of the ith node as the following function δ_i.

$$\delta_i(\epsilon) := \epsilon$$

$$\delta_i(\tilde{x}_1, \ldots, \tilde{x}_R) := \begin{cases} \tilde{x}_1, \ldots, \tilde{x}_R & \text{if } i < 1 \\ \delta(\tilde{x}_1, \ldots, \tilde{x}_R) & \text{if } i = 1 \\ \nu(\tilde{x}_1)\big(\delta_{i-1}(\bar{\varrho}(\tilde{x}_1))\big), \tilde{x}_2, \ldots, \tilde{x}_R & \text{if } 1 < i \leq |\tilde{x}_1| \\ \tilde{x}_1, \delta_{i-|\tilde{x}_1|}(\tilde{x}_2, \ldots, \tilde{x}_R) & \text{if } i > |\tilde{x}_1| \end{cases}$$

Now, consider the insertion $\text{ins}_{y,l,r}$. We define an insertion at the ith node as the following function $\text{ins}_{i,y,l,r}$.

$$\text{ins}_{i,y,l,r}(\tilde{x}_1, \ldots, \tilde{x}_R) := \begin{cases} \tilde{x}_1, \ldots, \tilde{x}_R & \text{if } i < 0 \\ \text{ins}_{y,l,r}(\tilde{x}_1, \ldots, \tilde{x}_R) & \text{if } i = 0 \\ \nu(\tilde{x}_1)\big(\text{ins}_{i-1,y,l,r}(\bar{\varrho}(\tilde{x}_1))\big), \tilde{x}_2, \ldots, \tilde{x}_R & \text{if } 1 \leq i \leq |\tilde{x}_1| \\ \tilde{x}_1, \text{ins}_{i-|\tilde{x}_1|,y,l,r}(\tilde{x}_2, \ldots, \tilde{x}_R) & \text{if } i > |\tilde{x}_1| \end{cases}$$

We define the tree edit distance edit set $\Delta_{\mathcal{A}}$ for the alphabet $\mathcal{A}$ as the following set: $\Delta_{\mathcal{A}} := \{\text{del}_i | i \in \mathbb{N}\} \cup \{\text{rep}_{i,y} | i \in \mathbb{N}, y \in \mathcal{A}\} \cup \{\text{ins}_{i,y,l,r} | i \in \mathbb{N}_0, l, r \in \mathbb{N}, y \in \mathcal{A}\}$.

As an example, consider the tree $\tilde{x} = \texttt{a}(\texttt{b}(\texttt{c},\texttt{d}),\texttt{e})$ from Figure 2.4 (left). By means of the edit script $\bar{\delta} = \text{rep}_{1,\texttt{f}}\text{del}_2\text{del}_2\text{rep}_{2,\texttt{g}}\text{del}_3$, we can transform $\tilde{x}$ successively into the trees $\text{rep}_{1,\texttt{f}}(\tilde{x}) = \texttt{f}(\texttt{b}(\texttt{c},\texttt{d}),\texttt{e})$, $\text{rep}_{1,\texttt{f}}\text{del}_2(\tilde{x}) = \texttt{f}(\texttt{c},\texttt{d},\texttt{e})$, $\text{rep}_{1,\texttt{f}}\text{del}_2\text{del}_2(\tilde{x}) = \texttt{f}(\texttt{d},\texttt{e})$, $\text{rep}_{1,\texttt{f}}\text{del}_2\text{del}_2\text{rep}_{2,\texttt{g}}(\tilde{x}) = \texttt{f}(\texttt{g},\texttt{e})$, and finally $\text{rep}_{1,\texttt{f}}\text{del}_2\text{del}_2\text{rep}_{2,\texttt{g}}\text{del}_3(\tilde{x}) = \texttt{f}(\texttt{g})$ (see Figure 2.4). Conversely, we can also edit in the inverse direction (see Figure 2.4, bottom).

Finally, we define cost functions for tree edits and the tree edit distance.

Definition 2.13 (Cost Function, Tree Edit Distance)**.** Let $\mathcal{A}$ be an alphabet with $- \notin \mathcal{A}$. We call $-$ the *gap symbol*. We define a *cost function* over $\mathcal{A}$ as a function $c : (\mathcal{A} \cup \{-\}) \times (\mathcal{A} \cup \{-\}) \to \mathbb{R}$.

We define the cost of applying deletion del_i to some input forest X as 0 if $\text{del}_i(X) = X$ and as $c(\text{del}_i, X) := c(x_i, -)$ otherwise.

We define the cost of applying replacement $\text{rep}_{i,y}$ to some input forest X as 0 if $\text{rep}_{i,y}(X) = X$, and as $c(\text{rep}_{i,y}, X) := c(x_i, y)$ otherwise.

We define the cost of applying insertion $\text{ins}_{i,y,l,r}$ to some input forest X as 0 if $\text{ins}_{i,y,l,r}(X) = X$, and as $c(-, y)$ otherwise.

We define the cost of applying an edit script $\bar{\delta} = \delta_1 \ldots \delta_T \in \Delta_{\mathcal{A}}^*$ to some input forest X recursively as $c(\epsilon, X) = 0$ and $c(\delta_1 \ldots \delta_T, X) := c(\delta_1, X) + c(\delta_2 \ldots \delta_T, \delta_1(X))$.

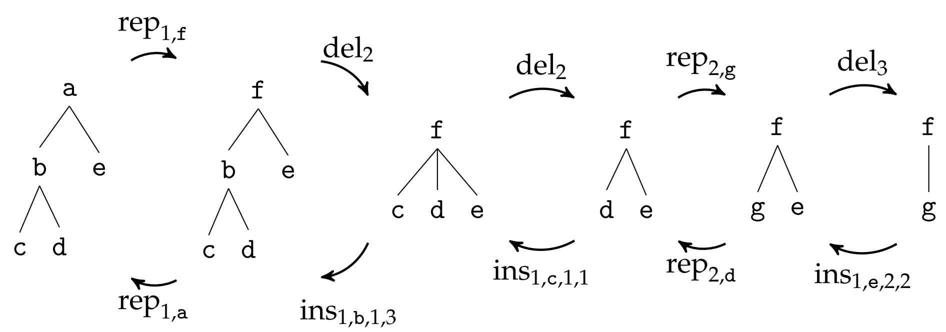


Figure 2.4: An illustration of an edit script transforming the tree $\tilde{x}$ on the left to the tree $\tilde{y}$ on the right, and an edit script transforming the tree $\tilde{y}$ on the right to the tree $\tilde{x}$ on the left. The intermediate trees resulting from the application of single tree edits are shown in the middle.

Finally, we define the *tree edit distance* according to c as the following function.

$$\begin{aligned} d_c :& \mathcal{T}(\mathcal{A}) \times \mathcal{T}(\mathcal{A}) \to \mathbb{R} \\ d_c(\tilde{x}, \tilde{y}) :=& \min_{\bar{\delta} \in \Delta^*_{\mathcal{A}}} \{c(\bar{\delta}, \tilde{x}) | \bar{\delta}(\tilde{x}) = \tilde{y}\} \end{aligned} \tag{2.20}$$

The cost function is our central interface for metric learning in Chapter 4. Indeed, we phrase edit distance learning for trees as learning the parameters of a cost function.

Consider again the example in Figure 2.4, displaying an edit script that transforms the tree $\tilde{x} = \mathtt{a}(\mathtt{b}(\mathtt{c}, \mathtt{d}), \mathtt{e})$ into the tree $\tilde{y} = \mathtt{f}(\mathtt{g})$. If we define $c(x, y) = 1$ if $x \neq y$ and as 0 otherwise, the cost of this edit script would be 5. Because there is no cheaper edit script, this is equivalent to the tree edit distance between $\tilde{x}$ and $\tilde{y}$. Note that the edit script at the bottom of Figure 2.4 also has a cost of 5 according to c, and is also the cheapest edit script transforming $\tilde{y}$ to $\tilde{x}$. Indeed, Zhang and Shasha (1989) have already remarked that a metric cost function c implies a metric tree edit distance d_c. While they did not provide a proof for this claim, the proof is fairly simple and can be found in Appendix A.4.

Theorem 2.4. *Let $\mathcal{A}$ be an alphabet with $- \notin \mathcal{A}$ and let c be a cost function over $\mathcal{A}$. Then it holds: For any trees $\tilde{x}, \tilde{y} \in \mathcal{T}(\mathcal{A})$, there exists at least one edit script $\bar{\delta} \in \Delta_{\mathcal{A}}$ such that $\bar{\delta}(\tilde{x}) = \tilde{y}$.*

Further it holds: If c is a (pseudo-)metric over $\mathcal{A} \cup \{-\}$, then d_c is a (pseudo-)metric over $\mathcal{T}(\mathcal{A})$. More specifically, the following claims hold if c is non-negative (i.e. $\forall x, y \in \mathcal{A} \cup \{-\} : c(x, y) \geq 0$).

Non-Negativity: $\forall \tilde{x}, \tilde{y} \in \mathcal{T}(\mathcal{A}) : d_c(\tilde{x}, \tilde{y}) \geq 0$.

Self-Equality: $\forall x \in \mathcal{A} \cup \{-\} : c(x, x) = 0$ *implies* $\forall \tilde{x} \in \mathcal{T}(\mathcal{A}) : d_c(\tilde{x}, \tilde{x}) = 0$.

Discernibility: $\forall x, y \in \mathcal{A} \cup \{-\} : x \neq y \Rightarrow c(x, y) > 0$ *implies* $\forall \tilde{x}, \tilde{y} \in \mathcal{T}(\mathcal{A}) : \tilde{x} \neq \tilde{y} \Rightarrow d_c(\tilde{x}, \tilde{y}) > 0$.

Symmetry: $\forall x, y \in \mathcal{A} \cup \{-\} : c(x, y) = c(y, x)$ *implies* $\forall \tilde{x}, \tilde{y} \in \mathcal{T}(\mathcal{A}) : d_c(\tilde{x}, \tilde{y}) = d_c(\tilde{y}, \tilde{x})$.

Triangular Inequality: $\forall \tilde{x}, \tilde{y}, \tilde{z} \in \mathcal{T}(\mathcal{A}) : d_c(\tilde{x}, \tilde{y}) + d_c(\tilde{y}, \tilde{z}) \geq d_c(\tilde{x}, \tilde{z})$

Proof. Refer to Appendix A.4. □

Note that this result implies that any tree edit distance d_c for a non-negative, symmetric, and self-equal cost function c is pseudo-Euclidean.

Unfortunately, our framework up to this point is not yet sufficient to derive an efficient algorithm to compute the tree edit distance. In particular, the search space of possible edit scripts which transform a tree $\tilde{x}$ into another tree $\tilde{y}$ is infinite, making search infeasible. As for the sequence edit distance, we can drastically reduce the search space by only considering edit scripts, which avoid detours in the sense that, once we have changed a node in a tree, we do not change it again. To avoid such detours, we introduce an alternative representation, namely tree mappings. Tree mappings also form a backbone of our tree edit distance learning algorithm in Chapter 4.

Definition 2.14 (Parents, Ancestors, Tree Mappings)**.** Let $\mathcal{A}$ be an alphabet and let $\tilde{x}$ be a tree over $\mathcal{A}$. Further, let $i \in \{1, \ldots, |\tilde{x}|\}$, let $\tilde{x}^i = x_i(\tilde{x}^i_1, \ldots, \tilde{x}^i_{R_i})$, let $r \in \{1, \ldots, R_i\}$, and let $i_r := i + 1 + \sum_{l=1}^{r-1} |\tilde{x}^i_l|$. Then, we define the *parent index* $\mathrm{par}_{\tilde{x}}(i_r)$ of i_r in $\tilde{x}$ as i, that is, $\mathrm{par}_{\tilde{x}}(i_r) := i$. Further, we define $\mathrm{par}_{\tilde{x}}(1) = 0$.

For any $i \in \{2, \ldots, |\tilde{x}|\}$ we define the *ancestors* $\mathrm{anc}_{\tilde{x}}(i)$ of i in $\tilde{x}$ recursively as $\mathrm{anc}_{\tilde{x}}(i) := \{\mathrm{par}_{\tilde{x}}(i)\} \cup \mathrm{anc}_{\tilde{x}}(\mathrm{par}_{\tilde{x}}(i))$, with $\mathrm{anc}_{\tilde{x}}(1) := \emptyset$.

Let $\tilde{y}$ be another tree over $\mathcal{A}$. Then, we define a *tree mapping* M between $\tilde{x}$ and $\tilde{y}$ as a subset $M \subseteq \{1, \ldots, |\tilde{x}|\} \times \{1, \ldots, |\tilde{y}|\}$ such that the following conditions hold for all entries $(i, j), (i', j') \in M$.

$$i \geq i' \iff j \geq j' \qquad \text{(pre-order preservation)} \tag{2.21}$$

$$i \in \mathrm{anc}_{\tilde{x}}(i') \iff j \in \mathrm{anc}_{\tilde{y}}(j') \qquad \text{(ancestral preservation)} \tag{2.22}$$

We define the *left-complement* of M as $I(M, \tilde{x}, \tilde{y}) := \{i \in \{1, \ldots, |\tilde{x}|\} | \nexists j \in \{1, \ldots, |\tilde{y}|\} : (i, j) \in M\}$ and we define the *right-complement* of M as $J(M, \tilde{x}, \tilde{y}) := \{j \in \{1, \ldots, |\tilde{y}|\} | \nexists i \in \{1, \ldots, |\tilde{x}|\} : (i, j) \in M\}$.

Now, let c be a cost function over $\mathcal{A}$. Then, we define the *cost* of M according to c as follows.

$$c(M, \tilde{x}, \tilde{y}) := \sum_{(i,j) \in M} c(x_i, y_j) + \sum_{i \in I(M, \tilde{x}, \tilde{y})} c(x_i, -) + \sum_{j \in J(M, \tilde{x}, \tilde{y})} c(-, y_j) \tag{2.23}$$

And we define the tree mapping edit distance between $\tilde{x}$ and $\tilde{y}$ according to c as:

$$D_c(\tilde{x}, \tilde{y}) := \min_{M \subset \{1, \ldots, |\tilde{x}|\} \times \{1, \ldots, |\tilde{y}|\}} \{c(M, \tilde{x}, \tilde{y}) | M \text{ is a tree mapping between } \tilde{x} \text{ and } \tilde{y}\} \tag{2.24}$$

Finally, we define a tree mapping M between $\tilde{x}$ and $\tilde{y}$ as *cooptimal* if it holds: $c(M, \tilde{x}, \tilde{y}) = D_c(\tilde{x}, \tilde{y})$.

Consider the example trees $\tilde{x} = \mathtt{a}(\mathtt{b}(\mathtt{c}, \mathtt{d}), \mathtt{e})$ and $\tilde{y} = \mathtt{f}(\mathtt{g})$ from Figure 2.4. An example tree mapping between $\tilde{x}$ and $\tilde{y}$ is $M = \{(1, 1), (4, 2)\}$ (see Figure 2.5, top left). By contrast, $\{(1, 1), (1, 2)\}$ would not be a valid mapping because $1 \geq 1$ but $1 \ngeq 2$, $\{(1, 1), (2, 1)\}$ would not be a valid mapping because $1 \ngeq 2$ but $1 \geq 1$, $\{(1, 2), (2, 1)\}$ would not be a valid mapping because $1 \leq 2$ but $2 \nleq 1$, and $\{(3, 1), (5, 2)\}$ would not be a valid mapping because $3 \notin \mathrm{anc}_{\tilde{x}}(5)$ but $1 \in \mathrm{anc}_{\tilde{y}}(2)$ (refer to Figure 2.5).

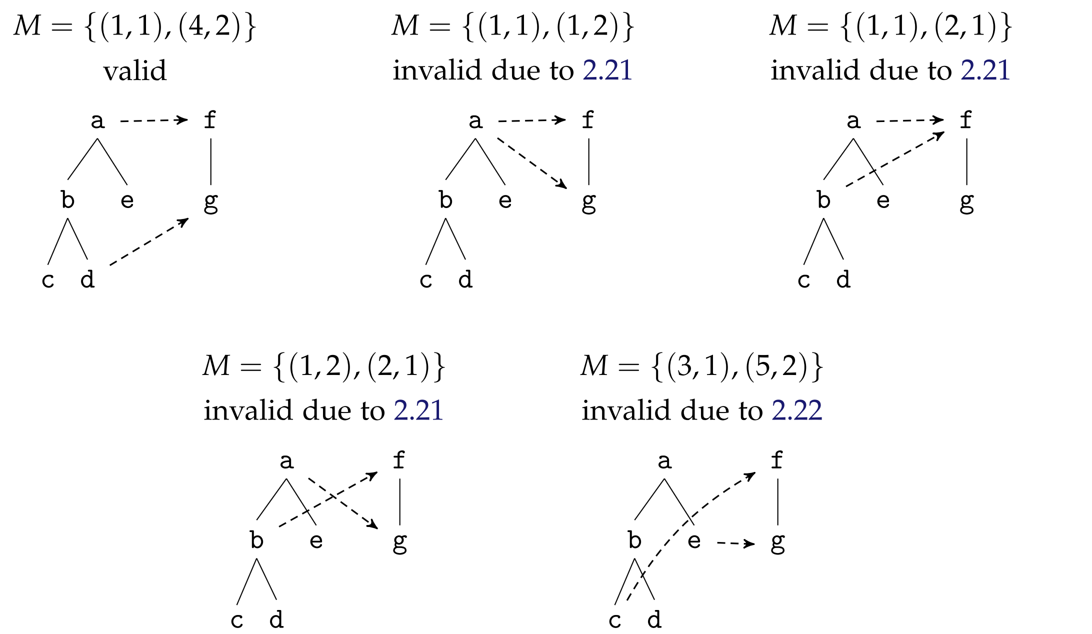


Figure 2.5: One example tree mapping (top left) between the trees $\tilde{x} = \mathtt{a}(\mathtt{b}(\mathtt{c},\mathtt{d}),\mathtt{e})$ and $\tilde{y} = \mathtt{f}(\mathtt{g})$ from Figure 2.4 and four sets, which are not valid tree mappings due to violations of one of the tree mapping constraints.

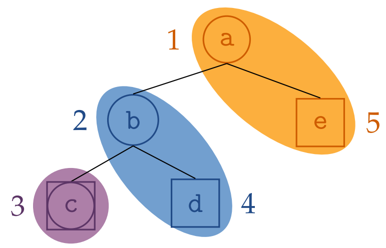
The left-complement of the tree mapping $M = \{(1,1),(4,2)\}$ would be $I(M, \tilde{x}, \tilde{y}) = \{2,3,5\}$ and the right-complement would be $J(M, \tilde{x}, \tilde{y}) = \emptyset$. Accordingly, the cost of M would be $c(M, \tilde{x}, \tilde{y}) = c(\mathtt{a}, \mathtt{f}) + c(\mathtt{d}, \mathtt{g}) + c(\mathtt{b}, -) + c(\mathtt{c}, -) + c(\mathtt{e}, -)$.

Because tree mappings are defined as subsets of $\{1, \ldots, |\tilde{x}|\} \times \{1, \ldots, |\tilde{y}|\}$, the search space becomes finite. However, the number of all such tree mappings between $\tilde{x}$ and $\tilde{y}$ is still exponential in $|\tilde{x}| \cdot |\tilde{y}|$ and thus infeasible to enumerate. As an additional trick, Zhang and Shasha (1989) devised an efficient dynamic programming scheme to compute the tree mapping edit distance between $\tilde{x}$ and $\tilde{y}$ from the tree mapping edit distance between subtrees and -forests of $\tilde{x}$ and $\tilde{y}$, yielding an $\mathcal{O}(|\tilde{x}|^2 \cdot |\tilde{y}|^2)$ runtime algorithm. Note that this is not optimal in the worst case, and more worst-case efficient algorithms have since been introduced by Demaine et al. (2009), Pawlik and Augsten (2011), and Pawlik and Augsten (2016). Still, the algorithm of Zhang and Shasha (1989) is considerably simpler, is still optimal in terms of space complexity, and performs still well in realistic comparisons (Pawlik and Augsten 2016) such that we focus on this algorithm during the course of this thesis.

We require only one additional concept for the algorithm of Zhang and Shasha (1989), namely the notion of keyroots, which we define as follows.

Definition 2.15 (Outermost right leaves, Keyroots)**.** Let $\mathcal{A}$ be an alphabet and let $\tilde{x}$ be a tree over $\mathcal{A}$. Then, for any $i \in \{1, \ldots, |\tilde{x}|\}$ we define the *outermost right leaf* $rl_{\tilde{x}}(i)$ of i in $\tilde{x}$ as $rl_{\tilde{x}}(i) := i + |\tilde{x}^i| - 1$; we define the *keyroot* $\mathrm{k}_{\tilde{x}}(i)$ of i in $\tilde{x}$ as $\mathrm{k}_{\tilde{x}}(i) := \min\{j | rl_{\tilde{x}}(i) = rl_{\tilde{x}}(j)\}$; and we define the *keyroots* $\mathcal{K}(\tilde{x})$ of $\tilde{x}$ as the set $\mathcal{K}(\tilde{x}) := \{j | \exists i \in \{1, \ldots, |\tilde{x}|\} : j = \mathrm{k}_{\tilde{x}}(i)\}$.

For example, consider the tree $\tilde{x} = \mathtt{a}(\mathtt{b}(\mathtt{c},\mathtt{d}),\mathtt{e})$ from Figure 2.4 (left). The subtrees, labels, parent indices, outermost right leaves and keyroots for this tree are illustrated in Figure 2.6. The set of keyroots is $\mathcal{K}(\tilde{x}) = \{1,2,3\}$.



i	$\tilde{x}^i$	x_i	$\text{par}_{\tilde{x}}(i)$	$rl_{\tilde{x}}(i)$	$\text{k}_{\tilde{x}}(i)$
1	a(b(c,d),e)	a	0	5	1
2	b(c,d)	b	1	4	2
3	c	c	2	3	3
4	d	d	2	4	2
5	e	e	1	5	1

Figure 2.6: Left: The tree $\tilde{x} = \texttt{a}(\texttt{b}(\texttt{c},\texttt{d}),\texttt{e})$ with pre-order indices drawn next to each node. Nodes with the same outermost right leaf and keyroot are encircled with a colored region. The outermost right leaf for that region is highlighted by a rectangle, the keyroot with a circle. Right: A table listing the subtree, label, parent index, outermost right leaf, and keyroot for all indices i for the tree $\tilde{x} = \texttt{a}(\texttt{b}(\texttt{c},\texttt{d}),\texttt{e})$.

Note that for trees that are sequences, such as $\tilde{x} = \texttt{a}(\texttt{b}(\texttt{c}))$, there exists only a single keyroot, namely the root of the tree.

These auxiliary concepts are sufficient to specify the tree edit distance algorithm of Zhang and Shasha (1989), which is provably correct for metric cost functions.

Theorem 2.5. *Let $\mathcal{A}$ be an alphabet, let c be a cost function over $\mathcal{A}$, and let $\tilde{x}$ and $\tilde{y}$ be trees over $\mathcal{A}$. Then, Algorithm 2.1 computes the tree mapping edit distance $D_c(\tilde{x}, \tilde{y})$ between $\tilde{x}$ and $\tilde{y}$. Further, Algorithm 2.1 runs in $\mathcal{O}(|\tilde{x}| \cdot |\tilde{y}|)$ space complexity and $\mathcal{O}(|\tilde{x}|^2 \cdot |\tilde{y}|^2)$ time complexity.*

Finally, it holds: If c is self-equal, non-negative, and fulfills the triangular inequality, then $D_c(\tilde{x}, \tilde{y}) = d_c(\tilde{x}, \tilde{y})$.

Proof. The original proof is due to Zhang and Shasha (1989). For a version adapted to our notation here, refer to Appendix A.5. □

This concludes our introduction of the tree edit distance. Next, we turn to even more complicated data structures, namely graphs.

2.3.4 *Graph Edit Distance*

Following the prior work of Levenshtein (1965) and Tai (1979), it appears straightforward to extend the concept of an edit distance to general graphs. In particular, we can define the graph edit distance between $\mathcal{G}$ and $\mathcal{G}'$ as the cheapest edit script, which transforms $\mathcal{G}$ into $\mathcal{G}'$, where the edit set is given as the set of all possible node deletions, node insertions, node replacements, edge deletions, edge insertions, and edge replacements (Sanfeliu and Fu 1983). Unfortunately, computing the graph edit distance is provably NP-hard (Zeng et al. 2009). Still, many approximation schemes exist, e.g. relying on self-organizing maps, Gaussian mixture models, graph kernels, or binary linear programming (Gao et al. 2010). A particularly simple approximation scheme is to order the nodes of a graph in a sequence and then apply a standard string edit distance measure to these sequences (Robles-Kelly and Hancock 2003; Robles-Kelly and Hancock 2005). We utilize this method in Chapter 5 to obtain an approximate graph edit distance on enriched syntax trees. Other than that, we mostly focus on sequence and tree edit distances in this work. Next, we turn towards the topic of *learning* such edit distances.

Algorithm 2.1 The dynamic programming algorithm of Zhang and Shasha (1989) for computing the tree edit distance between two input trees $\tilde{x}$ and $\tilde{y}$ according to the cost function c. The algorithm iterates over all subtrees of $\tilde{x}$ and $\tilde{y}$ rooted at keyroots and computes the tree edit distance between them based on the forest edit distances between all subforests of the respective subtrees.

function TREE-EDIT-DISTANCE(trees $\tilde{x}$ and $\tilde{y}$, a cost function c.)
    $\boldsymbol{d} \leftarrow |\tilde{x}| \times |\tilde{y}|$ matrix of zeros.
    $\boldsymbol{D} \leftarrow (|\tilde{x}|+1) \times (|\tilde{y}|+1)$ matrix of zeros.
    for $k \in \mathcal{K}(\tilde{x})$ in descending order **do**
        for $l \in \mathcal{K}(\tilde{y})$ in descending order **do**
            $\boldsymbol{D}_{rl_{\tilde{x}}(k)+1, rl_{\tilde{y}}(l)+1} \leftarrow 0.$
            for $i \leftarrow rl_{\tilde{x}}(k), \ldots, k$ **do**
                $\boldsymbol{D}_{i, rl_{\tilde{y}}(l)+1} \leftarrow \boldsymbol{D}_{i+1, rl_{\tilde{y}}(l)+1} + c(x_i, -).$
            end for
            for $j \leftarrow rl_{\tilde{y}}(l), \ldots, l$ **do**
                $\boldsymbol{D}_{rl_{\tilde{x}}(k)+1, j} \leftarrow \boldsymbol{D}_{rl_{\tilde{x}}(k)+1, j+1} + c(-, y_j).$
            end for
            for $i \leftarrow rl_{\tilde{x}}(k), \ldots, k$ **do**
                for $j \leftarrow rl_{\tilde{y}}(l), \ldots, l$ **do**
                    if $rl_{\tilde{x}}(i) = rl_{\tilde{x}}(k) \wedge rl_{\tilde{y}}(j) = rl_{\tilde{y}}(l)$ **then**
                        $\boldsymbol{D}_{i,j} \leftarrow \min\{\boldsymbol{D}_{i+1,j} + c(x_i, -),$
                            $\boldsymbol{D}_{i,j+1} + c(-, y_j),$
                            $\boldsymbol{D}_{i+1,j+1} + c(x_i, y_j)\}.$
                        $\boldsymbol{d}_{i,j} \leftarrow \boldsymbol{D}_{i,j}.$
                    else
                        $\boldsymbol{D}_{i,j} \leftarrow \min\{\boldsymbol{D}_{i+1,j} + c(x_i, -),$
                            $\boldsymbol{D}_{i,j+1} + c(-, y_j),$
                            $\boldsymbol{D}_{rl_{\tilde{x}}(i)+1, rl_{\tilde{y}}(j)+1} + \boldsymbol{d}_{i,j}\}.$
                    end if
                end for
            end for
        end for
    end for
    return $\boldsymbol{d}_{1,1}$.
end function

2.4 METRIC LEARNING FOR EDIT DISTANCES

Even if an efficient edit distance for our data at hand does exist, default cost functions may not be optimal for our task at hand. For example, when analyzing protein sequences, the default cost function of the string edit distance assumes that all amino acids have the same pairwise distance, which does not correspond to biological reality (S. Henikoff and J. G. Henikoff 1992; Saigo, Vert, and Akutsu 2006; Kann, Qian, and Goldstein 2000; Hourai, Akutsu, and Akiyama 2004). As such, we would like to learn a cost function such that the resulting edit distance is better suited for the task at hand. In other words, we would like to obtain a cost function which pulls semantically close data (positive pairs) closer together and pushes semantically distant data (negative pairs) further apart (Bellet, Habrard, and Sebban 2014).

Past literature has almost exclusively focused on positive pairs, i.e. pulling semantically close data closer together. In that setting, we can re-phrase the edit distance as a negative log-likelihood of one structured datum being generated from another via edits, and the cost function as the atomic joint probability or conditional probability of single edits. Accordingly, metric learning means maximizing the joint or conditional probability of positive pairs by adjusting the atomic edit probabilities (Boyer, Habrard, and Sebban 2007; Emms 2012; Oncina and Sebban 2006; Ristad and Yianilos 1998). Unfortunately, such a setup can not prevent that negative pairs get close as well. In the extreme case, the entire metric can degenerate such that all data ends up in a single point (Bellet, Habrard, and Sebban 2012).

According to Bellet, Habrard, and Sebban (2014), the only edit distance learning scheme that also considers negative pairs is good edit similarity learning (GESL, Bellet, Habrard, and Sebban 2012). The authors have experimentally shown that this approach outperforms existing edit distance learning approaches and can thus be regarded as the state of the art. Therefore, we focus in our comparisons on good edit similarity learning (GESL) and introduce this method in more detail.

2.4.1 *Good Edit Similarity Learning*

GESL is intended to maximize the "goodness" of the similarity measure $s(x, y) = 2 \cdot \exp(-d_c(x, y)) - 1$, where $d_c(x, y)$ is an edit distance. Goodness, as suggested by Balcan, Blum, and Srebro (2008), quantifies how well a given similarity measure s lends itself for binary classification. In particular, assume data $x_1, \ldots, x_M$ with labels $y_1, \ldots, y_M \in \{-1, 1\}$. Then, Balcan, Blum, and Srebro (2008) suggest a binary classifier with the predictive function

$$f(x) = \text{sign}\Big(\sum_{i=1}^{M} \alpha_i \cdot s(x, x_i)\Big)$$

where α_i are real-valued coefficients that constitute the parameters of the classifier. One can learn the parameters $\vec{\alpha} \in \mathbb{R}^M$ by solving the following linear problem.

$$\min_{\vec{\alpha}} \sum_{i=1}^{M} \Big[1 - y_i \cdot \sum_{j=1}^{M} \alpha_j \cdot s(x_i, x_j)\Big]_+ + \nu \cdot \|\vec{\alpha}\|_1$$

where $[\cdot]_+ = \max\{\cdot, 0\}$ denotes the hinge loss, and where ν is a hyper-parameter for the L1 regularization and hence the sparsity of $\vec{\alpha}$.

To maximize the goodness of the similarity measure s, Bellet, Habrard, and Sebban (2012) propose that each data point x_i should pull its K closest neighbors from the same class N_i^+ closer and push the K furthest neighbors from a different class N_i^- away. In particular, Bellet, Habrard, and Sebban (2012) suggest to solve the following minimization problem[2].

$$\begin{aligned} \min_{c} \quad & \lambda \cdot \|c\|^2 + \sum_{i=1}^{M} \sum_{j \in N_i^+} [d_c(x_i, x_j) - \eta]_+ \sum_{j \in N_i^-} [\log(2) + \eta - d_c(x_i, x_j)]_+ \\ \text{s.t.} \quad & c(x, y) \geq 0 \qquad \forall x, y \in \mathcal{A}, \qquad 0 \leq \eta \leq \log(2) \end{aligned} \tag{2.25}$$

where $\eta \in [0, \log(2)]$ is a slack variable permitting higher distances between positive pairs if negative pairs are further apart, λ is a scalar regularization constant, and $\|c\|^2$ denotes $\sum_{x \in \mathcal{A} \cup \{-\}} \sum_{y \in \mathcal{A} \cup \{-\}} c(x, y)^2$.

Because the edit distance involves discrete minimum operations, which are discontinuous and non-differentiable, the minimization problem 2.25 is not immediately feasible. In order to make optimization tractable, Bellet, Habrard, and Sebban (2012) observe that most edit distances can be regarded as equivalent to a tree mapping edit distance (also refer to Definition 2.14). If that is the case, we can decompose the edit distance into a matrix expressing the tree mapping itself, and the cost function. In particular, we define tree mapping matrix for the tree edit distance as follows.

Definition 2.16 (Tree mapping matrix)**.** Let $\tilde{x}$ and $\tilde{y}$ be trees over some alphabet $\mathcal{A}$ and let M be a tree mapping between $\tilde{x}$ and $\tilde{y}$. Then, we define the *tree mapping matrix* $\boldsymbol{P}(M, \tilde{x}, \tilde{y})$ as a $|\tilde{x}| \times |\tilde{y}|$ matrix with $\boldsymbol{P}(M, \tilde{x}, \tilde{y})_{i,j} = 1$ if $(i, j) \in M$ and 0 otherwise.

Based on the concept of a tree mapping matrix, Bellet, Habrard, and Sebban (2012) have established the following decomposition.

Theorem 2.6. *Let $\tilde{x}$ and $\tilde{y}$ be trees over some alphabet $\mathcal{A}$, let M be a tree mapping between $\tilde{x}$ and $\tilde{y}$, and let c be a cost function over $\mathcal{A}$. Then, for any tree mapping M between $\tilde{x}$ and $\tilde{y}$ it holds:*

$$\begin{aligned} c(M, \tilde{x}, \tilde{y}) = & \sum_{i=1}^{|\tilde{x}|} \sum_{j=1}^{|\tilde{y}|} \boldsymbol{P}(M, \tilde{x}, \tilde{y})_{i,j} \cdot c(x_i, y_j) \\ & + \sum_{i=1}^{|\tilde{x}|} \Big(1 - \sum_{j=1}^{|\tilde{y}|} \boldsymbol{P}(M, \tilde{x}, \tilde{y})_{i,j} \Big) \cdot c(x_i, -) + \sum_{j=1}^{|\tilde{y}|} \Big(1 - \sum_{i=1}^{|\tilde{x}|} \boldsymbol{P}(M, \tilde{x}, \tilde{y})_{i,j} \Big) \cdot c(-, y_j) \end{aligned} \tag{2.26}$$

Proof. Due to its simplicity, Bellet, Habrard, and Sebban (2012) did not provide an explicit proof. Here, we provide the proof for completeness sake. In particular, we inspect the three terms on the right-hand-side of Equation 2.26 separately. First note that $\boldsymbol{P}(M, \tilde{x}, \tilde{y})_{i,j} = 1$ if and only if $(i, j) \in M$ and 0 otherwise. Therefore,

$$\sum_{i=1}^{|\tilde{x}|} \sum_{j=1}^{|\tilde{y}|} \boldsymbol{P}(M, \tilde{x}, \tilde{y})_{i,j} \cdot c(x_i, y_j) = \sum_{(i,j) \in M} c(x_i, y_j)$$

2 Note that our notation differs from the original Notation of Bellet, Habrard, and Sebban (2012). Equation 2.25 corresponds to the optimization problem $GESL_{HL}$ in their paper where we set the margin parameter η_γ to its maximum value $\log(2)$, where we denote B_2 as η, where we obtain $B_1 = \log(2) - \eta$, and where $e_G(x_i, x_j) = d_c(x_i, x_j)$. Also note that the positive neighbors N_i^+ in our notation are precisely the indices j such that $f_{land}(z_i, z_j) = 1$ and $\ell_i = \ell_j$, and that the negative neighbors N_i^- in our notation are precisely the indices j such that $f_{land}(z_i, z_j) = 1$ and $\ell_i \neq \ell_j$.

Accordingly, for all i we have $\left(1 - \sum_{j=1}^{|\tilde{y}|} \boldsymbol{P}_c(\tilde{x}, \tilde{y})_{i,j}\right) = 1$ if there exists no j such that $(i, j) \in M$ and 0 otherwise. Therefore,

$$\sum_{i=1}^{|\tilde{x}|} \left(1 - \sum_{j=1}^{|\tilde{y}|} \boldsymbol{P}_c(\tilde{x}, \tilde{y})_{i,j}\right) \cdot c(x_i, -) = \sum_{i \in I(M,\tilde{x},\tilde{y})} c(x_i, -)$$

By a symmetric argument it holds

$$\sum_{j=1}^{|\tilde{y}|} \left(1 - \sum_{i=1}^{|\tilde{x}|} \boldsymbol{P}_c(\tilde{x}, \tilde{y})_{i,j}\right) \cdot c(-, y_j) = \sum_{j \in J(M,\tilde{x},\tilde{y})} c(-, y_j)$$

such that the right-hand-side of Equation 2.26 is equivalent to the definition of $c(M, \tilde{x}, \tilde{y})$ according to Equation 2.23. □

Recall that it also holds: If M is a cheapest tree mapping between the trees $\tilde{x}$ and $\tilde{y}$ according to c, and if c is non-negative, self-equal, and fulfills the triangular inequality, then it holds: $d_c(\tilde{x}, \tilde{y}) = c(M, \tilde{x}, \tilde{y})$.

If we additionally assume that M remains constant, the optimization problem 2.25 is a simple quadratic problem, which is easy to solve. Moreover, Bellet, Habrard, and Sebban (2012) have shown that the resulting similarity measure s is guaranteed to be good according to the goodness framework of Balcan, Blum, and Srebro (2008).

Note that GESL has several key limitations that we attempt to address in this work. First, GESL does not guarantee that c is self-equal, symmetric, or conforms to the triangular inequality. This is problematic because the learned tree mapping edit distance may not correspond to the actual edit distance and may not be metric. Second, and more importantly, the assumption of a constant cheapest tree mapping does usually not hold. Especially if the triangular inequality is not enforced, alternative tree mappings may quickly become cheaper such that GESL vastly underestimates the actual error (Paaßen, Gallicchio, et al. 2018). Third, GESL chooses the positive and negative neighbors N_i^+ and N_i^- according to a rather ad-hoc rule that is not directly related to predictive performance.

In Chapter 4 we use the basic idea of the decomposition in Equation 2.26 to learn parameters for the tree edit distance, but we ensure metric axioms by means of a different cost function, we consider *all* cooptimal tree mappings instead of a single cooptimal tree mapping, suhc that the constant mapping assumption is easier to fulfill, and we use *prototypes* as reference neighbors. Such prototypes are directly related to the predictive performance of learning vector quantization (LVQ) classifiers, which we discuss in the next section.

2.5 LEARNING VECTOR QUANTIZATION

Learning vector quantization (LVQ) is a family of approaches to classify data via prototypes. In particular, a learning vector quantization (LVQ) approach represents classes in terms of few prototypes $w_1, \ldots, w_K$ with associated prototype labels $z_1, \ldots, z_K \in \{1, \ldots, L\}$, such that data can be classified correctly by assigning the label of the closest prototype (Kohonen 1995). In other words, LVQ is concerned with finding the best possible 1-nearest neighbor classifier using only K data points. More precisely, for M training data points LVQ attempts to solve the optimization problem:

$$\min_{w_1,\ldots,w_K} \sum_{i=1}^{M} \Phi(d_i^+ - d_i^-) \tag{2.27}$$

where d_i^+ is the distance of the ith datum to the closest prototype with the same label, d_i^- is the distance of the ith datum to the closest prototype with a different label, and Φ is the Heaviside function with $\Phi(\mu) = 1$ if $\mu \geq 0$ and $\Phi(\mu) = 0$ otherwise. Note that $d_i^+ - d_i^- \geq 0$ if and only if the ith data point is misclassified such that this loss exactly counts the number of misclassifications in the training data.

Because this problem is NP-hard, we need to apply heuristics (Hoffgen, Simon, and Van Horn 1995). Sato and Yamada (1995) proposed the following differentiable relaxation of the original loss 2.27, which they called generalized learning vector quantization (GLVQ).

$$E_{\text{GLVQ}}\Big((w_1, z_1), \ldots, (w_K, z_K), (x_1, y_1), \ldots, (x_M, y_M)\Big) := \sum_{i=1}^{M} \Phi\Big(\frac{d_i^+ - d_i^-}{d_i^+ + d_i^-}\Big), \tag{2.28}$$

where Φ is some differentiable, monotonic function such as the logistic function $\Phi(\mu) = 1/(1 + \exp(-\mu))$. Given that E_{GLVQ} is differentiable, we can apply standard gradient-based optimization procedures, such as stochastic gradient descent or efficient second-order optimization methods such as L-BFGS (Liu and Nocedal 1989).

GLVQ has multiple theoretical and practical advantages, making it a viable model for a wide range of application scenarios. First, to classify a new data point, we only need to compute the distances to all prototypes, which runs in $\mathcal{O}(K)$ time complexity and is thus very fast. Second, the model complexity of GLVQ can be scaled easily by increasing the number of prototypes, gracefully scaling from a linear classifier to a more and more nonlinear one. Third, learning vector quantization models have been shown to yield maximum-margin generalization bounds (Schneider, Biehl, and Hammer 2009a). Fourth, LVQ supports interpretation as the prototypes provide insight into the classification boundaries used by the classifier. We also note a nice interpretation of the GLVQ cost function E_{GLVQ}, which approximates the number of misclassification if Φ is close to the Heaviside function. We use the GLVQ loss throughout this work. In particular, we learn sequence edit distances according to the GLVQ loss in Chapter 3, we learn tree edit distances according to the GLVQ loss in Chapter 4, and we use GLVQ-based classifiers in Chapters 7 and 8. In the latter two cases, we use an extension of GLVQ with metric learning, namely generalized matrix learning vector quantization (GMLVQ).

2.5.1 *Generalized Matrix Learning Vector Quantization*

In standard GLVQ, d is the (squared) standard Euclidean distance. However, d can also be replaced by a generalized quadratic form

$$d_{\mathbf{\Omega}}(\vec{w}, \vec{x})^2 := (\vec{w} - \vec{x})^\top \cdot \mathbf{\Omega}^\top \cdot \mathbf{\Omega} \cdot (\vec{w} - \vec{x}), \tag{2.29}$$

where $\mathbf{\Omega}$ is some $n \times m$-dimensional projection matrix for some natural number $n \leq m$ (Bunte et al. 2012; Schneider, Biehl, and Hammer 2009a). Note that $\mathbf{\Omega}$ can be interpreted as a linear projection of the data and the prototypes to an auxiliary space where standard GLVQ is applied.

Equivalently, $\mathbf{\Omega}^\top \cdot \mathbf{\Omega}$ can be regarded as manipulating the unit distance ellipse of the Euclidean distance in order to support classification. If $\mathbf{\Omega}$ is treated as a learnable parameter, we obtain a metric learning variant of GLVQ, which we call generalized matrix learning vector quantization (GMLVQ, Schneider, Biehl, and Hammer 2009a). If we learn an individual matrix $\mathbf{\Omega}_k$ for each prototype, we call this scheme local generalized matrix learning vector quantization (LGMLVQ).

A drawback of GLVQ models is that the GLVQ cost function 2.28 is highly non-convex and includes bad local optima. This becomes particularly apparent in case of transfer learning in Chapter 7. For those cases, we suggest the related, generative approach of Gaussian Mixture Models (GMMs).

2.5.2 *Labeled Gaussian Mixture Models*

A Gaussian Mixture Model (GMM) is a generative model of some marginal density $p(\vec{x})$ via a weighted sum of Gaussian distributions as follows.

$$p(\vec{x}) = \sum_{k=1}^{K} p(\vec{x}|k) \cdot P(k), \tag{2.30}$$

where K is the number of Gaussians, $P(k)$ is the prior for the kth Gaussian, and the density $p(\vec{x}|k)$ is given as the density of the m-dimensional Gaussian distribution, that is,

$$p(\vec{x}|k) = \mathcal{N}(\vec{x}|\vec{\mu}_k, \mathbf{\Lambda}_k) = \sqrt{\frac{\det(\mathbf{\Lambda}_k)}{(2 \cdot \pi)^m}} \cdot \exp\Big(-\frac{1}{2} \cdot (\vec{\mu}_k - \vec{x})^\top \cdot \mathbf{\Lambda}_k \cdot (\vec{\mu}_k - \vec{x}) \Big), \tag{2.31}$$

where $\vec{\mu}_k \in \mathbb{R}^m$ is the mean of the kth Gaussian and $\mathbf{\Lambda}_k$ is a positive definite $m \times m$ matrix called the precision matrix of the kth Gaussian, that is, the inverse of the covariance matrix.

Similar to kernel density estimation, it can be shown that any density, which conforms to some very general conditions can be arbitrarily well approximated by a GMM (Barber 2012; Bishop 2006). However, in our case it is not necessary to approximate the marginal distribution of the data well. We care mostly about the posterior distribution $P(y|\vec{x})$. We can integrate label information into GMMs by assuming conditional independence of the label y and the data point $\vec{x}$ given the index of the generating Gaussian k. Then, we obtain the following model for the joint density $p(y, \vec{x})$.

$$p(y, \vec{x}) = \sum_{k=1}^{K} p(y, \vec{x}|k) \cdot P(k) = \sum_{k=1}^{K} P(y|k) \cdot p(\vec{x}|k) \cdot P(k), \tag{2.32}$$

that is, the only additional ingredient we need to model is a probability distribution $P(y|k)$ for each Gaussian. We can use such a model for classification by selecting the label with the largest posterior $P(y|\vec{x})$ according to Bayes' rule.

$$P(y|\vec{x}) = \frac{p(y, \vec{x})}{p(\vec{x})} = \frac{p(y, \vec{x})}{\sum_{y'} p(y', \vec{x})} \tag{2.33}$$

We call this kind of model a labeled Gaussian Mixture Model (lGMM). In Chapters 7 and 8, we use lGMMs for classification and transfer learning. To our knowledge, lGMMs have not been subject to extensive prior research. However, they can be seen as a trivial extension of standard GMMs. Indeed, the optimization strategies for standard GMMs carry over almost unchanged to this case. For previous descriptions of these learning strategies, we refer, for example, to Dempster, Laird, and Rubin (1977), Bishop (2006), and Barber (2012). In the following, we describe learning for the specific extension of lGMMs.

As with standard GMMs, we can learn lGMMs from data by minimizing the negative log-density for all data points, that is:

$$E_{\text{lGMM}} = -\log\Big[\prod_{i=1}^{M} p(y_i, \vec{x}_i)\Big] = \sum_{i=1}^{M} -\log\Big[\sum_{k=1}^{K} P(y|k) \cdot p(\vec{x}|k) \cdot P(k)\Big] \tag{2.34}$$

Note that this loss function is generally non-convex with respect to our parameters of interest. However, we can find a local optimum efficiently via an expectation maximization scheme (Dempster, Laird, and Rubin 1977). In general terms, expectation maximization is an approach to infer optimal parameters in optimization problems with latent variables. The approach has two steps: first, an expectation step in which we compute the posterior of the latent variables given the data and the current parameter values; and second, a maximization step in which we compute the parameter values, which maximize the expected log-likelihood Q of our data given the posterior for the latent variables. As Q always underestimates the actual log-likelihood, iterating these two steps is guaranteed to lead us to a local optimum of the actual log-likelihood (Dempster, Laird, and Rubin 1977).

In our case, we treat the assignment-variables of data points to prototypes k as latent variables, and our expectation and maximization steps take the following form (Bishop 2006; Barber 2012).

Expectation step: For each data point $\vec{x}_i$, we compute the posterior for the Gaussian that has generated the data point according to Bayes' rule.

$$\gamma_{k|i} := P(k|\vec{x}_i, y_i) = \frac{P(y_i|k) \cdot p(\vec{x}_i|k) \cdot P(k)}{\sum_{k'=1}^{K} P(y_i|k') \cdot p(\vec{x}_i|k') \cdot P(k')} \tag{2.35}$$

Maximization step: Assuming fixed posterior $\gamma_{k|i}$ we minimize the negative expected log-likelihood Q of the data with respect to the parameters of interest. The negative expected log-likelihood has the following form.

$$\begin{aligned}
Q &= -\sum_{i=1}^{M}\sum_{k=1}^{K} \gamma_{k|i} \cdot \log\big[p(y_i, \vec{x}_i, k)\big] \\
&= \sum_{i=1}^{M}\sum_{k=1}^{K} \gamma_{k|i} \cdot \Big(-\log\big[P(y_i|k)\big] - \log\big[p(\vec{x}_i|k)\big] - \log\big[P(k)\big]\Big) \\
&= \sum_{i=1}^{M}\sum_{k=1}^{K} \gamma_{k|i} \cdot \Big(-\log\big[P(y_i|k)\big] - \log\big[P(k)\big] \\
&\quad - \frac{1}{2}\log\big[\det(\mathbf{\Lambda}_k)\big] + \frac{m}{2} \cdot \log\big[2 \cdot \pi\big] + \frac{1}{2} \cdot (\vec{\mu}_k - \vec{x}_i)^\top \cdot \mathbf{\Lambda}_k \cdot (\vec{\mu}_k - \vec{x}_i)\Big)
\end{aligned} \tag{2.36}$$

In contrast to the original negative log-likelihood, Q is convex with respect to all our parameters of interest and even permits a closed-form solution.

Theorem 2.7. *Under the assumption of fixed $\gamma_{k|i}$, Q (Equation 2.36) is convex with respect to $P(k)$, $P(y|k)$, $\vec{\mu}_k$, and $\mathbf{\Lambda}_k$.*

Further, the optima of Q with respect to these parameters are given as follows.

$$P(k) = \frac{1}{M} \cdot \sum_{i=1}^{M} \gamma_{k|i} \tag{2.37}$$

$$P(y|k) = \frac{\sum_{i:y_i=y} \gamma_{k|i}}{\sum_{i=1}^{M} \gamma_{k|i}} \tag{2.38}$$

$$\vec{\mu}_k = \frac{\sum_{i=1}^{M} \gamma_{k|i} \cdot \vec{x}_i}{\sum_{i=1}^{M} \gamma_{k|i}} \tag{2.39}$$

$$\mathbf{\Lambda}_k = \Big(\frac{\sum_{i=1}^{M} \gamma_{k|i} \cdot (\vec{\mu}_k - \vec{x}_i) \cdot (\vec{\mu}_k - \vec{x}_i)^\top}{\sum_{i=1}^{M} \gamma_{k|i}} \Big)^{-1} \tag{2.40}$$

Finally, if we restrict the precision matrix to be shared across all Gaussians, that is, $\mathbf{\Lambda}_1 = \ldots = \mathbf{\Lambda}_K = \mathbf{\Lambda}$, we obtain the following optimum of Q with respect to $\mathbf{\Lambda}$.

$$\mathbf{\Lambda} = \Big(\frac{1}{M} \cdot \sum_{i=1}^{M} \sum_{k=1}^{K} \gamma_{k|i} \cdot (\vec{\mu}_k - \vec{x}_i) \cdot (\vec{\mu}_k - \vec{x}_i)^\top \Big)^{-1} \tag{2.41}$$

Proof. The argument mostly follows the description of the expectation maximization algorithm for standard GMMs as described, for example, by Bishop (2006) and Barber (2012). For a version adapted to our notation, please refer to Appendix A.6. □

For the full optimization algorithm we need to initialize $P(k)$, $P(y|k)$, $\vec{\mu}_k$, and $\mathbf{\Lambda}_k$ with some reasonable initial values and then iterate the expectation and the maximization step until convergence.

Note that an initialization of $P(y|k)$ as 1 for some y ensures that $P(y|k)$ stays 1 over the entire course of learning because the posterior $\gamma_{k'|i}$ for some other $k' \neq k$ is zero and thus does not contribute to any of the equations in Theorem 2.7.

Further note that $\mathbf{\Lambda}_k^{-1}$ is guaranteed to be positive semi-definite because it is a convex combination of outer products. In case the determinant of $\mathbf{\Lambda}_k^{-1}$ degenerates to zero, we have to add a small positive number to the diagonal of $\mathbf{\Lambda}_k^{-1}$ to ensure that the density generated by the lGMM still exists (Barber 2012). In the following, we will generally assume that such a treatment of $\mathbf{\Lambda}_k^{-1}$ has been performed and that $\mathbf{\Lambda}_k$ is thus strictly positive definite.

If we restrict all precision matrices $\mathbf{\Lambda}_k$ to be shared across the Gaussians, we call such a model a labeled Gaussian Mixture Model with shared precision matrix (slGMM). Using slGMMs can be advantageous because there are less parameters to optimize, which may speed up optimization and make it more robust. Indeed, we observe such effects empirically in Chapter 8.

Note that maximizing the data likelihood only captures how well our model generates the data, not how well the data is classified by the model. To obtain a discriminative lGMM, we can first train a LVQ model and then construct an lGMM from it as follows (Seo and Obermayer 2003; Schneider, Biehl, and Hammer 2009b). For every prototype $\vec{w}_k$ with label z_k and matrix $\mathbf{\Omega}_k$ we construct one Gaussian with prior $P(k) = 1/K$, with label distribution $P(y|k) = 1$ if $y = z_k$ and $P(y|k) = 0$ otherwise, with mean $\vec{\mu}_k = \vec{w}_k$, and with precision matrix $\mathbf{\Lambda}_k = \frac{1}{\sigma^2} \cdot \mathbf{\Omega}_k^\top \cdot \mathbf{\Omega}_k$, where $\sigma > 0$ is a hyper-parameter that regulates the crispness of the lGMM classification. For small σ, the decision of the lGMM becomes equivalent to LVQ classification (Seo and Obermayer 2003; Schneider, Biehl, and Hammer 2009b). In case we construct an lGMM from a generalized matrix learning vector quantization (GMLVQ) model, we obtain a slGMM.

Up to this point, all models we discussed were limited to vectorial data. Our focus, however, is structured data. To classify structured data, we need to turn to purely distance-based classification schemes, such as relational generalized learning vector quantization (RGLVQ).

2.5.3 *Relational Generalized Learning Vector Quantization*

Relational generalized learning vector quantization (RGLVQ) is essentially a GLVQ model which assumes that the distance measure used is Euclidean and that the prototypes are convex combinations of training data points (Hammer, D. Hofmann, et al. 2014). More precisely, if the distance d is Euclidean with spatial mapping $\phi : \mathcal{X} \rightarrow \mathbb{R}^m$, and we are given the training dataset $x_1, \ldots, x_M$, we assume that for all $k \in \{1, \ldots, K\}$ it holds:

$$\phi(w_k) = \boldsymbol{X} \cdot \vec{\alpha}_k$$

for $\boldsymbol{X} = \big(\phi(x_1), \ldots, \phi(x_M)\big) \in \mathbb{R}^{m \times M}$ and some coefficient vector $\vec{\alpha}_k \in \mathbb{R}^M$ such that $\sum_{i=1}^M \alpha_{k,i} = 1$ and $\alpha_{k,i} \geq 0$ for all i. This assumption is equivalent to stating that all prototypes should lie in the convex hull of the columns of $\boldsymbol{X}$, which is a reasonable constraint to ensure that the prototypes do not degenerate to arbitrary positions.

Accordingly, we can use Equation 2.11 to compute the distance $\|\phi(x) - \phi(w_k)\|$ between any data point x and any prototype w_k. RGLVQ learns the coefficient vectors $\vec{\alpha}_k$ by minimizing the GLVQ cost function 2.28 with respect to it (Hammer, D. Hofmann, et al. 2014). In Chapter 3, we use RGLVQ for sequence edit distance learning.

RGLVQ has two key limitations. First, if the distance d is not Euclidean, then prototype-to-data distances may become negative, which can lead to degenerate cases. Second, the runtime is considerably worse compared to standard GLVQ, because each distance computation according to Equation 2.11 requires linear time in the number of nonzero entries of $\vec{\alpha}_k$, with an additional quadratic runtime complexity to compute the second term in Equation 2.11. Median generalized learning vector quantization (MGLVQ) addresses both of these issues.

2.5.4 *Median Generalized Learning Vector Quantization*

In contrast to other LVQ approaches, median generalized learning vector quantization (MGLVQ) asserts that prototypes should be a strict subset of the training dataset, that is, for each prototype w_k there exists an index i_k such that $w_k = x_{i_k}$ (Nebel, Hammer, et al. 2015).

This restriction has several advantages. First, we can interpret every prototype, because its explicit form is given in terms of the data point x_{i_k}. Second, we can compute the distance $d(x, w_k)$ between a data point x and the kth prototype w_k as $d(x, x_{i_k})$, which is possible in constant time. Third, we can therefore perform classification in $\mathcal{O}(K)$ because we only need to compute K distances to determine the classification. Finally, we do not need to make any assumptions regarding the properties of the distance d, except non-negativity (Nebel, Hammer, et al. 2015).

The key disadvantage of restricting prototypes to be data points is that it complicates optimization. The prototype position is now a discrete, non-differentiable choice, which can not be improved by gradient-based methods. To address this issue, Nebel, Hammer, et al. (2015) have devised a generalized expectation maximization scheme for the surrogate problem

$$\max_{w_1,\ldots,w_K} \sum_{i=1}^{M} \log\left(\alpha + \frac{d_i^- - d_i^+}{d_i^- + d_i^+}\right) \tag{2.42}$$

where $\alpha \in \mathbb{R}$ with $\alpha \geq 4$ is a hyper-parameter and d_i^+ and d_i^- are defined as before.

In particular, the optimization scheme consists of the two following steps.

Expectation: Compute the pseudo-probabilities

$$\gamma_i^+ = \frac{g_i^+}{(g_i^+ + g_i^-)} \quad \text{and} \quad \gamma_i^- = \frac{g_i^-}{(g_i^+ + g_i^-)} \qquad \text{where}$$
$$g_i^+ = \frac{\alpha}{2} - \frac{d_i^+}{(d_i^+ + d_i^-)} \quad \text{and} \quad g_i^- = \frac{\alpha}{2} + \frac{d_i^-}{(d_i^+ + d_i^-)}$$

Maximization: Assuming fixed γ_i^+ and γ_i^-, but variable g_i^+ and g_i^-, switch the location of a single prototype to increase the pseudo-likelihood

$$\mathcal{L} = \sum_{i=1}^{M} \gamma_i^+ \cdot \log(g_i^+/\gamma_i^+) + \gamma_i^- \cdot \log(g_i^-/\gamma_i^-) \tag{2.43}$$

If no such prototype exists, the optimization scheme stops.

This optimization scheme provably converges to a local maximum of the loss 2.42, which in turn is (close to) a local minimum of the GLVQ loss 2.28 for sufficiently large α.

Theorem 2.8. *Let $x_1, \ldots, x_M$ be elements from some set $\mathcal{X}$ with labels $y_1, \ldots, y_M \in \{1, \ldots, L\}$ and let $w_1, \ldots, w_K \subseteq \{x_1, \ldots, x_M\}$.*

Then, the expectation maximization scheme above converges to a local optimum of the loss 2.42.

Further, the sum of the GLVQ loss 2.28 with nonlinearity $\Phi(\mu) = \log(\alpha + \mu)$ and the loss 2.42 lies in $\mathcal{O}\big(2 \cdot M \cdot \log(\alpha) - \frac{1}{\alpha^2} \cdot \sum_{i=1}^{M} \mu_i^2\big)$ with $\mu_i = (d_i^+ - d_i^-)/(d_i^+ + d_i^-)$.

Proof. The first claim has been proven by Nebel, Hammer, et al. (2015). For a proof adapted to our notation, and for a proof of the second claim, refer to Appendix A.7. □

Since the sum of the losses 2.28 and 2.42 becomes a constant for large enough α, we can replace the minimization of loss 2.28 with the maximization of loss 2.42.

Note that, in principle, any maximization step requires $\mathcal{O}(K \cdot M^2)$ operations because for every prototype we have to check $\mathcal{O}(M)$ possible data points we could switch to and for each of those we have to compute the new likelihood $\mathcal{L}$, which has $\mathcal{O}(M)$ terms. In practice, the optimization is considerably faster because we only switch a prototype to data points with the same label within its own Voronoi cell, and to compute the likelihood we only have to consider points for which d_i^+ or d_i^- changes. In Chapter 4, we use MGLVQ for tree edit distance learning.

This concludes our discussion of distance-based classification. We now turn to the topic of distance-based time series prediction.

2.6 DISTANCE-BASED TIME SERIES PREDICTION

While distance-based dimensionality reduction, clustering, and classification are already well covered in the literature (refer e.g. to Gisbrecht and Schleif 2015; Hammer and Hasenfuss 2007; Hammer, D. Hofmann, et al. 2014), distance-based time series prediction is to date limited to vectorial data. For example, support vector regression takes kernels as input and has been applied to predict time series in finance, business, environmental research, and engineering (Sapankevych and Sankar 2009). Another example is Gaussian process regression, which has been applied to predict chemical processes (Girard et al. 2003), motion data (J. Wang, Hertzmann, and Blei 2006), and physics data (Roberts et al. 2012). In Chapter 5, we generalize time series prediction to cases where the vectorial representation is implicit and not Euclidean.

We investigate three existing non-parametric regression techniques for prediction, namely one-nearest neighbor regression (1-NN), kernel regression (KR), and GPR. For the purpose of this background chapter we assume that the data does have vectorial form, and we cover the nonvectorial case in Chapter 5.

Now, assume that we are given a dataset of the form $(\vec{x}_1, \vec{y}_1), \ldots, (\vec{x}_M, \vec{y}_M) \in \mathbb{R}^m \times \mathbb{R}^m$, where $\vec{y}_i$ is the successor of $\vec{x}_i$ in a time series. Then, our aim is to find a predictive function $f : \mathbb{R}^m \to \mathbb{R}^m$ such that $f(\vec{x}_i) \approx \vec{y}_i$ for all i and such that the general underlying dynamics of our training dataset are captured. Note that our setup here already makes a Markov assumption, meaning that the state at time step $t+1$ is conditionally independent from the state at time steps $0, \ldots, t-1$ if conditioned on the state at time step t. We cover the more general case without Markov assumption in Chapter 5 as well.

To illustrate the difference between the three predictive schemes, we consider the two-dimensional dynamical system $\frac{\partial}{\partial t} f(\vec{x}) = \frac{1}{2}(1 - \|\vec{x}\|) \cdot \vec{x} + 0.6 \cdot (-x_2, x_1)^\top$ illustrated in Figure 2.7 (top left). The dynamical system has a cyclic attractor at the unit circle and an instable fix point at the origin. From every other position in the two-dimensional space, points are pulled towards the cyclic attractor and move along the unit circle in counter-clockwise direction. The training data for our predictions consists of twenty points $\vec{x}_i$ selected uniformly at random from the interval $[-1.5, 1.5]^2$ and shown in orange in the figure. We define the desired next state via an Euler step $\vec{y}_i = \vec{x}_i + \frac{\partial}{\partial t} f(\vec{x}_i)$. As distance measure d we use the standard Euclidean distance, and as kernel k we use the radial basis function:

$$k_{d,\xi}(\vec{x}, \vec{x}') = \exp\Big(-\frac{1}{2} \cdot \frac{d(\vec{x}, \vec{x}')^2}{\xi^2} \Big) \tag{2.44}$$

where $\xi \in \mathbb{R}$ with $\xi > 0$ is a hyper-parameter, which we call *bandwidth*, set to 0.6 in this example. Note that the radial basis function is guaranteed to be a kernel for any

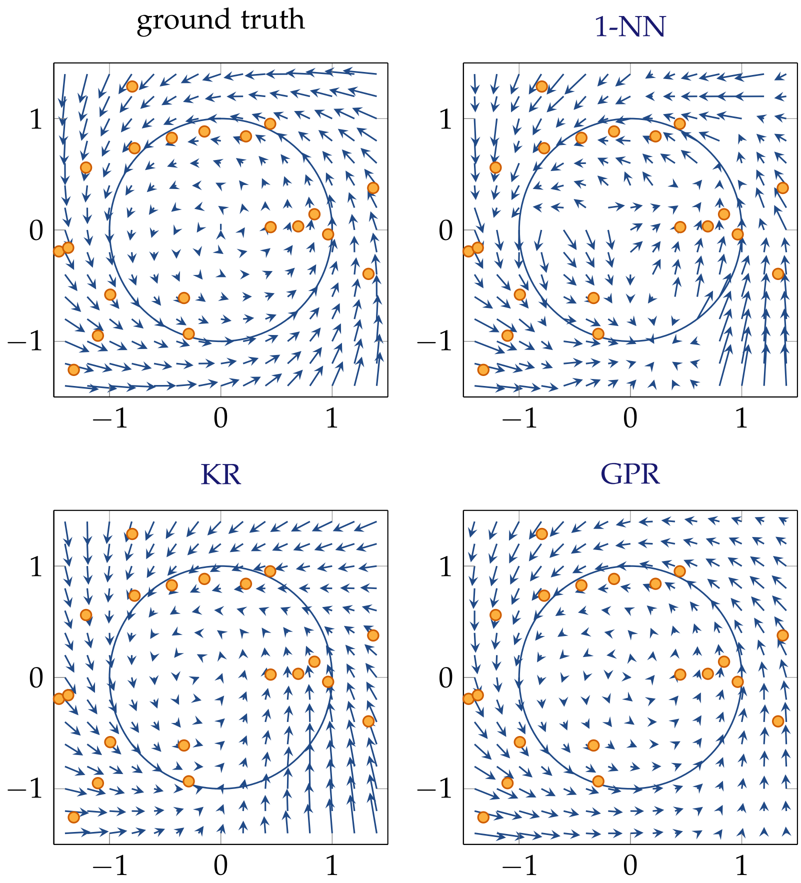


Figure 2.7: An illustration of 1-NN, KR, and GPR in predicting a dynamical system. Top left: The true underlying dynamical system. The circle marks the circle attractor of the system. Other panels: The predictions made by 1-NN, KR, and GPR respectively, based on the training data points shown in orange.

Euclidean distance, but not for general metrics (Jäkel, Schölkopf, and Wichmann 2008). For example, edit distances are metrics, but do generally not yield kernels via the radial basis function. Another property of the radial basis function is that it can be readily interpreted as a measure of similarity, in the sense that it decreases monotonously with the distance, and that it reaches its maximum of 1 if and only if $\vec x = \vec x'$ (Jäkel, Schölkopf, and Wichmann 2008; Nebel, Kaden, et al. 2017).

Equipped with the radial basis function and our example, we can now inspect 1-NN, KR, and GPR in more detail.

One-nearest neighbor regression (1-NN): We define the predictive function for 1-NN as follows.

$$f(\vec x) := \vec y_{i^+} \qquad \text{where } i^+ = \operatorname*{argmin}_{i \in \{1,\ldots,M\}} d(\vec x, \vec x_i) \tag{2.45}$$

Figure 2.7 (top right) displays the prediction of 1-NN for the dynamical system example. As is clearly visible, the prediction is relatively inaccurate and suffers from discontinuous changes. These are caused by the discontinuity of the argmin function. In particular, the argmin function is ill-defined for points $\vec x$ where two different training data points $\vec x_i$

and $\vec{x}_j$ exist such that $d(\vec{x}, \vec{x}_i) = d(\vec{x}, \vec{x}_j)$ but $\vec{y}_i \neq \vec{y}_j$. A straightforward way to smoothen the prediction is to utilize averages of training data with continuous weights, which is the technique employed by KR.

Kernel regression (KR): KR was first proposed by Nadaraya (1964) and can be seen as a generalization of 1-NN to a smooth predictive function f by weighting training data points according to their distance. In particular, let s_d be any non-negative function that decreases monotonously with the distance d. Then, the predictive function of KR is given as:

$$f(\vec{x}) := \frac{\sum_{i=1}^{M} s_d(\vec{x}, \vec{x}_i) \cdot \vec{y}_i}{\sum_{i=1}^{M} s_d(\vec{x}, \vec{x}_i)} \tag{2.46}$$

Note that KR requires for each possible input $\vec{x}$ at least one training data point with $s(\vec{x}, \vec{x}_i) > 0$, that is, if the test data point is not similar to any training data point, the prediction degenerates. Another limitation of KR is that it generally does not reproduce the training data, i.e. $f(\vec{x}_i) \neq \vec{y}_i$. This also results in a somewhat inaccurate prediction for the dynamical system example, as shown in Figure 2.7 (bottom left). While KR predicts the global behaviour roughly correctly, the predictions especially for the bottom right of the state space are considerably off. To achieve a more accurate prediction, we turn to Gaussian process regression.

Gaussian process regression (GPR): In GPR we assume that the output points (training as well as test) are a realization of a multivariate random variable with a Gaussian distribution (Rasmussen and Williams 2005). The model extends KR in several ways. First, we can encode prior knowledge regarding the output points via the mean of our prior distribution, denoted as $\vec{\theta}_i$ and $\vec{\theta}$ for $\vec{y}_i$ and $\vec{y}$ respectively. Second, we can cover Gaussian noise on our training output points within our model. For this noise, we assume mean 0 and standard deviation $\tilde{\sigma}$.

Let now k be a kernel on $\mathcal{X}$, let

$$\vec{k} := \big(k(\vec{x}, \vec{x}_1), \ldots, k(\vec{x}, \vec{x}_M)\big)^\top \text{ and let} \tag{2.47}$$

$$\boldsymbol{K} := \big(k(\vec{x}_i, \vec{x}_j)\big)_{i,j=1\ldots M} \tag{2.48}$$

Then, under the GPR model, the conditional probability density of the output points $\vec{y}_1, \ldots, \vec{y}_M, \vec{y}$ given the input points $\vec{x}_1, \ldots, \vec{x}_M, \vec{x}$ is given as follows.

$$p(\vec{y}_1, \ldots, \vec{y}_M, \vec{y} | \vec{x}_1, \ldots, \vec{x}_M, \vec{x}) =$$
$$\mathcal{N}\left(\vec{y}_1, \ldots, \vec{y}_M, \vec{y} \middle| \vec{\theta}_1, \ldots, \vec{\theta}_M, \vec{\theta}, \begin{pmatrix} \boldsymbol{K} + \tilde{\sigma}^2 \cdot \boldsymbol{I}^M & \vec{k} \\ \vec{k}^\top & k(\vec{x}, \vec{x}) \end{pmatrix}^{-1} \right)$$

where $\boldsymbol{I}^M$ is the M-dimensional identity matrix and $\mathcal{N}(\cdot|\vec{\mu}, \boldsymbol{\Lambda})$ is the multivariate Gaussian probability density function for mean $\vec{\mu}$ and precision matrix $\boldsymbol{\Lambda}$. Note that our assumed distribution takes *all* outputs $\vec{y}_1, \ldots, \vec{y}_M, \vec{y}$ as argument, not just a single point. The posterior distribution for just $\vec{y}$ can be obtained by marginalization as follows.

Theorem 2.9 (Gaussian Process Posterior Distribution)**.** *Let $\boldsymbol{Y}$ be the matrix $(\vec{y}_1, \ldots, \vec{y}_M)$ and $\boldsymbol{\Theta} := (\vec{\theta}_1, \ldots, \vec{\theta}_M)$. Then the posterior density function for Gaussian process regression is* *given as:*

$$p(\vec{y}|\vec{x}, \vec{x}_1, \ldots, \vec{x}_M, \vec{y}_1, \ldots, \vec{y}_M) = \mathcal{N}(\vec{y}|\vec{\mu}, \sigma^{-2} \cdot \boldsymbol{I}^m) \quad \textit{where} \tag{2.49}$$

$$\vec{\mu} = \vec{\theta} + (\boldsymbol{Y} - \boldsymbol{\Theta}) \cdot (\boldsymbol{K} + \tilde{\sigma}^2 \cdot \boldsymbol{I}^M)^{-1} \cdot \vec{k} \tag{2.50}$$

$$\sigma^2 = k(\vec{x}, \vec{x}) - \vec{k}^\top \cdot (\boldsymbol{K} + \tilde{\sigma}^2 \cdot \boldsymbol{I}^M)^{-1} \cdot \vec{k} \tag{2.51}$$

We call $\vec{\mu}$ the predictive mean *and σ^2 the* predictive variance.

Proof. Refer e.g. to Rasmussen and Williams (2005, p. 27). □

Note that the posterior distribution is, again, Gaussian. For a Gaussian distribution, the mean corresponds to the point of maximum density, such that we can define our predictive function as $f(\vec{x}) := \vec{\mu}$ where $\vec{\mu}$ is the predictive mean of the posterior distribution for point $\vec{x}$. Further note that the predictive mean becomes the prior mean if $\vec{k}$ is the zero vector, i.e. if the test data point is dissimilar to all training data points.

Figure 2.7 (bottom right) shows the predictions of GPR for the dynamical system example with the prior being the identity, i.e. $\vec{\theta}_i = \vec{x}_i$ for all i. Apparently, GPR captures the actual underlying dynamical system quite well. The main drawback of GPR is the high computational complexity: For training, the inversion of the matrix $(\boldsymbol{K} + \tilde{\sigma}^2 \cdot \boldsymbol{I}^M)^{-1}$ requires cubic time. This issue can be addressed by several approximation schemes such as using only a subset of the data for training, or using a low-rank approximation of the kernel matrix such as the Nyström method (Rasmussen and Williams 2005). In this work, we focus on the state-of-the-art approximation scheme entitled robust Bayesian committee machine (rBCM) (Deisenroth and Ng 2015).

The rBCM relies on a partition of the training samples into C disjoint sets, ideally a clustering in the input data. For each of these sets, we perform a separate GPR, yielding the predictive distributions $\mathcal{N}(\vec{x}|\vec{\mu}_c, \sigma_c^{-2} \cdot \boldsymbol{I}^m)$ for $c \in \{1, \ldots, C\}$. These distributions are combined to the final predictive distribution $\mathcal{N}(\vec{x}|\vec{\mu}_{\text{rBCM}}, \sigma_{\text{rBCM}}^{-2} \cdot \boldsymbol{I}^m)$ with the following variance and mean.

$$\sigma_{\text{rBCM}}^{-2} = \sum_{c=1}^{C} \frac{\beta_c}{\sigma_c^2} + \left(1 - \sum_{c=1}^{C} \beta_c\right) \cdot \frac{1}{\sigma_{\text{prior}}^2} \tag{2.52}$$

$$\vec{\mu}_{\text{rBCM}} = \sigma_{\text{rBCM}}^2 \cdot \left(\sum_{c=1}^{C} \frac{\beta_c}{\sigma_c^2} \cdot \vec{\mu}_c + \left(1 - \sum_{c=1}^{C} \beta_c\right) \cdot \frac{1}{\sigma_{\text{prior}}^2} \cdot \vec{\theta} \right) \tag{2.53}$$

where $\sigma_{\text{prior}}^2 > 0$ is a hyper-parameter for the assumed variance of the prior distribution, and $\beta_c > 0$ are weights for the importance of the cth GPR for the current prediction. We follow the suggestion of Deisenroth and Ng (2015) and set $\beta_c = \frac{1}{2} \cdot \left(\log(\sigma_{\text{prior}}^2) - \log(\sigma_c^2) \right)$, also called the *differential entropy*. This setting assigns a higher weight for the cth GPR if its prediction has lower variance.

The rBCM runs in linear time if the size of any single cluster is considered to be constant (i.e. the number of clusters is proportional to M) such that we only need to invert kernel matrices of constant size.

In Chapter 5, we evaluate all of these methods for the purpose of time series prediction for data, which are represented in terms of pairwise distances.

This concludes our description of background knowledge for this thesis. In the following chapters, we build upon this background knowledge to push the boundaries of learning distances and distance-based learning. We begin by learning parameters of the sequence edit distance.

3 SEQUENCE EDIT DISTANCE LEARNING

Summary: Sequence edit distances are efficient, popular, and interpretable distance measures in many application domains, especially for RNA, DNA, and protein sequence processing in biology. A challenge in applying such edit distances is that their default parameters may not be optimal for the task at hand. In this chapter, we develop a novel, flexible metric learning approach for sequence edit distances, and we evaluate our approach on datasets from biology and intelligent tutoring systems.

Publications: This chapter is based on the following publications.

- Mokbel, Bassam, Benjamin Paaßen, et al. (2015). "Metric learning for sequences in relational LVQ". English. In: *Neurocomputing* 169, pp. 306–322. DOI: 10.1016/j.neucom.2014.11.082.
- Paaßen, Benjamin, Bassam Mokbel, and Barbara Hammer (2015a). "A Toolbox for Adaptive Sequence Dissimilarity Measures for Intelligent Tutoring Systems". In: *Proceedings of the 8th International Conference on Educational Data Mining (EDM 2015)*. (Madrid, Spain). Ed. by Olga Christina Santos et al. International Educational Datamining Society, pp. 632–632. URL: http://www.educationaldatamining.org/EDM2015/uploads/papers/paper_257.pdf.
- — (2015b). "Adaptive structure metrics for automated feedback provision in Java programming". English. In: *Proceedins of the 23rd European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN 2015)*. (Bruges, Belgium). Ed. by Michel Verleysen. **Best student paper award**. i6doc.com, pp. 307–312. URL: http://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2015-43.pdf.
- — (2016). "Adaptive structure metrics for automated feedback provision in intelligent tutoring systems". In: *Neurocomputing* 192, pp. 3–13. DOI: 10.1016/j.neucom.2015.12.108.

Source Code: The MATLAB(R) source code for relational generalized learning vector quantization is available at http://www.techfak.uni-bielefeld.de/~xzhu/published_code/relational_glvq.zip.

The Java(R) source code for sequence edit distances and gradients thereof is available at https://openresearch.cit-ec.de/projects/tcs.

Sequence edit distances provide an intuitive measure of distance between two sequences $\bar{x}$ and $\bar{y}$ by counting the number of characters that have to be deleted, inserted, or replaced to transform $\bar{x}$ into $\bar{y}$. While originally devised to count the number of spelling errors in written text (Levenshtein 1965; Damerau 1964), sequence edit distances have become popular far beyond this initial application domain. Most importantly, sequence edit distances serve as models of distance between RNA, DNA, or protein sequences in biology (S. Henikoff and J. G. Henikoff 1992; Hourai, Akutsu, and Akiyama 2004; Kann, Qian, and Goldstein 2000; McKenna et al. 2010; Saigo, Vert, and Akutsu 2006; T. F. Smith and Waterman 1981). Recently, sequence edit distances have also been suggested as actionable measures of distance for intelligent tutoring systems (Gross, Mokbel, et al. 2014; Mokbel, Gross, et al. 2013; Rivers and Koedinger 2015; Price, Dong, and Lipovac 2017). In particular, edit distances could help students to solve a learning task by telling them what precisely they have to change in their current solution attempt to arrive at a correct solution.

A challenge in applying sequence edit distances in practice is that the default parametrization may not be suitable for the domain at hand, that is, not every character may be equidistant from all other characters. For example, in counting spelling errors, not every kind of misspelling is equally likely because some characters are closer to each other on a keyboard (F. Ahmad and Kondrak 2005). In biology, some bases are more likely to change into a specific other base compared to others (Hourai, Akutsu, and Akiyama 2004; Kann, Qian, and Goldstein 2000; Saigo, Vert, and Akutsu 2006). Finally, in intelligent tutoring systems, different syntactic parts of a student solution may be easier to replace, for example due to functional equivalence (Mokbel, Gross, et al. 2013; Paaßen, Jensen, and Hammer 2016). This begs the question how edit distances can be *adapted* to be better suited for the domain and task at hand, that is, how to perform metric learning on edit distances (Bellet, Habrard, and Sebban 2012; Bellet, Habrard, and Sebban 2014).

In this chapter, we provide a general-purpose scheme to learn metric parameters of a broad class of sequence edit distances for classification, based on algebraic dynamic programming (ADP, Giegerich, Meyer, and Steffen 2004, refer to Section 2.3.2). We extend the state of the art in the field in several respects:

- Our metric learning scheme is applicable to a broad class of edit distances, whereas existing approaches focus on a specific type of sequence edit distance (Bellet, Habrard, and Sebban 2014).
- We select reference pairs for metric learning in a principled fashion based on learning vector quantization prototypes for each class instead of ad-hoc selection schemes in prior approaches (Bellet, Habrard, and Sebban 2012; Bellet, Habrard, and Sebban 2014).
- Our approach is compatible with any differentiable parametrization of the edit distance, whereas prior work is limited to learning pairwise symbol replacement costs (Bellet, Habrard, and Sebban 2014).

In the following section, we describe our method in more detail, before we go on to evaluate it experimentally. We conclude this chapter with a short summary and a list of limitations that we intend to address in the next chapter.

3.1 METHOD

We begin our method description by establishing a general-purpose algorithm to compute sequence edit distances based on algebraic dynamic programming (ADP, Giegerich, Meyer, and Steffen 2004, also refer to Section 2.3.2). We then show how to learn parameters of these sequence edit distances via gradient-based optimization.

Algebraic Dynamic Programming

First, recall that a sequence edit distance between two input sequences $\bar{x}$ and $\bar{y}$ is defined as the cost $c(\bar{\delta}, \bar{x})$ of the cheapest edit script $\bar{\delta}$ over some edit set Δ, such that $\bar{\delta}(\bar{x}) = \bar{y}$ (also refer to Definitions 2.5 and 2.6).

Also recall our alternative formalism to express sequence edit distances via ADP. According to ADP, a sequence edit distance between two input sequences $\bar{x}$ and $\bar{y}$ is defined as the cost $c_{\mathcal{F}}(\tilde{\delta})$ of the cheapest script tree $\tilde{\delta}$ according to some algebra $\mathcal{F}$, such that $\tilde{\delta}$ can be generated by some edit tree grammar $\mathcal{G}$, and such that the yield of $\tilde{\delta}$ is exactly $\mathcal{Y}(\tilde{\delta}) = (\bar{x}, \bar{y})$ (also refer to Definition 2.10). However, to our knowledge, the existing literature on ADP does not show that the cheapest edit script and the cheapest script tree are indeed equivalent, and that both notions of edit distance are thus equivalent. Therefore, we prove this result here.

Theorem 3.1. *Let $\mathcal{A}$ be an alphabet with $\$, \text{match} \notin \mathcal{A}$, let $\mathcal{S} = (\text{Del}, \text{Rep}, \text{Ins})$ be a signature with $\$, \text{match} \notin \text{Del} \cup \text{Rep} \cup \text{Ins}$, and let $\mathcal{F}$ be an algebra over $\mathcal{S}$ and $\mathcal{A}$. Finally, let $\tilde{\delta} \in \mathcal{T}(\mathcal{S}, \mathcal{A})$ be a script tree and let $(\bar{x}, \bar{y}) := \mathcal{Y}(\tilde{\delta})$ be the yield of $\tilde{\delta}$. Then, there exists an edit script $\bar{\delta}_{\tilde{\delta}} \in \Delta_{\mathcal{S},\mathcal{A}}$ such that $\bar{y} = \bar{\delta}_{\tilde{\delta}}(\bar{x})$ and $c_{\mathcal{F}}(\tilde{\delta}) = c_{\mathcal{F}}(\bar{\delta}_{\tilde{\delta}}, \bar{x})$.*

Now, let $\mathcal{F}$ conform to the following conditions.

$$\begin{aligned}
\forall \text{rep} \in \text{Rep} : \forall x, y \in \mathcal{A} : c_{\text{rep}}(x, y) \geq 0 \\
\forall \text{del} \in \text{Del} : \forall x \in \mathcal{A} : c_{\text{del}}(x) \geq 0 \\
\forall \text{ins} \in \text{Ins} : \forall y \in \mathcal{A} : c_{\text{ins}}(y) \geq 0 \\
\forall \text{rep}, \text{rep}' \in \text{Rep} : \forall x, y, z \in \mathcal{A} : c_{\text{rep}'}(x, y) + c_{\text{rep}}(y, z) \geq c_{\text{rep}}(x, y) \\
\forall \text{rep} \in \text{Rep} : \forall \text{ins} \in \text{Ins} : \forall x, y \in \mathcal{A} : c_{\text{ins}}(x) + c_{\text{rep}}(x, y) \geq c_{\text{ins}}(y) \\
\forall \text{del} \in \text{Del} : \forall \text{rep} \in \text{Rep} : \forall x, y \in \mathcal{A} : c_{\text{rep}}(x, y) + c_{\text{del}}(y) \geq c_{\text{del}}(x)
\end{aligned}$$

Then, for all edit scripts $\bar{\delta} \in \Delta_{\mathcal{S},\mathcal{A}}$ and all $\bar{x} \in \mathcal{A}^*$, there exists a script tree $\tilde{\delta}_{\bar{\delta},\bar{x}}$, such that $\mathcal{Y}(\tilde{\delta}_{\bar{\delta},\bar{x}}) = (\bar{x}, \bar{\delta}(\bar{x}))$ and $c_{\mathcal{F}}(\tilde{\delta}_{\bar{\delta},\bar{x}}) \leq c_{\mathcal{F}}(\bar{\delta}, \bar{x})$.

Further, it holds for all sequences $\bar{x}, \bar{y} \in \mathcal{A}^*$:

$$d_{\mathcal{S},\mathcal{F}}(\bar{x}, \bar{y}) = \min_{\tilde{\delta} \in \mathcal{T}(\mathcal{S},\mathcal{A})} \{c_{\mathcal{F}}(\tilde{\delta}) | \mathcal{Y}(\tilde{\delta}) = (\bar{x}, \bar{y})\} \tag{3.1}$$

Proof. Refer to Appendix A.8. □

As an example for the first construction in Theorem 3.1, consider the example sequences $\bar{x} = \texttt{ab}$ and $\bar{y} = \texttt{cd}$ over the alphabet $\mathcal{A} = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}$, and the script tree $\tilde{\delta} = \text{del}\Big(\texttt{a}, \text{ins}\big(\text{rep}(\texttt{b}, \$, \texttt{d}), \texttt{c}\big)\Big)$ over the signature $\mathcal{S}_{\text{ALI}} = (\{\text{del}\}, \{\text{rep}\}, \{\text{ins}\})$. This script tree would be translated into a edit script as follows. We first initialize our edit script as $\bar{\delta}_{\$} = \epsilon$. Next, consider the subtree $\text{rep}(\texttt{b}, \$, \texttt{d})$, which corresponds to the edit script $\bar{\delta}_{\text{rep}(\texttt{b},\$,\texttt{d})} = \text{rep}_{1,\texttt{d}}$. Further, consider the subtree $\text{ins}\big(\text{rep}(\texttt{b}, \$, \texttt{d}), \texttt{c}\big)$, which then corresponds to the edit script $\bar{\delta}_{\text{ins}\left(\text{rep}(\texttt{b},\$,\texttt{d}),\texttt{c}\right)} = \text{ins}_{1,\texttt{c}}\text{rep}_{2,\texttt{d}}$. Note that we have increased the index of the replacement operation by one. Finally, consider the entire script tree $\tilde{\delta}$, which then corresponds to the edit script $\bar{\delta}_{\tilde{\delta}} = \text{del}_1\text{ins}_{1,\texttt{c}}\text{rep}_{2,\texttt{d}}$. Note that this edit script does indeed map $\bar{x}$ to $\bar{y}$ and has the costs $c_{\mathcal{F}}(\bar{\delta}, \bar{x}) = c_{\text{del}}(\texttt{a}) + c_{\text{ins}}(\texttt{c}) + c_{\text{rep}}(\texttt{b}, \texttt{d}) = c_{\mathcal{F}}(\tilde{\delta})$ for any algebra $\mathcal{F}$.

As an example for the second construction in Theorem 3.1, consider the example sequences $\bar{x} = \mathtt{a}$ and $\bar{y} = \mathtt{b}$ over the alphabet $\mathcal{A} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$, and the edit script $\bar{\delta} = \text{ins}_{1,\mathtt{c}}\text{rep}_{1,\mathtt{b}}\text{del}_2$ over the edit set $\Delta_{\mathcal{S}_{\text{ALI}},\mathcal{A}}$. This edit script would be translated into a script tree as follows. We first initialize our script tree as $\tilde{\delta}_{\epsilon,\mathtt{a}} = \text{match}(\mathtt{a}, \$, \mathtt{a})$. Next, consider the first edit $\delta_1 = \text{ins}_{1,\mathtt{c}}$, which changes our script tree to $\tilde{\delta}_{\text{ins}_{1,\mathtt{c}},\mathtt{a}} = \text{ins}\big(\text{match}(\mathtt{a}, \$, \mathtt{a}), \mathtt{c}\big)$. Further, consider the second edit $\delta_2 = \text{rep}_{1,\mathtt{b}}$, which changes our script tree to $\tilde{\delta}_{\text{ins}_{1,\mathtt{c}}\text{rep}_{1,\mathtt{b}},\mathtt{a}} = \text{ins}\big(\text{match}(\mathtt{a}, \$, \mathtt{a}), \mathtt{b}\big)$. Note that the insertion operation now inserts $\mathtt{b}$ instead of $\mathtt{c}$. Finally, consider the last edit $\delta_3 = \text{del}_2$, which changes our script tree to $\tilde{\delta}_{\bar{\delta},\mathtt{a}} = \text{ins}\big(\text{del}(\mathtt{a}, \$), \mathtt{b}\big)$. Note that the yield of this script tree is indeed $\mathcal{Y}(\tilde{\delta}_{\bar{\delta},\mathtt{a}}) = (\mathtt{a}, \mathtt{b}) = (\bar{x}, \bar{y})$ and that the costs are $c_{\mathcal{F}}(\tilde{\delta}_{\bar{\delta},\mathtt{a}}) = c_{\text{del}}(\mathtt{a}) + c_{\text{ins}}(\mathtt{b}) \leq c_{\text{ins}}(\mathtt{c}) + c_{\text{rep}}(\mathtt{c}, \mathtt{b}) + c_{\text{del}}(\mathtt{a}) = c_{\mathcal{F}}(\bar{\delta}, \mathtt{a})$ for any algebra $\mathcal{F}$ that conforms to the conditions in Theorem 3.1.

Another result that is missing from the previous literature on ADP is the proof of metric conditions of the resulting sequence edit distance. To us, such a result is important because we need to ensure that at least a pseudo-Euclidean embedding of the edit distance exists in order to apply some distance-based classifiers, such as RGLVQ. We prove metric properties of ADP sequence edit distances in the following theorem.

Theorem 3.2. *Let $\mathcal{A}$ be an alphabet, let $\mathcal{S} = (\text{Del}, \text{Rep}, \text{Ins})$ be a non-trivial signature, and let $\mathcal{F}$ be an algebra over $\mathcal{S}$ and $\mathcal{A}$. Further, let $\Delta_{\mathcal{S},\mathcal{A}}$ be the edit set with respect to $\mathcal{S}$ and $\mathcal{A}$, and let $c_{\mathcal{F}}$ be the cost function with respect to $\mathcal{F}$, such that the following conditions hold.*

$$\begin{aligned}
\forall \text{del} \in \text{Del} : \forall x \in \mathcal{A} : c_{\text{del}}(x) &\geq 0 \\
\forall \text{ins} \in \text{Ins} : \forall y \in \mathcal{A} : c_{\text{ins}}(y) &\geq 0 \\
\forall \text{del} \in \text{Del} : \exists \text{ins} \in \text{Ins} : \forall x \in \mathcal{A} : c_{\text{del}}(x) &= c_{\text{ins}}(x) \\
\forall \text{ins} \in \text{Ins} : \exists \text{del} \in \text{Del} : \forall y \in \mathcal{A} : c_{\text{ins}}(y) &= c_{\text{del}}(y) \\
\forall \text{rep} \in \text{Rep} : \forall x, y \in \mathcal{A} : c_{\text{rep}}(x, y) &= c_{\text{rep}}(y, x) \geq 0
\end{aligned}$$

Then, the edit distance $d_{\mathcal{S},\mathcal{F}}$ is a pseudo-metric over $\mathcal{A}^*$.

Proof. Refer to Appendix A.9. □

As a final result, we show that any sequence edit distances that can be represented via ADP can be efficiently computed, which is a simplified version of the general ADP results by Giegerich, Meyer, and Steffen (2004).

Theorem 3.3. *Let $\mathcal{S}$ be a signature, let $\mathcal{G}$ be an edit tree grammar over $\mathcal{S}$, let $\mathcal{A}$ be an alphabet, and let $\mathcal{F}$ be an algebra over $\mathcal{S}$ and $\mathcal{A}$. Then, for any two sequences $\bar{x}, \bar{y} \in \mathcal{A}^*$, Algorithm 3.1 computes the edit distance $d_{\mathcal{G},\mathcal{F}}(\bar{x}, \bar{y})$ in $\mathcal{O}(|\bar{x}| \cdot |\bar{y}|)$ time and space complexity.*

Proof. This result is a consequence of the much more general work of Giegerich, Meyer, and Steffen (2004) on ADP. However, we provide a specific version here that is tailored to our application. For the details of the proof, refer to Appendix A.10. □

Consider the example sequences $\bar{x} = \mathtt{aaacac}$ and $\bar{y} = \mathtt{ccbbb}$ from Figure 2.3. The dynamic programming tables resulting from Algorithm 3.1 with the edit tree grammar $\mathcal{G}_{\text{AFFINE}}$ from Equation 2.18, the algebra $\mathcal{F}_{\text{AFFINE}}$ from Equation 2.19, and the input sequences $\bar{x} = \mathtt{aaacac}$ and $\bar{y} = \mathtt{ccbbb}$ are shown in Table 3.1. The resulting edit distance is thus $d_{\mathcal{G},\mathcal{F}}(\bar{x}, \bar{y}) = 5$.

Algorithm 3.1 A general-purpose dynamic programming algorithm computing the edit distance $d_{\mathcal{G},\mathcal{F}}(\bar{x},\bar{y})$ between two sequences $\bar{x}$ and $\bar{y}$ according to the edit tree grammar $\mathcal{G}$, and the algebra $\mathcal{F}$.

function EDIT_DISTANCE(edit tree grammar $\mathcal{G}$, algebra $\mathcal{F}$, sequences $\bar{x},\bar{y}$)
  Let $\mathcal{G} = (\Phi, \mathcal{S}, \mathcal{R}, \mathrm{S})$, and let $\mathcal{S} = (\mathrm{Del}, \mathrm{Rep}, \mathrm{Ins})$.
  Let $\bar{x} = x_1 \ldots x_m$ and $\bar{y} = y_1 \ldots y_n$.
  for $A \in \Phi$ **do**
    Initialize $\boldsymbol{D}^A$ as $(m+1) \times (n+1)$ array of ∞ entries.
    if $A ::= \$ \in \mathcal{R}$ **then**
      $\boldsymbol{D}^A_{m+1,n+1} \leftarrow 0$.
    end if
  end for
  for $i \leftarrow m+1 \ldots 1$ **do**
    for $j \leftarrow n+1 \ldots 1$ **do**
      for $A \in \Phi$ **do**
        $L \leftarrow 0$.
        if $i \leq m$ **then**
          for $A ::= \mathrm{del}B \in \mathcal{R}$ with $\mathrm{del} \in \mathrm{Del}$, $B \in \Phi$ **do**
            $L \leftarrow L+1$.
            $\theta_L \leftarrow \boldsymbol{D}^B_{i+1,j} + c_{\mathrm{del}}(x_i)$.
          end for
        end if
        if $i \leq m$ and $j \leq n$ **then**
          if $x_i = y_j$ **then**
            for $B \in \Phi$ such that $A ::= \mathrm{match}B \in \mathcal{R}$ **do**
              $L \leftarrow L+1$.
              $\theta_L \leftarrow \boldsymbol{D}^B_{i+1,j+1}$.
            end for
          end if
          for $A ::= \mathrm{rep}B \in \mathcal{R}$ with $\mathrm{rep} \in \mathrm{Rep}$, $B \in \Phi$ **do**
            $L \leftarrow L+1$.
            $\theta_L \leftarrow \boldsymbol{D}^B_{i+1,j+1} + c_{\mathrm{rep}}(x_i, y_j)$.
          end for
        end if
        if $j \leq n$ **then**
          for $A ::= \mathrm{ins}B \in \mathcal{R}$ with $\mathrm{ins} \in \mathrm{Ins}$, $B \in \Phi$ **do**
            $L \leftarrow L+1$.
            $\theta_L \leftarrow \boldsymbol{D}^B_{i,j+1} + c_{\mathrm{ins}}(y_j)$.
          end for
        end if
        if $L > 0$ **then**
          $\boldsymbol{D}^A_{i,j} \leftarrow \min\{\theta_1, \ldots, \theta_L\}$.
        end if
      end for
    end for
  end for
  return $\boldsymbol{D}^{\mathrm{S}}_{1,1}$.
end function

Table 3.1: The dynamic programming tables $\boldsymbol{A}$ (left) and $\boldsymbol{S}$ (right) resulting from applying Algorithm 3.1 with the edit tree grammar $\mathcal{G}_{\text{AFFINE}}$ from Equation 2.18, the algebra $\mathcal{F}_{\text{AFFINE}}$ from Equation 2.19, and the input sequences $\bar{x}=$ `aaacac` and $\bar{y}=$ `ccbbb`, as in Figure 2.3.

$\boldsymbol{A}_{i,j}$		1	2	3	4	5	6
		c	c	b	b	b	-
1	a	5.0	5.0	5.0	4.5	4.0	3.5
2	a	4.5	4.5	4.5	4.0	3.5	3.0
3	a	4.0	4.0	4.0	3.5	3.0	2.5
4	c	3.0	3.0	3.0	3.0	2.5	2.0
5	a	3.0	3.0	3.0	2.0	2.0	1.5
6	c	2.5	2.0	2.5	2.0	1.0	1.0
7	-	3.0	2.5	2.0	1.5	1.0	0.0

$\boldsymbol{S}_{i,j}$		1	2	3	4	5	6
		c	c	b	b	b	-
1	a	4.5	4.5	4.5	4.0	3.5	3.0
2	a	4.0	4.0	4.0	3.5	3.0	2.5
3	a	3.5	3.5	3.5	3.0	2.5	2.0
4	c	3.0	3.0	3.0	2.5	2.0	1.5
5	a	3.0	2.5	2.5	2.0	1.5	1.0
6	c	2.5	2.0	2.0	1.5	1.0	0.5
7	-	2.5	2.0	1.5	1.0	0.5	0.0

Now that we have established how to compute sequence edit distances, our next task is to learn them.

Metric Learning via RGLVQ

Our aim is to learn the parameters $\vec{\lambda}$ of some algebra $\mathcal{F}_{\vec{\lambda}}$, such that the sequence edit distance $d_{\mathcal{G},\mathcal{F}_{\vec{\lambda}}}$ for some fixed edit tree grammar $\mathcal{G}$ is optimized for classification. In our case, we assume that a relational generalized learning vector quantization (RGLVQ) model has already been learned for a dataset with M points, and we now wish to adapt the parameters $\vec{\lambda}$ such that the GLVQ cost function E_{GLVQ} from Equation 2.28 for this model is minimized. For the purpose of this minimization, we employ gradient-based optimization. The gradient of E_{GLVQ} with respect to the parameters $\vec{\lambda}$ is given as follows.

$$\nabla_{\vec{\lambda}} E_{\text{GLVQ}} = \sum_{i=1}^{M} \Phi'(\mu_i) \cdot \frac{2}{(d_i^+ + d_i^-)^2} \cdot \left(d_i^- \cdot \nabla_{\vec{\lambda}} d_i^+ - d_i^+ \cdot \nabla_{\vec{\lambda}} d_i^- \right) \tag{3.2}$$

where d_i^+ is the distance between the ith training data point and its closest prototype with the same label, d_i^- is the distance between the ith training data point and its closest prototype with a different label, $\mu_i = (d_i^+ - d_i^-)/(d_i^+ + d_i^-)$, and Φ is some differentiable, monotonously increasing function.

Recall that the prototypes in RGLVQ are given as convex combinations of data points and that we compute the distances d_i^+ and d_i^- in RGLVQ via Equation 2.11. In particular, we obtain the following gradient for the distance between data point x_i and prototype w_k with $\phi(w_k) = \sum_{j=1}^{M} \alpha_{k,j} \cdot \phi(x_j)$.

$$\nabla_{\vec{\lambda}} \|\phi(x_i) - \phi(w_k)\|^2 = \sum_{j=1}^{M} \alpha_{k,j} \cdot \nabla_{\vec{\lambda}} d_{\mathcal{G},\mathcal{F}_{\vec{\lambda}}}(x_i, x_j)^2 - \frac{1}{2} \sum_{j=1}^{M} \sum_{j'=1}^{M} \alpha_{k,j} \cdot \alpha_{k,j'} \cdot \nabla_{\vec{\lambda}} d_{\mathcal{G},\mathcal{F}_{\vec{\lambda}}}(x_j, x_{j'})^2 \tag{3.3}$$

which in turn depends on the gradient of the (squared) edit distances $d_{\mathcal{G},\mathcal{F}_{\vec{\lambda}}}(x_i, x_j)^2$ and $d_{\mathcal{G},\mathcal{F}_{\vec{\lambda}}}(x_j, x_{j'})^2$ with respect to $\vec{\lambda}$. This poses two challenges. First, the above gradient equation only holds if $d_{\mathcal{G},\mathcal{F}_{\vec{\lambda}}}$ is Euclidean, which is generally not the case. Therefore, we would have to apply eigenvalue correction first, which may distort the distances. For now, we heuristically assume that an optimization of the uncorrected edit distances will also yield favorable results for the eigenvalue-corrected version.

Second, the edit distance is non-differentiable because Algorithm 3.1 involves a non-differentiable minimum operation in line 36. To address this issue, we replace the minimum operation with a differentiable approximation, namely the softmin operation, which is defined as follows.

$$\text{softmin}_\beta(\theta_1, \ldots, \theta_L) := \frac{\sum_{l=1}^L \exp(-\beta \cdot \theta_l) \cdot \theta_l}{\sum_{l=1}^L \exp(-\beta \cdot \theta_l)}, \tag{3.4}$$

where $\beta \geq 0$ is a hyper-parameter that we call crispness. We can show that softmin is indeed differentiable, and that it approximates the strict minimum with increasing β.

Theorem 3.4. *Let $\theta_1, \ldots, \theta_L \in \mathbb{R}$. Then, for any $\beta > 0$, softmin_β is differentiable with the following gradient.*

$$\nabla_{\vec{\lambda}} \text{softmin}_\beta(\theta_1, \ldots, \theta_L) = \sum_{l=1}^L \text{softmin}'_{\beta,l}(\theta_1, \ldots, \theta_L) \cdot \nabla_{\vec{\lambda}} \theta_l \qquad \textit{where} \tag{3.5}$$

$$\text{softmin}'_{\beta,l}(\theta_1, \ldots, \theta_L) = \frac{\exp(-\beta \cdot \theta_l)}{\sum_{l'=1}^L \exp(-\beta \cdot \theta_{l'})} \cdot \Big(1 - \beta \cdot \big[\theta_l - \text{softmin}_\beta(\theta_1, \ldots, \theta_L)\big]\Big)$$

Further, there exists a constant $C_L \in \mathbb{R}$, such that for all $\beta > 0$ it holds:

$$0 \leq \text{softmin}_\beta(\theta_1, \ldots, \theta_L) - \min\{\theta_1, \ldots, \theta_L\} \leq \frac{C_L}{\beta}$$

Proof. Refer to Appendix A.11. □

Using the gradient formula 3.5, we can adjust Algorithm 3.1 to compute the gradient of the edit distance with respect to $\vec{\lambda}$ instead of the edit distance itself. This yields Algorithm 3.2.

Theorem 3.5. *Let $\mathcal{S}$ be a signature, let $\mathcal{G}$ be an edit tree grammar over $\mathcal{S}$, let $\mathcal{A}$ be an alphabet, and let $\mathcal{F}$ be an algebra over $\mathcal{S}$ and $\mathcal{A}$. Finally, let $\vec{\lambda}$ be arbitrary parameters of $\mathcal{F}$, and let $\beta \in \mathbb{R}$ with $\beta > 0$.*

Then, for any two sequences $\bar{x}, \bar{y} \in \mathcal{A}$, we define the β-softmin-approximated edit distance $d_{\beta,\mathcal{G},\mathcal{F}}(\bar{x}, \bar{y})$ as the result of Algorithm 3.1 with a softmin operation in line 39 instead of a strict minimum operation.

Further, it holds: Algorithm 3.2 computes the gradient of the β-softmin-approximated edit distance $d_{\beta,\mathcal{G},\mathcal{F}}(\bar{x}, \bar{y})$ with respect to $\vec{\lambda}$ in $\mathcal{O}(|\bar{x}| \cdot |\bar{y}|)$ time and space complexity.

Proof. Refer to Appendix A.12. □

In summary, we can perform sequence edit distance learning using RGLVQ as follows. First, we learn a RGLVQ model on our data set. Then, we perform a gradient-based optimization of the GLVQ cost function from Equation 2.28 with the gradient 3.2. For each gradient step, we need to compute all pairwise edit distances via Algorithm 3.1 and all pairwise gradients via Algorithm 3.2 and plug these into Equations 3.3 and 3.2 to obtain the overall gradient. Complexity-wise, we require $\mathcal{O}(M^2 \cdot m^2)$ steps to compute all pairwise edit distances and gradients, where M is the number of sequences in our data set and m is the maximum length of a sequence in our data set. Further, we obtain a

Algorithm 3.2 A general-purpose dynamic programming algorithm computing the gradient $\nabla_{\vec{\lambda}} d_{\beta,\mathcal{G},\mathcal{F}}(\bar{x}, \bar{y})$ for two sequences $\bar{x}$ and $\bar{y}$ according to the edit tree grammar $\mathcal{G}$, and the algebra $\mathcal{F}$.

```
function EDIT_DISTANCE_GRADIENT(edit tree grammar $\mathcal{G}$, algebra $\mathcal{F}$, sequences $\bar{x}, \bar{y}$,
  crispness $\beta$)
    Let $\mathcal{G} = (\Phi, \mathcal{S}, \mathcal{R}, \mathrm{S})$, and let $\mathcal{S} = (\mathrm{Del}, \mathrm{Rep}, \mathrm{Ins})$.
    Let $\bar{x} = x_1 \dots x_m$ and $\bar{y} = y_1 \dots y_n$.
    for $A \in \Phi$ do
        Initialize $\boldsymbol{D}^A$ as $(m+1) \times (n+1)$ array of $\infty$ entries.
        Initialize $\boldsymbol{G}^A$ as $(m+1) \times (n+1)$ array of $\vec{0}$ vectors.
        if $A ::= \$ \in \mathcal{R}$ then
            $\boldsymbol{D}^A_{m+1,n+1} \leftarrow 0$.
        end if
    end for
    for $i \leftarrow m+1 \dots 1$ do
        for $j \leftarrow n+1 \dots 1$ do
            for $A \in \Phi$ do
                $L \leftarrow 0$.
                if $i \leq m$ then
                    for $A ::= \mathrm{del}B \in \mathcal{R}$ with $\mathrm{del} \in \mathrm{Del}$, $B \in \Phi$ do
                        $L \leftarrow L + 1$.
                        $\theta_L \leftarrow \boldsymbol{D}^B_{i+1,j} + c_{\mathrm{del}}(x_i)$.
                        $\nabla_{\vec{\lambda}} \theta_L \leftarrow \boldsymbol{G}^B_{i+1,j} + \nabla_{\vec{\lambda}} c_{\mathrm{del}}(x_i)$.
                    end for
                end if
                if $i \leq m$ and $j \leq n$ then
                    if $x_i = y_j$ then
                        for $B \in \Phi$ with $A ::= \mathrm{match}B \in \mathcal{R}$ do
                            $L \leftarrow L + 1$.
                            $\theta_L \leftarrow \boldsymbol{D}^B_{i+1,j+1}$.
                            $\nabla_{\vec{\lambda}} \theta_L \leftarrow \boldsymbol{G}^B_{i+1,j+1}$.
                        end for
                    end if
                    for $A ::= \mathrm{rep}B \in \mathcal{R}$ with $\mathrm{rep} \in \mathrm{Rep}$, $B \in \Phi$ do
                        $L \leftarrow L + 1$.
                        $\theta_L \leftarrow \boldsymbol{D}^B_{i+1,j+1} + c_{\mathrm{rep}}(x_i, y_j)$.
                        $\nabla_{\vec{\lambda}} \theta_L \leftarrow \boldsymbol{G}^B_{i+1,j+1} + \nabla_{\vec{\lambda}} c_{\mathrm{rep}}(x_i, y_j)$.
                    end for
                end if
                if $j \leq n$ then
                    for $A ::= \mathrm{ins}B \in \mathcal{R}$ with $\mathrm{ins} \in \mathrm{Ins}$, $B \in \Phi$ do
                        $L \leftarrow L + 1$.
                        $\theta_L \leftarrow \boldsymbol{D}^B_{i,j+1} + c_{\mathrm{ins}}(y_j)$.
                        $\nabla_{\vec{\lambda}} \theta_L \leftarrow \boldsymbol{G}^B_{i,j+1} + \nabla_{\vec{\lambda}} c_{\mathrm{ins}}(y_j)$.
                    end for
                end if
                if $L > 0$ then
                    $\boldsymbol{D}^A_{i,j} \leftarrow \mathrm{softmin}(\theta_1, \dots, \theta_L)$.
                    $\boldsymbol{G}^A_{i,j} \leftarrow \sum_{l=1}^{L} \mathrm{softmin}'_{\beta,l}(\theta_1, \dots, \theta_L) \cdot \nabla_{\vec{\lambda}} \theta_l$.
                end if
            end for
        end for
    end for
    return $\boldsymbol{G}^{\mathrm{S}}_{1,1}$.
end function
```

space complexity of $\mathcal{O}(M^2 + m^2)$ to store the pairwise distance matrix and the dynamic programming matrices. Note that these computations can be made in parallel, relieving some of the computational burden.

This concludes our description of the proposed metric learning scheme. In the next section, we evaluate our scheme experimentally.

3.2 EXPERIMENTS

Artificial Datasets

We evaluate our metric learning scheme on two artificial datasets, namely:

Strings: An artificial, balanced two-class dataset of 200 strings of length 12. Strings in class 1 consist of 6 `a` or `b` symbols, followed by a `c` or `d`, followed by another 5 `a` or `b` symbols. Which of the two respective symbols is selected is chosen uniformly at random. Strings in class 2 are constructed in much the same way, except that they consist of 5 `a` or `b` symbols, followed by a `c` or `d`, followed by another 6 `a` or `b` symbols. Note that the classes can be neither discriminated via length nor via symbol frequency features. The decisive discriminative feature is where `c`s and `d`s are located in the string.

Gap: An artificial, balanced two-class dataset of 200 uniform random strings over the alphabet $\mathcal{A} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$, where the strings in class 1 have length 10 and the strings in class 2 have length 12. In this data set, the discriminative feature is the length, with replacement costs being irrelevant.

For these data, our aim is to optimize the standard string edit distance with signature $\mathcal{S}_{\text{ALI}} = (\{\text{del}\}, \{\text{rep}\}, \{\text{ins}\})$ and edit tree grammar $\mathcal{G}_{\text{ALI}}$ as defined in Equation 2.16. For each alphabet $\mathcal{A} = \{x_1, \ldots, x_m\}$ we employ the following algebra $\mathcal{F}_{\vec{\lambda}}$.

$$
\begin{aligned}
c_{\text{rep}}(x_i, x_j) &= \lambda_{(m+1)\cdot(i-1)+j} \\
c_{\text{del}}(x_i) &= \lambda_{(m+1)\cdot i} \\
c_{\text{ins}}(x_j) &= \lambda_{(m+1)\cdot m+j}
\end{aligned}
$$

In other words, we consider the replacement, deletion, and insertion costs as parameters, which results in $(|\mathcal{A}| + 1)^2 - 1$ parameters overall. As initialization, we use the algebra $\mathcal{F}_{\text{ALI}}$ specified in Equation 2.15.

For metric learning, we train a RGLVQ model with one prototype per class and then perform ten gradient descent steps to learn the parameters. As learning rate for gradient descent we employ $\eta = 1/M$ for both datasets, where M is the number of data points. After each gradient step, we normalize the parameters by clipping negative values to zero, by setting self-replacement costs to zero, by symmetrizing the parameters, and by using the Floyd-Warshal algorithm for pairwise shortest paths to enforce the triangular inequality (Floyd 1962). We set the crispness parameter to $\beta = 1$.

We evaluate the average classification error on our learned metric in a crossvalidation with 20 folds using four different classifiers, namely a 1-nearest neighbor classifier (1-NN), a RGLVQ classifier with one prototype per class, a support vector machine (SVM) with

Table 3.2: The mean classification error ± standard deviation of multiple classifiers across 20 crossvalidation trials on both artificial datasets. The first column lists the results for the standard string edit distance, the second column for GESL, and the final column for our proposed metric learning scheme. Datasets and the different classifiers are listed as rows. The best results for each dataset are highlighted in bold print.

classifier	Initial	GESL	proposed
		String	
1-NN	$20.0 \pm 9.2\%$	$10.5 \pm 9.4\%$	$\mathbf{0.0 \pm 0.0}\%$
RGLVQ	$39.5 \pm 14.3\%$	$18.0 \pm 12.8\%$	$\mathbf{0.0 \pm 0.0}\%$
SVM	$7.0 \pm 8.6\%$	$11.0 \pm 8.5\%$	$\mathbf{0.0 \pm 0.0}\%$
good	$4.5 \pm 5.1\%$	$4.5 \pm 5.1\%$	$\mathbf{0.0 \pm 0.0}\%$
		Gap	
1-NN	$51.5 \pm 15.7\%$	$10.0 \pm 11.2\%$	$5.0 \pm 15.4\%$
RGLVQ	$54.5 \pm 10.5\%$	$34.5 \pm 11.9\%$	$7.0 \pm 17.2\%$
SVM	$20.0 \pm 10.3\%$	$45.0 \pm 6.9\%$	$5.0 \pm 15.4\%$
good	$6.5 \pm 6.7\%$	$\mathbf{1.0 \pm 3.1}\%$	$5.0 \pm 15.4\%$

radial basis function kernel (refer to Equation 2.44) and clip eigenvalue correction, and the goodness classifier of Balcan, Blum, and Srebro (2008) with the similarity metric $s_d(\bar{x}, \bar{y}) = 2 \cdot \exp(-d_c(\bar{x}, \bar{y})) - 1$ as suggested in Section 2.4.1. We optimized the radial basis function bandwidth ξ for SVM and the sparsity hyperparameter ν for the goodness classifier in a nested crossvalidation with 5 folds. We compare these classification errors with the errors obtained via the initial string edit distance and the pseudo-edit distance obtained via good edit similarity learning (GESL, Bellet, Habrard, and Sebban 2012, also refer to Section 2.4.1) with $K = 1$ reference point from the same and from the other class for each point.

The results are displayed in Table 3.2. For the strings dataset, our proposed metric learning scheme could improve the string edit distance such that all classifiers could classify the data perfectly in all folds, whereas GESL only yielded improvements for the 1-NN and RGLVQ classifier. Overall, our proposed sequence edit distance learning scheme significantly outperformed both the initial string edit distance and GESL for all classifiers ($p < 0.01$ according to a Wilcoxon signed-rank test). Deeper inspection revealed that our proposed scheme did indeed reduce the pairwise replacement costs $c(\mathtt{a}, \mathtt{b}) = c(\mathtt{b}, \mathtt{a})$ to zero in all folds as expected.

For the gaps dataset, our proposed scheme could improve the metric such that perfect classification was possible in 17 out of 20 folds. In these cases, our approach did correctly set the pairwise replacement costs $c(\mathtt{a}, \mathtt{b}) = c(\mathtt{b}, \mathtt{a})$ to zero while gap costs remained nonzero. This result was surpassed by GESL for the goodness classifier, where GESL achieved 1% error. However, GESL achieved much worse results for the other classifiers, indicating that the learned metric is rather specific to the goodness classifier, whereas our proposed approach achieves a metric that is viable across the board. For all classifiers except the goodness classifier, our proposed approach achieved significantly better results than the initial string edit distance ($p < 0.01$); and for both RGLVQ and SVM we significantly outperformed GESL ($p < 0.001$).

```java
public class Adder {
  public int add(int a, int b) {
    return a+b;
  }
}
```

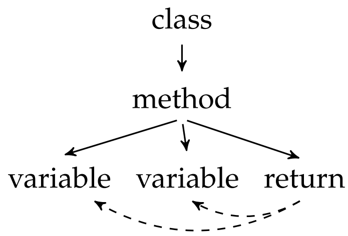


Figure 3.1: Left: A snippet of example Java source code. Right: The corresponding abstract syntax tree (AST). Note that the AST also includes backreferences from the "return" node to both variable nodes, because both variables are referenced in the return statement.

Real-World Data

We consider the following real-world datasets.

Copenhagen Chromosomes: A balanced two-class dataset of 400 strings, consisting of the classes 4 and 5 of the Copenhagen Chromosomes database (Lundsieen, Philip, and Granum 1980). Each string describes the density of a human chromosome in differential coding with a 13-letter alphabet $\mathcal{A} = \{\texttt{f}, \ldots, \texttt{a}, \texttt{=}, \texttt{A}, \ldots, \texttt{F}\}$, where lower case letters mark negative changes in density, upper case letters mark positive changes in density, and = codes equal density.

Sorting: An unbalanced, two-class dataset consisting of 64 Java programs collected from 37 different web sites (Paaßen 2016a). All programs are implementations of sorting algorithms for an input array of integers. In particular, 35 programs implement *BubbleSort*, and 29 programs implement *InsertionSort*. We preprocess all programs by extracting their abstract syntax trees (ASTs) using the Oracle Java™ Compiler API. To each node of these ASTs, we attach a feature vector incorporating characteristic properties, namely a discrete type label (e.g. class declaration, method, variable declaration, for loop, etc.), an encoding of the visibility scope the node is located in, the index of the parent node, the row and column index of the node within the original source code, the name of a declared class, method, or variable if applicable, the name of the class of the declared variable if applicable, the name of the class of a returned variable if applicable, the number of references to other nodes within the AST, and a list of strings of references to external classes, methods, or variables. Finally, we flatten all ASTs to sequences by considering the sequence of nodes in depth-first search oder. As an example, consider the source code listed in Figure 3.1. The corresponding depth-first-search sequence is shown in Table 3.3.

For the CopenhagenChromosomes dataset, we again evaluate the standard string edit distance and learn the pairwise replacement and gap costs directly. For the Sorting dataset, we learn both the standard string edit distance as well as the affine edit distance of Gotoh (1982) with the signature $\mathcal{S}_{\text{LOCAL}}$ as defined in Equation 2.17, the edit tree

Table 3.3: The depth-first search sequence generated for the "Adder" source code listed in Figure 3.1. Each column corresponds to one AST node, each row to one feature.

type	class	method	variable	variable	return
scope	[]	[0]	[0,0]	[0,0]	[0,0]
parent	-1	0	1	1	1
codePosition	(1,1)-(5,2)	(2,3)-(4,4)	(2,18)-(2,23)	(2,25)-(2,30)	(3,5)-(3,16)
name	Adder	add	a	b	–
className	–	–	int	int	–
returnType	–	int	–	–	–
numberOfEdges	1	3	0	0	2
externalDeps	–	int	–	–	–

grammar $\mathcal{G}_{\text{AFFINE}}$ as defined in Equation 2.18, and the following algebra $\mathcal{F}_{\vec{\lambda}}$.

$$\begin{aligned}
c_{\text{del}}(x) = c_{\text{rep}}(x) = c^{l,o}_{\text{skip}}(x) = c^{r,o}_{\text{skip}}(x) &= 1 && \forall x \in \mathcal{A} \\
c^{l}_{\text{skip}}(x) = c^{r}_{\text{skip}}(x) &= 0.5 && \forall x \in \mathcal{A} \\
c_{\text{rep}}(x,y) &= \sum_{r=1}^{9} \lambda_r \cdot c_r(x_r, y_r) && \forall x, y \in \mathcal{A}
\end{aligned}$$

where x_r denotes the rth feature of x, λ_r is a real number in the range $[0,1]$ such that $\sum_{r=1}^{9} \lambda_r = 1$, and c_r is a specific metric for the rth feature. In particular, for the type feature, we assign a distance of 1 if the type is not equal and a distance of 0 otherwise. For the scope feature, we use one minus the length of the longest common prefix divided by the longer scope. For the parent feature, the code position, and the number of edges, we use the Manhattan distance. For the name, the className, the returnType, and the externalDeps feature we compute character frequencies and use the Manhattan distance on the character frequency vectors. Our adaptable metric parameters are the weights λ_r. We initialize these weights as $\lambda_r = 1/9$.

As with the artificial data, we first train a RGLVQ model with one prototype per class and then perform ten gradient descent steps to learn the respective parameters. As learning rate for gradient descent, we employ $\eta = 0.45/M$ for the CopenhagenChromosomes dataset and $\eta = 2/(M \cdot |\bar{x}|)$ for the Sorting dataset, where M is the number of data points and $|\bar{x}|$ is the average sequence size in the dataset. After each gradient step, we normalize the parameters, using the same normalization as for the artificial data in case of the CopenhagenChromosomes dataset, and by clipping negative values to 0 and normalizing the sum to 1 for the Sorting dataset. We set the crispness parameter β to 7 for the CopenhagenChromosomes dataset, and to 200 for the Sorting dataset.

We evaluate the average classification error across 5 crossvalidation folds of three different classifiers, namely 5-nearest neighbor, RGLVQ, and a SVM with a kernel obtained via double-centering (refer to Equation 2.6). Note that we do not compare to GESL at this point because our parametrization for the Sorting dataset is not compatible with GESL. We repeat the crossvalidation 10 times for CopenhagenChromosomes and 5 times for Sorting.

The results are displayed in Table 3.4. For both datasets, metric learning improves the classification accuracy across classifiers. In particular, we improve the RGLVQ accuracy by

Table 3.4: The mean classification error of multiple classifiers across crossvalidation trials and repeats on the CopenhagenChromosomes and Sorting datasets. Columns correspond to classifiers, while rows correspond to different metrics on different datasets. The best results for each dataset are highlighted in bold print.

classifier	RGLVQ	SVM	5-NN
	CopenhagenChromosomes		
initial	11%	4%	**3**%
learned	5%	**3**%	**3**%
	Sorting		
global initial	26%	35%	23%
global learned	20%	37%	8%
affine initial	26%	26%	38%
affine learned	15%	22%	**0**%

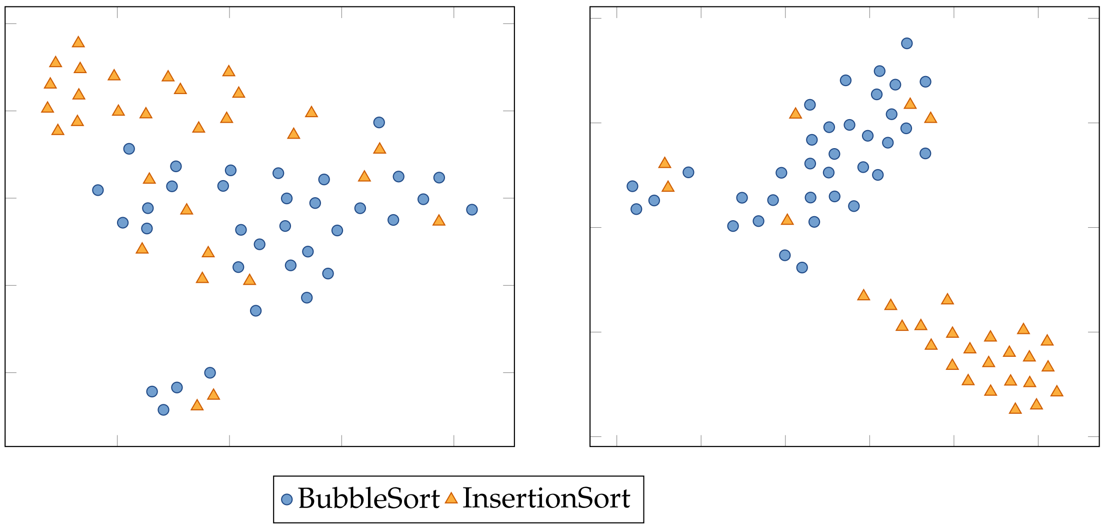


Figure 3.2: Two-dimensional t-SNE embeddings of the Sorting dataset without metric learning (left) and with metric learning (right) for the affine edit distance. *BubbleSort* programs are visualized as blue circles, *InsertionSort* programs as orange triangles.

about 6% for the standard string edit distance and by about 11% for the affine edit distance. For SVM and 5-NN we also observe improvements, except for SVM for the standard sequence edit distance on the Sorting dataset. For 5-NN, we observe particularly striking improvements on the Sorting dataset with 15% for the standard sequence edit distance and 38% for the affine edit distance. We can also inspect the change in representation visually. Figure 3.2 displays two-dimensional t-SNE embeddings (Van der Maaten and Hinton 2008) of the Sorting dataset for the default affine edit distance and the learned affine edit distance. As visible in the figure, classes get more compact and more distinct.

The resulting weights $\vec{\lambda}$ after metric learning on the Sorting dataset (normalized by their frequency in the data) are shown in Figure 3.3. As we can see, the weights for the numberofEdges feature, the codePosition feature, and the parent feature has been

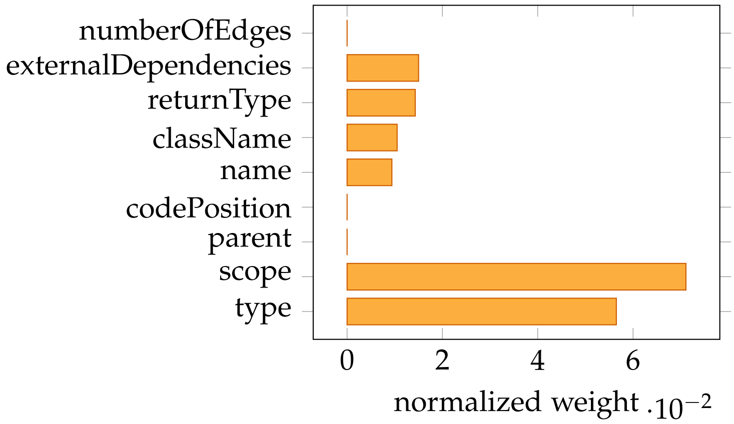


Figure 3.3: The weights $\vec{\lambda}$ after metric learning on the affine edit distance. The weights were normalized by their frequency in the dataset.

reduced to zero, whereas the scope and the type feature are strongly emphasized. This makes intuitive sense as the type feature is invariant against many stylistic choices and captures the local function of the current syntactic element, whereas the scope feature indicates the rough position in the tree of the current syntactic element. Less frequent elements such as name, className, and returnType, can fulfill auxiliary function and disambiguate special types of nodes, namely class declarations, method declarations, and variable declarations.

3.3 CONCLUSION AND LIMITATIONS

In this section, we have introduced a generic metric learning scheme for arbitrary parameters of a broad class of edit distances, namely those which can be characterized by a signature, an algebra, and an edit tree grammar. We have shown that these edit distances as well as their approximated gradient can be efficiently computed via algebraic dynamic programming (Giegerich, Meyer, and Steffen 2004). We have then utilized the resulting gradient to optimize the GLVQ cost function with respect to a RGLVQ model, that is, we have adjusted the parameters of the algebra, such that the resulting edit distance pulls data points closer to the closest prototype with the same label and pushes them away from the closest prototype with a different label. We have shown experimentally that our scheme yields edit distances that do not only improve the classification accuracy of the RGLVQ model it was trained on, but also enhances the accuracy of nearest neighbor classifiers and SVMs. On artificial data, we have also shown that our scheme outperforms sequence edit distance learning via GESL for the same number of reference data. On both artificial and real-world data, we found that the learned parameters corresponded well to the underlying domain semantics. Finally, we observed that improvements on a computer programming dataset are even more pronounced for the affine edit distance of Gotoh (1982) versus the standard edit distance of Levenshtein (1965).

While these results are promising, there are numerous limitations to our current approach. First, we have employed a classifier that relies on a eigenvalue-corrected distance matrix, which complicates the practical application. Second, the eigenvalue correction means that our optimization of the uncorrected edit distances does not necessarily imply an improvement on the corrected distance matrix, as the behavior of uncorrected and corrected distances may differ significantly (refer e.g. to Nebel, Kaden, et al. 2017). Third, our proposed method is relatively slow because each gradient step requires us to compute the gradients of all pairwise distances in the data set, which is only feasible for small datasets and a small number of gradient steps. Fourth, there is no guarantee regarding the approximation quality of the softmin-approximated edit distance. While we have shown that a single softmin application approximates the actual minimum, approximation errors may accumulate over the computation, leading to higher errors for longer input sequences. Fifth, we are currently limited to sequence edit distances, which can not directly process tree or graph data. In the next section, we will address all of these limitations with an extended metric learning scheme that works on trees.

4 TREE EDIT DISTANCE LEARNING

Summary: Trees are versatile data structures, which can be used to model syntax of natural and formal languages as well as biological data such as RNA secondary structures or glycan molecules. In all these cases, the tree edit distance offers an interpretable and actionable metric, which is useful for various downstream applications. However, the tree edit distance may be misleading if its parameters do not fit the task at hand. In this chapter, we present embedding edit distance learning (BEDL), an effective and scalable tree edit distance learning approach. In our evaluation on datasets from natural language processing, biology, and intelligent tutoring systems we demonstrate that our method can not only improve the default tree edit distance, but can also outperform the state-of-the-art.

Publications: This chapter is based on the following publications.

- Paaßen, Benjamin (2018). *Revisiting the tree edit distance and its backtracing: A tutorial*. arXiv: `1805.06869 [cs.DS]`.
- Paaßen, Benjamin, Claudio Gallicchio, et al. (2018). "Tree Edit Distance Learning via Adaptive Symbol Embeddings". In: *Proceedings of the 35th International Conference on Machine Learning (ICML 2018)*. (Stockholm, Sweden). Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research, pp. 3973–3982. URL: `http://proceedings.mlr.press/v80/paassen18a.html`.

Source Code: The Java(R) source code for the tree edit distance and cooptimal frequency matrix computation is available at `https://openresearch.cit-ec.de/projects/tcs`.

The Java(R) source code for median generalized vector quantization is available at `https://gitlab.ub.uni-bielefeld.de/bpaassen/median_relational_glvq`

The MATLAB(R) source code for tree edit distance learning is available at `http://doi.org/10.4119/unibi/2919994`.

Trees occur in many shapes across various application domains, such as syntax trees of natural language (Socher, Perelygin, et al. 2013), syntax trees of programming languages (Rivers and Koedinger 2015), or descriptors of RNA secondary structures and glycan molecules in biology (Akutsu 2010). In all these areas, the tree edit distance (Zhang and Shasha 1989) is a useful measure of distance, as it can support information retrieval and other downstream tasks (Akutsu 2010). For example, the tree edit distance has achieved increasing popularity in recent years in the field of intelligent tutoring systems for computer programming (Mokbel, Gross, et al. 2013; Gross, Mokbel, et al. 2014; Price, Dong, and Lipovac 2017; Rivers and Koedinger 2015). In such systems, the tree edit distance can pinpoint exactly which nodes in an abstract syntax tree of a student's current program have to be changed in order to arrive at a correct solution, and we can use these edits to guide a student through a programming task (Mokbel, Gross, et al. 2013; Gross, Mokbel, et al. 2014; Price, Dong, and Lipovac 2017; Rivers and Koedinger 2015).

As with sequence edit distances, the tree edit distance is only useful if its cost function fits the task at hand. Per default, the tree edit distance regards all possible tree nodes as equidistant, which may be misleading for all the domains above. In particular, natural language words may have overlapping semantics (Pennington, Socher, and Manning 2014; Socher, Perelygin, et al. 2013), glycan molecule descriptors may be referring to biologically similar elements (Gallicchio and Micheli 2013), and syntactic nodes in computer programs may fulfill the same or similar functions (Paaßen, Jensen, and Hammer 2016), in which case the pairwise replacement costs should be lowered. This begs the question how the tree edit distance can be *adapted* to be better suited for the domain and task at hand.

In this chapter, we propose a novel method to learn the tree edit distance that goes beyond the state of the art of good edit similarity learning (GESL, Bellet, Habrard, and Sebban 2012, also refer to Section 2.4.1) in several respects.

- We consider all cooptimal pairwise tree mappings instead of just one tree mapping by means of a novel forward-backward algorithm.
- We select reference pairs for metric learning in a principled fashion based on learning vector quantization prototypes for each class instead of ad-hoc selection schemes.
- Most importantly, we learn an embedding of the syntactic elements of trees instead of direct cost parameters, thus guaranteeing metric properties and higher efficiency for large alphabets.

We call our resulting metric learning approach em**b**edding **e**dit **d**istance **l**earning (BEDL).

In our evaluation, we show that BEDL outperforms GESL in terms of classification accuracy across several classifiers and several real-world datasets from natural language processing, biology, and intelligent tutoring systems. We also demonstrate that BEDL can be scaled up to a large natural language processing dataset. In the next section, we describe BEDL in detail, before we continue to our experimental evaluation.

4.1 METHOD

Our aim is to adapt the cost function of the tree edit distance of Zhang and Shasha (1989) to improve the accuracy of a classifier based on this edit distance. More specifically, we intend to improve the accuracy of a median generalized learning vector quantization (MGLVQ) model. For our purposes, MGLVQ has two key advantages compared to RGLVQ, which we used in the previous chapter. First, MGLVQ does not require an Euclidean distance as input such that we can avoid eigenvalue correction. Second, MGLVQ represents prototypes sparsely by setting each prototype to a single datapoint. As such, we only need to consider linearly many distance values to optimize the GLVQ cost function in Equation 2.28, namely the data-to-prototype distances.

More precisely, assume that we wish to optimize the parameters $\vec{\lambda}$ of the cost function $c_{\vec{\lambda}}$ via gradient-based techniques on the GLVQ cost function E_{GLVQ} with the gradient 3.2. To compute this gradient, we require the tree edit distance gradients $\nabla_{\vec{\lambda}} d_i^+$ and $\nabla_{\vec{\lambda}} d_i^-$. Computing these gradients is our next step. To do so, we decompose the tree edit distance into a scalar product of tree mapping matrices and pairwise costs.

Co-Optimal Frequency Matrices

Our decomposition is similar to the GESL approach of Bellet, Habrard, and Sebban (2012, also refer to Section 2.4.1). However, in contrast to their approach, we consider not only a single tree mapping matrix, but an average of all cooptimal tree mapping matrices. Recall that we denote the tree edit distance with respect to a cost function c as d_c (refer to Definition 2.12), the tree mapping edit distance as D_c (refer to Definition 2.14), and the tree mapping matrix with respect to a tree mapping M between two trees $\tilde{x}$ and $\tilde{y}$ as $\boldsymbol{P}(M, \tilde{x}, \tilde{y})$ (refer to Definition 2.16). We define the cooptimal frequency matrix as follows.

Definition 4.1 (Co-optimal Frequency Matrix)**.** Let $\tilde{x}$ and $\tilde{y}$ be trees over some alphabet $\mathcal{A}$, let M be a tree mapping between $\tilde{x}$ and $\tilde{y}$, and let c be a cost function over $\mathcal{A}$.

Further, let $\mathcal{M}(c, \tilde{x}, \tilde{y})$ be the set of all cooptimal tree mappings between $\tilde{x}$ and $\tilde{y}$, i.e. all tree mappings M such that $c(M, \tilde{x}, \tilde{y}) = D_c(\tilde{x}, \tilde{y})$.

We define the *cooptimal frequency matrix* $\boldsymbol{P}_c(\tilde{x}, \tilde{y})$ as the $|\tilde{x}| \times |\tilde{y}|$ matrix

$$\boldsymbol{P}_c(\tilde{x}, \tilde{y}) := \frac{\sum_{M \in \mathcal{M}(c, \tilde{x}, \tilde{y})} \boldsymbol{P}(M, \tilde{x}, \tilde{y})}{|\mathcal{M}(c, \tilde{x}, \tilde{y})|} \tag{4.1}$$

In other words, the cooptimal frequency matrix $\boldsymbol{P}_c(\tilde{x}, \tilde{y})$ is defined as the average tree mapping matrix $\boldsymbol{P}(M, \tilde{x}, \tilde{y})$ for all cooptimal tree mappings M. As an example, consider the trees $\tilde{x} = \texttt{a(b(c,d),e)}$ and $\tilde{y} = \texttt{f(g)}$. All cooptimal tree mappings between $\tilde{x}$ and $\tilde{y}$ according to default costs, along with the respective tree mapping matrix and the resulting cooptimal frequency matrix are listed in Figure 4.1.

Using the concepts of tree mapping matrices and cooptimal frequency matrices, we obtain the following decomposition for the tree edit distance.

Theorem 4.1. *Let $\tilde{x}$ and $\tilde{y}$ be trees over some alphabet $\mathcal{A}$ and let c be a cost function over $\mathcal{A}$.*

Then, if c is non-negative, self-equal, and conforms to the triangular inequality, it holds:

$$\begin{aligned} d_c(\tilde{x}, \tilde{y}) = & \sum_{i=1}^{|\tilde{x}|} \sum_{j=1}^{|\tilde{y}|} \boldsymbol{P}_c(\tilde{x}, \tilde{y})_{i,j} \cdot c(x_i, y_j) \\ & + \sum_{i=1}^{|\tilde{x}|} \Big(1 - \sum_{j=1}^{|\tilde{y}|} \boldsymbol{P}_c(\tilde{x}, \tilde{y})_{i,j}\Big) \cdot c(x_i, -) + \sum_{j=1}^{|\tilde{y}|} \Big(1 - \sum_{i=1}^{|\tilde{x}|} \boldsymbol{P}_c(\tilde{x}, \tilde{y})_{i,j}\Big) \cdot c(-, y_j) \end{aligned} \tag{4.2}$$

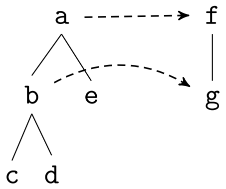
Proof. First, Theorem 2.5 implies that $d_c(\tilde{x}, \tilde{y}) = D_c(\tilde{x}, \tilde{y})$.

Further, note that for any $M \in \mathcal{M}(c, \tilde{x}, \tilde{y})$ it holds: $c(M, \tilde{x}, \tilde{y}) = D_c(\tilde{x}, \tilde{y})$. Accordingly, we obtain:

$$d_c(\tilde{x}, \tilde{y}) = \frac{\sum_{M \in \mathcal{M}(c, \tilde{x}, \tilde{y})} c(M, \tilde{x}, \tilde{y})}{|\mathcal{M}(c, \tilde{x}, \tilde{y})|}$$

By virtue of Equation 2.26 we obtain:

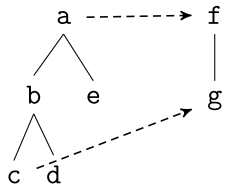
$$\begin{aligned} d_c(\tilde{x}, \tilde{y}) = & \frac{1}{|\mathcal{M}(c, \tilde{x}, \tilde{y})|} \cdot \sum_{M \in \mathcal{M}(c, \tilde{x}, \tilde{y})} \sum_{i=1}^{|\tilde{x}|} \sum_{j=1}^{|\tilde{y}|} \boldsymbol{P}(M, \tilde{x}, \tilde{y})_{i,j} \cdot c(x_i, y_j) \\ & + \sum_{i=1}^{|\tilde{x}|} \Big(1 - \sum_{j=1}^{|\tilde{y}|} \boldsymbol{P}(M, \tilde{x}, \tilde{y})_{i,j}\Big) \cdot c(x_i, -) + \sum_{j=1}^{|\tilde{y}|} \Big(1 - \sum_{i=1}^{|\tilde{x}|} \boldsymbol{P}(M, \tilde{x}, \tilde{y})_{i,j}\Big) \cdot c(-, y_j) \end{aligned}$$



$M = \{(1,1),(2,2)\}$

$\boldsymbol{P}(M, \tilde{x}, \tilde{y})$

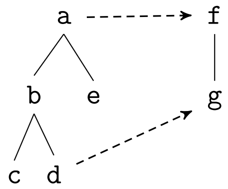
$i \setminus j$	1	2
1	1	0
2	0	1
3	0	0
4	0	0
5	0	0



$M = \{(1,1),(3,2)\}$

$\boldsymbol{P}(M, \tilde{x}, \tilde{y})$

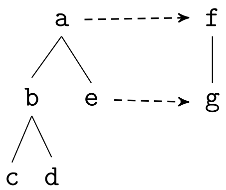
$i \setminus j$	1	2
1	1	0
2	0	0
3	0	1
4	0	0
5	0	0



$M = \{(1,1),(4,2)\}$

$\boldsymbol{P}(M, \tilde{x}, \tilde{y})$

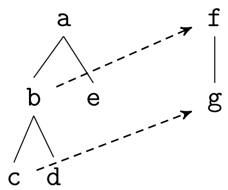
$i \setminus j$	1	2
1	1	0
2	0	0
3	0	0
4	0	1
5	0	0



$M = \{(1,1),(5,2)\}$

$\boldsymbol{P}(M, \tilde{x}, \tilde{y})$

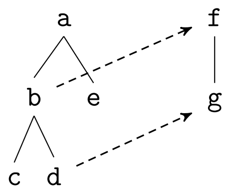
$i \setminus j$	1	2
1	1	0
2	0	0
3	0	0
4	0	0
5	0	1



$M = \{(2,1),(3,2)\}$

$\boldsymbol{P}(M, \tilde{x}, \tilde{y})$

$i \setminus j$	1	2
1	0	0
2	1	0
3	0	1
4	0	0
5	0	0



$M = \{(2,1),(4,2)\}$

$\boldsymbol{P}(M, \tilde{x}, \tilde{y})$

$i \setminus j$	1	2
1	0	0
2	1	0
3	0	0
4	0	1
5	0	0

$\boldsymbol{P}_c(\tilde{x}, \tilde{y})$

$i \setminus j$	1	2
1	2/3	0
2	1/3	1/6
3	0	1/3
4	0	1/3
5	0	1/6

Figure 4.1: All cooptimal tree mappings between the trees $\tilde{x} = \texttt{a}(\texttt{b}(\texttt{c}, \texttt{d}), \texttt{e})$ and $\tilde{y} = \texttt{f}(\texttt{g})$ according to default costs (top and middle), the corresponding tree mapping matrices $\boldsymbol{P}(M, \tilde{x}, \tilde{y})$ (below the tree mapping diagrams), and the resulting cooptimal frequency matrix $\boldsymbol{P}_c(\tilde{x}, \tilde{y})$ (bottom).

which can be re-written into Equation 4.2, which completes the proof. □

Note that it is not trivial to compute the cooptimal frequency matrix because the set $\mathcal{M}(c, \tilde{x}, \tilde{y})$ may have exponential size. To address this issue, we introduce a novel forward-backward algorithm.

Theorem 4.2. *Let $\tilde{x}$ and $\tilde{y}$ be trees over some alphabet $\mathcal{A}$ and let c be a cost function over $\mathcal{A}$ that conforms to the triangular inequality. Then, Algorithm 4.1 computes $\boldsymbol{P}_c(\tilde{x}, \tilde{y}) \cdot |\mathcal{M}(c, \tilde{x}, \tilde{y})|$ as first output and $|\mathcal{M}(c, \tilde{x}, \tilde{y})|$ as second output. Further, Algorithm 4.1 runs in $\mathcal{O}(|\tilde{x}|^6 \cdot |\tilde{y}|^6)$ time complexity and $\mathcal{O}(|\tilde{x}|^2 \cdot |\tilde{y}|^2)$ space complexity in the worst case.*

Proof. Refer to Appendix A.13. □

Algorithm 4.1 An algorithm that computes the matrix $\boldsymbol{P}_c(\tilde{x}, \tilde{y})$ for two given trees $\tilde{x}$ and $\tilde{y}$ and a cost function c. More specifically, the first output argument is $\boldsymbol{P}_c(\tilde{x}, \tilde{y}) \cdot |\mathcal{M}(c, \tilde{x}, \tilde{y})|$, and the second output argument is $|\mathcal{M}(c, \tilde{x}, \tilde{y})|$. For the forward and backward algorithm, refer to Appendix A.13.

function COOPTIMALS(Two trees $\tilde{x}$ and $\tilde{y}$, the matrices $\boldsymbol{d}$ and $\boldsymbol{D}$ after executing algorithm 2.1, and a cost function c)
    $(C, \boldsymbol{A}) \leftarrow \text{FORWARD}(\tilde{x}, \tilde{y}, \boldsymbol{d}, \boldsymbol{D}, c)$. ▷ Refer to Algorithm A.1.
    $\boldsymbol{B} \leftarrow \text{BACKWARD}(\tilde{x}, \tilde{y}, \boldsymbol{d}, \boldsymbol{D}, c, C)$. ▷ Refer to Algorithm A.2.
    Initialize $\boldsymbol{\Gamma}$ as a $|\tilde{x}| \times |\tilde{y}|$ matrix of zeros.
    for $(i, j) \in C$ **do**
        if $i = |\tilde{x}| + 1 \vee j = |\tilde{y}| + 1$ **then**
            continue
        end if
        if $(rl_{\tilde{x}}(i) = |\tilde{x}| \wedge rl_{\tilde{y}}(j) = |\tilde{y}|) \vee c(x_i, y_j) = c(x_i, -) + c(-, y_j)$ **then**
            if $\boldsymbol{D}_{i,j} = \boldsymbol{D}_{i+1,j+1} + c(x_i, y_j)$ **then**
                $\boldsymbol{\Gamma}_{i,j} \leftarrow \boldsymbol{\Gamma}_{i,j} + \boldsymbol{A}_{i,j} \cdot \boldsymbol{B}_{i+1,j+1}$.
            end if
        else
            if $\boldsymbol{D}_{i,j} = \boldsymbol{D}_{rl_{\tilde{x}}(i)+1, rl_{\tilde{y}}(j)+1} + \boldsymbol{d}_{i,j}$ **then**
                $\gamma \leftarrow \boldsymbol{A}_{i,j} \cdot \boldsymbol{B}_{rl_{\tilde{x}}(i)+1, rl_{\tilde{y}}(j)+1}$.
                Compute $\boldsymbol{D}'$ and $\boldsymbol{d}'$ via Algorithm 2.1 for the subtrees $\tilde{x}^i$ and $\tilde{y}^j$.
                $\boldsymbol{D}'_{1,2} \leftarrow \infty$. $\boldsymbol{D}'_{2,1} \leftarrow \infty$.
                $(\boldsymbol{\Gamma}', |\mathcal{M}(c, \tilde{x}^i, \tilde{y}^j)|) \leftarrow \text{COOPTIMALS}(\tilde{x}^i, \tilde{y}^j, \boldsymbol{D}', \boldsymbol{d}', c)$.
                for $i' \leftarrow 1, \ldots, |\tilde{x}^i|$ **do**
                    for $j' \leftarrow 1, \ldots, |\tilde{y}^j|$ **do**
                        $\boldsymbol{\Gamma}_{i+i'-1, j+j'-1} \leftarrow \boldsymbol{\Gamma}_{i+i'-1, j+j'-1} + \boldsymbol{\Gamma}'_{i',j'} \cdot \gamma$.
                    end for
                end for
            end if
        end if
    end for
    return $(\boldsymbol{\Gamma}, \boldsymbol{A}_{|\tilde{x}|+1, |\tilde{y}|+1})$.
end function

In rough terms, Algorithm 4.1 works as follows. We first compute the matrix $\boldsymbol{A}$, where $\boldsymbol{A}_{i,j}$ essentially contains the number of cooptimal tree mappings between $\tilde{x}$ and $\tilde{y}$ up to nodes x_i and y_j respectively. Then, we compute the matrix $\boldsymbol{B}$, where $\boldsymbol{B}_{i,j}$ essentially contains the number of cooptimal tree mappings between $\tilde{x}$ and $\tilde{y}$, starting from nodes x_i and y_j respectively. Accordingly, the number of cooptimal tree mappings which contain the pairing (i, j) can be computed as $\boldsymbol{\Gamma}_{i,j} = \boldsymbol{A}_{i,j} \cdot \boldsymbol{B}_{i+1,j+1}$, as is visible in line 11 of Algorithm 4.1. What complicates this process, however, is that we also need to compute cooptimal tree mappings between subtrees, which are recursively computed in lines 15-23 of Algorithm 4.1. After executing the algorithm, the desired frequency matrix $\boldsymbol{P}_c(\tilde{x}, \tilde{y})$ can easily be computed as $\boldsymbol{\Gamma}/\boldsymbol{A}_{|\tilde{x}|+1,|\tilde{y}|+1}$, where $/$ denotes the element-wise division.

Note that the version of the algorithm presented here is dedicated to minimize space complexity. By additionally tabulating $\boldsymbol{\Gamma}$ for all subtrees, space complexity rises to $\mathcal{O}(|\tilde{x}|^4 \cdot |\tilde{y}|^4)$ in the worst case, but runtime complexity is reduced to $\mathcal{O}(|\tilde{x}|^3 \cdot |\tilde{y}|^3)$. Another point to note is that the worst case for this algorithm is quite unlikely. First, both input trees would have to be left-heavy, such as in the worst case for the original tree edit distance (Zhang and Shasha 1989). Second, in every step of the computation, multiple options have to be cooptimal, which only occurs in degenerate cases where, for example, the deletion or insertion cost for all symbols is zero.

After obtaining the cooptimal frequency matrix $\boldsymbol{P}_c(\tilde{x}, \tilde{y})$, we can utilize the decomposition above to compute the gradient of the tree edit distance $d_{c_{\vec{\lambda}}}$ with respect to the parameters $\vec{\lambda}$. Similar to Bellet, Habrard, and Sebban (2012), we assume that the cooptimal frequency matrices $\boldsymbol{P}_{c_{\vec{\lambda}}}(\tilde{x}, \tilde{y})$ stay constant under changes of $\vec{\lambda}$. Thus, we obtain the gradient:

$$\begin{aligned}\nabla_{\vec{\lambda}} d_{c_{\vec{\lambda}}}(\tilde{x}, \tilde{y}) &\overset{\text{const. } \boldsymbol{P}_{c_{\vec{\lambda}}}}{=} \sum_{i=1}^{|\tilde{x}|} \sum_{j=1}^{|\tilde{y}|} \boldsymbol{P}_{c_{\vec{\lambda}}}(\tilde{x}, \tilde{y})_{i,j} \cdot \nabla_{\vec{\lambda}} c_{\vec{\lambda}}(x_i, y_j) \\ &+ \sum_{i=1}^{|\tilde{x}|} \Big(1 - \sum_{j=1}^{|\tilde{y}|} \boldsymbol{P}_{c_{\vec{\lambda}}}(\tilde{x}, \tilde{y})_{i,j}\Big) \cdot \nabla_{\vec{\lambda}} c_{\vec{\lambda}}(x_i, -) + \sum_{j=1}^{|\tilde{y}|} \Big(1 - \sum_{i=1}^{|\tilde{x}|} \boldsymbol{P}_{c_{\vec{\lambda}}}(\tilde{x}, \tilde{y})_{i,j}\Big) \cdot \nabla_{\vec{\lambda}} c_{\vec{\lambda}}(-, y_j)\end{aligned} \tag{4.3}$$

This gradient expression is efficiently computable and permits optimization via gradient-based techniques, such as stochastic gradient descent or L-BFGS (Liu and Nocedal 1989).

Note that optimizing the parameters $\vec{\lambda}$ with respect to the GLVQ cost function 2.28 may yield a tree edit distance under which a better MGLVQ model is possible. Therefore, we recommend an alternating optimization scheme with two steps, namely MGLVQ and metric learning, which are iterated until convergence. This yields Algorithm 4.2.

Algorithm 4.2 The tree edit distance learning algorithm for arbitrary parameters $\vec{\lambda}$.

function TED_LEARN(A dataset of trees $\tilde{x}_1, \ldots, \tilde{x}_M$ over some alphabet $\mathcal{A}$, class labels $y_1, \ldots, y_M$, no. of prototypes K, a cost function $c_{\vec{\lambda}}$, and initial parameters $\vec{\lambda}$)
    while E has changed **do**
        Compute pairwise tree edit distances $\boldsymbol{D}_{i,j} = d_{c_{\vec{\lambda}}}(\tilde{x}_i, \tilde{y}_j)$ via Algorithm 2.1.
        $(w_1, \ldots, w_K, E) \leftarrow$ MGLVQ for $\boldsymbol{D}$.
        Compute $\boldsymbol{P}_{c_{\vec{\lambda}}}(\tilde{x}_i, w_k)$ for all i, k via Algorithm 4.1.
        Optimize $\vec{\lambda}$ with respect to the GLVQ cost function E using Equation 4.3.
    end while
    return $(\vec{\lambda}, E)$.
end function

Regarding runtime complexity, computing all pairwise tree edit distance requires $\mathcal{O}(M^2 \cdot |\tilde{x}|^2 \cdot |\tilde{y}|^2)$ steps, training MGLVQ requires $\mathcal{O}(M^2)$ steps, computing the cooptimal frequency matrices for all datapoint-prototype pairs requires $\mathcal{O}(M \cdot |\tilde{x}|^6 \cdot |\tilde{y}|^6)$ in the worst case, after which each gradient computation according to Equations 3.2 and 4.3 requires only $\mathcal{O}(M \cdot |\tilde{x}| \cdot |\tilde{y}|)$ steps. In terms of space, we require $\mathcal{O}(M^2)$ to represent the pairwise distance matrix and $\mathcal{O}(|\tilde{x}|^2 \cdot |\tilde{y}|^2)$ for computing the cooptimal frequency matrices. Therefore, assuming a constant number of iterations τ, we obtain an overall runtime complexity in the order of $\mathcal{O}(\tau \cdot M \cdot |\tilde{x}|^2 \cdot |\tilde{y}|^2 \cdot (M + |\tilde{x}|^4 \cdot |\tilde{y}|^4))$ and an overall space complexity of $\mathcal{O}(M^2 + |\tilde{x}|^2 \cdot |\tilde{y}|^2)$.

Note that the decomposition in Equation 4.2 only holds under the assumption that $c_{\vec{\lambda}}$ is non-negative, self-equal, and fulfills the triangular inequality. Unfortunately, ensuring metric axioms on $c_{\vec{\lambda}}$ imposes additional constraints on the optimization in line 6 of Algorithm 4.2, which prevent us from directly applying gradient-based solvers. To avoid such explicit constraints, we introduce an additional innovation, namely a representation of the alphabet $\mathcal{A}$ via symbol embeddings.

Adaptive Symbol Embeddings

In this section, we introduce *symbol embeddings*, that is, we represent the elements of an alphabet $\mathcal{A}$ by vectors in an Euclidean space. The main motivation for this alternative representation is that the Euclidean distance guarantees pseudo-metric properties, which in turn ensure that the assumptions of the tree edit distance algorithm as well as the decomposition in Equation 4.2 are fulfilled without having to constrain the optimization process. Another advantage is that we can potentially interpret the positions of the vectorial representations and thus gain additional insight regarding the role different symbols play.

We define a symbol embedding as follows.

Definition 4.2 (Symbol Embedding)**.** Let $\mathcal{A}$ be an alphabet with $- \notin \mathcal{A}$. Then, a *symbol embedding* of $\mathcal{A}$ is defined as a mapping $\phi : \mathcal{A} \to \mathbb{R}^m$ for some $m \in \{1, \ldots, |\mathcal{A}|\}$. For any symbol embedding, we define $\phi(-) := \vec{0}$, where $\vec{0}$ is the m-dimensional zero-vector.

Further, we define the cost function c_ϕ with respect to the symbol embedding ϕ as follows.

$$c_\phi(x, y) := \|\phi(x) - \phi(y)\|$$

In other words, c_ϕ is an Euclidean distance on $\mathcal{A} \cup \{-\}$ with the spatial mapping ϕ.

Note that the gradient of $c_\phi(x, y)$ with respect to $\phi(x)$ is given as:

$$\begin{aligned}
\nabla_{\phi(x)} c_\phi(x, y) &= \nabla_{\phi(x)} \sqrt{\big(\phi(x) - \phi(y)\big)^\top \cdot \big(\phi(x) - \phi(y)\big)} \\
&= \frac{\nabla_{\phi(x)} \big(\phi(x) - \phi(y)\big)^\top \cdot \big(\phi(x) - \phi(y)\big)}{2 \cdot \sqrt{\big(\phi(x) - \phi(y)\big)^\top \cdot \big(\phi(x) - \phi(y)\big)}} \\
&= \frac{\phi(x) - \phi(y)}{\|\phi(x) - \phi(y)\|}
\end{aligned} \tag{4.4}$$

Plugging this result into Equation 4.3, we obtain a gradient of the tree edit distance with respect to the embedding vectors for every symbol, under the assumption that the cooptimal frequency matrices stay constant.

$$\nabla_{\phi(x)} \tilde{d}_{c_\phi}(\tilde{x}, \tilde{y}) \overset{\text{const. } \boldsymbol{P}_{c_\phi}}{=} \tag{4.5}$$
$$\sum_{i=1}^{|\tilde{x}|} \delta(x, x_i) \cdot \left[\sum_{j=1}^{|\tilde{y}|} \boldsymbol{P}_{c_\phi}(\tilde{x}, \tilde{y})_{i,j} \cdot \frac{\phi(x) - \phi(y_j)}{\|\phi(x) - \phi(y_j)\|} + \left(1 - \sum_{j=1}^{|\tilde{y}|} \boldsymbol{P}_{c_\phi}(\tilde{x}, \tilde{y})_{i,j}\right) \cdot \frac{\phi(x)}{\|\phi(x)\|} \right]$$
$$+ \sum_{j=1}^{|\tilde{y}|} \delta(x, y_j) \cdot \left[\sum_{i=1}^{|\tilde{x}|} \boldsymbol{P}_{c_\phi}(\tilde{x}, \tilde{y})_{i,j} \cdot \frac{\phi(x) - \phi(x_i)}{\|\phi(x) - \phi(x_i)\|} + \left(1 - \sum_{i=1}^{|\tilde{x}|} \boldsymbol{P}_{c_\phi}(\tilde{x}, \tilde{y})_{i,j}\right) \cdot \frac{\phi(x)}{\|\phi(x)\|} \right]$$

where δ is the Kronecker-Delta, i.e.: $\delta(x, y) = 1$ if $x = y$ and 0 otherwise.

Finally, we can plug this gradient into Equation 3.2 and thus obtain a gradient of the GLVQ cost function with respect to the rows of our symbol embedding matrix. Using this gradient, we can learn a symbol embedding via any gradient-based technique.

Two challenges remain in this setup. First, we have to obtain a viable initialization for the embedding ϕ. In this regard, we suggest to use an initialization, which yields the default edit costs of the tree and sequence edit distance, that is, $c(x, y)$ should be 1 in all cases, except if $x = y$, where it is zero. Indeed, such an initialization exists.

Theorem 4.3. *Let $\mathcal{A} = \{x_1, \ldots, x_n\}$ be an alphabet. Then, the following function $\phi : \mathcal{A} \to \mathbb{R}^n$ with*

$$\phi(x_i)_j := \begin{cases} 0 & \text{if } j > i \\ \rho_j & \text{if } j < i \\ \rho_i \cdot (i+1) & \text{if } j = i \end{cases} \qquad \text{where} \tag{4.6}$$
$$\rho_i = 1/\sqrt{2 \cdot i \cdot (i+1)} \tag{4.7}$$

is a symbol embedding of $\mathcal{A}$ such that:

$$c_\phi(x, y) = \begin{cases} 0 & \text{if } x = y \\ 1 & \text{otherwise} \end{cases}$$

Proof. Refer to Appendix A.14. □

As an example, consider the two-dimensional embedding generated for the alphabet $\mathcal{A} = \{\texttt{a}, \texttt{b}\}$ in Figure 4.2.

The second challenge concerns the handling of the following degenerate cases for the embedding ϕ. First, c_ϕ is non-differentiable with respect to $\phi(x)$ at the point $\phi(x) = \vec{0}$. However, this may not pose a problem as such. In particular, $\phi(x) = \vec{0}$ implies that x is unimportant to distinguish between trees, which is an interesting observation to make. Therefore, we simply extend the definition of the gradient of c_ϕ to be zero at $\phi(x) = \vec{0}$.

Second, ϕ may "flatten" the data too much, in the sense that the matrix $\boldsymbol{\Phi} = (\phi(x_1), \ldots, \phi(x_n))$ for the alphabet $\mathcal{A} = \{x_1, \ldots, x_n\}$ becomes low rank. Such oversimplification effects have previously been observed in generalized matrix learning vector quantization (GMLVQ) as well (Schneider, Bunte, et al. 2010). Indeed, we can make the

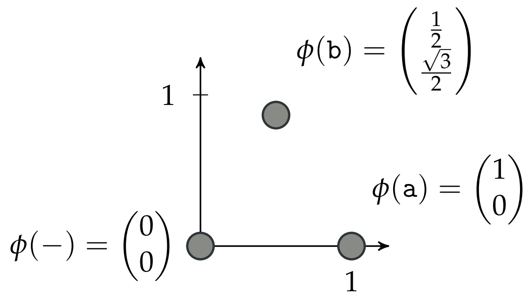


Figure 4.2: The initial 2-dimensional embedding of the alphabet $\mathcal{A} = \{\texttt{a}, \texttt{b}\}$. This embedding ensures that the pairwise costs of all symbols, including the $-$-symbol, is 1.

similarity between GMLVQ and our metric learning scheme more obvious by re-writing $c_\phi(x_i, x_j)$ as follows.

$$
\begin{aligned}
c_\phi(x_i, x_j) &= \sqrt{\left(\phi(x_i) - \phi(x_j)\right)^\top \cdot \left(\phi(x_i) - \phi(x_j)\right)} \\
&= \sqrt{\left(\mathbf{\Phi} \cdot e_i - \mathbf{\Phi} \cdot e_j\right)^\top \cdot \left(\mathbf{\Phi} \cdot e_i - \mathbf{\Phi} \cdot e_j\right)} \\
&= \sqrt{\left(e_i - e_j\right)^\top \cdot \mathbf{\Phi}^\top \cdot \mathbf{\Phi} \cdot \left(e_i - e_j\right)}
\end{aligned}
$$

where e_i is the ith unit vector and e_j is the jth unit vector.

In this representation it is obvious that $\mathbf{\Phi}$ plays the role of the projection matrix $\mathbf{\Omega}$ in GMLVQ (compare to Equation 2.29). As Biehl et al. (2015) have shown, $\mathbf{\Omega}$ tends to very low-rank solutions, which may be overly simplistic in practical cases. To prevent such a low-rank solution, we follow the recommendation of Schneider, Bunte, et al. (2010) and add the regularization term $-\lambda \cdot \log(\det(\mathbf{\Phi} \cdot \mathbf{\Phi}^\top))$ for some constant $\lambda > 0$ to the GLVQ cost function 2.28, which becomes large if any eigenvalue of $\mathbf{\Phi} \cdot \mathbf{\Phi}^\top$ gets close to zero. The gradient of the regularization with respect to $\mathbf{\Phi}$ is the Moore-Penrose Pseudo-Inverse $-\lambda \cdot \mathbf{\Phi}^\dagger$ (Schneider, Bunte, et al. 2010; Petersen and Pedersen 2012).

Finally, the GLVQ cost function 2.28 is inherently scale-invariant in terms of the distances, that is, if we multiply all pairwise distances with some constant, the loss will stay the same. Given this degree of freedom, we may converge to an an embedding with needlessly large scaling. To prevent such a case, we additionally add the regularization term $\lambda \cdot \|\mathbf{\Phi}\|_F$, where $\|\mathbf{\Phi}\|_F$ is the Frobenius norm of $\mathbf{\Phi}$.

This concludes our basic setup for tree edit distance learning via adaptive symbol embeddings. We call our approach em**b**edding **e**dit **d**istance **l**earning (BEDL).

Before we evaluate BEDL experimentally, we wish to highlight one additional desirable property of BEDL, namely that we can utilize alternative initializations and metrics if suitable for the domain at hand. For example, in the domain of natural language processing, we do not need to learn an embedding of words from scratch but can rely on existing word embeddings, such as GloVe (Pennington, Socher, and Manning 2014). We can then learn to *adapt* the word embedding instead of learning it from scratch by only learning a linear transformation $\mathbf{\Omega}$ that maps the pre-existing word embedding into another space in which classification is simpler. Furthermore, for word embeddings, the cosine similarity is typically favored over the Euclidean distance (Pennington, Socher, and Manning 2014). We can include the cosine distance in BEDL easily by re-defining the cost function $c_{\phi,\Omega}$ as follows.

$$c_{\phi,\Omega}(x,y) := \frac{1}{2} \cdot \Big(1 - s_{\Omega}(\phi(x), \phi(y))\Big) \qquad \text{where} \tag{4.8}$$
$$s_{\Omega}(\phi(x), \phi(y)) = \frac{(\Omega \cdot \phi(x))^\top \cdot \Omega \cdot \phi(y)}{\|\Omega \cdot \phi(x)\| \cdot \|\Omega \cdot \phi(y)\|}$$

For the gradient, we obtain:

$$\nabla_{\Omega} c_{\phi,\Omega}(x,y) = -\frac{1}{2} \nabla_{\Omega} s_{\Omega}(\phi(x), \phi(y))$$
$$= -\frac{1}{2} \cdot \Omega \cdot \frac{\phi(x) \cdot \phi(y)^\top + \phi(y) \cdot \phi(x)^\top}{\|\Omega \cdot \phi(x)\| \cdot \|\Omega \cdot \phi(y)\|}$$
$$+ \frac{1}{2} \cdot \Omega \cdot s_{\Omega}(\phi(x), \phi(y)) \cdot \Big[\frac{\phi(x) \cdot \phi(x)^\top}{\|\Omega \cdot \phi(x)\|^2} + \frac{\phi(y) \cdot \phi(y)^\top}{\|\Omega \cdot \phi(y)\|^2}\Big]$$

We will utilize both the basic Euclidean version of BEDL and the cosine distance variation in our next section, in which we evaluate BEDL experimentally.

4.2 EXPERIMENTS

In our experiments we compare the performance of em**b**edding **e**dit **d**istance **l**earning (BEDL) to both the default tree edit distance and the state-of-the-art in terms of tree edit distance learning, namely good edit similarity learning (GESL, Bellet, Habrard, and Sebban 2012).

On each dataset, we perform a crossvalidation[1] and compare the average test error across folds. In particular, we compare the error when using the initial tree edit distance with the error when using the pseudo-edit distance learned via GESL, and the tree edit distance learned via our proposed approach (BEDL).

In general, we would expect that a discriminative metric learned for one classifier also facilitates classification using other classifiers. Therefore, we report the classification error for four classifiers, namely the median generalized learning vector quantization (MGLVQ) classifier, for which our method is optimized, the goodness classifier, for which GESL is optimized (Bellet, Habrard, and Sebban 2012, also refer to Equation 2.4.1), the k-nearest neighbor (KNN) classifier, and the SVM based on the radial basis function kernel. In order to ensure a kernel matrix for SVM, we set negative eigenvalues to zero (clip eigenvalue correction; Gisbrecht and Schleif (2015)). Note that this eigenvalue correction requires cubic runtime in terms of the number of data points and is thus prohibitively slow for large dataset sizes. Therefore, for the Sentiment dataset, we trained the classifiers on a randomly selected sample of 300 points from the training data.

We optimized all hyper-parameters in a nested 5-fold crossvalidation, namely the number of prototypes K for MGLVQ and BEDL in the range $[1, 15]$, the number of

1 We used 20 folds for Strings, Gap, CopenhagenChromosomes, and Sentiment, 10 for Cystic and Leukemia, 8 for Sorting, and 6 for MiniPalindrome. For the programming datasets, the number of folds had to be reduced to ensure that each fold still contained a meaningful number of data points. For the Cystic and Leukemia dataset, our ten folds were consistent with the paper of Gallicchio and Micheli (2013). In all cases, folds were generated such that the label distribution of the overall dataset was maintained.

neighbors for KNN in the range $[1, 15]$, the kernel bandwidth for SVM in the range $[0.1, 10]$, the sparsity parameter ν for the goodness classifier in the range $[10^{-5}, 10]$, and the regularization strength λ for GESL and BEDL in the range $2 \cdot K \cdot M \cdot [10^{-6}, 10^{-2}]$. We chose the number of prototypes for BEDL, as well as the number of neighbors for GESL as the optimal number of prototypes K for MGLVQ.

As implementations, we used custom implementations of KNN, MGLVQ, the goodness classifier, GESL, and BEDL, which are availabe at `https://doi.org/10.4119/unibi/2919994`. For SVM, we utilized the LIBSVM standard implementation (Chang and Lin 2011). All experiments were performed on a consumer-grade laptop with an Intel Core i7-7700 HQ CPU.

Artificial Datasets

We evaluate the default tree edit distance, GESL, and BEDL on the Strings and on the Gap data set from Section 3.2. The results are shown in Table 4.1. In both datasets, BEDL could reduce the error consistently to 0%. Closer inspection revealed that BEDL did indeed consistently identify the desired representation, namely embedding the symbols `a` and `b` as well as `c` and `d` at the same point respectively (also refer to Figure 4.3, left).

By contrast, GESL only achieved low errors for the goodness classifier, while remaining at high errors for all other classifiers. Using a one sided Wilcoxon signed-rank test we found that BEDL significantly outperformed GESL and the initial edit distance for the KNN and MGLVQ classifiers on both datasets ($p < 0.001$ after Bonferroni correction).

Note that the GESL results differ from the results reported in Section 3.2. This is likely due to different crossvalidation folds, the fact that we used an MGLVQ instead of a RGLVQ classifier, and the fact that we optimized classifier hyper-parameters, which may have lead to different choices compared to the previous experiments.

Regarding runtime, we note that GESL is clearly faster due to its convex programming structure with a runtime advantage of about factor 20-30.

Real-World Data

Beyond the artificial data, we evaluated our methods on six real-world datasets.

CopenhagenChromosomes: A balanced two-class dataset of 400 chromosome density strings, as described in Section 3.2.

MiniPalindrome: A balanced eight-class dataset of 48 Java programs, where each class represents one strategy to detect whether an input string contains only palindromes (Paaßen 2016b). The programs are represented by their abstract syntax tree, where the label corresponds to one of 24 programming concepts (e.g. class declaration, function declaration, method call, etc.).

Sorting: A two-class dataset of 64 Java sorting programs as described in Section 3.2.

Cystic: A dataset of 160 glycan molecules where the class label 1 is assigned to every molecule associated with cystic fibrosis and 0 is assigned to other molecules. The molecules were extracted from the KEGG/Glycan data base (Hashimoto et al. 2006)

Table 4.1: The mean test classification error and runtimes for metric learning on the artificial datasets, averaged over the cross validation trials, as well as the standard deviation. The x-axis shows the metric learning schemes, the y-axis the different classifiers used for evaluation. The table is sub-divided for each dataset. The lowest classification error for each dataset is highlighted via bold print.

classifier	initial	GESL	BEDL
		Strings	
KNN	$21.0 \pm 10.2\%$	$23.0 \pm 10.8\%$	$\mathbf{0.0 \pm 0.0}\%$
MGLVQ	$36.0 \pm 15.7\%$	$34.0 \pm 11.0\%$	$\mathbf{0.0 \pm 0.0}\%$
SVM	$9.0 \pm 11.2\%$	$10.0 \pm 8.6\%$	$\mathbf{0.0 \pm 0.0}\%$
goodness	$11.5 \pm 9.3\%$	$0.5 \pm 2.2\%$	$\mathbf{0.0 \pm 0.0}\%$
runtime [s]	0 ± 0	0.030 ± 0.002	1.077 ± 0.098
		Gap	
KNN	$30.0 \pm 10.8\%$	$22.5 \pm 16.8\%$	$\mathbf{0.0 \pm 0.0}\%$
MGLVQ	$49.5 \pm 17.0\%$	$48.5 \pm 16.6\%$	$\mathbf{0.0 \pm 0.0}\%$
SVM	$\mathbf{0.0 \pm 0.0}\%$	$5.0 \pm 13.6\%$	$\mathbf{0.0 \pm 0.0}\%$
goodness	$0.5 \pm 2.2\%$	$0.5 \pm 2.2\%$	$\mathbf{0.0 \pm 0.0}\%$
runtime [s]	0 ± 0	0.037 ± 0.004	0.865 ± 0.139

according to the scheme described by Gallicchio and Micheli (2013). Each molecule is represented as a tree, where the label corresponds to mono-saccharide identifiers (one out of 29) and the roots are chosen according to biological meaning (Hashimoto et al. 2006).

Leukemia: A dataset of 442 glycan molecules from the same source as the Cystic dataset. For this dataset, a class label 1 represents that the molecule is associated with Leukemia.

Sentiment: A large-scale two-class dataset of 9613 sentences from movie reviews, where one class (4650 trees) corresponds to negative and the other class (4963 trees) to positive reviews. The sentences are represented by their syntax trees, where inner nodes are unlabeled and leaves are labeled with one of over $30,000$ words (Socher, Pennington, et al. 2011). Note that GESL is not practically applicable for this dataset, as the number of parameters to learn scales quadratically with the number of words, i.e. $> 30,000^2$. To make BEDL applicable in this case, we do not learn a full embedding, but instead we initialize the embedding matrix with the 300-dimensional Common Crawl GloVe embedding (Pennington, Socher, and Manning 2014), which we reduce via PCA, retaining 95% of the data variance ($m = 16.4 \pm 2.3$ dimensions on average $\pm$ standard deviation). We adapt this initial embedding via a linear transformation, using the cosine distance (refer to Equation 4.8) instead of the Euclidean distance, as introduced in the previous section.

The results of our experiments are displayed in Table 4.2. In all datasets and for all classifiers, BEDL yields lower classification error compared to GESL. Furthermore, in four of six datasets, BEDL yields the best overall classification results (the exceptions being CopenhagenChromosomes and Cystic). In five out of six cases, BEDL could improve the

Table 4.2: The mean test classification error and runtimes for metric learning on the real-world datasets, averaged over the cross validation trials, as well as the standard deviation. The x-axis shows the metric learning schemes, the y-axis the different classifiers used for evaluation. The table is sub-divided for each dataset. The lowest classification error for each dataset is highlighted via bold print.

classifier	initial	GESL	BEDL
		CopenhagenChromosomes	
KNN	$4.5 \pm 4.6\%$	$14.8 \pm 7.7\%$	$6.2 \pm 7.6\%$
MGLVQ	$13.2 \pm 7.8\%$	$26.8 \pm 9.4\%$	$11.2 \pm 8.4\%$
SVM	$\mathbf{2.7 \pm 3.4}\%$	$21.2 \pm 10.6\%$	$5.3 \pm 7.2\%$
goodness	$3.0 \pm 4.1\%$	$7.0 \pm 6.2\%$	$6.0 \pm 7.7\%$
runtime [s]	0 ± 0	4.833 ± 1.200	10.267 ± 1.954
		MiniPalindrome	
KNN	$12.5 \pm 11.2\%$	$12.5 \pm 7.9\%$	$10.4 \pm 9.4\%$
MGLVQ	$2.1 \pm 5.1\%$	$4.2 \pm 6.5\%$	$\mathbf{0.0 \pm 0.0}\%$
SVM	$4.2 \pm 6.5\%$	$20.8 \pm 15.1\%$	$\mathbf{0.0 \pm 0.0}\%$
goodness	$6.2 \pm 6.8\%$	$14.6 \pm 5.1\%$	$8.3 \pm 10.2\%$
runtime [s]	0 ± 0	0.103 ± 0.014	2.785 ± 0.631
		Sorting	
KNN	$15.6 \pm 8.8\%$	$18.8 \pm 16.4\%$	$10.9 \pm 8.0\%$
MGLVQ	$14.1 \pm 10.4\%$	$14.1 \pm 8.0\%$	$14.1 \pm 8.0\%$
SVM	$10.9 \pm 8.0\%$	$\mathbf{9.4 \pm 8.8}\%$	$\mathbf{9.4 \pm 8.8}\%$
goodness	$15.6 \pm 11.1\%$	$17.2 \pm 14.8\%$	$17.2 \pm 9.3\%$
runtime [s]	0 ± 0	0.352 ± 0.102	3.358 ± 0.748
		Cystic	
KNN	$31.2 \pm 6.6\%$	$32.5 \pm 10.1\%$	$28.1 \pm 8.5\%$
MGLVQ	$34.4 \pm 6.8\%$	$33.1 \pm 9.8\%$	$30.0 \pm 10.1\%$
SVM	$28.1 \pm 9.0\%$	$33.1 \pm 8.9\%$	$29.4 \pm 12.5\%$
goodness	$28.1 \pm 8.5\%$	$26.2 \pm 14.4\%$	$\mathbf{24.4 \pm 13.3}\%$
runtime [s]	0 ± 0	0.353 ± 0.292	0.864 ± 0.767
		Leukemia	
KNN	$7.5 \pm 2.6\%$	$8.2 \pm 4.6\%$	$7.3 \pm 4.3\%$
MGLVQ	$9.5 \pm 4.0\%$	$10.9 \pm 4.7\%$	$9.5 \pm 3.0\%$
SVM	$7.0 \pm 4.1\%$	$8.8 \pm 2.9\%$	$6.8 \pm 4.7\%$
goodness	$\mathbf{6.1 \pm 4.3}\%$	$10.0 \pm 4.4\%$	$6.3 \pm 3.8\%$
runtime [s]	0 ± 0	2.208 ± 0.919	6.550 ± 2.706
		Sentiment	
KNN	$40.2 \pm 2.8\%$	–	$38.2 \pm 3.3\%$
MGLVQ	$44.0 \pm 2.6\%$	–	$41.3 \pm 5.7\%$
SVM	$34.3 \pm 3.0\%$	–	$\mathbf{33.3 \pm 3.6}\%$
goodness	$43.7 \pm 1.9\%$	–	$42.5 \pm 3.1\%$
runtime [s]	0 ± 0	–	69.385 ± 58.064

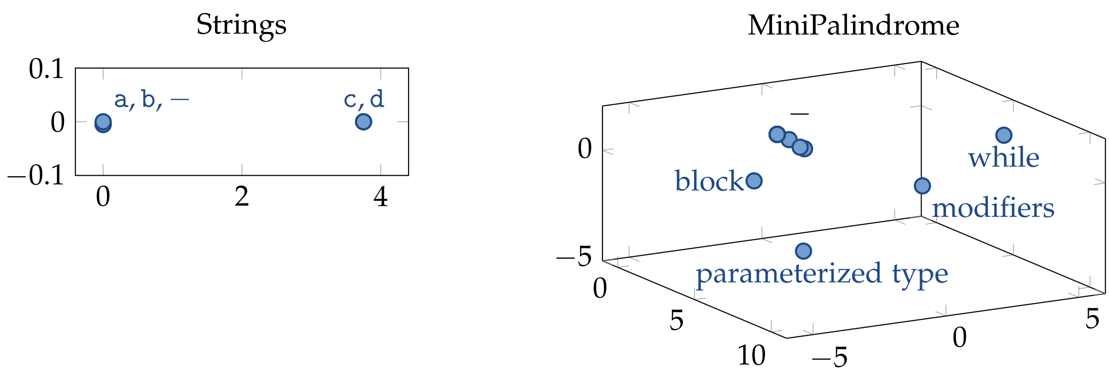


Figure 4.3: A PCA of the learned embeddings for the Strings (left) and the MiniPalindrome dataset (right), covering 100% and 83.54% of the variance respectively.

accuracy for KNN (except for CopenhagenChromosomes), in four out of six cases for SVM (the exception being CopenhagenChromosomes and Cystic), in four out of six cases for MGLVQ (in Sorting and Leukemia it stayed equal), and in two out of six cases for the goodness classifier. For the Sentiment datasets we can also verify this result statistically with $p < 0.05$ for all classifiers.

Note that the focus of our work is to improve classification accuracy via metric learning, not to develop state-of-the-art classifiers as such. However, we note that our results for the Sorting dataset outperform the best reported results by Paaßen, Mokbel, and Hammer (2016) of 15%. For the Cystic dataset we improve the AUC from $76.93 \pm 0.97\%$ mean and standard deviation across crossvalidation trials to $79.2 \pm 13.6\%$, and for the Leukemia dataset from $93.8 \pm 3.3\%$ to $94.6 \pm 4.5\%$. Both values are competitive with the results obtained via recursive neural networks and a multitude of graph kernels by Gallicchio and Micheli (2013). For the Sentiment dataset, we obtain a SVM classification error of 27.51% on the validation set, which is noticeably worse than the reported literature results of around 12.5% (Socher, Pennington, et al. 2011). However, we note that we used considerably less data to train our classifier due to the cost of eigenvalue correction (only 500 points for the validation).

While most embeddings of BEDL where too intrinsically high-dimensional to inspect visually, the embedding for the MiniPalindrome dataset revealed that most symbols could be embedded close to zero while a few discriminative syntactic concepts remained distinct from zero, thus giving an indication of the relevant syntactic concepts for the given task (refer to Figure 4.3).

Interestingly, GESL tended to decrease classification accuracy compared to the initial tree edit distance. Likely, GESL requires more neighbors K for better results (Bellet, Habrard, and Sebban 2012). However, scaling up to a high number of neighbors lead to prohibitively high runtimes for our experiments such that we do not report these results here. These high runtimes can be explained by the fact that the number of slack variables in GESL increases with $\mathcal{O}(M \cdot K)$ where M is the number of data points and K is the number of neighbors. The scaling behavior is also visible in our experimental results. For datasets with few data points and neighbors, such as Strings, MiniPalindrome, and Sorting, GESL is 10 to 30 times faster compared to BEDL. However, for CopenhagenChromosomes, Cystic, and Leukemia, the runtime advantage shrinks to a factor of 2 to 3.

Ablation Studies

In ablation studies, we studied the difference between GESL and BEDL in more detail. In particular, we tested the following different design choices

1. Classic GESL (G1),
2. GESL using cooptimal frequency matrices instead of a single tree mapping matrix (G2),
3. GESL using cooptimal frequency matrices and the prototypes from MGLVQ as neighbors N^+ and N^- (G3),
4. LVQ tree edit distance learning, directly learning the cost function parameters instead of an embedding, with a pseudo-metric normalization after each gradient step (L1), and
5. BEDL as proposed (L2).

Note that, for the ablation studies, we re-used the hyper-parameters which were optimal for the reference versions of the methods (G1 and L2).

Figure 4.4 shows the average classification error and standard deviation (as error bars) for all tree-structured datasets and the string dataset, both for the pseudo-edit distance as in Equation 4.2, and for the actual tree edit distance using the learned cost function.

We observe that using cooptimal frequency matrices (G2) and MGLVQ prototypes instead of ad-hoc nearest neighbors (G3) improved GESL on the MiniPalindrome dataset, worsened it for the strings dataset, and otherwise showed no remarkable difference for the Sorting, Cystic, and Leukemia dataset.

Regarding the LVQ tree edit distance learning variants L1 and L2, we note that BEDL improved the error for the actual tree edit distance but worsened the result for the pseudo-edit distance.

In general, GESL variants performed better for the pseudo-edit distance than for the actual tree edit distance, and LVQ variants performed better for the actual tree edit distance compared to the pseudo-edit distance.

4.3 CONCLUSION

In this chapter, we have proposed em**b**edding **e**dit **d**istance **l**earning (BEDL), a novel approach for tree edit distance learning that goes beyond the state-of-the-art in three key aspects. First, we optimize metric parameters with respect to *all* cooptimal tree mappings between the input trees, not only one Viterbi-mapping. Second, we utilize a median generalized learning vector quantization (MGLVQ) model, which enables us to perform the core metric optimization in linear time and permits us to optimize the metric directly for classification, without having to select ad hoc reference pairs. Third, and most importantly, we learn a symbol embedding instead of pairwise replacement costs that guarantees metric properties and permits additional interpretation.

In our experiments we have shown that BEDL improves upon the state-of-the-art of good edit similarity learning for trees on a diverse tree datasets including Java program

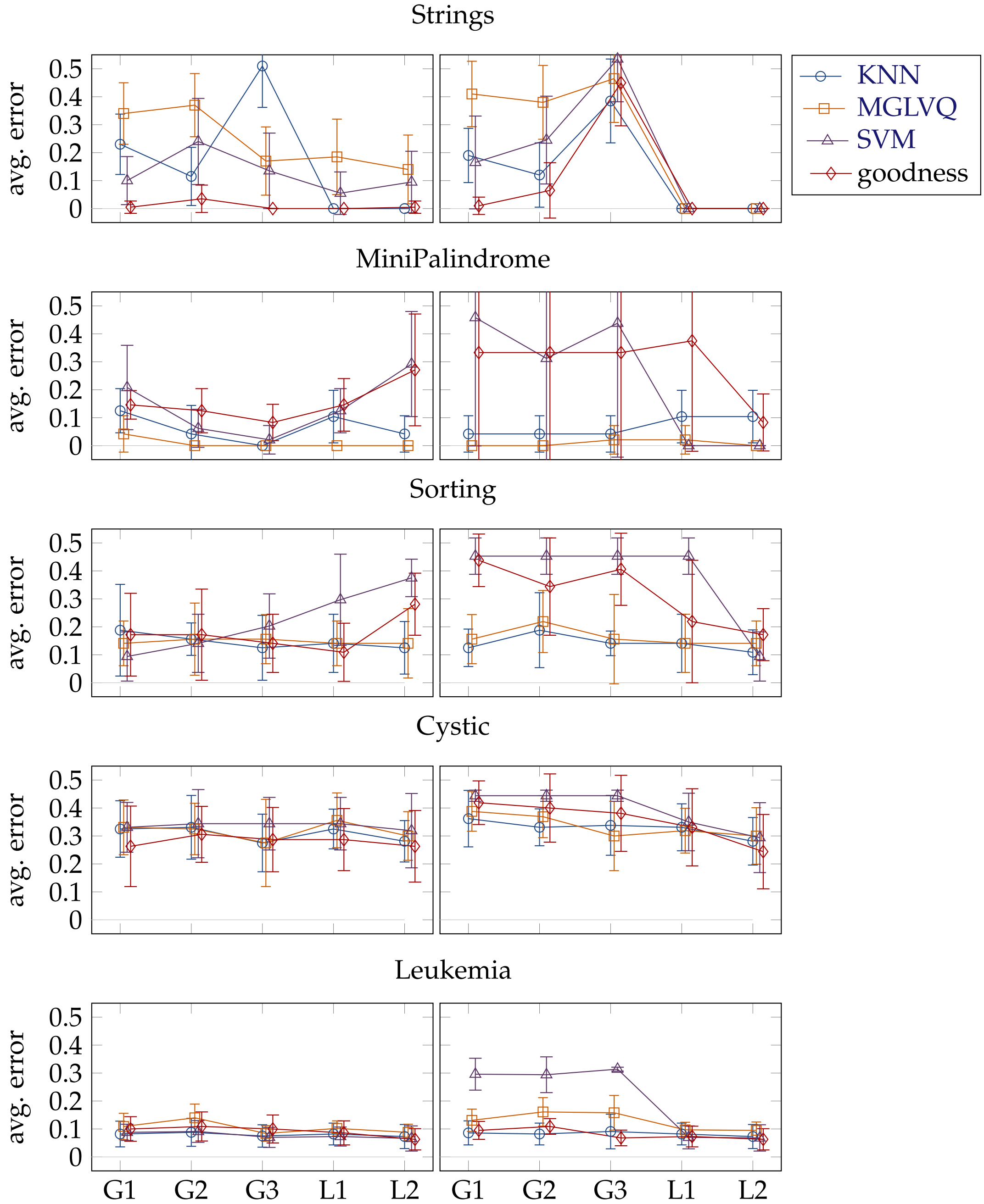


Figure 4.4: Ablation results for all tree-structured datasets and the strings dataset. Each row of the figure shows the results for one dataset. The left column shows the results for the pseudo-edit distance, the right column for the actual tree edit distance. The x-axis in each plot displays the different design choices as described in the text (from G1 to L2), the y-axis in each plot displays the mean classification error after metric learning, averaged across crossvalidation trials, with error bars displaying the standard deviation. The different lines in each plot display the different classifiers used for evaluation.

syntax trees, tree-based molecule representations from a biomedical task, and syntax trees in natural language processing.

Now that we have developed methods to obtain viable edit distances for various cases of structured data, our next challenge is to utilize these edit distances for downstream predictive tasks. We have already demonstrated our ability to perform classification. In the next chapter, we cover time series prediction.

5 TIME SERIES PREDICTION FOR STRUCTURED DATA

Summary: Graph theory is a flexible and general formalism providing rich models in various important domains, such as distributed computing, intelligent tutoring systems, or social network analysis. In many cases, such models need to take changes in the graph structure into account, that is, changes in the number of nodes or in the graph connectivity. Predicting such changes within graphs can be expected to yield insight with respect to the underlying dynamics, e.g. with respect to user behavior. However, predictive techniques in the past have almost exclusively focused on single edges or nodes. In this chapter, we attempt to predict the future state of a graph as a whole.

Using the theory of pseudo-Euclidean and kernel embeddings outlined in Section 2.1, we propose to phrase time series prediction as a regression problem in an implicit vectorial space. Under this perspective, we can perform time series prediction via non-parametric regression techniques, such as 1-nearest neighbor regression, kernel regression, or Gaussian process regression. The output of the regression is another point in the implicit space, which can be subsequently processed using distance-based or kernel techniques.

We evaluate our approach on two well-established theoretical models of graph evolution as well as two real datasets from the domain of intelligent tutoring systems. We find that simple regression methods, such as kernel regression, are sufficient to capture the dynamics in the theoretical models but that Gaussian process regression significantly improves the prediction error for real-world data.

Publications: This chapter is based on the following publications.

- Paaßen, Benjamin, Christina Göpfert, and Barbara Hammer (2016). "Gaussian process prediction for time series of structured data". In: *Proceedings of the 24th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN 2016).* (Bruges, Belgium). Ed. by Michel Verleysen. i6doc.com, pp. 41–46. URL: `http://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2016-109.pdf`.
- — (2018). "Time Series Prediction for Graphs in Kernel and Dissimilarity Spaces". In: *Neural Processing Letters* 48.2, pp. 669–689. DOI: `10.1007/s11063-017-9684-5`.

Source Code: The MATLAB(R) source code is available at `http://doi.org/10.4119/unibi/2913104`.

Graphs provide an ideal theoretical framework to model connective structure between entities, for example traffic connections between and within cities (Papageorgiou 1990), data lines between computing nodes (Casteigts et al. 2012), communication between people in social networks (Liben-Nowell and Kleinberg 2007), or the structure of a student's solution to a learning task in an intelligent tutoring system (Mokbel, Gross, et al. 2013, also refer to Chapter 6). However, a static view of graphs is seldom sufficient. In all the previous examples, nodes as well as connections change significantly over time. In traffic graphs, the traffic load changes significantly over the course of a day, making optimal routing a time-dependent problem (Papageorgiou 1990); in distributed computing, the distribution of computing load and communication between machines crucially depends on the availability and speed of connections and the current load of the machines, which changes over time (Casteigts et al. 2012); in social networks or communication networks, new users may enter the network, old users may leave, and the interactions between users may change rapidly (Liben-Nowell and Kleinberg 2007); and in intelligent tutoring systems, students change their solution over time to get closer to a correct solution (Koedinger et al. 2013; Mokbel, Gross, et al. 2013, also refer to Chapter 6). In all these cases it would be beneficial to predict the next state of the graph in question, because it provides the opportunity to optimize system behavior in light of possible future developments, for example by re-routing traffic, providing additional bandwidth where required, or by providing helpful hints to students.

Traditionally, predicting the future development based on knowledge of the past is the topic of *time series prediction*, which has wide-ranging applications in physics, sociology, medicine, engineering, finance, and other fields (Sapankevych and Sankar 2009; Shumway and Stoffer 2013). However, classic models in time series prediction such as ARIMA, NARX, Kalman filters, recurrent neural networks, or reservoir models focus on vectorial data representations and thus are not equipped to handle time series of graphs (Shumway and Stoffer 2013). Accordingly, past work on predicting changes in graphs has focused on simpler sub-problems that can be phrased as vectorial prediction problems, e.g. predicting the overall load in an energy network (A. Ahmad et al. 2014) or predicting the appearance of single edges in a social network (Liben-Nowell and Kleinberg 2007).

In this contribution, we develop an approach to address the time series prediction problem for graphs, which we frame as a regression problem with structured data as input *and as output*. Our approach has two key steps: First, we represent graphs via pairwise distances or kernel values, which are well-researched in the scientific literature (refer to Section 2.2). This representation implicitly embeds the discrete set of graphs in a continuous vectorial space (refer to Section 2.1). Second, within this space, we can apply non-parametric regression methods, such as nearest neighbor regression, kernel regression (Nadaraya 1964), or Gaussian processes (Rasmussen and Williams 2005) to predict the next position in the kernel space given the current position. Note that this does *not* provide us with the graph that corresponds to the predicted point in the kernel space. Indeed, identifying the corresponding graph in the primal space is a *kernel pre-image problem* that is in general hard to solve (Bakır, Weston, and Schölkopf 2003; Bakır, Zien, and Tsuda 2004; Kwok and I. W.-H. Tsang 2004). However, we will show that this data point can still be analyzed with subsequent kernel- or distance-based methods.

A drawback of GPR is its cubic computational complexity in the number of datapoints due to a kernel matrix inversion. Fortunately, Deisenroth and Ng (2015) have developed a simple strategy to permit predictions in linear time, namely distributing the prediction to multiple Gaussian processes, each of which handles only a constant-sized subset of the data.

The key contributions of our work are the following. First, we provide an integrative overview of research on time-varying graphs. Second, we provide a novel scheme for time series prediction in pseudo-Euclidean and kernel spaces. This scheme is compatible with explicit vectorial embeddings, as are provided by some graph kernel (Borgwardt and Kriegel 2005; Aiolli, Martino, and Sperduti 2015; Bacciu, Errica, and Micheli 2018), but does not require such a representation. Third, we discuss how the predictive result, which is a point in an implicit kernel feature space, can be analyzed using subsequent

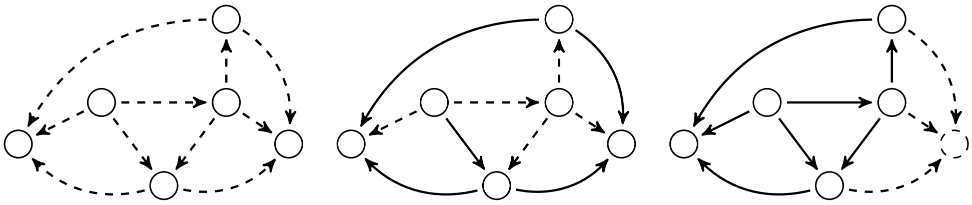


Figure 5.1: An example of a time-varying graph modeling a public transportation graph drawn for three points in time: night time (left), the early morning (middle) and mid-day (right). Present edges or nodes are drawn as solid lines, non-present edges or nodes are drawn as dashed lines.

kernel- or distance-based methods. Fourth, we provide an efficient realization of our prediction pipeline for Gaussian processes in linear time. Finally, we evaluate our proposed approaches on two theoretical and two practical data sets.

5.1 BACKGROUND AND RELATED WORK

Time-varying graphs are relevant in many different fields, such as traffic (Papageorgiou 1990), distributed computing (Casteigts et al. 2012), social networks (Liben-Nowell and Kleinberg 2007), or intelligent tutoring systems (Koedinger et al. 2013; Mokbel, Gross, et al. 2013). Due to the breadth of the field, we focus here on relatively general concepts that can be applied to a wide variety of domains.

Models of Graph Dynamics

Time-Varying Graphs: Time-varying graphs have been introduced by Casteigts et al. (2012) in an effort to integrate different notations found in the fields of delay-tolerant networks, opportunistic-mobility networks, and social networks. The authors note that changes in graphs for these domains should not be regarded as anomalies, but rather as an "integral part of the nature of the system" (Casteigts et al. 2012). Here, we present a slightly simplified version of the notation developed in their work.

Definition 5.1 (Time-Varying Graph (Casteigts et al. 2012))**.** A *time-varying graph* is defined as a five-tuple $\mathcal{G} = (V, E, \mathcal{T}, \psi, \rho)$ where

- V is an arbitrary set called *nodes*,
- $E \subseteq V \times V$ is a set of node tuples called *edges*,
- $\mathcal{T} = \{t \in \mathbb{N} | t_0 \leq t \leq T\}$ for some $t_0, T \in \mathbb{N}$ is called *lifetime* of the graph,
- $\psi : V \times \mathcal{T} \to \{0, 1\}$ is called *node presence function*, and node x is called *present* at time t if and only if $\psi(x, t) = 1$, and
- $\rho : E \times \mathcal{T} \to \{0, 1\}$ is called *edge presence function*, and edge e is called *present* at time t if and only if $\rho(e, t) = 1$.

In figure 5.1, we show an example of a time-varying graph modeling the connectivity in simple public transportation graph over the course of a day. In this example, nodes model stations and edges model train connections between stations. In the night (left), all nodes may be present but no edges, because no lines are active yet. During the early morning (middle), some lines become active while others remain inactive. Finally, in mid-day (right), all lines are scheduled to be active, but due to a disturbance - e.g. construction work - a station is closed and all adjacent connections become unavailable.

Note that the concept of time-varying graphs generally assumes all nodes and edges to be known in advance. In domains where that is not the case, one can frame the underlying graph as a fully connected graph with infinitely many nodes, from which only a finite subset is present at any given time.

Using the notion of a presence function, we can generalize many interesting concepts from classic graph theory to a dynamic version. In particular, we can define the *temporal subgraph* $\mathcal{G}_t$ of graph $\mathcal{G}$ at time t as the graph of all nodes and edges of $\mathcal{G}$ that are present at time t, that is $\mathcal{G}_t := (V_t, E_t)$ where

$$V_t := \{v \in V | \psi(v,t) = 1\}, \quad E_t := \{(u,v) \in E | \rho\big((u,v),t\big) = 1\} \tag{5.1}$$

Further, we can define the neighborhood of a node $u \in V_t$ at time t as the set of nodes $N_t(u) := \{v \in V_t | (u,v) \in E_t\}$; we can define a path between $u \in V_t$ and $v \in V_t$ at time t as a sequence of nodes $v_0, \ldots, v_K \in V_t$ such that $v_0 = u$, $v_K = v$, and for all $k \in \{1, \ldots, K\}$ it holds: $(v_{k-1}, v_k) \in E_t$; and we can call two nodes $u \in V_t$ and $v \in V_t$ connected at time t if a path between them exists at time t.

Note that we have assumed discrete time in our definition of a time-varying graph. This is justified by the following consideration. Even if time is continuous, changes to the graph take the form of discrete value changes in the node or edge presence function, because a presence function can only take the values 0 or 1. Let us call such discrete change points *events*. Assuming that there are only finitely many such events, we can write all events in the lifetime of a graph as an ascending sequence $t_1, \ldots, t_T$. Accordingly, all changes in the graph are fully described by the sequence of temporal subgraphs $\mathcal{G}_{t_1}, \ldots, \mathcal{G}_{t_T}$ (Casteigts et al. 2012; Scherrer et al. 2008). Therefore, even time-varying graphs defined on continuous time can be fully described by considering the discrete lifetime $\{1, \ldots, T\}$.

Sequential Dynamical Systems: Sequential dynamical systems (SDS) have been introduced by Barrett, Mortveit, and Reidys (2000) as a generalization of cellular automata to arbitrary neighborhood structures. In essence, SDSs assign a binary state $\psi(x,t)$ to each node x in a static graph $\mathcal{G} = (V, E)$. This state is updated according to a transition function f_x, which maps the current states of the node and all of its neighbors to the next state of the node x itself. This induces a discrete dynamical system on graphs (where edges and neighborhoods stay fixed) (Barrett, Mortveit, and Reidys 2000; Barrett, Mortveit, and Reidys 2003; Barrett and Reidys 1999). Interestingly, SDSs can be related to time-varying graphs by interpreting the binary state of a node x at time t as the value of its presence function $\psi(x,t)$. Note that we can predict the future state of an SDS by simply executing the SDS transition function f_x for all nodes x repeatedly. As such, SDSs provide elegant and compact models for time series prediction on graphs. Indeed, we use an SDS in our experimental section to compactly describe Conway's *Game of Life* (Gardner 1970). Unfortunately, there are no learning schemes to date that can infer an SDS from data. Therefore, other predictive methods are required.

Predicting Changes in Graphs

To our knowledge, there does not exist a time series prediction for graphs as a whole. However, ample prior work has focused on more specific predictive problems, namely the prediction of new edges and nodes.

Link Prediction: In the realm of social network analysis, Liben-Nowell and Kleinberg (2007) have formulated the *link prediction problem*, which can be stated as follows: Given a time series of temporal subgraphs $\mathcal{G}_0, \ldots, \mathcal{G}_t$ for a time-varying graph $\mathcal{G}$, which edges will be added to the graph in the next time step, i.e. for which edges do we find $\rho(e, t) = 0$ but $\rho(e, t+1) = 1$? For example, given all past collaborations in a scientific community, can we predict new collaborations in the future?

The simplest approach to address this problem is to compute a similarity index $s(u, v)$ between nodes (u, v) for which $\rho((u, v), t) = 0$, and to predict $\rho((u, v), t+1) = 1$ if and only if $s(u, v)$ exceeds a certain threshold (Liben-Nowell and Kleinberg 2007; Lichtenwalter, Lussier, and Chawla 2010). Typical similarity indices for this purpose include the number of common neighbors at time t, the Jaccard index at time t, or the Adar index at time t (Liben-Nowell and Kleinberg 2007). A more recent approach is to train a classifier that predicts the value of the edge presence function $\rho(e, t+1)$ for all edges with $\rho(e, t) = 0$ using a vectorial feature representation of the edge e at time t, where features include the similarity indices discussed above (Lichtenwalter, Lussier, and Chawla 2010). In a survey, Lü and Zhou (2011) further list maximum-likelihood approaches on stochastic models and probabilistic relational models for link prediction.

Growth models: In a seminal paper, Barabási and Albert (1999) described a simple model to incrementally grow an undirected graph node by node from a small, fully connected seed graph (also refer to the experimental section below). Since then, many other models of graph growth have emerged, most notably stochastic block models and latent space models (Clauset 2013; Goldenberg et al. 2010). Stochastic block models assign each node to a block and model the probability of an edge between two nodes only dependent on their respective blocks (Holland, Laskey, and Leinhardt 1983). Latent space models embed all nodes in an underlying, latent space and model the probability of an edge depending on the distance in this space (Hoff, Raftery, and Handcock 2002). Both classes of models can be used for link prediction as well as graph generation. Further, they can be trained with pre-observed data in order to provide more accurate models of the data. However, graph growth models have two severe drawbacks. First, they do not cover deletions of nodes or edges, and second, they typically can not guarantee accurate predictions in detail, but only high-level properties, such as a certain edge degree distribution. As such, using growth models for time series prediction would likely yield unsatisfactory results.

In the next section, we develop our own method to predict general changes in graphs.

5.2 METHOD

Starting from the theory of time-varying graphs, we can formalize time series prediction for graphs as the problem of predicting the next temporal subgraph $\mathcal{G}_{t+1}$, given the time series of past temporal subgraphs $\mathcal{G}_0, \ldots, \mathcal{G}_t$. More precisely, let $\mathcal{X}$ denote the set of possible subgraphs and let $\mathcal{X}^*$ denote the set of possible time series over $\mathcal{X}$. Then, we wish to construct a function $f : \mathcal{X}^* \to \mathcal{X}^*$ that maps any time series $\mathcal{G}_0, \ldots, \mathcal{G}_t$ to its continued version $\mathcal{G}_0, \ldots, \mathcal{G}_{t+1}$.

Distance-Based Time Series Prediction

In our case, we assume that either a pseudo-Euclidean distance d over $\mathcal{X}^*$ or a kernel k over $\mathcal{X}^*$ is available. Recall that the former case is more general since any kernel k implies a Euclidean distance, but not vice versa (refer to Section 2.1). Therefore, we will focus here on the more general case of pseudo-Euclidean distances.

First, recall that the set of pseudo-Euclidean distances is equivalent to the set of functions $d : \mathcal{X}^* \times \mathcal{X}^* \to \mathbb{R}$ that are symmetric and self-equal, thus covering a broad range of functions including all possible metrics, especially edit distances. We can construct such a function easily, for example by first applying any of the graph edit distance approaches from Section 2.3.4, and then plugging these distances into a sequence edit distance from Section 2.3.1. Another option is to assume that the next temporal subgraph $\mathcal{G}_{t+1}$ is conditionally independent from all temporal subgraphs $\mathcal{G}_0, \ldots, \mathcal{G}_{t-1}$ if conditioned on $\mathcal{G}_t$, i.e. a *Markov assumption*. In that case, we can compute a viable distance between two time series of graphs by simply computing a distance between the end points of these time series. We remain agnostic regarding any such design choice and only require that d is some pseudo-Euclidean distance over $\mathcal{X}^*$, that is, d is symmetric and self-equal.

Second, recall that d being pseudo-Euclidean means, per definition, that there exist two mappings $\phi^+ : \mathcal{X}^* \to \mathbb{R}^m$ and $\phi^- : \mathcal{X}^* \to \mathbb{R}^n$ such that for any $\bar{x}, \bar{y} \in \mathcal{X}^*$, the squared distance $d(\bar{x}, \bar{y})^2$ is equivalent to the difference between the squared standard Euclidean distances $\|\phi^+(\bar{x}) - \phi^+(\bar{y})\|^2$ and $\|\phi^-(\bar{x}) - \phi^-(\bar{y})\|^2$ (also refer to Equation 2.7). We denote the concatenation of both spatial maps as $\phi : \mathcal{X}^* \to \mathbb{R}^{m+n}$, that is, for all $\bar{x} \in \mathcal{X}^*$ we define:

$$\phi(\bar{x}) := \begin{pmatrix} \phi^+(\bar{x}) \\ \phi^-(\bar{x}) \end{pmatrix}$$

Given this representation, we can re-phrase our time series prediction task as follows. Assume we are given a training dataset $\{\mathcal{G}_t^j\}_{t=1,\ldots,T_j}^{j=1,\ldots,N} \subset \mathcal{X}$ of time series over graphs. Now, let $M = T_1 + \ldots + T_N$, let (j, t) be the ith time series/step index-tuple in lexicographic ordering, let $\bar{x}_i := \mathcal{G}_1^j, \ldots, \mathcal{G}_t^j$, and let $\bar{y}_i := \mathcal{G}_1^j, \ldots, \mathcal{G}_{t+1}^j$.

Then, our aim is to construct a function $f : \mathbb{R}^{m+n} \to \mathbb{R}^{m+n}$ such that for all $i \in \{1, \ldots, M\}$ it holds: $f\big(\phi(\bar{x}_i)\big) = \phi(\bar{y}_i)$. Note that this new version of the problem has the shape of a classic regression problem with input data $\vec{x}_i = \phi(\bar{x}_i)$ and output data $\vec{y}_i = \phi(\bar{y}_i)$ which we can address using non-parametric regression techniques as in Section 2.6.

In particular, without any further requirements on d, we can directly apply one-nearest neighbor regression (1-NN) via Equation 2.45. Using the radial basis function from Equation 2.44, we can also apply kernel regression (KR) via Equation 2.46. What remains more challenging is the application of Gaussian process regression (GPR). In particular, GPR requires a kernel k over the input space, in this case $\mathbb{R}^{m+n}$. There are multiple methods to construct such a kernel. First, we can construct the pseudo-Euclidean embedding explicitly via Theorem 2.2 and then define a standard vectorial kernel. Due to an eigendecomposition, this approach either requires cubic complexity, which may be infeasible for large data sets, or a Nyström-approximation, which distorts the distances. Second, we can apply a transformation to the distance values $d(\bar{x}_i, \bar{x}_j)$, which yields a kernel. For Euclidean distances, this is straightforward. For example, the radial basis function $k_{d,\tilde{\xi}}$ in Equation 2.44 is a kernel for any Euclidean distance d. However, not all pseudo-Euclidean distances yield a kernel under such transformations (Jäkel, Schölkopf, and Wichmann 2008). Finally, we can combine the latter approach with eigenvalue correction, which is what we propose. We first apply a radial basis function transformation, yielding a matrix of similarities $\boldsymbol{K} \in \mathbb{R}^{M \times M}$ with $\boldsymbol{K}_{i,j} = k_{d,\tilde{\xi}}(\bar{x}_i, \bar{x}_j)$. Next, we add the noise variance $\tilde{\sigma}^2$ of GPR to the diagonal (refer to Equation 2.50), which is equivalent to a shift eigenvalue correction and potentially reduces the number of negative eigenvalues. Now, observe that we need to invert the resulting matrix $\boldsymbol{K} + \tilde{\sigma}^2 \cdot \boldsymbol{I}^M$ anyways to yield a prediction. Therefore, we can perform eigenvalue correction of this matrix without extra cost as follows. We first compute the eigenvalue decomposition $\boldsymbol{V}^\top \cdot \boldsymbol{\Lambda} \cdot \boldsymbol{V} = \boldsymbol{K} + \tilde{\sigma}^2 \cdot \boldsymbol{I}^M$, then apply eigenvalue correction to the diagonal matrix of eigenvalues $\boldsymbol{\Lambda}$, for example by clipping negative values to $\tilde{\sigma}^2$, taking the absolute value, or subtracting the smallest negative value minus $\tilde{\sigma}^2$, which is equivalent to increasing the noise variance. In either case, we obtain a diagonal matrix $\tilde{\boldsymbol{\Lambda}}$ with strictly positive entries on the diagonal such that the matrix is invertible. Finally, we compute the inverted matrix as $\boldsymbol{V}^\top \cdot \tilde{\boldsymbol{\Lambda}}^{-1} \cdot \boldsymbol{V}$. Note that this mechanism also profits from the speedup techniques of the rBCM, because we can apply the eigenvalue correction to each cluster separately.

Using our approach until now, we can perform time series prediction on any data for which a pseudo-Euclidean distance is available. However, our predictive result is only some vector in the pseudo-Euclidean embedding space, which may not be useful for downstream tasks. Indeed, inferring the original time series $\bar{x}$ that corresponds to the predictive result $\phi(\bar{x})$ is generally impossible, because the spatial map ϕ may not be invertible and even approximate solutions are challenging to find, especially for structured data (Bakır, Weston, and Schölkopf 2003; Bakır, Zien, and Tsuda 2004; Kwok and I. W.-H. Tsang 2004).

Fortunately, our established theory of pseudo-Euclidean distances permits further inferences even without an explicit representation. In particular, we can infer the distances or kernel values to our predicted point, enabling us to use distance- or kernel-based methods downstream.

Inferring Distances from Predictive Results

Our aim is to use Theorem 2.3 in order to infer the distance between a predictive result and any other time series in $\mathcal{X}^*$. Recall that Theorem 2.3 implies that the distance between any two points in a pseudo-Euclidean space that can be described as affine combinations can be computed solely based on the original distance values. To use this result, we need to prove that the output of our time series prediction scheme is always an affine combination.

Theorem 5.1 (Predictive results as affine combinations). *Let $\mathcal{X}$ be some set and let $\{\mathcal{G}_t^j\}_{t=1,\ldots,T_j}^{j=1,\ldots,N} \subset \mathcal{X}$ be a dataset of sequences over that set, let $M = T_1 + \ldots + T_N$, let (j,t) be the ith tuple in $\{(j,t)|j \in \{1,\ldots,N\}, t \in \{1,\ldots,T_j - 1\}\}$ according to lexicographic ordering, let $\bar{x}_i := \mathcal{G}_1^j, \ldots, \mathcal{G}_t^j$, and let $\bar{y}_i := \mathcal{G}_1^j, \ldots, \mathcal{G}_{t+1}^j$.*

Further, let d be a pseudo-Euclidean distance over $\mathcal{X}^*$ with positive spatial mapping $\phi^+ : \mathcal{X}^* \to \mathbb{R}^m$ and negative spatial mapping $\phi^- : \mathcal{X}^* \to \mathbb{R}^n$, and let

$$\phi(\bar{x}) := \begin{pmatrix} \phi^+(\bar{x}) \\ \phi^-(\bar{x}) \end{pmatrix} \qquad \text{and} \qquad \boldsymbol{X} := \big(\phi(\bar{x}_1), \ldots, \phi(\bar{x}_M)\big) \in \mathbb{R}^{(m+n)\times M}$$

Finally, let k be a kernel over $\mathbb{R}^{m+n}$, and let $\bar{x} \in \mathcal{X}^*$. Then, it holds:

1. *The predictive result of 1-NN according to Equation 2.45 has the form $f(\phi(\bar{x})) = \boldsymbol{X} \cdot \vec{\alpha}$ with $\vec{\alpha}$ having exactly one entry 1 and only zero entries otherwise.*
2. *For any non-negative similarity s_d, the predictive result of KR according to Equation 2.46 has the form $f(\phi(\bar{x})) = \boldsymbol{X} \cdot \vec{\alpha}$ where all α_i are non-negative and sum up to 1.*
3. *For the priors $\vec{\theta} = \phi(\bar{x})$ and $\theta_i = \phi(\bar{x}_i)$, the predictive result of GPR according to Equation 2.50 has the form $f(\phi(\bar{x})) = (\boldsymbol{X}, \phi(\bar{x})) \cdot \vec{\alpha}$, where $\alpha_{M+1} = 1$ and all other α_i add up to zero.*
4. *For the priors $\vec{\theta} = \phi(\bar{x})$ and $\theta_i = \phi(\bar{x}_i)$, the predictive result of rBCM according to Equation 2.53 has the form $f(\phi(\bar{x})) = (\boldsymbol{X}, \phi(\bar{x})) \cdot \vec{\alpha}$, where $\alpha_{M+1} = 1$ and all other α_i add up to zero.*

Proof. Refer to Appendix A.15. □

Based on the distance formula 2.9 we can now compute distance values to any other time series $\bar{x}$, without any need for constructing the pseudo-Euclidean embedding explicitly. Via these distance values, we make further distance- or kernel-based methods applicable, such as RGLVQ or MGLVQ for classification (Hammer, D. Hofmann, et al. 2014; Nebel, Hammer, et al. 2015) or relational neural gas for clustering (Hammer and Hasenfuss 2007). Therefore, we have achieved a full methodological pipeline for preprocessing, prediction and post-processing.

In case we use rBCM regression, we can summarize the pipeline as follows.

1. If we intend to use a distance measure, we start off by computing the matrix of pairwise distances $\boldsymbol{D}$ with $\boldsymbol{D}_{i,j} = d(\bar{x}_i, \bar{x}_j)$ on our training data. If required we symmetrize this matrix by setting $\boldsymbol{D} \leftarrow \frac{1}{2} \cdot (\boldsymbol{D} + \boldsymbol{D}^\top)$ and set the diagonal to zero. Implicitly, this step embeds our training data in a pseudo-Euclidean space where $\boldsymbol{D}$ are pairwise pseudo-Euclidean distances. We transform this matrix into a similarity matrix $\boldsymbol{K}$ using, for example, the radial basis function transformation.

   If we intend to use a kernel measure, we start off by computing the kernel matrix $\boldsymbol{K}$ with $\boldsymbol{K}_{i,j} = k(\bar{x}_i, \bar{x}_j)$ on our training data.
2. We cluster the input data either based on the distance matrix $\boldsymbol{D}$ e.g. via relational neural gas (Hammer and Hasenfuss 2007), or based on the kernel matrix $\boldsymbol{K}$, e.g. via kernel k-means, kernel self-organizing map, or kernel neural gas (Filippone et al. 2008).
3. For each cluster c we perform an eigenvalue correction and inversion of the matrix $\boldsymbol{K}_c + \tilde{\sigma}^2 \cdot \boldsymbol{I}^{M_c}$ involving only the data in the cth cluster.

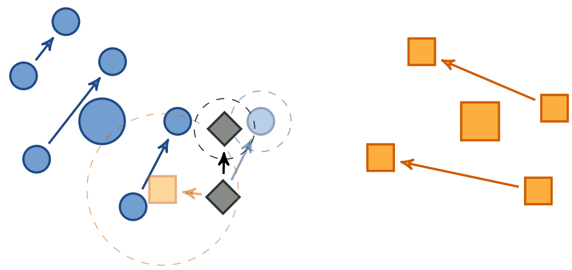


Figure 5.2: An illustration of the predictive pipeline. The data is first distributed into clusters via relational neural gas (blue circles and orange squares). Each cluster performs an independent prediction (half transparent blue circle and orange square) for the new data point (black diamond). Finally, these predictions are merged to a final prediction (black diamond) via the robust Bayesian committee machine (rBCM).

4. For any test time series $\bar{x}$ we compute the vector of distances $d(\bar{x}, \bar{x}_i)$ or kernel values $k(\bar{x}, \bar{x}_i)$ to the training data. In case of distances, we need to transform these distances to similarities and need to extend the eigenvalue correction of the training data to these new values via out-of-sample extension as described by Gisbrecht and Schleif (2015).
5. We perform rBCM to infer a prediction $f(\phi(\bar{x}))$ in form of an affine coefficient vector $\vec{\alpha}$.
6. We extend our distance matrix or kernel matrix using Theorem 2.3 to the predicted point.
7. We apply downstream distance- or kernel-based methods on the predicted point as desired.

The pipeline is illustrated in figure 5.2, where data points are shown as small shapes and points within the same time series are connected via arrows. First, we cluster the data via relational neural gas, which places prototypes (large circle and square) into the data (small circles and squares) and thereby partitions data points into disjoint clusters (distinguished by shape). For each cluster, we train a separate GPR model. For a test data point (diamond shape), each of the GPs provides a separate predictive Gaussian distribution, which are given in terms of their means (half-transparent circle and square) and their variance (dashed, half-transparent circles). The predictive distributions are merged to an overall predictive distribution with the mean from Equation 2.53 (solid diamond shape) and the variance from Equation 2.52 (dashed circle). Note that the overall predictive distribution is more similar to the prediction of the circle-cluster because the test data point is closer to this cluster and thus the predictive variance for the circle-cluster is lower, giving it a higher weight in the merge process.

This concludes our description of the predictive pipeline. We now go on to evaluate our pipeline experimentally.

5.3 EXPERIMENTS

In our experimental evaluation, we apply the pipeline introduced in the previous section to four datasets, two theoretical models and two Java program datasets. In all cases, we evaluate the root mean square error (RMSE) of the prediction for each method in a leave-one-out-crossvalidation over the time series in our dataset. We apply a Markov assumption, thus only considering the end points for all time series. More specifically, we denote the current test time series as $x'_1, \ldots, x'_T$, the training series as $\{x^j_1, \ldots, x^j_{T_j}\}_{j=1,\ldots,N}$, the predicted affine coefficients for point $x'_{t'}$ as $\vec{\alpha}_{t'} = (\alpha^1_{t',1}, \ldots, \alpha^N_{t',T_N}, \alpha'_{t'})$ and the matrix of squared pairwise distances (including the test data points) as $\boldsymbol{D}^2$. Accordingly, the RMSE for each fold has the following form (resulting from Theorem 2.3).

$$E = \sqrt{\frac{1}{T-1}\sum_{t'=1}^{T-1}\sum_{j=1}^{N}\sum_{t=1}^{T_j}\alpha^j_{t',t}d(x^j_t, x'_{t'+1})^2 + \alpha'_{t'}d(x'_{t'}, x'_{t'+1})^2 - \frac{1}{2}\vec{\alpha}_{t'}^\top \boldsymbol{D}^2\vec{\alpha}_{t'}} \tag{5.2}$$

We evaluate our four regression models, namely one-nearest neighbor regression (1-NN), kernel regression (KR), Gaussian process regression (GPR) and the robust Bayesian committee machine (rBCM), as well as the identity function as baseline, i.e. we predict the current point as next point.

We optimized the hyper parameters for all methods using a random search with 10 random trials (Bergstra and Bengio 2012). In particular, given the average distance $\bar{d}$ in the training data, we drew the radial basis function bandwidth ξ from a uniform distribution in the range $[0.05 \cdot \bar{d}, \bar{d}]$ for the theoretical datasets and fixed it to $0.3 \cdot \bar{d}$ for the Java datasets to avoid the need for a new eigenvalue correction in each random trial. We drew $\tilde{\sigma}$ from an exponential distribution in the range $[10^{-3} \cdot \bar{d}, \bar{d}]$ for the theoretical and $[10^{-2} \cdot \bar{d}, \bar{d}]$ for the Java datasets. We fixed the prior standard deviation $\sigma_{\text{prior}} = \bar{d}$ for all datasets. In each trial of the random search, we evaluated the RMSE in a nested leave-one-out-crossvalidation over the training time series and chose the hyper-parameters that corresponded to the lowest RMSE.

For rBCM we preprocessed the data via relational neural gas clustering with $\lfloor \frac{M}{100} \rfloor$ clusters for all datasets. As this pre-processing could be applied before hyper-parameter selection, the runtime overhead of clustering was negligible and we did not need to rely on the linear-time speedup described above but could compute the clustering on the whole training dataset.

Our experimental hypotheses are that all prediction methods should yield lower RMSE compared to the identity baseline (H1), that rBCM should outperform 1-NN and KR (H2) and that rBCM should not be significantly worse compared to GPR (H3). To evaluate significance we use a Wilcoxon signed-rank test.

Theoretical Data Sets

We investigate the following theoretical datasets:

Barabási-Albert model: A simple stochastic model of graph growth in undirected graphs (Barabási and Albert 1999). The growth process starts with a fully connected initial graph of m_0 nodes and adds $m - m_0$ nodes one by one. Each newly added node is connected to k of the existing nodes. The existing nodes are randomly selected with the probability $P(u) = \deg_t(u)/(\sum_v \deg_t(v))$ where $\deg_t$ is the node degree at time t, i.e. $\deg_t(v) = \sum_u \rho((u,v),t)$. We generated time series data using this model by treating the graph after every newly generated node as a new entry of the time series. In particular,

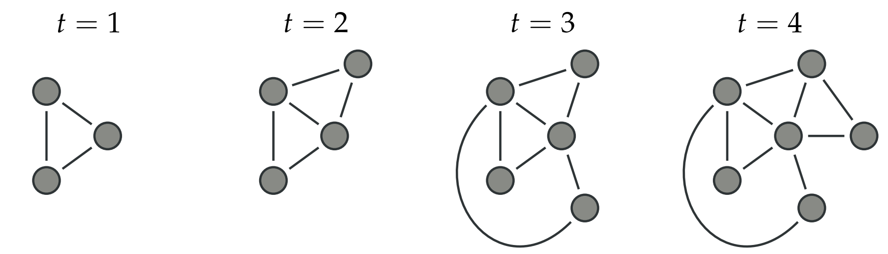


Figure 5.3: An excerpt of a time series resulting from the Barabási-Albert model. From left to right, the model starts with a fully connected graph with $m_0 = 3$ nodes and then grows, one node at a time, where each new node is connected with $k = 2$ new edges to the existing nodes. New edges preferentially attach to nodes with a high degree.

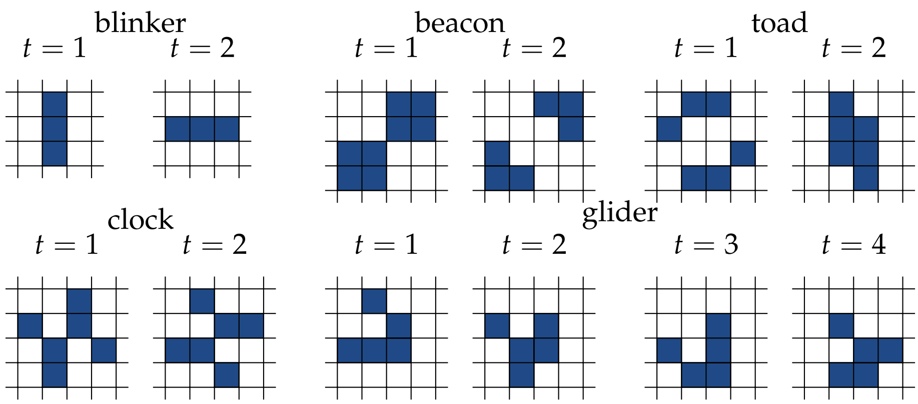


Figure 5.4: The standard patterns used for the *Game of Life*-dataset, except for the *block and glider* pattern. All unique states of the patterns are shown. Note that the state of glider at $t = 3$ equals the state at $t = 1$ up to rotation.

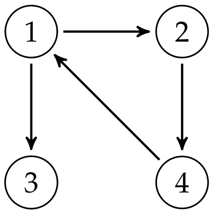
we generated 20 time series, each starting with a fully connected graph with $m_0 = 3$ nodes and then growing, one node at a time, to a total of $m = 27$ nodes with $k = 2$ new edges per node. This resulted in 500 graphs overall. Also refer to Figure 5.3 for an illustration of the growth process.

Conway's *Game of Life*: John Conway's *Game of Life* (Gardner 1970) is a simple, 2-dimensional cellular automaton model. Nodes are ordered in a regular, 2-dimensional grid and connected to their eight neighbors in the grid. Let $N(x)$ denote this eight-neighborhood in the grid. Then, we can describe Conway's *Game of Life* with the following sequential dynamical system for the node presence function ψ and the edge presence function ρ respectively:

$$\psi(v,t) = \begin{cases} 1 & \text{if } 5 \leq \psi(v,t-1) + 2 \cdot \sum_{u \in N(v)} \psi(u,t-1) \leq 7 \\ 0 & \text{otherwise} \end{cases} \tag{5.3}$$

$$\rho((u,v),t) = \begin{cases} 1 & \text{if } \psi(u,t) = 1 \wedge \psi(v,t) = 1 \\ 0 & \text{otherwise} \end{cases} \tag{5.4}$$

graph



shortest path length matrix

$u \setminus v$	1	2	3	4
1	0	1	1	2
2	2	0	3	1
3	∞	∞	0	∞
4	1	2	2	0

features

length	count
∞	3
1	4
2	4
3	1

Figure 5.5: An example graph, the associated matrix of shortest path lengths as returned by the Floyd- Warshall algorithm (Floyd 1962) and the histogram over path lengths used as feature representation for our approach. Note that self-distances are ignored.

Table 5.1: The mean RMSE and runtime across cross validation trials for both theoretical datasets (x-axis) and all methods (y-axis). The standard deviation is shown in brackets. Runtime entries with 0.000 had a shorter runtime (and standard deviation) than 10^{-3} milliseconds. The best (lowest) value in each column is highlighted by bold print.

	Barabási-Albert		Game of Life	
method	RMSE	runtime [ms]	RMSE	runtime [ms]
identity	0.137 (0.005)	**0.000** (0.000)	1.199 (0.455)	**0.000** (0.000)
1-NN	0.073 (0.034)	0.111 (0.017)	1.191 (0.442)	0.112 (0.025)
KR	0.095 (0.039)	0.122 (0.016)	0.986 (0.398)	0.120 (0.040)
GPR	0.064 (0.028)	0.148 (0.022)	**0.965** (0.442)	0.127 (0.026)
rBCM	**0.062** (0.015)	0.312 (0.083)	0.967 (0.461)	0.267 (0.077)

Note that Conway's *Game of Life* is Turing-complete and its evolution is, in general, unpredictable without computing every single step according to the rules (Adamatzky 2002). We created 30 time series by initializing a 20×20 grid with one of six standard patterns at a random position, namely *blinker*, *beacon*, *toad*, *block*, *glider*, and *block and glider* (see figure 5.4). The first four patterns are simple oscillators with a period of two, the glider is an infinitely moving structure with a period of two (up to rotation) and the *block and glider* is a chaotic structure which converges to a block of four and a glider after 105 steps[1]. We let the system run for $T = 10$ time steps resulting in 300 graphs overall. In every step, we further activated 5% of the cells at random, simulating observational noise.

As data representation for both theoretical datasets we use an explicit feature embedding inspired by the shortest-path-kernel of Borgwardt and Kriegel (2005). In particular, we compute the pairwise shortest paths between all nodes in the graph via the Floyd-Warshall algorithm (Floyd 1962) and then use the histogram over the lengths of these shortest paths as features. Figure 5.5 displays the feature computation for an example graph. We use the standard Euclidean distance on these features as our graph distance and normalize this distance by the average distance across the dataset. We obtained a kernel via the radial basis function transformation from Equation 2.44.

The RMSE and runtimes for the two theoretical datasets are shown in Table 5.1. As expected, KR, GPR and rBCM outperform the identity-baseline ($p < 10^{-3}$ for both

1 Also refer to the *Life Wiki* `http://conwaylife.com/wiki/` for more information on the patterns.

Table 5.2: The mean RMSE and runtime across cross validation trials for both Java datasets (x-axis) and all methods (y-axis). The standard deviation is shown in brackets. Runtime entries with 0.000 had a shorter runtime (and standard deviation) than 10^{-3} seconds. The best (lowest) value in each column is highlighted by bold print.

	MiniPalindrome		Sorting	
method	RMSE	runtime [s]	RMSE	runtime [s]
identity	0.295 (0.036)	**0.000** (0.000)	0.391 (0.029)	**0.000** (0.000)
1-NN	0.076 (0.047)	0.000 (0.000)	0.090 (0.042)	0.000 (0.000)
KR	0.115 (0.031)	1.308 (0.171)	0.112 (0.027)	1.979 (0.231)
GPR	0.075 (0.064)	111.417 (0.304)	0.020 (0.034)	114.394 (0.301)
rBCM	**0.044** (0.052)	11.698 (0.085)	**0.010** (0.025)	18.5709 (0.121)

datasets), supporting H1. 1-NN outperforms the baseline only in the Barabási-Albert dataset ($p < 10^{-3}$). Also, our results lend support to H2 as rBCM outperforms 1-NN in both datasets ($p < 0.05$ for Barabási-Albert, and $p < 0.01$ for Conway's *Game of Life*). However, rBCM is significantly better than KR only for the Barabási-Albert dataset ($p < 0.001$), indicating that for simple datasets such as our theoretical ones, KR might already provide sufficient predictive quality. Finally, we do not observe a significant difference between rBCM and GPR, as expected in H3. Interestingly, for these datasets, rBCM is slower compared to GP, which is explained by the overhead for maintaining multiple models.

Java Programs

Our two real-world Java datasets are *MiniPalindrome* and *Sorting* from Section 4.2. The motivation for time series prediction on such data is to help students achieve a correct solution in an intelligent tutoring system (ITS). In such an ITS, students incrementally work on their program until they might get stuck and do not know how to proceed. Then, we would like to predict the most likely next state of their program, given the time series of other students who have already correctly solved the problem; a setting that we will investigate in more detail in Chapter 6.

Note that our datasets only contain final, working versions of the programs. We simulated the graph growth as follows. First, we represented the programs as abstract syntax trees and then recursively removed the last node that opened a new scope in the Java program, until the abstract syntax tree was entirely deleted. Reversing this deletion process results in time series of a growing program. In particular, we thus obtained 834 syntax trees for the MiniPalindrome and 800 trees for the Sorting dataset respectively. As a distance, we employed the learned affine edit distance from Section 3.2 and obtained a kernel via the radial basis function in Equation 2.44 and clip eigenvalue correction as described in the previous sections.

We show the RMSEs and runtimes for both Java datasets in Table 5.2. In line with H1, 1-NN, KR, GPR, and rBCM all outperform the identity baseline ($p < 0.01$ in all cases). Further, rBCM outperforms both 1-NN and KR ($p < 0.01$ in all cases), which supports H2. Interestingly, rBCM apparently achieves better results compared to GPR, which might be the case due to additional smoothing provided by the averaging operation over all cluster-wise GPR results. This result supports H3. Finally, we observe that rBCM is about 10 times faster compared to GPR on these data.

5.4 DISCUSSION AND CONCLUSION

We have developed a novel pipeline to perform time series prediction for structured data, given either a distance measure or a kernel. Our results indicate that this pipeline is indeed able to capture information about the time series structure, significantly outperforming the identity baseline. We further showed that, for programming data, sophisticated predictive models such as the robust Bayesian committee machine (rBCM) can outperform simpler models such as one-nearest neighbor regression and kernel regression. Finally, we showed that the rBCM performs comparably to Gaussian process regression and is considerably faster for larger datasets.

The key idea to our approach is to perform the time series prediction not on the original structured data, but on an implicit vectorial representation in a pseudo-Euclidean space. A limitation of our approach is that this resulting point can generally not be interpreted as a structured datum. In the next chapter, we address this limitation and show how the prediction in the implicit pseudo-Euclidean space can be utilized to generate interpretable hints for students in intelligent tutoring systems.

6 APPLICATION TO INTELLIGENT TUTORING SYSTEMS

Summary: A challenge in learning complex skills such as computer programming lies in applying learned knowledge in practical exercises. For example, students may fail to write an entire program from scratch and get stuck along the way. In such situations, individualized next-step hints could support students and enhance their learning. Unfortunately, providing such hints in large courses or for large state spaces goes far beyond the capabilities of human instructors or rule-based systems.

In this chapter, we summarize existing work on automated generation of individualized next-step hints in light of the edit distance theory established in Section 2.3. Further, we extend the predictive pipeline of Chapter 5 to achieve a novel automatized mechanism that can predict what successful past students' would have done and that uses this prediction to generate hints; a mechanism that we call the Continuous Hint Factory (CHF).

In an experimental evaluation on two real-world tutoring datasets, we demonstrate that our pipeline outperforms previous approaches in terms of predictive accuracy and performs comparably in terms of the pedagogic quality of the generated hints.

Publications: This chapter is based on the following publications.

- Paaßen, Benjamin, Barbara Hammer, et al. (2018). "The Continuous Hint Factory - Providing Hints in Vast and Sparsely Populated Edit Distance Spaces". In: *Journal of Educational Datamining* 10.1, pp. 1–35. URL: https://jedm.educationaldatamining.org/index.php/JEDM/article/view/158.

Many learning tasks require more than a single step to solve. For example, programming tasks require a student to iteratively write, test, and refine code that accomplishes a given objective (Gross, Mokbel, et al. 2014; Price, Dong, and Lipovac 2017; Rivers and Koedinger 2015). When working on such multi-step-tasks, students start with an initial state and then apply actions to change their state (such as inserting or deleting a piece of code) in order to get closer to a correct solution. At some point, a student may not know how to proceed or may be unable to find an error in her current state, in which case external help is required. In particular, such a student may benefit from a next-step hint, guiding her a little closer toward a correct solution and helping her to continue on her own (Aleven, Roll, et al. 2016). Many intelligent tutoring systems attempt to create such next-step hints automatically, and adjust such hints to the student's current state as well as her underlying strategy (Van Lehn 2006). Typically, hints are generated by an expert-crafted, rule-based model (N.-T. Le 2016). However, designing such expert models becomes infeasible if the space of possible states is too large to cover with expert rules (Murray, Blessing, and Ainsworth 2003; Koedinger et al. 2013; Rivers and Koedinger 2015). For example, the space of possible computer programs grows exponentially with the program length and the set of programs that perform the same function is infinite (Piech, Sahami, et al. 2015). Other examples are so-called ill-defined domains where explicit domain knowledge is not available or at least very hard to formalize (Lynch et al. 2009).

Several approaches have emerged which provide next-step hints without an expert model. Typically, these approaches provide hints in the form of edits, that is, actions that can be applied to the student's current state to change it into a more correct and/or more complete state, based on the edits that successful students have applied in the past (Gross and Pinkwart 2015; Price, Dong, and Barnes 2016; Rivers and Koedinger 2015; Zimmerman and Rupakheti 2015). The most basic version of this approach requires only two ingredients: an edit distance and at least one correct solution for the task. If a student issues a help request, the system can simply compute the cheapest edit script $\bar{\delta}$ which transforms the student's current state to the closest correct solution and use the first edit in that edit script as a hint (Rivers and Koedinger 2015; Zimmerman and Rupakheti 2015). Note that only the correct solutions need to be task-specific, whereas the same edit distance can be applied across tasks or even across domains (Mokbel, Gross, et al. 2013). Furthermore, we can adjust an edit distance to a task by adapting it to student data via metric learning as suggested in Chapters 3 and 4. Finally, the approach achieves fine grained and personalized feedback, because the hint is based on the student's current, individual state and thus fits to her specific solution strategy and style (N.-T. Le and Pinkwart 2014).

A problem with this basic hint generation mechanism is that the generated hints may still be counter-intuitive to a human programmer because the cheapest edit script towards a correct solution does not necessarily traverse the most intuitive states. Most existing approaches address this problem by constraining the generated hints to states that have been visited often by past students (Barnes and J. Stamper 2008; Lazar and Bratko 2014; Rivers and Koedinger 2014; Piech, Sahami, et al. 2015). Unfortunately, for many programming tasks, the space of possible programs is so large that hardly any state is visited more than once, even if aggressive pre-processing methods are applied to canonicalize program representations (Price and Barnes 2015).

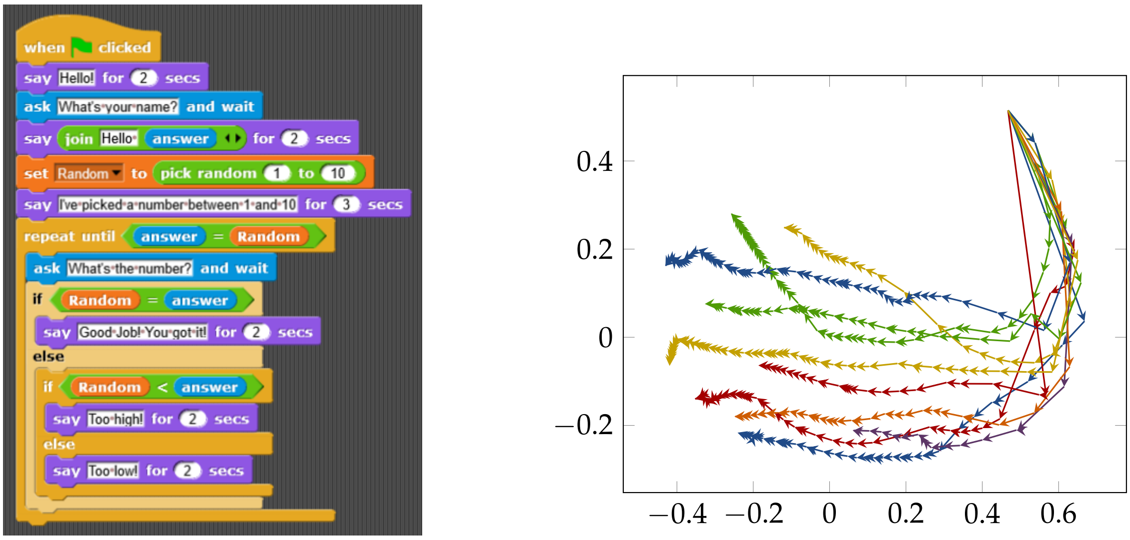
Therefore, a novel approach is needed that can select intuitive edits even in cases where frequency information is not available. We base this approach on the *Hint Factory*, which generates hints that have led past students in the same situation to a correct solution (Barnes and J. Stamper 2008; J. C. Stamper et al. 2012). To transfer this approach to vast and sparsely populated spaces, we consider not only the data of past students who have visited the same state, but also *similar* states as quantified by an edit distance, and we predict the ideal next state via our predictive pipeline from Chapter 5. Because the prediction occurs in a latent, continuous space, we call our approach the Continuous Hint Factory (CHF).

In more detail, the key contributions of this chapter are as follows. First, we provide precise definitions of key concepts in the field of edit-based hint policies and integrate them into the mathematical framework of this thesis. Second, we apply the predictive pipeline from Chapter 5 to predict student behavior. Finally, we provide a method to translate a prediction generated by our predictive pipeline into human-readable edits.

In experiments on two real-world datasets we provide evidence that the CHF is able to predict what capable students would do in solving a learning task, that the CHF is able to disambiguate between many possible edits, and that the hints provided by the CHF match the hints of human tutors at least as well as other established hint techniques.

6.1 AN INTEGRATED VIEW OF EDIT-BASED HINT POLICIES

In this section, we review existing approaches to edit-based hint policies. We guide this review by formal definitions of key concepts in the hint policy literature, which we can



(a) (b)

Figure 6.1: (a) A screenshot from the Snap programming environment. (b) A 2D embedding of ten example traces in the Snap dataset. The 2D embedding was obtained via non-metric multi-dimensional scaling (Sammon 1969) using the pairwise edit distances as input. Colors are used to distinguish between different traces. States within one trace are connected by arrows.

connect to the theory of edit distances as established in Section 2.3. This connection will also motivate the application of the predictive pipeline developed in Chapter 5.

To illustrate our scenario of interest, consider the task of programming a guessing game. The program should first ask the player for their name, then generate a random number between 1 and 10, and finally let the player guess the number, providing feedback to the player regarding whether the number was too low, too high, or correct. A correct solution for this task in the *Snap* programming language[1] is shown in Figure 6.1(a). In a tutoring system involving this task, a student would start off with an empty program and then would add blocks to the program, delete blocks, or replace blocks with other blocks until the student obtains a correct solution or gets stuck. In the latter case, the student may hit a "help" button which would trigger the system to provide a hint in the form of an edit which leads the student closer to a correct solution (e.g., to add an "ask" block to ask for the player's name in the beginning).

From a pedagogical point of view, it may be suboptimal to immediately tell the student which edit to apply. After all, we deprive students of the possibility of finding the correct next step themselves and do not require the students to reflect on underlying concepts, as suggested by Fleming and Levie (1993) as well as N.-T. Le (2016). Indeed, Aleven, Roll, et al. (2016) suggest displaying hints that reveal the solution only as a last resort ("bottom-out hints") after exhausting options for more principle-based hints. This begs the question why we focus here on such bottom-out hints.

First, edit hints are different from other bottom-out hints in that they display only a very small part of the solution, namely a single edit such that students still need finish most of the problem themselves. Second, bottom-out hints may facilitate learning if

1 http://snap.berkeley.edu

students reflect on the hint and engage in sense-making behavior (Aleven, Roll, et al. 2016; Shih, Koedinger, and Scheines 2008). Third, many students skip through the principle-based hints anyway to reach the bottom-out hint, indicating that they regard such hints as more useful (Aleven, Roll, et al. 2016; Shih, Koedinger, and Scheines 2008). Fourth, we point to a study by Price, Zhi, and Barnes (2017b), which indicates that edit hints are judged as relevant and interpretable by human tutors. Finally, and most importantly, we argue that more elaborate hint strategies are simply not available in many important learning tasks because they require expert-crafted hint messages which are difficult to apply at scale (N.-T. Le and Pinkwart 2014; Murray, Blessing, and Ainsworth 2003; Rivers and Koedinger 2015).

In particular, there have been some approaches to make expert-crafted hints available in larger state spaces, for example authoring tools for tutoring systems, which aim at reducing the expert work required for designing feedback. A prime example are the Cognitive Tutor Authoring Tools (CTAT), which support the construction of cognitive tutors (Aleven, McLaren, et al. 2006). Cognitive tutors can be seen as a gold standard of intelligent tutoring systems because their effectiveness has been established in classroom studies, and they have been successfully applied in classrooms across the US (Koedinger et al. 2013; Pane et al. 2014). However, even with authoring tools, covering all possible variations in a sufficiently variable state space with many viable solutions may be infeasible (N.-T. Le and Pinkwart 2014; Murray, Blessing, and Ainsworth 2003; Rivers and Koedinger 2015). For example, in our programming dataset (see Figure 6.1(a)), we consider more than 40 different solution strategies, each of which involves more than 40 steps.

Another approach is "force multiplication", which assumes that a relatively small number of expert-crafted hint messages are available, which are then applied to new situations automatically, thereby "multiplying the force" of expert work (Piech, Jonathan Huang, et al. 2015). Examples include the work of Choudhury, Yin, and Fox (2016), Head et al. (2017), as well as Yin, Moghadam, and Fox (2015) who apply clustering methods to aggregate many different states and then provide the same hint to all states in the same cluster. Another example is the work of Piech, Jonathan Huang, et al. (2015) who annotate each possible expert hint with a set of example states for which this hint makes sense and a set of example states for which this hint does *not* make sense. Then, they train a classifier for each hint that can decide for any new state whether the hint should be displayed or not. Finally, Marin et al. (2017) annotate expert-crafted hints with small snippets of Java code for which the given hint makes sense and then display the hint whenever the respective snippet is discovered in a student's state. Note that these approaches are limited by the number of hints that are provided by the teaching experts. If experts did not foresee a situation that requires specific help, the system can not provide help in that situation. Moreover, these approaches are limited in resolution as experts can hardly be expected to devise specific recommended edits for any conceivable student state. As such, we regard force multiplication as a complementary approach to edit-based hints, with the former being coarse-grained and principled, and the latter being fine-grained and concrete.

In the remainder of this section, we will analyze edit-based next-step hint approaches in more detail. We start our investigation by defining the state space, edits on that space, traces through the state space, a generalized notion of edit distance on the state space, and hint policies. Using these definitions, we provide an overview of hint policies in the literature and compare them in light of our mathematical framework.

Edit Distances and Legal Move Graphs

Recall that we wish to support students in solving a multi-step learning task by providing on-demand edit hints. More precisely, we assume the following scenario. A student starts in some initial state provided by the system, and then successively edits this initial state until she finishes the task or gets stuck and asks the system for help. In the latter case, we wish to generate an edit hint for the student, meaning a change that she can apply to her current state in order to proceed toward a correct solution and hopefully continue on her own. To define edits, we generalize the notion of sequence edits (refer to Definition 2.5) and tree edits (refer to Definition 2.12) as follows.

Definition 6.1 (Edits, Edit Sets, Scripts)**.** Let X be some set, for example the *state space* of a learning task. We define an *edit* on X as a function $\delta : X \to X$. We call a set Δ of edits on X an edit set on X. We call an edit set *symmetric* if for all all edits $\delta \in \Delta$ and all states $x \in X$ there exists an edit $\delta^{-1} \in \Delta$ such that $\delta^{-1}(\delta(x)) = x$. We call δ^{-1} an *inverse* edit for δ on x.

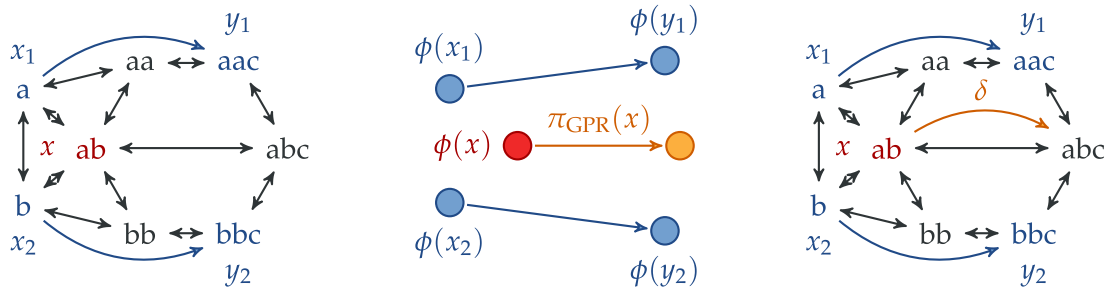
We define an *edit script* over Δ as a finite list of elements $\bar{\delta} = \delta_1 \ldots \delta_T$ from Δ. We denote the set of all possible edit scripts over edit set as Δ^*. We define the application $\bar{\delta}(x)$ of an edit script $\bar{\delta} = \delta_1 \ldots \delta_T$ to a state x as the function composition $\delta_T \circ \ldots \circ \delta_1(x)$, where $\delta \circ \delta'(x) := \delta(\delta'(x))$. If $\bar{\delta} = \epsilon$, we define $\bar{\delta}(x) = x$.

The notion of an edit set should cover all actions that a student can perform to change their current state to a different state. Recall our example of the guessing game programming task in Figure 6.1(a). In this scenario, the set of possible states is the set of possible Snap programs. The edit set includes adding a single block at any point in the program, replacing a block with another one, and deleting a block. For example, we may delete the "say 'Hello!' for 2 secs" block in Figure 6.1(a) or replace it with a "say 'Hello!' for 1 sec"-block. Note that this edit set is *symmetric*, in the sense that we can reverse every edit we have applied by deleting an inserted block, re-inserting a deleted block, or replacing a replaced block with its prior version. This is a desirable property for edit sets because it ensures that we can reach a correct solution from any state by reversing erroneous actions and then continueing towards the correct solution. We can make this notion of reachability precise by introducing the notions of legal move graphs, traces, interaction networks, and solution spaces following the work of Piech, Sahami, et al. (2015), Eagle, M. Johnson, and Barnes (2012), as well as Rivers and Koedinger (2014).

Definition 6.2 (Legal Move Graph, Trace, Solution Space)**.** Let X be a state set and Δ be an edit set on X. Then, the *legal move graph* according to X and Δ is defined as the directed graph $\mathcal{G}_{X,\Delta} = (X, E)$ where $E = \{(x, y) | \exists \delta \in \Delta : \delta(x) = y\}$.

Now, let $x, y \in X$. We define a *trace* between x and y as a sequence $x_0, \delta_1, \ldots, \delta_T, x_T$ with $x_0 = x$, $x_T = y$, and for all $t \in \{1, \ldots, T\} : x_t \in X$, $\delta_t \in \Delta$, and $\delta_t(x_{t-1}) = x_t$.[2] We call a state y *reachable* from x if a trace p from x to y exists.

2 Note that this definition is not exactly equivalent to the one given by Eagle, M. Johnson, and Barnes (2012), because they do not require actions to be *deterministic*. In their framework, the same action applied to the same state may lead to different subsequent states. For the sake of brevity, we refrain from this probabilistic extension here.



(a) The legal move graph according to the state set $X = \{\mathrm{a}, \mathrm{aa}, \mathrm{aac}, \mathrm{ab}, \mathrm{abc}, \mathrm{b}, \mathrm{bb}, \mathrm{bbc}\}$ and the edit set of the string edit distance. $x = \mathrm{ab}$ is the student's current state (red). Further, two traces with states $x_1 = \mathrm{a}, y_1 = \mathrm{aac}$, and $x_2 = \mathrm{b}, y_2 = \mathrm{bbc}$ respectively form an interaction network (blue).

(b) The embedding of the trace states (blue) and the student state (red) from the left into the edit distance space via the embedding ϕ. The recommendation of the Gaussian process regression (GPR) policy $\pi_{\mathrm{GPR}}(x)$ for the current student state x is shown in orange.

(c) The legal move graph from the left figure, including the edit δ (orange) which corresponds to the recommended edit of GPR from the center figure.

Figure 6.2: An illustration of the Continuous Hint Factory (CHF) on a simple dataset of strings. First, we compute pairwise edit distances between the student's current state (red) and trace data (blue). These edit distances correspond to the shortest paths in the legal move graph (left). The edit distances also correspond to a continuous embedding, which we call the edit distance space (center). In this space, we can infer an optimal edit (orange) using machine learning techniques, such as Gaussian process regression (GPR). Finally, we infer the corresponding hint in the original legal move graph (right), which can then be displayed to the student.

Now, let $\bar{X} = \{(x_0^j, \delta_1^j, \dots, \delta_{T_j}^j, x_{T_j}^j)\}_{j=1,\dots,N}$ be a set of traces. The *interaction network* corresponding to this set of traces is defined as the graph $\mathcal{G}_{\bar{X}} = (V, E)$ where

$$V = \left\{ x_t^j \middle| j \in \{0, \dots, N\}, t \in \{1, \dots, T_j\} \right\} \tag{6.1}$$

$$E = \left\{ (x_{t-1}^j, x_t^j) \middle| j \in \{1, \dots, N\}, t \in \{1, \dots, T_j\} \right\} \tag{6.2}$$

We also call V a *solution space*.

As an example, consider the set of strings $X = \{\mathrm{a}, \mathrm{aa}, \mathrm{aac}, \mathrm{ab}, \mathrm{abc}, \mathrm{b}, \mathrm{bb}, \mathrm{bbc}\}$ and the edit set $\Delta_{\mathrm{ALI}, \{\mathrm{a},\mathrm{b},\mathrm{c}\}}$ from Section 2.3.2. An excerpt of the legal move graph for this example is shown in Figure 6.2(a). In particular, "ab" is connected to "a", "aa", "b", "bb", and "abc" because we can delete b, replace b with a, delete a, replace a with b, and insert c to transform "ab" to the respective other strings. Note that all edges in this legal move graph are bi-directional, indicating the symmetry of the edit set.

Figure 6.2(a) also shows two traces in blue. These traces cover the strings "a", "aac", "b", and "bbc". Therefore, the interaction network for this case would only contain these four strings and the edges ("a", "aac") as well as ("b", "bbc"). Note that these traces use multiple edits at the same time and thus are defined over a different edit set compared to the original legal move graph - in particular the edit set is Δ^*. Such "jumps" in the legal move graph are typical if not every action of a user in the system can be recorded (Piech, Sahami, et al. 2015).

The basic suggestion of Piech, Sahami, et al. (2015) to construct a hint is the following. If a student gets stuck in state x, our hint should guide them to the first state x_1 on a trace $x_0, \delta_1, \ldots, \delta_T, x_T$ from x to the closest correct solution y in the legal move graph. Per default, we could consider the number of states T in a trace as its length. However, we can also generalize this notion by using the concept of a cost function as in Definition 2.6. This concept also yields a generalized version of the edit distance as given in Definitions 2.6 and 2.13

Definition 6.3 (Cost Function and Edit Distance)**.** Let X be a set and Δ be an edit set on X. A function $c : \Delta \times X \to \mathbb{R}^+$ is called a *cost function* on Δ. We call $c(\delta, x)$ the *cost* of applying edit δ to the state x.

We call a cost function *symmetric* if $c(\delta, x) = c(\delta^{-1}, \delta(x))$ for all states $x \in X$, all edits $\delta \in \Delta$, and at least one inverse edit δ^{-1} for δ on x.

We define the *cost* of an edit script $\bar{\delta} \in \Delta^*$ recursively as $c(\epsilon, x) = 0$ and $c(\delta_1 \ldots \delta_T, x) = c(\delta_1, x) + c(\delta_2 \ldots \delta_T, \delta_1(x))$. We define the *edit distance* according to Δ and c as follows.

$$
\begin{aligned}
d_{\Delta,c} &: X \times X \to \mathbb{R}^+ \\
d_{\Delta,c}(x, y) &:= \min_{\bar{\delta} \in \Delta^*} \left\{ c(\bar{\delta}, x) \middle| \bar{\delta}(x) = y \right\}
\end{aligned}
\tag{6.3}
$$

Let $\mathcal{G}_{X,\Delta} = (V, E)$ be the legal move graph according to X and Δ and let c be an edit cost function on Δ. We define the *length* or *cost* of a trace $p = x_0, \delta_1, \ldots, \delta_T, x_T$ in $\mathcal{G}_{X,\Delta}$ as $c(p) := c(\delta_1 \ldots \delta_T, x_0)$.

We call any trace p such that $c(p) = \min\{c(p)|p \text{ is a trace from } x \text{ to } y\}$ a *shortest trace* from x to y.

We can show that, searching for a shortest trace in the legal move graph is essentially equivalent to computing the edit distance. In particular, we obtain the following results.

Theorem 6.1. *Let X be a state set, let Δ be an edit set on X, and let c be a cost function over Δ.*

Then, the following statements hold for any $x, y \in X$ where y is reachable from x.

First, for each trace $p = x_0, \delta_1, \ldots, \delta_T, x_T$ from x to y, $\delta_1, \ldots, \delta_T$ is an edit script such that $\bar{\delta}(x) = y$ and $c(\bar{\delta}, x) = c(p)$.

Second, for each edit script $\bar{\delta} = \delta_1, \ldots, \delta_T$ such that $\bar{\delta}(x) = y$ there exists a trace $p = x_0, \delta_1, \ldots, \delta_T, x_T$ from x to y, such that $c(\bar{\delta}, x) = c(p)$.

Third, it holds:

$$
d_{\Delta,c} = \min\{c(p)|p \textit{ is a trace from } x \textit{ to } y\}. \tag{6.4}
$$

Proof. Let $x, y \in X$ such that y is reachable from x.

We prove all claims in turn. First, let $p = x_0, \delta_1 \ldots, \delta_T, x_T$ be a path from x to y. Then, per definition of a trace, for each $t \in \{1, \ldots, T\}$ it holds: $\delta_t(x_{t-1}) = x_t$. Therefore, $\delta_1, \ldots, \delta_T$ is an edit script such that $\bar{\delta}(x) = y$. $c(\bar{\delta}, x) = c(p)$ follows from the definition of $c(p)$.

Second, let $\bar{\delta} = \delta_1, \ldots, \delta_T$ be an edit script such that $\bar{\delta}(x) = y$. We can construct the corresponding trace recursively as $x_0 := x$ and $x_t := \delta_t(x_{t-1})$. Accordingly, $x_0, \delta_1, \ldots, \delta_T, x_T$ is a trace from x to y. Further, $c(\bar{\delta}, x) = c(p)$ follows from the definition of $c(p)$.

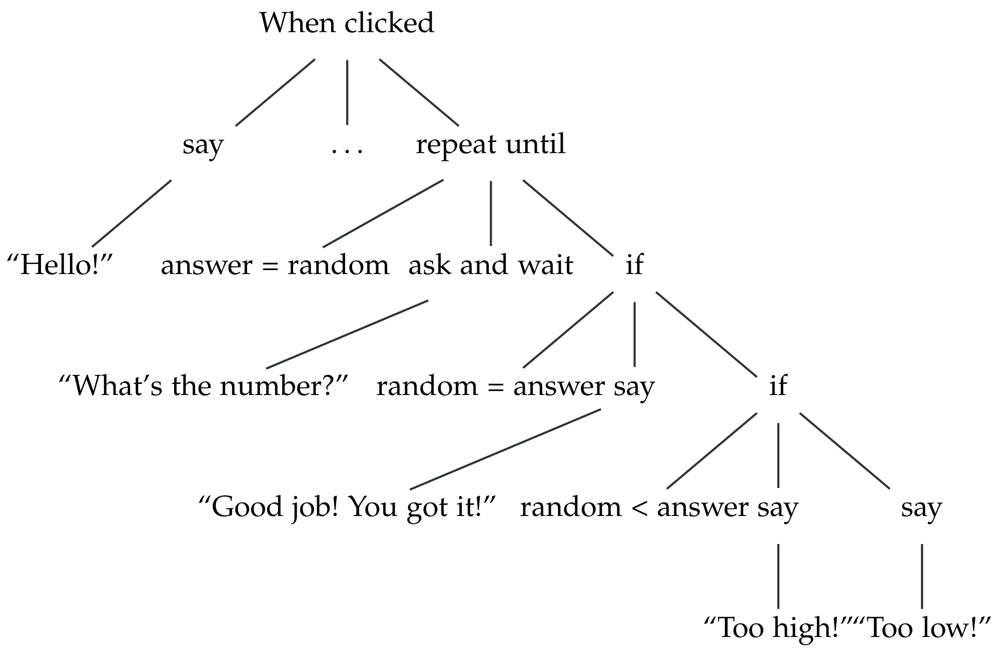


Figure 6.3: An abstract syntax tree, simplified for clarity, corresponding to the Snap program shown in Figure 6.1(a).

Finally, consider the third claim. If the claim would not hold, then either there exists a path from x to y that is shorter than the cost of the cheapest edit script, or there exists an edit script that is cheaper than the length of the shortest path. Due to the first two claims, neither case can occur. □

In other words, we can construct a hint mechanism by computing the cheapest edit script $\bar{\delta} = \delta_1, \ldots, \delta_T$ which transforms the student's current state x into the closest correct solution y and return δ_1 as hint. Because the cheapest edit script corresponds to a shortest trace, this leads the student toward the correct solution with the least amount of work. Unfortunately, not all edit distances permit the efficient computation of the cheapest edit script.

Consider the Snap example from Figure 6.1(a). In this domain, the order of many blocks in the program is insignificant to the function of the program. Therefore, one may wish to apply an edit distance that works on unordered trees. However, edit distances on such unordered trees are NP-hard (Zhang, Statman, and Shasha 1992), making them infeasible in practice. Therefore, we focus here on the subset of efficiently computable edit distances, namely the edit distances covered in Section 2.3.

For our scenario, the tree edit distance of Zhang and Shasha (1989) is particularly interesting, because many learning environments for computer programming have applied the tree edit distance to compare *abstract syntax trees* of computer programs (e.g. Choudhury, Yin, and Fox 2016; Freeman, Watson, and Denny 2016; Nguyen et al. 2014; Rivers and Koedinger 2015). An abstract syntax tree covers the syntactic structure of a computer program with syntactic building blocks as nodes. For example, the program shown in Figure 6.1(a) would correspond to the abstract syntax tree shown in Figure 6.3. Mokbel, Gross, et al. (2013) as well as Price, Zhi, and Barnes (2017a) have extended the tree edit distance to a two-stage approach where some special subtrees, such as functions in a program, may be arbitrarily re-ordered but all subtrees below these order-invariant nodes are still compared using a classic tree edit distance. In another approach, Zimmerman and Rupakheti (2015) have suggested to reduce the computational complexity of the tree edit distance by approximating it with the *pq*-gram-edit distance of Augsten, Böhlen, and Gamper (2008), which results in a considerably faster runtime of $\mathcal{O}(m \cdot \log(m))$.

Beyond computational complexity, a key challenge to edit distance is that it does not necessarily correspond to the *semantic* distance between states. Consider again the Snap example in Figure 6.1(a). Here, we could replace any of the strings in "say" or "ask" blocks without changing the basic computed function of the program. More generally, we can apply arbitrarily many edits to a given program without changing the computed function. Conversely, even small syntactic changes can result in severe semantic changes, for example if we would remove the "repeat until" block in the program. This mismatch between edit distance and semantic distance can negatively impact the utility of generated hints. In particular, edits may be recommended that get the student syntactically closer to a correct solution but may be semantically irrelevant or even confusing.

One approach to address this issue is *canonicalization*, which essentially transforms the raw states in a state space X to a canonic form such that semantically equivalent states have the same canonic form. The edit distance is then defined between canonic forms instead of raw states, yielding a much smaller legal move graph and edits that put stronger emphasis on semantically relevant changes. Canonicalization is particularly common for computer programs, where we can normalize variable names or the order of binary relations (such as $<$) and remove unreachable code (Rivers and Koedinger 2012). In all these cases, a canonicalization is a function from the state space to a subset of itself. However, more generally, one could define a canonicalization as any kind of mapping ϕ into an auxiliary space. For example, Paaßen, Jensen, and Hammer (2016) canonicalize computer programs by representing them in terms of their execution trace, to which they apply a string edit distance $\tilde{d}$, yielding the distance $d(x, y) = \tilde{d}(\phi(x), \phi(y))$ between any two states x and y.

A challenge in canonicalization lies in the fact that edits on the canonic form may not be directly applicable or interpretable for students. For example, students cannot easily adapt their program to directly influence the program's execution in the way indicated by an edit on the execution trace. To address this problem, Rivers and Koedinger (2015) suggest aligning the edits on the canonic form with the student's original state in a process called *state reification*. Another challenge lies in the fact that too drastic canonicalization may remove features of the original state for which feedback would be desirable. For example, tutoring systems for computer programming often not only intend to teach functionally correct programming but also programming style such that important stylistic differences, even though semantically irrelevant, need to be preserved in the canonic form (Piech, Jonathan Huang, et al. 2015; Choudhury, Yin, and Fox 2016). Furthermore, there can be in principle no canonicalization which uniquely identifies all relevant functions because this would solve the halting problem. As such, we propose to combine modest canonicalization with other adaptation approaches, especially metric learning, to achieve a semantic-aware distance measure on states. In our experiment, we normalize variable names, the order of variable declarations, and the order of binary relations for canonicalization purposes.

In summary, we have introduced the concepts of edits, legal move graphs, shortest paths, edit distances, and canonicalization. These concepts cover everything we need to know to provide a review of existing hint policies in the literature.

Hint policies

Formally, our goal is to devise a function π that can, for any state x students may visit, return an edit $\delta = \pi(x)$ they should apply next. Inspired by Piech, Sahami, et al. (2015), we call such a function a *hint policy*.[3]

Definition 6.4 (Hint Policy)**.** Let X be a state set and Δ be an edit set on X. A *hint policy* is a function $\pi : X \to \Delta$.

The arguably simplest policy is the one of Zimmerman and Rupakheti (2015), which always recommends the first edit δ_1 in a cheapest edit script $\delta_1, \ldots, \delta_T$ toward the closest correct solution. Such an approach does not even require student data, except for at least one example of a correct solution of the task. A drawback of the Zimmerman policy is that it can not disambiguate between multiple possible cheapest edit scripts and thus may recommend edits which do lead to the correct solution but are still counter-intuitive.

Rivers and Koedinger (2015) address this issue in their Intelligent Teaching Assistant for Programming (ITAP). Their technique involves the following steps: First, they apply canonicalization. Second, they retrieve the closest solution according to the tree edit distance on canonic forms. Third, they compute a shortest trace $p = x_0, \delta_1, \ldots, \delta_T, x_T$ from the student's state to the closest correct solution. Fourth, of the states $x_1, \ldots, x_T$, they select the one with the highest desirability score, where the desirability score is a weighted sum of the frequency in past student trace data, the edit distance to the student's state, the number of successful test cases the state passes, and the edit distance to the solution (Rivers and Koedinger 2015). Finally, they apply an inverse canonicalization (state reification) to infer edits that can be directly applied to the student's state to transform it to the selected state. This approach has been shown to provide helpful edits in almost all cases for a broad range of tasks (Rivers and Koedinger 2015). Note that the success of the Rivers policy hinges upon meaningful frequency information. If no or little frequency information is available, the hints provided by the Rivers policy may not be representative of generic steps toward a solution but rather of specificities of the reference solution that was selected.

Piech, Sahami, et al. (2015) have suggested a similar approach to the previous two by also recommending the first edit on a shortest trace towards the next correct solution, but assigning different costs to edits. In particular, they defined the cost of any edit connecting two states x and y as the inverse frequency of y in student data, such that the policy is more likely to recommend states that were visited often. In an evaluation on a large-scale dataset consisting of over a million states from the *Hour of Code* Massive Open Online Course (MOOC), Piech, Sahami, et al. (2015) found that this policy outperformed all other approaches, including the previous two. Note that this approach still relies on frequency information, which may not be available in sparsely populated spaces where almost no state is visited more than once.

3 Note that Piech, Sahami, et al. (2015) define a hint policy differently, namely as a function π' mapping a state to a state. Our definition is a proper generalization of this concept because we can always generate a Piech-style hint policy π' from a policy π in our sense by setting $\pi'(x) := \delta(x)$ where $\delta = \pi(x)$. The inverse conversion is *not* always possible because there may be multiple edits leading to the same state.

As an alternative to policies that approach the closest correct solution directly, Gross and Pinkwart (2015) suggest to guide students along the traces of successful past students. They first construct an interaction network of past students' trace data. When a student requests help, they retrieve the closest state x_t^j to the student's current state in the interaction network according to an edit distance. Then, they distinguish between two kinds of help-seeking behavior. If students are trying to find an error in their code, the system recommends an edit toward x_t^j, thereby attempting to correct the error. If students assume that their current state is correct, but they are looking for a next step, the system recommends an edit toward the *successor* x_{t+1}^j of x_t^j, thereby guiding the student closer to a solution (Gross and Pinkwart 2015). This policy can be seen as an instance of *case-based reasoning*, where recommendations are based on a similar case from an underlying case base. Similarly, Freeman, Watson, and Denny (2016) have taken this view to analyze Python programs and used a weighted tree edit distance to retrieve similar cases. Further, Gross, Mokbel, et al. (2014) proposed example-based feedback, in which the closest prototypical state in a dataset is retrieved and shown to the student to elicit self-reflection and sense-making in order to improve their own state. If the closest state in the case base is sufficiently similar to the student's state and corresponds to a capable student, such an approach can provide hints that emulate the actions of a capable "virtual twin" of the student. However, if only few reference solutions exist, the selected next state may still be fairly dissimilar and edits toward the next state may include not only error-correcting hints or next-step hints but also stylistic or strategic choices that do not correspond to the student's goals.

Lazar and Bratko (2014) propose yet a different approach by recommending edits that have been frequent in past student traces and increase unit test scores. As with the Piech policy, the policy of Lazar and Bratko (2014) critically relies on frequency information, albeit for edits instead of states, which may not always be available. Furthermore, edits that may be generally important for a task may not necessarily be helpful in a specific situation.

An alternative view is provided by the Hint Factory, which analyzes the question of choosing the optimal edit according to a Markov Decision Process (Barnes and J. Stamper 2008). In particular, the Hint Factory always returns the edit that maximizes the expected future reward, where a reward is given whenever a student has achieved a correct solution. Several studies have demonstrated that the Hint Factory reduces student dropout and helps students to complete more problems more efficiently in logic problem solving (J. C. Stamper et al. 2012; Eagle and Barnes 2013). The Hint Factory has also been applied to further domains, such as the serious game BOTS (Hicks, Peddycord, and Barnes 2014) or the SNAP programming environment (Price, Dong, and Lipovac 2017).

Note that the Markov Decision Process model relies on an estimate of the transition probability distribution $P(x'|x, \delta)$ of moving to state x' from x via the edit δ. The Hint Factory estimates this probability distribution based on transition frequencies in the trace data and therefore requires meaningful frequency information. As such, the Hint Factory can provide hints only for states that are part of the interaction network, and for which a trace to a correct solution in the interaction network exists. This has been dubbed the *hintable subgraph* (Barnes, Mostafavi, and Eagle 2016). In practice, students may move outside the hintable subgraph. Indeed, research has shown that for a reasonably small, open-ended programming task, over 90% of states are visited only once, indicating that future students will likely visit states that have not been seen before and may not even be connected to previously seen states in the legal move graph (Price and Barnes 2015).

Also note that the number of unique states remained high even after applying harsh canonicalization (Price and Barnes 2015). This result matches our own two datasets, where 97.23% and 82.79% of states were visited only once. This begs the question, how can the Hint Factory be extended to such sparsely populated state spaces? To address this question, Price, Dong, and Barnes (2016) have introduced *contextual tree decomposition* (CTD), which generates interaction networks only for small subtrees of the students' abstract syntax trees. Due to the size limitation, the state space for each subtree is significantly smaller and thus more densely populated with student data. However, the approach faces an ambiguity challenge in that one hint is generated for each (small) subtree of the student's state, and the student or the system has to select from these possible hints (Price, Zhi, and Barnes 2017a).

Overall, we observe that previous approaches are either limited by their reliance on frequency data, namely the Hint Factory, the Piech policy, and the Lazar policy, or by generating hints based only on a single reference solution, namely the Zimmerman policy, the Gross policy, or the Rivers policy. Our approach is an attempt to generate hints based on *multiple* reference solutions, but without relying on frequency information. More specifically, we use an affine combination of multiple reference solutions to express a virtual state to which the student should move, and we generate this affine combination such that it predicts what a capable student would have done in the same situation. As such, we use the same basic approach as the Hint Factory, in that we also try to bring the student closer to the next state of capable students in the same situation. However, we extend the Hint Factory by basing our prediction not on frequency, but on the movements of students in *similar* situations through the space of possible solutions. This state of possible virtual solutions, expressible as affine combinations of states we have seen before, is continuous; hence the name Continuous Hint Factory (CHF).

Note that embedding states in a continuous space has already been proposed by Piech, Jonathan Huang, et al. (2015), who constructed such an embedding via neural networks. More precisely, the embedding is computed by executing the programs on example data and recording the variable states P before executing a block of code A as well as the variable states Q after A has been executed. Both P and Q are embedded in a common space via a single-layer neural network, yielding the representations f_P and f_Q. Then, a matrix M_A is constructed that maps f_P to f_Q, that is, M_A is constructed such that $f_Q \approx M_A \cdot f_P$. This matrix M_A is the embedding of the code block A (Piech, Jonathan Huang, et al. 2015). However, this work has two crucial limitations: First, it relies on a task-specific representation of f_P and f_Q that is generated via execution, whereas the CHF only relies on edit distances, which are not task-specific (Mokbel, Gross, et al. 2013). Second, we provide a technique to convert the predictive result in the continuous space to an actual, human-readable edit, which the Piech approach lacks.

We also note connections to other approaches cited before. First, the CHF is connected to the work of Gross and Pinkwart (2015), in that we also recommend following the actions of students in a similar situation, but we integrate knowledge of more than one student. Second, similar to the work of Lazar and Bratko (2014), we recommend edits that are frequent in past student data but focus on those edits which have been applied in similar states. Finally, we incorporate many of the key concepts and approaches of Rivers and Koedinger (2015), in that we also apply canonicalization, and build upon the concept of path construction, a desirability score, as well as state reification to infer an edit that corresponds to the optimal hint in the embedding space. However, we extend this approach by considering not only edits toward the closest correct solution but edits toward all reference solutions and by replacing their desirability score with the distance to the recommended next state in the edit distance space. This alternative score incorporates the spirit of many of the criteria proposed by Rivers and Koedinger (2015), as it also punishes going too far away from the student's current solution, rewards getting closer to the goal, and represents what other students generally did, but it relies neither on frequency information, nor on an expert-chosen weighting between the different criteria.

In the next section, we introduce the CHF in more detail.

6.2 METHOD

The goal of the Continuous Hint Factory (CHF) is to predict what capable students would do and to generate an edit that corresponds to this prediction. To implement this goal, the CHF involves three steps. First, we collect trace data of capable past students. Second, we apply the predictive pipeline from Chapter 5. Third, we translate the prediction into a human-readable edit.

With respect to the first step, we recommend to record trace data of students whose success could be verified either by human tutors or via auto-grading approaches (e.g. unit tests). Further, we propose to pre-process these traces to avoid detours. More precisely, let $x_0, \delta_1, \ldots, \delta_T, x_T$ be a trace. If for any $t, t' \in \{0, \ldots, T\}$ with $t' > t$ it holds that $d(x_t, x_T) < d(x_{t'}, x_T)$, we remove the entire subtrace $\delta_{t+1}, \ldots, \delta_{t'}, x_{t'}$. This way, we ensure that the edit distance to the final solution always shrinks along the trace.

Once such a dataset is collected, we apply the predictive pipeline from Chapter 5 using Gaussian process regression (GPR). In particular, given a student's current state x, the GPR predictive function f yields a vector $f(x) = \phi(y)$ in an implicit kernel space for some unknown $y \in X$. As an example, consider the string edit distance example shown in Figure 6.2(b). In this example, the string edit distances are $d_{\Delta,c}(x, x_1) = d_{\Delta,c}(x, x_2) = 1$ and $d_{\Delta,c}(x_1, x_2) = d_{\Delta,c}(x_2, x_1) = 1$. For the hyper-parameters $\xi = 1$ and $\tilde{\sigma}^2 = 0$ we obtain

$$\vec{k}(x) = \left(\frac{1}{\sqrt{e}}, \frac{1}{\sqrt{e}}\right), \quad \boldsymbol{K} = \begin{pmatrix} 1 & \frac{1}{\sqrt{e}} \\ \frac{1}{\sqrt{e}} & 1 \end{pmatrix}, \text{ and } \quad \vec{\gamma}(x) = \boldsymbol{K}^{-1} \cdot \vec{k}(x) \approx \begin{pmatrix} 0.3775 \\ 0.3775 \end{pmatrix}$$

Thus, the recommended next state in the kernel space, indicated by the orange arrow in Figure 6.2(b), is

$$f(x) \approx \phi(x) + 0.3775 \cdot \big(\phi(y_1) - \phi(x_1)\big) + 0.3775 \cdot \big(\phi(y_2) - \phi(x_2)\big)$$

Now, the key remaining challenge is that we do not know the predicted next state y in the original state, but only its vectorial representation $f(x) = \phi(y)$ in terms of an affine combination. Further, as discussed in Chapter 5, inverting ϕ is a hard problem, especially for structured data (Bakır, Weston, and Schölkopf 2003; Bakır, Zien, and Tsuda 2004; Kwok and I. W.-H. Tsang 2004). Fortunately, our problem is conceptually simpler. We only need to infer an edit δ that brings the student closer to y, that is, we wish to solve the problem

$$\min_{\delta \in \Delta} d_{\Delta,c}\big(\delta(x), y\big) \tag{6.5}$$

where c is some cost function over Δ. We can address this problem by generalizing the edit distance theory we have already established in Section 2.3. In particular, we can show that Equation 6.5 simplifies drastically.

Theorem 6.2. *Let X be a state set, let Δ be a symmetric edit set over X, and let c be a symmetric cost function over Δ.*

Then, $d_{\Delta,c}$ is a pseudo-Euclidean distance for some positive spatial map $\phi^+ : X \to \mathbb{R}^m$ and some negative spatial map $\phi^- : X \to \mathbb{R}^n$.

Now, let $\{x_i\}_{i=1,\ldots,M} \subset X$, let $x, y \in X$, let $\boldsymbol{X}^+ := \big(\phi^+(x_1), \ldots, \phi^+(x_M), \phi(x)\big) \in \mathbb{R}^{m \times M+1}$, let $\boldsymbol{X}^- := \big(\phi^-(x_1), \ldots, \phi^-(x_M), \phi(x)\big) \in \mathbb{R}^{n \times M+1}$, and let $\vec{\alpha} \in \mathbb{R}^{M+1}$ such that:

$$\phi^+(y) = \boldsymbol{X}^+ \cdot \vec{\alpha}, \quad \phi^-(y) = \boldsymbol{X}^- \cdot \vec{\alpha}, \quad \textit{and} \quad \sum_{i=1}^{M+1} \alpha_i = 1$$

Then, the maximization problem in Equation 6.5 can be re-written as:

$$\min_{\delta \in \Delta} \alpha_{M+1} \cdot d_{\Delta,c}\big(\delta(x), x\big)^2 + \sum_{i=1}^{M} \alpha_i \cdot d_{\Delta,c}\big(\delta(x), x_i\big)^2 \tag{6.6}$$

Proof. The outline of the proof is as follows: We first show that the edit distance $\Delta_{c,\Delta}$ is self-equal and symmetric such that Theorem 2.2 applies and guarantees pseudo-Euclideanicity. Then, we apply Equation 2.9 to arrive at Equation 6.6.

For the details, refer to Appendix A.16. □

The minimization problem in Equation 6.6 has multiple key advantages. First, it does not require us to compute the vectorial embedding for any state. Instead, we can infer the optimal edit script solely based on the edit distance $d_{\Delta,c}(\delta(x), x)$, as well as the edit distances $d_{\Delta,c}(\delta(x), x_i)$, which we can compute explicitly. Second, our revised form of the problem provides the following, useful re-interpretation. We need to find an edit δ such that the resulting state stays close to the original state x, gets closer to states x_i for which α_i is positive, and gets further away from state x_i for which α_i is negative. Note that this re-interpretation is consistent with the criterion of Rivers and Koedinger (2014) that a next state should stay close to the student's current state. Finally, the re-formulation shrinks our search space, because we only have to consider edits that bring us closer to states x_i with positive coefficients α_i. We can extract such edits from the cheapest edit scripts between x and states x_i with positive coefficients α_i. For all these possible edits we can evaluate the error in Equation 6.6 and select the edit with the lowest error.

Consider the example illustrated in Figure 6.2(c). Recall that the coefficients α resulting from the GPR hint policy are $\alpha_{x_1} = \alpha_{x_2} \approx -0.3775$ and $\alpha_{y_1} = \alpha_{y_1} \approx +0.3775$. So we need to find an edit that brings us closer to $y_1 = \text{aac}$ and $y_2 = \text{bbc}$ but further away from $x_1 = \text{a}$ and $x_2 = \text{b}$. The cheapest edit script between x and y_1 is $\text{rep}_{2,\text{a}}\text{ins}_{3,\text{c}}$, and the cheapest edit script between x and y_2 is $\text{rep}_{1,\text{b}}\text{ins}_{3,\text{c}}$. Therefore, we need to consider the edits $\text{rep}_{2,\text{a}}$, $\text{ins}_{3,\text{c}}$, and $\text{rep}_{1,\text{b}}$. The resulting states of these edits would be aa, abc, and bb respectively. Amongst these options, abc minimizes our error because it is closer to both aac and bbc, further away from both a and b, and stays close to ab. Therefore, we would recommend $\text{ins}_{3,\text{c}}$ as hint.

In practical examples, this approach would be limited by the number of edits to be considered. For many training data points with positive coefficients and long edit scripts, this number could become infeasibly large. One way to limit the number of edits is to incorporate more of the criteria suggested by Rivers and Koedinger (2014) and consider only edits that result in syntactically correct states, result in programs that fulfill at least

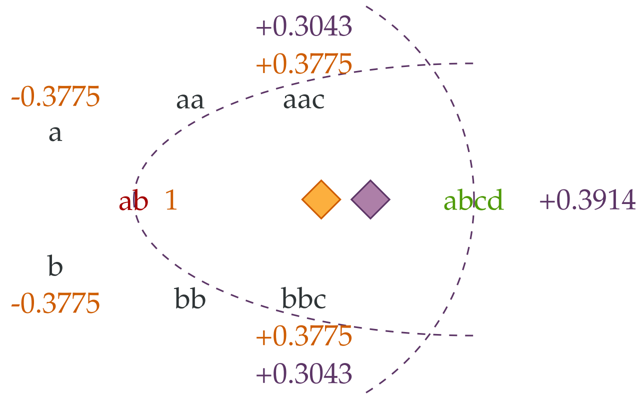


Figure 6.4: An illustration of the sparse representation of the recommended state for the string example of Figure 6.2. The student's current state is the string $x =$ ab (shown in red), the closest correct solution is the string $x^* =$ abcd (shown in green). The coefficients α_i and the represented state $\phi(x) + \pi_{\text{GPR}}(x)$ returned by the Gaussian process regression (GPR) policy are shown in orange. The sparse coefficients and the corresponding represented state are drawn in purple. The constraints $d(x_i, x) \leq d(x, x^*)$ and $d(x_i, x^*) \leq d(x, x^*)$ are illustrated by dashed purple lines.

as many test cases or get us closer to a correct solution. In addition, we propose to limit the search space to a reasonable size by using fewer coefficients to represent the recommended state, namely a subset of those states which lie between the student's current state x and the next correct solution x^*. This is consistent with another criterion proposed by Rivers and Koedinger (2014), namely that the recommended state should both be close to a correct solution and to the student's current state.

In formal terms, we look for a coefficient vector $\tilde{\alpha}$ such that $\tilde{\alpha}_i$ is nonzero only if $d(x_i, x) \leq d(x, x^*)$ and $d(x_i, x^*) \leq d(x, x^*)$, such that at most m entries are nonzero, such that the sum over all entries of $\tilde{\alpha}$ is 1, and such that the state represented by $\tilde{\alpha}$ is as close as possible to the state represented by $\vec{\alpha}$. While this is a NP-hard problem, multiple simple heuristics exist, which have been summarized by D. Hofmann et al. (2014). In our experiments, we apply both kernelized orthogonal matching pursuit and an approximation via the largest entries of $\vec{\alpha}$ and use whatever approximation is closer to the actual recommended state.[4]

Consider the example illustrated in Figure 6.4. Here, the original coefficients α returned by the GPR hint policy are shown in orange and represent the state shown as an orange diamond. Now, assume that the student's current state is the string "ab" and the closest correct solution is the string "abcd". In that case, only the strings "ab", "aac", "bbc", and "abcd" fulfill the constraints $d(x_i, x) \leq d(x, x^*)$ and $d(x_i, x^*) \leq d(x, x^*)$ (indicated by dashed purple lines). If we now try to represent the recommended state by using only 3 of those four strings, this results in a representation via the strings "aac", "bbc", and "abcd" with roughly equal coefficients, resulting in a represented state (shown in purple) close to the original hint. The selected hint, in this case, would still be $\text{ins}_{3,\text{c}}$.

4 We note that kernelized OMP does, per default, not guarantee an affine combination. For an adjusted version that guarantees an affine combination, refer to Appendix A.17.

Summary

To conclude our description of the Continuous Hint Factory (CHF), we provide a short summary of all steps involved. First, we need to perform the following preparation steps:

1. Collect trace data from successful students.
2. Remove all intermediate states in the traces that do not get closer to the goal.
3. Compute the canonic forms of the trace data and their pairwise edit distances.
4. Compute the pairwise radial basis function kernel values $\boldsymbol{K}$. The length scale parameter $\tilde{\xi}$, as well as the noise parameter $\tilde{\sigma}$, can be selected such that the predictive accuracy of the GPR model on unseen evaluation data is as high as possible.
5. Perform eigenvalue correction on $\boldsymbol{K}$.

Now, assume that a new student is in state x and requests help. In that case, the following steps need to be performed.

1. Compute the canonic form of x and the edit distance of this canonic form to all canonic forms in the trace data before.
2. Compute the radial basis function kernel values $\vec{k}(x)$ based on these distances.
3. Extend the eigenvalue correction to the new kernel values.
4. Compute the coefficients α of the GPR hint policy via the formulae in Theorem 5.1.
5. Optionally, sparsify these coefficients via one of the techniques of D. Hofmann et al. (2014).
6. Compute the cheapest edit scripts between x and all training states x_i for which α_i is positive.
7. Optionally, subselect edits δ from these edit scripts that result in states $\delta(x)$ that conform to further criteria, e.g., unit test fulfillment, or syntactic correctness (Rivers and Koedinger 2014).
8. Compute the error term in Equation 6.6 for all remaining edits.
9. Select the edit with the lowest error as hint.

This concludes our description of the CHF. In the next section, we evaluate the CHF approach experimentally.

6.3 EXPERIMENTS

We consider two datasets for our analysis.

Guessing Game Dataset

First, we consider a dataset collected in an introductory undergraduate computing course for non-computer science majors during the Fall of 2015 at a research university in the south-eastern United States. The course had approximately 80 students, split among six lab sections. The first half of the course focused on learning the Snap[5] programming language through a curriculum based on the *Beauty and Joy of Computing* (Garcia, Harvey, and Barnes 2015). Here, we focus on the "Guessing Game" task, which had the following description: "The computer chooses a random number between 1 and 10 and continuously asks the user to guess the number until they guess correctly." Students did not receive specific instructions regarding the form of the program. An example solution for the task is presented in Figure 6.1(a). Students worked on this assignment during class for approximately one hour, with a teaching assistant available to assist them and the option of working in pairs. The class was conducted as normal, and the students were not informed that data was being collected. The state of the student's program was recorded after every edit. Students who did not correctly select the assignment they were working on were excluded from the analysis. The dataset consists of 52 traces with 8669 states overall.

Each of the final states was graded by two independent graders. The graders used a rubric consisting of nine assignment objectives and marked whether each state successfully or unsuccessfully completed each objective. The graders had an initial agreement of 94.5%, with Cohen's $\kappa = 0.544$. After clarifying objective criteria, each grader independently regraded each state where there was disagreement, reaching an agreement of 98.1%, with Cohen's $\kappa = 0.856$. Any remaining disagreements were discussed to create final grades for each assignment. As our aim is to predict what *capable* students would do, we kept only traces that successfully completed at least eight of the nine objectives. This left 47 traces with 7864 states.

UML Dataset

As a second dataset, we utilize data collected in an introductory programming course for computer scientists at a German university in 2012. The students were asked to draw a Unified Modelling Language (UML) activity diagram that describes the process of adding two binary numbers. An example solution is shown in Figure 6.5. From the available student data, we extracted six typical strategies and created two correct traces and one erroneous trace for each strategy. Overall, the correct traces contained 364 states and the erroneous traces 115 states. We presented each state in the erroneous traces to three graders who independently were asked to suggest all possible edit hints that could be given to a student in the particular situation, taking past states into account. We also instructed the tutors to provide an estimate of hint quality in the interval $[0, 1]$ for each of their hints, taking into account the following criteria: 1) Does the hint follow the strategy of the student? 2) Does the hint conform to the student's current focus of attention or does it address a different part of the state? 3) Is the hint effective in addressing the problems in the student's state? 4) Is the hint effective in guiding the student toward a solution? In a second meeting, all tutors met to add ratings for the hints of the respective other tutors and to discuss discrepancies in the ratings. If after discussion at least one

5 http://snap.berkeley.edu

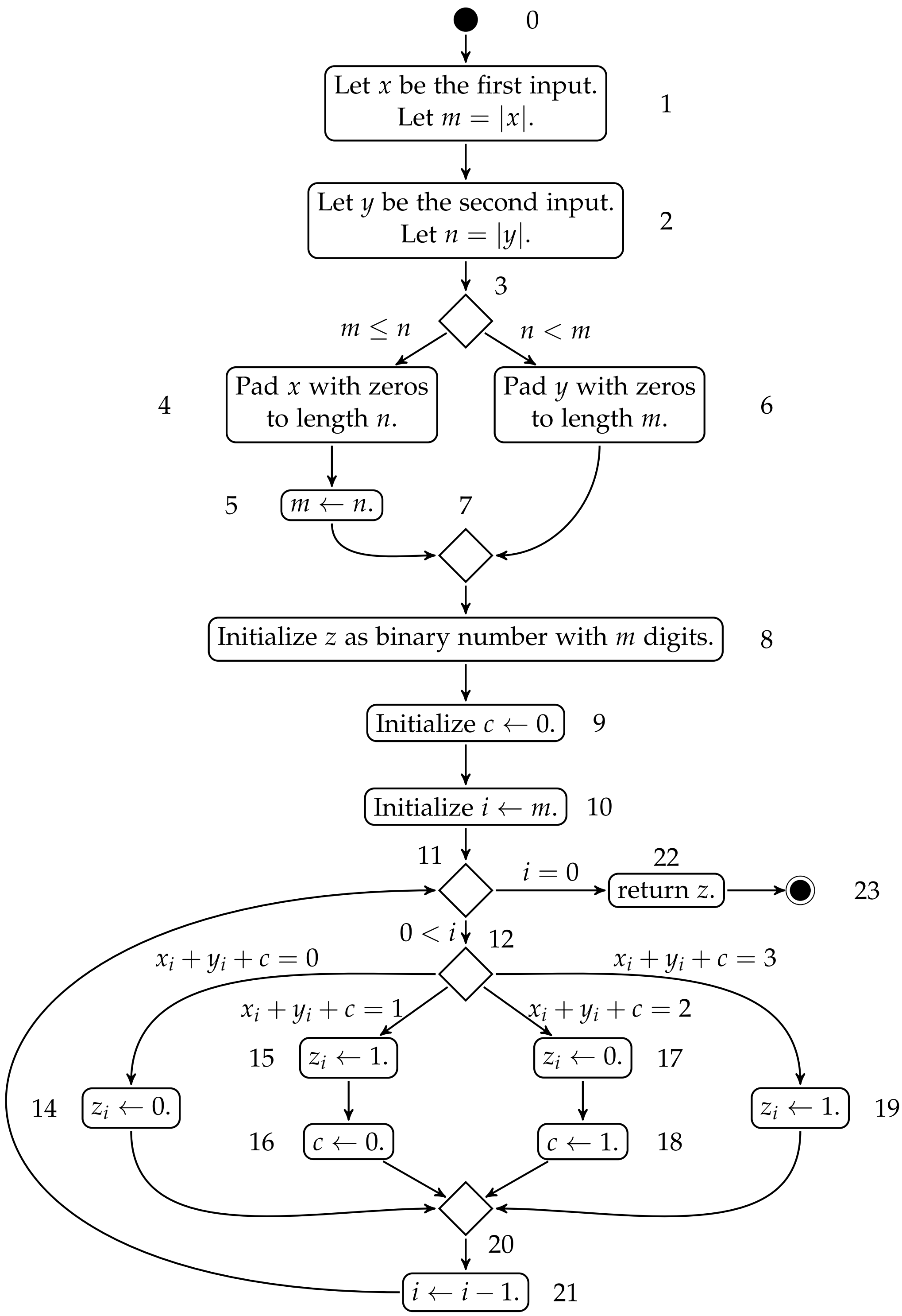


Figure 6.5: A correct example solution for the UML binary adder task. Numbers indicate the order in which nodes have been added to the UML diagram.

expert rated a hint with a grade below 0.5, the hint was excluded from the set. 1053 hints remained after this process. The average inter-rater correlation via Pearson's r was $r = 0.588$, indicating moderate agreement.

Procedure

We represent the states of both datasets as trees. In case of the Snap dataset, we directly used the abstract syntax trees as displayed in Figure 6.3. For the UML dataset, we removed back-references, such as the arrow from node 21 to node 11 in Figure 6.5, to obtain a tree structure. We also added the text of the respective node to the label, for example, "return z" for node 21 in Figure 6.5. In both datasets, we canonicalized the trees by normalizing variable names and literals, normalizing the order of binary relations, and removing non-executable code, as recommended by Rivers and Koedinger (2012).

As an edit distance, we employ the tree edit distance of Zhang and Shasha (1989) as introduced in Section 2.3.3. For the Snap dataset we use a uniform cost function of 1 for deletions, insertions, and replacements. For the UML dataset, we define deletion and insertion costs as 1, replacement costs between unequal node types as infinite, and replacement costs between action nodes (displayed as ellipses in Figure 6.5) as the string edit distance between the node text, normalized to the interval $[0, 1]$. We ensure Euclideanicity of the edit distances via clip eigenvalue correction (Gisbrecht and Schleif 2015). Based on the tree edit distance, we exclude states that do not get closer to the final state in the respective trace. For the guessing game dataset, this left 1005 states, 812 of which were unique. Of these 812 unique states, 94.09% were visited only once. For the UML dataset, the procedure did not remove any states. Of the 364 states in the UML dataset, 215 were unique, and of these unique states, 82.79% were visited only once. These numbers indicate that meaningful frequency information is only available for very few training states, which is consistent with the findings reported by Price and Barnes (2015) on similar data from an open-ended Snap programming task.

We considered all hint policies mentioned in this chapter as reference policies for comparison. Due to the lack of meaningful frequency information in our data, however, we can neither apply the Hint Factory (Barnes and J. Stamper 2008) nor the Piech policy (Piech, Sahami, et al. 2015). Furthermore, to keep the approach generic, we do not use task-specific syntactic or unit test information for our experiments, which rules out the policy of Lazar and Bratko (2014). There remain the policy of Gross and Pinkwart (2015), which uses the successor of the next state in the trace data to construct a hint, the policy of Zimmerman and Rupakheti (2015), which uses the closest correct solution to construct a hint, and the policy of Rivers and Koedinger (2015), which also uses the closest correct solution. The Zimmerman policy and the Rivers policy mainly differ in *how* hints are constructed from the closest correct solution. However, given that we use neither frequency nor syntactic or semantic correctness information, and consider only single edits instead of edit combinations, both policies become very similar such that we only consider the Gross policy and the Zimmerman policy in this case.

We implemented all hint policies in MATLAB® and utilized the same implementation for Gaussian process regression (GPR) as in Chapter 5. To optimize the kernel length scale ξ and the noise standard deviation $\tilde{\sigma}$ of the GPR model, we employ a random hyper-parameter search with 10 repeats as recommended by Bergstra and Bengio (2012). We set the maximum number of training states to represent the hint of the CHF policy to $m = 11$.

Table 6.1: Mean RMSE ± standard deviation in predicting the next step and the final step of capable students for both the Snap dataset, as well as the UML dataset. The first column lists the different prediction schemes. Lower values are better, and a value of 0 is ideal.

	Snap		UML	
Prediction scheme	Next	Final	Next	Final
Do nothing	17.5 ± 3.89	39.3 ± 9.36	5.27 ± 0.53	28.9 ± 5.28
1-NN	23.7 ± 5.39	39.1 ± 9.49	7.89 ± 3.50	29.1 ± 6.00
Closest-correct	26.7 ± 5.46	43.0 ± 8.60	25.50 ± 1.23	19.9 ± 8.42
GPR	16.6 ± 4.09	37.8 ± 9.15	3.18 ± 1.66	27.8 ± 5.32

Research Questions & Results

In our experiments, we investigate two research questions, which we will cover in turn. We evaluate statistical significance using a one-sided Wilcoxon sign-rank test. Further, we apply a Bonferroni correction to avoid type I errors due to multiple tests.

RQ1: How well does the GPR model capture the behavior of capable students, that is, can GPR predict what a capable student would do?

To investigate RQ1, we consider two measures of predictive accuracy. First, we measure the distance between the predicted next state of the GPR model and the actual next state of the respective student (next-step error). Second, we measure the distance between the predicted next state and the *final* state of the respective student (final-step error). We measure both distances in terms of root mean square error (RMSE) as in Equation 5.2. We evaluate the next-step error and the final-step error in a leave-one-out crossvalidation over the traces, which means that in each fold we use all but one trace as training data for the prediction and the remaining trace to evaluate the model.

Note that RQ1 is only concerned with the prediction module of each hint policy, that is, the reference state based on which edits are generated, not the edits that are used as hints. As such, we do not directly compare with the Gross or Zimmerman policy but with the reference states they would use, namely the successor of the closest next solution (1-NN), and the closest correct solution (closest-correct) respectively. Given the nature of these references, we would expect that the 1-NN prediction would perform well in the next-step error but badly in the final-step error and that the closest-correct prediction would perform badly in terms of the next-step error but good in terms of the final-step error. As an additional reference, we provide the error for the trivial prediction of staying in the same state, that is, $\pi(x) = x$ (do nothing).

Table 6.1 shows the RMSE averaged over the crossvalidation folds (± standard deviation) for both datasets where each column lists one error measure for all prediction schemes[6].

6 Note that the RMSE cannot be interpreted directly as the average number of edits between the predicted next state and the gold standard because the RMSE assigns higher weight to larger deviations due to the square. Further, in this particular evaluation, but not for RQ2, eigenvalue correction distorts the edit distances to become larger.

Statistical analysis reveals that GPR is significantly better in predicting the next state compared to all other baselines for both datasets ($p < .01$). Further, GPR is significantly better in predicting the final state compared to the "Do nothing" and the 1-NN prediction for both datasets ($p < .01$), and better than the Closest-correct prediction for the Snap dataset ($p < .001$). Interestingly, the 1-NN prediction does not perform better in predicting the actual next state of a student compared to staying in the same state, indicating that students in both data sets do not necessarily move along the same states, even though their directions may be consistent. This is also visible in the embedding in Figure 6.1(b).

Furthermore, we note that, counter to our expectations, the Closest-correct prediction has a higher final-step error on the Snap dataset than any other prediction scheme, which indicates that students' final solutions are quite diverse. This effect is likely explained by the high strategic variability in an open-ended programming task such as the guessing game task. For such tasks, we expect that the averaging approach of GPR to be particularly helpful, because the general trends in the datasets may be more akin to the student's actual plans than a single closest correct solution. Conversely, the UML dataset features less strategic variability, and the closest correct solution of another student is still close to the final state of the student for which the prediction is made, which is reflected in significantly better predictions of the Closest-correct prediction compared to all other prediction schemes ($p < 10^{-3}$). Overall, we can conclude that GPR is more accurate in predicting the next state of students compared to other baselines on our example datasets and that this is especially the case for the Snap dataset, which is characterized by high strategic variability.

RQ2: Do the hints of the Continuous Hint Factory correspond to the hints of human tutors?

To investigate RQ2, we require a reference measure of hint quality, which is provided by the quality judgments of human tutors in the UML dataset. In particular, we iterate over every state in the erroneous traces of the UML dataset and generate a hint with each hint policy, using all correct traces as training data. If multiple edits achieve the lowest error rank, we resolve ties by selecting the edit as hint that is closest to the root of the tree. If the recommended hint of the policy matches at least one tutor hint, we assign the average quality rating of the human tutors for that hint. Otherwise we set the rating to 0. This is similar to the evaluation scheme suggested by Price, Zhi, Dong, et al. (2018). We report five evaluation measures, namely the median and mean hint quality, the fraction of hints with a quality > 0, the distance between the policy hint and the closest human tutor hint in terms of RMSE, and the fraction of states for which a hint could be generated. In addition to the Gross and the Zimmerman policy, we also compare to a random policy, which selects a random reference state from the training state and recommends an edit on the shortest path towards that state as hint. Finally, we also evaluate the best-rated tutor hint as the gold standard.

The experimental results are shown in Table 6.2, where each column displays one evaluation measure, and each row lists the results for one hint policy. Regarding hint quality, we observe that the CHF performs significantly better compared to a random policy ($p < .01$), and significantly worse compared to human tutor hints ($p < .001$), but otherwise there are no significant differences between the hint policies. This indicates that for simple datasets like the UML dataset, which feature low strategic variability, single reference states are sufficient to generate viable hints. Interestingly, though, we could also observe cases where the CHF did perform better. In particular, Figure 6.6 displays a UML

Table 6.2: The hint evaluation measures for all hint policies on the UML dataset. Mean hint quality and mean ambiguity are reported with standard deviation. For all measures except the RMSE, higher numbers are better with a value of 1 and 100% respectively being ideal.

	Hint quality			RMSE	Hintable
Hint policy	Median	Mean	> 0		
Random	0.0	0.360 ± 0.456	39.1%	1.42	83.5%
Tutor	1.0	0.994 ± 0.021	100.0%	0.00	100.0%
Gross	0.8	0.569 ± 0.465	60.9%	1.42	100.0%
Zimmerman	0.8	0.557 ± 0.431	64.3%	1.48	100.0%
CHF	0.9	0.590 ± 0.471	61.7%	1.36	97.4%

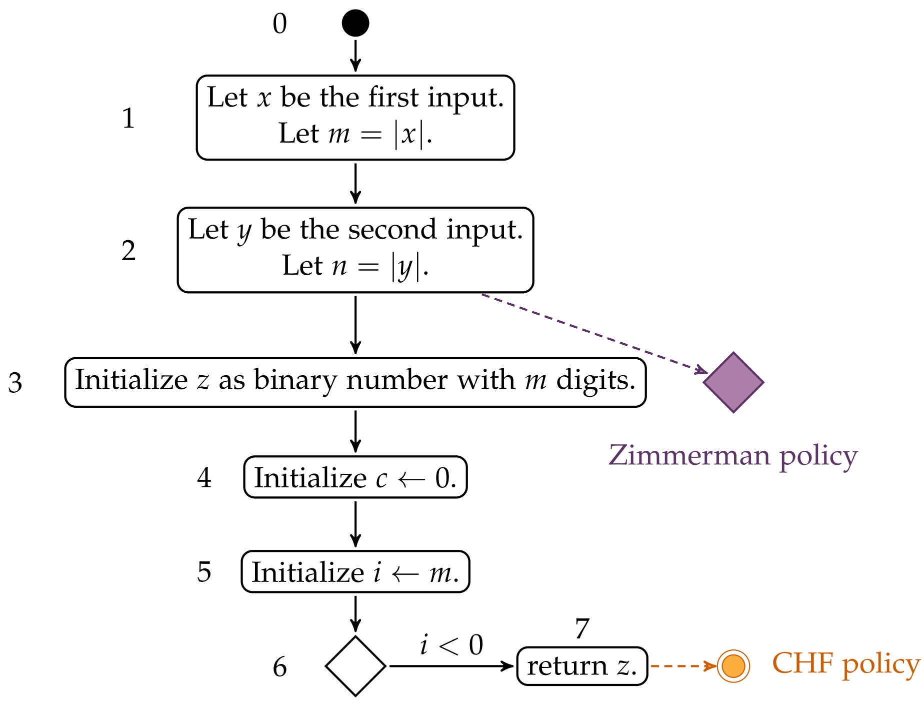


Figure 6.6: An example state from the UML dataset, where the Zimmerman policy generates a hint (purple) that is not in the student's current focus of attention. In contrast, the Continuous Hint Factory (CHF) generates a hint (orange) that acts upon the last added node.

diagram where the Zimmerman policy recommends appending a decision node close to the root (purple), which is outside the student's current focus of attention because the last node the student added was the "return z" node at the bottom of the diagram. Accordingly, the CHF recommends appending a "finish" node to that branch (orange).

Another interesting finding is that the CHF and the Gross policy consistently achieved perfect hint quality for the first three steps in each trace. This is important in light of the research of Price, Zhi, and Barnes (2017b), which indicates that students are more likely to seek help and follow hints if *early* hints provided by the system were useful.

6.4 CONCLUSION

This chapter makes three primary contributions. First, we have reviewed existing work on edit-based hint policies in light of the mathematical framework of edit distance theory. Second, we have applied our time series prediction pipeline to predict the behavior of capable students in solving a multi-step learning task. Finally, we have introduced a simple algorithm to compute an edit that brings students closer to the predicted next state. We call this scheme the Continuous Hint Factory (CHF).

In our experiments, we have shown that the CHF model outperforms other approaches in predicting what capable students would do, especially in an open-ended programming dataset with high strategic variability. We also showed that the CHF reproduces human tutor hints about as well as existing hint policies on a simple UML diagram task. These results indicate that the averaging approach of the CHF is beneficial for prediction, but that this advantage is not necessarily reflected in higher hint quality, at least for a simple learning task with low strategic variability.

We note that the CHF still has several limitations. In particular, the CHF can only be applied if an edit distance is available that is efficient, takes syntax and semantics into account appropriately, and yields edits that are actionable for students. Further, as in any data-driven hint approach, hint quality will suffer if the strategy of a new student is substantially different from anything that the system has seen before. More subtly, eigenvalue correction may distort distances and loose information that may be critical for prediction or hint generation (Nebel, Kaden, et al. 2017). Finally, our approximation scheme to infer an edit from a predictive result may fail to capture all of the predictive information.

With regards to evaluation, our assessment of hint quality is not definitive, and it appears likely that our proposed approach only yields significant advantages compared to existing work on more complicated tasks than to the ones we investigated. Further, we do not yet know how a difference in hint quality translates to learning outcomes in students. After all, better hints from the view of a tutor may not always yield better learning outcomes, due to difficulties in sense-making or lack of prior knowledge on the student's side (Aleven, Roll, et al. 2016). Finally, we acknowledge that our evaluation is rather narrow, including only two learning tasks from different domains.

These limitations notwithstanding, we have developed an accurate predictive approach for student behavior in multi-step tasks and have provided a general hint-generation pipeline that is applicable far beyond the domain of computer progamming.

7 SUPERVISED TRANSFER LEARNING

Summary: Most machine learning models implicitly assume *stationarity* of the data, meaning that the data distribution does not change over time. Whenever this stationarity assumption is violated, models trained at one point in time may not correctly process later data. Transfer learning methods try to account for the difference between training and test data and learn mappings between the two. We propose a novel transfer learning framework where a mapping from test to training data is learned based on a supervised loss on the training data. We implement our framework for linear transfer mappings and the loss functions of generalized learning vector quantization as well as labelled Gaussian mixture models. On artificial data we demonstrate that we are able to successfully transfer target data back to the source space even in cases where reference methods in the literature fail and that our approach is orders of magnitude faster compared to training a new model.

Publications: This chapter is based on the following publications.

- Paaßen, Benjamin, Alexander Schulz, and Barbara Hammer (2016). "Linear Supervised Transfer Learning for Generalized Matrix LVQ". In: *Proceedings of the Workshop New Challenges in Neural Computation (NC^2 2016)*. (Hannover, Germany). Ed. by Barbara Hammer, Thomas Martinetz, and Thomas Villmann. **Best presentation award**, pp. 11–18. URL: https://www.techfak.uni-bielefeld.de/~fschleif/mlr/mlr_04_2016.pdf#page=14.
- Paaßen, Benjamin et al. (2017). "An EM transfer learning algorithm with applications in bionic hand prostheses". In: *Proceedings of the 25th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN 2017)*. (Bruges, Belgium). Ed. by Michel Verleysen. i6doc.com, pp. 129–134. URL: http://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2017-57.pdf.
- — (2018). "Expectation maximization transfer learning and its application for bionic hand prostheses". In: *Neurocomputing* 298, pp. 122–133. DOI: 10.1016/j.neucom.2017.11.072.

Source Code: The MATLAB(R) source code corresponding to the content of this chapter is available at http://doi.org/10.4119/unibi/2912671.

The aim of machine learning is to identify patterns in a set of training data such that these patterns hold for unseen and new data. The ability to correctly apply patterns to unseen data is called *generalization* (Bishop 2006). Generalization is simple if the training data and the new data are *similar*, in the sense that they stem from the same underlying distribution. However, in many scenarios, this assumption is violated (Cortes et al. 2008). For example, the training data may have been selected in a biased way and thus patterns that hold for the training data may not hold for the overall population (Cortes et al. 2008). Further, the generative process of the data may change over time, for example due to external disturbances (Ditzler et al. 2015). Finally, one may want to generalize to data which are generated from another source (Ben-David et al. 2006). Each of these scenarios

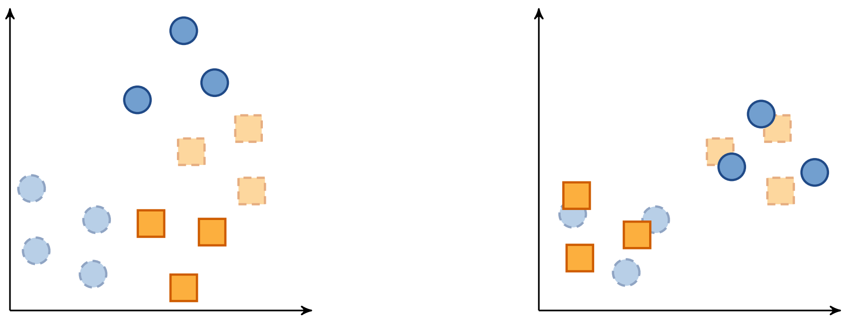


Figure 7.1: An illustration of two kinds of concept drift. Left: Virtual concept drift, also known as covariate shift or sample selection bias. Right: Real concept drift, where source and target data are related by rotation. Colors and shapes illustrate class assignments. Source data is drawn dashed and transparent, while target data is opaque.

leads to a mismatch between a model derived from the training data and the new data, which in turn may limit generalization.

In cases of abrupt changes in the data distribution, many classical approaches would suggest to discard the entire learned model and start learning a new model using only data from the new distribution (Ditzler et al. 2015). However, if only few data from the target distribution are available, this newly trained model may be inaccurate. Instead, we propose to re-use the trained model from the source domain, and to only learn the *transfer function* between the source and target domain, which makes our proposed framework an instance of *transfer learning* (S. J. Pan and Q. Yang 2010; Weiss, Khoshgoftaar, and D. Wang 2016). In particular, our proposed approach can be regarded as a case of *heterogeneous domain adaptation*, which is concerned with learning mappings between domains such that knowledge can be transferred from one to the other (Weiss, Khoshgoftaar, and D. Wang 2016).

In more detail, we propose to learn a mapping h from the target domain to the source domain using only few target domain data such that the loss of the source model on these target data is minimized. In other words, we adapt the representation of the target space data to the source model. The main contributions of this chapter are to formalize this supervised transfer learning framework, and to provide two instances of supervised transfer learning, one for learning vector quantization models and one for labeled Gaussian mixture models. Note that both models can be seen as an instance of metric learning. In particular, our transfer learning approach adapts the target space representation such that the source space metric becomes applicable to the target space.

We begin by covering some related work on changing data distributions and adaptations, then describe our own method, before we evaluate our approach experimentally and close with a conclusion.

7.1 RELATED WORK

In this chapter, we consider classification tasks. In particular, we assume a list of tuples $(\vec{x}_1, y_1), \ldots, (\vec{x}_M, y_M)$, which we call the source dataset. Each of these tuples consists of an input data point $\vec{x}_i \in \mathbb{R}^m$ for some $m \in \mathbb{N}$, and a label of interest $y_i \in \{1, \ldots, L\}$ for some $L \in \mathbb{N}$. Our task is to construct a machine learning model $f : \mathbb{R}^m \to \{1, \ldots, L\}$, such that f predicts the correct label for the source dataset, and generalizes to target data. However, the literature covers several scenarios in which the model f may be able able to correctly predict the source data, but may fail to generalize.

For example, Shimodaira (2000) has introduced the notion of *covariate shift*, which refers to differences in the marginal density $p(\vec{x})$ between source and target data, while the conditional label distribution $P(y|\vec{x})$ remains the same. In that case, the target data may contain more samples in a region of the data space where the model is inaccurate and thus the model may fail to generalize (see Figure 7.1, left).

Similarly, Cortes et al. (2008) have established *sample selection bias correction theory*, which assumes that a *true* underlying distribution $P(y, \vec{x})$ exists, but that the source data is sampled not from this distribution directly but only from a limited region of the space. In that case, the model may fail to correctly predict samples in the regions from which no samples were available and thus fail to generalize.

Note that both scenarios assume that the change from source to target data is discrete, without regard for the time dimension. By contrast, research on *concept drift* is concerned with changes over time. In particular, a change in the marginal distribution $p(\vec{x})$ is called *virtual concept drift*, while a change that also affects the conditional distribution $P(y|\vec{x})$ is called *real concept drift* (Ditzler et al. 2015). Furthermore, one can distinguish between *gradual* drift and *sudden* drift (Ditzler et al. 2015). From the perspective of concept drift, covariate shift and sample selection bias would be special cases of sudden, virtual concept drift. In our work, we focus on cases of real concept drift because in these cases even target data that are close to source data may be misclassified (see Figure 7.1, right).

A final perspective is provided by the fields of *transfer learning* and *domain adaptation*, which are concerned with settings in which source and target data stem from different domains (Ben-David et al. 2006; S. J. Pan and Q. Yang 2010; Weiss, Khoshgoftaar, and D. Wang 2016). In these cases, a model f learned on the source data is a priori not applicable and needs to be adapted to the target domain.

The first step in adapting to changes between source and target data is to detect whether a change has occurred. In some cases, a change may be obvious, for example in case of domain adaptation. For non-obvious cases, various change detection tests exist, for example based on deviations in the sample mean, the sample variance, or the classification error (Ditzler et al. 2015). Once a change has been detected, the next step is to adapt to the change.

In case of gradual concept drift, be it virtual or real, one can apply incremental learning schemes to smoothly adapt a model to a new distribution via single samples or mini-batches, such as incremental support vector machines, Learn++, on-line random forests, or incremental learning vector quantization (Ditzler et al. 2015; Gepperth and Hammer 2016; Losing, Hammer, and Wersing 2016a).

In case of a sudden virtual concept drift, such as covariate shift or sample selection bias, the source data can be augmented by re-weighing the source data points $\vec{x}_i$, such that the distribution of the re-weighted source data corresponds to the distribution of the target data (Cortes et al. 2008; Jiayuan Huang et al. 2007; Sugiyama et al. 2008). If the drift is sudden and real, the source data is typically considered to be invalid and should be forgotten entirely, which also means that the old model f should be discarded and replaced by a new one (Ditzler et al. 2015). Note that models are typically optimized only for either sudden or gradual drift. To our knowledge, only the the long-and-short-term-memory model by Losing, Hammer, and Wersing (2016b), has the ability to adapt to both kinds of drift.

A lacuna in all these approaches is that they do not take the relatedness between source and target data into account. By contrast, transfer learning and domain adaptation approaches assume that source and target data can be embedded in a common latent space in which a model can be learned that applies to all data (S. J. Pan and Q. Yang 2010; Weiss, Khoshgoftaar, and D. Wang 2016). One class of transfer learning approaches are concerned with *invariant feature representations* that can be computed for both source and target data and then permit a correct classification of both, such as the first layers of deep convolutional neural networks or scale-invariant features (Glorot, Bordes, and Bengio 2011; Long et al. 2015; Lowe 1999). Note that this approach does not help in cases of real concept drift where the label for a region of the data space changes, because this region would have to be mapped to different locations for correct classification, which a single mapping can intrinsically not do.

By contrast, Blitzer, McDonald, and Pereira (2006) and Blöbaum, Schulz, and Hammer (2015) as well as others use *different* mappings from source and target space to a common latent space. As such, the approach is conceptually strong enough to deal with real concept drift and re-use a learned model on source data for the target domain. However, these approaches do not take label information in the target data into account, which leads to failure in all cases where the relation between source and target data is ambiguous. Consider the right plot in Figure 7.1 as an example. In this case, source and target data are related by a 180° rotation. However, without label information for the target data, it would be equally plausible to assume that no change between source and target data has occurred, because the marginal density $p(\vec{x})$ is the same for source and target data.

Only few approaches to date have taken label information into account as well. First, the adaptive support vector machine (a-SVM) (J. Yang, Yan, and Hauptmann 2007), which assumes that source and target space are the same, but that real concept drift has occurred. In turn, a model f on the source data may misclassify some target data points. The a-SVM learns a support vector machine model f' that predicts the difference between the predicted labels of f for some target sample points and the actual labels of these points. As such, the source model f is re-used for all data points that are still correctly classified but adapts the source model for all other points. However, the a-SVM may still fail for the real concept drift example in Figure 7.1, because it has to re-learn the entire model and does not exploit the simple, linear relationship between source and target data.

By contrast, the asymmetric regularized cross-domain transformation (ARC-t) approach (Kulis, Saenko, and Darrell 2011) learns a linear mapping $\boldsymbol{H}$ between source data points $\vec{x}$ and target data points $\hat{x}$ by maximizing the inner product $\vec{x}^\top \cdot \boldsymbol{H} \cdot \hat{x}$ if $\vec{x}$ and $\hat{x}$ have the same label and minimizing it otherwise. The mapping can then be used to transfer source data to the target space and train a target domain classifier there. In line with our framework, it is also possible to transfer target space data to the source space to make a source classifier applicable again. Note, however, that ARC-t is challenged whenever classes are multi-modal because in that case, maximizing the inner product between all points within classes may yield conflicting objectives.

A more flexible approach is offered by heterogeneous feature augmentation (HFA) (Duan, Xu, and I. Tsang 2012), which learns two linear mappings $\boldsymbol{P}$ and $\boldsymbol{Q}$ from the source and the target space to a shared latent space such that the loss of a support vector machine trained on all data in the latent space is minimized. Note that this bears some similarity to our proposed framework as the transfer mappings $\boldsymbol{P}$ and $\boldsymbol{Q}$ are also learned based on a classifier loss function, namely that of the support vector machine. In contrast to our method, though, the mappings $\boldsymbol{P}$ and $\boldsymbol{Q}$ are learned only implicitly in a kernel-based approach and can not be used to transfer target space data to the source space, which would be necessary to re-use an already trained source space classifier. Instead, HFA has to train a new classifier in the latent space.

As such, our proposed framework fills a notable gap in the existing literature by a) learning a transfer mapping explicitly (other than a-SVM and HFA) that b) permits the application of an already learned source space classifier without retraining (other than HFA) and c) is trained based on the loss of that classifier (other than ARC-t). In the following section, we will formalize this problem and develop two learning approaches, one based on learning vector quantization, and one based on labeled Gaussian mixture models.

7.2 METHOD

We consider the following scenario. In a first step, we are given a source dataset $(\vec{x}_1, y_1), \ldots, (\vec{x}_M, y_M)$ of data points $\vec{x}_i \in \mathbb{R}^m$ and corresponding labels $y_i \in \{1, \ldots, L\}$. Based on this source dataset, we train a source model f from some set of possible models $\mathcal{F}$ by solving the optimization problem

$$\min_{f \in \mathcal{F}} E\Big(f, (\vec{x}_1, y_1), \ldots, (\vec{x}_M, y_M)\Big) \tag{7.1}$$

for some loss function $E : \mathcal{F} \times (\mathbb{R}^m \times \{1, \ldots, L\})^* \to \mathbb{R}$.

In a second step, we receive a target dataset $(\hat{x}_1, \hat{y}_1), \ldots, (\hat{x}_N, \hat{y}_N)$ of data points $\vec{\hat{x}}_j \in \mathbb{R}^n$ and corresponding labels $\hat{y}_j \in \{1, \ldots, L\}$. Note that we generally assume that the target dataset is much smaller than the source dataset (i.e. $N \ll M$), and may not even cover all labels (i.e. $|\{\hat{y}_1, \ldots, \hat{y}_N\}| < L$). Given this target dataset, our goal is to find a *transfer function* $h : \mathbb{R}^n \to \mathbb{R}^m$ from some function set $\mathcal{H}$ such that $f \circ h$ is a classifier that generalizes well to other target data. We can learn this transfer function by minimizing the loss E with respect to h instead of f, that is:

$$\min_{h \in \mathcal{H}} E\Big(f, \big(h(\hat{x}_1), \hat{y}_1\big), \ldots, \big(h(\hat{x}_N), \hat{y}_N\big)\Big) \tag{7.2}$$

This learning problem has an intuitive interpretation. We are looking for a function h that "cleans up" any disturbance that has occurred for the target data such that for each transferred target data point the original model f can be applied as before. It is important to note that the transfer function h does *not* need to match the marginal distribution of the source space, but only the conditional distribution, which may be a considerably simpler task.

Note that we could also take the approach of discarding the source model f entirely and solving problem 7.1 only for the target space data. However, in two cases, problem 7.2 is preferable to problem 7.1. First, in case the available target data available is insufficient to accurately estimate the conditional distribution $P(\hat{y}|\hat{x})$, for example because of too few samples, because not all labels are covered, or because of biased sampling. Second, in case the transfer function h is sufficiently simple such that it is easier to optimize compared to the model f.

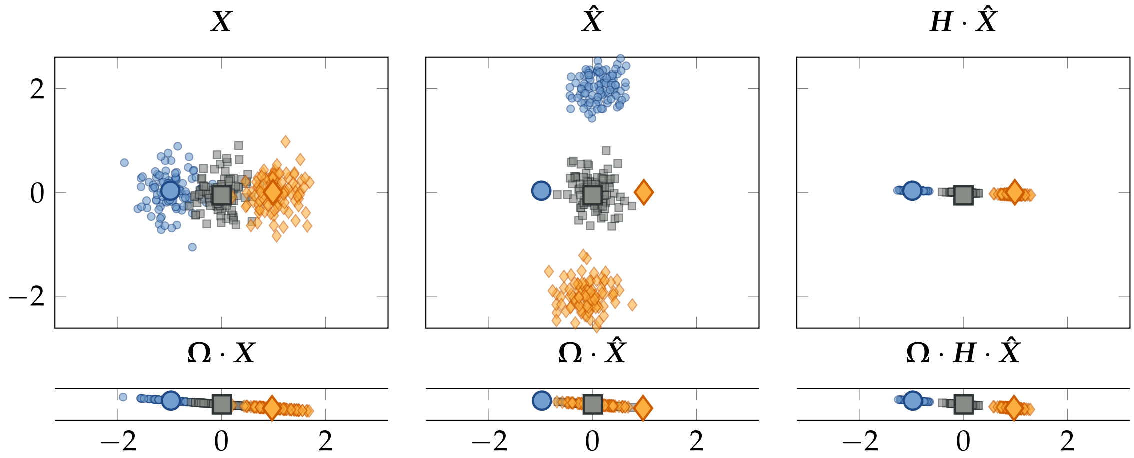


Figure 7.2: A visualization of a GMLVQ model trained on a toy dataset with three Gaussian clusters. Prototypes are highlighted via bigger size. Shape and color indicate the class assignment. Top row: The source space data $\boldsymbol{X}$, target space data $\hat{\boldsymbol{X}}$ and transferred target space data $\boldsymbol{H} \cdot \hat{\boldsymbol{X}}$. Bottom row: The same data after multiplication via the GMLVQ projection matrix $\boldsymbol{\Omega}$.

In order to solve problem 7.2, we need to define the set of possible models $\mathcal{F}$, the loss function E, and the set of possible transfer functions $\mathcal{H}$. In our further investigation, we will restrict $\mathcal{H}$ to the set of linear functions between the target and source space, or, equivalently, the set of matrices $\mathbb{R}^{m \times n}$. This restriction is equivalent to the assumption that the 'true' transfer function h can be approximated by its first-order Taylor expansion with a zero constant term (Saralajew and Villmann 2017). By virtue of this assumption, we guarantee that the transfer function is at most as hard to learn as the original model in case the original model is at least a linear function. In case the linearity assumption fails, we may be better off by learning a new model for the target space directly.

With respect to $\mathcal{F}$ we will consider two classes of models, namely generalized learning vector quantization models and labeled Gaussian mixture models.

Transfer Learning for Generalized Learning Vector Quantization

Recall that generalized matrix learning vector quantization (GMLVQ) is a prototype-based metric learning classifier (refer to Section 2.5.1). Consider the example of the three-class toy dataset shown in Figure 7.2 (top left). One hundred data points for each class are drawn from a two-dimensional normal distribution with standard deviation 0.3 in both dimensions and with respective means $(-1, 0)$, $(0, 0)$, and $(1, 0)$. If we train a GMLVQ model with $K = 3$ prototypes, one per class, we end up with prototypes $\vec{w}_1$, $\vec{w}_2$, and $\vec{w}_3$ close to the class means (see Figure 7.2, top left) and a projection matrix $\boldsymbol{\Omega}$ that discards the second dimension of the data and emphasizes the first dimension (see Figure 7.2, bottom left).

After we have learned the prototypes $w_1, \ldots, w_K$ and the matrix $\boldsymbol{\Omega}$ for the source data, we are now confronted with target data points $\hat{x}_1, \ldots, \hat{x}_N$ with labels $\hat{y}_1, \ldots, \hat{y}_N$ that do not fit our model anymore. In our toy dataset in Figure 7.2, the target data is rotated by just over 90° (middle column). In this case, almost all data from the outer classes would be misclassified, implying a classification error of about 2/3.

Following our transfer learning scheme introduced above, our next step is to learn a transfer matrix $\boldsymbol{H} \in \mathbb{R}^{m \times n}$ that minimizes the GLVQ cost function in Equation 2.28. This can be done by initializing $\boldsymbol{H}$ as the identity matrix (padded with zeros if necessary) and learning $\boldsymbol{H}$ via stochastic gradient descent, including a Frobenius-norm regularization. For the gradient of the GLVQ cost function with respect to $\boldsymbol{H}$ we obtain:

$$\begin{aligned}
&\nabla_{\boldsymbol{H}} E_{\mathrm{GLVQ}}\Big((w_1, z_1), \ldots, (w_K, z_K), (\boldsymbol{H} \cdot \hat{x}_1, \hat{y}_1), \ldots, (\boldsymbol{H} \cdot \hat{x}_N, \hat{y}_N)\Big) \\
=&4 \cdot \boldsymbol{\Omega}^\top \cdot \boldsymbol{\Omega} \cdot \sum_{j=1}^{N} \frac{\Phi'(\mu_j)}{(d_j^+ + d_j^-)^2} \cdot \Big(\boldsymbol{H} \cdot \hat{x}_j \cdot \hat{x}_j^\top \cdot (d_j^- - d_j^+) - (d_j^- \cdot \vec{w}_j^+ - d_j^+ \cdot \vec{w}_j^-) \cdot \hat{x}_j^\top\Big)
\end{aligned} \tag{7.3}$$

where $\vec{w}_j^+$ is the closest prototype to the jth transferred target data point $\boldsymbol{H} \cdot \hat{x}_j$ with the same label, where $\vec{w}_j^-$ is the closest prototype to the jth transferred target data point $\boldsymbol{H} \cdot \hat{x}_j$ with a different label label, where $d_j^+ = \|\boldsymbol{\Omega} \cdot \boldsymbol{H} \cdot \hat{x}_j - \boldsymbol{\Omega} \cdot \vec{w}_j^+\|^2$, where $d_j^- = \|\boldsymbol{\Omega} \cdot \boldsymbol{H} \cdot \hat{x}_j - \boldsymbol{\Omega} \cdot \vec{w}_j^-\|^2$, where $\mu_j = (d_j^+ - d_j^-)/(d_j^+ + d_j^-)$, and where Φ is some differentiable, monotonously increasing function. To this gradient, we add the gradient of the Frobenius-norm regularization $\nabla_{\boldsymbol{H}} \lambda \cdot \|\boldsymbol{H}\|_F^2 = \lambda \cdot 2 \cdot \boldsymbol{H}$, where $\lambda \in \mathbb{R}$ is a small, positive regularization constant. We can thus perform transfer learning using any gradient-based optimization method of choice, such as stochastic gradient descent or limited-memory BFGS (Liu and Nocedal 1989).

For our example, this optimization does indeed rotate our target data such that our model applies again (see Figure 7.2, right column). Also note that the rotation flattens the data in the second dimension because this dimension of the data is not relevant for classification.

A challenge in transfer learning via the GLVQ cost function is that we require numeric solvers to optimize the cost function E_{GLVQ}. These solvers may be computationally expensive. As a fast alternative, we also provide a fast expectation maximization approach for transfer learning based on labeled Gaussian mixture models.

Transfer Learning for Labeled Gaussian Mixture Models

Recall that a lGMM is a supervised, generative model that is trained via an expectation maximization scheme (refer to Section 2.5.2). After having learned a lGMM for our source data, we are confronted with disturbed target space data $\hat{x}_1, \ldots, \hat{x}_N$ with labels $\hat{y}_1, \ldots, \hat{y}_N$, for which the likelihood according to our model is low. As an example, consider Figure 7.3. If we have trained a model for the source data shown on the top left, the likelihood of target space data shown on the top right according to this source model would be low.

Following our transfer learning formalization in Equation 7.2, we are now looking for a linear transformation $\boldsymbol{H} \in \mathbb{R}^{m \times n}$ that minimizes the negative log-likelihood of our target space data, that is:

$$\min_{\boldsymbol{H} \in \mathbb{R}^{m \times n}} \sum_{j=1}^{N} -\log\Big[p(\boldsymbol{H} \cdot \hat{x}_j, \hat{y}_j)\Big] \tag{7.4}$$

Note that the negative log-likelihood may be non-convex with respect to $\boldsymbol{H}$, such as in our example in Figure 7.3 (bottom left). However, it is worth noting that more label information may make the problem convex. For example, if we had instead considered

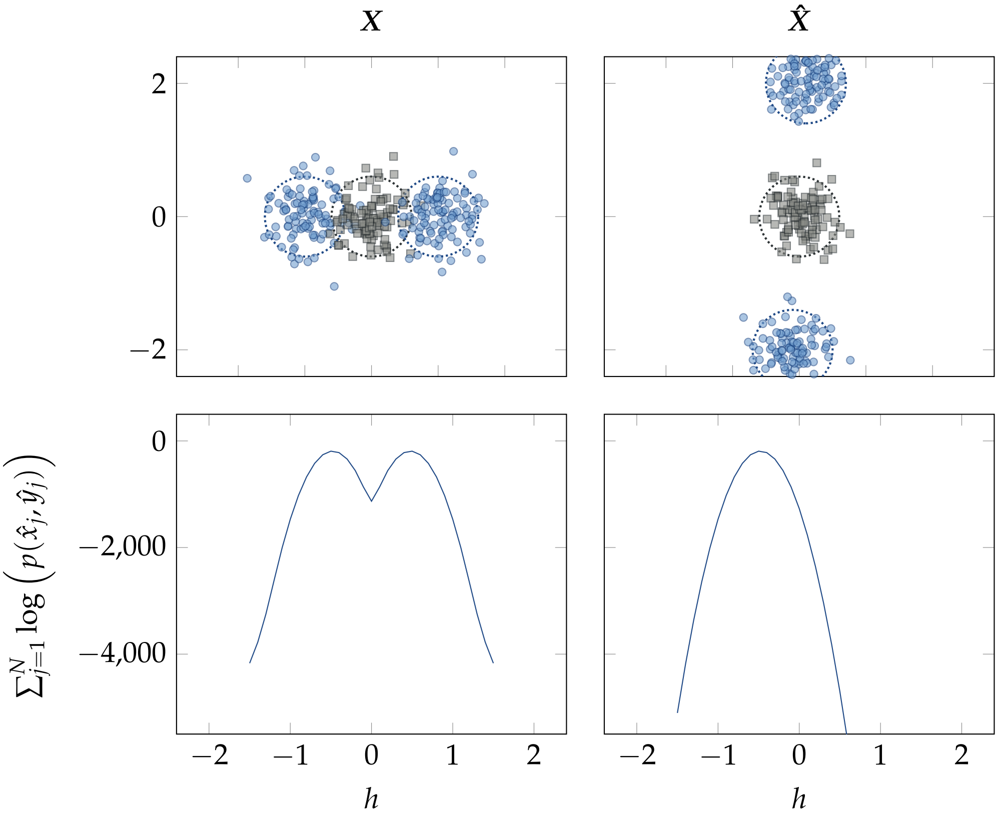


Figure 7.3: Top left and right: A transfer learning problem with an ambiguous transfer mapping. The data has been generated as in Figure 7.2 with the difference that the data clusters to the left and right in the left figure and the top and bottom in the right figure share the same label. Bottom left: The log-likelihood according to Equation 7.4 of transfer mappings of the form $\boldsymbol{H} = (0, h; 0, 0)$. h is shown on the x-axis, while the log-likelihood is shown on the y-axis. As can be seen, the log-likelihood has two local optima at $h \approx \pm 0.5$. Bottom right: The log-likelihood if the data is labeled as in Figure 7.2. In this case, there is only one global optimum at $h \approx -0.5$.

the labeling from Figure 7.2, the negative log-likelihood would be convex (see Figure 7.3, bottom right).

As in the case of lGMM learning, we can find a local optimum of the negative log-likelihood via an expectation maximization scheme as suggested by Dempster, Laird, and Rubin (1977). In particular, our expectation step and maximization step take the following form.

Expectation step: For each data point $\hat{x}_j$, we compute the posterior for the Gaussian that has generated the data point according to Bayes' rule.

$$\gamma_{k|j} := P(k|\boldsymbol{H} \cdot \hat{x}_j, \hat{y}_j) = \frac{P(\hat{y}_j|k) \cdot p(\boldsymbol{H} \cdot \hat{x}_j|k) \cdot P(k)}{\sum_{k'=1}^{K} P(\hat{y}_j|k') \cdot p(\boldsymbol{H} \cdot \hat{x}_j|k') \cdot P(k')} \tag{7.5}$$

Maximization step: Assuming fixed posterior $\gamma_{k|j}$ we minimize the negative expected log-likelihood $\hat{Q}$ of the data with respect to $\boldsymbol{H}$. $\hat{Q}$ has the following form:

$$\begin{aligned}
\hat{Q} = &- \sum_{j=1}^{N} \sum_{k=1}^{K} \gamma_{k|j} \cdot \log\left[p(\hat{y}_j, \boldsymbol{H} \cdot \hat{x}_j, k)\right] \qquad (7.6)\\
=& \sum_{j=1}^{N} \sum_{k=1}^{K} \gamma_{k|j} \cdot \Big(- \log\left[P(\hat{y}_j|k)\right] - \log\left[p(\boldsymbol{H} \cdot \hat{x}_j|k)\right] - \log\left[P(k)\right] \Big)\\
=& \sum_{j=1}^{N} \sum_{k=1}^{K} \gamma_{k|j} \cdot \Big(- \log\left[P(\hat{y}_j|k)\right] - \log\left[P(k)\right] - \frac{1}{2} \log\left[\det(\boldsymbol{\Lambda}_k)\right]\\
&+ \frac{m}{2} \cdot \log\left[2 \cdot \pi\right] + \frac{1}{2} \cdot (\vec{\mu}_k - \boldsymbol{H} \cdot \hat{x}_j)^\top \cdot \boldsymbol{\Lambda}_k \cdot (\vec{\mu}_k - \boldsymbol{H} \cdot \hat{x}_j) \Big)
\end{aligned}$$

We can show that $\hat{Q}$ is convex with respect to $\boldsymbol{H}$ and that we obtain a closed-form solution in case all Gaussians of our lGMM share the same precision matrix.

Theorem 7.1. *Under the assumption of fixed $\gamma_{k|j}$, $\hat{Q}$ (Equation 7.6) is convex with respect to $\boldsymbol{H}$.*

Further, if our source model is a slGMM, $\hat{Q}$ takes a unique optimum at $\boldsymbol{H} = \boldsymbol{W} \cdot \boldsymbol{\Gamma} \cdot \hat{\boldsymbol{X}}^\dagger$, where $\hat{\boldsymbol{X}} := (\hat{x}_1, \ldots, \hat{x}_N) \in \mathbb{R}^{n \times N}$, $\boldsymbol{W} := (\vec{\mu}_1, \ldots, \vec{\mu}_K) \in \mathbb{R}^{m \times K}$, $\boldsymbol{\Gamma}$ denotes the $K \times N$-dimensional matrix with the entries $\boldsymbol{\Gamma}_{k,j} = \gamma_{k|j}$, and $\hat{\boldsymbol{X}}^\dagger$ denotes the Moore-Penrose Pseudo-Inverse of $\hat{\boldsymbol{X}}$.

Proof. Refer to Appendix A.18. □

Algorithm 7.1 shows how we can learn $\boldsymbol{H}$ based on this theorem.

Regarding computational complexity, we observe that we need to compute $\mathcal{O}(K \cdot N)$ posterior values $\gamma_{k|j}$ in each iteration, each of which takes constant time if we regard the source space dimensionality m and the target spaces dimensionality n as constants. If the input model is a slGMM, the maximization step requires $\mathcal{O}(K \cdot N)$ operations to evaluate Equation A.78. Otherwise, we need to minimize $\hat{Q}$ with respect to $\boldsymbol{H}$ via a gradient-based solver, where each gradient computation according to Equation A.76 requires $\mathcal{O}(K \cdot N)$ operations. As $\hat{Q}$ is convex with respect to $\boldsymbol{H}$, we can assume that the optimum can be found by evaluating the gradient only a constant number of times. Overall, we obtain a complexity of $\mathcal{O}(T \cdot K \cdot N)$ where T is the number of iterations required until $|E - E'| < \epsilon$. Because $\hat{Q}$ is bounded below and is guaranteed to decrease in every step by at least ϵ, T needs to be finite. However, an exact estimate is challenging. For the special case that every Gaussian generates only data with a single label and there is only one Gaussian for each label, we can infer that $\gamma_{k|j}$ is independent of $\boldsymbol{H}$ because $\gamma_{k|j}$ is 1 if y_j is the label of the kth Gaussian and 0 otherwise. Therefore, the global optimum is already found in the first optimization step and will not change subsequently. Thus, the error in the second iteration will be the same, which implies $|E - E'| = 0 < \epsilon$, such that the algorithm terminates. As such, T is at least 2 and increases with the ambiguity in assigning data points to Gaussians. One way to reduce the amount of ambiguity is to use lGMMs based on LVQ models because these models feature a crisp assignment of Gaussians to labels leading to a lot of constant zeros entries for $\gamma_{k|j}$. Alternatively, one can impose a strict limit on T.

This concludes our description of transfer learning schemes. In the next section, we evaluate our transfer learning algorithms experimentally.

Algorithm 7.1 An expectation maximization algorithm for linear supervised transfer learning on a labeled Gaussian Mixture Model (lGMM) with K Gaussians. As input, the algorithm receives an lGMM, a set of labeled target space data points $(\hat{x}_j, \hat{y}_j)$, and an error threshold ϵ. The transfer matrix is initialized as $\boldsymbol{I}^{m \times n}$, which denotes the $\min\{m, n\}$-dimensional identity matrix, padded with zeros where necessary.

$E \leftarrow \infty$, $\boldsymbol{H} \leftarrow \boldsymbol{I}^{m \times n}$
while true **do**
    for $k \in \{1, \ldots, K\}$ **do**
        for $j \in \{1, \ldots, N\}$ **do**
            Compute $\gamma_{k|j}$ according to Equation 7.5. ▷ Expectation
        end for
    end for
    if the input model is a slGMM **then**
        $\boldsymbol{H} \leftarrow \boldsymbol{W} \cdot \boldsymbol{\Gamma} \cdot \hat{\boldsymbol{X}}^{\dagger}$ (Theorem 7.1). ▷ Maximization
    else
        Minimize $\hat{Q}$ with respect to $\boldsymbol{H}$ using gradient A.76. ▷ Maximization
    end if
    $E' \leftarrow \hat{Q}(\boldsymbol{H})$.
    if $|E - E'| < \epsilon$ **then**
        return $\boldsymbol{H}$.
    end if
    $E \leftarrow E'$.
end while

7.3 EXPERIMENTS

In this section, we evaluate our proposed transfer learning algorithms experimentally on two balanced, artificial, three-class datasets. For both datasets, we first train a LVQ classifier model on the source data. Then, we perform both GLVQ transfer learning (GLVQ) and expectation maximization transfer learning (EM). We further compare with the following baseline methods.

- naively applying the source space model to the target data (naive),
- re-training a new model solely on the target space data (retrain),
- training an adaptive support vector machine (a-SVM, J. Yang, Yan, and Hauptmann 2007),
- transfer learning via asymmetric regularized cross-domain transformation (ARC-t, Kulis, Saenko, and Darrell 2011), and
- transfer learning via heterogeneous feature augmentation (HFA, Duan, Xu, and I. Tsang 2012).

Note that, to ensure a fair comparison, ARC-t and HFA only received the LVQ prototype positions as source data for training and the same target space data as the other transfer learning methods.

Our evaluation measure is the mean classification error on the test dataset across crossvalidation folds (10 folds in the first, 30 folds in the second dataset). We analyze the

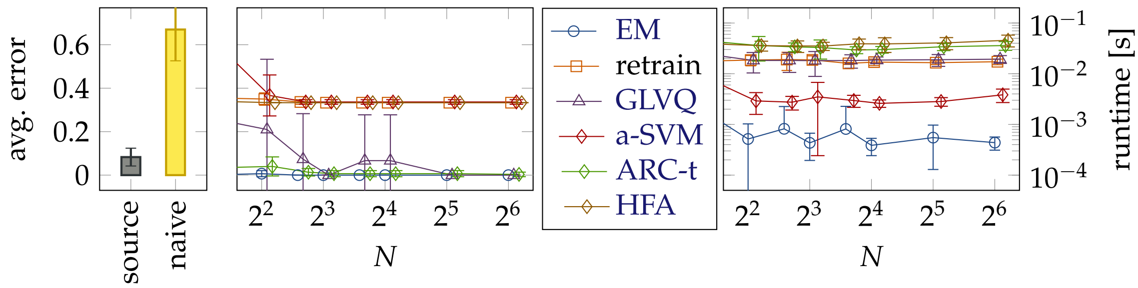


Figure 7.4: Mean classification error (left) and mean runtime (right) in a ten-fold crossvalidation on the toy dataset shown in Figure 7.2 with the left and middle class being available for transfer learning. The x-axis indicates the number of available target space training data points N (in log scaling) while the y-axis displays the mean classification error (left) or the runtime (right, log scale). Error bars indicate the standard deviation.

average classification error on unseen target data across multiple sizes of the target space training dataset. In all cases, we only use training data from the target space from the first two classes. As such, the classification error of a newly trained model necessarily stays above 1/3. Our hypothesis is that transfer learning can achieve a considerably better error than a newly trained model (H1), and that transfer learning generally achieves a better error with less data (H2) because it only needs to learn a simple linear transformation compared to a non-linear classification model.

We also report the runtime of all transfer learning approaches running on a Intel Core i7-7700 HQ CPU. We expect that our proposed expectation maximization (EM) transfer learning approach will be considerably faster compared to re-training a new model, the a-SVM and GLVQ transfer learning (H3), because it involves only a convex optimization for a linear transformation matrix with fairly few parameters.

For all significance tests we employ a one-sided Wilcoxon signed rank test.

Data Set 1

Our first dataset is the two-dimensional toy dataset shown in Figure 7.2. The data is generated via a labeled Gaussian mixture model with one component for each of the three classes with means $\vec{\mu}_1 = (-1, 0)$, $\vec{\mu}_2 = (0, 0)$, and $\vec{\mu}_3 = (1, 0)$ and shared covariance matrix $\mathbf{\Lambda}^{-1} = 0.3^2 \cdot \boldsymbol{I}^2$. The target data is generated with a similar model but with the means set to $\vec{\mu}_1 = (-0.1, -2)$, $\vec{\mu}_2 = (0, 0)$, and $\vec{\mu}_3 = (0.1, 2)$. In both source and target space we generate 100 data points per class.

On the source space data, we train a GMLVQ model with one prototype per class, as shown in Figure 7.2 (left column). The GMLVQ model correctly identifies the x-axis as discriminative and learns a linear projection matrix $\mathbf{\Omega}$ that disregards the y-axis (see Figure 7.2, left bottom). However, for the target space data this model does not apply because all target space data points are now close to the prototype for the center class (see Figure 7.2, middle column), resulting in 2/3 misclassifications for a naive application of the source space model (Figure 7.4, left). As such, we require a transfer learning approach to make our data fit to the model again.

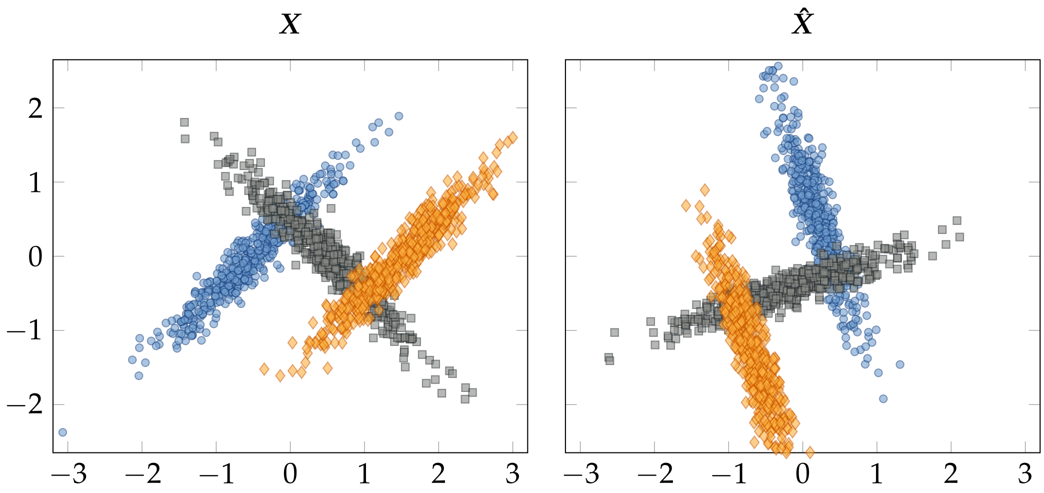


Figure 7.5: The source (left) and target data (right) for the second experiment.

For transfer learning, we only use target space data from the first and second class (blue circles and black squares), but not the third class (orange diamonds). Figure 7.4 (left) displays the mean classification error in a ten-fold crossvalidation for each approach. We observe that the EM transfer learning scheme performs best and achieves a consistent classification error of below 1%. ARC-t achieves the same level of performance if at least 4 data points are available and GLVQ transfer learning requires at least $2^6 = 64$ data points to achieve the same level of performance. By constrast, the a-SVM, HFA, and retraining methods can not classify the third class (orange diamonds) correctly without having training data for that class such that their error stays above 1/3. These results lend support for both H1 and H2.

Figure 7.4 (right) displays runtime results. We observe that our proposed transfer learning scheme is roughly 10 times faster compared to a-SVM and roughly 30 times faster compared to GLVQ transfer learning and learning a new GMLVQ model (see Figure 7.4, right), supporting H3. Interestingly, we also observe that there is little if any runtime advantage in performing GLVQ transfer learning compared to learning a new GMLVQ model.

Data Set 2

Our second artificial dataset illustrates the advantage of individual precision matrices in cases of strong class overlap. The dataset is inspired by the *cigars* dataset by (Schneider, Biehl, and Hammer 2009a) and consists of 1000 data points for each of the three classes, which are generated via a lGMM with one Gaussian per class, with means at $\vec{\mu}_1 = (-0.5, 0)$, $\vec{\mu}_2 = (0.5, 0)$, and $\vec{\mu}_3 = (1.5, 0)$, and with covariance matrices

$$\mathbf{\Lambda}_1^{-1} = \mathbf{\Lambda}_3^{-1} = \begin{pmatrix} 0.485 & 0.36 \\ 0.36 & 0.485 \end{pmatrix}, \qquad \text{and} \quad \mathbf{\Lambda}_2^{-1} = \begin{pmatrix} 0.485 & -0.36 \\ -0.36 & 0.485 \end{pmatrix}$$

The target data is generated from the same distribution, with the model being rotated by 90° (see Figure 7.5).

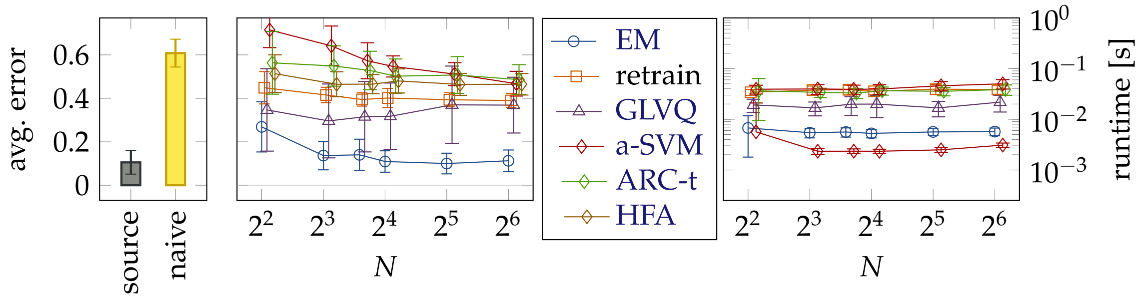


Figure 7.6: Mean classification error (left, middle) and mean runtime (right) in a 30-fold crossvalidation on the cigars dataset shown in Figure 7.5 with the left and middle class being available for transfer learning. The x-axis indicates the number of available target space training data points N (in log scaling) while the y-axis displays the mean classification (left middle) or the runtime (right, log scale). Error bars indicate the standard deviation.

As a source space model, we employ GMLVQ and LGMLVQ with one prototype per class. The challenge for GMLVQ in this dataset is that there is not one consistent discriminative direction for the whole dataset. While the direction is the same for the first and second class (blue circles and orange diamonds), it differs for the second class (black squares).

LGMLVQ can account for this by learning different precision matrices for each prototype. Accordingly, we observe that the source classification error for GMLVQ is much higher compared to LGMLVQ with 21.07% versus 10.53% on average.

For transfer learning, we only use target space data from the first and second class (blue circles and black squares), but not the third class (orange diamonds), and we use the LGMLVQ model as basis for transfer learning for all methods. Figure 7.6 (left) displays the mean classification error in a 30-fold crossvalidation for each approach. In line with H1 and H2, we observe that EM transfer learning performs better compared to all other reference methods. Indeed, the difference is significant if at least 8 training data points are available for transfer learning ($p < 10^{-3}$). The relatively weak performance of both a-SVM and HFA is likely explained by the missing class. In contrast to the first data set, however, ARC-t also underperforms on this data set, which is likely due to the fact that ARC-t can not take local metric information into account.

Also note that the error after EM transfer learning with 64 data points is close to the performance of the source model (11.27% versus 10.53% on average) and is statistically indistinguishable. Interestingly, GLVQ transfer learning performs about as well as a newly trained model, which is likely due to unfortunate local optima.

Figure 7.4 (right) displays runtime results. In this regard we notice the overhead implied by a numeric solution for $\boldsymbol{H}$ compared of a closed-form one. The runtime of our EM transfer learning approach is now about 2 times slower compared to the a-SVM but still around 3 times faster compared to GLVQ transfer learning, 6 times faster compared to ARC-t, 8 times faster compared to HFA, and about 5 times faster compared to learning a new LGMLVQ model on the target data. Thus, H3 is partially supported.

7.4 CONCLUSION

In this chapter, we have introduced a novel approach to transfer learning, namely learning an explicit linear mapping between target and source space such that the mapped target data fits to a learned source space model. We have developed two transfer learning algorithms based on this general setup, namely gradient-based learning on the generalized learning vector quantization (GLVQ) cost function, and expectation maximization (EM) transfer learning to maximize the likelihood of the mapped target space data according to a labeled Gaussian mixture model for the source space.

We have evaluated both approaches on two artificial datasets. In both cases, EM transfer learning identified linear transfer mappings which significantly improved the classification accuracy compared to naively applying the source space model to the target data, learning a new model only on target space data, or performing transfer learning via a-SVM, ARC-t, HFA, or GLVQ approaches. We also observed that EM transfer learning was generally faster compared to all alternatives, especially in case of models with shared precision matrices.

In summary, EM transfer learning is a simple, data- and time-efficient alternative compared to re-learning a new classification model, as well as our tested alternative domain adaptation and transfer learning approaches. These properties make our transfer learning scheme ideal for the domain of bionic hand prostheses, which we will cover in the next chapter.

8 APPLICATION TO BIONIC HAND PROSTHESES

Summary: Research on hand prostheses has shown impressive progress in recent years with bionic prostheses that enable amputees to achieve comparable hand function to able-bodied people in lab studies. Unfortunately, these promising results are limited to the lab because prosthetic user interfaces tend to break down under everyday disturbances. Electrode shifts are particularly challenging because they disturb the user's control signal abruptly and cause a high rate of misclassifications.

In this chapter, we apply the transfer learning algorithms from Chapter 7 to counteract electrode shifts. In an experimental evaluation on two real-world datasets we show that as little as a few seconds of recorded training data from an incomplete set of motions are sufficient to adapt a user interface to disturbed data. As such, transfer learning requires less data, computation time, and class coverage compared to all tested baselines.

Publications: This chapter is based on the following publications.

- Prahm, Cosima et al. (2016). "Transfer Learning for Rapid Re-calibration of a Myoelectric Prosthesis after Electrode Shift". In: *Proceedings of the 3rd International Conference on NeuroRehabilitation (ICNR 2016)*. (Segovia, Spain). Ed. by Jaime Ibáñez et al. Vol. 15. Converging Clinical and Engineering Research on Neurorehabilitation II. Biosystems & Biorobotics. **Runner-Up for Best Student Paper Award**. Springer, pp. 153–157. DOI: 10.1007/978-3-319-46669-9_28.
- Paaßen, Benjamin et al. (2018). "Expectation maximization transfer learning and its application for bionic hand prostheses". In: *Neurocomputing* 298, pp. 122–133. DOI: 10.1016/j.neucom.2017.11.072.

The human hand is a tremendously versatile and precise tool that we use for a wide range of our everyday actions (Napier 1956). As such, losing a hand can have dramatic impact on life quality, including the ability to work (Biddiss and Chau 2007; Raichle et al. 2008; Ziegler-Graham et al. 2008). Over 40,000 people in the US alone are classified as having suffered major upper limb loss, highlighting the relevance of the problem (Ziegler-Graham et al. 2008). Bionic hand prostheses promise to regain lost hand function by executing desired hand motions with a robotic hand attached to the patient's arm stump (Farina et al. 2014). Indeed, amputees have achieved similar performance as able-bodied participants in a variety of lab studies (Hahne, Dähne, et al. 2015; Jiang et al. 2014). Unfortunately, these results are still limited to constrained laboratory settings because prostheses tend to not work as desired under everyday disturbances, leading users to be less confident in using their prosthesis or abandoning the prosthesis altogether (Biddiss and Chau 2007; Farina et al. 2014; Hargrove, Englehart, and Hudgins 2008; Khushaba et al. 2014; Young, Hargrove, and Kuiken 2011). As such, we sorely need a mechanism to make bionic prostheses more robust in patients' everyday lives.

The reason bionic prostheses are so brittle is their user interface. The state-of-the-art in prosthesis control is to apply a small number of electromyography (EMG) electrodes to the patient's stump and to infer the desired motion from the EMG signal of those electrodes (Farina et al. 2014). More specifically, the user interface is based on some model

f that infers for every time t the desired motion y_t from the EMG signal $\vec{x}_t$. Training data is generated by letting patients execute precisely timed motions with their phantom hand, which triggers activity of the residual muscles in the patients' stump, which is in turn reflected in the EMG signal $\vec{x}_t$. With sufficient training and a sufficient number of electrodes, patients learn to generate a characteristic EMG pattern for each motion such that the model f can be trained via machine learning. In each time step, the model's prediction is then forwarded to the bionic prosthesis itself, which executes the motion with a time delay below 200ms (Farina et al. 2014).

Unfortunately, patients' EMG signals tend to be non-stationary. For example, when donning and doffing their prostheses, patients tend to apply the EMG electrodes slightly differently, or electrodes may shift due to external force or soft materials (Farina et al. 2014; Hargrove, Englehart, and Hudgins 2008; Khushaba et al. 2014; Young, Hargrove, and Kuiken 2011). In all these cases, the signal $\vec{x}_t$ is disturbed such that the model f misclassifies the signal, i.e. $f(\vec{x}_t) \neq y_t$ (also refer to Figure 8.1).

Several approaches in the past have attempted to address this issue. In particular, Hargrove, Englehart, and Hudgins (2008) have proposed to record training data in all plausible shift conditions to achieve a model f that is invariant against shifts. Additionally, various authors have suggested alternatives to time-domain features which are supposedly more shift-invariant, such as auto-regressive features (Hargrove, Englehart, and Hudgins 2008; Young, Hargrove, and Kuiken 2012) or spectral features (Khushaba et al. 2014). While all these approaches improve classification accuracy, they are limited to cases of virtual concept drift. In case of real concept drift, there exists at least one region of conflict where patterns of one class in the source data overlap with patterns of a different class in the target data. In invariant feature representations, this region of conflict has to be mapped to one class such that either the source or the target space data in this region are necessarily misclassified.

A different route is to improve the input signal itself by virtue of alternative sensors. For example, Muceli, Jiang, and Farina (2014) and L. Pan et al. (2015) propose high-density electrode grids in conjunction with alternative features to improve robustness and Hahne, Farina, et al. (2016), Ortiz-Catalan, Brånemark, Håkansson, and Delbeke (2012), and Pasquina et al. (2015) developed implantable sensors that are not affected by electrode shifts. Unfortunately, neither of these advanced sensor technologies are likely to be featured in commercially available prostheses in the near future (Farina et al. 2014).

As such, it is unlikely that we will be able to completely prevent disturbances to the input signal $\vec{x}_i$. However, we may still be able to adapt our user interface to the disturbed situation using only little new training data. For example, Vidovic et al. (2015) adapt their model to changed means and covariances in the disturbed data. While this approach is certainly viable, it fails to exploit the structured nature of the disturbance. We argue that electrode shifts are structurally simple and that learning the electrode shift explicitly is advantageous compared to adapting a potentially complicated model. As such, our transfer learning scheme from Chapter 7 appears as a perfect fit for the electrode shift scenario. Not only are we likely to save data and computation time, we can also perform learning using only few training motions. Reducing the number of precisely timed motions a patient has to record is a critical advantage because it makes the recording process easier for patients and reduces the likelihood of label noise.

The main contribution of this chapter is to empirically evaluate transfer learning on two real-world datasets of EMG data. We show that transfer learning can improve

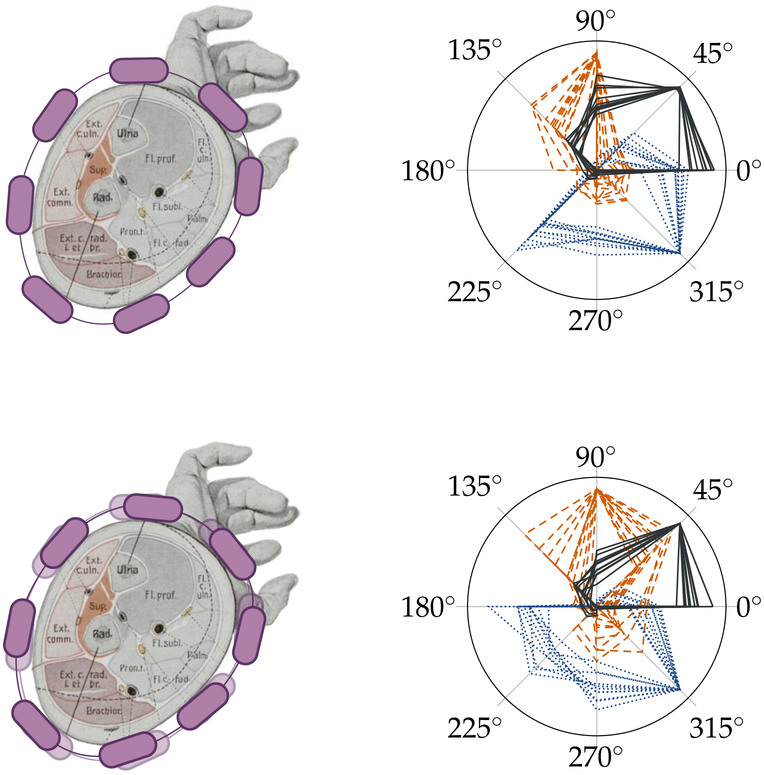


Figure 8.1: An illustration of electrode shifts in electromyographic (EMG) data. Top left: A grid of eight EMG electrodes placed around the forearm of a user. Top right: Example EMG signals from an eight-electrode EMG recording for two different hand motions (dashed and dotted lines) as well as resting (solid lines). Bottom left: The electrode grid is shifted around the forearm (electrode shift). Bottom right: Another set of EMG signals from a shifted eight-electrode EMG recording for the same set of hand motions (dashed and dotted lines) as well as resting (solid lines). Due to the shifted signal, a model trained on the source data (top right) may misclassify shifted data (bottom right).

classification accuracy beyond a disturbed model using less data, fewer classes, and less computational time compared to learning a new model.

8.1 EXPERIMENTS

We evaluate the transfer learning schemes from Chapter 7 on two real-world datasets of EMG data. In both cases, the data was recorded by instructing able-bodied participants to execute a sequence of pre-defined hand motions at pre-defined times and recording EMG data during the execution of those motion sequences. We considered three degrees of freedom (DoFs) that are key to prosthesis control, namely a wrist rotation DoF (pronation and supination), a wrist pitch DoF (flexion and extension), and a finger opening DoF (finger spread and fist).

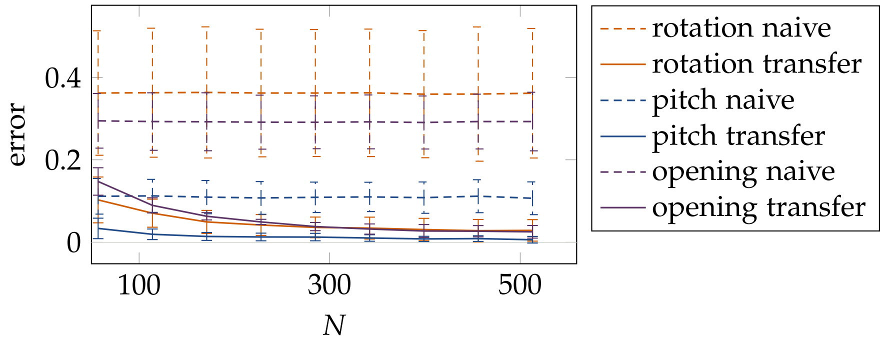


Figure 8.2: The experimental results for the first EMG dataset. The x axis shows the number of target space data points used for training the transfer matrix $\boldsymbol{H}$. The y axis shows the average error and standard deviation across participants. Different DoFs are differentiated by color. Dashed lines show the naive error in all three degrees of freedom, solid lines the transfer learning error.

Data Set 1

In the first dataset, we evaluate whether transfer learning can improve accuracy beyond the baseline of naively applying the source space model. We recorded data from four able-bodied subjects who each executed a sequence of all atomic motions in our three DoFs, as well as all pairwise combinations, resulting in nineteen motions overall. We recorded each movement for five seconds, followed by two seconds of rest. To simulate disturbance we moved the EMG electrodes by 8mm transversally around the forearm and repeated the protocol. This work was approved by the ethics commitee of the Medical University of Vienna (#1301/2015).

As recording device we utilized an eight channel Ottobock Healthcare electrode array (13E200) at 1000Hz sampling rate placed equidistantly in a ring around the forearm (see figure 8.1, top left). We preprocessed the data using a 90Hz to 450Hz band pass filter and computed the 17 standard features of the BioPatRec suite (Ortiz-Catalan, Brånemark, and Håkansson 2013) on time windows of 100ms with 50ms overlap, combined with the log-variance as suggested by Hahne, Biebmann, et al. (2014).

We coded the motion at time t as a three-dimensional vector $\vec{y}_t \in \{-1, 0, 1\}^3$ where $y_{t,l} = -1$ denotes motion in negative direction in the lth DoF, where $y_{t,l} = 1$ denotes motion in positive direction, and where $y_{t,l} = 0$ denotes no motion. For example, $\vec{y}_t = (0, 0, 0)^\top$ codes resting, i.e. no motion in any DoF, $\vec{y}_t = (1, 0, 0)^\top$ denotes supination, and $\vec{y}_t = (0, -1, 1)^\top$ denotes extension combined with a fist. As a classifier architecture we trained three GMLVQ models, one for each degree of freedom, with five prototypes per class. We trained each model five times using random initializations and used the one with highest training accuracy.

For the experiment, we distributed the data randomly into 10 crossvalidation folds, both for source and target space data. In each fold, we used the source data to train the GMLVQ models and we used $N \in [50, 512]$ randomly selected samples from the target data to train the transfer matrix $\boldsymbol{H}$. As algorithm for transfer learning we used the gradient-based GLVQ scheme from Section 7.2.

Figure 8.2 displays the average classification error for the source model on the target space (dashed lines), and after transfer learning (solid lines). On the source data, the

GMLVQ models achieved below 1% test error consistently. On the target data, the performance of the source model dropped to 36% for the rotation DoF, to 11% for the flexion/extension DoF, and to 29% for the open/close DoF. After transfer learning, even with as little as $N = 50$ training samples, classification error was notably lower at about 10%, 3%, and 15% respectively. With more samples, this dropped further to 3%, 1%, and 3% respectively for $N = 350$ samples. The difference between the error before and after transfer learning was highly significant for all participants and all models ($p < 0.01$ using the Wilcoxon rank sum test and Bonferroni correction).

These results provide a proof of concept that transfer learning can indeed enhance accuracy. However, for a fair comparison, we also need to show that our transfer learning scheme outperforms a newly trained model and alternative transfer learning approaches. To this end, we evaluate a second, larger dataset.

Data Set 2

Our second dataset contains EMG recordings of 10 able-bodied participants who performed all six atomic hand motions in our three DoFs as well as resting. Each participant performed 15 to 35 repetitions (236 repetitions in total) of these seven motions. Each motion lasted 3 seconds from which the first and the last second were cut to avoid label noise, leaving 1 seconds of each motion for analysis. The experiments are in accordance with the declaration of Helsinki and approved by the ethics commission of the Medical University of Göttingen. Further details on the experimental protocol are provided by Hahne, Graimann, and Müller (2012).

The EMG data was recorded with a high-density grid of 96 EMG electrodes with 8 mm inter-electrode distance, located around the forearm at 1/3 of the distance from elbow to wrist. The raw EMG signal was filtered with a low pass (500 Hz, fourth-order Butterworth), a high pass (20 Hz, fourth-order Butterworth), and a band stop filter ($45 - 55$ Hz, second-order Butterworth) to remove noise, movement artefacts, and power line interferences respectively. As features, we computed the logarithm of the signal variance for each electrode, computed on non-overlapping time windows of $100ms$ length. Thus, depending on the number of runs, 1925 to 3255 samples were available per participant, balanced for all classes (for the participant with the fewest runs we obtained 275 samples per class, for the participant with the most runs 465 samples per class).

Since high-density EMG recordings are not common in prosthetic hardware (Farina et al. 2014), we only used recordings from a subset of 8 equidistant electrodes located on a ring around the forearm (see figure 8.1, top left). In order to obtain disturbed target data, we simulated an electrode shift by utilizing eight different electrodes, located one step within the array ($8mm$) transversally to the forearm (see figure 8.1, bottom left).

For this experiment, we coded motions as a scalar label $y_t \in \{1, \ldots, 7\}$ and trained a single model for classification.

Model Selection: In a pre-analysis, we evaluated multiple classifiers on the source data. In particular, we compared a generalized matrix learning vector quantization (GMLVQ), a local generalized matrix learning vector quantization (LGMLVQ), a labeled Gaussian Mixture Model with shared precision matrix (slGMM), a labeled Gaussian Mixture Model (lGMM) with individual precision matrices, a slGMM with GMLVQ initialization (GMLVQ + slGMM), and a lGMM with LGMLVQ initialization (LGMLVQ + lGMM). The

Table 8.1: Mean classification test error and standard deviation on the source space data across all runs on the second dataset. The different classification models are listed on the x axis, the number of prototypes / Gaussians K per class for the model on the y axis. The best results in each row are highlighted via bold print.

K	GMLVQ	LGMLVQ	slGMM	lGMM	GMLVQ + slGMM	LGMLVQ + lGMM
1	$6.7 \pm 7.1\%$	$7.0 \pm 7.2\%$	$\mathbf{5.9 \pm 6.7}\%$	$6.7 \pm 7.1\%$	$\mathbf{5.9 \pm 6.7}\%$	$6.7 \pm 7.1\%$
2	$6.5 \pm 6.8\%$	$8.4 \pm 7.9\%$	$5.8 \pm 6.5\%$	$6.5 \pm 6.6\%$	$\mathbf{5.6 \pm 6.2}\%$	$9.9 \pm 8.0\%$
3	$6.7 \pm 7.3\%$	$9.3 \pm 8.5\%$	$6.1 \pm 6.7\%$	$7.1 \pm 7.7\%$	$\mathbf{5.7 \pm 6.4}\%$	$9.6 \pm 8.7\%$
4	$6.5 \pm 7.4\%$	$9.9 \pm 8.9\%$	$\mathbf{5.9 \pm 6.6}\%$	$7.8 \pm 7.4\%$	$\mathbf{5.9 \pm 6.7}\%$	$11.9 \pm 12.8\%$
5	$6.4 \pm 7.4\%$	$10.1 \pm 8.8\%$	$\mathbf{5.9 \pm 6.7}\%$	$7.8 \pm 7.3\%$	$\mathbf{5.9 \pm 6.4}\%$	$23.1 \pm 28.9\%$

Gaussian mixture models were trained with expectation maximization while restricting the standard deviation in each dimension to be at least 0.001, as recommended by Barber (2012). For each of the methods, we varied the number of prototypes/Gaussians K per class from 1 to 5. In our analysis, we iterate over all 236 runs in the dataset and treat the data of the current run as test data, yielding a leave-one-out crossvalidation over the 236 runs. As training data we utilize a random sample of 175 data points, balanced over the classes, drawn from the remaining runs of the same subject. We train each model starting from 5 random initializations and select the model with the lowest training error. For this model, we then record the classification error on the test data.

The results of our pre-experiment are shown in Table 8.1. As can be seen, a slGMM with GMLVQ initialization consistently achieves the best results. The difference in error is significant compared to GMLVQ ($p < 0.05$), LGMLVQ ($p < 0.001$), lGMM ($p < 0.001$), and lGMM with LGMLVQ initialization ($p < 0.001$; all p-values stem from one-sided Wilcoxon signed rank tests). The difference to a slGMM without GMLVQ initialization is insignificant. Regarding the number of prototypes, we obtain the best results for $K = 2$ prototypes per class, although the error difference to other values for K is insignificant. For the main analysis, we select the overall best model, namely slGMM with GMLVQ initialization and $K = 2$.

Transfer Learning: In our main analysis, we first considered the case where data from all classes is available for transfer learning. Again, we iterate over all 236 runs and treat the data in the current run as test data, both for the source as well as for the target space. As training data in the source space, we use the data from all remaining runs of the same subject. We train a slGMM with GMLVQ initialization and $K = 2$ prototypes per class starting from 5 random initializations and select the one with the lowest training error. Then, we use $N \in [4, 8, 16, 32, 64, 128]$ randomly selected target samples from the remaining runs of the same subject as training data for transfer learning and record the classification error on the test target space data from the current run. For transfer learning, we compare gradient-based transfer learning based on the GMLVQ cost function (refer to Section 7.2), EM transfer learning (refer to Section 7.2), and the adaptive support vector machine (a-SVM) of J. Yang, Yan, and Hauptmann (2007). We also ran the experiment with the asymmetric regularized cross-domain transformation (ARC-t) and heterogeneous feature augmentation (HFA) techniques, but these resulted in errors consistently above 70%, such that we do not report these results here. As additional

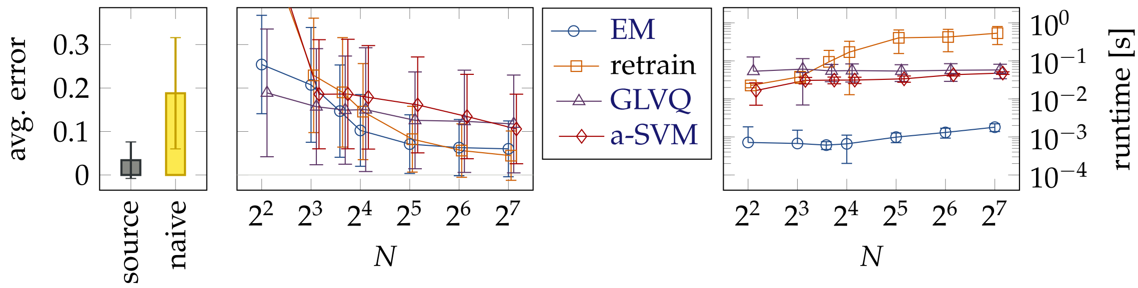


Figure 8.3: Mean classification error (left, middle) and mean runtime (right) across all runs in the second dataset. The x-axis indicates the number of available target space training data points N (in log scaling) while the y-axis displays the mean classification error (left, middle) or the runtime (right, log scale). Error bars indicate the standard deviation.

Table 8.2: Mean classification test error and standard deviation across all runs in the second dataset. The different transfer learning approaches are listed on the x axis, the number of data points N for transfer learning on the y axis. The best results in each row are highlighted via bold print.

N	naive	EM	retrain	GMLVQ	a-SVM
4	$18.8 \pm 12.8\%$	$24.3 \pm 10.5\%$	$53.1 \pm 8.3\%$	$\mathbf{18.0 \pm 13.6}\%$	$53.3 \pm 7.9\%$
8	$18.8 \pm 12.8\%$	$21.8 \pm 13.8\%$	$21.5 \pm 13.3\%$	$\mathbf{16.5 \pm 13.9}\%$	$18.6 \pm 12.7\%$
12	$18.8 \pm 12.8\%$	$\mathbf{13.1 \pm 9.5}\%$	$17.8 \pm 11.4\%$	$14.4 \pm 11.8\%$	$18.6 \pm 12.7\%$
16	$18.8 \pm 12.8\%$	$\mathbf{10.5 \pm 8.9}\%$	$14.6 \pm 10.7\%$	$14.5 \pm 13.2\%$	$17.8 \pm 12.2\%$
32	$18.8 \pm 12.8\%$	$\mathbf{7.1 \pm 7.1}\%$	$8.8 \pm 8.4\%$	$12.9 \pm 11.5\%$	$16.1 \pm 11.3\%$
64	$18.8 \pm 12.8\%$	$6.8 \pm 7.0\%$	$\mathbf{6.0 \pm 6.4}\%$	$12.5 \pm 11.3\%$	$13.7 \pm 10.1\%$
128	$18.8 \pm 12.8\%$	$6.2 \pm 6.5\%$	$\mathbf{4.4 \pm 5.5}\%$	$11.5 \pm 10.9\%$	$11.0 \pm 8.4\%$

baselines, we also compare to the classification error of the source model both on the source and on the target data (naive), and to a newly trained model. Our hypotheses are that EM transfer learning should achieve batter accuracy than a retrained model when trained with few data (H1) or with few classes (H2). Further, we expect that EM transfer learning is considerably faster compared to all alternatives, given that we can utilize a closed-form optimization (H3).

The mean classification error across all 236 runs is shown in Table 8.2 and Figure 8.3 (left and middle). We observe several significant effects using a one-sided Wilcoxon signed rank test:

1. After electrode shift, the classification performance degrades, i.e. the naive error is significantly higher than the source error ($p < 10^{-3}$).

2. If at least 12 data points are available for training, EM transfer learning outperforms a naive application of the source space model ($p < 10^{-3}$).

3. If between 12 and 32 data points are available, EM transfer learning outperforms a retrained model on the target data ($p < 10^{-3}$), lending support for H1.

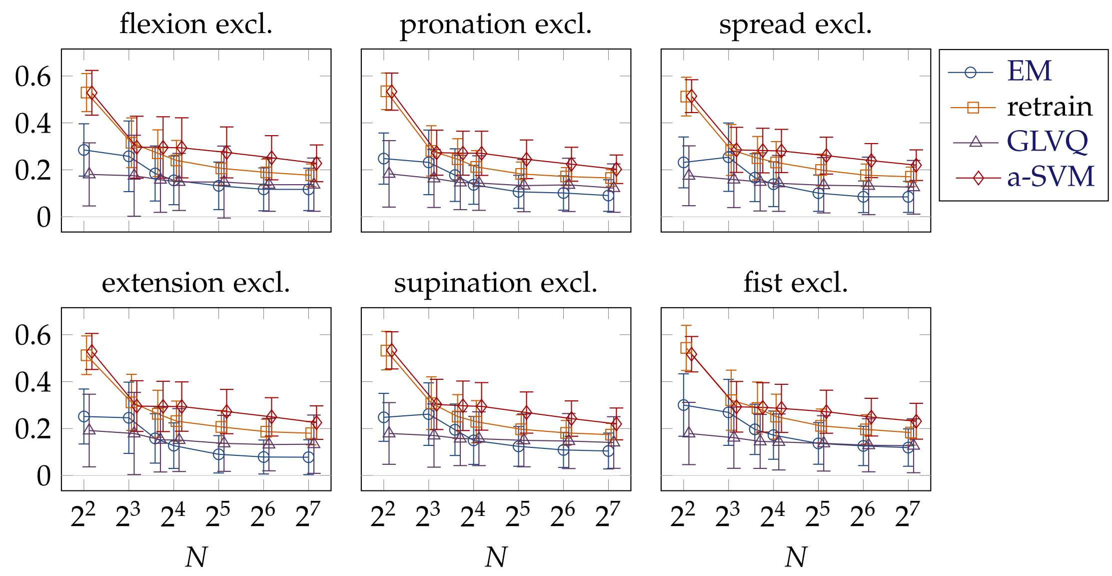


Figure 8.4: Mean classification error across all runs in the myoelectric dataset if one movement was excluded from the training data for transfer learning. The excluded class is listed in the title of each plot. The x-axis indicates the number of available target space training data points N (in log scaling) while the y-axis displays the mean classification error. Error bars indicate the standard deviation.

Table 8.3: Mean classification test error and standard deviation across all runs in the second dataset when no samples for the extension movement were available for transfer learning. The different transfer learning approaches are listed on the x axis, the number of data points N for transfer learning on the y axis. The best results in each row are highlighted via bold print.

N	naive	EM	retrain	GMLVQ	a-SVM
4	**18.8 ± 12.8%**	25.4 ± 11.5%	52.4 ± 9.2%	18.9 ± 15.0%	52.8 ± 7.6%
8	18.8 ± 12.8%	24.4 ± 14.7%	31.0 ± 11.5%	**17.8 ± 17.1%**	29.7 ± 10.8%
12	18.8 ± 12.8%	16.5 ± 12.8%	27.3 ± 10.6%	**15.0 ± 12.5%**	29.5 ± 10.8%
16	18.8 ± 12.8%	**13.6 ± 10.2%**	23.9 ± 8.9%	15.1 ± 12.3%	29.3 ± 10.6%
32	18.8 ± 12.8%	**9.3 ± 8.0%**	21.0 ± 7.0%	13.8 ± 11.9%	26.7 ± 9.1%
64	18.8 ± 12.8%	**8.2 ± 7.6%**	18.9 ± 6.0%	13.3 ± 11.8%	24.5 ± 8.1%
128	18.8 ± 12.8%	**7.7 ± 7.2%**	17.9 ± 5.0%	12.4 ± 11.6%	22.8 ± 7.5%

4. If at least 12 data points are available for training, EM transfer learning outperforms the adaptive SVM ($p < 10^{-3}$).

5. If at least 16 data points are available for training, EM transfer learning outperforms GLVQ transfer learning ($p < 10^{-3}$).

With regards to runtime, we note that our proposed algorithm is roughly 30 times faster compared to GMLVQ transfer learning and a-SVM and roughly 100 times faster compared to re-training a new model on the target space data (see Figure 8.3, right), supporting H3. We also observed a runtime advantage of around factor 100 versus HFA and of around 500 compared to ARC-t.

To investigate H2, we repeated our experiments six times, each time excluding one of the atomic hand motions from the training data for transfer learning. We also experimented with omitting more than one class in the training data but observed that no transfer method outperformed the baseline of naively applying the source model to the target space data.

The average results across participants and trials are depicted in Figure 8.4. Table 8.3 shows the results without extension motions in the training data. We observe the following significant effects using a one-sided Wilcoxon signed rank test.

1. If at least 32 data points are available for training, EM transfer learning outperforms a naive application of the source space model ($p < 10^{-3}$).
2. Irrespective of the number of available data points, EM transfer learning outperforms a retrained model on the target data ($p < 10^{-3}$).
3. If at least 12 data points are available for training, EM transfer learning outperforms the a-SVM ($p < 10^{-3}$).
4. If extension, pronation, supination, or spread are excluded and at 32 data points are available for training, EM transfer learning outperforms GLVQ transfer learning function ($p < 0.01$).

In conjunction, these results support H2. We also note again that ARC-t and HFA resulted in errors consistently above 70% on these data, such that our method significantly outperforms these references across all conditions.

8.2 CONCLUSION

In this chapter, we have introduced the application domain of bionic hand prostheses and the challenge of electrode shifts. In particular, any machine learning model that serves as a user interface to map muscle signals to desired actions of the prosthesis can be disturbed by shifts of EMG electrodes on the skin. To counteract such shifts, we have proposed to record a small calibration set of disturbed data and to learn a linear function that cleans up the disturbed data such that the model can correctly classify the data again. To learn this linear function, we have applied the transfer learning algorithms from Chapter 7.

In our experimental evaluation on two EMG datasets we found that transfer learning can indeed improve classification error after disturbance, and that EM transfer learning can improve classification error beyond retraining a new model and other transfer learning techniques if few data or few classes are available. We also showed that EM transfer learning is orders of magnitude faster compared to learning a new model or other transfer learning techniques. These results give reason to hope that transfer learning can give patients a quick, efficient, and robust tool to re-calibrate their prosthesis after everyday disturbances and thus improve their quality of life.

9 CONCLUSIONS AND OUTLOOK

In this dissertation, I have addressed the challenge of metric learning for structured data and enhanced the utility of a learned metric. In Chapter 3, I have developed a gradient-based metric learning scheme for all sequence edit distances that can be expressed in terms of a signature, a differentiable algebra, and an edit tree grammar. Experimentally, I have shown that this scheme can improve classification of biological sequences and computer programs. Further, I have extended this scheme to trees in Chapter 4, decreased the runtime complexity, thus making metric learning applicable to much larger data sets, and parametrized the edit distance in terms of symbol embeddings, which guarantees metric properties, is more interpretable, and simplifies the application to large alphabets. I also demonstrated experimentally that my proposed metric learning scheme outperforms a state-of-the-art method in terms of metric learning for structured data.

Once we have learned a metric, we typically wish to apply it for downstream tasks. Existing methods already cover mappings to vectorial outputs, such as dimensionality reduction, classification, clustering, and regression. However, mapping to a distance representation *as output* has not yet been subject to extensive research. In Chapter 5, I established such an approach based on Gaussian process regression to perform time series prediction on structured data. Experimentally, I have shown that my proposed scheme outperforms baselines such as one-nearest neighbor regression and kernel regression. I applied this novel technique in Chapter 6 to support students in learning computer programming. Whenever a student gets stuck before completing a programming task, my proposed scheme can predict what a capable student would do in the student's situation and I can infer an edit that guides a student closer to a correct solution along a path that a capable student would take. In experiments on real-world student data, I showed that my proposed model could accurately predict what capable students would do and that the pedagogical quality of the resulting hints was on par with state-of-the-art baselines.

Another challenge in applying a learned metric is that the distribution or representation of target data may differ from the source data on which the metric was learned. In Chapter 7, I have developed a novel framework to address this challenge by learning a transfer mapping from the target space to the source space, such that the learned source space metric is applicable again. I have provided two implementations of this framework, one for transfer learning on generalized matrix learning vector quantization classifiers, and one for transfer learning on labeled Gaussian Mixture Model. Further, I applied transfer learning in Chapter 8 to counteract disturbances in bionic prostheses control. To date, such disturbances prevent patients from using bionic prostheses to their full potential because the prostheses fail to execute the desired motions in everyday life. Using transfer learning, I could clean up electrode shifts in the data and thus enhance the accuracy of a bionic prosthesis user interface. I also showed that transfer learning needs much less data and computation time compared to several baselines.

Limitations: The work presented in this dissertation still offers opportunity for further improvement. First, as mentioned in Chapter 3, the gradient computation via ADP for edit distances learning is too slow to be applicable for large-scale tasks and our proposed improved version of the method from Chapter 4, em**b**edding **e**dit **d**istance **l**earning (BEDL), has not yet been combined with the ADP framework, which is a gap in this work.

Second, BEDL does not yet reliably improve classification accuracy on all tasks, which indicates that there are still generalization issues to be addressed.

Third, the time series prediction method via Gaussian process regression (GPR) suggested in Chapter 5 still relies on an eigenvalue correction, which distorts the space and complicates the application to novel data. Further, our proposed method requires storing all training samples to perform predictions, which may become prohibitive for very large structured datasets. In such large-scale scenarios, a parametric model with an explicit vectorial embedding, such as a recursive neural network, may be more promising. Fourth, while we could improve predictive performance over several baselines, these results did not translate to significantly better hint quality for intelligent tutoring systems in Chapter 6, indicating that the translation from kernel to primal space still could be improved, either by secondary criteria like syntactic correctness or unit test performance, or by using multiple edits instead of a single edit.

Fifth, our transfer learning method proposed in Chapter 7 is currently limited to linear functions, which may be insufficient for more complicated disturbances. Conversely, a full linear transformation may entail too many free parameters for very simple disturbances like electrode shifts in Chapter 8. In this scenario, we could inject more prior knowledge to simplify the problem further and thus achieve better results with even less data, especially less classes to record.

Outlook: Beyond improvements of the methods presented in this paper, this thesis opens up multiple exciting avenues for further research.

First, I have shown that grammars and automata can serve as efficient and general interfaces to compute continuous gradients over discrete structures. In Chapter 3, I have used this connection to compute gradients over general string edit distances. Beyond edit distances, this connection could be useful for any domain that can be modeled in terms of formal grammars, such as computer programs (Aho et al. 2006), biological structures (Searls 2012), or chemical molecules (Weininger 1988). Kusner, Paige, and Hernández-Lobato (2017) have done first promising steps in this direction by modeling chemical molecules via a grammar and then learning continuous vectorial representations for the words produced by said grammar.

Second, this work has explored the connection between representation learning and metric learning. In vectorial metric learning, this connection is obvious since metric learning corresponds to a linear mapping of the input data into an alternative space, i.e. an alternative representation (Bunte et al. 2012). However, this connection has not yet been well explored for structured data. Previous work has shown that any pseudo-Euclidean distance and any kernel, including those for structured data, correspond to an implicit vectorial representation (Pekalska and Duin 2005, also refer to Section 2.1). In this work, I have developed metric learning for edit distances on structured data by learning an explicit vectorial representation of symbols (refer to Chapter 4), which can be seen as a supervised version of word embedding learning (Mikolov et al. 2013; Pennington, Socher, and Manning 2014). I have also shown that we can translate affine combinations in the pseudo-Euclidean space of edit distances back to actual structured data (refer to Chapter 6). Future work could extend this link between metric learning on and vectorial representations of structured data with the aim to make such representations easier to learn, easier to interpret, and easier to invert.

Third, we have seen that we can interpret edit distances as shortest paths in a graph of possible structured data. This re-interpretation also enabled us to generate hints for students in intelligent tutoring systems (refer to Chapter 6). Future work could explore this application in more detail, for example in the form of classroom studies regarding how much students actually profit from edit hints, and by incorporating additional constraints for possible edits, such as syntactic of semantic correctness. Beyond this application, edit distances provide an avenue towards interpreting learned models in machine learning more generally. For example, we could ask which edits we would need to apply to a structured datum such that it is classified differently, maximizes a certain property, or moves along a desired trajectory in the space of possible structured data.

Finally, I posed the general problem of supervised transfer learning with explicit transfer functions, and achieved a particularly data- and time-efficient expectation maximization transfer learning algorithm in order to make a learned model from one domain applicable in another domain. This makes bionic hand prostheses easy to re-calibrate after everyday disturbances. Future work in this regard could go further and incorporate more domain-specific knowledge regarding the form of the transfer function and evaluate the utility of transfer learning in clinical studies. Supervised transfer learning could also be applicable far beyond prosthetic research. By exploring nonlinear transfer functions, alternative parametrizations, and transfer functions for structured data, supervised transfer learning could become a useful tool in transferring machine learning models from the lab to actual, real-world applications using only minimal data and computational effort.

Overall, this thesis provides ample opportunity for further research incorporating knowledge from classical grammar theory, representation learning, and application domains to push the boundaries of machine learning on structured data.

PUBLICATIONS IN THE CONTEXT OF THIS THESIS

Mokbel, Bassam, Benjamin Paaßen, et al. (2015). "Metric learning for sequences in relational LVQ". English. In: *Neurocomputing* 169, pp. 306–322. DOI: 10.1016/j.neucom.2014.11.082.

Paaßen, Benjamin, Bassam Mokbel, and Barbara Hammer (2015a). "A Toolbox for Adaptive Sequence Dissimilarity Measures for Intelligent Tutoring Systems". In: *Proceedings of the 8th International Conference on Educational Data Mining (EDM 2015)*. (Madrid, Spain). Ed. by Olga Christina Santos et al. International Educational Datamining Society, pp. 632–632. URL: http://www.educationaldatamining.org/EDM2015/uploads/papers/paper_257.pdf.

— (2015b). "Adaptive structure metrics for automated feedback provision in Java programming". English. In: *Proceedins of the 23rd European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN 2015)*. (Bruges, Belgium). Ed. by Michel Verleysen. **Best student paper award**. i6doc.com, pp. 307–312. URL: http://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2015-43.pdf.

Göpfert, Christina, Benjamin Paaßen, and Barbara Hammer (2016). "Convergence of Multi-pass Large Margin Nearest Neighbor Metric Learning". In: *Proceedings of the 25th International Conference on Artificial Neural Networks (ICANN 2016)*. (Barcelona, Spain). Ed. by Alessandro E.P. Villa, Paolo Masulli, and Antonio Javier Pons Rivero. Vol. 9886. Lecture Notes in Computer Science. Springer, pp. 510–517. DOI: 10.1007/978-3-319-44778-0_60.

Paaßen, Benjamin, Christina Göpfert, and Barbara Hammer (2016). "Gaussian process prediction for time series of structured data". In: *Proceedings of the 24th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN 2016)*. (Bruges, Belgium). Ed. by Michel Verleysen. i6doc.com, pp. 41–46. URL: http://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2016-109.pdf.

Paaßen, Benjamin, Joris Jensen, and Barbara Hammer (2016). "Execution Traces as a Powerful Data Representation for Intelligent Tutoring Systems for Programming". English. In: *Proceedings of the 9th International Conference on Educational Data Mining (EDM 2016)*. (Raleigh, North Carolina, USA). Ed. by Tiffany Barnes, Min Chi, and Mingyu Feng. **Exemplary Paper**. International Educational Datamining Society, pp. 183–190. URL: http://www.educationaldatamining.org/EDM2016/proceedings/paper_17.pdf.

Paaßen, Benjamin, Bassam Mokbel, and Barbara Hammer (2016). "Adaptive structure metrics for automated feedback provision in intelligent tutoring systems". In: *Neurocomputing* 192, pp. 3–13. DOI: 10.1016/j.neucom.2015.12.108.

Paaßen, Benjamin, Alexander Schulz, and Barbara Hammer (2016). "Linear Supervised Transfer Learning for Generalized Matrix LVQ". In: *Proceedings of the Workshop New Challenges in Neural Computation (NC^2 2016)*. (Hannover, Germany). Ed. by Barbara Hammer, Thomas Martinetz, and Thomas Villmann. **Best presentation award**, pp. 11–18. URL: https://www.techfak.uni-bielefeld.de/~fschleif/mlr/mlr_04_2016.pdf#page=14.

Prahm, Cosima et al. (2016). "Transfer Learning for Rapid Re-calibration of a Myoelectric Prosthesis after Electrode Shift". In: *Proceedings of the 3rd International Conference on NeuroRehabilitation (ICNR 2016)*. (Segovia, Spain). Ed. by Jaime Ibáñez et al. Vol. 15. Converging Clinical and Engineering Research on Neurorehabilitation II. Biosystems

& Biorobotics. **Runner-Up for Best Student Paper Award**. Springer, pp. 153–157. DOI: 10.1007/978-3-319-46669-9_28.

Paaßen, Benjamin et al. (2017). “An EM transfer learning algorithm with applications in bionic hand prostheses”. In: *Proceedings of the 25th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN 2017)*. (Bruges, Belgium). Ed. by Michel Verleysen. i6doc.com, pp. 129–134. URL: http://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2017-57.pdf.

Paaßen, Benjamin (2018). *Revisiting the tree edit distance and its backtracing: A tutorial*. arXiv: 1805.06869 [cs.DS].

Paaßen, Benjamin, Claudio Gallicchio, et al. (2018). “Tree Edit Distance Learning via Adaptive Symbol Embeddings”. In: *Proceedings of the 35th International Conference on Machine Learning (ICML 2018)*. (Stockholm, Sweden). Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research, pp. 3973–3982. URL: http://proceedings.mlr.press/v80/paassen18a.html.

Paaßen, Benjamin, Christina Göpfert, and Barbara Hammer (2018). “Time Series Prediction for Graphs in Kernel and Dissimilarity Spaces”. In: *Neural Processing Letters* 48.2, pp. 669–689. DOI: 10.1007/s11063-017-9684-5.

Paaßen, Benjamin, Barbara Hammer, et al. (2018). “The Continuous Hint Factory - Providing Hints in Vast and Sparsely Populated Edit Distance Spaces”. In: *Journal of Educational Datamining* 10.1, pp. 1–35. URL: https://jedm.educationaldatamining.org/index.php/JEDM/article/view/158.

Paaßen, Benjamin et al. (2018). “Expectation maximization transfer learning and its application for bionic hand prostheses”. In: *Neurocomputing* 298, pp. 122–133. DOI: 10.1016/j.neucom.2017.11.072.

REFERENCES

Adamatzky, Andrew (2002). *Collision-Based Computing*. Berlin/Heidelberg, Germany: Springer. ISBN: 978-1-4471-0129-1.

Ahmad, A.S. et al. (2014). “A review on applications of ANN and SVM for building electrical energy consumption forecasting”. In: *Renewable and Sustainable Energy Reviews* 33, pp. 102–109. DOI: 10.1016/j.rser.2014.01.069.

Ahmad, Farooq and Grzegorz Kondrak (2005). “Learning a Spelling Error Model from Search Query Logs”. In: *Proceedings of the Conference on Human Language Technology (HLT 2005)*. Ed. by Raymond Mooney, pp. 955–962. DOI: 10.3115/1220575.1220695.

Aho, Alfred et al. (2006). *Compilers: Principles, Techniques, and Tools*. 2nd ed. Boston, MA, US: Addison Wesley. ISBN: 978-0321486813.

Aiolli, Fabio and Michele Donini (2015). “EasyMKL: a scalable multiple kernel learning algorithm”. In: *Neurocomputing* 169, pp. 215–224. DOI: 10.1016/j.neucom.2014.11.078.

Aiolli, Fabio, Giovanni Da San Martino, and Alessandro Sperduti (2015). “An Efficient Topological Distance-Based Tree Kernel”. In: *IEEE Transactions on Neural Networks and Learning Systems* 26.5, pp. 1115–1120. DOI: 10.1109/TNNLS.2014.2329331.

Akutsu, Tatsuya (2010). “Tree Edit Distance Problems: Algorithms and Applications to Bioinformatics”. In: *IEICE Transactions on Information and Systems* E93-D.2, pp. 208–218. DOI: 10.1587/transinf.E93.D.208.

Aleven, Vincent, Bruce M. McLaren, et al. (2006). “The Cognitive Tutor Authoring Tools (CTAT): Preliminary Evaluation of Efficiency Gains”. In: *Proceedings of the 8th International Conference on Intelligent Tutoring Systems (ITS 2006)*. Ed. by Mitsuru Ikeda, Kevin D. Ashley, and Tak-Wai Chan. Springer, pp. 61–70. DOI: 10.1007/11774303_7.

Aleven, Vincent, Ido Roll, et al. (2016). “Help Helps, But Only So Much: Research on Help Seeking with Intelligent Tutoring Systems”. In: *International Journal of Artificial Intelligence in Education* 26.1, pp. 205–223. DOI: 10.1007/s40593-015-0089-1.

Augsten, Nikolaus, Michael Böhlen, and Johann Gamper (2008). “The pq-gram Distance Between Ordered Labeled Trees”. In: *ACM Transactions on Database Systems* 35.1, 4:1–4:36. DOI: 10.1145/1670243.1670247.

Bacciu, Davide, Federico Errica, and Alessio Micheli (2018). “Contextual Graph Markov Model: A Deep and Generative Approach to Graph Processing”. In: *Proceedings of the 35th International Conference on Machine Learning (ICML 2018)*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research, pp. 294–303. URL: http://proceedings.mlr.press/v80/bacciu18a.html.

Bacciu, Davide, Claudio Gallicchio, and Alessio Micheli (2016). “A reservoir activation kernel for trees”. In: *Proceedings of the 24th European Symposium on Artificial Neural Networks (ESANN 2016)*. Ed. by Michel Verleysen. URL: http://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2016-172.pdf.

Bakır, Gökhan H., Jason Weston, and Bernhard Schölkopf (2003). “Learning to Find Pre-images”. In: *Proceedings of the 16th International Conference on Neural Information Processing Systems (NIPS 2003)*. Ed. by S. Thrun, L. K. Saul, and B. Schölkopf, pp. 449–456. URL: https://papers.nips.cc/paper/2417-learning-to-find-pre-images.

Bakır, Gökhan H., Alexander Zien, and Koji Tsuda (2004). “Learning to Find Graph Pre-images”. In: *Proceedings of the fourth German Conference on Pattern Recognition (DAGM 2004)*, pp. 253–261. DOI: 10.1007/978-3-540-28649-3_31.

Balcan, Maria-Florina, Avrim Blum, and Nathan Srebro (2008). "A theory of learning with similarity functions". In: *Machine Learning* 72.1, pp. 89–112. DOI: 10.1007/s10994-008-5059-5.

Barabási, Albert-László and Réka Albert (1999). "Emergence of Scaling in Random Networks". In: *Science* 286.5439, pp. 509–512. DOI: 10.1126/science.286.5439.509.

Barber, David (2012). *Bayesian Reasoning and Machine Learning*. Cambridge, UK: Cambridge University Press. ISBN: 978-0-521-51814-7. URL: http://www0.cs.ucl.ac.uk/staff/d.barber/brml/.

Barnes, Tiffany, Behrooz Mostafavi, and Michael Eagle (2016). "Data-driven domain models for problem solving". In: *Domain Modeling*. Ed. by Robert A. Sottilare et al. Vol. 4. Design Recommendations for Intelligent Tutoring Systems. US Army Research Laboratory, pp. 137–145. ISBN: 978-0-9893923-9-6. URL: https://gifttutoring.org/documents/105.

Barnes, Tiffany and John Stamper (2008). "Toward Automatic Hint Generation for Logic Proof Tutoring Using Historical Student Data". In: *Proceedings of the 9th International Conference on Intelligent Tutoring Systems (ITS 2008)*. Ed. by Beverley P. Woolf et al., pp. 373–382. DOI: 10.1007/978-3-540-69132-7_41.

Barnett, Susan and Stephen Ceci (2002). "When and where do we apply what we learn?: A taxonomy for far transfer". In: *Psychological bulletin* 128.4, p. 612. DOI: 10.1037/0033-2909.128.4.612.

Barrett, Christopher L., Henning S. Mortveit, and Christian M. Reidys (2000). "Elements of a theory of simulation II: sequential dynamical systems". In: *Applied Mathematics and Computation* 107.2-3, pp. 121–136. DOI: 10.1016/S0096-3003(98)10114-5.

— (2003). "ETS IV: Sequential dynamical systems: fixed points, invertibility and equivalence". In: *Applied Mathematics and Computation* 134.1, pp. 153–171. DOI: 10.1016/S0096-3003(01)00277-6.

Barrett, Christopher L. and Christian M. Reidys (1999). "Elements of a Theory of Computer Simulation I: Sequential CA over Random Graphs". In: *Applied Mathematics and Computation* 98.2-3, pp. 241–259. DOI: 10.1016/S0096-3003(97)10166-7.

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

— (2014). *A Survey on Metric Learning for Feature Vectors and Structured Data*. arXiv: 1306.6709 [cs.LG].

Ben-David, Shai et al. (2006). "Analysis of representations for domain adaptation". In: *Proceedings of the 19th Advances in Neural Information Processing Systems Conference (NIPS 2006)*. Ed. by Bernhard Schölkopf, John C. Platt, and T. Hoffman, pp. 137–144. URL: https://papers.nips.cc/paper/2983-analysis-of-representations-for-domain-adaptation.

Bengio, Yoshua, Aaron Courville, and Pascal Vincent (2013). "Representation Learning: A Review and New Perspectives". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35.8, pp. 1798–1828. DOI: 10.1109/TPAMI.2013.50.

Bergstra, James and Yoshua Bengio (2012). "Random Search for Hyper-Parameter Optimization". In: *Journal of Machine Learning Research* 13, pp. 281–305. URL: http://www.jmlr.org/papers/v13/bergstra12a.html.

Biddiss, Elaine A. and Tom T. Chau (2007). "Upper limb prosthesis use and abandonment: A survey of the last 25 years". In: *Prosthetics and Orthotics International* 31.3, pp. 236–257. DOI: 10.1080/03093640600994581.

Biehl, Michael et al. (2015). "Stationarity of Matrix Relevance LVQ". In: *Proceedings of the 2015 International Joint Conference on Neural Networks (IJCNN 2015)*. Ed. by Yoonsuck Choe De-Shuang Huang, pp. 1–8. DOI: 10.1109/IJCNN.2015.7280441.

Bille, Philip (2005). "A survey on tree edit distance and related problems". In: *Theoretical Computer Science* 337.1, pp. 217–239. DOI: 10.1016/j.tcs.2004.12.030.

Bishop, Christopher M. (2006). *Pattern Recognition and Machine Learning*. Berlin/Heidelberg, Germany: Springer. ISBN: 0387310738.

Blitzer, John, Ryan McDonald, and Fernando Pereira (2006). "Domain Adaptation with Structural Correspondence Learning". In: *Proceedings of the 2006 Conference on Empirical Methods in Natural Language Processing (EMNLP 2006)*. Ed. by Dan Jurafsky and Eric Gaussier, pp. 120–128. URL: https://aclanthology.info/pdf/W/W06/W06-1615.pdf.

Blöbaum, Patrick, Alexander Schulz, and Barbara Hammer (2015). "Unsupervised Dimensionality Reduction for Transfer Learning". In: *Proceedings of the 23rd European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN 2015)*. Ed. by Michel Verleysen, pp. 507–512. URL: http://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2015-134.pdf.

Borgwardt, Karsten and Hans-Peter Kriegel (2005). "Shortest-path kernels on graphs". In: *Proceedings of the 5th IEEE International Conference on Data Mining (ICDM 2005)*. Ed. by Jiawei Han et al. DOI: 10.1109/ICDM.2005.132.

Boyer, Laurent, Amaury Habrard, and Marc Sebban (2007). "Learning Metrics Between Tree Structured Data: Application to Image Recognition". In: *Proceedings of the 18th European Conference on Machine Learning (ECML 2007)*. Ed. by Joost N. Kok et al., pp. 54–66. DOI: 10.1007/978-3-540-74958-5_9.

Bunte, Kerstin et al. (2012). "Limited Rank Matrix Learning, discriminative dimension reduction and visualization". In: *Neural Networks* 26, pp. 159–173. DOI: 10.1016/j.neunet.2011.10.001.

Casteigts, Arnaud et al. (2012). "Time-varying graphs and dynamic networks". In: *International Journal of Parallel, Emergent and Distributed Systems* 27.5, pp. 387–408. DOI: 10.1080/17445760.2012.668546.

Chang, Chih-Chung and Chih-Jen Lin (2011). "LIBSVM: A Library for Support Vector Machines". In: *ACM Transactions on Intelligent Systems and Technology* 2.3. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm, 27:1–27:27. DOI: 10.1145/1961189.1961199.

Cho, Kyunghyun et al. (2014). "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP 2014)*. Ed. by Alessandro Moschitti, Bo Pang, and Walter Daelemans, pp. 1724–1734. URL: https://www.aclweb.org/anthology/D14-1179.

Choudhury, Rohan Roy, Hezheng Yin, and Armando Fox (2016). "Scale-Driven Automatic Hint Generation for Coding Style". In: *Proceedings of the 13th International Conference on Intelligent Tutoring Systems (ITS 2016)*. Ed. by Alessandro Micarelli, John Stamper, and Kitty Panourgia, pp. 122–132. DOI: 10.1007/978-3-319-39583-8_12.

Chung, Junyoung et al. (2015). "A Recurrent Latent Variable Model for Sequential Data". In: *Proceedings of the 28th Conference on Advances in Neural Information Processing Systems (NIPS 2015)*. Ed. by C. Cortes et al., pp. 2980–2988. URL: http://papers.nips.cc/paper/5653-a-recurrent-latent-variable-model-for-sequential-data.

Clauset, Aaron (2013). "Generative Models for Complex Network Structure". In: *Proceedings of the 8th International School and Conference on Network Science (NetSci 2013)*. URL: http://www2.imm.dtu.dk/~tuhe/cnmml/pdf/clauset.pdf.

Cortes, Corinna et al. (2008). "Sample Selection Bias Correction Theory". In: *Proceedings of the 19th International Conference on Algorithmic Learning Theory (ALT 2008)*. Ed. by Yoav Freund et al., pp. 38–53. DOI: 10.1007/978-3-540-87987-9_8.

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

Da San Martino, Giovanni and Alessandro Sperduti (2010). "Mining Structured Data". In: *Computational Intelligence Magazine* 5.1, pp. 42–49. DOI: 10.1109/MCI.2009.935308.

Damerau, Fred (1964). "A Technique for Computer Detection and Correction of Spelling Errors". In: *Communications of the ACM* 7.3, pp. 171–176. DOI: 10.1145/363958.363994.

Davis, Jason et al. (2007). "Information-theoretic Metric Learning". In: *Proceedings of the 24th International Conference on Machine Learning (ICML 2007)*. Ed. by Claude Sammut and Zoubin Ghahramani, pp. 209–216. DOI: 10.1145/1273496.1273523.

— (2010). "Metric learning to Rank". In: *Proceedings of the 27th International Conference on Machine Learning (ICML 2010)*. Ed. by Stefan Wrobel, Johannes Fürnkranz, and Thorsten Joachims, pp. 775–782. URL: https://bmcfee.github.io/papers/mlr.pdf.

De Vries, Harm, Roland Memisevic, and Aaron Courville (2016). "Deep learning vector quantization". In: *Proceedings of the 2th European Symposium on Artificial Neural Networks (ESANN 2016)*. Ed. by Michel Verleysen. URL: https://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2016-112.pdf.

Deisenroth, Marc Peter and Jun Wei Ng (2015). "Distributed Gaussian Processes". In: *Proceedings of the 32nd International Conference on Machine Learning (ICML 2015)*. Ed. by Francis Bach and David Blei, pp. 1481–1490. URL: http://proceedings.mlr.press/v37/deisenroth15.html.

Demaine, Erik D. et al. (2009). "An Optimal Decomposition Algorithm for Tree Edit Distance". In: *ACM Transactions on Algorithms* 6.1, 2:1–2:19. DOI: 10.1145/1644015.1644017.

Dempster, A. P., N. M. Laird, and D. B. Rubin (1977). "Maximum likelihood from incomplete data via the EM algorithm". In: *Journal of the Royal Statistical Society. Series B* 39.1, pp. 1–38. URL: https://www.jstor.org/stable/2984875.

Ditzler, Gregory et al. (2015). "Learning in Nonstationary Environments: A Survey". In: *IEEE Computational Intelligence Magazine* 10.4, pp. 12–25. DOI: 10.1109/MCI.2015.2471196.

Duan, Lixin, Dong Xu, and Ivor Tsang (2012). "Learning with Augmented Features for Heterogeneous Domain Adaptation". In: *Proceedings of the 29th International Conference on Machine Learning (ICML 2012)*. (Edinburgh, UK). Ed. by Andrew McCallum, John Langford, and Joelle Pineau. URL: https://arxiv.org/abs/1206.4660.

Eagle, Michael and Tiffany Barnes (2013). "Evaluation of automatically generated hint feedback". In: *Proceedings of the 6th International Conference on Educational Data Mining (EDM 2013)*. Ed. by S. K. D'Mello, R. A. Calvo, and A. Olney, pp. 372–374. URL: http://www.educationaldatamining.org/EDM2013/papers/rn_paper_87.pdf.

Eagle, Michael, Matthew Johnson, and Tiffany Barnes (2012). "Interaction Networks: Generating High Level Hints Based on Network Community Clustering". In: *Proceedings of the 5th International Conference on Educational Data Mining (EDM 2012)*. Ed. by K. Yacef et al., pp. 164–167. URL: https://eric.ed.gov/?id=ED537223.

Emms, Martin (2012). "On Stochastic Tree Distances and Their Training via Expectation-Maximisation". In: *Proceedings of the 1st International Conference on Pattern Recognition Applications and Methods (ICPRAM 2012)*. Ed. by Pedro Carmona, Salvador Sánchez, and Ana Fred, pp. 144–153.

Fackler, Paul L. (2005). *Notes on matrix calculus*. Tech. rep. North Carolina State University. URL: http://www4.ncsu.edu/~pfackler/MatCalc.pdf.

Farina, Dario et al. (2014). "The Extraction of Neural Information from the Surface EMG for the Control of Upper-Limb Prostheses: Emerging Avenues and Challenges". In: *IEEE Transactions on Neural Systems and Rehabilitation Engineering* 22.4, pp. 797–809. DOI: 10.1109/TNSRE.2014.2305111.

Feragen, Aasa et al. (2013). "Scalable kernels for graphs with continuous attributes". In: *Proceedings of the 26th conference on Advances in Neural Information Processing Systems (NIPS 2013)*. Ed. by C. J. C. Burges et al., pp. 216–224. URL: http://papers.nips.cc/paper/5155-scalable-kernels-for.

Filippone, Maurizio et al. (2008). "A survey of kernel and spectral methods for clustering". In: *Pattern Recognition* 41.1, pp. 176–190. DOI: 10.1016/j.patcog.2007.05.018.

Fleming, Malcolm L. and W. Howard Levie (1993). *Instructional Message Design: Principles from the Behavioral and Cognitive Sciences*. Englewood Cliffs, NJ, USA: Educational Technology Publications. ISBN: 978-0877782537.

Floyd, Robert W. (1962). "Algorithm 97: Shortest Path". In: *Communications of the ACM* 5.6, pp. 345–345. DOI: 10.1145/367766.368168.

Freeman, Paul, Ian Watson, and Paul Denny (2016). "Inferring Student Coding Goals Using Abstract Syntax Trees". In: *Proceedings of the 24th International Conference on Case-Based Reasoning Research and Development (ICCBR 2016)*. Ed. by Ashok Goel, M Belén Díaz-Agudo, and Thomas Roth-Berghofer, pp. 139–153. DOI: 10.1007/978-3-319-47096-2_10.

Gallicchio, Claudio and Alessio Micheli (2010). "Graph Echo State Networks". In: *Proceedings of the 23rd International Joint Conference on Neural Networks (IJCNN 2010)*. Ed. by Pillar Sobrevilla et al., pp. 1–8. DOI: 10.1109/IJCNN.2010.5596796.

— (2013). "Tree Echo State Networks". In: *Neurocomputing* 101, pp. 319–337. DOI: 10.1016/j.neucom.2012.08.017.

Gao, Xinbo et al. (2010). "A survey of graph edit distance". In: *Pattern Analysis and Applications* 13.1, pp. 113–129. DOI: 10.1007/s10044-008-0141-y.

Garcia Duran, Alberto and Mathias Niepert (2017). "Learning Graph Representations with Embedding Propagation". In: *Proceedings of the 30th Conference on Advances in Neural Information Processing Systems (NIPS 2017)*. Ed. by I. Guyon et al., pp. 5119–5130. URL: http://papers.nips.cc/paper/7097-learning-graph-representations-with-embedding-propagation.

Garcia, Dan, Brian Harvey, and Tiffany Barnes (2015). "The Beauty and Joy of Computing". In: *ACM Inroads* 6.4, pp. 71–79. DOI: 10.1145/2835184.

Gardner, Martin (1970). "Mathematical Games – The fantastic combinations of John Conway's new solitaire game 'life'". In: *Scientific American* 223, pp. 120–123.

Gentner, Dedre and Arthur Markman (1997). "Structure mapping in analogy and similarity". In: *American psychologist* 52.1, p. 45. DOI: 10.1037/0003-066X.52.1.45.

Gepperth, Alexander and Barbara Hammer (2016). "Incremental learning algorithms and applications". In: *Proceedings of the 24th European Symposium on Artificial Neural Networks (ESANN 2016)*. Ed. by Michel Verleysen. URL: http://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2016-19.pdf.

Giegerich, Robert, Carsten Meyer, and Peter Steffen (2004). "A discipline of dynamic programming over sequence data". In: *Science of Computer Programming* 51.3, pp. 215–263. DOI: 10.1016/j.scico.2003.12.005.

Girard, Agathe et al. (2003). "Gaussian process priors with uncertain inputs-application to multiple-step ahead time series forecasting". In: *Proceedings of the 15th conference on Advances in neural information processing systems (NIPS 2002)*. Ed. by S. Becker, S.

Thrun, and K. Obermayer, pp. 545–552. URL: http://papers.nips.cc/paper/2313-gaussian-process-priors-with-uncertain-inputs-application-to-multiple-step-ahead-time-series-forecasting.

Gisbrecht, Andrej, Bassam Mokbel, and Barbara Hammer (2010). “Relational generative topographic mapping”. In: *Proceedings of the 18th European Symposium on Artificial Neural Networks (ESANN 2010)*. Ed. by Michel Verleysen, pp. 277–282. URL: http://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2010-94.pdf.

Gisbrecht, Andrej and Frank-Michael Schleif (2015). “Metric and non-metric proximity transformations at linear costs”. In: *Neurocomputing* 167, pp. 643–657. DOI: 10.1016/j.neucom.2015.04.017.

Gisbrecht, Andrej, Alexander Schulz, and Barbara Hammer (2015). “Parametric nonlinear dimensionality reduction using kernel t-SNE”. In: *Neurocomputing* 147, pp. 71–82. DOI: 10.1016/j.neucom.2013.11.045.

Glorot, Xavier, Antoine Bordes, and Yoshua Bengio (2011). “Domain Adaptation for Large-Scale Sentiment Classification: A Deep Learning Approach”. In: *Proceedings of the 28th International Conference on Machine Learning (ICML 2011)*. Ed. by Lise Getoor and Tobias Scheffer, pp. 513–520. URL: http://www.icml-2011.org/papers/342_icmlpaper.pdf.

Goldenberg, Anna et al. (2010). “A Survey of Statistical Network Models”. In: *Foundations and Trends in Machine Learning* 2.2, pp. 129–233. DOI: 10.1561/2200000005.

Gönen, Mehmet and Ethem Alpaydın (2011). “Multiple kernel learning algorithms”. In: *Journal of Machine Learning Research* 12, pp. 2211–2268. URL: http://www.jmlr.org/papers/v12/gonen11a.html.

Gordon, A. D. (1987). “A Review of Hierarchical Classification”. In: *Journal of the Royal Statistical Society. Series A (General)* 150.2, pp. 119–137. DOI: 10.2307/2981629.

Gotoh, Osamu (1982). “An improved algorithm for matching biological sequences”. In: *Journal of Molecular Biology* 162.3, pp. 705–708. DOI: 10.1016/0022-2836(82)90398-9.

Gross, Sebastian, Bassam Mokbel, et al. (2014). “Example-based Feedback Provision Using Structured Solution Spaces”. In: *International Journal of Learning Technology* 9.3, pp. 248–280. DOI: 10.1504/IJLT.2014.065752.

Gross, Sebastian and Niels Pinkwart (2015). “How Do Learners Behave in Help-Seeking When Given a Choice?” In: *Proceedings of the 17th International Conference on Artificial Intelligence in Education (AIED 2015)*. Ed. by Cristina Conati et al., pp. 600–603. DOI: 10.1007/978-3-319-19773-9_71.

Hahne, Janne, Felix Biebmann, et al. (2014). “Linear and nonlinear regression techniques for simultaneous and proportional myoelectric control”. In: *IEEE Transactions on Neural Systems and Rehabilitation Engineering* 22.2, pp. 269–279. DOI: 10.1109/TNSRE.2014.2305520.

Hahne, Janne, Sven Dähne, et al. (2015). “Concurrent Adaptation of Human and Machine Improves Simultaneous and Proportional Myoelectric Control”. In: *IEEE Transactions on Neural Systems and Rehabilitation Engineering* 23.4, pp. 618–627. DOI: 10.1109/TNSRE.2015.2401134.

Hahne, Janne, Dario Farina, et al. (2016). “A Novel Percutaneous Electrode Implant for Improving Robustness in Advanced Myoelectric Control”. In: *Frontiers in Neuroscience* 10.114. DOI: 10.3389/fnins.2016.00114.

Hahne, Janne, Bernhard Graimann, and Klaus-Robert Müller (2012). “Spatial Filtering for Robust Myoelectric Control”. In: *IEEE Transactions on Biomedical Engineering* 59.5, pp. 1436–1443. DOI: 10.1109/TBME.2012.2188799.

Hamilton, William L, Rex Ying, and Jure Leskovec (2017). “Representation learning on graphs: Methods and applications”. In: *Bulletin of the Technical Committee on Data*

Engineering 40.3, pp. 52–74. URL: http://sites.computer.org/debull/A17sept/p52.pdf.
Hammer, Barbara and Alexander Hasenfuss (2007). "Relational Neural Gas". In: *KI 2007: Advances in Artificial Intelligence*. Ed. by Joachim Hertzberg, Michael Beetz, and Roman Englert, pp. 190–204. DOI: 10.1007/978-3-540-74565-5_16.
— (2010). "Topographic Mapping of Large Dissimilarity Data Sets". In: *Neural Computation* 22.9, pp. 2229–2284. DOI: 10.1162/NECO_a_00012.
Hammer, Barbara, Daniela Hofmann, et al. (2014). "Learning vector quantization for (dis-)similarities". In: *Neurocomputing* 131, pp. 43–51. DOI: 10.1016/j.neucom.2013.05.054.
Hargrove, Levi J., Kevin Englehart, and Bernard Hudgins (2008). "A training strategy to reduce classification degradation due to electrode displacements in pattern recognition based myoelectric control". In: *Biomedical signal processing and control* 3.2, pp. 175–180. DOI: 10.1016/j.bspc.2007.11.005.
Hashimoto, Kosuke et al. (2006). "KEGG as a glycome informatics resource". In: *Glycobiology* 16.5, 63R–70R. DOI: 10.1093/glycob/cwj010.
Head, Andrew et al. (2017). "Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis". In: *Proceedings of the Fourth ACM Conference on Learning@Scale (L@S 2017)*. Ed. by Claudia Urrea, Justin Reich, and Candace Thille, pp. 89–98. DOI: 10.1145/3051457.3051467.
Henikoff, Steven and Jorja G. Henikoff (1992). "Amino acid substitution matrices from protein blocks". In: *Proceedings of the National Academy of Sciences* 89.22, pp. 10915–10919. URL: http://www.pnas.org/content/89/22/10915.
Hicks, Andrew, Barry Peddycord, and Tiffany Barnes (2014). "Building Games to Learn from Their Players: Generating Hints in a Serious Game". In: *Proceedings of the 12th International Conference Intelligent Tutoring Systems (ITS 2014)*. Ed. by Stefan Trausan-Matu et al., pp. 312–317. DOI: 10.1007/978-3-319-07221-0_39.
Hochreiter, Sepp and Jürgen Schmidhuber (1997). "Long Short-Term Memory". In: *Neural Computation* 9.8, pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.
Hodgetts, Carl J., Ulrike Hahn, and Nick Chater (2009). "Transformation and alignment in similarity". In: *Cognition* 113.1, pp. 62–79. DOI: 10.1016/j.cognition.2009.07.010.
Hoff, Peter D, Adrian E Raftery, and Mark S Handcock (2002). "Latent Space Approaches to Social Network Analysis". In: *Journal of the American Statistical Association* 97.460, pp. 1090–1098. DOI: 10.1198/016214502388618906.
Hoffgen, Klaus, Hans Simon, and Kevin Van Horn (1995). "Robust Trainability of Single Neurons". In: *Journal of Computer and System Sciences* 50.1, pp. 114–125. DOI: 10.1006/jcss.1995.1011.
Hofmann, Daniela et al. (2014). "Learning interpretable kernelized prototype-based models". In: *Neurocomputing* 141, pp. 84–96. DOI: 10.1016/j.neucom.2014.03.003.
Hofmann, Thomas, Bernhard Schölkopf, and Alexander J. Smola (2008). "Kernel Methods in Machine Learning". In: *The Annals of Statistics* 36.3, pp. 1171–1220. URL: http://www.jstor.org/stable/25464664.
Holland, Paul W., Kathryn B. Laskey, and Samuel Leinhardt (1983). "Stochastic Block Models: First Steps". In: *Social Networks* 5, pp. 109–137. DOI: 10.1016/0378-8733(83)90021-7.
Hourai, Yuichiro, Tatsuya Akutsu, and Yutaka Akiyama (2004). "Optimizing substitution matrices by separating score distributions". In: *Bioinformatics* 20.6, pp. 863–873. DOI: 10.1093/bioinformatics/btg494.
Hu, Junlin, Jiwen Lu, and Yap-Peng Tan (2014). "Discriminative Deep Metric Learning for Face Verification in the Wild". In: *Proceedings of the 27th IEEE Conference on Computer*

Vision and Pattern Recognition (CVPR 2014). Ed. by Sven Dickinson et al., pp. 1875–1882. DOI: 10.1109/CVPR.2014.242.

Huang, Jiayuan et al. (2007). "Correcting Sample Selection Bias by Unlabeled Data". In: *Proceedings of the 19th Advances in Neural Information Processing Systems Conference (NIPS 2006)*. Ed. by Bernhard Schölkopf, John C. Platt, and T. Hoffman, pp. 601–608. URL: http://papers.nips.cc/paper/3075-correcting-sample-selection-bias-by-unlabeled-data.

Huang, Wen-bing et al. (2015). "Scalable Gaussian Process Regression Using Deep Neural Networks". In: *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI 2015)*. Ed. by Michael Wooldridge and Qiang Yang, pp. 3576–3582. URL: https://www.ijcai.org/Abstract/15/503.

Irsoy, Ozan and Claire Cardie (2014). "Deep Recursive Neural Networks for Compositionality in Language". In: *Proceedings of the 27th International Conference on Advances in Neural Information Processing Systems (NIPS 2014)*. Ed. by Z. Ghahramani et al., pp. 2096–2104. URL: http://papers.nips.cc/paper/5551-deep-recursive-neural-networks-for-compositionality-in-language.pdf.

Jaeger, Herbert and Harald Haas (2004). "Harnessing Nonlinearity: Predicting Chaotic Systems and Saving Energy in Wireless Communication". In: *Science* 304.5667, pp. 78–80. DOI: 10.1126/science.1091277.

Jäkel, Frank, Bernhard Schölkopf, and Felix A. Wichmann (2008). "Similarity, kernels, and the triangle inequality". In: *Journal of Mathematical Psychology* 52.5, pp. 297–303. DOI: 10.1016/j.jmp.2008.03.001.

Jiang, Ning et al. (2014). "Intuitive, online, simultaneous, and proportional myoelectric control over two degrees-of-freedom in upper limb amputees". In: *IEEE Transactions on Neural Systems and Rehabilitation Engineering* 22.3, pp. 501–510. DOI: 10.1109/TNSRE.2013.2278411.

Johnson, Stephen (1967). "Hierarchical clustering schemes". In: *Psychometrika* 32.3, pp. 241–254. DOI: 10.1007/BF02289588.

Kann, Maricel, Bin Qian, and Richard A. Goldstein (2000). "Optimization of a new score function for the detection of remote homologs". In: *Proteins: Structure, Function, and Bioinformatics* 41.4, pp. 498–503. DOI: 10.1002/1097-0134(20001201)41:4<498::AID-PROT70>3.0.CO;2-3.

Keogh, Eamonn and Chotirat Ann Ratanamahatana (2005). "Exact indexing of dynamic time warping". In: *Knowledge and Information Systems* 7.3, pp. 358–386. DOI: 10.1007/s10115-004-0154-9.

Khushaba, Rami N. et al. (2014). "Towards limb position invariant myoelectric pattern recognition using time-dependent spectral features". In: *Neural Networks* 55, pp. 42–58. DOI: 10.1016/j.neunet.2014.03.010.

Koedinger, Kenneth R. et al. (2013). "New potentials for data-driven intelligent tutoring system development and optimization". In: *AI Magazine* 34.3, pp. 27–41. DOI: 10.1609/aimag.v34i3.2484.

Kohonen, Teuvo (1995). "Learning Vector Quantization". In: *Self-Organizing Maps*. Berlin/Heidelberg, Germany: Springer, pp. 175–189. DOI: 10.1007/978-3-642-97610-0_6.

Köstinger, Martin et al. (2012). "Large scale metric learning from equivalence constraints". In: *Proceedings of the 25th IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2012)*. Ed. by Rama Chellappa et al., pp. 2288–2295. DOI: 10.1109/CVPR.2012.6247939.

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

Kulis, Brian, Kate Saenko, and Trevor Darrell (2011). "What you saw is not what you get: Domain adaptation using asymmetric kernel transforms". In: *Proceedings of the 24th IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2011)*. Ed. by Terrance Boult et al., pp. 1785–1792. DOI: 10.1109/CVPR.2011.5995702.

Kusner, Matt J., Brooks Paige, and José Miguel Hernández-Lobato (2017). "Grammar Variational Autoencoder". In: *Proceedings of the 34th International Conference on Machine Learning*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research, pp. 1945–1954. URL: http://proceedings.mlr.press/v70/kusner17a.html.

Kwok, James Tin-Yau and Ivor Wai-Hung Tsang (2004). "The pre-image problem in kernel methods". In: *IEEE Transactions on Neural Networks* 15.6, pp. 1517–1525. DOI: 10.1109/TNN.2004.837781.

Lazar, Timotej and Ivan Bratko (2014). "Data-Driven Program Synthesis for Hint Generation in Programming Tutors". In: *Proceedings of the 12th International Conference on Intelligent Tutoring Systems (ITS 2014)*. Ed. by Stefan Trausan-Matu et al., pp. 306–311. DOI: 10.1007/978-3-319-07221-0_38.

Le, Nguyen-Thinh (2016). "A Classification of Adaptive Feedback in Educational Systems for Programming". In: *Systems* 4.2, p. 22. DOI: 10.3390/systems4020022.

Le, Nguyen-Thinh and Niels Pinkwart (2014). "Towards a classification for programming exercises". In: *Proceedings of the 2nd Workshop on AI-supported Education for Computer Science (AIEDCS 2014)*. Ed. by Kristy Elizabeth Boyer et al., pp. 51–60. URL: https://cses.informatik.hu-berlin.de/pubs/2014/aiedcs/Towards-a-Classification-for-Programming-Exercises.pdf.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (2015). "Deep learning". In: *Nature* 521, pp. 436–444. DOI: 10.1038/nature14539.

Levenshtein, Vladimir (1965). "Binary codes capable of correcting deletions, insertions, and reversals". In: *Soviet Physics Doklady* 10.8, pp. 707–710.

Liao, Shengcai et al. (2015). "Person re-identification by Local Maximal Occurrence representation and metric learning". In: *Proceedings of the 28th IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2015)*. Ed. by Horst Bischof et al., pp. 2197–2206. DOI: 10.1109/CVPR.2015.7298832.

Liben-Nowell, David and Jon Kleinberg (2007). "The link-prediction problem for social networks". In: *Journal of the American Society for Information Science and Technology* 58.7, pp. 1019–1031. DOI: 10.1002/asi.20591.

Lichtenwalter, Ryan N., Jake T. Lussier, and Nitesh V. Chawla (2010). "New Perspectives and Methods in Link Prediction". In: *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2010)*. Ed. by Bharat Rao et al., pp. 243–252. DOI: 10.1145/1835804.1835837.

Lim, Daryl, Gert Lanckriet, and Brian McFee (2013). "Robust Structural Metric Learning". In: *Proceedings of the 30th International Conference on Machine Learning (ICML 2013)*. Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. Proceedings of Machine Learning Research 1, pp. 615–623. URL: http://proceedings.mlr.press/v28/lim13.html.

Liu, Dong C. and Jorge Nocedal (1989). "On the limited memory BFGS method for large scale optimization". In: *Mathematical Programming* 45.1, pp. 503–528. DOI: 10.1007/BF01589116.

Long, Mingsheng et al. (2015). "Learning Transferable Features with Deep Adaptation Networks". In: *Proceedings of the 32nd International Conference on Machine Learning (ICML 2015)*. Ed. by Francis Bach and David Blei. Vol. 37. Proceedings of Machine Learning Research, pp. 97–105. URL: http://proceedings.mlr.press/v37/long15.html.

Losing, Viktor, Barbara Hammer, and Heiko Wersing (2016a). "Choosing the Best Algorithm for an Incremental On-line Learning Task". In: *Proceedings of the 24th European Symposium on Artificial Neural Networks (ESANN 2016)*. Ed. by Michel Verleysen. URL: http://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2016-71.pdf.

— (2016b). "KNN Classifier with Self Adjusting Memory for Heterogeneous Concept Drift". In: *Proceedings of the 16th IEEE International Conference on Data Mining (ICDM 2016)*. Ed. by Ricardo Baeza-Yates et al., pp. 291–300. DOI: 10.1109/ICDM.2016.0040.

Lowe, David G. (1999). "Object recognition from local scale-invariant features". In: *Proceedings of the 7th IEEE International Conference on Computer Vision (ICCV1999)*. Ed. by John K. Tsotsos et al., pp. 1150–1157. DOI: 10.1109/ICCV.1999.790410.

Lü, Linyuan and Tao Zhou (2011). "Link prediction in complex networks: A survey". In: *Physica A: Statistical Mechanics and its Applications* 390.6, pp. 1150–1170. DOI: 10.1016/j.physa.2010.11.027.

Lundsieen, Claes, John Philip, and Erik Granum (1980). "Quantitative analysis of 6985 digitized trypsin G-banded human metaphase chromosomes". In: *Clinical Genetics* 18.5, pp. 355–370. DOI: 10.1111/j.1399-0004.1980.tb02296.x.

Lynch, Collin et al. (2009). "Concepts, Structures, and Goals: Redefining Ill-Definedness". In: *International Journal of Artificial Intelligence in Education* 19.3, pp. 253–266. URL: http://www.ijaied.org/pub/1294/.

MacQueen, J. (1967). "Some methods for classification and analysis of multivariate observations". In: *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, pp. 281–297. URL: https://projecteuclid.org/euclid.bsmsp/1200512992.

Marin, Victor J. et al. (2017). "Automated Personalized Feedback in Introductory Java Programming MOOCs". In: *Proceedings of the 33rd International IEEE Conference on Data Engineering (ICDE 2017)*. Ed. by Chaitanya Baru, Bhavani Thuraisingham, and Yanlei Diao Yannis Papakonstantinou, pp. 1259–1270. DOI: 10.1109/ICDE.2017.169.

Markman, Arthur (1998). *Knowledge Representation*. New York, NY, USA: Psychology Press. ISBN: 9781134802906.

McKenna, Aaron et al. (2010). "The Genome Analysis Toolkit: A MapReduce framework for analyzing next-generation DNA sequencing data". In: *Genome Research* 20.9, pp. 1297–1303. DOI: 10.1101/gr.107524.110.

Medin, Douglas, Robert Goldstone, and Dedre Genter (1993). "Respects for similarity". In: *Psychological Review* 100.2, pp. 254–278. DOI: 10.1037/0033-295X.100.2.254.

Mehrkanoon, Siamak and Johan A. K. Suykens (2018). "Deep hybrid neural-kernel networks using random Fourier features". In: *Neurocomputing* 298, pp. 46–54. DOI: https://doi.org/10.1016/j.neucom.2017.12.065.

Mikolov, Tomas et al. (2013). "Distributed Representations of Words and Phrases and their Compositionality". In: *Proceedings of the 26th conference on Advances in Neural Information Processing Systems (NIPS 2013)*. Ed. by C. J. C. Burges et al., pp. 3111–3119. URL: http://papers.nips.cc/paper/5021-distributed-representations-of-words-andphrases.

Mokbel, Bassam, Sebastian Gross, et al. (2013). "Domain-Independent Proximity Measures in Intelligent Tutoring Systems". In: *Proceedings of the 6th International Conference on Educational Data Mining (EDM 2013)*. Ed. by S. K. D'Mello, R. A. Calvo, and A. Olney. Memphis, Tennessee, USA, pp. 334–335. URL: http://www.educationaldatamining.org/EDM2013/papers/rn_paper_68.pdf.

Muceli, Silvia, Ning Jiang, and Dario Farina (2014). "Extracting Signals Robust to Electrode Number and Shift for Online Simultaneous and Proportional Myoelectric Con-

trol by Factorization Algorithms". In: *IEEE Transactions on Neural Systems and Rehabilitation Engineering* 22.3, pp. 623–633. DOI: 10.1109/TNSRE.2013.2282898.
Murray, T., S. Blessing, and S. Ainsworth (2003). *Authoring tools for advanced technology learning environments: Toward cost-effective adaptive, interactive and intelligent educational software*. Berlin/Heidelberg, Germany: Springer. ISBN: 978-94-017-0819-7.
Mussweiler, Thomas (2003). "Comparison processes in social judgment: mechanisms and consequences". In: *Psychological review* 110.3, p. 472. DOI: 10.1037/0033-295X.110.3.472.
Nadaraya, E. A. (1964). "On Estimating Regression". In: *Theory of Probability & Its Applications* 9.1, pp. 141–142. DOI: 10.1137/1109020.
Napier, John R. (1956). "The prehensile movements of the human hand". In: *The Journal of Bone and Joint Surgery. British volume* 38-B.4, pp. 902–913. DOI: 10.1302/0301-620X.38B4.902.
Navarro, Gonzalo (2001). "A Guided Tour to Approximate String Matching". In: *ACM Computing Surveys* 33.1, pp. 31–88. DOI: 10.1145/375360.375365.
Nebel, David, Barbara Hammer, et al. (2015). "Median variants of learning vector quantization for learning of dissimilarity data". In: *Neurocomputing* 169, pp. 295–305. DOI: 10.1016/j.neucom.2014.12.096.
Nebel, David, Marika Kaden, et al. (2017). "Types of (dis-)similarities and adaptive mixtures thereof for improved classification learning". In: *Neurocomputing* 268, pp. 42–54. DOI: 10.1016/j.neucom.2016.12.091.
Nguyen, Andy et al. (2014). "Codewebs: Scalable Homework Search for Massive Open Online Programming Courses". In: *Proceedings of the 23rd International Conference on World Wide Web (WWW 2014)*. Ed. by Chin-Wan Chung et al., pp. 491–502. DOI: 10.1145/2566486.2568023.
Nosofsky, Robert (1992). "Similarity scaling and cognitive process models". In: *Annual Review of Psychology* 43.1, pp. 25–53. DOI: 10.1146/annurev.ps.43.020192.000325.
Oh Song, Hyun et al. (2016). "Deep Metric Learning via Lifted Structured Feature Embedding". In: *Proceedings of the 29th IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2016)*. Ed. by Ruzena Bajcsy et al., pp. 4004–4012. DOI: 10.1109/CVPR.2016.434.
Oncina, Jose and Marc Sebban (2006). "Learning stochastic edit distance: Application in handwritten character recognition". In: *Pattern Recognition* 39.9, pp. 1575–1587. DOI: 10.1016/j.patcog.2006.03.011.
Ortiz-Catalan, Max, Rickard Brånemark, and Bo Håkansson (2013). "BioPatRec: A modular research platform for the control of artificial limbs based on pattern recognition algorithms". In: *Source Code for Biology and Medicine* 8.1, pp. 1–18. DOI: 10.1186/1751-0473-8-11.
Ortiz-Catalan, Max, Rickard Brånemark, Bo Håkansson, and Jean Delbeke (2012). "On the viability of implantable electrodes for the natural control of artificial limbs: Review and discussion". In: *Biomedical engineering online* 11.1, p. 33. DOI: 10.1186/1475-925X-11-33.
Paaßen, Benjamin (2016a). *Java Sorting Programs*. DOI: 10.4119/unibi/2900684.
— (2016b). *MiniPalindrome*. DOI: 10.4119/unibi/2900666.
Pan, Lizhi et al. (2015). "Improving robustness against electrode shift of high density EMG for myoelectric control through common spatial patterns". In: *Journal of NeuroEngineering and Rehabilitation* 12.1, pp. 1–16. DOI: 10.1186/s12984-015-0102-9.
Pan, Sinno J. and Qiang Yang (2010). "A Survey on Transfer Learning". In: *IEEE Transactions on Knowledge and Data Engineering* 22.10, pp. 1345–1359. DOI: 10.1109/TKDE.2009.191.

Pane, John F. et al. (2014). "Effectiveness of Cognitive Tutor Algebra I at Scale". In: *Educational Evaluation and Policy Analysis* 36.2, pp. 127–144. DOI: 10.3102/0162373713507480.

Papageorgiou, Markos (1990). "Dynamic modeling, assignment, and route guidance in traffic networks". In: *Transportation Research Part B: Methodological* 24.6, pp. 471–495. DOI: 10.1016/0191-2615(90)90041-V.

Pasquina, Paul F. et al. (2015). "First-in-man demonstration of a fully implanted myoelectric sensors system to control an advanced electromechanical prosthetic hand". In: *Journal of Neuroscience Methods* 244, pp. 85–93. DOI: 10.1016/j.jneumeth.2014.07.016.

Pawlik, Mateusz and Nikolaus Augsten (2011). "RTED: A Robust Algorithm for the Tree Edit Distance". In: *Proceedings of the VLDB Endowment* 5.4, pp. 334–345. DOI: 10.14778/2095686.2095692.

— (2016). "Tree edit distance: Robust and memory-efficient". In: *Information Systems* 56, pp. 157–173. DOI: 10.1016/j.is.2015.08.004.

Pekalska, Elzbieta and Robert Duin (2005). *The Dissimilarity Representation for Pattern Recognition: Foundations And Applications (Machine Perception and Artificial Intelligence)*. River Edge, NJ, USA: World Scientific Publishing Co., Inc. ISBN: 9812565302.

Pennington, Jeffrey, Richard Socher, and Christopher D. Manning (2014). "GloVe: Global Vectors for Word Representation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP 2014)*. Ed. by Alessandro Moschitti, Bo Pang, and Walter Daelemans, pp. 1532–1543. URL: http://www.aclweb.org/anthology/D14-1162.

Petersen, Kaare Brandt and Michael Syskind Pedersen (2012). *The Matrix Cookbook*. URL: https://www.math.uwaterloo.ca/~hwolkowi/matrixcookbook.pdf.

Piech, Chris, Jonathan Huang, et al. (2015). "Learning Program Embeddings to Propagate Feedback on Student Code". In: *Proceedings of the 32nd International Conference on Machine Learning (ICML 2015)*. Ed. by Francis Bach and David Blei, pp. 1093–1102. URL: http://proceedings.mlr.press/v37/piech15.html.

Piech, Chris, Mehran Sahami, et al. (2015). "Autonomously Generating Hints by Inferring Problem Solving Policies". In: *Proceedings of the Second ACM Conference on Learning @ Scale (L@S 2015)*. Ed. by Gregor Kiczales, Daniel M. Russel, and Beverly Woolf, pp. 195–204. DOI: 10.1145/2724660.2724668.

Pollack, Jordan B. (1990). "Recursive distributed representations". In: *Artificial Intelligence* 46.1, pp. 77–105. DOI: 10.1016/0004-3702(90)90005-K.

Price, Thomas W. and Tiffany Barnes (2015). "An Exploration of Data-Driven Hint Generation in an Open-Ended Programming Problem". In: *Workshops Proceedings of the 8th International Conference on Educational Data Mining (EDM 2015)*. Ed. by Olga Christina Santos et al. URL: http://ceur-ws.org/Vol-1446/GEDM_2015_Submission_4.pdf.

Price, Thomas W., Yihuan Dong, and Tiffany Barnes (2016). "Generating Data-driven Hints for Open-ended Programming". In: *Proceedings of the 9th International Conference on Educational Data Mining (EDM 2016)*. Ed. by Tiffany Barnes, Min Chi, and Mingyu Feng. URL: http://www.educationaldatamining.org/EDM2016/proceedings/paper_33.pdf.

Price, Thomas W., Yihuan Dong, and Dragan Lipovac (2017). "iSnap: Towards Intelligent Tutoring in Novice Programming Environments". In: *Proceedings of the 2017 ACM Technical Symposium on Computer Science Education (SIGCSE)*. Ed. by Michael Caspersen et al., pp. 483–488. DOI: 10.1145/3017680.3017762.

Price, Thomas W., Rui Zhi, and Tiffany Barnes (2017a). "Evaluation of a Data-driven Feedback Algorithm for Open-ended Programming". In: *Proceedings of the 10th In-*

ternational Conference on Educational Datamining (EDM 2017). Ed. by Xiangen Hu et al., pp. 192–197. URL: http://educationaldatamining.org/EDM2017/proc_files/papers/paper_36.pdf.

— (2017b). "Hint Generation Under Uncertainty: The Effect of Hint Quality on Help-Seeking Behavior". In: *Proceedings of the 18th International Conference on Artificial Intelligence in Education (AIED 2017)*. Ed. by Elisabeth André et al., pp. 311–322. DOI: 10.1007/978-3-319-61425-0_26.

Price, Thomas W., Rui Zhi, Yihuan Dong, et al. (2018). "The Impact of Data Quantity and Source on the Quality of Data-Driven Hints for Programming". In: *Proceedings of the 19th International Conference on Artificial Intelligence in Education (AIED 2018)*. Ed. by Carolyn Penstein Rosé et al., pp. 476–490. DOI: 10.1007/978-3-319-93843-1_35.

Raichle, Katherine A. et al. (2008). "Prosthesis use in persons with lower- and upper-limb amputation". In: *Journal of Rehabilitation Research and Development* 45.7, pp. 961–972. DOI: 10.1682/JRRD.2007.09.0151.

Rasmussen, Carl Edward and Christopher K. I. Williams (2005). *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. Cambridge, MA, USA: The MIT Press.

Ristad, Eric and Peter Yianilos (1998). "Learning string-edit distance". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20.5, pp. 522–532. DOI: 10.1109/34.682181.

Rivers, Kelly and Kenneth R. Koedinger (2012). "A Canonicalizing Model for Building Programming Tutors". In: *Proceedings of the 11th International Conference on Intelligent Tutoring Systems (ITS 2012)*. Ed. by Stefano A. Cerri et al., pp. 591–593. DOI: 10.1007/978-3-642-30950-2_80.

— (2014). "Automating Hint Generation with Solution Space Path Construction". In: *Proceedings of the 12th International Conference on Intelligent Tutoring Systems (ITS 2014)*. Ed. by Stefan Trausan-Matu et al., pp. 329–339. DOI: 10.1007/978-3-319-07221-0_41.

— (2015). "Data-Driven Hint Generation in Vast Solution Spaces: a Self-Improving Python Programming Tutor". In: *International Journal of Artificial Intelligence in Education* 27.1, pp. 37–64. DOI: 10.1007/s40593-015-0070-z.

Roberts, S. et al. (2012). "Gaussian processes for time-series modelling". In: *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 371.1984, p. 20110550. DOI: 10.1098/rsta.2011.0550.

Robles-Kelly, Antonio and Edwin R. Hancock (2003). "Edit distance from graph spectra". In: *Proceedings of the 9th IEEE International Conference on Computer Vision (ICCV 2003)*. Ed. by Bob Werner. Vol. 1, pp. 234–241. DOI: 10.1109/ICCV.2003.1238347.

— (2005). "Graph Edit Distance from Spectral Seriation". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 27.3, pp. 365–378. DOI: 10.1109/TPAMI.2005.56.

Saigo, Hiroto, Jean-Philippe Vert, and Tatsuya Akutsu (2006). "Optimizing amino acid substitution matrices with a local alignment kernel". In: *BMC Bioinformatics* 7.1, pp. 246–258. DOI: 10.1186/1471-2105-7-246.

Sammon, John W. (1969). "A Nonlinear Mapping for Data Structure Analysis". In: *IEEE Transactions on Computers* 18.5, pp. 401–409. DOI: 10.1109/T-C.1969.222678.

Sanfeliu, Alberto and King-Sun Fu (1983). "A distance measure between attributed relational graphs for pattern recognition". In: *IEEE Transactions on Systems, Man, and Cybernetics* SMC-13.3, pp. 353–362. DOI: 10.1109/TSMC.1983.6313167.

Sapankevych, N. I. and R. Sankar (2009). "Time Series Prediction Using Support Vector Machines: A Survey". In: *IEEE Computational Intelligence Magazine* 4.2, pp. 24–38. DOI: 10.1109/MCI.2009.932254.

Saralajew, Sascha and Thomas Villmann (2017). “Transfer Learning in Classification based on Manifold Models and its Relation to Tangent Metric Learning”. In: *Proceedings of the 2017 International Joint Conference on Neural Networks (IJCNN 2017)*. Ed. by Yoonsuck Choe and Chrisina Jayne, pp. 1756–1765. DOI: 10.1109/IJCNN.2017.7966063.

Sato, Atshushi and Keiji Yamada (1995). “Generalized Learning Vector Quantization”. In: *Proceedings of the 7th conference on Advances in Neural Information Processing Systems (NIPS 1995)*. Ed. by G. Tesauro, D. Touretzky, and T. Leen. Cambridge, MA, pp. 423–429. URL: https://papers.nips.cc/paper/1113-generalized-learning-vector-quantization.

Scherrer, A. et al. (2008). “Description and simulation of dynamic mobility networks”. In: *Computer Networks* 52.15, pp. 2842–2858. DOI: 10.1016/j.comnet.2008.06.007.

Schneider, Petra, Michael Biehl, and Barbara Hammer (2009a). “Adaptive Relevance Matrices in Learning Vector Quantization”. In: *Neural Computation* 21.12, pp. 3532–3561. DOI: 10.1162/neco.2009.11-08-908.

— (2009b). “Distance Learning in Discriminative Vector Quantization”. In: *Neural Computation* 21.10, pp. 2942–2969. DOI: 10.1162/neco.2009.10-08-892.

Schneider, Petra, Kerstin Bunte, et al. (2010). “Regularization in Matrix Relevance Learning”. In: *IEEE Transactions on Neural Networks* 21.5, pp. 831–840. DOI: 10.1109/TNN.2010.2042729.

Searls, David (2012). “The language of genes”. In: *Nature* 420, pp. 211–217. DOI: 10.1038/nature01255.

Selkow, Stanley M. (1977). “The tree-to-tree editing problem”. In: *Information Processing Letters* 6.6, pp. 184–186. DOI: 10.1016/0020-0190(77)90064-3.

Seo, Sambu and Klaus Obermayer (2003). “Soft Learning Vector Quantization”. In: *Neural Computation* 15.7, pp. 1589–1604. DOI: 10.1162/089976603321891819.

Shepard, Roger (1962). “The analysis of proximities: Multidimensional scaling with an unknown distance function. I.” In: *Psychometrika* 27.2, pp. 125–140. DOI: 10.1007/BF02289630.

Shervashidze, Nino et al. (2011). “Weisfeiler-Lehman Graph Kernels”. In: *Journal of Machine Learning Research* 12.Sep, pp. 2539–2561. URL: http://www.jmlr.org/papers/v12/shervashidze11a.html.

Shih, Benjamin, Kenneth R. Koedinger, and Richard Scheines (2008). “A response time model for bottom-out hints as worked examples”. In: *Proceedings of the 1st International Conference on Educational Datamining (EDM 2008)*. Ed. by Cristobal Romero et al., pp. 117–126. URL: http://www.educationaldatamining.org/EDM2008/uploads/proc/12_Shih_35.pdf.

Shimodaira, Hidetoshi (2000). “Improving predictive inference under covariate shift by weighting the log-likelihood function”. In: *Journal of statistical planning and inference* 90.2, pp. 227–244.

Shumway, Robert and David Stoffer (2013). *Time series analysis and its applications*. Berlin/Heidelberg, Germany: Springer. DOI: 10.1007/978-1-4419-7865-3.

Siederdissen, Christian Höner zu, Sonja J. Prohaska, and Peter F. Stadler (2015). “Algebraic Dynamic Programming over general data structures”. In: *BMC Bioinformatics* 16.19, S2. DOI: 10.1186/1471-2105-16-S19-S2.

Smith, Edward and Douglas Medin (1981). *Categories and Concepts*. Cambridge, MA, USA: Harvard University Press. ISBN: 9780674866270.

Smith, Eliot and Michael Zarate (1992). “Exemplar-based model of social judgment”. In: *Psychological review* 99.1, p. 3. DOI: 10.1037/0033-295X.99.1.3.

Smith, Temple F. and Michael S. Waterman (1981). "Identification of common molecular subsequences". In: *Journal of Molecular Biology* 147.1, pp. 195–197. DOI: 10.1016/0022-2836(81)90087-5.

Snover, Matthew et al. (2006). "A study of translation edit rate with targeted human annotation". In: *Proceedings of the 7th Conference of the Association for Machine Translation in the Americas (AMTA 2006)*. Ed. by Alon Lavie Laurie Gerber Nizar Habash. 6, pp. 223–231. URL: http://mt-archive.info/AMTA-2006-Snover.pdf.

Socher, Richard, Jeffrey Pennington, et al. (2011). "Semi-supervised Recursive Autoencoders for Predicting Sentiment Distributions". In: *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP 2011)*. Ed. by Paola Merlo, Regina Barzilay, and Mark Johnson, pp. 151–161. URL: https://aclanthology.info/pdf/D/D11/D11-1014.pdf.

Socher, Richard, Alex Perelygin, et al. (2013). "Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank". In: *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing (EMNLP 2013)*. Ed. by Timothy Baldwin and Anna Korhonen, pp. 1631–1642. URL: https://nlp.stanford.edu/~socherr/EMNLP2013_RNTN.pdf.

Sperduti, Alessandro and Antonina Starita (1997). "Supervised neural networks for the classification of structures". In: *IEEE Transactions on Neural Networks* 8.3, pp. 714–735. DOI: 10.1109/72.572108.

Stamper, John C. et al. (2012). "Experimental Evaluation of Automatic Hint Generation for a Logic Tutor". In: *International Journal of Artificial Intelligence in Education* 22.1. Ed. by Gautam Biswas et al., pp. 3–18. URL: http://ijaied.org/pub/1333/.

Sugiyama, Masashi et al. (2008). "Direct Importance Estimation with Model Selection and Its Application to Covariate Shift Adaptation". In: *Proceedings of the 20th Advances in Neural Information Processing Systems Conference (NIPS 2007)*. Ed. by John C. Platt et al., pp. 1433–1440. URL: http://papers.nips.cc/paper/3248-direct-importance-estimation-with-model-selection-and-its-application-to-covariate-shift-adaptation.

Sutskever, Ilya, Oriol Vinyals, and Quoc V Le (2014). "Sequence to Sequence Learning with Neural Networks". In: *Proceedings of the 27th International Conference on Advances in Neural Information Processing Systems (NIPS 2014)*. Ed. by Z. Ghahramani et al., pp. 3104–3112. URL: http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural.

Tai, Kuo-Chung (1979). "The Tree-to-Tree Correction Problem". In: *Journal of the ACM* 26.3, pp. 422–433. DOI: 10.1145/322139.322143.

Torgerson, Warren (1952). "Multidimensional scaling: I. Theory and method". In: *Psychometrika* 17.4, pp. 401–419. DOI: 10.1007/BF02288916.

Tversky, Amos (1977). "Features of similarity". In: *Psychological Review* 84.4, pp. 327–352. DOI: 10.1037/0033-295X.84.4.327.

Van der Maaten, Laurens and Geoffrey Hinton (2008). "Visualizing Data using t-SNE". In: *Journal of Machine Learning Research* 9, pp. 2579–2605. URL: http://www.jmlr.org/papers/v9/vandermaaten08a.html.

Van Lehn, Kurt (2006). "The Behavior of Tutoring Systems". In: *International Journal of Artificial Intelligence in Education* 16.3, pp. 227–265. URL: http://ijaied.org/pub/1063/.

Vidovic, Marina et al. (2015). "Improving the Robustness of Myoelectric Pattern Recognition for Upper Limb Prostheses by Covariate Shift Adaptation". In: *IEEE Transactions on Neural Systems and Rehabilitation Engineering* 24.9, pp. 961–970. DOI: 10.1109/TNSRE.2015.2492619.

Vintsyuk, T. K. (1968). "Speech discrimination by dynamic programming". In: *Cybernetics* 4.1, pp. 52–57. DOI: 10.1007/BF01074755.

Wang, Jack, Aaron Hertzmann, and David M. Blei (2006). "Gaussian process dynamical models". In: *Proceedings of the 18th conference on Advances in neural information processing systems (NIPS 2005)*. Ed. by Y. Weiss, P. B. Schölkopf, and J. C. Platt, pp. 1441–1448. URL: https://papers.nips.cc/paper/2783-gaussian-process-dynamical-models.

Weininger, David (1988). "SMILES, a chemical language and information system. 1. Introduction to methodology and encoding rules". In: *Journal of Chemical Information and Computer Sciences* 28.1, pp. 31–36. DOI: 10.1021/ci00057a005.

Weiss, Karl, Taghi M. Khoshgoftaar, and DingDing Wang (2016). "A survey of transfer learning". In: *Journal of Big Data* 3.1. DOI: 10.1186/s40537-016-0043-6.

Yanardag, Pinar and S.V.N. Vishwanathan (2015). "Deep Graph Kernels". In: *Proceedings of the 21th International Conference on Knowledge Discovery and Data Mining (KDD 2015)*. Ed. by Thorsten Joachims and Geoffrey Webb, pp. 1365–1374. DOI: 10.1145/2783258.2783417.

Yang, Jun, Rong Yan, and Alexander G. Hauptmann (2007). "Cross-domain Video Concept Detection Using Adaptive SVMs". In: *Proceedings of the 15th ACM International Conference on Multimedia (MM '07)*. Ed. by Rainer Lienhart et al., pp. 188–197. DOI: 10.1145/1291233.1291276.

Yang, Zichao et al. (2015). "Deep Fried Convnets". In: *Proceedings of the 15th IEEE International Conference on Computer Vision (ICCV 2015)*. Ed. by R. Bajcsy et al., pp. 1476–1483. DOI: 10.1109/ICCV.2015.173.

Yin, Hezheng, Joseph Moghadam, and Armando Fox (2015). "Clustering Student Programming Assignments to Multiply Instructor Leverage". In: *Proceedings of the Second ACM Conference on Learning @ Scale (L@S 2015)*. Ed. by Gregor Kiczales, Daniel M. Russel, and Beverly Woolf, pp. 367–372. DOI: 10.1145/2724660.2728695.

Young, Aaron J., Levi J. Hargrove, and Todd A. Kuiken (2011). "The effects of electrode size and orientation on the sensitivity of myoelectric pattern recognition systems to electrode shift". In: *IEEE Transactions on Biomedical Engineering* 58.9, pp. 2537–2544. DOI: 10.1109/TBME.2011.2159216.

— (2012). "Improving Myoelectric Pattern Recognition Robustness to Electrode Shift by Changing Interelectrode Distance and Electrode Configuration". In: *IEEE Transactions on Biomedical Engineering* 59.3, pp. 645–652. DOI: 10.1109/TBME.2011.2177662.

Zeng, Zhiping et al. (2009). "Comparing Stars: On Approximating Graph Edit Distance". In: *Proceedings of the VLDB Endowment* 2.1, pp. 25–36. DOI: 10.14778/1687627.1687631.

Zhang, Kaizhong and Dennis Shasha (1989). "Simple Fast Algorithms for the Editing Distance between Trees and Related Problems". In: *SIAM Journal on Computing* 18.6, pp. 1245–1262. DOI: 10.1137/0218082.

Zhang, Kaizhong, Rick Statman, and Dennis Shasha (1992). "On the editing distance between unordered labeled trees". In: *Information Processing Letters* 42.3, pp. 133–139. DOI: 10.1016/0020-0190(92)90136-J.

Ziegler-Graham, Kathryn et al. (2008). "Estimating the Prevalence of Limb Loss in the United States: 2005 to 2050". In: *Archives of Physical Medicine and Rehabilitation* 89.3, pp. 422–429. DOI: 10.1016/j.apmr.2007.11.005.

Zimmerman, Kurtis and Chandan R. Rupakheti (2015). "An Automated Framework for Recommending Program Elements to Novices". In: *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015)*. Ed. by Myra Cohen, Lars Grunske, and Michael Whalen, pp. 283–288. DOI: 10.1109/ASE.2015.54.

GLOSSARY

adaptive support vector machine a domain adaptation algorithm which trains a support vector machine as a classifier for the difference between target space labels and predicted labels of a source space model; also refer to J. Yang, Yan, and Hauptmann (2007). 118, 124, 134, 161, 167

algebra ($\mathcal{F}$) a specification of cost functions for all sequence edit types defined by a signature; refer to Definition 2.8. 15–20, 45–51, 54, 56, 139, 161, 195, 198, 200, 205

alphabet ($\mathcal{A}$) an arbitrary set; $\mathcal{A}^*$ denotes the set of all possible sequences over $\mathcal{A}$; refer to Definition 2.5. 13–19, 21–27, 30, 45, 46, 49, 51, 61, 63–66, 139, 161, 165, 166, 174, 175, 178–180, 184, 187, 195, 198, 200, 205, 207, 209, 210, 212–214, 216, 218, 222, 229

asymmetric regularized cross-domain transformation a domain adaptation algorithm which learns a linear mapping between source and target data by maximizing the inner product within classes and minimizing it between classes; also refer to Kulis, Saenko, and Darrell (2011). 118, 124, 134, 161, 167

cooptimal ($\mathcal{M}$) A tree mapping between two trees $\tilde{x}$ and $\tilde{y}$ is called co-optimal if its cost $c(M, \tilde{x}, \tilde{y})$ is equal to the edit distance $D_c(\tilde{x}, \tilde{y})$; refer to Definition 2.14. 25, 31, 59–66, 73, 161, 209–216, 218, 222, 226–228

cost function (c) a function that assigns real-valued costs to sequence edits or tree edits; refer to Definitions 2.6, 2.13 and 6.3. 12, 14–16, 21, 23–25, 27–30, 46, 60, 61, 63–65, 68, 73, 97, 103, 104, 109, 161, 163, 174, 178, 180, 183, 184, 187, 198, 207–210, 212–214, 216, 218, 222, 233

crispness (β) the crispness hyperparameter of the softmin operator; for high crispness, the softmin approaches the actual minimum; refer to Theorem 3.4. 49–51, 54, 161

data point (x, x_i) a single data point from some data space. 116, 161, 163, 164

data space ($\mathcal{X}$) a base set from which a dataset is drawn. 161, 163–165

dataset ($(\vec{x}_1, \ldots, \vec{x}_M), \boldsymbol{X}$) a finite ordered sequence of data points. $\boldsymbol{X}$ denotes the $m \times M$ matrix of data points. A dataset may also refer to a finite ordered sequence of tuples $(\vec{x}_1, y_1), \ldots, (\vec{x}_M, y_M)$ where y_i is the label for the ith data point. 116, 161, 163, 164

distance (d) a function $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ which measures the distance between two data points; refer to Definition 2.1. 7–11, 18, 29, 36–44, 46, 48, 59, 60, 65, 77–79, 82–86, 88–90, 99, 104, 110, 113, 139, 140, 161, 163–165, 172, 174, 231, 233

distance matrix ($\boldsymbol{D}$) a square matrix of pairwise distance between all data points in a set, or, more generally, a square, non-negative, and symmetric matrix with zero diagonal. 51, 56, 84, 85, 161, 166

edit (δ) a function which maps a data point to another data point, e.g. a sequence edit or a tree edit; refer to Definitions 2.5, 2.12, and 6.1. 12, 15, 16, 21, 29, 46, 59, 91–97, 99–104, 106, 107, 109–111, 113, 139, 141, 161, 163, 164, 183, 226

edit distance (d) a distance between structured data, based on an edit set and a cost function; refer to Definitions 2.6, 2.13, and 6.3. 9, 11–32, 36, 38, 39, 42–49, 51–53, 55–57, 59–61, 64–69, 72–75, 82, 89, 91–94, 96–104, 106, 109, 110, 113, 139–141, 161, 163, 178, 180, 183, 198, 200, 205

edit script ($\bar{\delta}$) a list of edits; refer to Definitions 2.5, 2.12, and 6.1. 12–17, 19, 21–25, 27, 45, 46, 92, 95, 97, 98, 100, 104, 106, 161, 174, 176–178, 180, 181, 183, 195, 196, 198, 199, 233

edit set (Δ) a set of edits; refer to Definitions 2.5, 2.12, and 6.1. 13–16, 21–23, 27, 45, 46, 95–97, 100, 104, 161, 163, 198, 233
edit tree grammar ($\mathcal{G}$) a formal grammar which restricts the set of possible script trees that can be generated; refer to Definition 2.10. 15, 18–20, 45–51, 53, 56, 139, 161, 165, 166, 200, 205
electromyography a technique to record electrical activity of muscles. In Chapter 8, electromyographic data is used to infer the intended motion of amputees who use a bionic hand prosthesis. 129, 161, 167
empty list (ϵ) the sequence of length 0 also known as the empty list. 13, 161
Euclidean (d) a distance is called Euclidean if it is equivalent to the square root of an inner product of difference vectors in some space; refer to Definition 2.2. 7–11, 32, 36, 38, 39, 48, 60, 65, 67, 68, 82, 83, 88, 109, 161, 169, 170, 173, 174
expectation maximization a general optimization scheme introduced by Dempster, Laird, and Rubin (1977) used in this thesis for maximizing the likelihood in training or transferring a Gaussian Mixture Models, and for training a MGLVQ model. Also refer to Sections 2.5.2, 7.2, and 2.5.4. 124, 125, 128, 161, 167

forest (X, Y) a list of trees; refer to Definition 2.11. 21–23, 161, 175, 178–180, 183–185, 209–214

Gaussian density function ($\mathcal{N}$) the probability density function for the multivariate Gaussian distribution. $\mathcal{N}(\vec{x}|\vec{\mu}, \mathbf{\Lambda})$ denotes the probability mass assigned to vector $\vec{x}$ by the Gaussian density function with mean $\vec{\mu}$ and precision matrix $\mathbf{\Lambda}$; also refer to Equation 2.31. 161, 164, 166
Gaussian Mixture Model a generative model of vectorial data via a sum of Gaussian density functions; also refer to Section 2.5.2. 33–35, 124, 133, 139, 161, 164, 167
GLVQ cost function (E_{GLVQ}) the loss function of generalized learning vector quantization; also refer to Equation 2.28. 32, 33, 36, 48, 49, 56, 60, 64, 66, 67, 121, 161
graph ($\mathcal{G}$) a tuple $\mathcal{G} = (V, E)$ of a (finite) node set V and an edge set $E \subseteq V \times V$; refer to Definition 5.1. 11–14, 21, 27, 77–82, 86–89, 161, 165

heterogeneous feature augmentation a domain adaptation algorithm which learns linear mappings from target and source space to a shared latent space where a shared support vector machine is trained; also refer to Duan, Xu, and I. Tsang (2012). 118, 124, 134, 161, 167

identity matrix ($\boldsymbol{I}^m$, $\boldsymbol{I}^{m \times n}$) the $m \times m$ or $m \times n$ identity matrix, that is, the matrix which contains only zeros except for ones on the diagonal. 40, 124, 161

kernel (k) a function $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ which measures the inner product between two data points; refer to Definition 2.3. 8, 9, 11, 12, 38, 39, 68, 77–79, 82–85, 89, 90, 103, 106, 140, 161, 165, 169, 170, 231, 234
kernel matrix ($\boldsymbol{K}$) a matrix of pairwise inner products between all points in a set, or, more generally, a square, symmetric, positive semi-definite matrix. 9, 41, 84, 85, 161, 235
keyroots ($\mathcal{K}(\tilde{x})$) the set of keyroots for the tree $\tilde{x}$; refer to Definition 2.15. 26, 28, 161, 178, 189

label (y, y_i) the label for the ith data point. In case of a classification scenario, this is assumed to be a natural number in the range $\{1, \dots, L\}$. In a time series prediction scenario, it is assumed to be a data point from the data space. 116, 161, 163–165
learning vector quantization a classification approach where a dataset is represented by a few prototypes which can classify the data by assigning the label of the closest

prototype. Variants include generalized learning vector quantization (GLVQ) with a cost function, generalized matrix learning vector quantization (GMLVQ) which also learns a metric, and local generalized matrix learning vector quantization (LGMLVQ) which learns an individual metric for each prototype. Also refer to Section 2.5. 8, 11, 31–33, 36, 44, 48, 60, 66, 68, 73, 120, 128, 133, 139, 161, 164, 165, 167

mean ($\vec{\mu}$, $\vec{\mu}_k$) the mean vector of the kth Gaussian in a Gaussian mixture model. 33, 161
model (f) a function which represents a classification or regression model. 161, 165
model space ($\mathcal{F}$) the base set from which models are drawn. 161
Moore-Penrose Pseudo-Inverse ($A^{\dagger}$) the Moore-Penrose Pseudo-Inverse of matrix $\boldsymbol{A}$ which is defined as $\boldsymbol{A}^{\top} \cdot (\boldsymbol{A} \cdot \boldsymbol{A}^{\top})^{-1}$. 67, 123, 161

node (u, v) a node in an graph. 161
nonterminal symbol (A, B, S) an auxiliary symbol in an edit tree grammar; the overall set of nonterminal symbls is denoted as Φ; refer to Definition 2.10. 18, 161, 166, 200, 206

outermost right leaf ($rl_{\tilde{x}}(i)$) the pre-order index of the outermost right leaf corresponding to the subtree $\tilde{x}_i$; refer to Definition 2.15. 26, 27, 161, 178, 189, 212, 213

precision matrix ($\boldsymbol{\Lambda}$, $\boldsymbol{\Lambda}_k$) the precision matrix of of the kth Gaussian in a Gaussian mixture model, or the relevance matrix of a GMLVQ model, or the relevance matrix of the kth prototype in a LGMLVQ model. 33, 35, 36, 133, 161, 190
probability density function (p) a probability density function over $\mathbb{R}^m$ for some $m \in \mathbb{N}$. $p(\vec{x})$ denotes the probability mass assigned to vector $\vec{x}$. 161
probability distribution (P) a probability distribution over some finite set. $P(x)$ denotes the probability assigned to outcome x. 161
production rule ($A ::= \delta(x, B, y)$) a rule of an edit tree grammar; the overall set of production rules is denoted as $\mathcal{R}$; refer to Definition 2.10. 18, 161, 166, 200, 206
projection matrix ($\boldsymbol{\Omega}$, $\boldsymbol{\Omega}_k$) the linear projection matrix of a generalized matrix learning vector quantization (GMLVQ) model or the linear projection matrix of the kth prototype in a local generalized matrix learning vector quantization (LGMLVQ) model; also refer to Equation 2.29. 32, 36, 67, 120, 161
prototype ($\vec{w}_k$, w_k, $\boldsymbol{W}$) the kth prototype of a learning vector quantization model. A prototype is a point from the data space. $\boldsymbol{W}$ denotes the matrix of all K prototypes of a learning vector quantization model. 31, 32, 36–38, 48, 51, 54, 56, 60, 64, 73, 120, 121, 125, 127, 132, 134, 161, 164, 165
prototype label (z_k) the label of the kth prototype of a learning vector quantization model. 31, 36, 161
pseudo-Euclidean (d) a distance is pseudo-Euclidean if it is symmetric and self-equal; refer to Definition 2.4 and Theorem 2.2. 9–11, 14, 25, 46, 77, 78, 82–84, 90, 104, 140, 161, 171, 172, 231, 233

radial basis function ($k_{d,\xi}$) also known as Gaussian kernel; the function $k_{d,\xi}(x, y) := \exp(-0.5 \cdot d(x, y)^2/\xi)$ with the hyperparameter $\xi > 0$ called *bandwidth*; also refer to Equation 2.44. 38, 39, 52, 68, 82–84, 86, 88, 89, 106, 161
regularization constant (λ) a positive real number regulating the strength of regularization. 161

script tree ($\tilde{\delta}$) a special kind of tree which takes symbols from an alphabet as leaves and edit types from a signature as inner nodes; refer to Definition 2.9. 16, 17, 19, 20, 45, 46, 161, 164, 166, 195–198, 200–202

sequence ($\bar{x}$, $\bar{y}$) a finite-length list over some alphabet $\mathcal{A}$; refer to Definition 2.5. 12–22, 25, 27, 32, 36, 42–50, 52–57, 60, 66, 82, 83, 139, 161, 164, 166, 195, 196, 198, 200, 205, 231
sequence edit (δ) a function which maps a sequence to another sequence; refer to Definition 2.5. 13, 17, 22, 95, 161, 163, 166, 195, 196, 199
signature ($\mathcal{S}$) a specification of *types* of sequence edits; refer to Definition 2.7. 13, 15–19, 45, 46, 49, 51, 53, 56, 139, 161, 163, 165, 195, 198, 200, 205
similarity matrix (S) a matrix of pairwise similarities between all data points in a set, for example the double-centered version of a distance matrix. 161
softmin (softmin) a differentiable approximation of the minimum operation; refer to Equation 3.4. 49, 57, 161, 163, 202, 205, 206
standard deviation (σ) the standard deviation of a sample or of a Gaussian density function. 161

tree ($\tilde{x}$, $\tilde{y}$) a tree over some alphabet $\mathcal{A}$, defined by a root and a list of trees as children; refer to Definition 2.11. 12, 21–28, 30–32, 38, 53, 57, 59–70, 73–75, 89, 98–102, 109, 111, 139, 161, 163–166, 174, 175, 177, 178, 180, 183, 187, 207, 208, 216, 218, 222, 226
tree edit (δ) a function which maps a tree to another tree; refer to Definition 2.11. 21–24, 95, 161, 163, 177
tree language ($\mathcal{L}(\mathcal{G}, \mathcal{A})$) the script tree language produced by the edit tree grammar $\mathcal{G}$ over the alphabet $\mathcal{A}$, that is, the set of all script trees which can be generated from the starting nonterminal symbol of $\mathcal{G}$ using the production rules of $\mathcal{G}$; refer to Definition 2.10. 18, 19, 161
tree mapping (M) a set of tuples which assigns nodes from one tree to another tree; refer to Definition 2.14. 25–27, 30, 31, 60–64, 73, 161, 163, 178, 180–183, 185–187, 209, 213–216, 218, 219, 221–223, 225–228
tree set ($\mathcal{T}(\mathcal{A})$) the set of all possible trees over alphabet $\mathcal{A}$; refer to Definition 2.11. 161

worst case complexity ($\mathcal{O}(p(n))$) the set of functions which are bounded above by $c \cdot p(n)$ for some sufficiently big n, some function $p : \mathbb{N} \to \mathbb{N}$ and some constant factor c. 161

yield ($\mathcal{Y}(\tilde{\delta})$) The concatenation of all left-hand-side and right-hand-side leaves of the script tree $\tilde{\delta}$; refer to Definition 2.9. 16, 17, 19, 20, 45, 46, 161, 195

ACRONYMS

1-NN one-nearest neighbor regression. 38–40, 82, 84, 86, 88–90, 110, 111, 139, 161, 231

a-SVM adaptive support vector machine. 118, 119, 124–128, 134–137, 161
ADP algebraic dynamic programming. 13–15, 20, 21, 44–46, 139, 161
ARC-t asymmetric regularized cross-domain transformation. 118, 119, 124–128, 134, 136, 137, 161

BEDL em**b**edding **e**dit **d**istance **l**earning. 59, 60, 67–73, 139, 140, 161

CHF Continuous Hint Factory. 91, 92, 96, 102, 103, 106, 109, 111–113, 161

EM expectation maximization. 124–128, 134–137, 161
EMG electromyography. 129–133, 137, 161

GESL good edit similarity learning. 29, 31, 52, 54, 56, 60, 61, 68–73, 161
GLVQ generalized learning vector quantization. 32, 33, 36, 37, 124–128, 132, 135–137, 161, 165, 193
GMLVQ generalized matrix learning vector quantization. 32, 33, 36, 66, 67, 120, 125–127, 132–136, 139, 161, 165
GMM Gaussian Mixture Model. 33–35, 161
GPR Gaussian process regression. 8, 38–41, 78, 82–86, 88–90, 96, 103–106, 109–111, 139, 140, 161, 231

HFA heterogeneous feature augmentation. 118, 119, 124–128, 134, 136, 137, 161

KNN k-nearest neighbor. 68–72, 74, 161
KR kernel regression. 38–40, 78, 82, 84, 86, 88–90, 139, 161, 231

LGMLVQ local generalized matrix learning vector quantization. 33, 127, 133, 134, 161, 165
lGMM labeled Gaussian Mixture Model. 34–36, 121–124, 126, 133, 134, 139, 161
LVQ learning vector quantization. 31, 32, 36, 123, 124, 161

MGLVQ median generalized learning vector quantization. 36, 38, 60, 64, 65, 68–74, 84, 161, 164

rBCM robust Bayesian committee machine. 41, 83–86, 88–90, 161, 231, 232
RGLVQ relational generalized learning vector quantization. 8, 11, 36, 46, 48, 49, 51, 52, 54–56, 60, 69, 84, 161
RMSE root mean square error. 86, 88, 89, 110–112, 161

slGMM labeled Gaussian Mixture Model with shared precision matrix. 35, 36, 123, 124, 133, 134, 161, 236
SVM support vector machine. 51, 52, 54–56, 68–72, 74, 161

A PROOFS

A.1 PROOF OF THEOREM 2.1

Recall the theorem we intend to prove.

Let $\mathcal{X}$ be some arbitrary set and let $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$. Then it holds: d is Euclidean if and only if there exists a kernel $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$, such that for all $x, y \in \mathcal{X}$: $d(x,y)^2 = k(x,x) - 2 \cdot k(x,y) + k(y,y)$.

Now, let $\mathcal{X} = \{x_1, \ldots, x_M\}$ be a finite set and let $s : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$. It holds: s is a kernel if and only if the matrix $\boldsymbol{S} \in \mathbb{R}^{M \times M}$ with entries $\boldsymbol{S}_{i,j} = s(x_i, x_j)$ is symmetric and positive semi-definite.

Further, let $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ be a self-equal and symmetric function on $\mathcal{X}$, and let s_d be defined as follows.

$$s_d(x_i, x_j) := \frac{1}{2}\Big(- d(x_i, x_j)^2 + \frac{1}{M}\sum_{k=1}^{M} d(x_i, x_k)^2 + d(x_k, x_j)^2 - \frac{1}{M}\sum_{l=1}^{M} d(x_k, x_l)^2 \Big) \tag{A.1}$$

Then it holds for all $i, j \in \{1, \ldots, M\}$: $d(x_i, x_j)^2 = s_d(x_i, x_i) - 2 \cdot s_d(x_i, x_j) + s_d(x_j, x_j)$.

Finally, it holds: d is Euclidean if and only if the matrix $\boldsymbol{S} \in \mathbb{R}^{M \times M}$ with entries $\boldsymbol{S}_{i,j} = s_d(x_i, x_j)$ is positive semi-definite.

Proof

The first claim is straightforward. Let $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ be a kernel with spatial map $\phi : \mathcal{X} \to \mathbb{R}^m$. Then, for all x, y it holds:

$$\begin{aligned}
d(x,y)^2 &= k(x,x) - 2 \cdot k(x,y) + k(y,y) \\
&= \phi(x)^\top \cdot \phi(x) - 2 \cdot \phi(x)^\top \cdot \phi(y) + \phi(y)^\top \cdot \phi(y) \\
&= \big(\phi(x) - \phi(y)\big)^\top \cdot \big(\phi(x) - \phi(y)\big) = \|\phi(x) - \phi(y)\|^2
\end{aligned}$$

Because this is a square number, it also holds $d(x,y) = \|\phi(x) - \phi(y)\|$, which shows that d is Euclidean with spatial mapping ϕ.

Conversely, if d is Euclidean with spatial mapping $\phi : \mathcal{X} \to \mathbb{R}^m$, then the function $k(x,y) = \phi(x)^\top \cdot \phi(y)$ is, per definition, a kernel and we obtain:

$$\begin{aligned}
d(x,y)^2 &= \|\phi(x) - \phi(y)\|^2 = \big(\phi(x) - \phi(y)\big)^\top \cdot \big(\phi(x) - \phi(y)\big) \\
&= \phi(x)^\top \cdot \phi(x) - 2 \cdot \phi(x)^\top \cdot \phi(y) + \phi(y)^\top \cdot \phi(y) \\
&= k(x,x) - 2 \cdot k(x,y) + k(y,y)
\end{aligned}$$

Now, consider the second claim, which we prove following Pekalska and Duin (2005, pp. 118-119). If s is a kernel with spatial mapping ϕ, then let $\boldsymbol{X} = \big(\phi(x_1), \ldots, \phi(x_M)\big)$. In that case, $\boldsymbol{S} = \boldsymbol{X}^\top \cdot \boldsymbol{X}$. Because $\boldsymbol{S}$ is an inner product of a matrix with itself, it follows that $\boldsymbol{S}$ must be symmetric and positive semi-definite.

Conversely, if $\boldsymbol{S}$ is symmetric, then the eigenvalue decompostion of $\boldsymbol{S}$ yields $\boldsymbol{V}^\top \cdot \boldsymbol{\Lambda} \cdot \boldsymbol{V} = \boldsymbol{S}$ for an $M \times M$ matrix of orthogonal eigenvectors $\boldsymbol{V}$ and a $M \times M$ diagonal matrix of corresponding eigenvalues $\boldsymbol{\Lambda}$.

If $\boldsymbol{S}$ is positive semi-definite, all entries of $\boldsymbol{\Lambda}$ are non-negative. Now, let $\sqrt{\boldsymbol{\Lambda}}$ denote the element-wise square-rooted version of $\boldsymbol{\Lambda}$, that is, $\sqrt{\boldsymbol{\Lambda}}_{i,j} = \sqrt{\boldsymbol{\Lambda}_{i,j}}$. Then, we set

$$\boldsymbol{X} := \big(\phi(x_1), \ldots, \phi(x_M)\big) := \sqrt{\boldsymbol{\Lambda}} \cdot \boldsymbol{V}$$

Accordingly, it holds: $\boldsymbol{X}^\top \cdot \boldsymbol{X} = \boldsymbol{V}^\top \cdot \sqrt{\boldsymbol{\Lambda}} \cdot \sqrt{\boldsymbol{\Lambda}} \cdot \boldsymbol{V} = \boldsymbol{S}$, meaning that for all i, j we obtain: $\boldsymbol{S}_{i,j} = \phi(x_i)^\top \cdot \phi(x_j)$ which proves that s is a kernel with the spatial mapping ϕ.

The third claim is due to *double-centering* as described by Torgerson (1952). First, we define the following auxiliary variables.

$$d_{i,j} := d(x_i, x_j)^2, \quad \bar{d} := \frac{1}{M^2} \sum_{i=1}^M \sum_{j=1}^M d_{i,j}, \quad \text{and} \quad \bar{d_i} := \frac{1}{M} \sum_{j=1}^M d_{i,j}$$

Now, recall that d is symmetric and self-equal. Therefore, we obtain:

$$\begin{aligned} s_d(x_i, x_i) - 2 \cdot s_d(x_i, x_j) + s_d(x_j, x_j) = &\frac{1}{2}\Big(- d_{i,i} + \bar{d_i} + \bar{d_i} - \bar{d}\Big) \\ &- 2 \cdot \frac{1}{2}\Big(- d_{i,j} + \bar{d_i} + \bar{d_j} + \bar{d}\Big) + \frac{1}{2}\Big(- d_{j,j} + \bar{d_j} + \bar{d_j} - \bar{d}\Big) \\ =&(\bar{d_i} - \frac{1}{2}\bar{d}) + (d_{i,j} - \bar{d_i} - \bar{d_j} + \bar{d}) + (\bar{d_j} - \frac{1}{2}\bar{d}) = d_{i,j} \end{aligned}$$

Finally, consider the fourth claim. If $\boldsymbol{S}$ with entries $\boldsymbol{S}_{i,j} = s_d(x_i, x_i)$ is symmetric and positive semi-definite, then s_d is a kernel according to the second claim. Further, if s_d is a kernel, then d is Euclidean according to the first claim.

Conversely, if d is Euclidean with spatial mapping $\phi : \mathcal{X} \to \mathbb{R}^m$, then it holds:

$$\begin{aligned} \boldsymbol{S}_{i,j} =&\frac{1}{2}\Big(- d_{i,j} + \bar{d_i} + \bar{d_j} - \bar{d}\Big) \\ =& - \frac{1}{2}(\phi(x_i) - \phi(x_j))^\top \cdot (\phi(x_i) - \phi(x_j)) \\ &+ \frac{1}{2M} \sum_{i'=1}^M \big(\phi(x_{i'}) - \phi(x_j)\big)^\top \cdot (\phi(x_{i'}) - \phi(x_j)) \\ &+ \frac{1}{2M} \sum_{j'=1}^M \big(\phi(x_i) - \phi(x_{j'})\big)^\top \cdot (\phi(x_i) - \phi(x_{j'})) \\ &- \frac{1}{2M^2} \sum_{i'=1}^M \sum_{j'=1}^M \big(\phi(x_{i'}) - \phi(x_{j'})\big)^\top \cdot (\phi(x_{i'}) - \phi(x_{j'})) \end{aligned}$$

$$
\begin{aligned}
=&-\frac{1}{2}\phi(x_i)^\top\cdot\phi(x_i)+\phi(x_i)^\top\cdot\phi(x_j)-\frac{1}{2}\phi(x_j)^\top\cdot\phi(x_j)\\
&+\frac{1}{M}\sum_{i'=1}^{M}\frac{1}{2}\phi(x_{i'})^\top\cdot\phi(x_{i'})-\phi(x_{i'})^\top\cdot\phi(x_j)+\frac{1}{2}\phi(x_j)^\top\cdot\phi(x_j)\\
&+\frac{1}{M}\sum_{j'=1}^{M}\frac{1}{2}\phi(x_i)^\top\cdot\phi(x_i)-\phi(x_i)^\top\cdot\phi(x_{j'})+\frac{1}{2}\phi(x_{j'})^\top\cdot\phi(x_{j'})\\
&+\frac{1}{M^2}\sum_{i'=1}^{M}\sum_{j'=1}^{M}-\frac{1}{2}\phi(x_{i'})^\top\cdot\phi(x_{i'})+\phi(x_{i'})^\top\cdot\phi(x_{j'})-\frac{1}{2}\phi(x_{j'})^\top\cdot\phi(x_{j'})\\
=&\phi(x_i)^\top\cdot\phi(x_j)-\frac{1}{M}\sum_{i'=1}^{M}\phi(x_{i'})^\top\cdot\phi(x_j)-\frac{1}{M}\sum_{j'=1}^{M}\phi(x_i)^\top\cdot\phi(x_{j'})\\
&+\frac{1}{M^2}\sum_{i'=1}^{M}\sum_{j'=1}^{M}\phi(x_{i'})^\top\cdot\phi(x_{j'})\\
=&\phi(x_i)^\top\cdot\phi(x_j)-\Big(\frac{1}{M}\sum_{i'=1}^{M}\phi(x_{i'})\Big)^\top\cdot\phi(x_j)-\phi(x_i)^\top\cdot\Big(\frac{1}{M}\sum_{j'=1}^{M}\phi(x_{j'})\Big)\\
&+\Big(\frac{1}{M}\sum_{i'=1}^{M}\phi(x_{i'})\Big)^\top\cdot\Big(\frac{1}{M}\sum_{j'=1}^{M}\phi(x_{j'})\Big)\\
=&\Big(\phi(x_i)-\frac{1}{M}\sum_{i'=1}^{M}\phi(x_{i'})\Big)^\top\cdot\Big(\phi(x_j)-\frac{1}{M}\sum_{j'=1}^{M}\phi(x_{j'})\Big)
\end{aligned}
$$

In other words, we can re-write $\boldsymbol{S}$ as $\boldsymbol{S}=\boldsymbol{X}^\top\cdot\boldsymbol{X}$, where

$$\boldsymbol{X}:=\Big(\phi(x_1)-\frac{1}{M}\sum_{i=1}^{M}\phi(x_i),\ldots,\phi(x_M)-\frac{1}{M}\sum_{i=1}^{M}\phi(x_i)\Big)$$

Because $\boldsymbol{S}$ is an inner product of a matrix with itself, $\boldsymbol{S}$ is positive semi-definite.

A.2 PROOF OF THEOREM 2.2

Recall the theorem we intend to prove.

Let $\mathcal{X}=\{x_1,\ldots,x_M\}$ be a finite set and let $d:\mathcal{X}\times\mathcal{X}\to\mathbb{R}$. It holds: d is pseudo-Euclidean if and only if d is symmetric and self-equal.

Proof

According to Theorem 2.1, we know that s_d fulfills the property: $d(x_i,x_j)^2=s_d(x_i,x_i)-2\cdot s_d(x_i,x_j)+s_d(x_j,x_j)$ for all $i,j\in\{1,\ldots,M\}$.

Further, because d is symmetric, we also know that the matrix $\boldsymbol{S}$ with entries $\boldsymbol{S}_{i,j}=s_d(x_i,x_j)$ is symmetric. Accordingly, the eigenvalue decompostion of $\boldsymbol{S}$ yields $\boldsymbol{V}^\top\cdot\boldsymbol{\Lambda}\cdot\boldsymbol{V}=\boldsymbol{S}$ for an $M\times M$ matrix of orthogonal eigenvectors $\boldsymbol{V}$ and a $M\times M$ diagonal matrix of corresponding eigenvalues $\boldsymbol{\Lambda}$. Without loss of generality, assume that the eigenvalues are sorted descendingly in Λ with $\Lambda_{m,m}$ being the smallest eigenvalue that is strictly positive, and with $\Lambda_{M-n+1,M-n+1}$ being the largest eigenvalue that is strictly negative.

Accordingly, all entries $\Lambda_{m+1,m+1}, \ldots \Lambda_{M-n,M-n}$ are zero. Further, let $\boldsymbol{V}^+ \in \mathbb{R}^{m \times M}$ be the matrix consisting of the first m rows of $\boldsymbol{V}$, let $\boldsymbol{V}^- \in \mathbb{R}^{n \times M}$ be the matrix consisting of the last n rows of $\boldsymbol{V}$, let $\boldsymbol{\Lambda}^+ \in \mathbb{R}^{m \times m}$ be the diagonal matrix with the diagonal entries $\sqrt{\Lambda_{1,1}}, \ldots, \sqrt{\Lambda_{m,m}}$, and let $\boldsymbol{\Lambda}^- \in \mathbb{R}^{n \times n}$ be the diagonal matrix with the diagonal entries $\sqrt{-\Lambda_{M-n+1,M-n+1}}, \ldots \sqrt{-\Lambda_{M,M}}$. Finally, let

$$\boldsymbol{X}^+ = \big(\phi^+(x_1), \ldots, \phi^+(x_M)\big) := \boldsymbol{\Lambda}^+ \cdot \boldsymbol{V}^+$$
$$\boldsymbol{X}^- = \big(\phi^-(x_1), \ldots, \phi^-(x_M)\big) := \boldsymbol{\Lambda}^- \cdot \boldsymbol{V}^-$$

Per construction, it holds:

$$\begin{aligned}
\boldsymbol{X}^{+\top} \cdot \boldsymbol{X}^+ - \boldsymbol{X}^{-\top} \cdot \boldsymbol{X}^- &= \boldsymbol{V}^{+\top} \cdot \boldsymbol{\Lambda}^+ \cdot \boldsymbol{\Lambda}^+ \cdot \boldsymbol{V}^+ + \boldsymbol{V}^{-\top} \cdot (-\boldsymbol{\Lambda}^- \cdot \boldsymbol{\Lambda}^-) \cdot \boldsymbol{V}^- \\
&= \begin{pmatrix} \boldsymbol{V}^+ \\ \boldsymbol{0}^{M-m-n \times M} \\ \boldsymbol{V}^- \end{pmatrix}^\top \cdot \begin{pmatrix} (\boldsymbol{\Lambda}^+)^2 & \boldsymbol{0}^{m \times M-m} \\ & \boldsymbol{0}^{M-m-n \times M} \\ \boldsymbol{0}^{n \times M-n} & -(\boldsymbol{\Lambda}^-)^2 \end{pmatrix} \cdot \begin{pmatrix} \boldsymbol{V}^+ \\ \boldsymbol{0}^{M-m-n \times M} \\ \boldsymbol{V}^- \end{pmatrix} \\
&= \boldsymbol{V}^\top \cdot \boldsymbol{\Lambda} \cdot \boldsymbol{V} = \boldsymbol{S}
\end{aligned}$$

where $\boldsymbol{0}^{rxs}$ is a $r \times s$ matrix of zeros. Accordingly, we obtain for any $i, j \in \{1, \ldots, M\}$: $s_d(x_i, x_j) = \phi^+(x_i)^\top \cdot \phi^+(x_j) - \phi^-(x_i)^\top \cdot \phi^-(x_j)$.

For the squared distance we thus obtain:

$$\begin{aligned}
& d(x_i, x_j)^2 = s_d(x_i, x_i) - 2 \cdot s_d(x_i, x_j) + s_d(x_j, x_j) \\
=& \phi^+(x_i)^\top \cdot \phi^+(x_i) - \phi^-(x_i)^\top \cdot \phi^-(x_i) - 2 \cdot \phi^+(x_i)^\top \cdot \phi^+(x_j) \\
& + 2 \cdot \phi^-(x_i)^\top \cdot \phi^-(x_j) + \phi^+(x_j)^\top \cdot \phi^+(x_j) - \phi^-(x_j)^\top \cdot \phi^-(x_j) \\
=& \big(\phi^+(x_i) - \phi^+(x_j)\big)^\top \cdot \big(\phi^+(x_i) - \phi^+(x_j)\big) - \big(\phi^-(x_i) - \phi^-(x_j)\big)^\top \cdot \big(\phi^-(x_i) - \phi^-(x_j)\big)
\end{aligned}$$

Therefore, d is pseudo-Euclidean with the positive spatial mapping ϕ^+ and the negative spatial mapping ϕ^-.

Conversely, if d is pseudo-Euclidean, then the right-hand-side of Equation 2.7 is obviously self-equal and symmetric.

A.3 PROOF OF THEOREM 2.3

Recall the theorem we intend to prove.

Let $\mathcal{X}$ be some arbitrary set and let $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ be a pseudo-Euclidean distance on $\mathcal{X}$ with the spatial mappings $\phi^+ : \mathcal{X} \to \mathbb{R}^m$ and $\phi^- : \mathcal{X} \to \mathbb{R}^n$. Further, let $\{x_1, \ldots, x_M\} \subseteq \mathcal{X}$ be a finite subset of $\mathcal{X}$, and let $\vec\alpha, \vec\beta \in \mathbb{R}^M$ such that $\sum_{i=1}^M \alpha_i = \sum_{i=1}^M \beta_i = 1$. Finally, let $\boldsymbol{X}^+ = \big(\phi^+(x_1), \ldots, \phi^+(x_M)\big) \in \mathbb{R}^{M \times m}$ and $\boldsymbol{X}^- = \big(\phi^-(x_1), \ldots, \phi^-(x_M)\big) \in \mathbb{R}^{M \times n}$ be the matrices of positive and negative spatial representations for all x_i, and let $\boldsymbol{D}^2$ be the $M \times M$ matrix with the entries $\boldsymbol{D}^2_{i,j} = d(x_i, x_j)^2$. Then, it holds:

$$\|\boldsymbol{X}^+ \cdot \vec\alpha - \boldsymbol{X}^+ \cdot \vec\beta\|^2 - \|\boldsymbol{X}^- \cdot \vec\alpha - \boldsymbol{X}^- \cdot \vec\beta\|^2 = \vec\alpha^\top \cdot \boldsymbol{D}^2 \cdot \vec\beta - \frac{1}{2}\vec\alpha^\top \cdot \boldsymbol{D}^2 \cdot \vec\alpha - \frac{1}{2}\vec\beta^\top \cdot \boldsymbol{D}^2 \cdot \vec\beta \quad \text{(A.2)}$$

Further, for any $x \in \mathcal{X}$ it holds:

$$\|\phi^+(x) - \boldsymbol{X}^+ \cdot \vec\alpha\|^2 - \|\phi^-(x) - \boldsymbol{X}^- \cdot \vec\alpha\|^2 = \sum_{i=1}^M \alpha_i \cdot d(x, x_i)^2 - \frac{1}{2}\vec\alpha^\top \cdot \boldsymbol{D}^2 \cdot \vec\alpha \quad \text{(A.3)}$$

If d is Euclidean with spatial mapping $\phi : \mathcal{X} \to \mathbb{R}^m$, then let $\boldsymbol{X} := \big(\phi(x_1), \ldots, \phi(x_M)\big) \in \mathbb{R}^{m \times M}$. It holds:

$$\|\boldsymbol{X} \cdot \vec{\alpha} - \boldsymbol{X} \cdot \vec{\beta}\|^2 = \vec{\alpha}^\top \cdot \boldsymbol{D}^2 \cdot \vec{\beta} - \frac{1}{2}\vec{\alpha}^\top \cdot \boldsymbol{D}^2 \cdot \vec{\alpha} - \frac{1}{2}\vec{\beta}^\top \cdot \boldsymbol{D}^2 \cdot \vec{\beta} \tag{A.4}$$

Further, for any $x \in \mathcal{X}$ it holds:

$$\|\phi(x) - \boldsymbol{X} \cdot \vec{\alpha}\|^2 = \sum_{i=1}^{M} \alpha_i \cdot d(x, x_i)^2 - \frac{1}{2}\vec{\alpha}^\top \cdot \boldsymbol{D}^2 \cdot \vec{\alpha} \tag{A.5}$$

Proof

This proof is adapted from Hammer and Hasenfuss (2010). We begin with Equation A.2. As a notational shorthand, let $\boldsymbol{S}_{i,j} := \phi^+(x_i)^\top \cdot \phi^+(x_j) - \phi^-(x_i)^\top \cdot \phi^-(x_j)$.

Now, consider the term $\vec{\alpha}^\top \cdot \boldsymbol{D}^2 \cdot \vec{\beta}$. We can re-write:

$$\begin{aligned}
\vec{\alpha}^\top \cdot \boldsymbol{D}^2 \cdot \vec{\beta} &= \sum_{i=1}^{M}\sum_{j=1}^{M} \alpha_i \cdot \beta_j \cdot d(x_i, x_j)^2 \\
&= \sum_{i=1}^{M}\sum_{j=1}^{M} \alpha_i \cdot \beta_j \cdot (\boldsymbol{S}_{i,i} - 2 \cdot \boldsymbol{S}_{i,j} + \boldsymbol{S}_{j,j}) \\
&= \sum_{i=1}^{M} \alpha_i \cdot \boldsymbol{S}_{i,i} \cdot \Big(\sum_{j=1}^{M} \beta_j\Big) - 2 \cdot \sum_{i=1}^{M}\sum_{j=1}^{M} \alpha_i \cdot \beta_j \cdot \boldsymbol{S}_{i,j} + \sum_{j=1}^{M} \beta_j \cdot \boldsymbol{S}_{j,j} \cdot \Big(\sum_{i=1}^{M} \alpha_i\Big) \\
&= \sum_{i=1}^{M} \alpha_i \cdot \boldsymbol{S}_{i,i} - 2 \cdot \sum_{i=1}^{M}\sum_{j=1}^{M} \alpha_i \cdot \beta_j \cdot \boldsymbol{S}_{i,j} + \sum_{j=1}^{M} \beta_j \cdot \boldsymbol{S}_{j,j}
\end{aligned}$$

Accordingly, we obtain the following results for $\vec{\alpha}^\top \cdot \boldsymbol{D}^2 \cdot \vec{\alpha}$ and $\vec{\beta}^\top \cdot \boldsymbol{D}^2 \cdot \vec{\beta}$

$$\begin{aligned}
\vec{\alpha}^\top \cdot \boldsymbol{D}^2 \cdot \vec{\alpha} &= 2 \cdot \sum_{i=1}^{M} \alpha_i \cdot \boldsymbol{S}_{i,i} - 2 \cdot \sum_{i=1}^{M}\sum_{j=1}^{M} \alpha_i \cdot \alpha_j \cdot \boldsymbol{S}_{i,j} \\
\vec{\beta}^\top \cdot \boldsymbol{D}^2 \cdot \vec{\beta} &= 2 \cdot \sum_{i=1}^{M} \beta_i \cdot \boldsymbol{S}_{i,i} - 2 \cdot \sum_{i=1}^{M}\sum_{j=1}^{M} \beta_i \cdot \beta_j \cdot \boldsymbol{S}_{i,j}
\end{aligned}$$

Hence, the right-hand-side of Equation A.2 adds up to:

$$\begin{aligned}
&\vec{\alpha}^\top \cdot \boldsymbol{D}^2 \cdot \vec{\beta} - \frac{1}{2}\vec{\alpha}^\top \cdot \boldsymbol{D}^2 \cdot \vec{\alpha} - \frac{1}{2}\vec{\beta}^\top \cdot \boldsymbol{D}^2 \cdot \vec{\beta} \\
&= \sum_{i=1}^{M} \alpha_i \cdot \boldsymbol{S}_{i,i} - 2 \cdot \sum_{i=1}^{M}\sum_{j=1}^{M} \alpha_i \cdot \beta_j \cdot \boldsymbol{S}_{i,j} + \sum_{j=1}^{M} \beta_j \cdot \boldsymbol{S}_{j,j} \\
&\quad - \sum_{i=1}^{M} \alpha_i \cdot \boldsymbol{S}_{i,i} + \sum_{i=1}^{M}\sum_{j=1}^{M} \alpha_i \cdot \alpha_j \cdot \boldsymbol{S}_{i,j} \\
&\quad - \sum_{i=1}^{M} \beta_i \cdot \boldsymbol{S}_{i,i} + \sum_{i=1}^{M}\sum_{j=1}^{M} \beta_i \cdot \beta_j \cdot \boldsymbol{S}_{i,j} \\
&= \sum_{i=1}^{M}\sum_{j=1}^{M} \alpha_i \cdot \alpha_j \cdot \boldsymbol{S}_{i,j} - 2 \cdot \alpha_i \cdot \beta_j \cdot \boldsymbol{S}_{i,j} + \beta_i \cdot \beta_j \cdot \boldsymbol{S}_{i,j}
\end{aligned}$$

$$
\begin{aligned}
=&\vec{\alpha}^\top \cdot \boldsymbol{X}^{+\top} \cdot \boldsymbol{X}^{+} \cdot \vec{\alpha} - \vec{\alpha}^\top \cdot \boldsymbol{X}^{-\top} \cdot \boldsymbol{X}^{-} \cdot \vec{\alpha} - 2 \cdot \vec{\alpha}^\top \cdot \boldsymbol{X}^{+\top} \cdot \boldsymbol{X}^{+} \cdot \vec{\beta} \\
&+ 2 \cdot \vec{\alpha}^\top \cdot \boldsymbol{X}^{-\top} \cdot \boldsymbol{X}^{-} \cdot \vec{\beta} + \vec{\beta}^\top \cdot \boldsymbol{X}^{+\top} \cdot \boldsymbol{X}^{+} \cdot \vec{\beta} - \vec{\beta}^\top \cdot \boldsymbol{X}^{-\top} \cdot \boldsymbol{X}^{-} \cdot \vec{\beta} \\
=&(\boldsymbol{X}^{+} \cdot \vec{\alpha} - \boldsymbol{X}^{+} \cdot \vec{\beta})^\top \cdot (\boldsymbol{X}^{+} \cdot \vec{\alpha} - \boldsymbol{X}^{+} \cdot \vec{\beta}) \\
&- (\boldsymbol{X}^{-} \cdot \vec{\alpha} - \boldsymbol{X}^{-} \cdot \vec{\beta})^\top \cdot (\boldsymbol{X}^{-} \cdot \vec{\alpha} - \boldsymbol{X}^{-} \cdot \vec{\beta}) \\
=&\|\boldsymbol{X}^{+} \cdot \vec{\alpha} - \boldsymbol{X}^{+} \cdot \vec{\beta}\|^2 - \|\boldsymbol{X}^{-} \cdot \vec{\alpha} - \boldsymbol{X}^{-} \cdot \vec{\beta}\|^2
\end{aligned}
$$

which concludes the proof of Equation A.2.

Regarding Equation A.3, let $\mathcal{X}' = \{x_1, \ldots, x_M, x_{M+1}\}$ with $x_{M+1} = x$, let $\vec{\alpha}' = (\alpha_1, \ldots, \alpha_M, 0)^\top \in \mathbb{R}^{M+1}$, let $\vec{\beta}$ be the $M+1$th unit vector, let $\boldsymbol{X}^{+\prime} = \big(\phi^+(x_1), \ldots, \phi^+(x_{M+1})\big)$, let $\boldsymbol{X}^{-\prime} = \big(\phi^-(x_1), \ldots, \phi^-(x_{M+1})\big)$, and let $\boldsymbol{D}'$ be the $M+1 \times M+1$ matrix with $\boldsymbol{D}'_{i,j} = d(x_i, x_j)$. Then, according to the first claim, it holds:

$$
\begin{aligned}
&\|\phi^+(x) - \boldsymbol{X}^{+} \cdot \vec{\alpha}\|^2 - \|\phi^-(x) - \boldsymbol{X}^{-} \cdot \vec{\alpha}\|^2 \\
=&\|\boldsymbol{X}^{+\prime} \cdot \vec{\beta} - \boldsymbol{X}^{+\prime} \cdot \vec{\alpha}'\|^2 - \|\boldsymbol{X}^{-\prime} \cdot \vec{\beta} - \boldsymbol{X}^{-\prime} \cdot \vec{\alpha}'\|^2 \\
=&\vec{\alpha}'^\top \cdot \boldsymbol{D}' \cdot \vec{\beta} - \frac{1}{2}\vec{\alpha}'^\top \cdot \boldsymbol{D}' \cdot \vec{\alpha}' - \frac{1}{2}\vec{\beta}^\top \cdot \boldsymbol{D}' \cdot \vec{\beta} \\
=&\sum_{i=1}^{M} \alpha_i \cdot 1 \cdot \boldsymbol{D}'_{i,M+1} - \frac{1}{2}\vec{\alpha}^\top \cdot \boldsymbol{D}^2 \cdot \vec{\alpha} - \frac{1}{2} 1 \cdot d(x,x)^2 \cdot 1 \\
=&\sum_{i=1}^{M} \alpha_i \cdot d(x, x_i)^2 - \frac{1}{2}\vec{\alpha}^\top \cdot \boldsymbol{D}^2 \cdot \vec{\alpha}
\end{aligned}
$$

which concludes the proof of Equation A.3.

With respect to the remaining two equations, we utilize the first result in Theorem 2.2. According to this result, we obtain for an Euclidean distance d:

$$
\begin{aligned}
\|\boldsymbol{X} \cdot \vec{\alpha} - \boldsymbol{X} \cdot \vec{\beta}\|^2 &= \|\boldsymbol{X}^{+} \cdot \vec{\alpha} - \boldsymbol{X}^{+} \cdot \vec{\beta}\|^2 - \|\boldsymbol{X}^{-} \cdot \vec{\alpha} - \boldsymbol{X}^{-} \cdot \vec{\beta}\|^2 \qquad \text{and} \\
\|\phi(x) - \boldsymbol{X} \cdot \vec{\alpha}\|^2 &= \|\phi^+(x) - \boldsymbol{X}^{+} \cdot \vec{\alpha}\|^2 - \|\phi^-(x) - \boldsymbol{X}^{-} \cdot \vec{\alpha}\|^2
\end{aligned}
$$

such that the equations follow directly from the first two claims.

A.4 PROOF OF THEOREM 2.4

Recall the theorem we intend to prove.

Let $\mathcal{A}$ be an alphabet with $- \notin \mathcal{A}$ and let c be a cost function over $\mathcal{A}$. Then it holds: For any trees $\tilde{x}, \tilde{y} \in \mathcal{T}(\mathcal{A})$, there exists at least one edit script $\bar{\delta} \in \Delta_{\mathcal{A}}$ such that $\bar{\delta}(\tilde{x}) = \tilde{y}$.

Further it holds: If c is a (pseudo-)metric over $\mathcal{A} \cup \{-\}$, then d_c is a (pseudo-)metric over $\mathcal{T}(\mathcal{A})$. More specifically, the following claims hold if c is non-negative (i.e. $\forall x, y \in \mathcal{A} \cup \{-\} : c(x,y) \geq 0$).

Non-Negativity: $\forall \tilde{x}, \tilde{y} \in \mathcal{T}(\mathcal{A}) : d_c(\tilde{x}, \tilde{y}) \geq 0$.

Self-Equality: $\forall \tilde{x} \in \mathcal{T}(\mathcal{A}) : d_c(\tilde{x}, \tilde{x}) = 0$.

Discernibility: $\forall x, y \in \mathcal{A} \cup \{-\} : x \neq y \Rightarrow c(x,y) > 0$ implies $\forall \tilde{x}, \tilde{y} \in \mathcal{T}(\mathcal{A}) : \tilde{x} \neq \tilde{y} \Rightarrow d_c(\tilde{x}, \tilde{y}) > 0$.

Table A.1: The number of children $R_X(i)$, the child indices $r_X(i)$, and the ancestors $\text{anc}_X(i)$ for the example tree $\tilde{x} = \texttt{a}(\texttt{b}(\texttt{c},\texttt{d}),\texttt{e})$ from Figure 2.6.

i	$\tilde{x}^i$	$R_X(i)$	$r_X(i)$	$\text{anc}_X(i)$
1	$\texttt{a}(\texttt{b}(\texttt{c},\texttt{d}),\texttt{e})$	2	1	$\emptyset$
2	$\texttt{b}(\texttt{c},\texttt{d})$	2	1	$\{1\}$
3	$\texttt{c}$	0	1	$\{1,2\}$
4	$\texttt{d}$	0	2	$\{1,2\}$
5	$\texttt{e}$	0	2	$\{1\}$

Symmetry: $\forall x, y \in \mathcal{A} \cup \{-\} : c(x,y) = c(y,x)$ implies $\forall \tilde{x}, \tilde{y} \in \mathcal{T}(\mathcal{A}) : d_c(\tilde{x}, \tilde{y}) = d_c(\tilde{y}, \tilde{x})$.

Triangular Inequality: $\forall \tilde{x}, \tilde{y}, \tilde{z} \in \mathcal{T}(\mathcal{A}) : d_c(\tilde{x}, \tilde{y}) + d_c(\tilde{y}, \tilde{z}) \geq d_c(\tilde{x}, \tilde{z})$

Proof

First, we need to introduce some auxiliary concepts for this proof.

Definition A.1 (Parents, children, and ancestors for forests)**.** Let $\mathcal{A}$ be an alphabet and let $X = \tilde{x}_1, \ldots, \tilde{x}_R$ be a forest over $\mathcal{A}$. Further, let $i \in \{1, \ldots, |X|\}$, let $\tilde{x}^i = x_i(\tilde{x}^i_1, \ldots, \tilde{x}^i_{R_i})$, let $r \in \{1, \ldots, R_i\}$, and let $i_r := i + 1 + \sum_{l=1}^{r-1} |\tilde{x}^i_l|$. We also include the special case $0_r := 1 + \sum_{l=1}^{r-1} |\tilde{x}_l|$.

Then, we denote the number of children R_i of subtree $\tilde{x}^i$ as $R_X(i)$, and we define for any i_r: $r_X(i_r) := r$.

Further, we define the *parent index* $\text{par}_X(i_r)$ of i_r in X as i, that is, $\text{par}_X(i_r) := i$.

For any $i \in \{1, \ldots, |X|\}$ we define the *ancestors* $\text{anc}_X(i)$ of i in X recursively as $\text{anc}_X(i) := \emptyset$ if $\text{par}_X(i) = 0$ and $\text{anc}_X(i) := \{\text{par}_X(i)\} \cup \text{anc}_X(\text{par}_X(i))$ otherwise.

Consider the example tree $\tilde{x} = \texttt{a}(\texttt{b}(\texttt{c},\texttt{d}),\texttt{e})$ from Figure 2.6. The number of children $R_X(i)$, the child indices $r_X(i)$, and the ancestors $\text{anc}_X(i)$ are shown in Table A.1.

To justify the definition of parent indices, we next show that the subtree identified with index i_r is actually the rth child of the subtree $\tilde{x}^i$.

Lemma A.1. *Let $\mathcal{A}$ be an alphabet and let $X = \tilde{x}_1, \ldots, \tilde{x}_R$ be a forest over $\mathcal{A}$.*

Then, the number of elements in the pre-order $\pi(X)$ is equal to $|X|$.

Further, for any $i \in \{1, \ldots, |X|\}$ and any $j \in \{1, \ldots, |\tilde{x}^i|\}$ it holds $\tilde{x}^{i,j} = \tilde{x}^{i+j-1}$, that is, the jth tree in the pre-order of the ith tree in the pre-order of X is the same as the $i+j-1$th tree in the pre-order of X.

Finally, let $i \in \{1, \ldots, |X|\}$, let $\tilde{x}^i = x_i(\tilde{x}^i_1, \ldots, \tilde{x}^i_{R_i})$, let $r \in \{1, \ldots, R_i\}$, and let $i_r := i + 1 + \sum_{l=1}^{r-1} |\tilde{x}^i_l|$. Then, it holds: $\tilde{x}^{i_r} = \tilde{x}^i_r$.

Proof. We prove the first claim via a simple induction over $|X|$. Let $\|\pi\|$ denote the number of elements in the list π.

If $|X| = 0$, then $X = \epsilon$ and $|X| = |\epsilon| = 0 = \|\epsilon\| = \|\pi(X)\|$. If $|X| > 0$, then $\|\pi(X)\| = \|\tilde{x}_1, \pi(\bar{\varrho}(\tilde{x}_1)), \pi(\tilde{x}_2, \ldots, \tilde{x}_R)\| = 1 + \|\pi(\bar{\varrho}(\tilde{x}_1))\| + \|\pi(\tilde{x}_2, \ldots, \tilde{x}_R)\|$. Per induction, this is equal to $1 + |\bar{\varrho}(\tilde{x}_1)| + |\tilde{x}_2, \ldots, \tilde{x}_R| = |X|$, which concludes the proof.

Regarding the second claim, we perform an induction over i. If $i = 1$, then $\tilde{x}^1 = \tilde{x}_1$. Accordingly, we obtain $\tilde{x}^{1,j} = \tilde{x}^j = \tilde{x}^{1+j-1}$ for any $j \in \{1, \ldots, |\tilde{x}^i|\}$.

Now, if $i > 1$, consider the pre-order of X, that is, $\pi(X) = \tilde{x}_1, \pi(\bar{\varrho}(\tilde{x}_1)), \pi(\tilde{x}, \ldots, \tilde{x}_R)$. We distinguish two cases.

First, if $i \in \{2, \ldots, |\tilde{x}_1|\}$, it must hold: $\tilde{x}^i = \pi(\bar{\varrho}(\tilde{x}_1))_{i-1}$. Further, because $i - 1 < i$, our induction hypothesis applies, which means that, for any $j \in \{1, \ldots, |\tilde{x}^i|\}$, we obtain $\tilde{x}^{i,j} = \pi(\bar{\varrho}(\tilde{x}_1))^j_{i-1} = \pi(\bar{\varrho}(\tilde{x}_1))_{i+j-2} = \pi(X)_{i+j-1} = \tilde{x}^{i+j-1}$ as claimed.

Now, if $i \in \{|\tilde{x}_1| + 1, \ldots, |X|\}$, it must hold: $\tilde{x}^i = \pi(\tilde{x}_2, \ldots, \tilde{x}_R)_{i-|\tilde{x}_1|}$, because $|\tilde{x}_1| = \|\tilde{x}^1, \pi(\bar{\varrho}(\tilde{x}_1))\|$. Because $i - |\tilde{x}_1| < i$, our induction hypothesis applies, which means that, for any $j \in \{1, \ldots, |\tilde{x}^i|\}$, we obtain: $\tilde{x}^{i,j} = \pi(\tilde{x}_2, \ldots, \tilde{x}_R)^j_{i-|\tilde{x}_1|} = \pi(\tilde{x}_2, \ldots, \tilde{x}_R)_{i-|\tilde{x}_1|+j-1} = \pi(X)_{i+j-1} = \tilde{x}^{i+j-1}$ as claimed. This concludes the proof.

Finally, regarding the third claim, consider the pre-order of $\tilde{x}^i$. It holds:

$$\begin{aligned}
\pi(\tilde{x}^i) &= \tilde{x}^i, \pi(\tilde{x}^i_1, \ldots, \tilde{x}^i_{R_i}), \pi(\epsilon) \\
&= \tilde{x}^i, \tilde{x}^i_1, \pi(\bar{\varrho}(\tilde{x}^i_1)), \pi(\tilde{x}^i_2, \ldots, \tilde{x}^i_{R_i}) \\
&= \ldots = \tilde{x}^i, \tilde{x}^i_1, \pi(\bar{\varrho}(\tilde{x}^i_1)), \tilde{x}^i_2, \pi(\bar{\varrho}(\tilde{x}^i_2)), \ldots, \tilde{x}^i_r, \pi(\bar{\varrho}(\tilde{x}^i_r)), \pi(\tilde{x}^i_{r+1}, \ldots, \tilde{x}^i_{R_i})
\end{aligned}$$

According to the first claim, the number of elements in $\pi(\bar{\varrho}(\tilde{x}^i_l))$ for any l is exactly $|\bar{\varrho}(\tilde{x}^i_l)|$. Accordingly, $\tilde{x}^i_r$ is exactly the $2 + \sum_{l=1}^{r-1} |\tilde{x}^i_l|$th element in the pre-order of $\tilde{x}^i$. In other words, we obtain $\tilde{x}^i_r = \tilde{x}^{i, 2 + \sum_{l=1}^{r-1} |\tilde{x}^i_l|}$, which, according to the second claim, is equal to $\tilde{x}^{i+1+\sum_{l=1}^{r-1} |\tilde{x}^i_l|} = \tilde{x}^{i_r}$. This concludes the proof. □

Now, consider the well-definedness claim. Let $\tilde{x}, \tilde{y} \in \mathcal{T}(\mathcal{A})$. Then, we define the two edit scripts $\bar{\delta}_{\text{del},\tilde{x}} := \text{del}_{|\tilde{x}|} \ldots \text{del}_1$ and $\bar{\delta}_{\text{ins},\tilde{y}} := \text{ins}_{p_{\tilde{y}}(1), y_1, r_{\tilde{y}}(1), r_{\tilde{y}}(1)} \ldots \text{ins}_{p_{\tilde{x}}(|\tilde{y}|), y_{|\tilde{y}|}, r_{\tilde{y}}(|\tilde{y}|), r_{\tilde{y}}(|\tilde{y}|)}$. Per construction we have $\bar{\delta}_{\text{del},\tilde{x}}(\tilde{x}) = \epsilon$ and $\bar{\delta}_{\text{ins},\tilde{y}}(\epsilon) = \tilde{y}$. Therefore, for the edit script $\bar{\delta} := \bar{\delta}_{\text{del},\tilde{x}} \bar{\delta}_{\text{ins},\tilde{y}}$ we obtain $\bar{\delta}(\tilde{x}) = \tilde{y}$ such that the set $\{\bar{\delta} \in \Delta^* | \bar{\delta}(\tilde{x}) = \tilde{y}\}$ is not empty.

Now, we consider each of the metric axioms in turn.

Non-Negativity: Let $\tilde{x}, \tilde{y} \in \mathcal{T}(\mathcal{A})$ and let $\bar{\delta}$ be some edit script over $\Delta_{\mathcal{A}}$ such that $\bar{\delta}(\tilde{x}) = \tilde{y}$. Because the cost $c(\bar{\delta}, \tilde{x})$ is a sum of non-negative contributions, the cost itself is non-negative. Because this holds for any edit script, we obtain $d_c(\tilde{x}, \tilde{y}) \geq 0$.

Self-Equality: Let $\tilde{x} \in \mathcal{T}(\mathcal{A})$. Then, the empty edit script $\bar{\delta} = \epsilon$ yields $\bar{\delta}(\tilde{x}) = \tilde{x}$ and has a cost of 0. Because we have already shown that d_c is non-negative we obtain $d_c(\tilde{x}, \tilde{y}) = 0$.

Discernibility: Let $\tilde{x}, \tilde{y} \in \mathcal{T}(\mathcal{A})$ with $\tilde{x} \neq \tilde{y}$ and let $\bar{\delta} = \delta_1 \ldots \delta_T$ be some edit script over $\Delta_{\mathcal{A}}$ such that $\bar{\delta}(\tilde{x}) = \tilde{y}$. Further, let $X_0 = \tilde{x}$ and $X_t = \delta_t(X_{t-1})$ for all $t \in \{1, \ldots, T\}$. Because $\tilde{x} \neq \tilde{y}$, there must exist at least one $t \in \{1, \ldots, T\}$ such that $\delta_t(X_{t-1}) \neq X_{t-1}$. If δ_t is a deletion or insertion, the cost of applying δ_t to X_{t-1} must be strictly positive, because for any $x \in \mathcal{A}$ it holds $c(x, -) > 0$ and $c(-, x) > 0$. If δ_t is a replacement, the cost of applying δ_t to X_{t-1} can only be 0 if the replaced node is replaced with itself. However, then $\delta_t(X_{t-1}) = X_{t-1}$, which is a contradiction. Therefore, in any case we obtain $c(\delta_t, X_{t-1}) > 0$. Because c is non-negative, the cost of applying $\bar{\delta}$ to $\tilde{x}$ is a sum of non-negative contributions with at least one strictly positive contribution, which means that $c(\bar{\delta}, \tilde{x}) > 0$. Because this reasoning applies to all edit scripts, we obtain $d_c(\tilde{x}, \tilde{y}) > 0$.

Symmetry: We prove a more general auxiliary claim, from which the symmetry of d_c follows.

Let $\bar{\delta}$ be an edit script in $\Delta^*_{\mathcal{A}}$ such that $\bar{\delta}(\tilde{x}) = \tilde{y}$. Then, there exists an edit script $\bar{\delta}^{-1}$ in $\Delta^*_{\mathcal{A}}$ such that $\bar{\delta}^{-1}(\tilde{y}) = \tilde{x}$, and $c(\bar{\delta}^{-1}, \tilde{y}) = c(\bar{\delta}, \tilde{x})$.

We prove this claim via induction over the length of $\bar{\delta}$. The base case is the empty edit script $\bar{\delta} = \epsilon$, i.e. $\tilde{x} = \tilde{y}$. In this case, we define $\bar{\delta}^{-1} = \epsilon$, such that $\bar{\delta}^{-1}(\bar{\delta}(\tilde{x})) = \bar{\delta}^{-1}(\tilde{x}) = \tilde{x}$ and $c(\bar{\delta}^{-1}, \tilde{y}) = c(\epsilon, \tilde{y}) = 0 = c(\bar{\delta}, \tilde{x})$.

Now, let $\bar{\delta} = \delta_1 \ldots \delta_T$ be a non-empty edit script. Per induction, there exists an edit script $\bar{\delta}_2^{-1}$ such that $\bar{\delta}_2^{-1}(\delta_2 \ldots \delta_T(\delta_1(\tilde{x}))) = \delta_1(\tilde{x})$ and $c(\bar{\delta}_2^{-1}, \tilde{y}) = c(\delta_2, \ldots, \delta_T, \delta_1(\tilde{x}))$. Now, consider the first tree edit δ_1. If $\delta_1(\tilde{x}) = \tilde{x}$ we define $\bar{\delta}^{-1} := \bar{\delta}_2^{-1}$ and we thus obtain $\bar{\delta}^{-1}(\bar{\delta}(\tilde{x})) = \bar{\delta}_2^{-1}(\delta_2 \ldots \delta_T(\delta_1(\tilde{x}))) = \delta_1(\tilde{x}) = \tilde{x}$, as well as $c(\bar{\delta}^{-1}, \tilde{y}) = c(\bar{\delta}_2^{-1}, \tilde{y}) = c(\delta_2 \ldots \delta_T, \tilde{x}) = c(\bar{\delta}, \tilde{x})$.

If $\delta_1(\tilde{x}) \neq \tilde{x}$, consider the following cases.

$\delta_1 = \mathrm{del}_i$: In that case, we define $\delta_1^{-1} := \mathrm{ins}_{\mathrm{par}_{\tilde{x}}(i), x_i, r_{\tilde{x}}(i), r_{\tilde{x}}(i)+R_{\tilde{x}}(i)}$. Per construction, we obtain $\delta_1^{-1}(\delta_1(\tilde{x})) = \tilde{x}$ as well as $c(\delta_1^{-1}, \delta_1(\tilde{x})) = c(-, x_i) = c(x_i, -) = c(\delta_1, \tilde{x})$.

$\delta_1 = \mathrm{rep}_{i,y}$: In that case, we define $\delta_1^{-1} := \mathrm{rep}_{i,x_i}$. Per construction we obtain $\delta_1^{-1}(\delta_1(\tilde{x})) = \tilde{x}$ as well as $c(\delta_1^{-1}, \delta_1(\tilde{x})) = c(y, x_i) = c(x_i, y) = c(\delta_1, \tilde{x})$.

$\delta_1 = \mathrm{ins}_{i,y,l,r}$: In that case, let $\tilde{x}^i = x_i(\tilde{x}^i_1, \ldots, \tilde{x}^i_{R_i})$ and let $i_l := i + 1 + \sum_{l'=1}^{l-1} |\tilde{x}^i_{l'}|$. We define $\delta_1^{-1} := \mathrm{del}_{i_l}$. Per construction we obtain $\delta_1^{-1}(\delta_1(\tilde{x})) = \tilde{x}$ as well as $c(\delta_1^{-1}, \delta_1(\tilde{x})) = c(y, -) = c(-, y) = c(\delta_1, \tilde{x})$.

We define the edit script $\bar{\delta}^{-1}$ as $\bar{\delta}^{-1} := \bar{\delta}_2^{-1}\delta_1^{-1}$ in all three cases. Therefore, we can conclude that $\bar{\delta}^{-1}(\bar{\delta}(\tilde{x})) = \delta_1^{-1}\Big(\bar{\delta}_2^{-1}(\delta_2 \ldots \delta_T(\delta_1(\tilde{x})))\Big) = \delta_1^{-1}(\delta_1(\tilde{x})) = \tilde{x}$ as well as $c(\bar{\delta}^{-1}, \tilde{y}) = c(\bar{\delta}_2^{-1}, \tilde{y}) + c(\delta_1^{-1}, \delta_1(\tilde{x})) = c(\delta_2 \ldots \delta_T, \tilde{x}) + c(\delta_1, \tilde{x}) = c(\bar{\delta}, \tilde{x})$.

As an example for this construction, consider Figure 2.4. In this example, the edit script $\bar{\delta} = \mathrm{rep}_{1,\mathtt{f}}\mathrm{del}_2\mathrm{del}_2\mathrm{rep}_{2,\mathtt{g}}\mathrm{del}_3$ transforms the tree $\tilde{x} = \mathtt{a}(\mathtt{b}(\mathtt{c}, \mathtt{d}), \mathtt{e})$ into the tree $\tilde{y} = \mathtt{f}(\mathtt{g})$ with the cost $c(\bar{\delta}, \tilde{x}) = c(\mathtt{a}, \mathtt{f}) + c(\mathtt{b}, -) + c(\mathtt{c}, -) + c(\mathtt{d}, \mathtt{g}) + c(\mathtt{e}, -)$. The corresponding inverse edit script via the construction above is $\bar{\delta}^{-1} = \mathrm{ins}_{1,\mathtt{e},2,2}\mathrm{rep}_{2,\mathtt{d}}\mathrm{ins}_{1,\mathtt{c},1,1}\mathrm{ins}_{1,\mathtt{b},1,3}\mathrm{rep}_{1,\mathtt{a}}$ with the cost $c(\bar{\delta}^{-1}, \tilde{y}) = c(-, \mathtt{e}) + c(\mathtt{g}, \mathtt{d}) + c(-, \mathtt{c}) + c(-, \mathtt{b}) + c(\mathtt{f}, \mathtt{a})$. Therefore, if c is symmetric, these edit scripts have the same cost.

Now, let $\bar{\delta}$ be an edit script over $\Delta_{\mathcal{A}}$ with $\bar{\delta}(\tilde{x}) = \tilde{y}$, such that $d_c(\tilde{x}, \tilde{y}) = c(\bar{\delta}, \tilde{x})$. Because d_c is well-defined, such an edit script exists. Per our induction above we know that there is a edit script $\bar{\delta}^{-1} \in \Delta^*_{\mathcal{A}}$ such that $\bar{\delta}(\bar{\delta}^{-1}, \tilde{y}) = \tilde{x}$ and $c(\bar{\delta}^{-1}, \tilde{y}) = c(\bar{\delta}, \tilde{x})$. Therefore, we obtain $d_c(\tilde{y}, \tilde{x}) \leq c(\bar{\delta}^{-1}, \tilde{y}) = d_c(\tilde{x}, \tilde{y})$. Using a symmetric reasoning, we can also conclude that $d_c(\tilde{y}, \tilde{x}) \geq d_c(\tilde{x}, \tilde{y})$, i.e. $d_c(\tilde{x}, \tilde{y}) = d_c(\tilde{y}, \tilde{x})$.

Triangular Inequality: Let $\tilde{x}, \tilde{y}, \tilde{z} \in \mathcal{T}(\mathcal{A})$ and let $\bar{\delta}, \bar{\delta}'$ be edit scripts over $\Delta_{\mathcal{A}}$ such that $\bar{\delta}(\tilde{x}) = \tilde{y}$, $\bar{\delta}'(\tilde{y}) = \tilde{z}$, $c(\bar{\delta}, \tilde{x}) = d_c(\tilde{x}, \tilde{y})$, and $c(\bar{\delta}', \tilde{y}) = d_c(\tilde{y}, \tilde{z})$. Then, $\bar{\delta}'' := \bar{\delta}\bar{\delta}'$ is an edit script over $\Delta_{\mathcal{A}}$ such that $\bar{\delta}''(\tilde{x}) = \tilde{z}$, and we obtain $d_c(\tilde{x}, \tilde{z}) \leq c(\bar{\delta}'', \tilde{x}) = c(\bar{\delta}, \tilde{x}) + c(\bar{\delta}', \tilde{y}) = d_c(\tilde{x}, \tilde{y}) + d_c(\tilde{y}, \tilde{z})$.

A.5 PROOF OF THEOREM 2.5

Recall the theorem we intend to prove.

Let $\mathcal{A}$ be an alphabet, let c be a cost function over $\mathcal{A}$, and let $\tilde{x}$ and $\tilde{y}$ be trees over $\mathcal{A}$. Then, Algorithm 2.1 computes the tree mapping edit distance $D_c(\tilde{x}, \tilde{y})$ between $\tilde{x}$ and $\tilde{y}$. Further, Algorithm 2.1 runs in $\mathcal{O}(|\tilde{x}| \cdot |\tilde{y}|)$ space complexity and $\mathcal{O}(|\tilde{x}|^2 \cdot |\tilde{y}|^2)$ time complexity.

Finally, it holds: If c is self-equal, non-negative, and fulfills the triangular inequality, then $D_c(\tilde{x}, \tilde{y}) = d_c(\tilde{x}, \tilde{y})$.

Proof

Note that our proof is conceptually equivalent to the original proof of Zhang and Shasha (1989). All differences are due to notational changes and the fact that we use the pre-order instead of the post-order for simplicity.

First, consider the complexity claims. Algorithm 2.1 maintains two matrices, each of size $\mathcal{O}(|\tilde{x}| \cdot |\tilde{y}|)$ and does otherwise only store the input such that the space complexity claim follows immediately. Regarding runtime, Algorithm 2.1 executes four nested loops, in which only constant operations are necessary (assuming that access to auxiliary tree properties is possible in constant time). Therefore, we obtain a worst-case runtime complexity of $\mathcal{O}(|\mathcal{K}(\tilde{x})| \cdot |\mathcal{K}(\tilde{y})| \cdot |\tilde{x}| \cdot |\tilde{y}|)$. Note that $|\mathcal{K}(\tilde{x})| \leq |\tilde{x}|$ and $|\mathcal{K}(\tilde{y})| \leq |\tilde{y}|$ such that we obtain our desired worst-case-bound of $\mathcal{O}(|\tilde{x}|^2 \cdot |\tilde{y}|^2)$ time complexity.

Proving the correctness of Algorithm 2.1 is substantially more complex and we will rely on multiple intermediate lemmata to do so. In a first step, we will generalize our tree concepts to forests and establish some lemmata regarding the relations between these auxiliary concepts. Then, we will go on to show that the tree edit distance and the tree mapping edit distance are equivalent. Finally, we will prove a decomposition lemma for the tree mapping edit distance and hence the correctness of Algorithm 2.1.

In a next step, we generalize the concepts of outermost right leaves and keyroots from trees (refer to Definition 2.15) to forests, because our remaining argument will apply to forests in general, not only to trees.

Definition A.2 (Outermost Right Leaves and Keyroots for Forests)**.** Let $\mathcal{A}$ be an alphabet and let X be a forest over $\mathcal{A}$. For any $i \in \{1, \ldots, |X|\}$ we define the outermost right leaf $rl_X(i)$ of i in X as $rl_X(i) := i + |\tilde{x}^i| - 1$; we define the *keyroot* $\mathrm{k}_X(i)$ of i in X as $\mathrm{k}_X(i) := \min\{j | rl_X(i) = rl_X(j)\}$; and we define the keyroots $\mathcal{K}(X)$ of X as the set $\mathcal{K}(X) := \{j | \exists i \in \{1, \ldots, |X|\} : j = \mathrm{k}_X(i)\}$.

Note that ancestral relationships have a deep connection to outermost right leaves. In particular, all i which have k as ancestor are in the range $(k, rl_X(k)]$ and vice versa.

Lemma A.2. *Let $\mathcal{A}$ be an alphabet, let X be a forest over $\mathcal{A}$ and let $k \in \{1, \ldots, |X|\}$. Then, it holds:*

$$\forall i \in \{1, \ldots, |X|\} : k \in \text{anc}_X(i) \Rightarrow k < i \leq rl_X(i) \leq rl_X(k) \tag{A.6}$$

$$\forall i \in \{1, \ldots, |X|\} : k < i \leq rl_X(k) \Rightarrow k \in \text{anc}_X(i) \tag{A.7}$$

$$\forall i, j \in \text{anc}_X(k) : i < j \Leftrightarrow i \in \text{anc}_X(j) \tag{A.8}$$

Proof. We first provide a proof for Equations A.6 and A.7, and then go on to prove Equation A.8. Our first proof works via induction over the size of the subtree $\tilde{x}^k$.

If $|\tilde{x}^k| = 1$, then k can not be an ancestor of any element and, likewise, there exist no i such that $k < i \leq rl_X(k) = k$ or $k < rl_X(i) \leq rl_X(k) = k$. Therefore, the base case holds for Equations A.6 and A.7.

Now, assume that $|\tilde{x}^k| > 1$. Let $\tilde{x}^k = x_k(\tilde{x}^k_1, \ldots, \tilde{x}^k_{R_k})$. Further, let for all $r \in \{1, \ldots, R_k + 1\} : k_r := k + \sum_{l=1}^{r-1} |\tilde{x}^k_l| + 1$. Recall that, according to Lemma A.1, for all $r \in \{1, \ldots, R_k\}$ it holds: $\tilde{x}^{k_r} = \tilde{x}^k_r$. Further note that $rl_X(k_r) = k_r + |\tilde{x}^k_r| - 1 = k + \sum_{l=1}^{r} |\tilde{x}^k_l| = k_{r+1} - 1$. Finally, it holds: $k_{R_k+1} = k + \sum_{l=1}^{R_k} |\tilde{x}^k_l| + 1 = k + |\tilde{x}^k| = rl_X(k) + 1$.

Regarding Equation A.6, we consider $k \in \text{anc}_X(i)$. Then per definition of ancestors, one of the following two cases applies.

$\text{par}_X(i) = k$: In that case, let r be the index such that $i = k_r$. Then, it holds:

$$k < k + \sum_{l=1}^{r-1} |\tilde{x}^k_1| + 1 = k_r = i \leq rl_X(i) = rl_X(k_r) = k_{r+1} - 1 \leq k_{R_k+1} - 1 = rl_X(k)$$

$\text{par}_X(i) \neq k$: In that case, there is some $j \in \text{anc}_X(i)$ such that $\text{par}_X(j) = k$, otherwise k would not be in $\text{anc}_X(i)$. Let r be the index such that $k_r = j$. Then, per induction, we know that

$$k < k + \sum_{l=1}^{r-1} |\tilde{x}^k_1| + 1 = k_r = j \overset{I.H.}{<} i \leq rl_X(i)$$
$$\overset{I.H.}{\leq} rl_X(j) = rl_X(k_r) = k_{r+1} - 1 \leq k_{R_k+1} - 1 = rl_X(k)$$

which concludes the proof.

Regarding Equation A.7, we consider $k < i \leq rl_X(k)$. Then, there exists exactly one r such that $k_r \leq i < k_{r+1}$. Now, if $k_r = i$, we obtain $\text{par}_X(i) = k$, which in turn implies $k \in \text{anc}_X(i)$. If $k_r < i$, then $k_r < i \leq k_{r-1} - 1 = rl_X(k_r)$. Therefore, per induction, it holds: $k_r \in \text{anc}_X(i)$. Due to the definition of ancestors, we also know that $k \in \text{anc}_X(k_r)$. Therefore, $k \in \text{anc}_X(i)$, which concludes the proof.

Now, consider Equation A.8. We perform an inductive proof over the size of the ancestor set $|\text{anc}_X(k)|$.

If $\text{anc}_X(k)$ is empty or contains only a single element, then the claim holds trivially. If $|\text{anc}_X(k)| > 1$, consider $i := \text{par}_X(k)$. Then, per definition of ancestors, we have $\text{anc}_X(k) = \{i\} \cup \text{anc}_X(i)$. Because $|\text{anc}_X(i)| < |\text{anc}_X(k)|$, our induction hypothesis applies and the claim holds for all pairwise comparisons within $\text{anc}_X(i)$. It remains to show that the claim holds for all pairwise comparisons (i, j) with $j \in \text{anc}_X(k)$. There are only two possible cases for j. Either $i = j$, in which case the claim holds trivially, or $j \in \text{anc}_X(i)$. In that case, Equation A.6 implies $j < i$, which means that the claim holds as well. This concludes the proof. $\square$

Next, we generalize the notion of tree mappings between trees (refer to Definition 2.14) to tree mappings between forests.

Definition A.3 (Mappings)**.** Let $\mathcal{A}$ be an alphabet and let X, Y be forests over $\mathcal{A}$. Then, we define a tree mapping M between X and Y as a subset $M \subseteq \{1, \ldots, |X|\} \times \{1, \ldots, |Y|\}$ such that the following conditions hold for all entries $(i, j), (i', j') \in M$.

$$i \geq i' \iff j \geq j' \qquad \text{(pre-order preservation)} \tag{A.9}$$

$$i \in \text{anc}_X(i') \iff j \in \text{anc}_Y(j') \qquad \text{(ancestral preservation)} \tag{A.10}$$

We define the *left-complement* of M as $I(M, X, Y) := \{i \in \{1, \ldots, |X|\} | \nexists j \in \{1, \ldots, |Y|\} : (i, j) \in M\}$ and we define the *right-complement* of M as $J(M, X, Y) := \{j \in \{1, \ldots, |Y|\} | \nexists i \in \{1, \ldots, |X|\} : (i, j) \in M\}$. Finally, we define the *cost* of M according to some cost function c over $\mathcal{A}$ as follows.

$$c(M, X, Y) := \sum_{(i,j) \in M} c(x_i, y_j) + \sum_{i \in I(M,X,Y)} c(x_i, -) + \sum_{j \in J(M,X,Y)} c(-, y_j)$$

In a next step, we show that we can always find an edit script which is exactly as expensive as the tree mapping in question. Conversely, we can always find a tree mapping which is at most as expensive as the edit script in question. This very fact permits us to search for cheapest tree mappings instead of cheapest edit scripts, as we show in the next Lemma. First, however, we define an alternative distance based on tree mappings, which we will then show to be equivalent.

Definition A.4 (Forest Edit Distance, Forest Mapping Distance)**.** Let $\mathcal{A}$ be an alphabet and let X, Y be forests over $\mathcal{A}$. Further, let c be a cost function over $\mathcal{A}$. Then, we define the forest edit distance $d_c(X, Y)$ between X and Y as

$$d_c(X, Y) := \min_{\bar{\delta} \in \Delta^*_{\mathcal{A}}} \{c(\bar{\delta}, X) | \bar{\delta}(X) = Y\} \tag{A.11}$$

Further, we define the forest tree mapping distance $D_c(X, Y)$ between X and Y as

$$D_c(X, Y) := \min_{M \subset \{1,\ldots,|X|\} \times \{1,\ldots,|Y|\}} \{c(M, X, Y) | M \text{ is a tree mapping between } X \text{ and } Y\} \tag{A.12}$$

In the next lemma, we demonstrate that under some conditions to the cost function, d_c and D_c are equivalent.

Lemma A.3. *Let $\mathcal{A}$ be an alphabet and let X, Y be forests over $\mathcal{A}$. Further, let c be a cost function over $\mathcal{A}$. Then, it holds:*

1. *For any tree mapping M between X and Y there exists an edit script $\bar{\delta}_M \in \Delta_{\mathcal{A}}$ such that $\bar{\delta}(X) = Y$ and $c(\bar{\delta}, X) = c(M, X, Y)$.*

2. *If c fulfills the triangular inequality and is self-equal, then for any edit script $\bar{\delta} \in \Delta_{\mathcal{A}}$ with $\bar{\delta}(X) = Y$ there exists a tree mapping $M_{\bar{\delta}}$ between X and Y such that $c(M_{\bar{\delta}}, X, Y) \leq c(\bar{\delta}, X)$.*
3. *If c fulfills the triangular inequality and is self-equal, then $d_c(X, Y) = D_c(X, Y)$.*

Proof. We will consider each claim in turn.

Regarding the first claim, we define two more auxiliary sets, namely $I^C(M, X, Y) := \{i \in \{1, \ldots, |X|\} | \exists j \in \{1, \ldots, |Y|\} : (i, j) \in M\}$ and $J^C(M, X, Y) := \{j \in \{1, \ldots, |Y|\} | \exists i \in \{1, \ldots, |X|\} : (i, j) \in M\}$. Then, we can construct $\bar{\delta}_M$ as the concatenation of three edit scripts $\bar{\delta}_M^{\text{rep}}$, $\bar{\delta}_M^{\text{del}}$, and $\bar{\delta}_M^{\text{ins}}$ as follows. We define $\bar{\delta}_M^{\text{rep}}$ as the list of rep_{i,y_j} for all $(i, j) \in M$ in lexical ascending order, first sorted according to i and then according to j. Per construction, this edit script replaces all x_i with the mapped label y_j according to the tree mapping M.

Next, we define $\bar{\delta}_M^{\text{del}}$ as the list of del_i for all $i \in I(M, X, Y)$ in *descending* order. Per construction, $\bar{\delta}_M^{\text{del}}(X)$ contains exactly those x_i such that $i \in I^C(M, X, Y)$.

Finally, we define $\bar{\delta}_M^{\text{ins}}$ as the list of $\text{ins}_{\text{par}_Y(j), y_j, r_Y(j), r_Y(j)+R_{M,X,Y}(j)}$ for all $j \in J(M, X, Y)$ in ascending order, where we define $R_{M,X,Y}(j)$ recursively as $R_{M,X,Y}(j) := |\text{adj}_Y(j) \cap J^C(M, X, Y)| + \sum_{j' \in \text{adj}_Y(j) \cap J(M,X,Y)} R_{M,X,Y}(j')$ and where $\text{adj}_Y(j) = \{j' | \text{par}_Y(j') = j\}$.

Per construction, $\bar{\delta}_M^{\text{ins}}$ inserts all labels of Y which are missing in $\bar{\delta}_M^{\text{rep}} \bar{\delta}_M^{\text{del}}(X)$. The definition of $r_Y(j)$ and $R_{M,X,Y}(j)$ ensures that label y_j is inserted at the correct position and uses all children which are mapped to labels in X and are descendants of $\tilde{y}^j$ in Y.

For $\bar{\delta}_M := \bar{\delta}_M^{\text{rep}} \bar{\delta}_M^{\text{del}} \bar{\delta}_M^{\text{ins}}$ we thus obtain $\bar{\delta}_M(X) = Y$ and

$$\begin{aligned} c(\bar{\delta}_M, X) &= c(\bar{\delta}_M^{\text{rep}}, X) + c(\bar{\delta}_M^{\text{del}}, \bar{\delta}_M^{\text{rep}}(X)) + c(\bar{\delta}_M^{\text{ins}}, \bar{\delta}_M^{\text{rep}} \bar{\delta}_M^{\text{del}}(X)) \\ &= \sum_{(i,j) \in M} c(x_i, y_j) + \sum_{i \in I(M,X,Y)} c(x_i, -) + \sum_{j \in J(M,X,Y)} c(-, y_j) = c(M, X, Y). \end{aligned}$$

Regarding the second claim, we perform an inductive proof. As base case, consider the empty edit script $\bar{\delta} = \epsilon$, which implies that $\bar{\delta}(X) = Y = X$. In that case, we define $M_{\bar{\delta}} = \{(i, i) | i \in \{1, \ldots, |X|\}\}$. Accordingly, we obtain $c(M_{\bar{\delta}}, X, Y) = c(M_{\bar{\delta}}, X, X) = \sum_{i=1}^{|X|} c(x_i, x_i)$. Because c is self equal, $c(x_i, x_i)$ is zero for all i, which in turn implies that $c(M_{\bar{\delta}}, X, Y) = 0 = c(\epsilon, X)$ as desired.

Now, consider a non-empty edit script $\bar{\delta} = \delta_1 \ldots \delta_{T+1}$ over $\Delta_{\mathcal{A}}$ such that $\bar{\delta}(X) = Y$ and let $\bar{\delta}' := \delta_1 \ldots \delta_T$ as well as $Y' := \bar{\delta}'(X)$. Due to induction, we know that there exists a tree mapping $M_{\bar{\delta}'}$ between X and Y' such that $c(M_{\bar{\delta}'}, X, Y') \leq c(\bar{\delta}', X)$. Now, consider the final edit δ_{T+1}. If $\delta_{T+1}(Y') = Y' = Y$, we define $M_{\bar{\delta}} := M_{\bar{\delta}'}$. Because $M_{\bar{\delta}'}$ is a valid tree mapping between X and Y' it is also a valid tree mapping between X and $Y = Y'$. Further, for the cost we obtain $c(\bar{\delta}, X) = c(\bar{\delta}', X) \overset{\text{Induction}}{\geq} c(M_{\bar{\delta}'}, X, Y') = c(M_{\bar{\delta}}, X, Y)$.

It remains to consider all cases in which $Y = \delta_{T+1}(Y') \neq Y'$. We distinguish the following cases.

$\delta_{T+1} = \text{rep}_{j,y_j}$ **for some** $j \in \{1, \ldots, |Y|\}$. Then, we define $M_{\bar{\delta}} := M_{\bar{\delta}'}$. $M_{\bar{\delta}}$ is a tree mapping between X and Y because the ancestral structure of Y' and Y is exactly the same and $M_{\bar{\delta}'}$ was per induction a valid tree mapping between X and Y'.

Further, if there exists an i such that $(i,j) \in M_{\bar{\delta}'}$ we obtain:

$$\begin{aligned} c(\bar{\delta}, X) = c(\bar{\delta}', X) + c(y'_j, y_j) &\overset{Induction}{\geq} c(M_{\bar{\delta}'}, X, Y') + c(y'_j, y_j) \\ &= c(M_{\bar{\delta}}, X, Y) - c(x_i, y_j) + c(x_i, y'_j) + c(y'_j, y_j) \\ &\overset{triang.}{\geq} c(M_{\bar{\delta}}, X, Y) - c(x_i, y_j) + c(x_i, y_j) = c(M_{\bar{\delta}}, X, Y) \end{aligned}$$

Conversely, if there is no i such that $(i,j) \in M_{\bar{\delta}'}$ we obtain:

$$\begin{aligned} c(\bar{\delta}, X) = c(\bar{\delta}', X) + c(y'_j, y_j) &\overset{Induction}{\geq} c(M_{\bar{\delta}'}, X, Y') + c(y'_j, y_j) \\ &= c(M_{\bar{\delta}}, X, Y) - c(-, y_j) + c(-, y'_j) + c(y'_j, y_j) \\ &\overset{triang.}{\geq} c(M_{\bar{\delta}}, X, Y) - c(-, y_j) + c(-, y_j) = c(M_{\bar{\delta}}, X, Y) \end{aligned}$$

$\delta_{T+1} = \text{del}_j$ **for some** $j \in \{1, \ldots, |Y'|\}$**.** Then, for all $j', \in \{1, \ldots, j-1\}$ it holds $\text{anc}_Y(j') = \text{anc}_{Y'}(j')$, and for all $j', \in \{j+1, \ldots, |Y'|\}$ it holds $\text{anc}_Y(j') = \{j''|j'' \in \text{anc}_{Y'}(j'), j'' < j\} \cup \{j''-1|j'' \in \text{anc}_{Y'}(j'), j'' \geq j\}$. Accordingly, we define $M_{\bar{\delta}} := \{(i,j') \in M_{\bar{\delta}'}|j' < j\} \cup \{(i,j'-1) \in M_{\bar{\delta}'}|j' > j\}$ such that $M_{\bar{\delta}}$ is a valid tree mapping between X and Y.

Further, if there exists an i such that $(i,j) \in M_{\bar{\delta}'}$ we obtain:

$$\begin{aligned} c(\bar{\delta}, X) = c(\bar{\delta}', X) + c(y'_j, -) &\overset{Induction}{\geq} c(M_{\bar{\delta}'}, X, Y') + c(y'_j, -) \\ &= c(M_{\bar{\delta}}, X, Y) - c(x_i, -) + c(x_i, y'_j) + c(y'_j, -) \\ &\overset{triang.}{\geq} c(M_{\bar{\delta}}, X, Y) - c(x_i, -) + c(x_i, -) = c(M_{\bar{\delta}}, X, Y) \end{aligned}$$

Conversely, if there is no i such that $(i,j) \in M_{\bar{\delta}'}$ we obtain:

$$\begin{aligned} c(\bar{\delta}, X) = c(\bar{\delta}', X) + c(y'_j, -) &\overset{Induction}{\geq} c(M_{\bar{\delta}'}, X, Y') + c(y'_j, -) \\ &= c(M_{\bar{\delta}}, X, Y) + c(-, y'_j) + c(y'_j, -) \\ &\overset{triang.}{\geq} c(M_{\bar{\delta}}, X, Y) + c(-, -) \overset{self-id.}{=} c(M_{\bar{\delta}}, X, Y) \end{aligned}$$

$\delta_{T+1} = \text{ins}_{\text{par}(j), y_j, l, r}$ **for some** $j \in \{1, \ldots, |Y|\}$**,** $l \leq r \in \{1, \ldots, |\bar{\varrho}(\tilde{y}^j)|\}$**.** Then, for all $j' < j$ it holds: $\text{anc}_Y(j') = \text{anc}_{Y'}(j')$. For all j' with $j \in \text{anc}_Y(j')$ it holds: $\text{anc}_Y(j') = \{j'' \in \text{anc}_{Y'}(j'-1)|j'' < j\} \cup \{j\} \cup \{j''+1|j'' \in \text{anc}_{Y'}(j'-1), j'' \geq j\}$. Finally, for all j' with $j' > j$ and $j \notin \text{anc}_Y(j')$ it holds: $\text{anc}_Y(j') = \{j'' \in \text{anc}_{Y'}(j'-1)|j'' < j\} \cup \{j''+1|j'' \in \text{anc}_{Y'}(j'-1), j'' \geq j\}$. In other words, the ancestors for all $j' < j$ are maintained, while the ancestors for $j' > j$ in Y are the ancestors of $j'-1$ in Y', except for j, which may be added as an ancestor. Accordingly, we define $M_{\bar{\delta}} := \{(i,j') \in M_{\bar{\delta}'}|j' < j\} \cup \{(i,j'+1)|(i,j') \in M_{\bar{\delta}'}, j' \geq j\}$ such that $M_{\bar{\delta}}$ is a valid tree mapping between X and Y.

Further, for the cost we obtain:

$$c(\bar{\delta}, X) = c(\bar{\delta}', X) + c(-, y_j) \overset{Induction}{\geq} c(M_{\bar{\delta}'}, X, Y') + c(-, y_j) = c(M_{\bar{\delta}}, X, Y)$$

Therefore, in all cases, we obtain $c(M_{\bar{\delta}}, X, Y) \leq c(\bar{\delta}, X)$ which concludes the proof by induction.

Finally, the third claim follows from the previous two. In particular, consider the following proof by contradition. If $D_c(X, Y) < d_c(X, Y)$, then there exists a tree mapping M between X and Y such that $c(M, X, Y) < d_c(X, Y)$. However, we have shown that we can construct an edit script $\bar{\delta}_M$ such that $\bar{\delta}_M(X) = Y$ and $c(\bar{\delta}_M, X) = c(M, X, Y)$. Therefore, $d_c(X, Y) \leq c(\bar{\delta}_M, X) = c(M, X, Y) < d_c(X, Y)$, which is a contradiction. Conversely, if $d_c(X, Y) < D_c(X, Y)$, then there exists an edit script $\bar{\delta}$ such that $\bar{\delta}(X) = Y$ and $c(\bar{\delta}, X) < D_c(X, Y)$. However, we have shown that we can construct a tree mapping $M_{\bar{\delta}}$ between X and Y such that $c(M_{\bar{\delta}}, X, Y) \leq c(\bar{\delta}, X)$. Therefore, $D_c(X, Y) \leq c(M_{\bar{\delta}}, X, Y) \leq c(\bar{\delta}, X) < D_c(X, Y)$, which is also a contradiction. This only leaves the option $D_c(X, Y) = d_c(X, Y)$, which concludes the proof. □

As an example for the first construction in Lemma A.3, consider the trees $\tilde{x} = \mathtt{a}(\mathtt{b})$ and $\tilde{y} = \mathtt{c}(\mathtt{d})$, as well as the tree mapping $M = \{(1, 2)\}$. M would be translated into the following three edit scripts. First, $\bar{\delta}_M^{\text{rep}} = \text{rep}_{1,y_2} = \text{rep}_{1,\mathtt{d}}$; second, $\bar{\delta}_M^{\text{del}} = \text{del}_2$; and third, $\bar{\delta}_M^{\text{ins}} = \text{ins}_{\text{par}_{\tilde{y}}(1), y_1, r_{\tilde{y}}(1), r_{\tilde{y}}(1) + R_{M,\tilde{x},\tilde{y}}(1)} = \text{ins}_{0,\mathtt{c},1,2}$. Note that the third construction works because

$$R_{M,\tilde{x},\tilde{y}}(1) = |\text{adj}_{\tilde{y}}(1) \cap J^C(M, \tilde{x}, \tilde{y})| + \sum_{j' \in \text{adj}_{\tilde{y}}(1) \cap J(M,\tilde{x},\tilde{y})} R_{M,\tilde{x},\tilde{y}}(j') = |\{2\} \cap \{2\}| + 0 = 1$$

Accordingly, the tree mapping $M = \{(1, 2)\}$ would be translated into the edit script $\bar{\delta}_M = \text{rep}_{1,\mathtt{d}}\text{del}_2\text{ins}_{0,\mathtt{c},1,2}$, which does indeed result in $\bar{\delta}_M(\tilde{x}) = \text{del}_2\text{ins}_{0,\mathtt{c},1,2}(\mathtt{d}(\mathtt{b})) = \text{ins}_{0,\mathtt{c},1,2}(\mathtt{d}) = \mathtt{c}(\mathtt{d}) = \tilde{y}$. The costs are $c(\bar{\delta}_M, \tilde{x}) = c(\mathtt{a}, \mathtt{d}) + c(\mathtt{b}, -) + c(-, \mathtt{d}) = c(M, \tilde{x}, \tilde{y})$.

As an example for the second construction in Lemma A.3, consider the trees $\tilde{x} = \mathtt{a}$ and $\tilde{y} = \mathtt{b}$, as well as the edit script $\bar{\delta} = \text{rep}_{1,\mathtt{c}}\text{ins}_{0,\mathtt{b},1,2}\text{del}_2$. This edit script would be translated into a tree mapping as follows. First, we initialize our tree mapping as $M_\epsilon = \{(1, 1)\}$. Next, consider the first edit, $\delta_1 = \text{rep}_{1,\mathtt{c}}$, which transforms $\tilde{x}$ into $\text{rep}_{1,\mathtt{c}}(\mathtt{a}) = \mathtt{c}$. The corresponding tree mapping remains $M_{\text{rep}_{1,\mathtt{c}}} = \{(1, 1)\}$. Next, consider the second edit, $\delta_2 = \text{ins}_{0,\mathtt{b},1,2}$, which transforms $\mathtt{c}$ into $\text{ins}_{0,\mathtt{b},1,2}(\mathtt{c}) = \mathtt{b}(\mathtt{c})$. The according tree mapping would thus be $M_{\text{rep}_{1,\mathtt{c}}\text{ins}_{0,\mathtt{b},1,2}} = \{(1, 2)\}$. Finally, consider the third edit, $\delta_3 = \text{del}_2$, which transforms $\mathtt{b}(\mathtt{c})$ into $\text{del}_2(\mathtt{b}(\mathtt{c})) = \tilde{y}$. The according tree mapping would thus become $M_{\bar{\delta}} = \emptyset$. For the costs we obtain

$$c(M_{\bar{\delta}}, \tilde{x}, \tilde{y}) = c(\mathtt{a}, -) + c(-, \mathtt{b}) \overset{\textit{triang.}}{\leq} c(\mathtt{a}, \mathtt{c}) + c(-, \mathtt{b}) + c(\mathtt{c}, -) = c(\bar{\delta}, \tilde{x})$$

By virtue of Lemma A.3 we can compute the cheapest tree mapping between two forests instead of the cheapest edit script which transforms one forest into the other, as long as our cost function fulfills the triangular inequality and is self-equal. This already simplifies our problem significantly because there is only a finite number of possible valid tree mappings between two input forests, while there is an infinite number of edit scripts. However, the number of tree mappings is in $\mathcal{O}(2^{|\tilde{x}| \cdot |\tilde{y}|})$ such that an exhaustive enumeration is infeasible. Instead, Zhang and Shasha (1989) propose a dynamic programming scheme which relies on decomposing the edit distance between two input forests into edit distances between subforests. In particular, we define subforests as follows.

Definition A.5 (subforest)**.** Let $\mathcal{A}$ be an alphabet, let X be a forest over $\mathcal{A}$, and let $k \in \mathbb{N}$, $i \in \mathbb{Z}$. Then, we define the *subforest* $X[k,i]$ from k to i recursively as follows. If $X = \epsilon$, then $X[k,i] := \epsilon$. Otherwise, let $X = x(X_1), X'$ for some $x \in \mathcal{A}$ and some forests $X_1, X' \in \mathcal{T}(\mathcal{A})^*$. In that case, we define:

$$X[k,i] := \begin{cases} \epsilon & \text{if } k > i \vee k > |X| \\ (X_1, X')[k-1, i-1] & \text{if } 1 < k \leq i \\ x(X_1[1, i-1]), X'[1, i - |X_1| - 1] & \text{if } 1 = k \leq i \end{cases} \tag{A.13}$$

For example, the subforest $(\mathtt{a}, \mathtt{b}, \mathtt{c})[2,3]$ would be $\mathtt{b}, \mathtt{c}$. The subforest $\tilde{x}[2,4]$ for $\tilde{x} = \mathtt{a}(\mathtt{b}(\mathtt{c}, \mathtt{d}), \mathtt{e})$ would be $\mathtt{b}(\mathtt{c}, \mathtt{d})$. In general, subforests maintain the structure of the input forest, as the following Lemma demonstrates.

Lemma A.4. *Let $\mathcal{A}$ be an alphabet, and let $X \neq \epsilon$ be a forest over $\mathcal{A}$. Then, for any $i \in \{1, \ldots, |X|\}$ it holds: $X[i, rl_X(i)] = \tilde{x}^i$, that is, the subforest from i to $rl_X(i)$ is the ith subtree according to pre-order.*

Proof. Note that $X \neq \epsilon$ and $i \leq rl_X(i) \leq |X|$ such that the first case of Equation A.13 does not apply. Now, let $X = x(X_1), X'$ for some $x \in \mathcal{A}$ and some forests $X_1, X' \in \mathcal{T}(\mathcal{A})^*$ and consider the third case of Equation A.13, that is, $i = 1$. In that case, we obtain $X[1, rl_X(1)] = X[1, |X_1| + 1] = x(X_1[1, |X_1|), X'[1, 0] = x(X_1[1, |X_1|)$. Recursive application of case 3 yields $x(X_1[1, |X_1|) = \ldots = x(X_1, \epsilon[1, 0]) = x(X_1) = \tilde{x}^1$.

Now, consider case 2 of Equation A.13, that is, $i > 1$, and distinguish the following subcases.

If $\text{par}_X(i) = 0$, let $X = \tilde{x}_1, \ldots, \tilde{x}_R$ and let $r \in \{1, \ldots, R\}$ be the index such that $\tilde{x}^i = \tilde{x}_r$. Accordingly, $i = \sum_{l=1}^{r-1} |\tilde{x}_l|$. Further, let $\tilde{x}^i = \tilde{x}_r = x_i(X^i)$ for some forest X^i. Now, recursive application of case 2 of Equation A.13 yields $X[i, rl_X(i)] = (X_1, X')[i-1, rl_X(i) - 1] = \ldots = (\tilde{x}_2, \ldots, \tilde{x}_R)[i - |\tilde{x}_1|, rl_X(i) - |\tilde{x}_1|] = \ldots = (\tilde{x}_r, \ldots, \tilde{x}_R)[1, |\tilde{x}_r|]$ At this point, case 3 of Equation A.13 applies and yields $(\tilde{x}_r, \ldots, \tilde{x}_R)[1, |\tilde{x}_r|] = x_i(X^i[1, |X^i|]), (\tilde{x}_{r+1}, \tilde{x}_R)[1, 0] = x_i(X^i[1, |X^i|]) = \ldots = \tilde{x}^i$, which concludes the proof. □

Using the concept of subforests, we can now go on to establish the Bellman equations which will form the basis for the dynamic programming Algorithm 2.1.

Lemma A.5. *Let $\mathcal{A}$ be an alphabet and let X, Y be non-empty forests over $\mathcal{A}$. Further, let c be a cost function over $\mathcal{A}$.*

Then, for any $i \in \{1, \ldots, |X| + 1\}$, $j \in \{1, \ldots, |Y| + 1\}$, $k \in \text{anc}_X(i) \cup \{i\}$, and $l \in \text{anc}_Y(j) \cup \{j\}$ it holds:

$$D_c(\epsilon, \epsilon) = 0 \tag{A.14}$$
$$D_c(X[i, rl_X(k)], \epsilon) = c(x_i, -) + D_c(X[i+1, rl_X(k)], \epsilon) \tag{A.15}$$
$$D_c(\epsilon, Y[j, rl_Y(l)]) = c(-, y_j) + D_c(\epsilon, Y[j+1, rl_Y(l)]) \tag{A.16}$$

$$\begin{aligned} D_c(X[i, rl_X(k)], Y[j, rl_Y(l)]) = \min \Big\{ \\ & c(x_i, -) + D_c(X[i+1, rl_X(k)], Y[j, rl_Y(l)]), \\ & c(-, y_j) + D_c(X[i, rl_X(k)], Y[j+1, rl_Y(l)]), \\ & c(x_i, y_j) + D_c(X[i+1, rl_X(i)], Y[j+1, rl_Y(j)]) + \\ & D_c(X[rl_X(i)+1, rl_X(k)], Y[rl_Y(j)+1, rl_Y(l)]) \Big\} \end{aligned} \tag{A.17}$$

$$\begin{aligned} D_c(X[i, rl_X(k)], Y[j, rl_Y(l)]) = \min \Big\{ \\ & c(x_i, -) + D_c(X[i+1, rl_X(k)], Y[j, rl_Y(l)]), \\ & c(-, y_j) + D_c(X[i, rl_X(k)], Y[j+1, rl_Y(l)]), \\ & D_c(\tilde{x}_i, \tilde{y}_j) + D_c(X[rl_X(i)+1, rl_X(k)], Y[rl_Y(j)+1, rl_Y(l)]) \Big\} \end{aligned} \tag{A.18}$$

$$\begin{aligned} D_c(\tilde{x}_i, \tilde{y}_j) = \min\{ & c(x_i, -) + D_c(X[i+1, rl_X(i)], Y[j, rl_Y(j)]), \\ & c(-, y_j) + D_c(X[i, rl_X(i)], Y[j+1, rl_Y(j)]), \\ & c(x_i, y_j) + D_c(X[i+1, rl_X(i)], Y[j+1, rl_Y(j)]) \Big\} \end{aligned} \tag{A.19}$$

Proof. First, consider Equations A.14, A.15, and A.16. In all these cases, only the empty tree mapping $M = \emptyset$ is possible because at least one input forest is empty. The cost of the empty tree mapping for any two forests X and Y is

$$c(\emptyset, X, Y) = \sum_{i=1}^{|X|} c(x_i, -) + \sum_{j=1}^{|Y|} c(-, y_j)$$

This cost decomposes as desired, in particular:

$$\begin{aligned} c(\emptyset, \epsilon, \epsilon) &= 0, \\ c(\emptyset, X[i, rl_X(k)], \epsilon) &= c(x_i, -) + c(\emptyset, X[i+1, rl_X(k)], \epsilon), \qquad \text{and} \\ c(\emptyset, \epsilon, Y[j, rl_Y(l)]) &= c(-, y_j) + c(\emptyset, \epsilon, Y[j+1, rl_Y(l)]) \end{aligned}$$

Next, consider Equations A.17 and A.18. In particular, let M be a tree mapping between the subforests $X[i, rl_X(k)]$ and $Y[j, rl_Y(l)]$ such that $c(M, X[i, rl_X(k)], Y[j, rl_Y(l)]) = D_c(X[i, rl_X(k)], Y[j, rl_Y(l)])$. To avoid symbol clutter, we will use the shorthands $X_i := X[i, rl_X(k)]$, $X_{i+1} := X[i+1, rl_X(k)]$, $Y_j := Y[j, rl_Y(l)]$, and $Y_{j+1} := Y[j+1, rl_Y(l)]$. Now, one of the following three cases has to apply:

$1 \in I(M, X_i, Y_j)$: In this case, $M' := \{(i'-1, j') | (i', j') \in M\}$ is a tree mapping between X_{i+1} and Y_j. Further, it holds $c(M', X_{i+1}, Y_j) = D_c(X_{i+1}, Y_j)$. Otherwise, there would exist a tree mapping $\tilde{M}'$ between X_{i+1} and Y_j, such that $c(\tilde{M}', X_{i+1}, Y_j) < c(M', X_{i+1}, Y_j)$. In that case, consider $\tilde{M} := \{(i'+1, j') | (i', j') \in \tilde{M}'\}$, which is a tree mapping between X_i and Y_j, such that:

$$\begin{aligned} D_c(X_i, Y_j) &\leq c(\tilde{M}, X_i, Y_j) = c(x_i, -) + c(\tilde{M}', X_{i+1}, Y_j) \\ &< c(x_i, -) + c(M', X_{i+1}, Y_j) = c(M, X_i, Y_j) = D_c(X_i, Y_j) \end{aligned}$$

which is a contradiction. Therefore, it holds:

$$\begin{aligned} D_c(X_i,Y_j) &= c(M, X_i, Y_j) = c(x_i, -) + c(M', X_{i+1}, Y_j) \\ &= c(x_i, -) + D_c(X_{i+1}, Y_j) \end{aligned} \tag{A.20}$$

$1 \in J(M, X_i, Y_j)$: In this case, $M' := \{(i', j' - 1)|(i', j') \in M\}$ is a tree mapping between X_i and Y_{j+1}. Further, it holds $c(M', X_i, Y_{j+1}) = D_c(X_i, Y_{j+1})$. Otherwise, there would exist a tree mapping $\tilde{M}'$ between X_i and Y_{j+1}, such that $c(\tilde{M}', X_i, Y_{j+1}) < c(M', X_i, Y_{j+1})$. In that case, consider $\tilde{M} := \{(i', j' + 1)|(i', j') \in \tilde{M}'\}$, which is a tree mapping between X_i and Y_j, such that:

$$\begin{aligned} D_c(X_i, Y_j) \le c(\tilde{M}, X_i, Y_j) &= c(-, y_j) + c(\tilde{M}', X_i, Y_{j+1}) \\ &< c(-, y_j) + c(M', X_i, Y_{j+1}) = c(M, X_i, Y_j) = D_c(X_i, Y_j) \end{aligned}$$

which is a contradiction. Therefore, it holds:

$$\begin{aligned} D_c(X_i,Y_j) &= c(M, X_i, Y_j) = c(-, y_j) + c(M', X_i, Y_{j+1}) \\ &= c(-, y_j) + D_c(X_i, Y_{j+1}) \end{aligned} \tag{A.21}$$

$1 \in I^C(M, X_i, Y_j) \wedge 1 \in J^C(M, X_i, Y_j)$: In this case, we first show that $(1, 1) \in M$. If that would not be the case, there would exist a $i \in \{1, \ldots, |X_i|\}$ and a $j \in \{1, \ldots, |Y_j|\}$, such that $(1, j) \in M$, $(i, 1) \in M$, and $i \neq 1$ or $j \neq 1$. If $i > 1$, Equation 2.21 implies that $j < 1$, which is a contradiction. Conversely, if $j > 1$, Equation 2.21 implies that $i < 1$, which is a contradiction. Therefore, $i = j = 1$ and, thus, $(1, 1) \in M$.

Now, Equation 2.22 implies that for all $(i', j') \in M$ it must hold: $1 \in \text{anc}_{X_i}(i') \iff 1 \in \text{anc}_{Y_j}(j')$. In conjunction with Equation A.6, we obtain $1 \le i' \le |\tilde{x}^i| \iff 1 \le j' \le |\tilde{y}^j|$. Accordingly, M must be decomposable as $M = M_1 \cup M_2$ where for all $(i', j') \in M_1$ it holds $i' \le |\tilde{x}^i|$ and $j' \le |\tilde{y}^j|$; and for all $(i', j') \in M_2$ it holds $i' > |\tilde{x}^i|$ and $j' > |\tilde{y}^j|$. This, in turn, implies that M_1 is a tree mapping between $X[i, rl_X(i)] \overset{\text{Lemma A.4}}{=} \tilde{x}^i$ and $Y[j, rl_Y(j)] \overset{\text{Lemma A.4}}{=} \tilde{y}^j$, and $M_2' := \{(i' - |\tilde{x}^i|, j' - |\tilde{y}^j|)|(i', j') \in M_2\}$ is a tree mapping between $X' := X[rl_X(i) + 1, rl_X(k)]$ and $Y' := Y[rl_Y(j) + 1, rl_Y(l)]$.

Further, it holds $c(M_1, \tilde{x}^i, \tilde{y}^j) = D_c(\tilde{x}^i, \tilde{y}^j)$. Otherwise, there would exist a tree mapping $\tilde{M}_1$ between $\tilde{x}^i$ and $\tilde{y}^j$, such that $c(\tilde{M}_1, \tilde{x}^i, \tilde{y}^j) < c(M_1, \tilde{x}^i, \tilde{y}^j)$. In that case, consider $\tilde{M} := \tilde{M}_1 \cup M_2$, which is a tree mapping between X_i and Y_j such that:

$$\begin{aligned} D_c(X_i, Y_j) \le c(\tilde{M}, X_i, Y_j) &= c(\tilde{M}_1, \tilde{x}^i, \tilde{y}^j) + c(M_2', X', Y') \\ &< c(M_1, \tilde{x}^i, \tilde{y}^j) + c(M_2', X', Y') = c(M, X_i, Y_j) = D_c(X_i, Y_j) \end{aligned}$$

which is a contradiction. Also, it holds $c(M_2, X', Y') = D_c(X', Y')$. Otherwise, there would exist a tree mapping $\tilde{M}_2'$ between X' and Y', such that $c(\tilde{M}_2', X', Y') < c(M_2, X', Y')$. In that case, consider $\tilde{M} := M_1 \cup \{(i' + |\tilde{x}_i|, j' + |\tilde{y}_j|)|(i', j') \in \tilde{M}_2'\}$ which is a tree mapping between X_i and Y_j such that:

$$\begin{aligned} D_c(X_i, Y_j) \le c(\tilde{M}, X_i, Y_j) &= c(M_1, \tilde{x}^i, \tilde{y}^j) + c(\tilde{M}_2', X', Y') \\ &< c(M_1, \tilde{x}^i, \tilde{y}^j) + c(M_2', X', Y') = c(M, X_i, Y_j) = D_c(X_i, Y_j) \end{aligned}$$

which is a contradiction. Therefore, we obtain:

$$\begin{aligned} D_c(X_i,Y_j) &= c(M, X_i, Y_j) = c(M_1, \tilde{x}^i, \tilde{y}^j) + c(M_2', X', Y') \\ &= D_c(\tilde{x}^i, \tilde{y}^j) + D_c\big(X[rl_X(i)+1, rl_X(k)], Y[rl_Y(j)+1, rl_Y(l)]\big) \end{aligned} \tag{A.22}$$

Finally, consider the term $D_c(\tilde{x}^i, \tilde{y}^j)$. Because $(1,1) \in M_1$, it follows that $M_1' := \{(i'-1, j'-1) | (i', j') \in M_1 \setminus \{(1,1)\}\}$ is a tree mapping between $X'_{i+1} := X[i+1, rl_X(i)]$ and $Y'_{j+1} := Y[j+1, rl_Y(j)]$. Further, it holds $c(M_1', X'_{i+1}, Y'_{j+1}) = D_c(X'_{i+1}, Y'_{j+1})$. If that would not be the case, there would exist a tree mapping $\tilde{M}_1'$ between X'_{i+1} and Y'_{j+1}, such that $c(\tilde{M}_1', X'_{i+1}, Y'_{j+1}) < c(M_1', X'_{i+1}, Y'_{j+1})$. In that case, consider $\tilde{M}_1 := \{(1,1)\} \cup \{(i'+1, j'+1) | (i', j') \in \tilde{M}_1'\}$, which is a tree mapping between $\tilde{x}^i$ and $\tilde{y}^j$, such that:

$$\begin{aligned} D_c(\tilde{x}^i, \tilde{y}^j) &\leq c(\tilde{M}_1, \tilde{x}^i, \tilde{y}^j) = c(x_i, y_j) + c(\tilde{M}_1', X'_{i+1}, Y'_{j+1}) \\ &< c(x_i, y_j) + c(M_1', X'_{i+1}, Y'_{j+1}) = c(M_1, \tilde{x}^i, \tilde{y}^j) = D_c(\tilde{x}^i, \tilde{y}^j) \end{aligned}$$

which is a contradiction. Therefore, it holds:

$$\begin{aligned} D_c(\tilde{x}^i,\tilde{y}^j) &= c(M_1, \tilde{x}^i, \tilde{y}^j) = c(x_i, y_j) + c(M_1', X'_{i+1}, Y'_{j+1}) \\ &= c(x_i, y_j) + D_c(X[i+1, rl_X(i)], Y[j+1, rl_Y(j)]) \end{aligned} \tag{A.23}$$

Note that these three cases are exhaustive, that is, one of the Equations A.20, A.21, or A.22 has to apply. Further, the cheapest option of these three has to apply, otherwise $c(M, X_i, Y_j) > D_c(X_i, Y_j)$, which would be a contradiction. The minimum of Equations A.20, A.21, and A.22 yields Equation A.18. If we then plug Equation A.23 into Equation A.18 we obtain Equation A.17.

Finally, consider Equation A.19. We obtain this equation by setting $k = i$ and $l = j$ in Equation A.17, thus yielding:

$$\begin{aligned} D_c(\tilde{x}^i,\tilde{y}^j) \overset{\text{Lemma A.4}}{=} D_c(X[i, rl_X(i)], Y[j, rl_Y(j)]) = \min \Big\{ \\ & c(x_i, -) + D_c(X[i+1, rl_X(i)], Y[j, rl_Y(j)]), \\ & c(-, y_j) + D_c(X[i, rl_X(i)], Y[j+1, rl_Y(j)]), \\ & c(x_i, y_j) + D_c(X[i+1, rl_X(i)], Y[j+1, rl_Y(j)]) + \\ & D_c(X[rl_X(i)+1, rl_X(i)], Y[rl_Y(j)+1, rl_Y(j)]) \Big\} \end{aligned}$$

Note that $X[rl_X(i)+1, rl_X(i)] = \epsilon$ and $Y[rl_Y(j)+1, rl_Y(j)] = \epsilon$. Therefore, $D_c(X[rl_X(i) + 1, rl_X(i)], Y[rl_Y(j)+1, rl_Y(j)]) = D_c(\epsilon, \epsilon) \overset{\text{Eq. A.14}}{=} 0$, which in turn yields Equation A.19. □

An example for the decompositions in Equations A.18 and A.19 is shown in Figure A.1.

Using these decompositions, we can finally prove the invariants of Algorithm 2.1, which then imply the correctness of the algorithm.

Lemma A.6. *Let $\mathcal{A}$ be an alphabet, let $\tilde{x}$ and $\tilde{y}$ be trees over $\mathcal{A}$, and let c be a cost function over $\mathcal{A}$. Then, after each completion of lines 6-26 in Algorithm 2.1 for the input $\tilde{x}$, $\tilde{y}$, and c it holds for all $i \in \{k, \ldots, rl_{\tilde{x}}(k)\}$ and all $j \in \{l, \ldots, rl_{\tilde{y}}(l)\}$:*

$$\boldsymbol{D}_{i,j} = D_c(\tilde{x}[i, rl_{\tilde{x}}(k)], \tilde{y}[j, rl_{\tilde{y}}(l)]) \quad \textit{and} \tag{A.24}$$

$$\boldsymbol{d}_{i,j} = D_c(\tilde{x}^i, \tilde{y}^j) \tag{A.25}$$

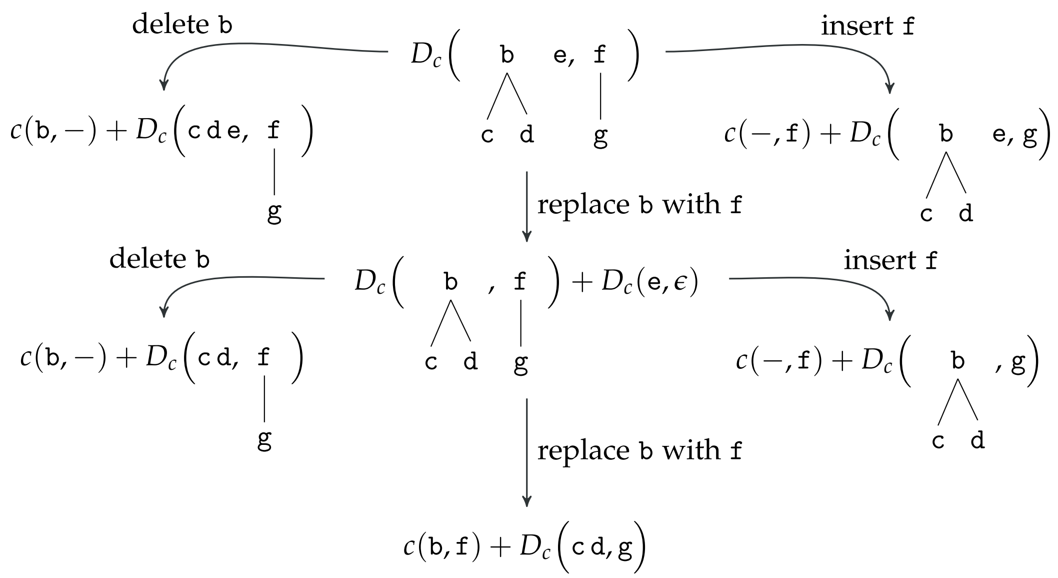


Figure A.1: An illustration of the decompositions in Equations A.18 (top) and A.19 (bottom) for the example subforests $X_i = \mathtt{b}(\mathtt{c},\mathtt{d}),\mathtt{e}$ and $Y_j = \mathtt{f}(\mathtt{g})$.

Proof. We perform an inductive argument over i and j in descending order. First, consider the base cases. If $i = rl_{\tilde{x}}(k)+1$ and $j = rl_{\tilde{y}}(l)+1$, we obtain $D_c(\tilde{x}[rl_{\tilde{x}}(k)+1, rl_{\tilde{x}}(k)], \tilde{y}[rl_{\tilde{y}}(l)+1, rl_{\tilde{y}}(l)]) = D_c(\epsilon,\epsilon) \overset{\text{Eq. A.14}}{=} 0$, which is correctly computed in line 6.

Further, if $i \leq rl_{\tilde{x}}(k)$ and $j = rl_{\tilde{y}}(l)+1$, we obtain $D_c(\tilde{x}[i, rl_{\tilde{x}}(k)], \tilde{y}[rl_{\tilde{y}}(l)+1, rl_{\tilde{y}}(l)]) = D_c(\tilde{x}[i, rl_{\tilde{x}}(k)], \epsilon) \overset{\text{Eq. A.15}}{=} c(x_i,-) + D_c(\tilde{x}[i+1, rl_{\tilde{x}}(k)], \tilde{y}[rl_{\tilde{y}}(l)+1, rl_{\tilde{y}}(l)])$, which is correctly computed in lines 7-9 for all $i \in \{k,\ldots,rl_{\tilde{x}}(k)\}$.

Similarly, if $j \leq rl_{\tilde{y}}(l)$ and $i = rl_{\tilde{x}}(k)+1$, we obtain $D_c(\tilde{x}[rl_{\tilde{x}}(k)+1, rl_{\tilde{x}}(k)], \tilde{y}[j, rl_{\tilde{y}}(l)]) = D_c(\epsilon, \tilde{y}[j, rl_{\tilde{y}}(l)]) \overset{\text{Eq. A.15}}{=} c(-,y_j) + D_c(\tilde{x}[rl_{\tilde{x}}(k)+1, rl_{\tilde{x}}(k)], \tilde{y}[j+1, rl_{\tilde{y}}(l)])$, which is correctly computed in lines 10-12 for all $j \in \{l,\ldots,rl_{\tilde{y}}(l)\}$.

Now, consider the case $i \leq rl_{\tilde{x}}(k)$ and $j \leq rl_{\tilde{y}}(l)$. Per induction, we already know that

$$\begin{aligned}
\boldsymbol{D}_{i+1,j} &= D_c(\tilde{x}[i+1, rl_{\tilde{x}}(k)], \tilde{y}[j, rl_{\tilde{y}}(l)]),\\
\boldsymbol{D}_{i,j+1} &= D_c(\tilde{x}[i, rl_{\tilde{x}}(k)], \tilde{y}[j+1, rl_{\tilde{y}}(l)]),\\
\boldsymbol{D}_{i+1,j+1} &= D_c(\tilde{x}[i+1, rl_{\tilde{x}}(k)], \tilde{y}[j+1, rl_{\tilde{y}}(l)]), \qquad \text{and}\\
\boldsymbol{D}_{rl_{\tilde{x}}(i)+1, rl_{\tilde{y}}(j)+1} &= D_c(\tilde{x}[rl_{\tilde{x}}(i)+1, rl_{\tilde{x}}(k)], \tilde{y}[rl_{\tilde{y}}(j)+1, rl_{\tilde{y}}(l)]).
\end{aligned}$$

Now, distinguish the following cases.

If $rl_{\tilde{x}}(i) = rl_{\tilde{x}}(k)$ and $rl_{\tilde{y}}(j) = rl_{\tilde{y}}(l)$, we obtain $\tilde{x}[i, rl_{\tilde{x}}(k)] = \tilde{x}[i, rl_{\tilde{x}}(i)] \overset{\text{Lemma A.4}}{=} \tilde{x}^i$ and $\tilde{y}[j, rl_{\tilde{y}}(l)] = \tilde{y}[j, rl_{\tilde{y}}(j)] \overset{\text{Lemma A.4}}{=} \tilde{y}^j$, such that $D_c(\tilde{x}[i, rl_{\tilde{x}}(k)], \tilde{y}[j, rl_{\tilde{y}}(l)]) = D_c(\tilde{x}^i, \tilde{y}^j)$, which can be computed according to Equation A.19. Therefore, lines 16-18 of Algorithm 2.1 ensure that $\boldsymbol{D}_{i,j} = D_c(\tilde{x}[i, rl_{\tilde{x}}(k)], \tilde{y}[j, rl_{\tilde{y}}(l)])$. Further, because $\boldsymbol{D}_{i,j}$ is now equivalent to $D_c(\tilde{x}^i, \tilde{y}^j)$, line 19 is correct as well.

If $rl_{\tilde{x}}(i) \neq rl_{\tilde{x}}(k)$ or $rl_{\tilde{y}}(j) \neq rl_{\tilde{y}}(l)$, the decomposition of $D_c(\tilde{x}[i, rl_{\tilde{x}}(k)], \tilde{y}[j, rl_{\tilde{y}}(l)])$ according to Equation A.18 applies. Accordingly, lines 21-23 of Algorithm 2.1 ensure $\boldsymbol{D}_{i,j} = D_c(\tilde{x}[i, rl_{\tilde{x}}(k)], \tilde{y}[j, rl_{\tilde{y}}(l)])$, under the condition that $d_{i,j} = D_c(\tilde{x}^i, \tilde{y}^j)$. We know that this condition holds if we have executed lines 6-26 before with the keyroots $\mathrm{k}_{\tilde{x}}(i)$ and $\mathrm{k}_{\tilde{y}}(j)$. Because the loops in lines 4-5 iterate the keyroots in descending oder, it remains to show that $k < \mathrm{k}_{\tilde{x}}(i)$ and $l \leq \mathrm{k}_{\tilde{y}}(j)$, or $k \leq \mathrm{k}_{\tilde{x}}(i)$ and $l < \mathrm{k}_{\tilde{y}}(j)$.

First, consider the case $rl_{\tilde{x}}(i) \neq rl_{\tilde{x}}(k)$. In that case, $k \neq \mathrm{k}_{\tilde{x}}(i)$ and $k \neq i$, otherwise $rl_{\tilde{x}}(k) = rl_{\tilde{x}}(i) = rl_{\tilde{x}}(\mathrm{k}_{\tilde{x}}(i))$, which is a contradiction. Further, it must hold $k < i \leq rl_{\tilde{x}}(k)$, otherwise i would not be accessed in the loop in line 13. In turn, Equation A.7 implies that $k \in \mathrm{anc}_{\tilde{x}}(i)$. Further, due to the definition of keyroots, $\mathrm{k}_{\tilde{x}}(i) \leq i \leq rl_{\tilde{x}}(i) = rl_{\tilde{x}}(\mathrm{k}_{\tilde{x}}(i))$. Now, if $\mathrm{k}_{\tilde{x}}(i) = i$, we obtain $k < i = \mathrm{k}_{\tilde{x}}(i)$ as desired. Otherwise, we obtain $\mathrm{k}_{\tilde{x}}(i) < i \leq rl_{\tilde{x}}(\mathrm{k}_{\tilde{x}}(i))$, such that Equation A.7 implies that $\mathrm{k}_{\tilde{x}}(i) \in \mathrm{anc}_{\tilde{x}}(i)$. Now, assume that $\mathrm{k}_{\tilde{x}}(i) < k$. In that case, Equation A.8 implies $\mathrm{k}_{\tilde{x}}(i) \in \mathrm{anc}_{\tilde{x}}(k)$. Consequently, Equation A.6 tells us that $rl_{\tilde{x}}(k) \leq rl_{\tilde{x}}(\mathrm{k}_{\tilde{x}}(i))$. However, due to $k \in \mathrm{anc}_{\tilde{x}}(i)$, Equation A.6 also tells us that $rl_{\tilde{x}}(k) \geq rl_{\tilde{x}}(i) = rl_{\tilde{x}}(\mathrm{k}_{\tilde{x}}(i))$, such that $rl_{\tilde{x}}(k) = rl_{\tilde{x}}(\mathrm{k}_{\tilde{x}}(i)) = rl_{\tilde{x}}(i)$, which is a contradiction. Therefore, we can conclude that $k < \mathrm{k}_{\tilde{x}}(i)$. It remains to show that $l \leq \mathrm{k}_{\tilde{y}}(j)$. If $rl_{\tilde{y}}(l) = rl_{\tilde{y}}(j)$, then $l = \mathrm{k}_{\tilde{y}}(j)$, because the minimum is unique. Otherwise, $rl_{\tilde{y}}(l) \neq rl_{\tilde{y}}(j)$.

If $rl_{\tilde{y}}(j) \neq rl_{\tilde{y}}(l)$, we know that $l \neq \mathrm{k}_{\tilde{y}}(j)$ and $l \neq j$, otherwise $rl_{\tilde{y}}(l) = rl_{\tilde{y}}(j) = rl_{\tilde{y}}(\mathrm{k}_{\tilde{y}}(j))$, which is a contradiction. Further, it must hold $l < j \leq rl_{\tilde{y}}(l)$, otherwise j would not be accessed in the loop in line 14. In turn, Equation A.7 implies that $l \in \mathrm{anc}_{\tilde{y}}(j)$. Further, due to the definition of keyroots, $\mathrm{k}_{\tilde{y}}(j) \leq j \leq rl_{\tilde{y}}(j) = rl_{\tilde{y}}(\mathrm{k}_{\tilde{y}}(j))$. Now, if $\mathrm{k}_{\tilde{y}}(j) = j$, we obtain $l < j = \mathrm{k}_{\tilde{y}}(j)$ as desired. Otherwise, we obtain $\mathrm{k}_{\tilde{y}}(j) < j \leq rl_{\tilde{y}}(\mathrm{k}_{\tilde{y}}(j))$, such that Equation A.7 implies that $\mathrm{k}_{\tilde{y}}(j) \in \mathrm{anc}_{\tilde{y}}(j)$. Now, assume that $\mathrm{k}_{\tilde{y}}(j) < l$. In that case, Equation A.8 implies $\mathrm{k}_{\tilde{y}}(j) \in \mathrm{anc}_{\tilde{y}}(l)$. Consequently, Equation A.6 tells us that $rl_{\tilde{y}}(l) \leq rl_{\tilde{y}}(\mathrm{k}_{\tilde{y}}(j))$. However, due to $l \in \mathrm{anc}_{\tilde{y}}(j)$, Equation A.6 also tells us that $rl_{\tilde{y}}(l) \geq rl_{\tilde{y}}(j) = rl_{\tilde{y}}(\mathrm{k}_{\tilde{y}}(j))$, such that $rl_{\tilde{y}}(l) = rl_{\tilde{y}}(\mathrm{k}_{\tilde{y}}(j)) = rl_{\tilde{y}}(j)$, which is a contradiction. Therefore, we can conclude that $l < \mathrm{k}_{\tilde{y}}(j)$. It remains to show that $k \leq \mathrm{k}_{\tilde{x}}(i)$. If $rl_{\tilde{x}}(k) = rl_{\tilde{x}}(i)$, then $k = \mathrm{k}_{\tilde{x}}(i)$, because the minimum is unique. Otherwise, $rl_{\tilde{x}}(k) \neq rl_{\tilde{x}}(i)$, which implies $k < \mathrm{k}_{\tilde{x}}(i)$, as we have shown above. □

Now, we can finally complete the proof. First, note that Lemma A.3 implies that $d_c(\tilde{x}, \tilde{y})$ is equivalent to $D_c(\tilde{x}^1, \tilde{y}^1)$ if c fulfills the triangular inequality and is self-equal. Further, Lemma A.6 tells us that the output of Algorithm 2.1, $\boldsymbol{d}_{1,1}$, is equal to $D_c(\tilde{x}^1, \tilde{y}^1)$ if keyroots $k \in \mathcal{K}(\tilde{x})$ and $l \in \mathcal{K}(\tilde{y})$ exist such that $k \leq 1 \leq rl_{\tilde{x}}(k)$ and $l \leq 1 \leq rl_{\tilde{y}}(l)$. Per definition of outermost right leaves, we know that $rl_{\tilde{x}}(1) = |\tilde{x}|$ and $rl_{\tilde{y}}(1) = |\tilde{y}|$. Further, per definition of keyroots, we know that $\mathrm{k}_{\tilde{x}}(|\tilde{x}|) = \min\{k | rl_{\tilde{x}}(k) = rl_{\tilde{x}}(|\tilde{x}|)\} = 1$, and $\mathrm{k}_{\tilde{y}}(|\tilde{y}|) = \min\{l | rl_{\tilde{y}}(l) = rl_{\tilde{y}}(|\tilde{y}|)\} = 1$ because 1 is the lowest possible index. Therefore, $1 \in \mathcal{K}(\tilde{x})$ and $1 \in \mathcal{K}(\tilde{y})$, which concludes the overall proof.

A.6 PROOF OF THEOREM 2.7

Recall the theorem we intend to prove.

Under the assumption of fixed $\gamma_{k|i}$, Q (Equation 2.36) is convex with respect to $P(k)$, $P(y|k)$, $\vec{\mu}_k$, and $\boldsymbol{\Lambda}_k$.

Further, the optima of Q with respect to these parameters are given as follows.

$$P(k) = \frac{1}{M} \cdot \sum_{i=1}^{M} \gamma_{k|i} \tag{A.26}$$

$$P(y|k) = \frac{\sum_{i:y_i=y} \gamma_{k|i}}{\sum_{i=1}^{M} \gamma_{k|i}} \tag{A.27}$$

$$\vec{\mu}_k = \frac{\sum_{i=1}^{M} \gamma_{k|i} \cdot \vec{x}_i}{\sum_{i=1}^{M} \gamma_{k|i}} \tag{A.28}$$

$$\boldsymbol{\Lambda}_k = \Big(\frac{\sum_{i=1}^{M} \gamma_{k|i} \cdot (\vec{\mu}_k - \vec{x}_i) \cdot (\vec{\mu}_k - \vec{x}_i)^\top}{\sum_{i=1}^{M} \gamma_{k|i}} \Big)^{-1} \tag{A.29}$$

Finally, if we restrict the precision matrix to be shared across all Gaussians, that is, $\boldsymbol{\Lambda}_1 = \ldots = \boldsymbol{\Lambda}_K = \boldsymbol{\Lambda}$, we obtain the following optimum of Q with respect to $\boldsymbol{\Lambda}$.

$$\boldsymbol{\Lambda} = \Big(\frac{1}{M} \cdot \sum_{i=1}^{M} \sum_{k=1}^{K} \gamma_{k|i} \cdot (\vec{\mu}_k - \vec{x}_i) \cdot (\vec{\mu}_k - \vec{x}_i)^\top \Big)^{-1} \tag{A.30}$$

Proof

The following derivations mostly follow Barber (2012) and Bishop (2006). Our matrix calculus is based on Fackler (2005) and Petersen and Pedersen (2012). In particular, we follow Fackler (2005) in defining the gradient of Q with respect to a $m \times m$ matrix $\boldsymbol{\Lambda}$ as the $m \times m$ matrix $\nabla_{\boldsymbol{\Lambda}} Q$ with entries $\Big(\nabla_{\boldsymbol{\Lambda}} Q\Big)_{r,s} := \frac{\partial}{\partial \boldsymbol{\Lambda}_{r,s}} Q$; and in defining the Hessian of Q with respect to $\boldsymbol{\Lambda}$ as the $m^2 \times m^2$ matrix $\nabla^2_{\boldsymbol{\Lambda}} Q$ which contains the second derivatives of Q with respect to all possible combinations of two matrix entries of $\boldsymbol{\Lambda}$. Equivalently, $\nabla^2_{\boldsymbol{\Lambda}} Q$ can be seen as the Hessian of Q with respect to a vector which contains all entries of $\boldsymbol{\Lambda}$ in concatenated form.

In the following, we check convexity and optima for every single parameter.

Priors $P(k)$: Note that we need to take a side constraint into account, namely that $\sum_{k=1}^{K} P(k) = 1$. This translates into a Lagrange multiplier ν in our optimization, which we take into account for our first and second derivatives:

$$\frac{\partial}{\partial P(k)} Q + \nu \cdot \Big(\sum_{k'=1}^{K} P(k') - 1 \Big) = - \sum_{i=1}^{M} \gamma_{k|i} \cdot \frac{1}{P(k)} + \nu \tag{A.31}$$

$$\frac{\partial^2}{\partial^2 P(k)} Q + \nu \cdot \Big(\sum_{k'=1}^{K} P(k') - 1 \Big) = \sum_{i=1}^{M} \gamma_{k|i} \cdot \frac{1}{P(k)^2} \tag{A.32}$$

Because all terms $\gamma_{k|i}$ are non-negative, we obtain a non-negative second derivative such that every point with zero first derivative is a global optimum. If we set the first derivative to zero we obtain:

$$P(k) = \frac{1}{\nu} \cdot \sum_{i=1}^{M} \gamma_{k|i} \tag{A.33}$$

Due to our side constraint we can infer the correct value for ν:

$$\sum_{k=1}^{K} P(k) = 1 \quad \iff \quad \frac{1}{\nu} \cdot \sum_{k=1}^{K} \sum_{i=1}^{M} \gamma_{k|i} = 1 \quad \iff \quad M = \nu \tag{A.34}$$

which yields Equation A.26.

Label Distributions $P(y|k)$: Again, for all terms $P(y|k)$ we have the side constraint $\sum_{y=1}^{L} P(y|k) = 1$. Accordingly, we consider the following first and second derivatives:

$$\frac{\partial}{\partial P(y|k)} Q + \sum_{k'=1}^{K} \nu_{k'} \cdot \Big(\sum_{y'=1}^{L} P(y'|k') - 1 \Big) = - \sum_{i:y_i=y}^{M} \gamma_{k|i} \cdot \frac{1}{P(y|k)} + \nu_k \tag{A.35}$$

$$\frac{\partial^2}{\partial^2 P(y|k)} Q + \sum_{k'=1}^{K} \nu_{k'} \cdot \Big(\sum_{y'=1}^{L} P(y'|k') - 1 \Big) = \sum_{i:y_i=y}^{M} \gamma_{k|i} \cdot \frac{1}{P(y|k)^2} \tag{A.36}$$

Because all terms $\gamma_{k|i}$ are non-negative, we obtain a non-negative second derivative such that every point with zero first derivative is a global optimum. If we set the first derivative to zero we obtain:

$$P(y|k) = \frac{1}{\nu_k} \cdot \sum_{i:y_i=y} \gamma_{k|i} \tag{A.37}$$

Due to our side constraint we can infer the correct value for ν_k:

$$\sum_{y=1}^{L} P(y|k) = 1 \iff \frac{1}{\nu_k} \cdot \sum_{y=1}^{L} \sum_{i:y_i=y} \gamma_{k|i} = 1 \iff \sum_{i=1}^{M} \gamma_{k|i} = \nu_k \tag{A.38}$$

which yields Equation A.27.

Means $\vec{\mu}_k$: The gradient and the Hessian of Q with respect to $\vec{\mu}_k$ are given as:

$$\nabla_{\vec{\mu}_k} Q = \sum_{i=1}^{M} \gamma_{k|i} \cdot \mathbf{\Lambda}_k \cdot (\vec{\mu}_k - \vec{x}_i) \tag{A.39}$$

$$\nabla^2_{\vec{\mu}_k} Q = \sum_{i=1}^{M} \gamma_{k|i} \cdot \mathbf{\Lambda}_k \tag{A.40}$$

If $\mathbf{\Lambda}_k$ is positive definite, the Hessian is also positive definite such that every vector $\vec{\mu}_k$ with zero gradient is a global optimum. If we set the gradient to zero we obtain:

$$\begin{aligned} & \sum_{i=1}^{M} \gamma_{k|i} \cdot \mathbf{\Lambda}_k \cdot (\vec{\mu}_k - \vec{x}_i) = 0 \\ \iff & \mathbf{\Lambda}_k \cdot \vec{\mu}_k \cdot \sum_{i=1}^{M} \gamma_{k|i} = \mathbf{\Lambda}_k \cdot \sum_{i=1}^{M} \gamma_{k|i} \vec{x}_i \\ \overset{*}{\iff} & \vec{\mu}_k = \frac{\sum_{i=1}^{M} \gamma_{k|i} \cdot \vec{x}_i}{\sum_{i=1}^{M} \gamma_{k|i}} \end{aligned} \tag{A.41}$$

The last step depends on $\mathbf{\Lambda}_k$ being invertible. This is fulfilled if $\mathbf{\Lambda}_k$ is positive definite.

Precision Matrices $\mathbf{\Lambda}_k$: The gradient and the Hessian of Q with respect to $\mathbf{\Lambda}_k$ are given as:

$$\nabla_{\mathbf{\Lambda}_k} Q = \frac{1}{2} \cdot \sum_{i=1}^{M} \gamma_{k|i} \cdot \Big((\vec{\mu}_k - \vec{x}_i) \cdot (\vec{\mu}_k - \vec{x}_i)^\top - \mathbf{\Lambda}_k^{\top -1} \Big) \tag{A.42}$$

$$\nabla^2_{\mathbf{\Lambda}_k} Q = \frac{1}{2} \cdot \sum_{i=1}^{M} \gamma_{k|i} \cdot (\mathbf{\Lambda}_k^{\top -1} \otimes \mathbf{\Lambda}_k^{-1}) \tag{A.43}$$

where $\otimes$ denotes the Kronecker product. Because the set of positive definite matrices is closed under inversion, the Kronecker product, multiplication with positive scalars, and addition, the Hessian is positive definite if $\mathbf{\Lambda}_k$ itself is positive definite (Fackler 2005). In turn, Q is convex with respect to $\mathbf{\Lambda}_k$ if $\mathbf{\Lambda}_k$ is positive definite. Finally, this implies that every positive definite $\mathbf{\Lambda}_k$ with zero gradient is a global optimum.

If we set the gradient to zero we obtain:

$$\begin{aligned}
&\frac{1}{2} \cdot \sum_{i=1}^{M} \gamma_{k|i} \cdot \Big((\vec{\mu}_k - \vec{x}_i) \cdot (\vec{\mu}_k - \vec{x}_i)^\top - \mathbf{\Lambda}_k^{\top\,-1} \Big) = 0 \\
\iff &\mathbf{\Lambda}_k^{\top\,-1} \cdot \sum_{i=1}^{M} \gamma_{k|i} = \sum_{i=1}^{M} \gamma_{k|i} \cdot (\vec{\mu}_k - \vec{x}_i) \cdot (\vec{\mu}_k - \vec{x}_i)^\top \\
\iff &\mathbf{\Lambda}_k = \Big(\frac{\sum_{i=1}^{M} \gamma_{k|i} \cdot (\vec{\mu}_k - \vec{x}_i) \cdot (\vec{\mu}_k - \vec{x}_i)^\top}{\sum_{i=1}^{M} \gamma_{k|i}} \Big)^{-1}
\end{aligned} \tag{A.44}$$

Note that the matrix $\mathbf{\Lambda}_k^{-1}$ is guaranteed to be symmetric and positive semi-definite, because it is a convex combination of outer products. To ensure strict positive definiteness, and thus invertibility, we can add a small positive number to the diagonal of $\mathbf{\Lambda}_k^{-1}$ as suggested by Barber (2012). Given that $\mathbf{\Lambda}_k$ is thus positive definite in every step, the convexity claims above hold.

Shared Precision Matrix $\mathbf{\Lambda}$: If all Gaussians have the same shared precision matrix $\mathbf{\Lambda}$, we obtain the following gradient and Hessian of Q with respect to $\mathbf{\Lambda}$:

$$\nabla_{\mathbf{\Lambda}} Q = \frac{1}{2} \cdot \sum_{i=1}^{M} \sum_{k=1}^{K} \gamma_{k|i} \cdot \Big((\vec{\mu}_k - \vec{x}_i) \cdot (\vec{\mu}_k - \vec{x}_i)^\top - \mathbf{\Lambda}^{\top\,-1} \Big) \tag{A.45}$$

$$\nabla^2_{\mathbf{\Lambda}} Q = \frac{1}{2} \cdot \sum_{i=1}^{M} \sum_{k=1}^{K} \gamma_{k|i} \cdot (\mathbf{\Lambda}^{\top\,-1} \otimes \mathbf{\Lambda}^{-1}) = \frac{M}{2} \cdot (\mathbf{\Lambda}^{\top\,-1} \otimes \mathbf{\Lambda}^{-1}) \tag{A.46}$$

where $\otimes$ denotes the Kronecker product. As noted before, the set of positive definite matrices is closed under inversion, the Kronecker product, and multiplication with positive scalars, such that the Hessian is positive definie if $\mathbf{\Lambda}$ itself is positive definite. In turn, Q is convex with respect to $\mathbf{\Lambda}$ if $\mathbf{\Lambda}$ is positive definite. Finally, this implies that every positive definite $\mathbf{\Lambda}$ with zero gradient is a global optimum.

If we set the gradient to zero we obtain:

$$\begin{aligned}
&\frac{1}{2} \cdot \sum_{i=1}^{M} \sum_{k=1}^{K} \gamma_{k|i} \cdot \Big((\vec{\mu}_k - \vec{x}_i) \cdot (\vec{\mu}_k - \vec{x}_i)^\top - \mathbf{\Lambda}^{\top\,-1} \Big) = 0 \\
\iff &\mathbf{\Lambda}^{\top\,-1} \cdot \sum_{i=1}^{M} \sum_{k=1}^{K} \gamma_{k|i} = \sum_{i=1}^{M} \gamma_{k|i} \cdot (\vec{\mu}_k - \vec{x}_i) \cdot (\vec{\mu}_k - \vec{x}_i)^\top \\
\iff &\mathbf{\Lambda} = \Big(\frac{1}{M} \cdot \sum_{i=1}^{M} \sum_{k=1}^{K} \gamma_{k|i} \cdot (\vec{\mu}_k - \vec{x}_i) \cdot (\vec{\mu}_k - \vec{x}_i)^\top \Big)^{-1}
\end{aligned} \tag{A.47}$$

Note that the matrix $\mathbf{\Lambda}^{-1}$ is guaranteed to be symmetric and positive semi-definite, because it is a convex combination of outer products. To ensure strict positive definiteness, and thus invertibility, we can add a small positive number to the diagonal of $\mathbf{\Lambda}^{-1}$ as suggested by Barber (2012). Given that $\mathbf{\Lambda}$ is thus positive definite in every step, the convexity claims above hold.

A.7 PROOF OF THEOREM 2.8

Recall the theorem we intend to prove.

Let $x_1, \ldots, x_M$ be elements from some set $\mathcal{X}$ with labels $y_1, \ldots, y_M \in \{1, \ldots, L\}$ and let $w_1, \ldots, w_K \subseteq \{x_1, \ldots, x_M\}$.

Then, the expectation maximization scheme above converges to a local optimum of the loss 2.42.

Further, the sum of the GLVQ loss 2.28 with nonlinearity $\Phi(\mu) = \log(\alpha + \mu)$ and the loss 2.42 lies in $\mathcal{O}\big(2 \cdot M \cdot \log(\alpha) - \frac{1}{\alpha^2} \cdot \sum_{i=1}^M \mu_i^2\big)$.

Proof

We prove the first claim by showing that the pseudo-likelihood $\mathcal{L}$ from Equation 2.43 underestimates the loss 2.42 and becomes equivalent to the loss after each expectation step. Thus, whenever the maximization step can not find an improvement for $\mathcal{L}$ anymore, there can also be no improvement in the original loss 2.42.

In detail, we define:

$$p_i^+ = \frac{g_i^+}{(g_i^+ + g_i^-)} \quad \text{and} \quad p_i^- = \frac{g_i^-}{(g_i^+ + g_i^-)}$$

Note that it holds: $p_i^+ \geq 0$, $p_i^- \geq 0$, and $p_i^+ + p_i^- = 1$.

Further, for all i, let $\gamma_i^+, \gamma_i^- \in \mathbb{R}$ with $\gamma_i^+ \geq 0$, $\gamma_i^- \geq 0$, and $\gamma_i^+ + \gamma_i^- = 1$. Then we define the *Kullback-Leibler divergence* $\mathcal{K}(p||\gamma)$ between p and γ as

$$\mathcal{K}(p||\gamma) := \sum_{i=1}^M \gamma_i^+ \cdot \log(\frac{\gamma_i^+}{p_i^+}) + \gamma_i^- \cdot \log(\frac{\gamma_i^-}{p_i^-})$$

where we define $0 \cdot \log(\frac{x}{0}) := 0$. Note that $\mathcal{K}(p||\gamma)$ is non-negative due to Gibb's inequality. Further, $\mathcal{K}(p||\gamma) = 0$ if and only if $\gamma_i^+ = p_i^+$ and $\gamma_i^- = p_i^-$ for all i.

Now, recall the pseudo-likelihood $\mathcal{L}$ from Equation 2.43. Using $\mathcal{K}(p||\gamma)$, we can now show that:

$$\begin{aligned}
\mathcal{L} + \mathcal{K}(p||\gamma) &= \sum_{i=1}^M \gamma_i^+ \cdot \log(\frac{g_i^+}{\gamma_i^+}) + \gamma_i^- \cdot \log(\frac{g_i^-}{\gamma_i^-}) + \sum_{i=1}^M \gamma_i^+ \cdot \log(\frac{\gamma_i^+}{p_i^+}) + \gamma_i^- \cdot \log(\frac{\gamma_i^-}{p_i^-}) \\
&= \sum_{i=1}^M \gamma_i^+ \cdot \Big[\log(\frac{g_i^+}{\gamma_i^+}) + \log(\frac{\gamma_i^+}{p_i^+})\Big] + \gamma_i^- \cdot \Big[\log(\frac{g_i^-}{\gamma_i^-}) + \log(\frac{\gamma_i^-}{p_i^-})\Big] \\
&= \sum_{i=1}^M \gamma_i^+ \cdot \log(\frac{g_i^+}{p_i^+}) + \gamma_i^- \cdot \log(\frac{g_i^-}{p_i^-}) \\
&= \sum_{i=1}^M \gamma_i^+ \cdot \log(g_i^+ + g_i^-) + \gamma_i^- \cdot \log(g_i^+ + g_i^-) \\
&= \sum_{i=1}^M \log(g_i^+ + g_i^-) \cdot (\gamma_i^+ + \gamma_i^-) = \sum_{i=1}^M \log(g_i^+ + g_i^-)
\end{aligned}$$

which is exactly the loss 2.42.

Since $\mathcal{K}(p||\gamma) \geq 0$, we know that $\mathcal{L}$ always underestimates this loss. Further, in every expectation step, we set $\gamma_i^+ = p_i^+$ and $\gamma_i^- = p_i^-$ for all i such that $\mathcal{K}(p||\gamma) = 0$. Therefore, whenever $\mathcal{K}(p||\gamma) = 0$ and $\mathcal{L}$ is (locally) optimal, the loss 2.42 must also be locally optimal.

Now, regarding the second claim, we perform a Taylor expansion of the functions $\Phi(\mu) = \log(\alpha + \mu)$ and $\tilde{\Phi}(\mu) = \log(\alpha - \mu)$ around the point $\mu = 0$. Since $\alpha \geq 4$, the log function is infinitely differentiable at that point. Thus, we obtain:

$$\Phi(\mu) = \sum_{n=0}^{\infty} \frac{1}{n!} \cdot \mu^n \cdot \frac{\partial^n \Phi(0)}{\partial^n \mu} = \log(\alpha) + \sum_{n=1}^{\infty} \frac{1}{n!} \cdot \mu^n \cdot \frac{1}{\alpha^n} \cdot (-1)^{n+1}$$
$$\tilde{\Phi}(\mu) = \sum_{n=0}^{\infty} \frac{1}{n!} \cdot \mu^n \cdot \frac{\partial^n \tilde{\Phi}(0)}{\partial^n \mu} = \log(\alpha) - \sum_{n=1}^{\infty} \frac{1}{n!} \cdot \mu^n \cdot \frac{1}{\alpha^n}$$

Now, let $\mu_i = \frac{d_i^+ - d_i^-}{d_i^+ + d_i^-}$ for all i. Accordingly, the sum of the loss 2.28 and the loss 2.42 is given as:

$$\begin{aligned}
&\sum_{i=1}^{M} \log(\alpha + \mu_i) + \log(\alpha - \mu_i) \\
=&\sum_{i=1}^{M} \log(\alpha) + \sum_{n=1}^{\infty} \frac{1}{n!} \cdot \mu_i^n \cdot \frac{1}{\alpha^n} \cdot (-1)^{n+1} + \log(\alpha) - \sum_{n=1}^{\infty} \frac{1}{n!} \cdot \mu_i^n \cdot \frac{1}{\alpha^n} \\
=&\sum_{i=1}^{M} 2 \cdot \log(\alpha) - 2 \cdot \sum_{n=1}^{\infty} \frac{1}{(2n)!} \cdot \mu_i^{2n} \cdot \frac{1}{\alpha^{2n}}
\end{aligned}$$

Since $\mu_i \in [-1, 1]$ it holds that $\mu_i^2 \geq \mu_i^{2n}$ for all $n \geq 1$. Further, $\frac{1}{2} \geq \frac{1}{(2n)!}$ for all $n \geq 1$ and $\frac{1}{\alpha^2} \geq \frac{1}{\alpha^{2n}}$ for all $n \geq 1$. Therefore, the sum lies in $\mathcal{O}(2 \cdot M \cdot \log(\alpha) - \frac{1}{\alpha^2} \cdot \sum_{i=1}^{M} \mu_i^2)$ as claimed. Accordingly, by increasing α, we can ensure that local maxima of the loss 2.42 coincide with local minima of the loss 2.28.

A.8 PROOF OF THEOREM 3.1

Recall the theorem we intend to prove.

Let $\mathcal{A}$ be an alphabet with $\$, \text{match} \notin \mathcal{A}$, let $\mathcal{S} = (\text{Del}, \text{Rep}, \text{Ins})$ be a signature with $\$, \text{match} \notin \text{Del} \cup \text{Rep} \cup \text{Ins}$, and let $\mathcal{F}$ be an algebra over $\mathcal{S}$ and $\mathcal{A}$. Finally, let $\tilde{\delta} \in \mathcal{T}(\mathcal{S}, \mathcal{A})$ be a script tree and let $(\bar{x}, \bar{y}) := \mathcal{Y}(\tilde{\delta})$ be the yield of $\tilde{\delta}$. Then there exists an edit script $\bar{\delta}_{\tilde{\delta}} \in \Delta_{\mathcal{S},\mathcal{A}}$ such that $\bar{y} = \bar{\delta}_{\tilde{\delta}}(\bar{x})$ and $c_{\mathcal{F}}(\tilde{\delta}) = c_{\mathcal{F}}(\bar{\delta}_{\tilde{\delta}}, \bar{x})$.

Now, let $\mathcal{F}$ conform to the following conditions.

$$\forall \text{rep} \in \text{Rep} : \forall x, y \in \mathcal{A} : c_{\text{rep}}(x, y) \geq 0 \tag{A.48}$$
$$\forall \text{del} \in \text{Del} : \forall x \in \mathcal{A} : c_{\text{del}}(x) \geq 0 \tag{A.49}$$
$$\forall \text{ins} \in \text{Ins} : \forall y \in \mathcal{A} : c_{\text{ins}}(y) \geq 0 \tag{A.50}$$
$$\forall \text{rep}, \text{rep}' \in \text{Rep} : \forall x, y, z \in \mathcal{A} : c_{\text{rep}'}(x, y) + c_{\text{rep}}(y, z) \geq c_{\text{rep}}(x, y) \tag{A.51}$$
$$\forall \text{rep} \in \text{Rep} : \forall \text{ins} \in \text{Ins} : \forall x, y \in \mathcal{A} : c_{\text{ins}}(x) + c_{\text{rep}}(x, y) \geq c_{\text{ins}}(y) \tag{A.52}$$
$$\forall \text{del} \in \text{Del} : \forall \text{rep} \in \text{Rep} : \forall x, y \in \mathcal{A} : c_{\text{rep}}(x, y) + c_{\text{del}}(y) \geq c_{\text{del}}(x) \tag{A.53}$$

Then, for all edit scripts $\bar{\delta} \in \Delta_{\mathcal{S},\mathcal{A}}$ and all $\bar{x} \in \mathcal{A}^*$, there exists a script tree $\tilde{\delta}_{\bar{\delta},\bar{x}}$ such that $\mathcal{Y}(\tilde{\delta}_{\bar{\delta},\bar{x}}) = (\bar{x}, \bar{\delta}(\bar{x}))$ and $c_{\mathcal{F}}(\tilde{\delta}_{\bar{\delta},\bar{x}}) \leq c_{\mathcal{F}}(\bar{\delta}, \bar{x})$.

Further, it holds for all sequences $\bar{x}, \bar{y} \in \mathcal{A}^*$:

$$d_{\mathcal{S},\mathcal{F}}(\bar{x}, \bar{y}) = \min_{\tilde{\delta} \in \mathcal{T}(\mathcal{S},\mathcal{A})} \{c_{\mathcal{F}}(\tilde{\delta}) | \mathcal{Y}(\tilde{\delta}) = (\bar{x}, \bar{y})\} \tag{A.54}$$

Proof

We prove the first claim via induction over the size of the script tree $\tilde{\delta}$. First, let $\tilde{\delta} = \$$. In that case, $\mathcal{Y}(\tilde{\delta}) = (\epsilon, \epsilon)$. Therefore, the edit script $\bar{\delta}_{\tilde{\delta}} = \epsilon$ fulfills the conditions $\bar{\delta}_{\tilde{\delta}}(\epsilon) = \epsilon$ and $c_{\mathcal{F}}(\bar{\delta}_{\tilde{\delta}}, \epsilon) = 0 = c_{\mathcal{F}}(\tilde{\delta})$.

Now, consider a script tree $\tilde{\delta}$ with a size larger than 0 and distinguish the following cases:

$\tilde{\delta} = \text{match}(x, \tilde{\delta}', x)$ for some $x \in \mathcal{A}$ and some script tree $\tilde{\delta}'$. Let $(\bar{x}, \bar{y}) := \mathcal{Y}(\tilde{\delta}')$. Then it holds: $\mathcal{Y}(\tilde{\delta}) = (x\bar{x}, x\bar{y})$. Further, per induction, there exists an edit script $\bar{\delta}_{\tilde{\delta}'}$ such that $\bar{\delta}_{\tilde{\delta}'}(\bar{x}) = \bar{y}$ and $c_{\mathcal{F}}(\bar{\delta}_{\tilde{\delta}'}, \bar{x}) = c_{\mathcal{F}}(\tilde{\delta}')$. Now, consider the edit script $\bar{\delta}_{\tilde{\delta}}$ which contains the same sequence edits as $\bar{\delta}_{\tilde{\delta}'}$, but with all indices being incremented by one. Then, it holds: $\bar{\delta}_{\tilde{\delta}}(x\bar{x}) = x\bar{y}$ and $c_{\mathcal{F}}(\bar{\delta}_{\tilde{\delta}}, x\bar{x}) = c_{\mathcal{F}}(\bar{\delta}_{\tilde{\delta}'}, \bar{x}) = c_{\mathcal{F}}(\tilde{\delta}') = c_{\mathcal{F}}(\tilde{\delta})$.

$\tilde{\delta} = \text{del}(x, \tilde{\delta}')$ for some $\text{del} \in \text{Del}$, some $x \in \mathcal{A}$, and some script tree $\tilde{\delta}'$. Let $(\bar{x}, \bar{y}) := \mathcal{Y}(\tilde{\delta}')$. Then it holds: $\mathcal{Y}(\tilde{\delta}) = (x\bar{x}, \bar{y})$. Further, per induction, there exists an edit script $\bar{\delta}_{\tilde{\delta}'}$ such that $\bar{\delta}_{\tilde{\delta}'}(\bar{x}) = \bar{y}$ and $c_{\mathcal{F}}(\bar{\delta}_{\tilde{\delta}'}, \bar{x}) = c_{\mathcal{F}}(\tilde{\delta}')$. Now, consider the edit script $\bar{\delta}_{\tilde{\delta}} := \text{del}_1 \bar{\delta}_{\tilde{\delta}'}$. Then, it holds: $\bar{\delta}_{\tilde{\delta}}(x\bar{x}) = \bar{\delta}_{\tilde{\delta}'}(\text{del}_1(x\bar{x})) = \bar{\delta}_{\tilde{\delta}'}(\bar{x}) = \bar{y}$, and $c_{\mathcal{F}}(\bar{\delta}_{\tilde{\delta}}, x\bar{x}) = c_{\text{del}}(x) + c_{\mathcal{F}}(\bar{\delta}_{\tilde{\delta}'}, \bar{x}) = c_{\text{del}}(x) + c_{\mathcal{F}}(\tilde{\delta}') = c_{\mathcal{F}}(\tilde{\delta})$.

$\tilde{\delta} = \text{rep}(x, \tilde{\delta}', y)$ for some $\text{rep} \in \text{Rep}$, some $x, y \in \mathcal{A}$, and some script tree $\tilde{\delta}'$. Let $(\bar{x}, \bar{y}) := \mathcal{Y}(\tilde{\delta}')$. Then it holds: $\mathcal{Y}(\tilde{\delta}) = (x\bar{x}, y\bar{y})$. Further, per induction, there exists an edit script $\bar{\delta}_{\tilde{\delta}'}$ such that $\bar{\delta}_{\tilde{\delta}'}(\bar{x}) = \bar{y}$ and $c_{\mathcal{F}}(\bar{\delta}_{\tilde{\delta}'}, \bar{x}) = c_{\mathcal{F}}(\tilde{\delta}')$. Now, consider the edit script $\bar{\delta}_{\tilde{\delta}} := \text{rep}_{1,y} \bar{\delta}'$ where $\bar{\delta}'$ contains the same sequence edits as $\bar{\delta}_{\tilde{\delta}'}$, but with all indices being incremented by one. Then, it holds: $\bar{\delta}_{\tilde{\delta}}(x\bar{x}) = \bar{\delta}'(\text{rep}_{1,y}(x\bar{x})) = \bar{\delta}'(y\bar{x}) = y\bar{\delta}_{\tilde{\delta}'}(\bar{x}) = y\bar{y}$ and $c_{\mathcal{F}}(\bar{\delta}_{\tilde{\delta}}, x\bar{x}) = c_{\text{rep}}(x,y) + c_{\mathcal{F}}(\bar{\delta}', y\bar{x}) = c_{\text{rep}}(x,y) + c_{\mathcal{F}}(\bar{\delta}_{\tilde{\delta}'}, \bar{x}) = c_{\text{rep}}(x,y) + c_{\mathcal{F}}(\tilde{\delta}') = c_{\mathcal{F}}(\tilde{\delta})$.

$\tilde{\delta} = \text{ins}(\tilde{\delta}', y)$ for some ins $\in$ Ins, some $y \in \mathcal{A}$, and some script tree $\tilde{\delta}'$. Let $(\bar{x}, \bar{y}) := \mathcal{Y}(\tilde{\delta}')$. Then it holds: $\mathcal{Y}(\tilde{\delta}) = (\bar{x}, y\bar{y})$. Further, per induction, there exists an edit script $\bar{\delta}_{\tilde{\delta}'}$ such that $\bar{\delta}_{\tilde{\delta}'}(\bar{x}) = \bar{y}$ and $c_{\mathcal{F}}(\bar{\delta}_{\tilde{\delta}'}, \bar{x}) = c_{\mathcal{F}}(\tilde{\delta}')$. Now, consider the edit script $\bar{\delta}_{\tilde{\delta}} := \text{ins}_{1,y}\bar{\delta}'$ where $\bar{\delta}'$ contains the same sequence edits as $\bar{\delta}_{\tilde{\delta}'}$, but with all indices being incremented by one. Then, it holds: $\bar{\delta}_{\tilde{\delta}}(\bar{x}) = \bar{\delta}'(\text{ins}_{1,y}(\bar{x})) = \bar{\delta}'(y\bar{x}) = y\bar{\delta}_{\tilde{\delta}'}(\bar{x}) = y\bar{y}$ and $c_{\mathcal{F}}(\bar{\delta}_{\tilde{\delta}}, \bar{x}) = c_{\text{ins}}(y) + c_{\mathcal{F}}(\bar{\delta}', y\bar{x}) = c_{\text{ins}}(y) + c_{\mathcal{F}}(\bar{\delta}_{\tilde{\delta}'}, \bar{x}) = c_{\text{ins}}(y) + c_{\mathcal{F}}(\tilde{\delta}') = c_{\mathcal{F}}(\tilde{\delta})$.

This concludes the proof of the first claim.

Regarding the second claim, we first need to introduce an auxiliary concept. Let $\tilde{\delta}$ be a script tree over $\mathcal{S}$ and $\mathcal{A}$. Then, we define the jth right-subtree $\tilde{\delta}_j$ of $\tilde{\delta}$ as follows.

$$\tilde{\delta}_j := \begin{cases} \$ & \text{if } \tilde{\delta} = \$ \\ \tilde{\delta}'_j & \text{if } \tilde{\delta} = \text{del}(x, \tilde{\delta}') \\ \tilde{\delta} & \text{if } j = 1 \quad \text{and } \tilde{\delta} = \text{match}(x, \tilde{\delta}', x) \text{ or } \tilde{\delta} = \text{rep}(x, \tilde{\delta}', y) \text{ or } \tilde{\delta} = \text{ins}(\tilde{\delta}', y) \\ \tilde{\delta}'_{j-1} & \text{if } j > 1 \quad \text{and } \tilde{\delta} = \text{match}(x, \tilde{\delta}', x) \text{ or } \tilde{\delta} = \text{rep}(x, \tilde{\delta}', y) \text{ or } \tilde{\delta} = \text{ins}(\tilde{\delta}', y) \end{cases}$$

where rep $\in$ Rep, del $\in$ Del, ins $\in$ Ins, $x, y \in \mathcal{A}$ and $\tilde{\delta}'$ is a script tree over $\mathcal{S}$ and $\mathcal{A}$.

Let $\mathcal{Y}(\tilde{\delta}) = (\bar{x}, y_1 \ldots y_n)$. It is simple to show via induction that for all $j \in \mathbb{N}$ it holds: If $j > n$, then $\tilde{\delta}_j = \$$. Otherwise, $\mathcal{Y}(\tilde{\delta}_j) = (\bar{x}', y_j \ldots y_n)$, where $\bar{x}'$ is some suffix of $\bar{x}$. Further, it holds that $\tilde{\delta}_j = \text{match}(x, \tilde{\delta}', y_j)$, or $\tilde{\delta}_j = \text{rep}(x, \tilde{\delta}', y_j)$, or $\tilde{\delta}_j = \text{ins}(\tilde{\delta}', y_j)$ for some x in $\bar{x}$, some rep $\in$ Rep, some ins $\in$ Ins, and some script tree $\tilde{\delta}'$.

Given this concept, we can now show the actual claim via induction over the length of $\bar{\delta}$. First, consider $\bar{\delta} = \epsilon$ and some sequence $\bar{x} = x_1 \ldots x_m$. In that case, the script tree $\tilde{\delta}_{\bar{\delta},\bar{x}} := \text{match}(x_1, \text{match}(\ldots \text{match}(x_m, \$, x_m) \ldots), x_1)$ fulfills the conditions $\mathcal{Y}(\tilde{\delta}_{\bar{\delta},\bar{x}}) = (\bar{x}, \bar{x}) = (\bar{x}, \bar{\delta}(\bar{x}))$ and $c_{\mathcal{F}}(\tilde{\delta}_{\bar{\delta},\bar{x}}) = 0 = c(\bar{\delta}, \bar{x})$.

Now, let $\bar{\delta} = \delta_1 \ldots \delta_{T+1}$ be a nonempty edit script, and let $\bar{x} = x_1 \ldots x_m$ some sequence over $\mathcal{A}$. Then, consider the script $\bar{\delta}' = \delta_1 \ldots \delta_T$, and let $\bar{y} = y_1 \ldots y_n := \bar{\delta}'(\bar{x})$. Per induction, there exists a script tree $\tilde{\delta}' := \tilde{\delta}_{\bar{\delta}',\bar{x}}$ such that $\mathcal{Y}(\tilde{\delta}') = (\bar{x}, \bar{y})$ and $c_{\mathcal{F}}(\tilde{\delta}') \leq c_{\mathcal{F}}(\bar{\delta}', \bar{x})$. It remains to show that there exists a script tree $\tilde{\delta}_{\bar{\delta},\bar{x}}$ such that $\mathcal{Y}(\tilde{\delta}_{\bar{\delta},\bar{x}}) = (\bar{x}, \bar{\delta}(\bar{x}))$ and $c_{\mathcal{F}}(\tilde{\delta}_{\bar{\delta},\bar{x}}) \leq c_{\mathcal{F}}(\bar{\delta}, \bar{x})$.

Consider the last sequence edit δ_{T+1}. We distinguish the following cases.

$\delta_{T+1} = \text{del}_j$ for some del $\in$ Del, and some $j \in \mathbb{N}$. If $j > n$, $\bar{\delta}(\bar{x}) = \text{del}_j(\bar{y}) = \bar{y}$. Thus, we can define $\tilde{\delta}_{\bar{\delta},\bar{x}} := \tilde{\delta}'$ and obtain $\mathcal{Y}(\tilde{\delta}_{\bar{\delta},\bar{x}}) = (\bar{x}, \bar{y}) = (\bar{x}, \bar{\delta}(\bar{x}))$, as well as $c_{\mathcal{F}}(\tilde{\delta}_{\bar{\delta},\bar{x}}) \leq c_{\mathcal{F}}(\bar{\delta}', \bar{x}) = c_{\mathcal{F}}(\text{del}_j, \bar{y}) + c_{\mathcal{F}}(\bar{\delta}', \bar{x}) = c_{\mathcal{F}}(\bar{\delta}, \bar{x})$, which proves the claim.

If $j \leq n$, consider $\tilde{\delta}'_j$ and distinguish the following cases.

$\tilde{\delta}'_j = \text{match}(y_j, \tilde{\delta}'', y_j)$ for some script tree $\tilde{\delta}''$. In that case, we define $\tilde{\delta}_{\bar{\delta},\bar{x}}$ to be the same as $\tilde{\delta}'$, but with the subtree $\tilde{\delta}'_j$ replaced by the subtree $\text{del}(y_j, \tilde{\delta}'')$. Accordingly, we obtain $\mathcal{Y}(\tilde{\delta}_{\bar{\delta},\bar{x}}) = (\bar{x}, y_1 \ldots y_{j-1} y_{j+1} \ldots y_n) = (\bar{x}, \text{del}_j(\bar{y})) = (\bar{x}, \bar{\delta}(\bar{x}))$, as well as $c_{\mathcal{F}}(\tilde{\delta}_{\bar{\delta},\bar{x}}) = c_{\mathcal{F}}(\tilde{\delta}') + c_{\text{del}}(y_j) \overset{Ind.}{\leq} c_{\mathcal{F}}(\bar{\delta}', \bar{x}) + c_{\text{del}}(y_j) = c_{\mathcal{F}}(\bar{\delta}, \bar{x})$.

$\tilde{\delta}'_j = \text{rep}(x, \tilde{\delta}'', y_j)$ for some rep $\in$ Rep, some $x \in \mathcal{A}$, and some script tree $\tilde{\delta}''$. In that case, we define $\tilde{\delta}_{\bar{\delta},\bar{x}}$ to be the same as $\tilde{\delta}'$, but with the subtree $\tilde{\delta}'_j$ replaced by the subtree $\text{del}(x, \tilde{\delta}'')$. Accordingly, we obtain $\mathcal{Y}(\tilde{\delta}_{\bar{\delta},\bar{x}}) = (\bar{x}, y_1 \ldots y_{j-1} y_{j+1} \ldots y_n) = (\bar{x}, \text{del}_j(\bar{y})) = (\bar{x}, \bar{\delta}(\bar{x}))$, as well as $c_{\mathcal{F}}(\tilde{\delta}_{\bar{\delta},\bar{x}}) = c_{\mathcal{F}}(\tilde{\delta}') + c_{\text{del}}(x) - c_{\text{rep}}(x, y_j) \overset{A.53}{\leq} c_{\mathcal{F}}(\tilde{\delta}') + c_{\text{del}}(y_j) \overset{Ind.}{\leq} c_{\mathcal{F}}(\bar{\delta}', \bar{x}) + c_{\text{del}}(y_j) = c_{\mathcal{F}}(\bar{\delta}, \bar{x})$.

$\tilde{\delta}'_j = \text{ins}(\tilde{\delta}'', y_j)$ for some ins $\in$ Ins and some script tree $\tilde{\delta}''$. In that case, we define $\tilde{\delta}_{\bar{\delta},\bar{x}}$ to be the same as $\tilde{\delta}'$, but with the subtree $\tilde{\delta}'_j$ replaced by the subtree $\tilde{\delta}''$. Accordingly, we obtain $\mathcal{Y}(\tilde{\delta}_{\bar{\delta},\bar{x}}) = (\bar{x}, y_1 \ldots y_{j-1} y_{j+1} \ldots y_n) = (\bar{x}, \text{del}_j(\bar{y})) = (\bar{x}, \bar{\delta}(\bar{x}))$, as well as $c_{\mathcal{F}}(\tilde{\delta}_{\bar{\delta},\bar{x}}) = c_{\mathcal{F}}(\tilde{\delta}') - c_{\text{ins}}(y_j) \overset{A.49, A.50}{\leq} c_{\mathcal{F}}(\tilde{\delta}') + c_{\text{del}}(y_j) \overset{Ind.}{\leq} c_{\mathcal{F}}(\bar{\delta}', \bar{x}) + c_{\text{del}}(y_j) = c_{\mathcal{F}}(\bar{\delta}, \bar{x})$.

$\delta_{T+1} = \text{rep}_{j,y}$ for some rep $\in$ Rep, some $j \in \mathbb{N}$, and some $y \in \mathcal{A}$. If $j > n$, $\bar{\delta}(\bar{x}) = \text{rep}_{j,y}(\bar{y}) = \bar{y}$. Thus, we can define $\tilde{\delta}_{\bar{\delta},\bar{x}} := \tilde{\delta}'$ and obtain $\mathcal{Y}(\tilde{\delta}_{\bar{\delta},\bar{x}}) = (\bar{x}, \bar{y}) = (\bar{x}, \bar{\delta}(\bar{x}))$, as well as $c_{\mathcal{F}}(\tilde{\delta}_{\bar{\delta},\bar{x}}) \leq c_{\mathcal{F}}(\bar{\delta}', \bar{x}) = c_{\mathcal{F}}(\text{rep}_{j,y}, \bar{y}) + c_{\mathcal{F}}(\bar{\delta}', \bar{x}) = c_{\mathcal{F}}(\bar{\delta}, \bar{x})$, which proves the claim.

If $j \leq n$, consider $\tilde{\delta}'_j$ and distinguish the following cases.

$\tilde{\delta}'_j = \text{match}(y_j, \tilde{\delta}'', y_j)$ for some script tree $\tilde{\delta}''$. In that case, we define $\tilde{\delta}_{\bar{\delta},\bar{x}}$ to be the same as $\tilde{\delta}'$, but with the subtree $\tilde{\delta}'_j$ replaced by the subtree $\text{rep}(y_j, \tilde{\delta}'', y)$. Accordingly, we obtain $\mathcal{Y}(\tilde{\delta}_{\bar{\delta},\bar{x}}) = (\bar{x}, y_1 \ldots y_{j-1} y y_{j+1} \ldots y_n) = (\bar{x}, \text{rep}_{j,y}(\bar{y})) = (\bar{x}, \bar{\delta}(\bar{x}))$, as well as $c_{\mathcal{F}}(\tilde{\delta}_{\bar{\delta},\bar{x}}) = c_{\mathcal{F}}(\tilde{\delta}') + c_{\text{rep}}(y_j, y) \overset{Ind.}{\leq} c_{\mathcal{F}}(\bar{\delta}', \bar{x}) + c_{\text{rep}}(y_j, y) = c_{\mathcal{F}}(\bar{\delta}, \bar{x})$.

$\tilde{\delta}'_j = \text{rep}'(x, \tilde{\delta}'', y_j)$ for some rep$'$ $\in$ Rep, some $x \in \mathcal{A}$, and some script tree $\tilde{\delta}''$. In that case, we define $\tilde{\delta}_{\bar{\delta},\bar{x}}$ to be the same as $\tilde{\delta}'$, but with the subtree $\tilde{\delta}'_j$ replaced by the subtree $\text{rep}(x, \tilde{\delta}'', y)$. Accordingly, we obtain $\mathcal{Y}(\tilde{\delta}_{\bar{\delta},\bar{x}}) = (\bar{x}, y_1 \ldots y_{j-1} y y_{j+1} \ldots y_n) = (\bar{x}, \text{rep}_{j,y}(\bar{y})) = (\bar{x}, \bar{\delta}(\bar{x}))$, as well as $c_{\mathcal{F}}(\tilde{\delta}_{\bar{\delta},\bar{x}}) = c_{\mathcal{F}}(\tilde{\delta}') - c_{\text{rep}'}(x, y_j) + c_{\text{rep}}(x, y) \overset{A.51}{\leq} c_{\mathcal{F}}(\tilde{\delta}') + c_{\text{rep}}(y_j, y) \overset{Ind.}{\leq} c_{\mathcal{F}}(\bar{\delta}', \bar{x}) + c_{\text{rep}}(y_j, y) = c_{\mathcal{F}}(\bar{\delta}, \bar{x})$.

$\tilde{\delta}'_j = \text{ins}(\tilde{\delta}'', y_j)$ for some ins $\in$ Ins and some script tree $\tilde{\delta}''$. In that case, we define $\tilde{\delta}_{\bar{\delta},\bar{x}}$ to be the same as $\tilde{\delta}'$, but with the subtree $\tilde{\delta}'_j$ replaced by the subtree $\text{ins}(\tilde{\delta}'', y)$. Accordingly, we obtain $\mathcal{Y}(\tilde{\delta}_{\bar{\delta},\bar{x}}) = (\bar{x}, y_1 \ldots y_{j-1} y y_{j+1} \ldots y_n) = (\bar{x}, \text{rep}_{j,y}(\bar{y})) = (\bar{x}, \bar{\delta}(\bar{x}))$, as well as $c_{\mathcal{F}}(\tilde{\delta}_{\bar{\delta},\bar{x}}) = c_{\mathcal{F}}(\tilde{\delta}') - c_{\text{ins}}(y_j) + c_{\text{ins}}(y) \overset{A.52}{\leq} c_{\mathcal{F}}(\tilde{\delta}') + c_{\text{rep}}(y_j, y) \overset{Ind.}{\leq} c_{\mathcal{F}}(\bar{\delta}', \bar{x}) + c_{\text{rep}}(y_j, y) = c_{\mathcal{F}}(\bar{\delta}, \bar{x})$.

$\delta_{T+1} = \text{ins}_{j,y}$ for some ins $\in$ Ins, some $j \in \mathbb{N}$, and some $y \in \mathcal{A}$. If $j > n+1$, $\bar{\delta}(\bar{x}) = \text{ins}_{j,y}(\bar{y}) = \bar{y}$. Thus, we can define $\tilde{\delta}_{\bar{\delta},\bar{x}} := \tilde{\delta}'$ and obtain $\mathcal{Y}(\tilde{\delta}_{\bar{\delta},\bar{x}}) = (\bar{x}, \bar{y}) = (\bar{x}, \bar{\delta}(\bar{x}))$, as well as $c_{\mathcal{F}}(\tilde{\delta}_{\bar{\delta},\bar{x}}) \leq c_{\mathcal{F}}(\bar{\delta}', \bar{x}) = c_{\mathcal{F}}(\text{ins}_{j,y}, \bar{y}) + c_{\mathcal{F}}(\bar{\delta}', \bar{x}) = c_{\mathcal{F}}(\bar{\delta}, \bar{x})$, which proves the claim.

If $j \leq n+1$, we define $\tilde{\delta}_{\bar{\delta},\bar{x}}$ to be the same as $\tilde{\delta}'$, but with the subtree $\tilde{\delta}'_j$ being replaced by the subtree $\text{ins}(\tilde{\delta}'_j, y)$. Accordingly, we obtain $\mathcal{Y}(\tilde{\delta}_{\bar{\delta},\bar{x}}) = (\bar{x}, y_1 \ldots y_{j-1} y y_j \ldots y_n) = (\bar{x}, \text{ins}_{j,y}(\bar{y})) = (\bar{x}, \bar{\delta}(\bar{x}))$, as well as $c_{\mathcal{F}}(\tilde{\delta}_{\bar{\delta},\bar{x}}) = c_{\mathcal{F}}(\tilde{\delta}') + c_{\text{ins}}(y) \overset{Ind.}{\leq} c_{\mathcal{F}}(\bar{\delta}', \bar{x}) + c_{\text{ins}}(y) = c_{\mathcal{F}}(\bar{\delta}, \bar{x})$.

This covers all possible cases such that we can always extend the script tree $\tilde{\delta}'$ corresponding to the edit script $\bar{\delta}'$ and the sequence $\bar{x}$ to a script tree $\tilde{\delta}_{\bar{\delta},\bar{x}}$ corresponding to the edit script $\bar{\delta}$ and the sequence $\bar{x}$, which concludes the induction.

Regarding the third claim, let $d^*_{\mathcal{S},\mathcal{F}}(\bar{x},\bar{y}) := \min_{\tilde{\delta}\in\mathcal{T}(\mathcal{S},\mathcal{A})}\{c_\mathcal{F}(\tilde{\delta})|\mathcal{Y}(\tilde{\delta}) = (\bar{x},\bar{y})\}$.

Now, assume that the claim does not hold for two sequences $\bar{x},\bar{y} \in \mathcal{A}^*$, that is, $d_{\mathcal{S},\mathcal{F}}(\bar{x},\bar{y}) \neq d^*_{\mathcal{S},\mathcal{F}}(\bar{x},\bar{y})$. Then, either $d_{\mathcal{S},\mathcal{F}}(\bar{x},\bar{y}) < d^*_{\mathcal{S},\mathcal{F}}(\bar{x},\bar{y})$ or $d_{\mathcal{S},\mathcal{F}}(\bar{x},\bar{y}) > d^*_{\mathcal{S},\mathcal{F}}(\bar{x},\bar{y})$.

In the first case, let $\bar{\delta}$ be an edit script in $\Delta_{\mathcal{S},\mathcal{F}}$, such that $\bar{\delta}(\bar{x}) = \bar{y}$ and $c_\mathcal{F}(\bar{\delta},\bar{x}) < d^*_{\mathcal{S},\mathcal{F}}(\bar{x},\bar{y})$. According to the second claim, there exists a script tree $\tilde{\delta}_{\bar{\delta},\bar{x}}$ such that $\mathcal{Y}(\tilde{\delta}_{\bar{\delta},\bar{x}}) = (\bar{x},\bar{y})$ and $c_\mathcal{F}(\tilde{\delta}_{\bar{\delta},\bar{x}}) \leq c_\mathcal{F}(\bar{\delta},\bar{x})$. This, however, implies that $d^*_{\mathcal{S},\mathcal{F}}(\bar{x},\bar{y}) \leq c_\mathcal{F}(\tilde{\delta}_{\bar{\delta},\bar{x}}) \leq c_\mathcal{F}(\bar{\delta},\bar{x}) < d^*_{\mathcal{S},\mathcal{F}}(\bar{x},\bar{y})$, which is a contradiction.

In the second case, let $\tilde{\delta}$ be a script tree in $\mathcal{T}(\mathcal{S},\mathcal{A})$, such that $\mathcal{Y}(\tilde{\delta}) = (\bar{x},\bar{y})$ and $c_\mathcal{F}(\tilde{\delta}) < d_{\mathcal{S},\mathcal{F}}(\bar{x},\bar{y})$. According to the first claim, there exists an edit script $\bar{\delta}_{\tilde{\delta}}$ such that $\bar{\delta}_{\tilde{\delta}}(\bar{x}) = \bar{y}$ and $c_\mathcal{F}(\bar{\delta}_{\tilde{\delta}},\bar{x}) = c_\mathcal{F}(\tilde{\delta})$. This, however, implies that $d_{\mathcal{S},\mathcal{F}}(\bar{x},\bar{y}) \leq c_\mathcal{F}(\bar{\delta}_{\tilde{\delta}},\bar{x}) = c_\mathcal{F}(\tilde{\delta}) < d_{\mathcal{S},\mathcal{F}}(\bar{x},\bar{y})$, which is a contradiction.

Therefore, $d_{\mathcal{S},\mathcal{F}}(\bar{x},\bar{y}) = d^*_{\mathcal{S},\mathcal{F}}(\bar{x},\bar{y})$, which concludes the proof.

A.9 PROOF OF THEOREM 3.2

Recall the theorem we intend to prove.

Let $\mathcal{A}$ be an alphabet, let $\mathcal{S} = (\text{Del}, \text{Rep}, \text{Ins})$ be a non-trivial signature, and let $\mathcal{F}$ be an algebra over $\mathcal{S}$ and $\mathcal{A}$. Further, let $\Delta_{\mathcal{S},\mathcal{A}}$ be the edit set with respect to $\mathcal{S}$ and $\mathcal{A}$, and let $c_\mathcal{F}$ be the cost function with respect to $\mathcal{F}$ such that the following conditions hold.

$$\forall \text{del} \in \text{Del} : \forall x \in \mathcal{A} : c_{\text{del}}(x) \geq 0$$
$$\forall \text{ins} \in \text{Ins} : \forall y \in \mathcal{A} : c_{\text{ins}}(y) \geq 0$$
$$\forall \text{del} \in \text{Del} : \exists \text{ins} \in \text{Ins} : \forall x \in \mathcal{A} : c_{\text{del}}(x) = c_{\text{ins}}(x)$$
$$\forall \text{ins} \in \text{Ins} : \exists \text{del} \in \text{Del} : \forall y \in \mathcal{A} : c_{\text{ins}}(y) = c_{\text{del}}(y)$$
$$\forall \text{rep} \in \text{Rep} : \forall x,y \in \mathcal{A} : c_{\text{rep}}(x,y) = c_{\text{rep}}(y,x) \geq 0$$

Then, the edit distance $d_{\mathcal{S},\mathcal{F}}$ is a pseudo-metric over $\mathcal{A}^*$.

Proof

Let $\bar{x},\bar{y},\bar{z} \in \mathcal{A}^*$, and let $\bar{x} = x_1 \ldots x_m$, as well as $\bar{y} = y_1 \ldots y_n$. Then, we can show the following properties of $d_{\mathcal{S},\mathcal{F}}$.

Well-Definedness: Because $\mathcal{S}$ is non-trivial, both Del and Ins are non-empty. Let del $\in$ Del and ins $\in$ Ins. Then, $\bar{\delta} := \text{del}_m \ldots \text{del}_1 \text{ins}_{1,y_1} \ldots \text{ins}_{n,y_n}$ is a edit script in $\Delta^*_{\mathcal{S},\mathcal{A}}$ and it holds $\bar{\delta}(\bar{x}) = \bar{y}$. Therefore, $d_{\mathcal{S},\mathcal{F}}(\bar{x},\bar{y})$ is well-defined, and is bounded from above by $c_\mathcal{F}(\bar{\delta},\bar{x})$.

Non-Negativity: Let $\bar{\delta}$ be an edit script in $\Delta^*_{\mathcal{S},\mathcal{A}}$ such that $\bar{\delta}(\bar{x}) = \bar{y}$ and $c_\mathcal{F}(\bar{\delta},\bar{x}) = d_{\mathcal{S},\mathcal{F}}(\bar{x},\bar{y})$. As shown above, such an edit script exists. Further, $c_\mathcal{F}(\bar{\delta},\bar{x})$ is a sum over non-negative contributions according to our conditions, and is thus non-negative itself. Therefore, $d_{\mathcal{S},\mathcal{F}}(\bar{x},\bar{y})$ is non-negative.

Self-Identity: The empty edit script ϵ transforms $\bar{x}$ to itself, and has per definition a cost of 0. Therefore, $d_{\mathcal{S},\mathcal{F}}(\bar{x},\bar{x}) \leq 0$. Further, $d_{\mathcal{S},\mathcal{F}}(\bar{x},\bar{x})$ is non-negative, therefore $d_{\mathcal{S},\mathcal{F}}(\bar{x},\bar{x}) = 0$.

Symmetry: We prove a more general auxiliary claim, from which the symmetry of $d_{\mathcal{S},\mathcal{F}}$ follows.

Let $\bar{\delta}$ be an edit script in $\Delta^*_{\mathcal{S},\mathcal{A}}$ such that $\bar{\delta}(\bar{x}) = \bar{y}$. Then, there exists an edit script $\bar{\delta}^{-1}$ in $\Delta^*_{\mathcal{S},\mathcal{A}}$ such that $\bar{\delta}^{-1}(\bar{y}) = \bar{x}$, and $c_{\mathcal{F}}(\bar{\delta}^{-1},\bar{y}) = c_{\mathcal{F}}(\bar{\delta},\bar{x})$.

We prove this claim via induction over the length of $\bar{\delta}$. First, consider $\bar{\delta} = \epsilon$. Then, we define $\bar{\delta}^{-1} = \epsilon$, and we obtain $\bar{\delta}^{-1}(\bar{y}) = \bar{x}$, as well as $c_{\mathcal{F}}(\bar{\delta}^{-1},\bar{y}) = 0 = c_{\mathcal{F}}(\bar{\delta},\bar{x})$. Now, assume that the claim holds for all edit scripts $\bar{\delta}$ up to length T. Finally, consider an edit script $\bar{\bar{\delta}} = \delta_1 \ldots \delta_{T+1}$.

Now, consider the sequence edit δ_1 and distinguish the following cases.

$\delta_1 = \text{del}_i$ for some del $\in$ Del, and some $i \in \mathbb{N}$. Then, there exists some ins $\in$ Ins such that $c_{\text{ins}}(x_i) = c_{\text{del}}(x_i)$. We now define $\delta_1^{-1} = \text{ins}_{i,x_i}$, such that $\delta_1^{-1}(\delta_1(\bar{x})) = \bar{x}$, and $c_{\mathcal{F}}(\delta_1^{-1},\delta_1(\bar{x})) = c_{\text{ins}}(x_i) = c_{\text{del}}(x_i) = c_{\mathcal{F}}(\delta_1,\bar{x})$.

$\delta_1 = \text{rep}_{i,y}$ for some rep $\in$ Rep, some $i \in \mathbb{N}$, and some $y \in \mathcal{A}$. Because c_{rep} is symmetric, we know that $c_{\text{rep}}(x_i,y) = c_{\text{rep}}(y,x_i)$. We now define $\delta_1^{-1} = \text{rep}_{i,x_i}$, such that $\delta_1^{-1}(\delta_1(\bar{x})) = \bar{x}$, and $c_{\mathcal{F}}(\delta_1^{-1},\delta_1(\bar{x})) = c_{\text{rep}}(y,x_i) = c_{\text{rep}}(x_i,y) = c_{\mathcal{F}}(\delta_1,\bar{x})$.

$\delta_1 = \text{ins}_{i,y}$ for some ins $\in$ Ins, some $i \in \mathbb{N}$, and some $y \in \mathcal{A}$. Then, there exists some del $\in$ Del such that $c_{\text{del}}(y) = c_{\text{ins}}(y)$. We now define $\delta_1^{-1} = \text{del}_i$, such that $\delta_1^{-1}(\delta_1(\bar{x})) = \bar{x}$, and $c_{\mathcal{F}}(\delta_1^{-1},\delta_1(\bar{x})) = c_{\text{del}}(y) = c_{\text{ins}}(y) = c_{\mathcal{F}}(\delta_1,\bar{x})$.

These three cases cover all possibilities for δ_1.

Now, consider the edit script $\bar{\delta} = \delta_2 \ldots \delta_{T+1}$. Since $|\bar{\delta}| \leq T$, we obtain via induction an edit script $\bar{\delta}^{-1}$ such that $\bar{\delta}^{-1}(\bar{y}) = \delta_1(\bar{x})$, and $c_{\mathcal{F}}(\bar{\delta}^{-1},\bar{y}) = c_{\mathcal{F}}(\bar{\delta},\delta_1(\bar{x}))$. Accordingly, we define the script $\bar{\bar{\delta}}^{-1} := \bar{\delta}^{-1}\delta_1^{-1}$, for which we obtain $\bar{\bar{\delta}}^{-1}(\bar{y}) = \delta_1^{-1}(\delta_1(\bar{x})) = \bar{x}$, and $c_{\mathcal{F}}(\bar{\bar{\delta}}^{-1},\bar{y}) = c_{\mathcal{F}}(\bar{\delta}^{-1},\bar{y}) + c_{\mathcal{F}}(\delta_1^{-1},\delta_1(\bar{x})) = c_{\mathcal{F}}(\bar{\delta},\delta_1(\bar{x})) + c_{\mathcal{F}}(\delta_1,\bar{x}) = c_{\mathcal{F}}(\bar{\bar{\delta}},\bar{x})$. This concludes our induction.

It follows that the cost of the cheapest edit script transforming $\bar{x}$ to $\bar{y}$ is also the cost of an edit script transforming $\bar{y}$ to $\bar{x}$, such that $d_{\mathcal{S},\mathcal{F}}(\bar{y},\bar{x}) \leq d_{\mathcal{S},\mathcal{F}}(\bar{x},\bar{y})$. By applying the same argument in the other direction we obtain $d_{\mathcal{S},\mathcal{F}}(\bar{y},\bar{x}) = d_{\mathcal{S},\mathcal{F}}(\bar{x},\bar{y})$.

Triangular Inequality: Let $\bar{\delta}$ be an edit script from $\Delta^*_{\mathcal{S},\mathcal{A}}$ such that $\bar{\delta}(\bar{x}) = \bar{z}$, as well as $c_{\mathcal{F}}(\bar{\delta},\bar{x}) = d_{\mathcal{S},\mathcal{F}}(\bar{y},\bar{z})$, and let $\bar{\delta}'$ be an edit script from $\Delta^*_{\mathcal{S},\mathcal{A}}$, such that $\bar{\delta}'(\bar{z}) = \bar{x}$, as well as $c_{\mathcal{F}}(\bar{\delta},\bar{z}) = d_{\mathcal{S},\mathcal{F}}(\bar{z},\bar{y})$.

Then, $\bar{\delta}'' := \bar{\delta}\bar{\delta}'$ is an edit script from $\Delta^*_{\mathcal{S},\mathcal{A}}$, such that $\bar{\delta}''(\bar{x}) = \bar{y}$, and $c_{\mathcal{F}}(\bar{\delta}'',\bar{x}) = c_{\mathcal{F}}(\bar{\delta},\bar{x}) + c_{\mathcal{F}}(\bar{\delta}',\bar{z}) = d_{\mathcal{S},\mathcal{F}}(\bar{x},\bar{z}) + d_{\mathcal{S},\mathcal{F}}(\bar{z},\bar{y})$. Thus, we obtain: $d_{\mathcal{S},\mathcal{F}}(\bar{x},\bar{y}) \leq d_{\mathcal{S},\mathcal{F}}(\bar{x},\bar{z}) + d_{\mathcal{S},\mathcal{F}}(\bar{z},\bar{y})$.

A.10 PROOF OF THEOREM 3.3

Recall the theorem we intend to prove.

Let $\mathcal{S}$ be a signature, let $\mathcal{G}$ be an edit tree grammar over $\mathcal{S}$, let $\mathcal{A}$ be an alphabet, and let $\mathcal{F}$ be an algebra over $\mathcal{S}$ and $\mathcal{A}$. Then, for any two sequences $\bar{x}, \bar{y} \in \mathcal{A}^*$, Algorithm 3.1 computes the edit distance $d_{\mathcal{G},\mathcal{F}}(\bar{x}, \bar{y})$ in $\mathcal{O}(|\bar{x}| \cdot |\bar{y}|)$ time and space complexity.

Proof

First, consider the claim regarding the complexity classes. Algorithm 3.1 maintains $|\Phi|$ arrays, each of size $(|\bar{x}|+1) \times (|\bar{y}|+1)$. We consider $|\Phi|$ to be a constant. Therefore, the space complexity is $\mathcal{O}(|\bar{x}| \cdot |\bar{y}|)$. Furthermore, Algorithm 3.1 involves two nested loops with $|\bar{x}|+1$ and $|\bar{y}|+1$ iterations respectively. All other loops inside the algorithm iterate over constants, namely the number of production rules $|\mathcal{R}|$, or the number of nonterminal symbols $|\Phi|$. Therefore, we obtain a runtime complexity of $\mathcal{O}(|\bar{x}| \cdot |\bar{y}|)$.

It remains to show that Algorithm 3.1 does indeed compute the edit distance $d_{\mathcal{G},\mathcal{F}}(\bar{x}, \bar{y})$. We first prove a more general, auxiliary result.

Let $\bar{x} = x_1 \ldots x_m$ and $\bar{y} = y_1 \ldots y_n$. Further, let $A \in \Phi$, and let:

$$\mathcal{T}(A, i, j) := \{\tilde{\delta} | \mathcal{Y}(\tilde{\delta}) = (x_i \ldots x_m, y_j \ldots y_n), A \to^* \tilde{\delta}\}$$

$$\tilde{\boldsymbol{D}}^A_{i,j} := \begin{cases} \min\{c_{\mathcal{F}}(\tilde{\delta}) | \tilde{\delta} \in \mathcal{T}(A, i, j)\} & \text{if } \mathcal{T}(A, i, j) \neq \emptyset \\ \infty & \text{otherwise} \end{cases}$$

Then, it holds for all $i \leq m+1$ and all $j \leq n+1$: $\boldsymbol{D}^A_{i,j} = \tilde{\boldsymbol{D}}^A_{i,j}$.

We prove this claim via induction over (i, j) in descending lexicographic order. First, consider $i = m+1$ and $j = n+1$. In that case, $\mathcal{T}(A, i, j) = \mathcal{T}(A, m+1, n+1) = \{\$\}$ if $A ::= \$ \in \mathcal{R}$ and $\mathcal{T}(A, i, j) = \emptyset$ otherwise. Further, $c_{\mathcal{F}}(\$) = 0$ for any algebra $\mathcal{F}$. Therefore, $\tilde{\boldsymbol{D}}^A_{i,j} = 0$ if $A ::= \$ \in \mathcal{R}$ and $\tilde{\boldsymbol{D}}^A_{i,j} = \infty$ otherwise. This is precisely modelled by lines 4-9 of Algorithm 3.1.

Now, consider the $i \leq m$ or $j \leq n$. Assume that $\boldsymbol{D}^A_{i,j} \neq \tilde{\boldsymbol{D}}^A_{i,j}$. In that case, one of the following cases has to apply.

$\boldsymbol{D}^A_{i,j} > \tilde{\boldsymbol{D}}^A_{i,j}$: In that case, there exists a script tree $\tilde{\delta} \in \mathcal{T}(A, i, j)$, such that $c_{\mathcal{F}}(\tilde{\delta}) < \boldsymbol{D}^A_{i,j}$. Distinguish the following cases with respect to $\tilde{\delta}$.

$\tilde{\delta} = \text{del}(x_i, \tilde{\delta}')$ for some del $\in$ Del, and some script tree $\tilde{\delta}'$ such that $\mathcal{Y}(\tilde{\delta}') = (x_{i+1} \ldots x_m, y_j \ldots y_n)$ and $\exists B \in \Phi$ with $A \to^1 \text{del}(x_i, B)$ as well as $B \to^* \tilde{\delta}'$. Then, per definition, $\tilde{\delta}' \in \mathcal{T}(B, i+1, j)$. Furthermore, we have $i \leq m$ such that all conditions in lines 14 and 15 of Algorithm 3.1 are fulfilled. This, in turn, implies that there exists a l such that $\theta_l = \boldsymbol{D}^B_{i+1,j} + c_{\text{del}}(x_i)$, which means that: $\boldsymbol{D}^A_{i,j} \leq \boldsymbol{D}^B_{i+1,j} + c_{\text{del}}(x_i) \overset{Ind.}{=} \tilde{\boldsymbol{D}}^B_{i+1,j} + c_{\text{del}}(x_i) \leq c_{\mathcal{F}}(\tilde{\delta}') + c_{\text{del}}(x_i) = c_{\mathcal{F}}(\tilde{\delta}) < \boldsymbol{D}^A_{i,j}$, which is a contradiction.

$\tilde{\delta} = \text{match}(x_i, \tilde{\delta}', y_j)$ for some script tree $\tilde{\delta}'$ such that $\mathcal{Y}(\tilde{\delta}') = (x_{i+1} \ldots x_m, y_{j+1} \ldots y_n)$ and $\exists B \in \Phi$ with $A \to^1 \text{match}(x_i, B, y_j)$ as well as $B \to^* \tilde{\delta}'$. Then, per definition,

$\tilde{\delta}' \in \mathcal{T}(B, i+1, j+1)$. Furthermore, we have $i \leq m$, $j \leq n$ and $x_i = y_j$, otherwise $A \to^1 \text{match}(x_i, B, y_j)$ would not hold. Therefore, all conditions in lines 20, 21, and 22 of Algorithm 3.1 are fulfilled, and thus there exists a l such that $\theta_l = \boldsymbol{D}^B_{i+1,j+1}$. This, in turn, implies that: $\boldsymbol{D}^A_{i,j} \leq \boldsymbol{D}^B_{i+1,j+1} \overset{Ind.}{=} \tilde{\boldsymbol{D}}^B_{i+1,j+1} \leq c_{\mathcal{F}}(\tilde{\delta}') = c_{\mathcal{F}}(\tilde{\delta}) < \boldsymbol{D}^A_{i,j}$, which is a contradiction.

$\tilde{\delta} = \text{rep}(x_i, \tilde{\delta}', y_j)$ for some rep $\in$ Rep, and some script tree $\tilde{\delta}'$ such that $\mathcal{Y}(\tilde{\delta}') = (x_{i+1} \dots x_m, y_{j+1} \dots y_n)$ and $\exists B \in \Phi$ with $A \to^1 \text{rep}(x_i, B, y_j)$ as well as $B \to^* \tilde{\delta}'$. Then, per definition, $\tilde{\delta}' \in \mathcal{T}(B, i+1, j+1)$. Furthermore, we have $i \leq m$ and $j \leq n$, such that the conditions in lines 20 and 27 of Algorithm 3.1 are fulfilled. This, in turn, implies that there exists some l such that $\theta_l = \boldsymbol{D}^B_{i+1,j+1} + c_{\text{rep}}(x_i, y_j)$, which means that: $\boldsymbol{D}^A_{i,j} \leq \boldsymbol{D}^B_{i+1,j+1} + c_{\text{rep}}(x_i, y_j) \overset{Ind.}{=} \tilde{\boldsymbol{D}}^B_{i+1,j+1} + c_{\text{rep}}(x_i, y_j) \leq c_{\mathcal{F}}(\tilde{\delta}') + c_{\text{rep}}(x_i, y_j) = c_{\mathcal{F}}(\tilde{\delta}) < \boldsymbol{D}^A_{i,j}$, which is a contradiction.

$\tilde{\delta} = \text{ins}(\tilde{\delta}', y_j)$ for some ins $\in$ Ins, and some script tree $\tilde{\delta}'$ such that $\mathcal{Y}(\tilde{\delta}') = (x_i \dots x_m, y_{j+1} \dots y_n)$ and $\exists B \in \Phi$ with $A \to^1 \text{ins}(B, y_j)$ as well as $B \to^* \tilde{\delta}'$. Then, per definition, $\tilde{\delta}' \in \mathcal{T}(B, i, j+1)$. Furthermore, we have $j \leq n$, such that all conditions in lines 32 and 33 of Algorithm 3.1 are fulfilled. This, in turn, implies that there exists some l such that $\theta_l = \boldsymbol{D}^B_{i,j+1} + c_{\text{ins}}(y_j)$, which means that: $\boldsymbol{D}^A_{i,j} \leq \boldsymbol{D}^B_{i,j+1} + c_{\text{ins}}(y_j) \overset{Ind.}{=} \tilde{\boldsymbol{D}}^B_{i,j+1} + c_{\text{ins}}(y_j) \leq c_{\mathcal{F}}(\tilde{\delta}') + c_{\text{ins}}(y_j) = c_{\mathcal{F}}(\tilde{\delta}) < \boldsymbol{D}^A_{i,j}$, which is a contradiction.

This covers all possible cases for $\tilde{\delta}$ such that we must conclude by contradiction that $\boldsymbol{D}^A_{i,j} \leq \tilde{\boldsymbol{D}}^A_{i,j}$.

$\boldsymbol{D}^A_{i,j} < \tilde{\boldsymbol{D}}^A_{i,j}$: In that case, one of the following cases must apply.

$i \leq m$ **and** $\boldsymbol{D}^B_{i+1,j} + c_{\text{del}}(x_i) < \tilde{\boldsymbol{D}}^A_{i,j}$ for some del $\in$ Del and some $B \in \Phi$ such that $A ::= \text{del}B \in \mathcal{R}$. Then, per induction, we have $\tilde{\boldsymbol{D}}^B_{i+1,j} = \boldsymbol{D}^B_{i+1,j}$, which means that there exists a script tree $\tilde{\delta}'$ such that $\mathcal{Y}(\tilde{\delta}') = (x_{i+1} \dots x_m, y_j \dots y_n)$, $B \to^* \tilde{\delta}'$, and $c_{\mathcal{F}}(\tilde{\delta}') = \boldsymbol{D}^B_{i+1,j}$. Now, consider the script tree $\tilde{\delta} := \text{del}(x_i, \tilde{\delta}')$. It holds $A \to^* \tilde{\delta}$ because $A \to^1 \text{del}(x_i, B)$ and $B \to^* \tilde{\delta}'$. Furthermore, it holds $\mathcal{Y}(\tilde{\delta}) = (x_i \dots x_m, y_j \dots y_n)$. Therefore, per definition, $\tilde{\delta} \in \mathcal{T}(A, i, j)$, which in turn implies that: $\tilde{\boldsymbol{D}}^A_{i,j} \leq c_{\mathcal{F}}(\tilde{\delta}) = c_{\mathcal{F}}(\tilde{\delta}') + c_{\text{del}}(x_i) = \boldsymbol{D}^B_{i+1,j} + c_{\text{del}}(x_i) < \tilde{\boldsymbol{D}}^A_{i,j}$, which is a contradiction.

$i \leq m$**,** $j \leq n$**,** $x_i = y_j$**, and** $\boldsymbol{D}^B_{i+1,j+1} < \tilde{\boldsymbol{D}}^A_{i,j}$ for some $B \in \Phi$ such that $A ::= \text{match}B \in \mathcal{R}$. Then, per induction, we have $\tilde{\boldsymbol{D}}^B_{i+1,j+1} = \boldsymbol{D}^B_{i+1,j+1}$, which means that there exists a script tree $\tilde{\delta}'$ such that $\mathcal{Y}(\tilde{\delta}') = (x_{i+1} \dots x_m, y_{j+1} \dots y_n)$, $B \to^* \tilde{\delta}'$, and $c_{\mathcal{F}}(\tilde{\delta}') = \boldsymbol{D}^B_{i+1,j+1}$. Now, consider the script tree $\tilde{\delta} := \text{match}(x_i, \tilde{\delta}', y_j)$. It holds $A \to^* \tilde{\delta}$ because $A \to^1 \text{match}(x_i, B, y_j)$ and $B \to^* \tilde{\delta}'$. Furthermore, it holds $\mathcal{Y}(\tilde{\delta}) = (x_i \dots x_m, y_j \dots y_n)$. Therefore, per definition, $\tilde{\delta} \in \mathcal{T}(A, i, j)$, which in turn implies that: $\tilde{\boldsymbol{D}}^A_{i,j} \leq c_{\mathcal{F}}(\tilde{\delta}) = c_{\mathcal{F}}(\tilde{\delta}') = \boldsymbol{D}^B_{i+1,j+1} < \tilde{\boldsymbol{D}}^A_{i,j}$, which is a contradiction.

$i \leq m$**,** $j \leq n$**, and** $\boldsymbol{D}^B_{i+1,j+1} + c_{\text{rep}}(x_i, y_j) < \tilde{\boldsymbol{D}}^A_{i,j}$ for some rep $\in$ Rep and some $B \in \Phi$ such that $A ::= \text{rep}B \in \mathcal{R}$. Then, per induction, we have $\tilde{\boldsymbol{D}}^B_{i+1,j+1} = \boldsymbol{D}^B_{i+1,j+1}$, which means that there exists a script tree $\tilde{\delta}'$ such that $\mathcal{Y}(\tilde{\delta}') = (x_{i+1} \dots x_m, y_{j+1} \dots y_n)$,

$B \to^* \tilde{\delta}'$, and $c_{\mathcal{F}}(\tilde{\delta}') = \boldsymbol{D}^B_{i+1,j+1}$. Now, consider the script tree $\tilde{\delta} := \text{rep}(x_i, \tilde{\delta}', y_j)$. It holds $A \to^* \tilde{\delta}$ because $A \to^1 \text{rep}(x_i, B, y_j)$ and $B \to^* \tilde{\delta}'$. Furthermore, it holds $\mathcal{Y}(\tilde{\delta}) = (x_i \ldots x_m, y_j \ldots y_n)$. Therefore, per definition, $\tilde{\delta} \in \mathcal{T}(A, i, j)$, which in turn implies that: $\tilde{\boldsymbol{D}}^A_{i,j} \leq c_{\mathcal{F}}(\tilde{\delta}) = c_{\mathcal{F}}(\tilde{\delta}') + c_{\text{rep}}(x_i, y_j) = \boldsymbol{D}^B_{i+1,j+1} + c_{\text{rep}}(x_i, y_j) < \tilde{\boldsymbol{D}}^A_{i,j}$, which is a contradiction.

$j \leq n$ **and** $\boldsymbol{D}^B_{i,j+1} + c_{\text{ins}}(y_j) < \tilde{\boldsymbol{D}}^A_{i,j}$ for some ins $\in$ Ins and some $B \in \Phi$ such that $A ::= \text{ins}B \in \mathcal{R}$. Then, per induction, we have $\tilde{\boldsymbol{D}}^B_{i,j+1} = \boldsymbol{D}^B_{i,j+1}$, which means that there exists a script tree $\tilde{\delta}'$ such that $\mathcal{Y}(\tilde{\delta}') = (x_i \ldots x_m, y_{j+1} \ldots y_n)$, $B \to^* \tilde{\delta}'$, and $c_{\mathcal{F}}(\tilde{\delta}') = \boldsymbol{D}^B_{i,j+1}$. Now, consider the script tree $\tilde{\delta} := \text{ins}(\tilde{\delta}', y_j)$. It holds $A \to^* \tilde{\delta}$ because $A \to^1 \text{ins}(B, y_j)$ and $B \to^* \tilde{\delta}'$. Furthermore, it holds $\mathcal{Y}(\tilde{\delta}) = (x_i \ldots x_m, y_j \ldots y_n)$. Therefore, per definition, $\tilde{\delta} \in \mathcal{T}(A, i, j)$, which in turn implies that: $\tilde{\boldsymbol{D}}^A_{i,j} \leq c_{\mathcal{F}}(\tilde{\delta}) = c_{\mathcal{F}}(\tilde{\delta}') + c_{\text{ins}}(y_j) = \boldsymbol{D}^B_{i,j+1} + c_{\text{ins}}(y_j) < \tilde{\boldsymbol{D}}^A_{i,j}$, which is a contradiction.

This covers all possible cases for a value of $\boldsymbol{D}^A_{i,j}$, such that we must conclude by contradiction that $\boldsymbol{D}^A_{i,j} \geq \tilde{\boldsymbol{D}}^A_{i,j}$, which in turn implies $\boldsymbol{D}^A_{i,j} = \tilde{\boldsymbol{D}}^A_{i,j}$.

Note that, in case $L = 0$ or $\mathcal{T}(A, i, j) = \emptyset$, both $\boldsymbol{D}^A_{i,j}$ as well as $\tilde{\boldsymbol{D}}^A_{i,j}$ are defined as ∞, keeping the equality intact. This concludes our proof by induction.

By virtue of this general result we can now conclude that it holds:

$$\begin{aligned}
\boldsymbol{D}^{\mathrm{S}}_{1,1} = \tilde{\boldsymbol{D}}^{\mathrm{S}}_{1,1} &= \min\{c_{\mathcal{F}}(\tilde{\delta}) | \tilde{\delta} \in \mathcal{T}(\mathrm{S}, 1, 1)\} = \min\{c_{\mathcal{F}}(\tilde{\delta}) | \mathcal{Y}(\tilde{\delta}) = (\bar{x}, \bar{y}), \mathrm{S} \to^* \tilde{\delta}\} \\
&= \min_{\tilde{\delta} \in \mathcal{L}(\mathcal{G})} \{c_{\mathcal{F}}(\tilde{\delta}) | \mathcal{Y}(\tilde{\delta}) = (\bar{x}, \bar{y})\} = d_{\mathcal{G},\mathcal{F}}(\bar{x}, \bar{y})
\end{aligned}$$

A.11 PROOF OF THEOREM 3.4

Recall the theorem we intend to prove.

Let $\theta_1, \ldots, \theta_L \in \mathbb{R}$. Then, for any $\beta > 0$, softmin_β is differentiable with the following gradient.

$$\begin{aligned}
\nabla_{\vec{\lambda}} \text{softmin}_\beta(\theta_1, \ldots, \theta_L) &= \sum_{l=1}^{L} \text{softmin}'_{\beta,l}(\theta_1, \ldots, \theta_L) \cdot \nabla_{\vec{\lambda}} \theta_l \qquad \text{where} \qquad (A.55) \\
\text{softmin}'_{\beta,l}(\theta_1, \ldots, \theta_L) &= \frac{\exp(-\beta \cdot \theta_l)}{\sum_{l'=1}^{L} \exp(-\beta \cdot \theta_{l'})} \cdot \Big(1 - \beta \cdot \big[\theta_l - \text{softmin}_\beta(\theta_1, \ldots, \theta_L)\big]\Big)
\end{aligned}$$

Further, there exists a constant $C_L \in \mathbb{R}$ such that for all $\beta > 0$ it holds:

$$0 \leq \text{softmin}_\beta(\theta_1, \ldots, \theta_L) - \min\{\theta_1, \ldots, \theta_L\} \leq \frac{C_L}{\beta}$$

Proof

We prove the claims in turn. First, recall the definition of the softmin from Equation 3.4.

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

To avoid symbol clutter, we introduce the notational shorthands $e_l := \exp(-\beta \cdot \theta_l)$ and $Z := \sum_{l=1}^{L} e_l$. Accordingly, we can write the gradient of $\text{softmin}_\beta(\theta_1, \ldots, \theta_L)$ with respect to parameters $\vec{\lambda}$ as follows.

$$\begin{aligned}\nabla_{\vec{\lambda}}\text{softmin}_\beta(\theta_1, \ldots, \theta_L) &= \nabla_{\vec{\lambda}} \Big(\frac{1}{Z} \cdot \sum_{l=1}^{L} e_l \cdot \theta_l \Big) \\ =&\frac{1}{Z^2} \cdot \Big(Z \cdot \sum_{l=1}^{L} \nabla_{\vec{\lambda}}(e_l \cdot \theta_l) - \Big(\sum_{l=1}^{L} e_l \cdot \theta_l \Big) \cdot \nabla_{\vec{\lambda}} Z \Big) \\ =&\frac{1}{Z} \cdot \Big(\sum_{l=1}^{L} \nabla_{\vec{\lambda}}(e_l \cdot \theta_l) - \text{softmin}_\beta(\theta_1, \ldots, \theta_L) \cdot \nabla_{\vec{\lambda}} Z \Big)\end{aligned} \tag{A.56}$$

It remains to compute the gradients of $e_l \cdot \theta_l$ and Z with respect to $\vec{\lambda}$. In that regard, we obtain

$$\begin{aligned}\nabla_{\vec{\lambda}} e_l \cdot \theta_l &= \Big(\nabla_{\vec{\lambda}} e_l \Big) \cdot \theta_l + e_l \cdot \nabla_{\vec{\lambda}} \theta_l \\ &= -\beta \cdot e_l \cdot \theta_l \cdot \nabla_{\vec{\lambda}} \theta_l + e_l \cdot \nabla_{\vec{\lambda}} \theta_l \\ &= e_l \cdot (-\beta \cdot \theta_l + 1) \cdot \nabla_{\vec{\lambda}} \theta_l\end{aligned}$$

as well as

$$\nabla_{\vec{\lambda}} Z = \sum_{l=1}^{L} \nabla_{\vec{\lambda}} e_l = \sum_{l=1}^{L} -\beta \cdot e_l \cdot \nabla_{\vec{\lambda}} \theta_l$$

Plugging these results into Equation A.56, we obtain:

$$\begin{aligned}&\frac{1}{Z} \cdot \Bigg(\Big[\sum_{l=1}^{L} e_l \cdot (-\beta \cdot \theta_l + 1) \cdot \nabla_{\vec{\lambda}} \theta_l \Big] - \text{softmin}_\beta(\theta_1, \ldots, \theta_L) \cdot \Big[\sum_{l=1}^{L} -\beta \cdot e_l \cdot \nabla_{\vec{\lambda}} \theta_l \Big] \Bigg) \\ =&\sum_{l=1}^{L} \frac{e_l}{Z} \cdot \Big(-\beta \cdot \theta_l + 1 + \text{softmin}_\beta(\theta_1, \ldots, \theta_L) \cdot \beta \Big) \cdot \nabla_{\vec{\lambda}} \theta_l \\ =&\sum_{l=1}^{L} \frac{e_l}{Z} \cdot \Big(1 - \beta \cdot [\theta_l - \text{softmin}_\beta(\theta_1, \ldots, \theta_L)] \Big) \cdot \nabla_{\vec{\lambda}} \theta_l\end{aligned}$$

which concludes the gradient computation.

Regarding the second claim, we re-write $\text{softmin}_\beta(\theta_1, \ldots, \theta_L)$ as follows. Let $\theta^* := \min_l\{\theta_l\}$. Then, we obtain:

$$\begin{aligned}\text{softmin}_\beta(\theta_1, \ldots, \theta_L) &= \frac{\sum_{l=1}^{L} \exp(-\beta \cdot \theta_l) \cdot (\theta_l - \theta^* + \theta^*)}{\sum_{l=1}^{L} \exp(-\beta \cdot \theta_l)} \\ &= \frac{\exp(-\beta \cdot \theta^*)}{\exp(-\beta \cdot \theta^*)} \cdot \frac{\sum_{l=1}^{L} \exp(-\beta \cdot \theta_l) \cdot (\theta_l - \theta^*)}{\sum_{l=1}^{L} \exp(-\beta \cdot \theta_l)} + \theta^* \\ &= \frac{\sum_{l=1}^{L} \exp(-\beta \cdot [\theta_l - \theta^*]) \cdot (\theta_l - \theta^*)}{\sum_{l=1}^{L} \exp(-\beta \cdot [\theta_l - \theta^*])} + \theta^*\end{aligned}$$

Now, let E denote the difference $\text{softmin}_\beta(\theta_1, \ldots, \theta_L) - \theta^*$. Further, let Θ^* denote the set $\Theta^* := \{l | \theta_l = \theta^*\}$. Note that, per construction, this set must contain at least one element. We can re-write E using this set as follows.

$$E = \frac{\sum_{l \notin \Theta^*} \exp(-\beta \cdot [\theta_l - \theta^*]) \cdot (\theta_l - \theta^*)}{|\Theta^*| + \sum_{l \notin \Theta^*} \exp(-\beta \cdot [\theta_l - \theta^*])}$$

Note that this term is non-negative and strictly positive for $|\Theta^*| < L$, because $\exp(-\beta \cdot [\theta_l - \theta^*]) > 0$, $(\theta_l - \theta^*) > 0$ for all $l \notin \Theta^*$, and $|\Theta^*| > 0$. It remains to show that the upper bound holds.

First, note that decreasing the number of elements in Θ^* increases the enumerator and decreases the denominator. Therefore, we obtain an upper bound for $|\Theta^*| = 1$. Further, note that E is symmetric with respect to all θ_l, auch that we obtain a maximum if for all $l \notin \Theta^*$ we have $\theta_l = \theta^* + \varepsilon$ for some constant $\varepsilon > 0$. Plugging this into our expression above, we obtain:

$$E \leq \max_{\varepsilon \in \mathbb{R}} \frac{(L-1) \cdot \exp(-\beta \cdot \varepsilon) \cdot \varepsilon}{1 + (L-1) \cdot \exp(-\beta \cdot \varepsilon)} = \max_{\varepsilon \in \mathbb{R}} \frac{(L-1) \cdot \varepsilon}{\exp(\beta \cdot \varepsilon) + L - 1} \tag{A.57}$$

We obtain the maximum by deriving with respect to ε. In particular, let $E(\varepsilon) := \varepsilon / (\exp(\beta \cdot \varepsilon) + L - 1)$. Then, we obtain the following first and second derivative of $E(\varepsilon)$ with respect to ε.

$$\frac{\partial}{\partial \varepsilon} E(\varepsilon) = \frac{\exp(\beta \cdot \varepsilon) + L - 1 - \varepsilon \cdot \beta \cdot \exp(\beta \cdot \varepsilon)}{\Big(\exp(\beta \cdot \varepsilon) + L - 1\Big)^2} = \frac{1 - \beta \cdot \exp(\beta \cdot \varepsilon) \cdot E(\varepsilon)}{\exp(\beta \cdot \varepsilon) + L - 1} \tag{A.58}$$

$$\begin{aligned}\frac{\partial^2}{\partial^2 \varepsilon} E(\varepsilon) = \Big(& -\beta \cdot \exp(\beta \cdot \varepsilon) \cdot \big(\frac{\partial}{\partial \varepsilon} E(\varepsilon) + \beta \cdot E(\varepsilon)\big) \cdot (\exp(\beta \cdot \varepsilon) + L - 1) \\ & - (1 - \beta \cdot \exp(\beta \cdot \varepsilon) \cdot E(\varepsilon)) \cdot \beta \cdot \exp(\beta \cdot \varepsilon)\Big) \cdot \frac{1}{\big(\exp(\beta \cdot \varepsilon) + L - 1\big)^2}\end{aligned} \tag{A.59}$$

First, consider the second derivative with respect to any candidate $\varepsilon > 0$ with $\frac{\partial}{\partial \varepsilon} E(\varepsilon) = 0$. For these points, we obtain the following result for the enumerator of Equation A.59.

$$\begin{aligned}& -\beta^2 \cdot \exp(\beta \cdot \varepsilon) \cdot E(\varepsilon) \cdot (\exp(\beta \cdot \varepsilon) + L - 1) - (1 - \beta \cdot \exp(\beta \cdot \varepsilon) \cdot E(\varepsilon)) \cdot \beta \cdot \exp(\beta \cdot \varepsilon) \\ = & -\beta \cdot \exp(\beta \cdot \varepsilon) \cdot \Big(\beta \cdot E(\varepsilon) \cdot \exp(\beta \cdot \varepsilon) + (L-1) \cdot \beta \cdot E(\varepsilon) + 1 - \beta \cdot \exp(\beta \cdot \varepsilon) \cdot E(\varepsilon)\Big) \\ = & -\beta \cdot \exp(\beta \cdot \varepsilon) \cdot \Big((L-1) \cdot \beta \cdot E(\varepsilon) + 1\Big)\end{aligned}$$

Note that this expression is strictly negative because $L - 1 \geq 0$, $\beta > 0$, $E(\varepsilon) \geq 0$ and $\exp(\beta \cdot \varepsilon) > 0$. Further, due to the square, the denominator in Equation A.59 is strictly positive. Therefore, every $\varepsilon > 0$ with $\frac{\partial}{\partial \varepsilon} E(\varepsilon) = 0$ is a maximum.

Next, we solve the equation $\frac{\partial}{\partial \varepsilon} E(\varepsilon) = 0$, for which we obtain:

$$\begin{aligned}& \exp(\beta \cdot \varepsilon) + L - 1 - \beta \cdot \varepsilon \cdot \exp(\beta \cdot \varepsilon) \stackrel{!}{=} 0 \\ \iff & \exp(\beta \cdot \varepsilon) \cdot (1 - \beta \cdot \varepsilon) \stackrel{!}{=} 1 - L\end{aligned} \tag{A.60}$$

Note that the function $\exp(\beta \cdot \varepsilon) \cdot (1 - \beta \cdot \varepsilon)$ is strictly descending with respect to ε, starting from $\exp(\beta \cdot 0) \cdot (1 - \beta \cdot 0) = 1$. We can verify this finding by considering the derivative with respect to ε, which yields $-\beta^2 \cdot \varepsilon \cdot \exp(\beta \cdot \varepsilon)$. Since $\beta > 0$ and $\varepsilon > 0$, this value is strictly negative. Due to this shape, this equation has only one solution for any $L \in \mathbb{N}$.

To solve Equation A.60, we require the Lambert W function, which is defined via the equation $W(x \cdot \exp(x)) = x$. Note that W is invertible for non-negative values of x and that we can thus obtain the equation

$$W(x) \cdot \exp(W(x)) = W^{-1}\Big[W\big(W[x] \cdot \exp(W[x])\big)\Big] = W^{-1}(W[x]) = x \tag{A.61}$$

As unique solution of Equation A.60 we now obtain $\varepsilon = \frac{1}{\beta} \cdot \left(W\left(\frac{L-1}{e}\right) + 1\right)$, which we can verify by plugging this solution into Equation A.60.

$$\begin{aligned}
&\exp(W\Big(\frac{L-1}{e}\Big) + 1) \cdot (1 - (W\Big(\frac{L-1}{e}\Big) + 1)) && \stackrel{!}{=} 1 - L \\
\iff & - e \cdot \exp(W\Big(\frac{L-1}{e}\Big)) \cdot W\Big(\frac{L-1}{e}\Big) && \stackrel{!}{=} 1 - L \\
\stackrel{A.61}{\iff} & - e \cdot \frac{L-1}{e} && \stackrel{!}{=} 1 - L \\
\iff & 1 - L && \stackrel{!}{=} 1 - L
\end{aligned}$$

Finally, note that $E(\varepsilon)$ tends to zero for boundary values. In particular, we obtain $E(0) = 0$ and $\lim_{\varepsilon\to\infty} E(\varepsilon) = 0$. Therefore, our solution is a global maximum of $E(\varepsilon)$.

We obtain our upper bound by plugging our result for ε into Equation A.57.

$$E \leq \frac{1}{\beta} \cdot \frac{(L-1) \cdot \left(W\left(\frac{L-1}{e}\right) + 1\right)}{\exp(W\left(\frac{L-1}{e}\right) + 1) + L - 1}$$

Therefore, we can set our constant C_L to

$$C_L = \frac{W\left(\frac{L-1}{e}\right) + 1}{\frac{1}{L-1} \cdot \exp(W\left(\frac{L-1}{e}\right) + 1) + 1}$$

which concludes our proof.

A.12 PROOF OF THEOREM 3.5

Recall the theorem we intend to prove.

Let $\mathcal{S}$ be a signature, let $\mathcal{G}$ be an edit tree grammar over $\mathcal{S}$, let $\mathcal{A}$ be an alphabet, and let $\mathcal{F}$ be an algebra over $\mathcal{S}$ and $\mathcal{A}$. Finally, let $\vec{\lambda}$ be arbitrary parameters of $\mathcal{F}$, and let $\beta \in \mathbb{R}$ with $\beta > 0$.

Then, for any two sequences $\bar{x}, \bar{y} \in \mathcal{A}$, we define the β-softmin-approximated edit distance $d_{\beta,\mathcal{G},\mathcal{F}}(\bar{x}, \bar{y})$ as the result of Algorithm 3.1 with a softmin operation in line 39 instead of a strict minimum operation.

Further, it holds: Algorithm 3.2 computes the gradient of the β-softmin-approximated edit distance $d_{\beta,\mathcal{G},\mathcal{F}}(\bar{x}, \bar{y})$ with respect to $\vec{\lambda}$ in $\mathcal{O}(|\bar{x}| \cdot |\bar{y}|)$ time and space complexity.

Proof

First, consider the claim regarding the complexity classes. Algorithm 3.2 maintains $|\Phi|$ arrays of size $(|\bar{x}| + 1) \times (|\bar{y}| + 1)$, and $|\Phi|$ arrays of size $(|\bar{x}| + 1) \times (|\bar{y}| + 1) \times |\vec{\lambda}|$. We consider $f|\Phi|$ and $|\vec{\lambda}|$ to be constants. Therefore, the space complexity is $\mathcal{O}(|\bar{x}| \cdot |\bar{y}|)$. Furthermore, Algorithm 3.2 involves two nested loops with $|\bar{x}| + 1$ and $|\bar{y}| + 1$ iterations respectively. All other loops inside the algorithm iterate over constants, namely the number of production rules $|\mathcal{R}|$, or the number of nonterminal symbols $|\Phi|$. Therefore, we obtain a runtime complexity of $\mathcal{O}(|\bar{x}| \cdot |\bar{y}|)$.

It remains to show that Algorithm 3.1 does indeed compute the gradient $\nabla_{\vec{\lambda}} d_{\beta,\mathcal{G},\mathcal{F}}(\bar{x}, \bar{y})$.

Recall that $d_{\beta,\mathcal{G},\mathcal{F}}(\bar{x}, \bar{y})$ is defined as the result of Algorithm 3.1 with softmin instead of the minimum in line 39. Observe that Algorithm 3.2 computes this approximated distance, which is contained in $\boldsymbol{D}^{\mathrm{S}}_{1,1}$ after executing the loops in line 11 and 12. To obtain the gradient $\nabla_{\vec{\lambda}} \boldsymbol{D}^{\mathrm{S}}_{1,1}$, we can trace back its computation through the algorithm and adjust it accordingly.

First, note that $\boldsymbol{D}^{\mathrm{S}}_{1,1}$ is computed in the last iteration of the loops in line 11 and 12. In particular $\boldsymbol{D}^{\mathrm{S}}_{1,1}$ is computed in line 44, which reads $A_{i,j} \leftarrow \mathrm{softmin}(\theta_1, \ldots, \theta_L)$. To obtain the gradient $\nabla_{\vec{\lambda}} \boldsymbol{D}^{A}_{i,j}$, we utilize Equation 3.5, which yields line 45, i.e.: $\boldsymbol{G}^{A}_{i,j} \leftarrow \sum_{l=1}^{L} \mathrm{softmin}'_{\beta,l}(\theta_1, \ldots, \theta_L) \cdot \nabla_{\vec{\lambda}} \theta_l$.

Note that this equation depends on the gradients $\nabla_{\vec{\lambda}} \theta_l$ for all $l \in \{1, \ldots, L\}$. The terms θ_l are computed in lines 18, 26, 32, and 39 of Algorithm 3.2. Accordingly, we introduce lines 19, 27, 33, and 40, which compute the gradients $\nabla_{\vec{\lambda}} \theta_l$. These gradients in turn depend on the gradients $\nabla_{\vec{\lambda}} \boldsymbol{D}^{B}_{i+1,j} = \boldsymbol{G}^{B}_{i+1,j}$, $\nabla_{\vec{\lambda}} \boldsymbol{D}^{B}_{i+1,j+1} = \boldsymbol{G}^{B}_{i+1,j+1}$, and $\nabla_{\vec{\lambda}} \boldsymbol{D}^{B}_{i,j+1} = \boldsymbol{G}^{B}_{i,j+1}$, all of which are already pre-computed due to prior runs of the loop.

It remains to consider the base case. The entries $\boldsymbol{D}^{A}_{m+1,n+1}$ are either zero or ∞. In the former case, the entry is a constant independent of $\vec{\lambda}$. Therefore, the initialization with zeros is correct. In case the entry is ∞, the value of $\boldsymbol{G}^{A}_{m+1,n+1}$ is irrelevant for the final result, because for any term $\theta_l = \infty$ it holds: $\mathrm{softmin}'_{\beta,l}(\theta_1, \ldots, \theta_L) = 0$ such that the corresponding term in the sum in line 45 is discarded.

Therefore, we can conclude that the result of Algorithm 3.2 is indeed the desired gradient, which concludes the proof.

A.13 PROOF OF THEOREM 4.2

Recall the Theorem we intend to prove.

Let $\tilde{x}$ and $\tilde{y}$ be trees over some alphabet $\mathcal{A}$ and let c be a cost function over $\mathcal{A}$ which conforms to the triangular inequality. Then, Algorithm 4.1 computes $\boldsymbol{P}_c(\tilde{x}, \tilde{y}) \cdot |\mathcal{M}(c, \tilde{x}, \tilde{y})|$ as first output and $|\mathcal{M}(c, \tilde{x}, \tilde{y})|$ as second output. Further, Algorithm 4.1 runs in $\mathcal{O}(|\tilde{x}|^6 \cdot |\tilde{y}|^6)$ time complexity and $\mathcal{O}(|\tilde{x}|^2 \cdot |\tilde{y}|^2)$ space complexity in the worst case.

Algorithm A.1 An algorithm which computes the matrix $\boldsymbol{A}$ for two trees $\tilde{x}$ and $\tilde{y}$ and a cost function c.

function FORWARD(Two trees $\tilde{x}$ and $\tilde{y}$, the matrices $\boldsymbol{d}$ and $\boldsymbol{D}$ after executing Algorithm 2.1, and a cost function c)
  Initialize $\boldsymbol{A}$ as a $(|\tilde{x}|+1) \times (|\tilde{y}|+1)$ matrix of zeros.
  $\boldsymbol{A}_{1,1} \leftarrow 1$, $Q \leftarrow \{(1,1)\}$
  $C \leftarrow \emptyset$.
  while $Q \neq \emptyset$ **do**
    $(i,j) \leftarrow \min Q$. ▷ Lexicographic ordering
    $Q \leftarrow Q \setminus \{(i,j)\}$.
    $C \leftarrow C \cup \{(i,j)\}$.
    if $i \leq |\tilde{x}| \wedge \boldsymbol{D}_{i,j} = c(x_i, -) + \boldsymbol{D}_{i+1,j}$ **then**
      $\boldsymbol{A}_{i+1,j} \leftarrow \boldsymbol{A}_{i+1,j} + \boldsymbol{A}_{i,j}$.
      $Q \leftarrow Q \cup \{(i+1,j)\}$.
    end if
    if $j \leq |\tilde{y}| \wedge \boldsymbol{D}_{i,j} = c(-, y_j) + \boldsymbol{D}_{i,j+1}$ **then**
      $\boldsymbol{A}_{i,j+1} \leftarrow \boldsymbol{A}_{i,j+1} + \boldsymbol{A}_{i,j}$.
      $Q \leftarrow Q \cup \{(i,j+1)\}$.
    end if
    if $i = |\tilde{x}|+1 \vee j = |\tilde{y}|+1 \vee c(x_i, y_j) = c(x_i, -) + c(-, y_j)$ **then**
      continue
    end if
    if $rl_{\tilde{x}}(i) = rl_{\tilde{x}}(1) \wedge rl_{\tilde{y}}(j) = rl_{\tilde{y}}(1)$ **then**
      if $\boldsymbol{D}_{i,j} = \boldsymbol{D}_{i+1,j+1} + c(x_i, y_j)$ **then**
        $\boldsymbol{A}_{i+1,j+1} \leftarrow \boldsymbol{A}_{i+1,j+1} + \boldsymbol{A}_{i,j}$
        $Q \leftarrow Q \cup \{(i+1, j+1)\}$.
      end if
    else
      if $\boldsymbol{D}_{i,j} = \boldsymbol{D}_{rl_{\tilde{x}}(i)+1, rl_{\tilde{y}}(j)+1} + \boldsymbol{d}_{i,j}$ **then**
        Compute $\boldsymbol{D}'$ and $\boldsymbol{d}'$ via Algorithm 2.1 for the subtrees $\tilde{x}^i$ and $\tilde{y}^j$.
        $\boldsymbol{D}'_{1,2} \leftarrow \infty$. $\boldsymbol{D}'_{2,1} \leftarrow \infty$.
        $(Q', \boldsymbol{A}') \leftarrow$ FORWARD($\tilde{x}^i$, $\tilde{y}^j$, $\boldsymbol{d}'$, $\boldsymbol{D}'$, c).
        $\boldsymbol{A}_{rl_{\tilde{x}}(i)+1, rl_{\tilde{y}}(j)+1} \leftarrow \boldsymbol{A}_{rl_{\tilde{x}}(i)+1, rl_{\tilde{y}}(j)+1} + \boldsymbol{A}'_{|\tilde{x}^i|+1, |\tilde{y}^j|+1} \cdot \boldsymbol{A}_{i,j}$.
        $Q \leftarrow Q \cup \{(rl_{\tilde{x}}(i)+1, rl_{\tilde{y}}(j)+1)\}$.
      end if
    end if
  end while
  return $(C, \boldsymbol{A})$.
end function

Algorithm A.2 An algorithm which computes the matrix $\boldsymbol{B}$ for two trees $\tilde{x}$ and $\tilde{y}$ and a cost function c.

function BACKWARD(Two trees $\tilde{x}$ and $\tilde{y}$, the matrices $\boldsymbol{d}$ and $\boldsymbol{D}$ after executing Algorithm 2.1, a cost function c, and a set of tuples C as returned by Algorithm A.1.)
    Initialize $\boldsymbol{B}$ as a $(|\tilde{x}|+1) \times (|\tilde{y}|+1)$ matrix of zeros.
    $\boldsymbol{B}_{|\tilde{x}|+1,|\tilde{y}|+1} \leftarrow 1$.
    while $C \neq \emptyset$ **do**
        $(i, j) \leftarrow \max C$. ▷ Lexicographic ordering
        $C \leftarrow C \setminus \{(i, j)\}$.
        if $i \leq |\tilde{x}| \wedge \boldsymbol{D}_{i,j} = c(x_i, -) + \boldsymbol{D}_{i+1,j}$ **then**
            $\boldsymbol{B}_{i,j} \leftarrow \boldsymbol{B}_{i,j} + \boldsymbol{B}_{i+1,j}$
        end if
        if $j \leq |\tilde{y}| \wedge \boldsymbol{D}_{i,j} = c(-, y_j) + \boldsymbol{D}_{i,j+1}$ **then**
            $\boldsymbol{B}_{i,j} \leftarrow \boldsymbol{B}_{i,j} + \boldsymbol{B}_{i,j+1}$
        end if
        if $i = |\tilde{x}| + 1 \vee j = |\tilde{y}| + 1 \vee c(x_i, y_j) = c(x_i, -) + c(-, y_j)$ **then**
            continue
        end if
        if $rl_{\tilde{x}}(i) = rl_{\tilde{x}}(1) \wedge rl_{\tilde{y}}(j) = rl_{\tilde{y}}(1)$ **then**
            if $\boldsymbol{D}_{i,j} = \boldsymbol{D}_{i+1,j+1} + c(x_i, y_j)$ **then**
                $\boldsymbol{B}_{i,j} \leftarrow \boldsymbol{B}_{i,j} + \boldsymbol{B}_{i+1,j+1}$
            end if
        else
            if $\boldsymbol{D}_{i,j} = \boldsymbol{D}_{rl_{\tilde{x}}(i)+1, rl_{\tilde{y}}(j)+1} + \boldsymbol{d}_{i,j}$ **then**
                Compute $\boldsymbol{D}'$ and $\boldsymbol{d}'$ via Algorithm 2.1 for the subtrees $\tilde{x}^i$ and $\tilde{y}^j$.
                $\boldsymbol{D}'_{1,2} \leftarrow \infty$. $\boldsymbol{D}'_{2,1} \leftarrow \infty$.
                $(Q', \boldsymbol{A}') \leftarrow$ FORWARD$(\tilde{x}^i, \tilde{y}^j, \boldsymbol{d}', \boldsymbol{D}', c)$.
                $\boldsymbol{B}_{i,j} \leftarrow \boldsymbol{B}_{i,j} + \boldsymbol{B}_{rl_{\tilde{x}}(i)+1, rl_{\tilde{y}}(j)+1} \cdot \boldsymbol{A}'_{|\tilde{x}^i|+1, |\tilde{y}^j|+1}$.
            end if
        end if
    end while
    return $\boldsymbol{B}$.
end function

Proof

First, consider the efficiency claims and consider Algorithm A.1. In the worst case, lines 27-31 need to be executed in each possible iteration. In that case, $\boldsymbol{D}'$ and $\boldsymbol{d}'$ need to be computed via Algorithm 2.1, which requires $\mathcal{O}(|\tilde{x}|^2 \cdot |\tilde{y}|^2)$ steps and $\mathcal{O}(|\tilde{x}| \cdot |\tilde{y}|)$ space. Including the recursive calls, this can occur $\mathcal{O}(|\tilde{x}|^2 \cdot |\tilde{y}|^2)$ times at worst such that Algorithm A.1 has an overall runtime complexity of $\mathcal{O}(|\tilde{x}|^4 \cdot |\tilde{y}|^4)$.

Regarding space complexity, each level of recursion needs to maintain a constant number of matrices of size $\mathcal{O}(|\tilde{x}| \cdot |\tilde{y}|)$. A worst, there can be $\mathcal{O}(|\tilde{x}| \cdot |\tilde{y}|)$ levels of recursion active at the same time, implying a space complexity of $\mathcal{O}(|\tilde{x}|^2 \cdot |\tilde{y}|^2)$.

Now, note that Algorithm A.2, by construction, iterates over the same elements as Algorithm A.1 and has the same structure, such that the complexity results carry over.

Finally, regarding Algorithm 4.1 itself, we find that, in the worst case, lines 15-23 get executed in every possible iteration. These lines include a recursive call to Algorithm 4.1, and in each such recursive call, Algorithm A.1 and Algorithm A.2 get executed. With the same argument as before, we perform at most $\mathcal{O}(|\tilde{x}|^2 \cdot |\tilde{y}|^2)$ of such recursive calls, yielding an overall runtime complexity of $\mathcal{O}(|\tilde{x}|^6 \cdot |\tilde{y}|^6)$ in the worst case.

Regarding space complexity, each level of recursion needs to maintain a constant number of matrices of size $\mathcal{O}(|\tilde{x}| \cdot |\tilde{y}|)$. A worst, there can be $\mathcal{O}(|\tilde{x}| \cdot |\tilde{y}|)$ levels of recursion active at the same time, implying a space complexity of $\mathcal{O}(|\tilde{x}|^2 \cdot |\tilde{y}|^2)$.

The remainder of this section is now dedicated to proving the correctness of Algorithm 4.1, that is, that the first output of Algorithm 4.1 is $\boldsymbol{P}_c(\tilde{x}, \tilde{y}) \cdot |\mathcal{M}(c, \tilde{x}, \tilde{y})|$, and that the second output is $|\mathcal{M}(c, \tilde{x}, \tilde{y})|$.

The outline of the correctness proof is as follows. We will first show that the problem of counting cooptimal tree mappings is related to the graph-theoretic problem of counting backtracing paths through the dynamic programming matrix $\boldsymbol{D}$ of Algorithm 4.1. Then, we show that Algorithm A.1 computes the number of paths from the first cell of matrix $\boldsymbol{D}$ to any other cell, and Algorithm A.2 computes the number of paths from any cell to the last cell of matrix $\boldsymbol{D}$. Finally, the correctness of Algorithm 4.1 follows naturally from these two prior claims because $\boldsymbol{P}_c(\tilde{x}, \tilde{y})_{i,j}$ is proportional to the number of paths to cell (i, j) multiplied with the number of paths from cell $(i+1, j+1)$.

We begin our proof by establishing auxiliary concepts, namely the concept of the cooptimal edit graph, and paths through that graph.

Definition A.6 (Co-optimal Edit Graph)**.** Let X and Y be forests over some alphabet $\mathcal{A}$ and let c be a cost function over $\mathcal{A}$. Then, we define the *cooptimal edit graph* between X and Y according to c as the directed graph $\mathcal{G}_{c,X,Y} = (V, E)$ with nodes V and edges E as follows.

If $X = \epsilon$ and $Y = \epsilon$ we define $V := \{(1,1,1,1)\}$ and $E := \emptyset$.

If $X = \epsilon$ but $Y \neq \epsilon$ we define $V := \big\{(1,1,1,j) \big| j \in \{1, \ldots, |Y|+1\}\big\}$ and $E := \big\{\big((1,1,1,j), (1,1,1,j+1)\big) \big| j \in \{1, \ldots, |Y|\}\big\}$.

If $X \neq \epsilon$ but $Y = \epsilon$ we define $V := \big\{(1,i,1,1) \big| i \in \{1, \ldots, |X|+1\}\big\}$ and $E := \big\{\big((1,i,1,1), (1,i+1,1,1)\big) \big| i \in \{1, \ldots, |X|\}\big\}$.

If neither forest is empty, we define:

$$
\begin{aligned}
V :=& \Big\{ (k,i,l,j) \Big| k \in \mathcal{K}(X), i \in \{k, \ldots, rl_X(k)+1\}, l \in \mathcal{K}(Y), j \in \{l, \ldots, rl_Y(l)+1\} \Big\} \\
E :=& \Big\{ \big((k,i,l,j), (k,i+1,l,j)\big) \Big| D_c\big(X[i, rl_X(k)], Y[j, rl_Y(l)]\big) \\
&\qquad = c(x_i, -) + D_c\big(X[i+1, rl_X(k)], Y[j, rl_Y(l)]\big) \Big\} \cup \\
&\Big\{ \big((k,i,l,j), (k,i,l,j+1)\big) \Big| D_c\big(X[i, rl_X(k)], Y[j, rl_Y(l)]\big) \\
&\qquad = c(-, y_j) + D_c\big(X[i, rl_X(k)], Y[j+1, rl_Y(l)]\big) \Big\} \cup
\end{aligned}
$$

$$\Big\{\big((k,i,l,j),(k,i+1,l,j+1)\big)\Big|D_c\big(X[i,rl_X(k)],Y[j,rl_Y(l)]\big)$$
$$= c(x_i,y_j)+D_c\big(X[i+1,rl_X(k)],Y[j+1,rl_Y(l)]\big)$$
$$\wedge\, rl_X(i)=rl_X(k)\wedge rl_Y(j)=rl_Y(l)\Big\}\cup$$
$$\Big\{\big((k,i,l,j),(k,i+1,l,j+1)\big)\Big|D_c\big(X[i,rl_X(k)],Y[j,rl_Y(l)]\big)$$
$$= c(x_i,y_j)+D_c\big(X[i+1,rl_X(k)],Y[j+1,rl_Y(l)]\big)$$
$$\wedge\, c(x_i,y_j)=c(x_i,-)+c(-,y_j)\Big\}\cup$$
$$\Big\{\big((k,i,l,j),(\mathrm{k}_X(i),i+1,\mathrm{k}_Y(j),j+1)\big)\Big|D_c\big(X[i,rl_X(k)],Y[j,rl_Y(l)]\big)$$
$$= D_c(\tilde{x}^i,\tilde{y}^j)+D_c\big(X[rl_X(i)+1,rl_X(k)],Y[rl_Y(j)+1,rl_Y(l)]\big)$$
$$\wedge\,\big(rl_X(i)\neq rl_X(k)\vee rl_Y(j)\neq rl_Y(l)\big)\wedge c(x_i,y_j)<c(x_i,-)+c(-,y_j)\Big\}\cup$$
$$\Big\{\big((k,rl_X(k)+1,l,rl_Y(l)+1),(\mathrm{k}_X(rl_X(k)+1),rl_X(k)+1,\mathrm{k}_Y(rl_Y(l)+1),rl_Y(l)+1)\big)\Big|$$
$$rl_X(k)+1\leq|X|\wedge rl_Y(l)+1\leq|Y|\Big\}$$
$$\Big\{\big((k,rl_X(k)+1,l,|Y|+1),(\mathrm{k}_X(rl_X(k)+1),rl_X(k)+1,1,|Y|+1)\big)\Big|rl_X(k)+1\leq|X|\Big\}$$
$$\Big\{\big((k,|X|+1,l,rl_Y(l)+1),(1,|X|+1,\mathrm{k}_Y(rl_Y(l)+1),rl_Y(l)+1)\big)\Big|rl_Y(l)+1\leq|Y|\Big\}$$

Further, we define a *path* between two nodes $u,v\in V$ as a sequence of nodes $v_0,\ldots,v_T$ where $v_0=u$, $v_T=v$, and for all $t\in\{1,\ldots,T\}$ it holds: $(v_{t-1},v_t)\in E$. If $T=0$, we call a path *trivial*.

We call a path between $(1,1,1,1)$ and $(1,|X|+1,1,|Y|+1)$ a path *through* $\mathcal{G}_{c,X,Y}$.

We call a node $v\in V$ *reachable* from another node $u\in V$, if a path from u to v exists.

As an example, consider the cooptimal edit graph between the trees $\tilde{x}=\mathtt{a}(\mathtt{b}(\mathtt{c},\mathtt{d}),\mathtt{e})$ and $\tilde{y}=\mathtt{f}(\mathtt{g})$ in Figure 2.4. An excerpt of this graph is shown in Figure A.2.

A key property of this graph is that, from any node, $(1,|X|+1,1,|Y|+1)$ is reachable. We will require this property to prove the correctness of Algorithms A.1 and A.2 later on.

Lemma A.7. *Let X and Y be forests over some alphabet $\mathcal{A}$, let c be a cost function over $\mathcal{A}$, and let $\mathcal{G}_{c,X,Y}$ be the cooptimal edit graph between X and Y according to c. Then it holds: From any node in $\mathcal{G}_{c,X,Y}$, $(1,|X|+1,1,|Y|+1)$ is reachable.*

Proof. First, consider the cases in which one of the forests is empty.

If $X=Y=\epsilon$, only the node $(1,1,1,1)=(1,|X|+1,1,|Y|+1)$ exists. As any node is reachable from itself via a trivial path, the claim holds.

If $X=\epsilon$ but $Y\neq\epsilon$, then $(1,1,1,1),(1,1,1,2),\ldots,(1,1,1,|Y|+1)=(1,|X|+1,1,|Y|+1)$ is a valid path in $\mathcal{G}_{c,X,Y}$ which connects any node in $\mathcal{G}_{c,X,Y}$ to $(1,|X|+1,1,|Y|+1)$. Therefore, the claim holds.

If $X\neq\epsilon$ but $Y=\epsilon$, then $(1,1,1,1),(1,2,1,1),\ldots,(1,|X|+1,1,1)=(1,|X|+1,1,|Y|+1)$ is a valid path in $\mathcal{G}_{c,X,Y}$ which connects any node in $\mathcal{G}_{c,X,Y}$ to $(1,|X|+1,1,|Y|+1)$. Therefore, the claim holds.

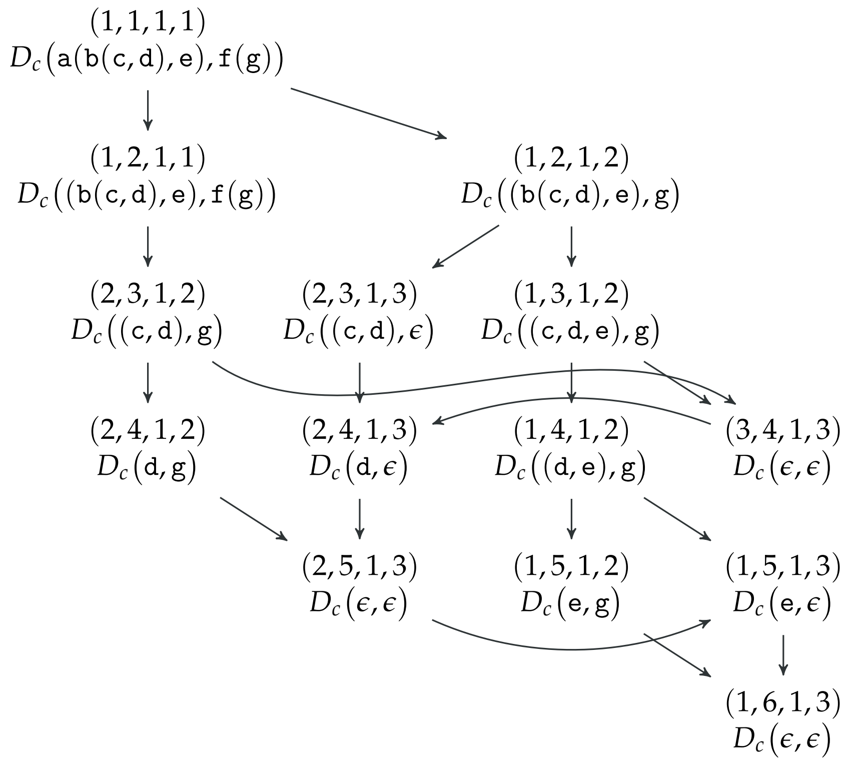


Figure A.2: An excerpt of the cooptimal edit graph between the trees $\tilde{x}$ and $\tilde{y}$ from Figure 2.4. The figure only shows nodes which are reachable from $(1,1,1,1)$. Further, to support clarity, the nodes are labelled with indices *and* with the corresponding subforest edit distance.

Finally, if neither X nor Y is empty, consider the following argument. Let $u = (k, i, l, j)$ and $v = (k', i', l', j')$ be two nodes in $\mathcal{G}_{c,X,Y}$. We define the binary relation $\succ$ as follows. $u \succ v$ if and only if $i > i'$, or $i = i'$ and $j > j'$, or $i = i'$ and $j = j'$ and $k < k'$, or $i = i'$ and $j = j'$ and $k = k'$ and $l < l'$. In other words, $\succ$ is equivalent to the lexicographic ordering on $(i, j, -k, -l')$.

Now, note that this binary relation is antisymmetric and transitive and that for any two nodes u and v always either $u \succ v$ or $v \succ u$ holds except if $u = v$. Further note that $(1, |X| + 1, 1, |Y| + 1)$ is the maximum according to this relation, meaning that for any other node u it holds: $(1, |X| + 1, 1, |Y| + 1) \succ u$.

Turning back to the cooptimal edit graph, we observe that for any two nodes u and v, $(u, v) \in E$ implies that $v \succ u$. Finally, we observe that any node except $(1, |X| + 1, 1, |Y| + 1)$ has at least one outgoing edge. Therefore, it must be the case that, from any node, we can continue a path with strictly growing nodes according to the relation $\succ$, until we reach the maximum, that is, $(1, |X| + 1, 1, |Y| + 1)$. Therefore, from all nodes in the graph, $(1, |X| + 1, 1, |Y| + 1)$ is reachable and the claim holds. □

We can check this theorem exemplarly in Figure A.2.

Another important property we will require for inductive proofs over the cooptimal edit graph is the relation between an entire cooptimal edit graph and the cooptimal edit graph for subforests. To establish such properties, we first introduce a lemma regarding the relationship between forests and subforests as such.

Lemma A.8. *Let $X = x(X_1)X_2$ be a forest over some alphabet $\mathcal{A}$ with $|X| > 0$, and let $X' := X[2, |X|]$.*

Then, for any $k \in \mathcal{K}(X)$, and any $i \in \{k, \ldots, rl_X(k) + 1\}$ with $i > 1$ it holds:

$$\tilde{x}^i = (\tilde{x}')^{i-1} \tag{A.62}$$
$$rl_X(i) = rl_{X'}(i-1) + 1 \tag{A.63}$$
$$\mathcal{K}(X') = \{\mathrm{k}_{X'}(|X_1|)\} \cup \{k - 1 | k > 1 \wedge k \in \mathcal{K}(X)\} \tag{A.64}$$
$$X[i, rl_X(k)] = X'[i-1, rl_{X'}(k')] \quad \text{where} \tag{A.65}$$
$$k' := \begin{cases} \mathrm{k}_{X'}(|X_1|) & \text{if } k = 1 \\ k - 1 & \text{otherwise} \end{cases}$$

Proof. We prove the single claims in turn. Per definition of the pre-order, the first claim follows directly.

The second claim follows by applying the definition of outermost right leaves, that is, $rl_X(i) = i + |\tilde{x}^i| - 1 = i + |(\tilde{x}')^{i-1}| - 1 = i - 1 + |(\tilde{x}')^{i-1}| - 1 + 1 = rl_{X'}(i-1) + 1$.

We prove the third claim by contradiction. Let $k \in \mathcal{K}(X')$ but not in $\{\mathrm{k}_{X'}(|X_1|)\} \cup \{k - 1 | k > 1 \wedge k \in \mathcal{K}(X)\}$. Then, in particular, $k + 1 \notin \mathcal{K}(X)$. Therefore, there exists some $k' \in \mathcal{K}(X)$ such that $k' < k + 1$ and $rl_X(k') = rl_X(k+1)$. Now, consider the case $k' = 1$. Then, it holds: $rl_X(k') = |X_1| + 1$. Further, by virtue of the second claim, we obtain: $rl_{X'}(k) = rl_X(k+1) - 1 = rl_X(k') - 1 = |X_1|$. Therefore, $k = \mathrm{k}_{X'}(|X_1|)$, which is a contradiction.

Now, consider the case $k' > 1$. Then, by virtue of the second claim, we obtain: $rl_{X'}(k'-1) = rl_X(k') - 1 = rl_X(k+1) - 1 = rl_{X'}(k)$. Further, we know that $k' - 1 < k$. Therefore, $k \notin \mathcal{K}(X')$, which is a contradiction.

Regarding the last claim, we observe that, per definition of subforests, it holds: $X[2, |X|] = X_1 X_2$, which, in turn, implies:

$$X[i, rl_X(k)] = (X_1 X_2)[i-1, rl_X(k) - 1] = X'[i-1, rl_X(k) - 1]$$

It remains to show that $rl_X(k) - 1 = rl_{X'}(k')$. In case $k > 1$ it holds: $k' = k - 1$. By virtue of the second claim we obtain $rl_X(k) - 1 = rl_{X'}(k')$.

Now, consider the case $k = 1$. In that case, we obtain $rl_X(1) - 1 = |X_1| = rl_{X'}(|X_1|) = rl_{X'}(\mathrm{k}_{X'}(|X_1|))$.

□

This lemma has an important implication regarding the structure of the cooptimal edit graph for subforests. In particular, we obtain the following result.

Lemma A.9. *Let X and Y be non-empty forests over some alphabet $\mathcal{A}$, let $X' := X[2, |X|]$, let $Y' := Y[2, |Y|]$, and let c be a cost function over $\mathcal{A}$. Further, let $\mathcal{G}_{c,X,Y} = (V, E)$. Then, it holds:*

1. *Let $\mathcal{G}_{c,X',Y} = (V', E')$. Further, for any $u = (k, i, l, j)$ let $u' = (k', i-1, l, j)$ with k' defined as in the previous lemma. Then, for any u with $i > 1$ it holds: $u \in V \iff u' \in V'$. Further, for any (u, v) it holds: $(u, v) \in E \iff (u', v') \in E'$.*

2. *Let $\mathcal{G}_{c,X,Y'} = (V', E')$. Further, for any $u = (k, i, l, j)$ let $u' = (k, i, l', j-1)$ with l' defined as in the previous lemma. Then, for any u with $j > 1$ it holds: $u \in V \iff u' \in V'$. Further, for any (u, v) it holds: $(u, v) \in E \iff (u', v') \in E'$.*

3. *Let $\mathcal{G}_{c,X',Y'} = (V', E')$. Further, for any $u = (k, i, l, j)$ let $u' = (k', i-1, l', j-1)$ with k', l' defined as in the previous lemma. Then, for any u with $i > 1$ and $j > 1$ it holds: $u \in V \iff u' \in V'$. Further, for any (u, v) it holds: $(u, v) \in E \iff (u', v') \in E'$.*

Proof. First, we conclude from the third claim in the previous lemma that all claims regarding nodes hold.

Second, we conclude from the last claim in the previous lemma:

$$D_c\Big(X'[i-1, rl_{X'}(k')], Y[j, rl_Y(l)]\Big) = D_c\Big(X[i, rl_X(k)], Y[j, rl_Y(l)]\Big)$$
$$D_c\Big(X[i, rl_X(k)], Y'[j-1, rl_{Y'}(l')]\Big) = D_c\Big(X[i, rl_X(k)], Y[j, rl_Y(l)]\Big)$$
$$D_c\Big(X'[i-1, rl_{X'}(k')], Y'[j-1, rl_{Y'}(l')]\Big) = D_c\Big(X[i, rl_X(k)], Y[j, rl_Y(l)]\Big)$$

Therefore, the edge conditions on for the subforest cooptimal edit graphs are equivalent to the respective edge conditions on the overall cooptimal edit graph. From the second claim in the previous lemma we can also conclude that the outermost right leaf structure is equivalent, which concludes the proof. □

Based on this lemma we can show that the cooptimal edit graph does indeed capture cooptimal tree mappings. For that purpose, we define the tree mapping corresponding to a path.

Definition A.7 (path mapping)**.** Let X and Y be forests over some alphabet $\mathcal{A}$, let c be a cost function over $\mathcal{A}$, and let $\mathcal{G}_{c,X,Y} = (V, E)$ be the cooptimal edit graph with respect to X, Y, and c. Further, let $p = v_0, \ldots, v_T$ be a path in $\mathcal{G}_{c,X,Y}$. Then, we define the path tree mapping M_p for path p as follows.

$$M_p = \Big\{(i,j)\Big|\exists t \in \{1, \ldots, T\}, k, k', l, l' : \quad v_{t-1} = (k, i, l, j) \wedge v_t = (k', i+1, l', j+1)\Big\} \tag{A.66}$$

As an example, consider the cooptimal edit graph in Figure A.2. A path through this graph would be $p = (1,1,1,1), (1,2,1,2), (1,3,1,2), (3,4,1,3), (2,4,1,3), (2,5,1,3), (1,5,1,3), (1,6,1,3)$. The corresponding tree mapping would be $M_p = \{(1,1), (3,2)\}$.

An important property of such tree mappings is that they decompose along the path, that is, if we cut the path in two parts at any points, the corresponding path mapping also decomposes into two parts.

Lemma A.10. *Let X and Y be forests over some alphabet $\mathcal{A}$, let c be a cost function over $\mathcal{A}$, and let $\mathcal{G}_{c,X,Y} = (V, E)$ be the cooptimal edit graph with respect to X, Y, and c. Further, let $p = v_0, \ldots, v_T$ be a path in $\mathcal{G}_{c,X,Y}$.*

Then, for any $t \in \{1, \ldots, T\}$ with $v_{t-1} = (k, i, l, j)$ and $v_t = (k', i+1, l', j+1)$ for some k, k', l, l', i, j it holds:

$$M_p = M_{v_0,\ldots,v_{t-1}} \cup \{(i,j)\} \cup M_{v_t,\ldots,v_T}$$

where the union is disjoint. For all other t it holds:

$$M_p = M_{v_0,\ldots,v_{t-1}} \cup M_{v_t,\ldots,v_T}$$

where the union is disjoint.

Proof. The result follows directly from the definition of the path tree mapping above. □

Note that the example tree mapping above is one of the cooptimal tree mappings shown in Figure 4.1. This is no coincidence. Indeed, it holds generally that the path tree mapping for any path through the cooptimal edit graph is cooptimal, and that any cooptimal tree mapping has a corresponding path through the cooptimal edit graph.

Lemma A.11. *Let X and Y be forests over some alphabet $\mathcal{A}$, let c be a cost function over $\mathcal{A}$, and let $\mathcal{G}_{c,X,Y} = (V, E)$ be the cooptimal edit graph with respect to X, Y, and c. Then it holds:*

1. *For all paths p through $\mathcal{G}_{c,X,Y}$, the tree mapping M_p is in $\mathcal{M}(c, X, Y)$.*
2. *For all $M \in \mathcal{M}(c, X, Y)$ it holds: There exists a path p through $\mathcal{G}_{c,X,Y}$ such that $M_p = M$.*

Proof. We start by considering the trivial cases of empty forests. If $X = \epsilon$ or $Y = \epsilon$, the only possible tree mapping is $M = \emptyset$. It remains to show that, in these cases, there is only one possible path p through $\mathcal{G}_{c,X,Y}$ and that for this path it holds $M_p = \emptyset$. Let $(V, E) := \mathcal{G}_{c,X,Y}$ and consider the following cases:

$X = \epsilon$ **and** $Y = \epsilon$**:** In this case, we obtain $V = \{(1, 1, 1, 1)\}$ and $E = \emptyset$. Accordingly, the trivial path $p = (1, 1, 1, 1)$ is the only possible path through $\mathcal{G}_{c,X,Y}$ and it does indeed hold $M_p = \emptyset$.

$X = \epsilon$ **and** $Y \neq \epsilon$**:** In this case, we obtain $V = \big\{(1, 1, 1, j)\big|j \in \{1, \ldots, |Y| + 1\}\big\}$ and $E = \big\{\big((1, 1, 1, j), (1, 1, 1, j + 1)\big)\big|j \in \{1, \ldots, |Y|\}\big\}$. Accordingly, the only possible path through $\mathcal{G}_{c,X,Y}$ is $p = (1, 1, 1, 1), (1, 1, 1, 2), \ldots, (1, 1, 1, |Y| + 1)$. And indeed it holds $M_p = \emptyset$.

$X \neq \epsilon$ **and** $Y = \epsilon$**:** In this case, we obtain $V = \big\{(1, i, 1, 1)\big|i \in \{1, \ldots, |X| + 1\}\big\}$ and $E = \big\{\big((1, i, 1, 1), (1, i + 1, 1, 1)\big)\big|i \in \{1, \ldots, |X|\}\big\}$. Accordingly, the only possible path through $\mathcal{G}_{c,X,Y}$ is $p = (1, 1, 1, 1), (1, 2, 1, 1), \ldots, (1, |X| + 1, 1, 1)$. And indeed it holds $M_p = \emptyset$.

It remains to show both claims for the case of non-empty forests. For both claims, we apply an induction over $|X| + |Y|$. We have already covered the base cases of empty forests, so consider now $|X|$ and $|Y|$ to be larger than 0.

Regarding the first claim, let $p = v_0, \ldots, v_T$ be a path through $\mathcal{G}_{c,X,Y}$, let $X' := X[2, |X|]$, let $Y' := Y[2, |Y|]$, and consider the following cases regarding v_1.

$v_1 = (1, 2, 1, 1)$**:** In this case, it must hold $D_c(X, Y) = c(x_1, -) + D_c(X', Y)$, otherwise $(v_0, v_1) \notin E$. Now, if X' is empty, then p must have the form $p = (1, 1, 1, 1), (1, 2, 1, 1), (1, 2, 1, 2), \ldots, (1, 2, 1, |Y| + 1)$, and $\emptyset$ must be a cooptimal tree mapping between X' and Y. Accordingly, $\emptyset = M_p$ must also be a cooptimal tree mapping between X and Y, because $c(\emptyset, X, Y) = c(x_1, -) + c(\emptyset, X', Y) = c(x_1, -) + D_c(X', Y) = D_c(X, Y)$.

If X' is *not* empty, then the first result in Lemma A.9 tells us that $p' := v'_1, \ldots, v'_T$ with v'_t constructed as in the lemma, is a path through $\mathcal{G}_{c,X',Y}$. Accordingly, by virtue of our induction hypothesis, $M_{p'}$ is a cooptimal tree mapping between X' and Y. Further, we obtain per construction $M_p = \{(i+1,j)|(i,j) \in M_{p'}\}$. Accordingly, it holds: $c(M_p, X, Y) = c(x_1, -) + c(M_{p'}, X', Y) = c(x_1, -) + D_c(X', Y) = D_c(X, Y)$, which means that M_p is cooptimal, as claimed.

$v_1 = (1,1,1,2)$: In this case, it must hold $D_c(X,Y) = c(-, y_1) + D_c(X, Y')$, otherwise $(v_0, v_1) \notin E$. Now, if Y' is empty, then p must have the form $p = (1,1,1,1), (1,1,1,2), (1,2,1,2), \ldots, (1, |X|+1, 1, 2)$, and $\emptyset$ must be a cooptimal tree mapping between X and Y'. Accordingly, $\emptyset = M_p$ must also be a cooptimal tree mapping between X and Y, because $c(\emptyset, X, Y) = c(-, y_1) + c(\emptyset, X, Y') = c(-, y_1) + D_c(X, Y') = D_c(X, Y)$.

If Y' is *not* empty, then the second result in Lemma A.9 tells us that $p' := v'_1, \ldots, v'_T$ with v'_t constructed as in the lemma, is a path through $\mathcal{G}_{c,X,Y'}$. Accordingly, by virtue of our induction hypothesis, $M_{p'}$ is a cooptimal tree mapping between X and Y'. Further, we obtain per construction $M_p = \{(i,j+1)|(i,j) \in M_{p'}\}$. Accordingly, it holds: $c(M_p, X, Y) = c(-, y_1) + c(M_{p'}, X, Y') = c(-, y_1) + D_c(X, Y') = D_c(X, Y)$, which means that M_p is cooptimal, as claimed.

$v_1 = (1,2,1,2)$: In this case, it must hold $D_c(X,Y) = c(x_1, y_1) + D_c(X', Y')$. Now, if X' is empty, then p must have the form $p = (1,1,1,1), (1,2,1,2), \ldots, (1,2,1,|Y|+1)$, and $\emptyset$ must be a cooptimal tree mapping between X' and Y'. Accordingly, $\{(1,1)\} = M_p$ must also be a cooptimal tree mapping between X and Y, because $c(\{(1,1)\}, X, Y) = c(x_1, y_1) + c(\emptyset, X', Y') = c(x_1, y_1) + D_c(X', Y') = D_c(X, Y)$.

If Y' is empty, then p must have the form $p = (1,1,1,1), (1,2,1,2), \ldots, (1, |X| + 1, 1, 2)$, and $\emptyset$ must be a cooptimal tree mapping between X and Y'. Accordingly, $\{(1,1)\} = M_p$ must also be a cooptimal tree mapping between X and Y, because $c(\{(1,1)\}, X, Y) = c(x_1, y_1) + c(\emptyset, X', Y') = c(x_1, y_1) + D_c(X', Y') = D_c(X, Y)$.

If neither X' nor Y' are empty, then the third result in Lemma A.9 tells us that $p' := v'_0, \ldots, v'_T$ with v'_t constructed as in the lemma, is a path through $\mathcal{G}_{c,X',Y'}$. Accordingly, by virtue of our induction hypothesis, $M_{p'}$ is a cooptimal tree mapping between X' and Y'. Further, we obtain per construction $M_p = \{(1,1)\} \cup \{(i+1, j+1)|(i,j) \in M_{p'}\}$. Accordingly, it holds: $c(M_p, X, Y) = c(x_1, y_1) + c(M_{p'}, X', Y') = c(x_1, y_1) + D_c(X', Y') = D_c(X, Y)$, which means that M_p is cooptimal, as claimed.

Other cases can not occur such that our induction is concluded.

Regarding the second claim, let $M \in \mathcal{M}(c, X, Y)$, i.e. $c(M, X, Y) = D_c(X, Y)$, and distinguish the following cases.

$1 \in I(M, X, Y)$: In this case it holds $c(M, X, Y) = c(x_1, -) + c(M', X', Y)$ with $M' = \{(i-1, j)|(i,j) \in M\}$. It must hold that $M' \in \mathcal{M}(c, X', Y)$. Otherwise, we would obtain $D_c(X, Y) \leq D_c(X', Y) + c(x_1, -) < c(M', X', Y) + c(x_1, -) = c(M, X, Y) = D_c(X, Y)$, which is a contradiction. This also implies that $D_c(X, Y) = c(x_1, -) + D_c(X', Y)$, which in turn implies that $\big((1,1,1,1), (1,2,1,1)\big) \in E$.

Now, if $X' = \epsilon$, M must be $\emptyset$, and we can construct the path $p = (1,1,1,1), (1,2,1,1), \ldots, (1,2,1,|Y|+1)$, which is a path through $\mathcal{G}_{c,X,Y}$ such that $M_p = \emptyset$.

If X' is not empty, our induction hypothesis implies that there exists a path p' through $\mathcal{G}_{c,X',Y}$ such that $M_{p'} = M'$. By virtue of the first result in Lemma A.9, we can construct an isomorphic path $\tilde{p}$ between $(1,2,1,1)$ and $(1,|X|+1,|Y|+1)$ in $\mathcal{G}_{c,X,Y}$. Accordingly, $p := (1,1,1,1), \tilde{p}$ must be a path through $\mathcal{G}_{c,X,Y}$, and per construction it must hold that $M_p = M$.

$1 \in J(M,X,Y)$**:** In this case it holds $c(M,X,Y) = c(-,y_1) + c(M',X,Y')$ with $M' = \{(i,j-1)|(i,j) \in M\}$. It must hold that $M' \in \mathcal{M}(c,X,Y')$. Otherwise, we would obtain $D_c(X,Y) \leq D_c(X,Y') + c(-,y_1) < c(M',X,Y') + c(-,y_1) = c(M,X,Y) = D_c(X,Y)$, which is a contradiction. This also implies that $D_c(X,Y) = c(-,y_1) + D_c(X,Y')$, which in turn implies that $\big((1,1,1,1),(1,1,1,2)\big) \in E$.

Now, if $Y' = \epsilon$, M must be $\emptyset$, and we can construct the path $p = (1,1,1,1),(1,1,1,2), \ldots, (1,|X|+1,1,2)$, which is a path through $\mathcal{G}_{c,X,Y}$ such that $M_p = \emptyset$.

If X' is not empty, our induction hypothesis implies that there exists a path p' through $\mathcal{G}_{c,X,Y'}$ such that $M_{p'} = M'$. By virtue of the second result in Lemma A.9, we can construct an isomorphic path $\tilde{p}$ between $(1,2,1,1)$ and $(1,|X|+1,|Y|+1)$ in $\mathcal{G}_{c,X,Y}$. Accordingly, $p := (1,1,1,1), \tilde{p}$ must be a path through $\mathcal{G}_{c,X,Y}$, and per construction it must hold that $M_p = M$.

$1 \in I^C(M,X,Y)$ **and** $1 \in J^C(M,X,Y)$**:** In this case, $(1,1) \in M$, which we can show as follows. Let $(1,j) \in M$ and $(i,1) \in M$. Now, consider the case $j > 1$. In that case, $i < 1$, which is impossible. Similarly, if $i > 1$, it must hold $j < 1$, which is impossible. Therefore $i = 1$ and $j = 1$.

In this case it holds $c(M,X,Y) = c(x_1,y_1) + c(M',X,Y')$ with $M' = \{(i-1,j-1)|(i,j) \in M \setminus \{(1,1)\}\}$. It must hold that $M' \in \mathcal{M}(c,X',Y')$. Otherwise, we would obtain $D_c(X,Y) \leq D_c(X',Y') + c(x_1,y_1) < c(M',X',Y') + c(x_1,y_1) = c(M,X,Y) = D_c(X,Y)$, which is a contradiction. This also implies that $D_c(X,Y) = c(x_1,y_1) + D_c(X',Y')$, which in turn implies that $\big((1,1,1,1),(1,2,1,2)\big) \in E$.

Now, if $X' = \epsilon$, M must be $\{(1,1)\}$, and we can construct the path $p = (1,1,1,1)$, $(1,2,1,2), \ldots, (1,2,1,|Y|+1)$, which is a path through $\mathcal{G}_{c,X,Y}$ such that $M_p = \{(1,1)\}$.

If $Y' = \epsilon$, M must be $\{(1,1)\}$, and we can construct the path $p = (1,1,1,1),(1,2,1,2), \ldots, (1,|X|+1,1,2)$, which is a path through $\mathcal{G}_{c,X,Y}$ such that $M_p = \{(1,1)\}$.

If neither X' nor Y' is empty, our induction hypothesis implies that there exists a path p' through $\mathcal{G}_{c,X',Y'}$ such that $M_{p'} = M'$. By virtue of the third result in Lemma A.9, we can construct an isomorphic path $\tilde{p}$ between $(1,2,1,2)$ and $(1,|X|+1,|Y|+1)$ in $\mathcal{G}_{c,X,Y}$. Accordingly, $p := (1,1,1,1), \tilde{p}$ must be a path through $\mathcal{G}_{c,X,Y}$, and per construction it must hold that $M_p = M$.

As no other cases can occur, this concludes the proof.

□

This lemma implies that we can replace the computation of cooptimal tree mappings with the computation of paths through the cooptimal edit graph. Therefore, we will limit our consideration mostly to paths from now on.

We continue by proving the correctness of Algorithm A.1. First, we establish an auxiliary statement regarding the order of executions in Algorithm A.1.

Lemma A.12. *Let $\tilde{x}$ and $\tilde{y}$ be trees over some alphabet $\mathcal{A}$, let c be a cost function over $\mathcal{A}$ which conforms to the triangular inequality, and let $\mathcal{G}_{c,\tilde{x},\tilde{y}} = (V,E)$ be the cooptimal edit graph with* *respect to $\tilde{x}$, $\tilde{y}$, and c. Further, we say that Algorithm A.1* visits *a node $(1, i, 1, j)$ if (i, j) is pulled from Q in line 6.*

Then, it holds: Algorithm A.1 visits all reachable nodes $(1, i, 1, j)$ from $(1, 1, 1, 1)$, and only those nodes, exactly once in lexicographic order.

Proof. First, to see that only reachable nodes from $(1, 1, 1, 1)$ are visited, observe that the algorithm starts at the tuple $(1, 1)$ and then only adds tuples (i', j') during the visit of $(1, i, 1, j)$ if an edge from $(1, i, 1, j)$ to $(1, i', 1, j')$ exists. Therefore, all visited nodes must be reachable.

Regarding the inverse claim, that all reachable nodes are visited in lexicographic ascending order, we perform an induction over the nodes of $\mathcal{G}_{c,\tilde{x},\tilde{y}} = (V, E)$ in lexicographic order. The first node in lexicographic order is $(1, 1, 1, 1)$, which is indeed visited first by Algorithm A.1.

Now, assume that the claim holds for all nodes $v \in V$ with $v \leq u$ for some u, and let $v = (1, i, 1, j)$ be the lexicographically smallest node larger than u which is reachable from $(1, 1, 1, 1)$. Because v is reachable, there must exist a path $p = v_0, \ldots, v_T$ from $(1, 1, 1, 1)$ to v. Now, consider the following cases regarding $v' := v_{T-1}$.

$v' = (1, i-1, 1, j)$: Then, $v' < v$ and v' is reachable from $(1, 1, 1, 1)$. Therefore, per induction, v' has been visited before. Further, because $(v', v) \in E$ it must hold that $\boldsymbol{D}_{i-1,j} = D_c(\tilde{x}[i-1, rl_{\tilde{x}}(1)], \tilde{y}[j, rl_{\tilde{y}}(1)]) = c(x_{i-1}, -) + D_c(\tilde{x}[i, rl_{\tilde{x}}(1)], \tilde{y}[j, rl_{\tilde{y}}(1)]) = c(x_{i-1}, -) + \boldsymbol{D}_{i,j}$. Therefore, line 11 has been executed during the visit of v' and thus (i, j) has been added to Q. Per induction, all reachable nodes smaller than v have been visited before v, and therefore v is the minimum in Q and is visited next.

$v' = (1, i, 1, j-1)$: Then, $v' < v$ and v' is reachable from $(1, 1, 1, 1)$. Therefore, per induction, v' has been visited before. Further, because $(v', v) \in E$ it must hold that $\boldsymbol{D}_{i,j-1} = D_c(\tilde{x}[i, rl_{\tilde{x}}(1)], \tilde{y}[j-1, rl_{\tilde{y}}(1)]) = c(-, y_{j-1}) + D_c(\tilde{x}[i, rl_{\tilde{x}}(1)], \tilde{y}[j, rl_{\tilde{y}}(1)]) = c(-, y_{j-1}) + \boldsymbol{D}_{i,j}$. Therefore, line 15 has been executed during the visit of v' and thus (i, j) has been added to Q. Per induction, all reachable nodes smaller than v have been visited before v, and therefore v is the minimum in Q and is visited next.

$v' = (1, i-1, 1, j-1)$: Then, $v' < v$ and v' is reachable from $(1, 1, 1, 1)$. Therefore, per induction, v' has been visited before. Further, because $(v', v) \in E$ it must hold that $\boldsymbol{D}_{i-1,j-1} = D_c(\tilde{x}[i-1, rl_{\tilde{x}}(1)], \tilde{y}[j-1, rl_{\tilde{y}}(1)]) = c(x_{i-1}, y_{j-1}) + D_c(\tilde{x}[i, rl_{\tilde{x}}(1)], \tilde{y}[j, rl_{\tilde{y}}(1)]) = c(x_{i-1}, y_{j-1}) + \boldsymbol{D}_{i,j}$.

Now, consider two different cases. If $c(x_{i-1}, y_{j-1}) < c(x_{i-1}, -) + c(-, y_{j-1})$, then line 23 has been executed during the visit of v' and thus (i, j) has been added to Q. Per induction, all reachable nodes smaller than v have been visited before v, and therefore v is the minimum in Q and is visited next.

Otherwise, due to the triangular inequality we have $c(x_{i-1}, y_{j-1}) = c(x_{i-1}, -) + c(-, y_{j-1})$. This, in turn, implies $\boldsymbol{D}_{i-1,j-1} = c(x_{i-1}, -) + \boldsymbol{D}_{i,j-1}$. Otherwise, it would hold that $\boldsymbol{D}_{i-1,j-1} < c(x_{i-1}, -) + \boldsymbol{D}_{i,j-1} \leq c(x_{i-1}, -) + c(-, y_{j-1}) + \boldsymbol{D}_{i,j} = c(x_{i-1}, y_{j-1}) + \boldsymbol{D}_{i,j} = \boldsymbol{D}_{i-1,j-1}$ which is a contradiction. Now, let $v'' = (1, i, 1, j-1)$. Due to $\boldsymbol{D}_{i-1,j-1} = c(x_{i-1}, -) + \boldsymbol{D}_{i,j-1}$ we know that $(v', v'') \in E$, which implies that v'' is reachable. Further, because $v'' < v$, the induction hypothesis implies that v'' has been visited before. Finally, we also know that $\boldsymbol{D}_{i,j-1} = c(-, y_{j-1}) + \boldsymbol{D}_{i-1,j-1}$. Otherwise, we would obtain $\boldsymbol{D}_{i-1,j-1} = c(x_{i-1}, -) + \boldsymbol{D}_{i,j-1} < c(x_{i-1}, -) + c(-, y_{j-1}) +$

$\boldsymbol{D}_{i,j} = c(x_{i-1}, y_{j-1}) + \boldsymbol{D}_{i,j} = \boldsymbol{D}_{i-1,j-1}$, which is a contradiction. Therefore, line 15 has been executed during the visit of v'' and thus (i,j) has been added to Q. Per induction, all reachable nodes smaller than v have been visited before v, and therefore v is the minimum in Q and is visited next.

$v' = (k,i,l,j)$ **with** $rl_{\tilde{x}}(k) + 1 = i$ **and** $rl_{\tilde{y}}(l) + 1 = j$**:** Then, there must exist some $t < T - 1$ such that $v_t = (1,i',1,j')$ with $\mathrm{k}_{\tilde{x}}(i') = k$, $\mathrm{k}_{\tilde{y}}(j') = l$, $rl_{\tilde{x}}(k) \neq rl_{\tilde{x}}(1)$ or $rl_{\tilde{y}}(l) \neq rl_{\tilde{y}}(1)$, and $v_{t+1} = (k, i'+1, l, j'+1)$. Otherwise v' would not be reachable.

Because E only contains edges in non-descending lexicographic order of (i,j) and because $v_{t+1} > v_t$, it must also hold that $v > v_t$. Therefore, per induction hypothesis, v_t has been visited before. Further, because $(v_t, v_{t+1}) \in E$ it must hold that $\boldsymbol{D}_{i',j'} = D_c(\tilde{x}[i', rl_{\tilde{x}}(1)], \tilde{y}[j', rl_{\tilde{y}}(1)]) = D_c(\tilde{x}^{i'}, \tilde{y}^{j'}) + D_c(\tilde{x}[i, rl_{\tilde{x}}(1)], \tilde{y}[j, rl_{\tilde{y}}(1)]) = \boldsymbol{d}_{i',j'} + \boldsymbol{D}_{i,j}$.

Therefore, line 31 has been executed during the visit of v_t and thus (i,j) has been added to Q. Per induction, all reachable nodes smaller than v have been visited before v, and therefore v is the minimum in Q and is visited next.

This concludes the proof by induction.

□

By virtue of this lemma, we can now go on to prove the correctness of Algorithm A.1.

Lemma A.13. *Let $\tilde{x}$ and $\tilde{y}$ be trees over some alphabet $\mathcal{A}$, let c be a cost function over $\mathcal{A}$ which conforms to the triangular inequality, and let $\mathcal{G}_{c,\tilde{x},\tilde{y}} = (V,E)$ be the cooptimal edit graph with respect to $\tilde{x}$, $\tilde{y}$, and c. Then it holds:*

After Algorithm A.1 has been executed, C contains all nodes of the form $(1,i,1,j)$ which are reachable from $(1,1,1,1)$. Further, for all (i,j) it holds:

$$\boldsymbol{A}_{i,j} = |\{M_p | p \text{ is a path from } (1,1,1,1) \text{ to } (1,i,1,j) \text{ in } \mathcal{G}_{c,\tilde{x},\tilde{y}}\}|. \tag{A.67}$$

It also holds: $\boldsymbol{A}_{|\tilde{x}|+1,|\tilde{y}|+1} = |\{\mathcal{M}(c,\tilde{x},\tilde{y})\}|$.

Proof. The first claim follows directly from the previous lemma.

Regarding the second claim, consider an alternative version of Algorithm A.1. In this alternative version, we do not just count the number of cooptimal path tree mappings, but we accumulate these tree mappings themselves. In particular:

- We replace line 2 with "Initialize $\tilde{\boldsymbol{A}}$ as a $(|\tilde{x}|+1) \times (|\tilde{y}|+1)$ matrix of empty sets".
- We replace the first statement in line 3 with $\tilde{\boldsymbol{A}}_{1,1} \leftarrow \{\emptyset\}$.
- We replace line 10 with $\tilde{\boldsymbol{A}}_{i+1,j} \leftarrow \tilde{\boldsymbol{A}}_{i+1,j} \cup \tilde{\boldsymbol{A}}_{i,j}$.
- We replace line 14 with $\tilde{\boldsymbol{A}}_{i,j+1} \leftarrow \tilde{\boldsymbol{A}}_{i,j+1} \cup (\tilde{\boldsymbol{A}}_{i,j} \setminus \tilde{\boldsymbol{A}}_{i,j+1}) \cup \big\{\{(i-1,j)\} \cup M \big| M \in \tilde{\boldsymbol{A}}_{i,j} \cap \tilde{\boldsymbol{A}}_{i,j+1}\big\}$.
- We replace line 22 with $\tilde{\boldsymbol{A}}_{i+1,j+1} \leftarrow \tilde{\boldsymbol{A}}_{i+1,j+1} \cup \big\{\{(i,j)\} \cup M \big| M \in \tilde{\boldsymbol{A}}_{i,j}\big\}$.
- We replace line 30 with $\tilde{\boldsymbol{A}}_{rl_{\tilde{x}}(i)+1, rl_{\tilde{y}}(j)+1} \leftarrow \tilde{\boldsymbol{A}}_{rl_{\tilde{x}}(i)+1, rl_{\tilde{y}}(j)+1} \cup \big\{M \cup \{(i-1+i', j-1+j') | (i',j') \in M'\} \big| M \in \tilde{\boldsymbol{A}}_{i,j}, M' \in \tilde{\boldsymbol{A}}'_{|\tilde{x}^i|+1, |\tilde{y}^j|+1}\big\}$.

Our proof now works as follows. First, we will show that

$$\tilde{\boldsymbol{A}}_{i,j} = \{M_p | p \text{ is a path from } (1,1,1,1) \text{ to } (1,i,1,j) \text{ in } \mathcal{G}_{c,\tilde{x},\tilde{y}}\} \tag{A.68}$$

Then, we will show that $\boldsymbol{A}_{i,j} = |\tilde{\boldsymbol{A}}_{i,j}|$, which will conclude our proof.

We show Equation A.68 via induction over all entries (i,j) in lexicographic order. In case $i = j = 1$, we obtain $\tilde{\boldsymbol{A}}_{1,1} = \{\emptyset\}$. Indeed, the trivial path $p = (1,1,1,1)$ is the only path from $(1,1,1,1)$ to $(1,1,1,1)$ and the corresponding tree mapping M_p is $\emptyset$.

Now, consider some entry $(i,j) > (1,1)$ and assume that the claim holds for all $(i',j') < (i,j)$.

First, we show that for any $M \in \tilde{\boldsymbol{A}}_{i,j}$, there exists a path p from $(1,1,1,1)$ to $(1,i,1,j)$ such that $M_p = M$. We distinguish the following cases.

If M has been added to $\tilde{\boldsymbol{A}}_{i,j}$ via line 10, then $M \in \tilde{\boldsymbol{A}}_{i-1,j}$ and $\boldsymbol{D}_{i-1,j} = c(x_{i-1},-) + \boldsymbol{D}_{i,j}$. Further, per induction, there exists a path p' from $(1,1,1,1)$ to $(1,i-1,1,j)$ such that $M_{p'} = M$. Finally, due to $\boldsymbol{D}_{i-1,j} = c(x_{i-1},-) + \boldsymbol{D}_{i,j}$, we know that $\big((1,i-1,1,j),(1,i,1,j)\big) \in E$, such that $p := p',(1,i,1,j)$ is a path from $(1,1,1,1)$ to $(1,i,1,j)$ in $\mathcal{G}_{c,\tilde{x},\tilde{y}}$ and $M_p = M$.

If M has been added to $\tilde{\boldsymbol{A}}_{i,j}$ via line 14, then it must hold $\boldsymbol{D}_{i,j-1} = c(-,y_{j-1}) + \boldsymbol{D}_{i,j}$. Now, consider the case that $(i-1,j-1) \notin M$, that is, M is *not* a duplicate with a tree mapping that has been added before. Then, per induction, there exists a path p' from $(1,1,1,1)$ to $(1,i-1,1,j)$ such that $M_{p'} = M$. Finally, due to $\boldsymbol{D}_{i,j-1} = c(-,y_j) + \boldsymbol{D}_{i,j}$, we know that $\big((1,i,1,j-1),(1,i,1,j)\big) \in E$, such that $p := p',(1,i,1,j)$ is a path from $(1,1,1,1)$ to $(1,i,1,j)$ in $\mathcal{G}_{c,\tilde{x},\tilde{y}}$ and $M_p = M$.

Next, consider the case that $(i-1,j-1) \in M$, that is, $M' = M \setminus \{(i-1,j-1)\}$ *is* a duplicate with a tree mapping that has been added before. However, M' can not have been via line 22 because in that case $(i-1,j-1) \in M'$, which is a contradiction. Further, it can not have been via line 30 because in that case, line 30 would have to have been executed before during the visit of some entry $(1,k,1,l)$, such that $(k,l) \in M'$. However, in that case, there exists no path from $(1,1,1,1)$ to $(1,i,1,j-1)$ such that $(k,l) \in M_{p'}$, which would be a contradiction to our induction hypothesis. Therefore, the only option remaining is that M' has been added before via line 10, which implies that $\boldsymbol{D}_{i-1,j} = c(x_{i-1},-) + \boldsymbol{D}_{i,j}$.

Further, due to our induction hypothesis, there must exist two paths p' from $(1,1,1,1)$ to $(1,i,1,j-1)$ and $p'' = v_0,\ldots,v_T$ from $(1,1,1,1)$ to $(1,i-1,1,j)$, such that $M_{p''} = M_{p'} = M'$. Because of this latter constraint, $i-1 \in I(M',\tilde{x},\tilde{y})$ and $j-1 \in J(M',\tilde{x},\tilde{y})$, which means that there exists some $t < T$ such that $v_t = (1,k,1,j-1)$ for some $k \leq i-1$ and $v_{t+1} = (1,k,1,j)$, $v_{t+2} = (1,k+1,1,j)$, and so forth, until $v_T = (1,i-1,1,j)$. Due to this form of the path, we can further conclude that $\boldsymbol{D}_{k,j-1} = c(-,y_{j-1}) + \boldsymbol{D}_{k,j} = \ldots = c(-,y_{j-1}) + c(x_k,-) + \ldots + c(x_{i-1},-) + \boldsymbol{D}_{i,j}$. Due to $\boldsymbol{D}_{i,j-1} = c(-,y_{j-1}) + \boldsymbol{D}_{i,j}$, we can re-write this expression as $\boldsymbol{D}_{k,j-1} = c(x_k,-) + \ldots + c(x_{i-1},-) + \boldsymbol{D}_{i,j-1}$. This, in turn, implies that $\boldsymbol{D}_{i-1,j-1} = c(x_{i-1},-) + \boldsymbol{D}_{i,j-1}$. Otherwise, we would obtain $\boldsymbol{D}_{k,j-1} \leq c(x_k,-) + \ldots + c(x_{i-2},-) + \boldsymbol{D}_{i-1,j-1} < c(x_k,-) + \ldots + c(x_{i-1},-) + c(x_{i-1},-) + \boldsymbol{D}_{i,j-1} = \boldsymbol{D}_{k,j-1}$, which is a contradiction.

Furthermore, it holds $c(x_{i-1},-) + c(-,y_{j-1}) + \boldsymbol{D}_{i,j} = c(x_{i-1},-) + \boldsymbol{D}_{i,j-1} = \boldsymbol{D}_{i-1,j-1} \leq c(x_{i-1},y_{j-1}) + \boldsymbol{D}_{i,j}$ which implies $c(x_{i-1},-) + c(-,y_{j-1}) \leq c(x_{i-1},y_{j-1})$. In conjunction with the triangular inequality on c this gives us $c(x_{i-1},-) + c(-,y_{j-1}) =$

$c(x_{i-1}, y_{j-1})$, which in turn implies that $\big((1, i-1, 1, j-1), (1, i, 1, j)\big)$ is an edge in the graph. Finally, we can construct the path $p = v_0, \ldots, v_t, (1, k+1, 1, j-1), \ldots, (1, i-1, 1, j-1), (1, i, 1, j)$. For this path it holds per construction that $M_p = M$.

If M has been added via line 22, then $M \setminus \{(i-1, j-1)\} \in \tilde{\boldsymbol{A}}_{i-1,j-1}$, $\boldsymbol{D}_{i-1,j-1} = c(x_{i-1}, y_{j-1}) + \boldsymbol{D}_{i,j}$, $rl_{\tilde{x}}(i) = rl_{\tilde{x}}(1)$, and $rl_{\tilde{y}}(j) = rl_{\tilde{y}}(1)$. Further, per induction, there exists a path p' from $(1,1,1,1)$ to $(1, i-1, 1, j-1)$ such that $M_{p'} = M$. Finally, due to $\boldsymbol{D}_{i-1,j-1} = c(x_{i-1}, y_{j-1}) + \boldsymbol{D}_{i,j}$, we know that $\big((1, i-1, 1, j-1), (1, i, 1, j)\big) \in E$, such that $p := p', (1, i, 1, j)$ is a path from $(1,1,1,1)$ to $(1, i, 1, j)$ in $\mathcal{G}_{c,\tilde{x},\tilde{y}}$ and $M_p = M$.

Finally, if M has been added via line 30, there exist two indices k, l such that $rl_{\tilde{x}}(k) + 1 = i$, $rl_{\tilde{y}}(l) + 1 = j$, $i \leq |\tilde{x}|$ or $j \leq |\tilde{y}|$, $\boldsymbol{D}_{k,l} = \boldsymbol{d}_{k,l} + \boldsymbol{D}_{i,j}$, and M can be re-written as the disjoint union of two sets $\tilde{M}$ and $\tilde{M}'$, where $\tilde{M} \in \tilde{\boldsymbol{A}}_{k,l}$ and $M' := \{(i' - k + 1, j' - l + 1) | (i', j') \in \tilde{M}'\} \in \tilde{\boldsymbol{A}}'_{|\tilde{x}^k|+1, |\tilde{y}^l|+1}$. By virtue of our induction hypothesis, there exist two paths, $\tilde{p}$ from $(1,1,1,1)$ to $(1, k, 1, l)$ in $\mathcal{G}_{c,\tilde{x},\tilde{y}}$ and $p' = v_0, \ldots, v_T$ through $\mathcal{G}_{c,\tilde{x}^i,\tilde{y}^j}$ such that $M_{\tilde{p}} = \tilde{M}$ and $M_{p'} = M'$. Note that v_1 is necessarily $(1,2,1,2)$ because any other path has been blocked via line 29.

Further, because $\boldsymbol{D}_{k,l} = \boldsymbol{d}_{k,l} + \boldsymbol{D}_{i,j}$ and $i \leq |\tilde{x}|$ or $j \leq |\tilde{y}|$ we know that $\big((1, k, 1, l), (\mathrm{k}_{\tilde{x}}(k), k+1, \mathrm{k}_{\tilde{y}}(l), l+1)\big) \in E$ and due to the definition of $\mathcal{G}_{c,\tilde{x},\tilde{y}}$ we know that $\big((\mathrm{k}_{\tilde{x}}(k), i, \mathrm{k}_{\tilde{y}}(l), j), (1, i, 1, j)\big) \in E$.

Now, let $\tilde{p}' = \tilde{v}_1, \ldots, \tilde{v}_T$ with $\tilde{v}_t = (\tilde{k}, k + i' - 1, \tilde{l}, l + j' - 1)$ if and only if $v_t = (k', i', l', j')$ where $\tilde{k} := \mathrm{k}_{\tilde{x}}(k)$ if $k' = 1$ and $\tilde{k} := k' + k - 1$ otherwise, and where $\tilde{l} := \mathrm{k}_{\tilde{y}}(l)$ if $l' = 1$ and $\tilde{l} := l' + l - 1$ otherwise. This is a path in $\mathcal{G}_{c,\tilde{x},\tilde{y}}$ and it holds, per construction, $M_{\tilde{p}'} = \tilde{M}' \setminus \{(k, l)\}$. Accordingly, the concatenated path $p := \tilde{p}, \tilde{p}', (1, i, 1, j)$ is a path from $(1,1,1,1)$ to $(1, i, 1, j)$ in $\mathcal{G}_{c,\tilde{x},\tilde{y}}$ and $M_p = M$.

Now, it remains to show that for any path p from $(1,1,1,1)$ to $(1, i, 1, j)$, M_p is in $\tilde{\boldsymbol{A}}_{i,j}$. Let $p = v_0, \ldots, v_T$, $p' = v_0, \ldots, v_{T-1}$ and $v' = v_{T-1}$. Then, consider the following cases for v'.

$v' = (1, i-1, 1, j)$: Then, $\boldsymbol{D}_{i-1,j} = c(x_{i-1}, -) + \boldsymbol{D}_{i,j}$ and line 10 has been executed during the visit of v'. Further, due to the induction hypothesis, we know that $M_{p'} \in \tilde{\boldsymbol{A}}_{i-1,j}$, and thus $M_{p'}$ will now be added to $\tilde{\boldsymbol{A}}_{i,j}$. Finally, because $M_p = M_{p'}$, it follows that $M_p \in \boldsymbol{A}_{i,j}$.

$v' = (1, i, 1, j-1)$: Then, $\boldsymbol{D}_{i,j-1} = c(-, y_{j-1}) + \boldsymbol{D}_{i,j}$ and line 14 has been executed during the visit of v'. Further, due to the induction hypothesis, we know that $M_{p'} \in \tilde{\boldsymbol{A}}_{i,j-1}$, and thus $M_{p'}$ will now be added to $\tilde{\boldsymbol{A}}_{i,j}$ (or is already element of this set). Finally, because $M_p = M_{p'}$, it follows that $M_p \in \boldsymbol{A}_{i,j}$.

$v' = (1, i-1, 1, j-1)$: Then, $\boldsymbol{D}_{i-1,j-1} = c(x_{i-1}, y_{j-1}) + \boldsymbol{D}_{i,j}$.

Now, consider two different cases. If $c(x_{i-1}, y_{j-1}) < c(x_{i-1}, -) + c(-, y_{j-1})$, then $rl_{\tilde{x}}(i-1) = rl_{\tilde{x}}(1)$ and $rl_{\tilde{y}}(j-1) = rl_{\tilde{y}}(1)$, otherwise $\big((1, i-1, 1, j-1), (1, i, 1, j)\big)$ would not be an edge in $\mathcal{G}_{c,\tilde{x},\tilde{y}}$. Therefore, line 20 has been executed during the visit of v'. Further, due to the induction hypothesis, we know that $M_{p'} \in \tilde{\boldsymbol{A}}_{i-1,j-1}$, and thus $M_{p'} \cup \{(i-1, j-1)\}$ will now be added to $\tilde{\boldsymbol{A}}_{i,j}$. Finally, because $M_p = M_{p'} \cup \{(i-1, j-1)\}$, it follows that $M_p \in \boldsymbol{A}_{i,j}$.

If $c(x_{i-1}, y_{j-1}) = c(x_{i-1}, -) + c(-, y_{j-1})$, we obtain $\boldsymbol{D}_{i-1,j-1} \leq c(x_{i-1}, -) + \boldsymbol{D}_{i,j-1} \leq c(x_{i-1}, -) + c(-, y_{j-1}) + \boldsymbol{D}_{i,j} = c(x_{i-1}, y_{j-1}) + \boldsymbol{D}_{i,j} = \boldsymbol{D}_{i-1,j-1}$ and $\boldsymbol{D}_{i-1,j-1} \leq c(-, y_{j-1})$

$+ \boldsymbol{D}_{i-1,j} \leq c(x_{i-1}, -) + c(-, y_{j-1}) + \boldsymbol{D}_{i,j} = c(x_{i-1}, y_{j-1}) + \boldsymbol{D}_{i,j} = \boldsymbol{D}_{i-1,j-1}$, which in turn implies $\boldsymbol{D}_{i-1,j-1} = c(x_{i-1}, -) + \boldsymbol{D}_{i,j-1}$, $\boldsymbol{D}_{i-1,j-1} = c(-, y_{j-1}) + \boldsymbol{D}_{i-1,j}$, $\boldsymbol{D}_{i,j-1} = c(-, y_{j-1}) + \boldsymbol{D}_{i,j}$, and $\boldsymbol{D}_{i-1,j} = c(x_{i-1}, -) + \boldsymbol{D}_{i,j}$.

Therefore, $(1, i-1, 1, j)$ is reachable via the path $p^l = p', (1, i-1, 1, j)$ and line 10 has been executed while visiting $(1, i-1, 1, j)$. Further, due to the induction hypothesis, we know that $M_{p^l} \in \tilde{\boldsymbol{A}}_{i-1,j}$ and thus $M_{p^l} \in \tilde{\boldsymbol{A}}_{i,j}$.

We also know that $(1, i, 1, j-1)$ is reachable via the path $p^r = p', (1, i, 1, j-1)$ and that line 14 has been executed while visiting $(1, i, 1, j-1)$. Further, due to the induction hypothesis, we know that $M_{p^r} \in \tilde{\boldsymbol{A}}_{i,j-1}$. Now, note that $M_{p'} = M_{p^l} = M_{p^r}$. Therefore, at the execution time of line 14 while visiting $(1, i, 1, j-1)$, M_{p^r} is already element of $\tilde{\boldsymbol{A}}_{i,j}$. Therefore, line 14 adds $M_{p^r} \cup \{(i-1, j-1)\}$ to $\tilde{\boldsymbol{A}}_{i,j}$. Because $M_p = M_{p'} \cup \{(i-1, j-1)\} = M_{p^r} \cup \{(i-1, j-1)\}$, this implies that $M_p \in \tilde{\boldsymbol{A}}_{i,j}$.

$v' = (k, i, l, j)$ **with** $rl_{\tilde{x}}(k) + 1 = i$ **and** $rl_{\tilde{y}}(l) + 1 = j$**:** Then, there must exist some $t < T - 1$ and some i', j', such that $v_t = (1, i', 1, j')$ with $\mathrm{k}_{\tilde{x}}(i') = k$, $\mathrm{k}_{\tilde{y}}(j') = l$, $rl_{\tilde{x}}(k) \neq rl_{\tilde{x}}(1)$ or $rl_{\tilde{y}}(l) \neq rl_{\tilde{y}}(1)$, $v_{t+1} = (k, i'+1, l, j'+1)$, and $c(x_{i'}, y_{j'}) < c(x_{i'}, -) + c(-, y_{j'})$. Otherwise, v' would not be reachable. Therefore, lines 27-31 get executed during the visit of $(1, i', 1, j')$.

Now, let $\tilde{p} = v_0, \ldots, v_t$, let $\tilde{p}^* = v_{t+1}, \ldots, v_{T-1}$, and let $p^* = (1, 1, 1, 1), \tilde{v}_{t+1}, \ldots, \tilde{v}_{T-1}$ with $\tilde{v}_{t'} = (\tilde{k}, i'' - i' + 1, \tilde{l}, j'' - j' + 1)$ if and only if $v_{t'} = (k', i'', l', j'')$ where $\tilde{k} := 1$ if $k' = k$ and $\tilde{k} := k' - i' + 1$ otherwise and where $\tilde{l} := 1$ if $l' = l$ and $\tilde{l} := l' - j' + 1$ otherwise. Per induction hypothesis, $M_{\tilde{p}} \in \tilde{\boldsymbol{A}}_{i',j'}$ and $M_{p^*} \in \tilde{\boldsymbol{A}}'_{|\tilde{x}^{i'}|+1, |\tilde{y}^{j'}|+1}$. Finally, note that $M_p = M_{\tilde{p}} \cup \{(i', j')\} \cup M_{\tilde{p}^*}$. Therefore, M_p gets added to $\tilde{\boldsymbol{A}}_{i,j}$ during the execution of line 30 during the visit of $(1, i', 1, j')$.

This concludes the proof of Equation A.68. Now, it remains to show that, for all $(1, i, 1, j)$ which are reachable from $(1, 1, 1, 1)$ it holds: $\boldsymbol{A}_{i,j} = |\tilde{\boldsymbol{A}}_{i,j}|$.

First, observe that $\boldsymbol{A}_{1,1} = 1 = |\{\emptyset\}| = |\tilde{\boldsymbol{A}}_{i,j}|$.

Second, we note that, whenever line 10 is executed, none of the tree mappings in $\tilde{\boldsymbol{A}}_{i,j}$ are in $\tilde{\boldsymbol{A}}_{i+1,j}$ yet. Otherwise, these tree mappings would have to have been added via another line. However, line 14 can not yet have been executed with $(i+1, j-1)$ due to lexicographic ordering, line 22 only adds tree mappings which contain the element $(i, j-1)$, which is not contained in any tree mapping in $\boldsymbol{A}_{i,j}$, and line 30 only adds tree mappings which contain some element (k, l) for which $rl_{\tilde{x}}(k) \neq rl_{\tilde{x}}(1)$ or $rl_{\tilde{y}}(l) \neq rl_{\tilde{y}}(1)$, which is not contained in any tree mapping in $\boldsymbol{A}_{i,j}$. Therefore, the union operation in the changed line 10 is disjoint, which in turn implies that $|\tilde{\boldsymbol{A}}_{i+1,j}| = |\tilde{\boldsymbol{A}}_{i+1,j}| + |\tilde{\boldsymbol{A}}_{i,j}|$, which yields the original line 10 in Algorithm A.1.

Third, we show that the set unions in the modified line 14 are also disjoint. In particular, any tree mapping which does not contain $(i-1, j)$ is not added as a duplicate. Thus, it remains to show that all tree mappings of the type $M \cup \{(i-1, j)\}$ where $M \in \tilde{\boldsymbol{A}}_{i,j} \cap \boldsymbol{A}_{i,j+1}$ are not yet contained in $\boldsymbol{A}_{i,j+1}$. As we have shown above, the fact that the intersection $M \in \tilde{\boldsymbol{A}}_{i,j} \cap \boldsymbol{A}_{i,j+1}$ is not empty means that any elements in this intersection have been previously added via line 10, which in turn implies that $c(x_{i-1}, y_j) = c(x_{i-1}, -) + c(-, y_j)$. Therefore, $(i-1, j)$ could not have been added via line 22 or 31, because these lines are not executed for the entry $(i-1, j)$ due to the **continue** statement in line 18. Therefore, we obtain $|\tilde{\boldsymbol{A}}_{i,j+1}| = |\tilde{\boldsymbol{A}}_{i,j+1}| + |\tilde{\boldsymbol{A}}_{i,j}|$, which yields the original line 14 in Algorithm A.1.

Fourth, we show that the set union in the modified line 22 is disjoint. This follows from the fact that $(1, i, 1, j)$ is visited before $(1, i+1, 1, j)$ as well as $(1, i, 1, j+1)$, such that there can be no duplicates due to lines 10 or 14. Further, there can be no duplicates due to line 30 because the conditions for line 30 and line 22 are mutually exclusive. Therefore, we obtain $|\tilde{\boldsymbol{A}}_{i+1,j+1}| = |\tilde{\boldsymbol{A}}_{i+1,j+1}| + |\tilde{\boldsymbol{A}}_{i,j}|$, which yields the original line 22 in Algorithm A.1.

Finally, we show that the set union in the modified line 30 is disjoint. This follows from the fact that for any two indices k, l, $(1, k, 1, l)$ is visited before $(1, rl_{\tilde{x}}(k), 1, rl_{\tilde{y}}(l) + 1)$ as well as $(1, rl_{\tilde{x}}(k) + 1, 1, rl_{\tilde{y}}(l))$, such that there can be no duplicates due to lines 10 or 14. Further, there can be no duplicates due to line 22 because the conditions for line 22 and line 30 are mutually exclusive. Therefore, we obtain $|\tilde{\boldsymbol{A}}_{rl_{\tilde{x}}(k)+1, rl_{\tilde{y}}(l)+1}| = |\tilde{\boldsymbol{A}}_{rl_{\tilde{x}}(k)+1, rl_{\tilde{y}}(l)+1}| + |\tilde{\boldsymbol{A}}_{i,j}| \cdot |\tilde{\boldsymbol{A}}'_{|\tilde{x}^{i'}|+1, |\tilde{y}^{j'}|+1}|$, which yields the original line 30 in Algorithm A.1.

This concludes the proof of Equation A.67.

The second claim, $\boldsymbol{A}_{|\tilde{x}|+1,|\tilde{y}|+1} = |\{\mathcal{M}(c, \tilde{x}, \tilde{y})\}|$, follows from the first. By virtue of Lemma A.7 we know that $(1, |\tilde{x}| + 1, 1, |\tilde{y}| + 1)$ is reachable from $(1, 1, 1, 1)$. Therefore, Lemma A.12 tells us that it will be visited. As we have just shown, this implies that $\boldsymbol{A}_{|\tilde{x}|+1,|\tilde{y}|+1} = |\{M_p | p \text{ is a path through } \mathcal{G}_{c,\tilde{x},\tilde{y}}\}|$. Finally, by virtue of Lemma A.11 we know that $\mathcal{M}(c, \tilde{x}, \tilde{y}) = \{M_p | p \text{ is a path through } \mathcal{G}_{c,\tilde{x},\tilde{y}}\}$, which concludes the proof.

□

Now that we have shown the correctness of Algorithm A.1, we can go on to show the correctness of Algorithm A.2.

Lemma A.14. *Let $\tilde{x}$ and $\tilde{y}$ be trees over some alphabet $\mathcal{A}$, let c be a cost function over $\mathcal{A}$ which conforms to the triangular inequality, let $\mathcal{G}_{c,\tilde{x},\tilde{y}} = (V, E)$ be the cooptimal edit graph with respect to $\tilde{x}$, $\tilde{y}$, and c, and let C be the second output of Algorithm A.1 for $\tilde{x}$, $\tilde{y}$, and c. Then, after the execution of Algorithm A.2, it holds for all $(1, i, 1, j)$ which are reachable from $(1, 1, 1, 1)$:*

$$\boldsymbol{B}_{i,j} = |\{M_p | p \text{ is a path from } (1, i, 1, j) \text{ to } (1, |\tilde{x}| + 1, 1, |\tilde{y}| + 1) \text{ in } \mathcal{G}_{c,\tilde{x},\tilde{y}}\}|.$$

Proof. First, note that, due to the previous lemma, C contains all (i, j) such that $(1, i, 1, j)$ is reachable from $(1, 1, 1, 1)$. Therefore, Algorithm A.2 visits all these notes exactly once in descending lexicographic order, starting with $(1, |\tilde{x}| + 1, 1, |\tilde{y}| + 1)$.

The outline of this proof is the same as in the previous lemma: We first consider an alternative version of Algorithm A.2 which accumulates all cooptimal tree mappings. In particular:

- We replace line 2 with "Initialize $\tilde{\boldsymbol{B}}$ as a $(|\tilde{x}| + 1) \times (|\tilde{y}| + 1)$ matrix of empty sets".
- We replace line 3 with $\tilde{\boldsymbol{B}}_{|\tilde{x}|+1,|\tilde{y}|+1} \leftarrow \{\emptyset\}$.
- We replace line 8 with $\tilde{\boldsymbol{B}}_{i,j} \leftarrow \tilde{\boldsymbol{B}}_{i,j} \cup \tilde{\boldsymbol{B}}_{i+1,j}$.
- We replace line 11 with $\tilde{\boldsymbol{B}}_{i,j} \leftarrow \tilde{\boldsymbol{B}}_{i,j} \cup \tilde{\boldsymbol{B}}_{i,j+1} \setminus \tilde{\boldsymbol{B}}_{i,j} \cup \big\{\{(i, j)\} \cup M \big| M \in \tilde{\boldsymbol{B}}_{i,j+1} \cap \tilde{\boldsymbol{B}}_{i,j}\big\}$.
- We replace line 18 with $\tilde{\boldsymbol{B}}_{i,j} \leftarrow \tilde{\boldsymbol{B}}_{i,j} \cup \big\{\{(i, j)\} \cup M \big| M \in \tilde{\boldsymbol{B}}_{i+1,j+1}\big\}$.
- We replace line 25 with $\tilde{\boldsymbol{B}}_{i,j} \leftarrow \tilde{\boldsymbol{B}}_{i,j} \cup \big\{M \cup \{(i' - 1 + i, j' - 1 + j) | (i', j') \in M'\} \big| M \in \tilde{\boldsymbol{B}}_{rl_{\tilde{x}}(i)+1, rl_{\tilde{y}}(j)+1}, M' \in \tilde{\boldsymbol{A}}'_{|\tilde{x}^i|+1, |\tilde{y}^j|+1}\big\}$.

Note that we also assume that the call to Algorithm A.1 refers to the changed version as presented in the proof of the previous lemma.

As in the previous lemma, we will first show that

$$\tilde{\boldsymbol{B}}_{i,j} = \{M_p | p \text{ is a path from } (1,i,1,j) \text{ to } (1,|\tilde{x}|+1,1,|\tilde{y}|+1) \text{ in } \mathcal{G}_{c,\tilde{x},\tilde{y}}\} \quad \text{(A.69)}$$

and then go on to show that $\boldsymbol{B}_{i,j} = |\tilde{\boldsymbol{B}}_{i,j}|$, which will conclude our proof.

We prove Equation A.69 via induction over all entries $(i,j) \in C$ in descending lexicographic order. The lexicographic maximum is $(|\tilde{x}|+1,|\tilde{y}|+1)$. In this case it holds $\boldsymbol{B}_{|\tilde{x}|+1,|\tilde{y}|+1} = \{\emptyset\}$. Indeed, the trivial path $p = (1,|\tilde{x}|+1,1,|\tilde{y}|+1)$ is the only path from $(1,|\tilde{x}|+1,1,|\tilde{y}|+1)$ to $(1,|\tilde{x}|+1,1,|\tilde{y}|+1)$ and the corresponding tree mapping M_p is $\emptyset$.

Now, consider some entry $(i,j) < (|\tilde{x}|+1,|\tilde{y}|+1)$ and assume that claim holds for all entries $(i',j') > (i,j)$.

First, we show that for any $M \in \tilde{\boldsymbol{B}}_{i,j}$ there exists a path p from $(1,i,1,j)$ to $(1,|\tilde{x}|+1,1,|\tilde{y}|+1)$, such that $M_p = M$. We distinguish the following cases.

If M has been added to $\tilde{\boldsymbol{B}}_{i,j}$ via line 8, then $M \in \tilde{\boldsymbol{B}}_{i+1,j}$ and $\boldsymbol{D}_{i,j} = c(x_i,-) + \boldsymbol{D}_{i+1,j}$. Further, per induction, there exists a path p' from $(1,i+1,1,j)$ to $(1,|\tilde{x}|+1,1,|\tilde{y}|+1)$ such that $M_{p'} = M$. Finally, due to $\boldsymbol{D}_{i,j} = c(x_i,-) + \boldsymbol{D}_{i+1,j}$ we kow that $\big((1,i,1,j),(1,i+1,1,j)\big) \in E$, such that $p := (1,i,1,j), p'$ is a path from $(1,i,1,j)$ to $(1,|\tilde{x}|+1,1,|\tilde{y}|+1)$ in $\mathcal{G}_{c,\tilde{x},\tilde{y}}$ and $M_p = M$.

If M has been added to $\tilde{\boldsymbol{B}}_{i,j}$ via line 11, then it must hold $\boldsymbol{D}_{i,j} = c(-,y_j) + \boldsymbol{D}_{i,j+1}$. Now, we distinguish two cases.

First, consider the case that $(i,j) \notin M$, that is, M is *not* a duplicate with a tree mapping that has been added to $\tilde{\boldsymbol{B}}_{i,j}$ before. Then, per induction, there exists a path p' from $(1,i,1,j+1)$ to $(1,|\tilde{x}|+1,1,|\tilde{y}|+1)$, such that $M_{p'} = M$. Further, due to $\boldsymbol{D}_{i,j} = c(-,y_j) + \boldsymbol{D}_{i,j+1}$ we kow that $\big((1,i,1,j),(1,i,1,j+1)\big) \in E$, such that $p := (1,i,1,j), p'$ is a path from $(1,i,1,j)$ to $(1,|\tilde{x}|+1,1,|\tilde{y}|+1)$ in $\mathcal{G}_{c,\tilde{x},\tilde{y}}$ and $M_p = M$.

Next, consider the case that $(i,j) \in M$, that is, $M' := M \setminus \{(i,j)\}$ *is* a duplicate with a tree mapping that has been added to $\tilde{\boldsymbol{B}}_{i,j}$ before. The only line in which that could have happened is line 8 such that it must hold $\boldsymbol{D}_{i,j} = c(x_i,-) + \boldsymbol{D}_{i+1,j}$. Due to our induction hypothesis, there must exist two paths, p' from $(1,i,1,j+1)$ to $(1,|\tilde{x}|+1,1,|\tilde{y}|+1)$ and $p'' = v_0,\ldots,v_T$ from $(1,i+1,1,j)$ to $(1,|\tilde{x}|+1,1,|\tilde{y}|+1)$ such that $M_{p''} = M_{p'} = M'$. Because of this latter constraint, $i \in I(M',\tilde{x},\tilde{y})$ and $j \in J(M',\tilde{x},\tilde{y})$, which means that there exists some $t > 0$ such that $v_t = (1,k,1,j+1)$ for some $k \geq i+1$ and $v_{t-1} = (1,k,1,j)$, $v_{t-2} = (1,k-1,1,j)$, and so forth, until $v_0 = (1,i+1,1,j)$. Due to this form of the path we can further conlude that $\boldsymbol{D}_{i,j} = c(x_i,-) + \boldsymbol{D}_{i+1,j} = \ldots = c(x_i,-) + \ldots + c(x_{k-1},-) + \boldsymbol{D}_{k,j} = c(x_i,-) + \ldots + c(x_{k-1},-) + c(-,y_j) + \boldsymbol{D}_{k,j+1}$. Due to $\boldsymbol{D}_{i,j} = c(-,y_j) + \boldsymbol{D}_{i,j+1}$ we can re-write this expression as $\boldsymbol{D}_{i,j+1} = c(x_i,-) + \ldots + c(x_{k-1},-) + \boldsymbol{D}_{k,j+1}$. This, in turn, implies that $\boldsymbol{D}_{i,j+1} = c(x_i,-) + \boldsymbol{D}_{i+1,j+1}$. Otherwise, we would obtain $\boldsymbol{D}_{i,j+1} < c(x_i,-) + \boldsymbol{D}_{i+1,j+1} \leq c(x_i,-) + \ldots + c(x_{k-1},-) + \boldsymbol{D}_{k,j+1} = \boldsymbol{D}_{i,j+1}$, which is a contradiction.

Furthermore, it holds: $c(x_i,-) + c(-,y_j) + \boldsymbol{D}_{i+1,j+1} = c(-,y_j) + \boldsymbol{D}_{i,j+1} = \boldsymbol{D}_{i,j} \leq c(x_i,y_j) + \boldsymbol{D}_{i+1,j+1}$, which in turn implies $c(x_i,-) + c(-,y_j) \leq c(x_i,y_j)$. In conjunction with the triangular inequality on c this gives us $c(x_i,-) + c(-,y_j) = c(x_i,y_j)$, which in turn implies that $\big((1,i,1,j),(1,i+1,1,j+1)\big)$ is an edge in the graph. Finally, we can construct the path $p := (1,i,1,j),(1,i+1,1,j+1),\ldots,(1,k-1,1,j+1),v_t,\ldots,v_T$. For this path it holds per construction that $M_p = M$.

If M has been added via line 18, then $M' := M \setminus \{(i,j)\} \in \tilde{\boldsymbol{B}}_{i+1,j+1}$, $\boldsymbol{D}_{i,j} = c(x_i,y_j) + \boldsymbol{D}_{i+1,j+1}$, $rl_{\tilde{x}}(i) = rl_{\tilde{x}}(1)$, and $rl_{\tilde{y}}(j) = rl_{\tilde{y}}(1)$. Further, per induction, there exists a path p' from $(1,i+1,1,j+1)$ to $(1,|\tilde{x}|+1,1,|\tilde{y}|+1)$ such that $M_{p'} = M$. Finally, due to $\boldsymbol{D}_{i,j} = c(x_i,y_j) + \boldsymbol{D}_{i+1,j+1}$ we know that $\big((1,i,1,j),(1,i+1,1,j+1)\big) \in E$, such that $p := (1,i,1,j),p'$ is a path from $(1,i,1,j)$ to $(1,|\tilde{x}|+1,1,|\tilde{y}|+1)$ in $\mathcal{G}_{c,\tilde{x},\tilde{y}}$ and $M_p = M$.

Finally, if M has been added via line 25, then M can be re-written as the disjoint union of two sets, $\tilde{M}$ and $\tilde{M}'$, where $\tilde{M} \in \tilde{\boldsymbol{B}}_{rl_{\tilde{x}}(i)+1,rl_{\tilde{y}}(j)+1}$ and $M' := \{(i'-i+1,j'-j+1)|(i',j') \in \tilde{M}'\} \in \tilde{\boldsymbol{A}}'_{|\tilde{x}^i|,|\tilde{y}^j|}$. By virtue of our induction hypothesis, there exist two paths, $\tilde{p}$ from $(1,rl_{\tilde{x}}(i)+1,1,rl_{\tilde{y}}(j)+1)$ to $(1,|\tilde{x}|+1,1,|\tilde{y}|+1)$ in $\mathcal{G}_{c,\tilde{x},\tilde{y}}$, and $p' = v_0,\ldots,v_T$ through $\mathcal{G}_{c,\tilde{x}^i,\tilde{y}^j}$ such that $M_{\tilde{p}} = \tilde{M}$ and $M_{p'} = M'$. Note that v_1 is necessarily $(1,2,1,2)$ because any other path has been blocked via line 24.

Further, because $\boldsymbol{D}_{i,j} = \boldsymbol{d}_{i,j} + \boldsymbol{D}_{rl_{\tilde{x}}(i)+1,rl_{\tilde{y}}(j)+1}$ we know that $\big((1,i,1,j),(k,i+1,l,j+1)\big) \in E$ with $k = \mathrm{k}_{\tilde{x}}(i)$ and $l = \mathrm{k}_{\tilde{y}}(j)$. Because $rl_{\tilde{x}}(i) < |\tilde{x}|$ or $rl_{\tilde{y}}(j) < |\tilde{y}|$ we know that $\big((k,rl_{\tilde{x}}(i)+1,l,rl_{\tilde{y}}(j)+1),(1,rl_{\tilde{x}}(i)+1,1,rl_{\tilde{y}}(j)+1)\big) \in E$.

Now, let $\tilde{p}' := \tilde{v}_1,\ldots,\tilde{v}_T$ with $\tilde{v}_t = (\tilde{k},i+i'-1,\tilde{l},j+j'-1)$ if and only if $v_t = (k',i',l',j')$ where $\tilde{k} := k$ if $k' = 1$ and $\tilde{k} := k'+i-1$ otherwise, and where $\tilde{l} := l$ if $l' = 1$ and $\tilde{l} := l'+j-1$ otherwise. This is a path in $\mathcal{G}_{c,\tilde{x},\tilde{y}}$ and, per construction, it holds that $M_{\tilde{p}'} = \tilde{M}' \setminus \{(i,j)\}$. Accordingly, the concatenated path $p := (1,i,1,j),\tilde{p}',\tilde{p}$ is a path from $(1,i,1,j)$ to $(1,|\tilde{x}|+1,1,|\tilde{y}|+1)$ in $\mathcal{G}_{c,\tilde{x},\tilde{y}}$ and $M_p = M$.

Now, it remains to show that for any path p from $(1,i,1,j)$ to $(1,|\tilde{x}|+1,1,|\tilde{y}|+1)$ in $\mathcal{G}_{c,\tilde{x},\tilde{y}}$, M_p is in $\tilde{\boldsymbol{B}}_{i,j}$. Let $p = v_0,\ldots,v_T$ and $p' = v_1,\ldots,v_T$. Then, consider the following cases for v_1.

$v_1 = (1,i+1,1,j)$**:** Then, $\boldsymbol{D}_{i,j} = c(x_i,-) + \boldsymbol{D}_{i+1,j}$ and line 8 is executed during the visit of v_0. Further, due to the induction hypothesis, we know that $M_{p'} \in \tilde{\boldsymbol{B}}_{i+1,j}$, and thus $M_{p'}$ is added to $\tilde{\boldsymbol{B}}_{i,j}$ during the execution of line 8. Finally, because $M_p = M_{p'}$ it follows that $M_p \in \tilde{\boldsymbol{B}}_{i,j}$.

$v_1 = (1,i,1,j+1)$**:** Then, $\boldsymbol{D}_{i,j} = c(-,y_j) + \boldsymbol{D}_{i,j+1}$ and line 11 is executed during the visit of v_0. Further, due to the induction hypothesis, we know that $M_{p'} \in \tilde{\boldsymbol{B}}_{i,j+1}$, and thus $M_{p'}$ is added to $\tilde{\boldsymbol{B}}_{i,j}$ during the execution of line 11 (or is already element of this set). Finally, because $M_p = M_{p'}$ it follows that $M_p \in \tilde{\boldsymbol{B}}_{i,j}$.

$v_1 = (1,i+1,1,j+1)$**:** Then, $\boldsymbol{D}_{i,j} = c(x_i,y_j) + \boldsymbol{D}_{i+1,j+1}$.

Now, consider two different cases. If $c(x_i,y_j) < c(x_i,-) + c(-,y_j)$, then $rl_{\tilde{x}}(i) = rl_{\tilde{x}}(1)$ and $rl_{\tilde{y}}(j) = rl_{\tilde{y}}(1)$, otherwise $\big((1,i,1,j),(1,i+1,1,j+1)\big)$ would not be an edge in $\mathcal{G}_{c,\tilde{x},\tilde{y}}$. Therefore, line 18 is executed during the visit of v_0. Further, due to the induction hypothesis, we know that $M_{p'} \in \tilde{\boldsymbol{B}}_{i,j+1}$, and thus $M_{p'} \cup \{(i,j)\}$ is added to $\tilde{\boldsymbol{B}}_{i,j}$ during the execution of line 18. Finally, because $M_p = M_{p'} \cup \{(i,j)\}$ it follows that $M_p \in \tilde{\boldsymbol{B}}_{i,j}$.

If $c(x_i, y_j) = c(x_i, -) + c(-, y_j)$, we obtain $\boldsymbol{D}_{i,j} \leq c(x_i, -) + \boldsymbol{D}_{i+1,j} \leq c(x_i, -) + c(-, y_j) + \boldsymbol{D}_{i+1,j+1} = c(x_i, y_j) + \boldsymbol{D}_{i+1,j+1} = \boldsymbol{D}_{i,j}$ and $\boldsymbol{D}_{i,j} \leq c(-, y_j) + \boldsymbol{D}_{i,j+1} \leq c(x_i, -) + c(-, y_j) + \boldsymbol{D}_{i+1,j+1} = c(x_i, y_j) + \boldsymbol{D}_{i+1,j+1} = \boldsymbol{D}_{i,j}$, which in turn implies $\boldsymbol{D}_{i,j} = c(x_i, -) + \boldsymbol{D}_{i+1,j}$, $\boldsymbol{D}_{i,j} = c(-, y_j) + \boldsymbol{D}_{i,j+1}$, $\boldsymbol{D}_{i+1,j} = c(-, y_j) + \boldsymbol{D}_{i+1,j+1}$, and $\boldsymbol{D}_{i,j+1} = c(x_i, -) + \boldsymbol{D}_{i+1,j+1}$.

Therefore, $p^l := (1, i+1, 1, j), p'$ is a path in $\mathcal{G}_{c,\tilde{x},\tilde{y}}$ and line 8 is executed during the visit of v_0. Further, due to the induction hypothesis, we know that $M_{p^l} \in \tilde{\boldsymbol{B}}_{i+1,j}$ and thus $M_{p^l} \in \tilde{\boldsymbol{B}}_{i,j}$.

We also know that $p^r := (1, i, 1, j+1), p'$ is a path in $\mathcal{G}_{c,\tilde{x},\tilde{y}}$ and line 11 is executed during the visit of v_0. Further, due to the induction hypothesis, we know that $M_{p^r} \in \tilde{\boldsymbol{B}}_{i,j+1}$. Now, note that $M_{p'} = M_{p^l} = M_{p^r}$. Therefore, when line 11 is executed, M_{p^r} is already element of $\tilde{\boldsymbol{B}}_{i,j}$. Therefore, line 11 adds $M_{p^r} \cup \{(i,j)\}$ to $\tilde{\boldsymbol{B}}_{i,j}$. Because $M_p = M_{p'} \cup \{(i,j)\} = M_{p^r} \cup \{(i,j)\}$, this implies that $M_p \in \tilde{\boldsymbol{B}}_{i,j}$.

$v_1 = (\mathrm{k}_{\tilde{x}}(i), i+1, \mathrm{k}_{\tilde{y}}(j), j+1)$: Then, there must exist some $t > 1$ such that $v_t = (1, rl_{\tilde{x}}(i) + 1, 1, rl_{\tilde{y}}(i) + 1)$ and $v_{t-1} = (\mathrm{k}_{\tilde{x}}(i), rl_{\tilde{x}}(i) + 1, \mathrm{k}_{\tilde{y}}(j), rl_{\tilde{y}}(i) + 1)$, otherwise p would never reach $(1, |\tilde{x}| + 1, 1, |\tilde{y}| + 1)$. Further, $\boldsymbol{D}_{i,j} = \boldsymbol{d}_{i,j} + \boldsymbol{D}_{rl_{\tilde{x}}(i)+1, rl_{\tilde{y}}(i)+1}$, $c(x_i, y_j) < c(x_i, -) + c(-, y_j)$, and $rl_{\tilde{x}}(i) \neq rl_{\tilde{x}}(1)$ or $rl_{\tilde{y}}(j) \neq rl_{\tilde{y}}(1)$, otherwise $(v_0, v_1) \notin E$. Therefore, lines 22-26 are executed.

Now, let $\tilde{p}^* = v_0, v_1, \ldots, v_{t-1}$, and let $p^* = (1,1,1,1), v'_1, \ldots, v'_{t-1}$ with $v'_{t'} = (\tilde{k}, i' - i + 1, \tilde{l}, j' - j + 1)$ if and only if $v_{t'} = (k', i', l', j')$ where $\tilde{k} := 1$ if $k' = \mathrm{k}_{\tilde{x}}(i)$ and $\tilde{k} := k' - i + 1$ otherwise, and where $\tilde{l} := 1$ if $l' = \mathrm{k}_{\tilde{y}}(j)$ and $\tilde{l} := l' - j + 1$ otherwise. Further, let $\tilde{p} = v_t, \ldots, v_T$. Per induction hypothesis, $M_{p^*} \in \tilde{\boldsymbol{A}}'_{|\tilde{x}^i|+1, |\tilde{y}^j|+1}$ and $M_{\tilde{p}} \in \tilde{\boldsymbol{B}}_{rl_{\tilde{x}}(i)+1, rl_{\tilde{y}}(j)+1}$. Further, note that $M_p = M_{\tilde{p}^*} \cup M_{\tilde{p}}$. Therefore, M_p gets added to $\tilde{\boldsymbol{B}}_{i,j}$ during the execution of line 25.

This concludes the proof of Equation A.69. Now, it remains to show that, for all $(1, i, 1, j)$ which are reachable from $(1,1,1,1)$ it holds: $\boldsymbol{B}_{i,j} = |\tilde{\boldsymbol{B}}_{i,j}|$.

First, observe that $\boldsymbol{B}_{|\tilde{x}|+1, |\tilde{y}|+1} = 1 = |\{\emptyset\}| = |\tilde{\boldsymbol{B}}_{|\tilde{x}|+1, |\tilde{y}|+1}|$.

Second, note that, whenever line 8 is executed, $\tilde{\boldsymbol{B}}_{i,j}$ is empty. So it holds $|\tilde{\boldsymbol{B}}_{i,j}| = |\tilde{\boldsymbol{B}}_{i,j}| + |\tilde{\boldsymbol{B}}_{i+1,j}|$, which yields the original line 8 in Algorithm A.2.

Third, note that line 11, per construction, never adds any tree mappings which have already been added in line 8. Therefore, it holds $|\tilde{\boldsymbol{B}}_{i,j}| = |\tilde{\boldsymbol{B}}_{i,j}| + |\tilde{\boldsymbol{B}}_{i,j+1}|$, which yields the original line 11 in Algorithm A.2.

Fourth, note that, whenever line 18 is executed, either line 8 or line 11 can not have been executed. Otherwise, as we have shown above, $c(x_i, -) + c(-, y_j) = c(x_i, y_j)$, which would in turn imply that the **continue** statement in line 14 would have been executed, which prevents the execution of line 18. Because either line 8 or line 11 have thus not been executed, the duplicate case in line 11 can not apply, such that no tree mappings which contain (i,j) have yet been added to $\tilde{\boldsymbol{B}}_{i,j}$. Because line 18 only adds tree mappings which contain (i,j), the set untion in line 18 must be disjoint. Therefore, it holds $|\tilde{\boldsymbol{B}}_{i,j}| = |\tilde{\boldsymbol{B}}_{i,j}| + |\tilde{\boldsymbol{B}}_{i+1,j+1}|$, which yields the original line 18 in Algorithm A.2.

Finally, note that, whenever line 25 is executed, either line 8 or line 11 can not have been executed by the same reasoning as above, and line 18 can not have been executed, otherwise line 25 would not be executed. Therefore, $\tilde{\boldsymbol{B}}_{i,j}$ does not yet contain tree mappings which contain (i,j). However, since $\tilde{\boldsymbol{A}}'_{|\tilde{x}^i|+1,|\tilde{y}^j|+1}$ only contains such tree mappings, the set union in line 25 must be disjoint. Therefore, it holds $|\tilde{\boldsymbol{B}}_{i,j}| = |\tilde{\boldsymbol{B}}_{i,j}| + |\tilde{\boldsymbol{B}}_{rl_{\tilde{x}}(i)+1,rl_{\tilde{y}}(j)+1}| \cdot |\tilde{\boldsymbol{A}}'_{|\tilde{x}^i|+1,|\tilde{y}^j|+1}|$, which yields the original line 25 in Algorithm A.2.

This concludes the proof. □

Now we have established all intermediate results we can prove the overall correctness of Algorithm 4.1. As in the lemmas above, we first consider a variant of Algorithm 4.1 where $\tilde{\boldsymbol{\Gamma}}_{i,j}$ contains all tree mappings which contain (i,j) and then go on to show that $\boldsymbol{\Gamma}_{i,j} = |\tilde{\boldsymbol{\Gamma}}_{i,j}|$. In particular, consider the following variations of Algorithm 4.1.

- We replace line 2 with $(C, \tilde{\boldsymbol{A}}) \leftarrow \text{forward}(\tilde{x}, \tilde{y}, \boldsymbol{d}, \boldsymbol{D}, c)$, where "forward" refers to the variant of the forward algorithm introduced in the proof to Lemma A.13.
- We replace line 3 with $\tilde{\boldsymbol{B}} \leftarrow \text{backward}(\tilde{x}, \tilde{y}, \boldsymbol{d}, \boldsymbol{D}, c, C)$, where "backward" refers to the variant of the backward algorithm introduced in the proof to Lemma A.14.
- We replace line 4 with "Initialize $\tilde{\boldsymbol{\Gamma}}$ as a $|\tilde{x}| \times |\tilde{y}|$ matrix of empty sets".
- We replace line 11 with $\tilde{\boldsymbol{\Gamma}}_{i,j} \leftarrow \tilde{\boldsymbol{\Gamma}}_{i,j} \cup \big\{\tilde{M} \cup \tilde{M}' \cup \{(i,j)\}\big|\tilde{M} \in \tilde{\boldsymbol{A}}_{i,j}, \tilde{M}' \in \tilde{\boldsymbol{B}}_{i+1,j+1}\big\}$.
- We replace line 15 with $\tilde{\gamma} \leftarrow \{\tilde{M} \cup \tilde{M}' | \tilde{M} \in \tilde{\boldsymbol{A}}_{i,j}, \tilde{M}' \in \tilde{\boldsymbol{B}}_{rl_{\tilde{x}}(i)+1,rl_{\tilde{y}}(j)+1}\}$.
- We replace the symbol $\boldsymbol{\Gamma}'$ in line 19 with the symbol $\tilde{\boldsymbol{\Gamma}}'$.
- We replace line 21 with $\tilde{\boldsymbol{\Gamma}}_{i+i'-1,j+j'-1} \leftarrow \tilde{\boldsymbol{\Gamma}}_{i+i'-1,j+j'-1} \cup \big\{\tilde{M} \cup \{(i+i''-1, j+j''-1)|(i'',j'') \in M^*\}\big|\tilde{M} \in \tilde{\gamma}, M^* \in \boldsymbol{\Gamma}'_{i',j'}\big\}$.

We now show that, after Algorithm 4.1 has been executed, $\tilde{\boldsymbol{\Gamma}}_{i,j}$ contains for all (i,j) exactly the cooptimal tree mappings which contain (i,j).

To prove this claim, we perform an induction over $|\tilde{x}| + |\tilde{y}|$.

If both $\tilde{x}$ and $\tilde{y}$ contain only a single node, then, $\boldsymbol{D}_{2,2} = 0$ and

$$\begin{aligned}\boldsymbol{D}_{1,1} &= \min\{c(x_1,y_1) + \boldsymbol{D}_{2,2}, c(x_1,-) + \boldsymbol{D}_{2,1}, c(-,y_1) + \boldsymbol{D}_{1,2}\} \\ &= \min\{c(x_1,y_1) + \boldsymbol{D}_{2,2}, c(x_1,-) + c(-,y_1) + \boldsymbol{D}_{2,2}\}\end{aligned}$$

Due to the triangular inequality, $c(x_1,y_1) \leq c(x_1,-) + c(-,y_1)$. Therefore, $\boldsymbol{D}_{1,1} = c(x_1,y_1) + \boldsymbol{D}_{2,2}$. Further, $rl_{\tilde{x}}(1) = 1 = |\tilde{x}|$ and $rl_{\tilde{y}}(1) = 1 = |\tilde{y}|$. Finally, $(1,1,1,1)$ is always trivially reachable in the cooptimal edit graph $\mathcal{G}_{c,\tilde{x},\tilde{y}}$ such that line 11 is executed for $i = 1$ and $j = 1$.

Due to the previous lemmas, we know that $\tilde{\boldsymbol{A}}_{1,1} = \{\emptyset\}$ and $\tilde{\boldsymbol{B}}_{2,2} = \tilde{\boldsymbol{B}}_{|\tilde{x}|+1,|\tilde{y}|+1} = \{\emptyset\}$. Therefore, $\boldsymbol{\Gamma}_{1,1} = \{\emptyset \cup \emptyset \cup \{(1,1\}\} = \{\{(1,1)\}\}$. Indeed $\{(1,1)\}$ is the only possible cooptimal tree mapping containing $(1,1)$. Therefore, the claim holds.

Now, consider two trees $\tilde{x}$ and $\tilde{y}$ such that $|\tilde{x}| + |\tilde{y}| > 2$, let $(V,E) := \mathcal{G}_{c,\tilde{x},\tilde{y}}$ be the cooptimal edit graph for $\tilde{x}$ and $\tilde{y}$ with respect to c, and assume that the claim holds for all combinations of trees with added size smaller than $|\tilde{x}| + |\tilde{y}|$.

We now show that all tree mappings in $\boldsymbol{\Gamma}_{i,j}$ are cooptimal tree mappings between $\tilde{x}$ and $\tilde{y}$ which contain (i,j). In particular, let $M \in \boldsymbol{\Gamma}_{i,j}$ and consider the following cases.

If M has been added via line 11, then it must hold that both $rl_{\tilde{x}}(i) = |\tilde{x}| = rl_{\tilde{x}}(1)$ and $rl_{\tilde{y}}(j) = |\tilde{y}| = rl_{\tilde{y}}(1)$ or $c(x_i, y_j) = c(x_i, -) + c(-, y_j)$, and it must hold that $\boldsymbol{D}_{i,j} = c(x_i, y_j) + \boldsymbol{D}_{i+1,j+1}$. Otherwise, line 11 would not have been executed for i and j. Further, it must be possible to re-write M as the disjoint union of three sets $\{(i,j)\}$, $\tilde{M} \in \tilde{\boldsymbol{A}}_{i,j}$, and $\tilde{M}' \in \tilde{\boldsymbol{B}}_{i+1,j+1}$. Otherwise, M could not have been added via line 11.

Since $\tilde{M} \in \tilde{\boldsymbol{A}}_{i,j}$, there must exist a path $\tilde{p}$ from $(1,1,1,1)$ to $(1,i,1,j)$ with $\tilde{M} = M_{\tilde{p}}$. Further, because $\tilde{M}' \in \tilde{\boldsymbol{B}}_{i+1,j+1}$, there must exist a path $\tilde{p}'$ from $(1, i+1, 1, j+1)$ to $(1, |\tilde{x}|+1, 1, |\tilde{y}|+1)$ with $\tilde{M}' = M_{\tilde{p}'}$.

Due to the conditions necessary to execute line 11, we also know that $\big((1,i,1,j),(1,i+1,1,j+1)\big)$ is an edge in $\mathcal{G}_{c,\tilde{x},\tilde{y}}$. Therefore, $p := \tilde{p}, \tilde{p}'$ is a path through $\mathcal{G}_{c,\tilde{x},\tilde{y}}$, and therefore $M_p = M_{\tilde{p}} \cup \{(i,j)\} \cup M_{\tilde{p}'} = \tilde{M} \cup \{(i,j)\} \cup \tilde{M}' = M$ is a cooptimal mapping which contains (i,j).

If M has been added via line 21, then it must hold that there exist two indices, k and l such that $rl_{\tilde{x}}(k) \neq rl_{\tilde{x}}(1)$ or $rl_{\tilde{y}}(l) \neq rl_{\tilde{y}}(1)$, $\boldsymbol{D}_{k,l} = \boldsymbol{d}_{k,l} + \boldsymbol{D}_{rl_{\tilde{x}}(k)+1, rl_{\tilde{y}}(l)+1}$, and such that M can be re-written as the disjoint union of three sets, $\tilde{M} \in \tilde{\boldsymbol{A}}_{k,l}$, $\tilde{M}' \in \tilde{\boldsymbol{B}}_{rl_{\tilde{x}}(k)+1, rl_{\tilde{y}}(l)+1}$, and $\tilde{M}^*$ such that $M^* = \{(i'-k+1, j'-l+1) | (i',j') \in \tilde{M}^*\} \in \boldsymbol{\Gamma}'_{i-k+1, j-l+1}$. Otherwise, M would not have been added via line 21.

Since $\tilde{M} \in \tilde{\boldsymbol{A}}_{k,l}$, there exists a path $\tilde{p}$ from $(1,1,1,1)$ to $(1,k,1,l)$ in $\mathcal{G}_{c,\tilde{x},\tilde{y}}$ such that $\tilde{M}_{\tilde{p}} = \tilde{M}$. Further, since $\tilde{M}' \in \tilde{\boldsymbol{B}}_{rl_{\tilde{x}}(k)+1, rl_{\tilde{y}}(l)+1}$, there exists a path $\tilde{p}'$ from $(1, rl_{\tilde{x}}(k)+1, 1 rl_{\tilde{y}}(l)+1)$ to $(1, |\tilde{x}|+1, 1, |\tilde{y}|+1)$ such that $\tilde{M}' = M_{\tilde{p}'}$. Finally, since $M^* \in \boldsymbol{\Gamma}'_{i-k+1, j-l+1}$, the induction hypothesis implies $(i-k+1, j-l+1) \in M^*$. Also due to the induction hypothesis, M^* is a cooptimal tree mapping for $\tilde{x}^k$ and $\tilde{y}^l$. Therefore, there exists a path $p^* = v_0, \ldots, v_T$ through $\mathcal{G}_{c,\tilde{x}^k,\tilde{y}^l}$, such that $M^* = M_{p^*}$. Accordingly, we can construct the path $\tilde{p}^* = \tilde{v}_1, \ldots, \tilde{v}_T$ with $\tilde{v}_t = (\tilde{k}, k+i'-1, \tilde{l}, l+j'-1)$ for $v_t = (k', i', l', j')$ where $\tilde{k} = \mathrm{k}_{\tilde{x}}(k)$ if $k' = 1$ and $i + k' - 1$ if $k' > 1$, as well as $\tilde{l} = \mathrm{k}_{\tilde{y}}(l)$ if $l' = 1$ and $j + l' - 1$ if $l' > 1$ such that $\tilde{M}^* = M_{\tilde{p}^*} \cup \{(k,l)\}$. Also note that, per construction, $(i,j) \in \tilde{M}^*$.

Because $rl_{\tilde{x}}(k) \neq rl_{\tilde{x}}(1)$ or $rl_{\tilde{y}}(l) \neq rl_{\tilde{y}}(1)$ and $\boldsymbol{D}_{k,l} = \boldsymbol{d}_{k,l} + \boldsymbol{D}_{rl_{\tilde{x}}(k)+1, rl_{\tilde{y}}(l)+1}$ we know that $\big((1,k,1,l),(\mathrm{k}_{\tilde{x}}(k), k+1, \mathrm{k}_{\tilde{y}}(l), l+1)\big) \in E$. Further, we know that $\big((\mathrm{k}_{\tilde{x}}(k), rl_{\tilde{x}}(k)+1, \mathrm{k}_{\tilde{y}}(l), rl_{\tilde{y}}(l)+1), (1, rl_{\tilde{x}}(k)+1, 1, rl_{\tilde{y}}(l)+1)\big) \in E$. Therefore, $p := \tilde{p}, \tilde{p}^*, \tilde{p}'$ is a path through $\mathcal{G}_{c,\tilde{x},\tilde{y}}$, and therefore $M_p = M_{\tilde{p}} \cup \{(k,l)\} \cup M_{\tilde{p}^*} \cup M_{\tilde{p}'} = \tilde{M} \cup \tilde{M}^* \cup \tilde{M}' = M$ is a cooptimal mapping which contains (i,j).

Next, we show that all cooptimal tree mappings which contain (i,j) are contained in $\tilde{\boldsymbol{\Gamma}}_{i,j}$. In particular, let $M \in \mathcal{M}(c, \tilde{x}, \tilde{y})$ such that $(i,j) \in M$. Then, a path $p = v_0, \ldots, v_T$ through $\mathcal{G}_{c,\tilde{x},\tilde{y}}$ exists such that $M = M_p$. Further, because $(i,j) \in M$, there must exist an index $t \in \{1, \ldots, T\}$, such that $v_{t-1} = (k, i, l, j)$ and $v_t = (k', i+1, l', j+1)$ for some k, k', l, l'. Now, consider the following cases.

$k = k' = l = l' = 1$: In that case, it must hold $\boldsymbol{D}_{i,j} = c(x_i, y_j) + \boldsymbol{D}_{i+1,j+1}$, and it must hold $rl_{\tilde{x}}(i) = |\tilde{x}|$ and $rl_{\tilde{y}}(j) = |\tilde{y}|$ or $c(x_i, y_j) = c(x_i, -) + c(-, y_j)$. Otherwise, $(v_{t-1}, v_t) \notin E$. Further, $v_0, \ldots, v_{t-1}$ is a path from $(1,1,1,1)$ to $(1,i,1,j)$ in $\mathcal{G}_{c,\tilde{x},\tilde{y}}$, such that $(1,i,1,j)$ is reachable from $(1,1,1,1)$ and thus $(i,j) \in C$. Therefore, line 11 is executed for (i,j).

Now, let $\tilde{p} := v_0, \ldots, v_{t-1}$ and let $\tilde{p}' := v_t, \ldots, v_T$. Then, $M_{\tilde{p}} \in \tilde{\boldsymbol{A}}_{i,j}$ and $M_{\tilde{p}'} \in \tilde{\boldsymbol{B}}_{i+1,j+1}$. Accordingly, $M_{\tilde{p}} \cup \{(i,j)\} \cup M_{\tilde{p}'} = M_p = M \in \tilde{\boldsymbol{\Gamma}}_{i,j}$.

$k' > 1 \vee l' > 1$**:** In that case, there must exist two indices $\tau, \rho \in \{0, \ldots, T\}$ with $\tau < t$ and $\rho > t$ such that $v_\tau = (1, i_\tau, 1, j_\tau)$ and $v_{\tau+1} = (\mathrm{k}_{\tilde{x}}(i_\tau), i_\tau + 1, \mathrm{k}_{\tilde{y}}(j_\tau), j_\tau + 1)$, as well as $v_{\rho-1} = (\mathrm{k}_{\tilde{x}}(i_\tau), rl_{\tilde{x}}(i_\tau) + 1, \mathrm{k}_{\tilde{y}}(j_\tau), rl_{\tilde{y}}(j_\tau) + 1)$ and $v_\rho = (1, rl_{\tilde{x}}(i_\tau) + 1, 1, rl_{\tilde{y}}(j_\tau) + 1)$ for some i_τ, j_τ such that $i_\tau \leq i$ and $j_\tau \leq j$. Otherwise, the path would be impossible.

Now, let $\tilde{p} := v_0, \ldots, v_\tau$, let $\tilde{p}^* := v_{\tau+1}, \ldots, v_{\rho-1}$, and let $\tilde{p}' := v_\rho, \ldots, v_T$. Then, it holds: $M_{\tilde{p}} \in \tilde{\boldsymbol{A}}_{i_\tau, j_\tau}$ and $M_{\tilde{p}'} \in \tilde{\boldsymbol{B}}_{rl_{\tilde{x}}(i_\tau)+1, rl_{\tilde{y}}(j_\tau)+1}$.

Further, consider the path $p^* := (1,1,1,1), \tilde{v}_{\tau+1}, \ldots, \tilde{v}_{\rho-1}$ with $\tilde{v}_{t'} = (\tilde{k}, i'' - i_\tau + 1, \tilde{l}, j'' - j_\tau + 1)$ if and only if $v_{t'} = (k'', i'', l'', j'')$, where $\tilde{k} = 1$ if $k'' = \mathrm{k}_{\tilde{x}}(i_\tau)$ and $\tilde{k} = k'' - i_\tau + 1$ otherwise, as well as $\tilde{l} = 1$ if $l'' = \mathrm{k}_{\tilde{y}}(j_\tau)$ and $\tilde{l} = l'' - j_\tau + 1$ otherwise. This path is, per construction, a path through $\mathcal{G}_{c, \tilde{x}^{i_\tau}, \tilde{y}^{j_\tau}}$. Therefore, $M_{p^*} \in \mathcal{M}(c, \tilde{x}^{i_\tau}, \tilde{y}^{j_\tau})$. Further, per construction, $(i - i_\tau + 1, j - j_\tau + 1) \in M_{p^*}$. Therefore, per induction, $M^*_p \in \tilde{\boldsymbol{\Gamma}}'_{i-i_\tau+1, j-j_\tau+1}$. It also holds per construction $M_{\tilde{p}^*} = \{(i' + i_\tau - 1, j' + j_\tau - 1) | (i', j') \in M_{p^*}\} \setminus \{(i_\tau, j_\tau)\}$.

Finally, because $(v_\tau, v_{\tau+1}) \in E$, it must hold that $rl_{\tilde{x}}(i_\tau) \neq rl_{\tilde{x}}(1)$ or $rl_{\tilde{y}}(j_\tau) \neq rl_{\tilde{y}}(1)$, $\boldsymbol{D}_{i_\tau, j_\tau} = \boldsymbol{d}_{i_\tau, j_\tau} + \boldsymbol{D}_{rl_{\tilde{x}}(i_\tau)+1, rl_{\tilde{y}}(j_\tau)+1}$, and $c(x_i, y_j) < c(x_i, -) + c(-, y_j)$. Otherwise $(v_\tau, v_{\tau+1})$ would not be an edge in $\mathcal{G}_{c, \tilde{x}, \tilde{y}}$. Therefore, lines 15-25 are executed for i_τ and j_τ and $M_p = M_{\tilde{p}} \cup \{(i_\tau, j_\tau)\} \cup M_{\tilde{p}^*} \cup M_{\tilde{p}'}$ is added to $\tilde{\boldsymbol{\Gamma}}_{i,j}$.

This concludes the proof that $\tilde{\boldsymbol{\Gamma}}_{i,j}$ contains precisely the cooptimal tree mappings which contain (i,j). It remains to show that $\boldsymbol{\Gamma}_{i,j} = |\tilde{\boldsymbol{\Gamma}}_{i,j}|$.

To that end, note that any tree mapping M which is added in line 11 has a corresponding path p which traverses the edge $\big((1,i,1,j), (1,i+1,1,j+1)\big)$, as shown above. Because of this edge, the path can not traverse an edge $\big((1,k,1,l), (\mathrm{k}_{\tilde{x}}(k), k+1, \mathrm{k}_{\tilde{y}}(l), l+1)\big)$ such that $k < i \leq rl_{\tilde{x}}(k)$ or $l < j \leq rl_{\tilde{y}}(l)$. Otherwise, it would not also reach $(1,i,1,j)$. This, in turn, implies that $(k,l) \notin M$, because no edge $\big((1,k,1,l), (1,k+1,1,l+1)\big)$ can exist if an edge $\big((1,k,1,l), (\mathrm{k}_{\tilde{x}}(k), k+1, \mathrm{k}_{\tilde{y}}(l), l+1)\big)$ exists.

By contrast, note that any tree mapping M which is added in line 21 has a corresponding path which *does* traverse an edge $\big((1,k,1,l), (\mathrm{k}_{\tilde{x}}(k), k+1, \mathrm{k}_{\tilde{y}}(l), l+1)\big)$ before it traverses some edge $\big((k', i, l', j), (k'', i+1, l'', j+1)\big)$. Therefore, $(k,l) \in M$. It follows that the tree mappings added via line 11 and via line 21 have no overlap. Therefore, the union in line 11 is disjoint and it holds: $|\tilde{\boldsymbol{\Gamma}}_{i,j}| = |\tilde{\boldsymbol{\Gamma}}_{i,j}| + |\tilde{\boldsymbol{A}}_{i,j}| \cdot |\tilde{\boldsymbol{B}}_{i+1,j+1}|$, which yields the original line 11.

Further, the tree mappings added via line 21 also have no overlap with other tree mappings added via line 21, because our argument above applies recursively to subtrees as well. Therefore, the union in line 21 is disjoint and it holds: $|\boldsymbol{\Gamma}_{i+i'-1, j+j'-1}| \leftarrow |\boldsymbol{\Gamma}_{i+i'-1, j+j'-1}| + |\boldsymbol{\Gamma}'_{i',j'}| \cdot |\gamma|$, which yields the original line 21.

This concludes the correctness proof.

A.14 PROOF OF THEOREM 4.3

Recall the Theorem we intend to prove.

Let $\mathcal{A} = \{x_1, \ldots, x_n\}$ be an alphabet. Then, the following function $\phi : \mathcal{A} \to \mathbb{R}^n$ with

$$\phi(x_i)_j := \begin{cases} 0 & \text{if } j > i \\ \rho_j & \text{if } j < i \\ \rho_i \cdot (i+1) & \text{if } j = i \end{cases} \qquad \text{where} \tag{A.70}$$

$$\rho_i = 1/\sqrt{2 \cdot i \cdot (i+1)} \tag{A.71}$$

is a symbol embedding of $\mathcal{A}$ such that:

$$c_\phi(x, y) = \begin{cases} 0 & \text{if } x = y \\ 1 & \text{otherwise} \end{cases}$$

Proof

We consider an alphabet $\mathcal{A} = \{x_1, \ldots, x_n\}$.

Before we go on to prove the actual result, we first show an auxiliary claim regarding the array ρ. In particular, we show that:

$$\sqrt{1 - \sum_{j=1}^{i-1} \rho_j^2} = \rho_i \cdot (i+1) \tag{A.72}$$

Our proof works via induction. For the base case, we observe that $\sqrt{1 - \sum_{j=1}^{1-1} \rho_j^2} = \sqrt{1-0} = 1 = \frac{1}{2} \cdot 2 = \frac{1}{\sqrt{2 \cdot 1 \cdot (1+1)}} \cdot (1+1) = \rho_1 \cdot (1+1)$.

Now, let's assume that the claim holds for all $i' \leq i$ and consider $i+1$. Then, we obtain

$$\begin{aligned} \sqrt{1 - \sum_{j=1}^{i} \rho_j^2} &= \sqrt{\Big(1 - \sum_{j=1}^{i-1} \rho_j^2\Big) - \rho_i^2} \overset{I.H.}{=} \sqrt{\Big(\rho_i \cdot (i+1)\Big)^2 - \rho_i^2} \\ &= \rho_i \cdot \sqrt{(i+1)^2 - 1} = \sqrt{\frac{(i+1)^2 - 1}{2 \cdot i \cdot (i+1)}} = \sqrt{\frac{i^2 + 2 \cdot i}{2 \cdot i \cdot (i+1)}} \\ &= \sqrt{\frac{i+2}{2 \cdot (i+1)}} \cdot \frac{\sqrt{i+2}}{\sqrt{i+2}} = \frac{i+2}{\sqrt{2 \cdot (i+1) \cdot (i+2)}} = \rho_{i+1} \cdot (i+2) \end{aligned}$$

which completes the induction.

Now, consider the cost $c_\phi(x_i, -)$. We obtain:

$$\begin{aligned} c_\phi(x_i, -) = \|\phi(x_i)\| &= \sqrt{\sum_{j=1}^{n} \phi(x_i)_j^2} = \sqrt{\sum_{j=1}^{i-1} \phi(x_i)_j^2 + \phi(x_i)_i^2} \\ &= \sqrt{\sum_{j=1}^{i-1} \rho_j^2 + (\rho_i \cdot (i+1))^2} \\ &\overset{A.72}{=} \sqrt{\sum_{j=1}^{i-1} \rho_j^2 + 1 - \sum_{j=1}^{i-1} \rho_j^2} = \sqrt{1} = 1 \end{aligned}$$

Due to symmetry reasons, this is the same as $c_\phi(-, x_i)$.

Finally, consider $c_\phi(x_i, x_j)$. If $i = j$, then $c_\phi(x_i, x_i) = 0$ per definition. Now, consider the case $j > i$. In this case, we obtain:

$$\begin{aligned}
c_\phi(x_i, x_j) =& \|\phi(x_i) - \phi(x_j)\|^2 = \sum_{l=1}^{n} (\phi(x_i)_l - \phi(x_j)_l)^2 \\
=& \sum_{l=1}^{i-1} (\phi(x_i)_l - \phi(x_j)_l)^2 + (\phi(x_i)_i - \phi(x_j)_i)^2 \\
& + \sum_{l=i+1}^{j-1} (\phi(x_i)_l - \phi(x_j)_l)^2 + (\phi(x_i)_j - \phi(x_j)_j)^2 \\
=& \sum_{l=1}^{i-1} (\rho_l - \rho_l)^2 + (\rho_i \cdot (i+1) - \rho_i)^2 \\
& + \sum_{l=i+1}^{j-1} (0 - \rho_l)^2 + (0 - \rho_{j,j} \cdot (j+1))^2 \\
\overset{A.72}{=}& \rho_i^2 \cdot (i+1-1)^2 + \sum_{l=i+1}^{j-1} (\rho_l)^2 + 1 - \sum_{l=1}^{j-1} (\rho_l)^2 \\
=& \rho_i^2 \cdot i^2 + 1 - \sum_{l=1}^{i} (\rho_l)^2 \\
\overset{A.72}{=}& \rho_i^2 \cdot i^2 + \rho_{i+1}^2 \cdot (i+2)^2 \\
=& \frac{i^2}{2 \cdot i \cdot (i+1)} + \frac{(i+2)^2}{2 \cdot (i+1) \cdot (i+2)} \\
=& \frac{i}{2 \cdot (i+1)} + \frac{i+2}{2 \cdot (i+1)} = \frac{i+i+2}{2 \cdot (i+1)} = 1
\end{aligned}$$

Due to symmetry reasons, the same holds for $c_\phi(x_j, x_i)$, which concludes our proof.

A.15 PROOF OF THEOREM 5.1

Recall the theorem we intend to prove.

Let $\mathcal{X}$ be some set and let $\{\mathcal{G}_t^j\}_{t=1,\ldots,T_j}^{j=1,\ldots,N} \subset \mathcal{X}$ be a dataset of sequences over that set, let $M = T_1 + \ldots + T_N$, let (j,t) be the ith tuple in $\{(j,t)|j \in \{1,\ldots,N\}, t \in \{1,\ldots,T_j - 1\}\}$ according to lexicographic ordering, let $\bar{x}_i := \mathcal{G}_1^j, \ldots, \mathcal{G}_t^j$, and let $\bar{y}_i := \mathcal{G}_1^j, \ldots, \mathcal{G}_{t+1}^j$.

Further, let d be a pseudo-Euclidean distance over $\mathcal{X}^*$ with positive spatial mapping $\phi^+ : \mathcal{X}^* \to \mathbb{R}^m$ and negative spatial mapping $\phi^- : \mathcal{X}^* \to \mathbb{R}^n$, and let

$$\phi(\bar{x}) := \begin{pmatrix} \phi^+(\bar{x}) \\ \phi^-(\bar{x}) \end{pmatrix} \qquad \text{and} \qquad \boldsymbol{X} := \big(\phi(\bar{x}_1), \ldots, \phi(\bar{x}_M)\big) \in \mathbb{R}^{(m+n)\times M}$$

Finally, let k be a kernel over $\mathbb{R}^{m+n}$, and let $\bar{x} \in \mathcal{X}^*$. Then, it holds:

1. The predictive result of 1-NN according to Equation 2.45 has the form $f(\phi(\bar{x})) = \boldsymbol{X} \cdot \vec{\alpha}$ with $\vec{\alpha}$ having exactly one entry 1 and only zero entries otherwise.

2. For any non-negative similarity s_d, the predictive result of KR according to Equation 2.46 has the form $f(\phi(\bar{x})) = \boldsymbol{X} \cdot \vec{\alpha}$ where all α_i are non-negative and sum up to 1.

3. For the priors $\vec{\theta} = \phi(\bar{x})$ and $\theta_i = \phi(\bar{x}_i)$, the predictive result of GPR according to Equation 2.50 has the form $f(\phi(\bar{x})) = (\boldsymbol{X}, \phi(\bar{x})) \cdot \vec{\alpha}$, where $\alpha_{M+1} = 1$ and all other α_i add up to zero.

4. For the priors $\vec{\theta} = \phi(\bar{x})$ and $\theta_i = \phi(\bar{x}_i)$, the predictive result of rBCM according to Equation 2.53 has the form $f(\phi(\bar{x})) = (\boldsymbol{X}, \phi(\bar{x})) \cdot \vec{\alpha}$, where $\alpha_{M+1} = 1$ and all other α_i add up to zero.

Proof

We prove the single claims in turn.

1. This follows directly from the form of the predictive function in Equation 2.45. In particular, $\alpha_i = 1$ if $i = i^+$ and $\alpha_i = 0$ otherwise.

2. The form of the coefficients follows directly from Equation 2.46. In particular, we obtain
$$\alpha_i = \frac{s_d(\bar{x}, \bar{x}_i)}{\sum_{j=1}^M s_d(\bar{x}, \bar{x}_j)}$$
Due to the normalization, these coefficients necessarily add up to 1, which makes the combination affine. Given that we assumed that s_d is non-negative, the combination is convex.

3. Let $\vec{k} = \big(k(\bar{x}, \bar{x}_1), \ldots, k(\bar{x}, \bar{x}_M)\big)^\top$ and let $\boldsymbol{K}$ be the $M \times M$ matrix with entries $\boldsymbol{K}_{i,j} = k(\bar{x}_i, \bar{x}_j)$. Then, the predictive mean of GPR according to Equation 2.50 is given as $\vec{\mu} = \phi(\bar{x}) + (\boldsymbol{Y} - \boldsymbol{X}) \cdot \vec{\gamma}$ for $\vec{\gamma} = (\boldsymbol{K} + \tilde{\sigma}^2 \cdot \boldsymbol{I}^M)^{-1} \cdot \vec{k}$. Now, let $(\gamma_1^1, \ldots, \gamma_{T_1-1}^1, \ldots, \gamma_{T_N-1}^N)$

$:= \vec{\gamma}^\top$, and let $\bar{x}_t^j = \mathcal{G}_1^j, \ldots, \mathcal{G}_t^j$ for all $j \in \{1, \ldots, N\}$ and all $t \in \{1, \ldots, T_j\}$. Then, we obtain:

$$\begin{aligned}\vec{\mu} &= \phi(\bar{x}) + \sum_{j=1}^{N} \sum_{t=1}^{T_j-1} \gamma_t^j \cdot \left(\phi(\bar{x}_{t+1}^j) - \phi(\bar{x}_t^j)\right) \\ &= 1 \cdot \phi(\bar{x}) + \sum_{j=1}^{N} -\gamma_1^j \cdot \phi(\bar{x}_1^j) + \left(\sum_{t=2}^{T_j-1} (\gamma_{t-1}^j - \gamma_t^j) \cdot \phi(\bar{x}_t^j)\right) + \gamma_{T_j-1}^j \cdot \phi(\bar{x}_{T_j}^j)\end{aligned}$$

Therefore, we can re-write $\vec{\mu}$ as $\vec{\mu} = (\boldsymbol{X}, \phi(\bar{x})) \cdot \vec{\alpha}$ with the coefficients $\vec{\alpha}^\top = (\alpha_1^1, \ldots, \alpha_{T_1}^1, \ldots, \alpha_{T_N}^N, 1)$, where $\alpha_1^j = -\gamma_1^j$, $\alpha_2^j = \gamma_1^j - \gamma_2^j$, …, $\alpha_{T_j-2}^j = \gamma_{T_j-1}^j - \gamma_{T_j-1}^j$, and $\alpha_{T_j}^j = \gamma_{T_j-1}^j$ for all $j \in \{1, \ldots, N\}$. Note that $\alpha_1^j + \ldots + \alpha_{T_j}^j = 0$ for all j. Therefore, the sum over all coefficients in $\vec{\alpha}$ is 1, which makes the combination affine.

4. First, we observe that the previous result implies that the predictive mean for every single c has the shape

$$\vec{\mu}_c = \phi(\bar{x}) + \sum_{i=1}^{M} \alpha_i^c \cdot \phi(\bar{x}_i) \qquad \text{where} \qquad \sum_{i=1}^{M} \alpha_i^c = 0$$

Accordingly, using Equation 2.53, we can re-write the predictive result of rBCM as $\alpha_{M+1} \cdot \phi(\bar{x}) + (\boldsymbol{X}, \phi(\bar{x})) \cdot \vec{\alpha}$ with the following coefficients.

$$\begin{aligned}\alpha_i &= \sigma_{\mathrm{rBCM}}^2 \cdot \sum_{c=1}^{C} \frac{\beta_c}{\sigma_c^2} \cdot \alpha_i^c \qquad \text{for all } i \leq M \\ \alpha_{M+1} &= \sigma_{\mathrm{rBCM}}^2 \cdot \left(\sum_{c=1}^{C} \frac{\beta_c}{\sigma_c^2} \cdot 1 + \left(1 - \sum_{c=1}^{C} \beta_c\right) \cdot \frac{1}{\sigma_{\mathrm{prior}}^2}\right)\end{aligned}$$

Note that the latter coefficient is equal to $\sigma_{\mathrm{rBCM}}^2 \cdot \sigma_{\mathrm{rBCM}}^{-2} = 1$.

Further, for the sum of all other coefficients we obtain:

$$\begin{aligned}\sum_{i=1}^{M} \alpha_i =& \sum_{i=1}^{M} \sigma_{\mathrm{rBCM}}^2 \cdot \sum_{c=1}^{C} \frac{\beta_c}{\sigma_c^2} \cdot \alpha_i^c \\ =& \sigma_{\mathrm{rBCM}}^2 \cdot \sum_{c=1}^{C} \frac{\beta_c}{\sigma_c^2} \cdot \left(\sum_{i=1}^{M} \alpha_i^c\right) = \sigma_{\mathrm{rBCM}}^2 \cdot \sum_{c=1}^{C} \frac{\beta_c}{\sigma_c^2} \cdot 0 = 0\end{aligned}$$

Therefore, we obtain an affine combination as claimed.

A.16 PROOF OF THEOREM 6.2

Recall the theorem we intend to prove.

Let X be a state set, let Δ be a symmetric edit set over X, and let c be a symmetric cost function over Δ.

Then, $d_{\Delta,c}$ is a pseudo-Euclidean distance for some positive spatial map $\phi^+ : X \to \mathbb{R}^m$ and some negative spatial map $\phi^- : X \to \mathbb{R}^n$.

Now, let $\{x_i\}_{i=1,\ldots,M} \subset X$, let $x, y \in X$, let $\boldsymbol{X}^+ := \big(\phi^+(x_1), \ldots, \phi^+(x_M), \phi(x)\big) \in \mathbb{R}^{m \times M+1}$, let $\boldsymbol{X}^- := \big(\phi^-(x_1), \ldots, \phi^-(x_M), \phi(x)\big) \in \mathbb{R}^{n \times M+1}$, and let $\vec{\alpha} \in \mathbb{R}^{M+1}$ such that:

$$\phi^+(y) = \boldsymbol{X}^+ \cdot \vec{\alpha}, \quad \phi^-(y) = \boldsymbol{X}^- \cdot \vec{\alpha}, \quad \text{and} \quad \sum_{i=1}^{M+1} \alpha_i = 1$$

Then, the maximization problem in Equation 6.5 can be re-written as:

$$\min_{\delta \in \Delta} \alpha_{M+1} \cdot d_{\Delta,c}\big(\delta(x), x\big)^2 + \sum_{i=1}^{M} \alpha_i \cdot d_{\Delta,c}\big(\delta(x), x_i\big)^2 \tag{A.73}$$

Proof

The proof has multiple steps. First, we show that $d_{\Delta,c}$ is a non-negative, self-equal and symmetric function.

In particular, let $x, y \in X$. Then, it holds:

The cost of any edit script $\bar{\delta} \in \Delta^*$ with $\bar{\delta}(x) = y$ is a sum over outputs of c. Since c maps to $\mathbb{R}^+$, the cost of any edit script must thus be non-negative. Therefore, $d_{\Delta,c}$ is non-negative as well.

For all x we can use the empty edit script to transform x to x. The cost of the empty edit script is 0, independent of the cost function c. Therefore, $d_{\Delta,c}(x, x) \leq 0$. Because $d_{\Delta,c}(x, x) \geq 0$, we obtain $d_{\Delta,c}(x, x) = 0$

Let $x, y \in X$ such that x and y are connected in the legal move graph. Let $\delta_1, \ldots, \delta_T$ be a cheapest edit script that transforms x to y, let $x_0 = x$ and let $x_t = \delta_t(x_{t-1})$ for all $t \in \{1, \ldots, T\}$.

Because Δ and c are symmetric, we can construct the edit script $\delta_T^{-1}, \ldots, \delta_1^{-1}$ which transforms y to x such that for all $t \in \{1, \ldots, T\}$ we obtain $\delta_t^{-1}(x_t) = x_{t-1}$ and $c(\delta_t^{-1}, x_t) = c(\delta_t, x_{t-1})$, which in turn implies $c(\delta_T^{-1}, \ldots, \delta_1^{-1}, y) = c(\delta_1, \ldots, \delta_T, x)$. Finally, we conclude that $d_{\Delta,c}(y, x) \leq c(\delta_T^{-1}, \ldots, \delta_1^{-1}, y) = c(\delta_1, \ldots, \delta_T, x) = d_c(x, y)$.

We can apply a symmetric argument in the inverse direction yielding $d_{\Delta,c}(x, y) \leq d_{\Delta,c}(y, x)$, which in turn implies $d_{\Delta,c}(x, y) = d_{\Delta,c}(y, x)$. If there is no path from x to y in the legal move graph, then there is also no path from y to x, and it holds $d_{\Delta,c}(x, y) = \infty = d_{\Delta,c}(y, x)$.

Because $d_{\Delta,c}$ is self-equal and symmetric, it is pseudo-Euclidean for some positive spatial map $\phi^+ : X \to \mathbb{R}^m$ and some negative spatial map $\phi^- : X \to \mathbb{R}^n$ according to

Theorem 2.2. Accordingly, we obtain:

$$\begin{aligned} d_{\Delta,c}\big(\delta(x),y\big)^2 =& \|\phi^+(\delta(x)) - \phi^+(y)\|^2 - \|\phi^-(\delta(x)) - \phi^-(y)\|^2 \\ =& \|\phi^+(\delta(x)) - \boldsymbol{X}^+ \cdot \vec{\alpha}\|^2 - \|\phi^-(\delta(x)) - \boldsymbol{X}^- \cdot \vec{\alpha}\|^2 \end{aligned}$$

Following Equation 2.9 from Theorem 2.3, we obtain:

$$\begin{aligned} & \|\phi^+(\delta(x)) - \boldsymbol{X}^+ \cdot \vec{\alpha}\|^2 - \|\phi^-(\delta(x)) - \boldsymbol{X}^- \cdot \vec{\alpha}\|^2 \\ =& \alpha_{M+1} \cdot d(\delta(x),x)^2 + \sum_{i=1}^{M} \alpha_i \cdot d(\delta(x),x_i)^2 - \frac{1}{2}\vec{\alpha}'^\top \cdot \boldsymbol{D}^2 \cdot \vec{\alpha}' \end{aligned} \tag{A.74}$$

where $\boldsymbol{D}^2$ is the $M+1 \times M+1$ matrix with entries $\boldsymbol{D}^2_{i,j} = d(x_i, x_j)^2$ for $i \leq M$ and $j \leq M$, with $\boldsymbol{D}^2_{M+1,i} = \boldsymbol{D}^2_{i,M+1} = d(x, x_i)^2$ for all $i \leq M$, and with $\boldsymbol{D}^2_{M+1,M+1} = 0$.

Because the latter term in Equation A.74 does not depend on δ, we can ignore it for the minimization problem in Equation 6.5. This directly yields Equation 6.6.

A.17 KERNELIZED OTHOGONAL MATCHING PURSUIT

Let $\mathcal{X}$ be some set and let $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ be a kernel on $\mathcal{X}$ with spatial map $\phi : \mathcal{X} \to \mathbb{R}^m$. Further, let $\{x_1, \ldots, x_M\} \subseteq \mathcal{X}$, let $\boldsymbol{X} = (\phi(x_1), \ldots, \phi(x_M))$, and let $\vec{\alpha} \in \mathbb{R}^M$, such that $\sum_{i=1}^M \alpha_i = 1$. Finally, let $J \subseteq \{1, \ldots, M\}$ and let $m \in \mathbb{N}$.

Our aim is to identify a coefficient vector $\tilde{\alpha} \in \mathbb{R}^m$, such that $\sum_{i=1}^M \tilde{\alpha}_i = 1$, such that at most m entries of $\tilde{\alpha}$ are nonzero, and such that $\tilde{\alpha}_i \neq 0$ implies that $i \in J$.

To do so, we adapt kernelized orthogonal matching pursuit (k-OMP) as introduced by D. Hofmann et al. (2014). Recall that OMP has the following basic structure: Initialize I as an empty set. Then, until $|I| = m$, select the element i from $\{1, \ldots, M\} \setminus I$ such that the absolute value of the inner product between $\phi(x_i)$ and the residual $\boldsymbol{X} \cdot \vec{\alpha} - \boldsymbol{X} \cdot \tilde{\alpha}$ is maximized. Add i to I. Then, adapt the coefficients $\tilde{\alpha}_I$ for the selected indices I such that the Euclidean distance between $\boldsymbol{X} \cdot \vec{\alpha}$ and $\boldsymbol{X} \cdot \tilde{\alpha}$ is minimized.

Now, let $\boldsymbol{K} = \boldsymbol{X}^\top \cdot \boldsymbol{X}$, let $\boldsymbol{K}_{I,I}$ be the submatrix of $\boldsymbol{K}$ limited to the indices in I, let $\boldsymbol{K}_{I,:}$ be the submatrix of $\boldsymbol{K}$ containing only the rows I of $\boldsymbol{K}$, and let $\boldsymbol{K}_{:,I}$ bet the submatrix of $\boldsymbol{K}$ containing only the columns I of $\boldsymbol{K}$.

We can re-write the selection process of the next index as follows.

$$\operatorname*{argmax}_{i \in J \setminus I} |\phi(x_i)^\top \cdot (\boldsymbol{X} \cdot \vec{\alpha} - \boldsymbol{X} \cdot \tilde{\alpha})| = \operatorname*{argmax}_{i \in J \setminus I} |\phi(x_i)^\top \cdot \boldsymbol{X} \cdot (\vec{\alpha} - \tilde{\alpha})| = \operatorname*{argmax}_{i \in J \setminus I} |\boldsymbol{K}_{\{i\},:} \cdot (\vec{\alpha} - \tilde{\alpha})|$$

Now, let $\tilde{\alpha}_I$ be the subvector of $\tilde{\alpha}$ limited to the entries I, and let $\vec{1}_{|I|}$ be the $|I|$-dimensional vector of ones. Then, we can re-write the optimization step for the coefficients $\tilde{\alpha}_I$ as follows, using a Lagrangian dual to express the side-constraint $\vec{1}_{|I|}^\top \cdot \tilde{\alpha}_I = 1$.

$$\begin{aligned} & \min_{\tilde{\alpha}_I} \quad \|\boldsymbol{X} \cdot \vec{\alpha} - \boldsymbol{X} \cdot \tilde{\alpha}\|^2 + \nu \cdot (\vec{1}_{|I|}^\top \cdot \tilde{\alpha}_I - 1) \\ =& \min_{\tilde{\alpha}_I} \quad \vec{\alpha}^\top \cdot \boldsymbol{X}^\top \cdot \boldsymbol{X} \cdot \vec{\alpha} - 2 \cdot \vec{\alpha}^\top \cdot \boldsymbol{X}^\top \cdot \boldsymbol{X} \cdot \tilde{\alpha} + \tilde{\alpha}^\top \cdot \boldsymbol{X}^\top \cdot \boldsymbol{X} \cdot \tilde{\alpha} + \nu \cdot (\vec{1}_{|I|}^\top \cdot \tilde{\alpha}_I - 1) \\ =& \min_{\tilde{\alpha}_I} \quad \vec{\alpha}^\top \cdot \boldsymbol{K} \cdot \vec{\alpha} - 2 \cdot \vec{\alpha}^\top \cdot \boldsymbol{K} \cdot \tilde{\alpha} + \tilde{\alpha}^\top \cdot \boldsymbol{K} \cdot \tilde{\alpha} + \nu \cdot (\vec{1}_{|I|}^\top \cdot \tilde{\alpha}_I - 1) \\ =& \min_{\tilde{\alpha}_I} \quad \vec{\alpha}^\top \cdot \boldsymbol{K} \cdot \vec{\alpha} - 2 \cdot \vec{\alpha}^\top \cdot \boldsymbol{K}_{:,I} \cdot \tilde{\alpha}_I + \tilde{\alpha}_I^\top \cdot \boldsymbol{K}_{I,I} \cdot \tilde{\alpha}_I + \nu \cdot (\vec{1}_{|I|}^\top \cdot \tilde{\alpha}_I - 1) \end{aligned}$$

For the gradient and the Hessian we obtain the following expressions according to Petersen and Pedersen (2012).

$$\nabla_{\tilde{\alpha}_I} \|\boldsymbol{X} \cdot \vec{\alpha} - \boldsymbol{X} \cdot \tilde{\alpha}\|^2 + \nu \cdot (\vec{1}_{|I|}^\top \cdot \tilde{\alpha}_I - 1) = -2 \cdot \boldsymbol{K}_{I,:} \cdot \vec{\alpha} + 2 \cdot \boldsymbol{K}_{I,I} \cdot \tilde{\alpha}_I + \nu \cdot \vec{1}_{|I|}$$
$$\nabla^2_{\tilde{\alpha}_I} \|\boldsymbol{X} \cdot \vec{\alpha} - \boldsymbol{X} \cdot \tilde{\alpha}\|^2 + \nu \cdot (\vec{1}_{|I|}^\top \cdot \tilde{\alpha}_I - 1) = 2 \cdot \boldsymbol{K}_{I,I}$$

First, note that the Hessian is positive semi-definite because it is a kernel matrix. Therefore, any coefficient vector with zero gradient is a global optimum. Now, we set the gradient to zero and solve for $\tilde{\alpha}_I$.

$$-2 \cdot \boldsymbol{K}_{I,:} \cdot \vec{\alpha} + 2 \cdot \boldsymbol{K}_{I,I} \cdot \tilde{\alpha}_I + \nu \cdot \vec{1}_{|I|} \overset{!}{=} 0$$
$$\boldsymbol{K}_{I,I}^{-1} \cdot (\boldsymbol{K}_{I,:} \cdot \vec{\alpha} - \frac{1}{2} \cdot \nu \cdot \vec{1}_{|I|}) \overset{!}{=} \tilde{\alpha}_I$$

Using the side-constraint $\vec{1}_{|I|}^\top \cdot \tilde{\alpha}_I = 1$ we can solve for ν.

$$\vec{1}_{|I|}^\top \cdot \boldsymbol{K}_{I,I}^{-1} \cdot (\boldsymbol{K}_{I,:} \cdot \vec{\alpha} - \frac{1}{2} \cdot \nu \cdot \vec{1}_{|I|}) \overset{!}{=} 1$$
$$-2 \cdot \frac{1 - \vec{1}_{|I|}^\top \cdot \boldsymbol{K}_{I,I}^{-1} \cdot \boldsymbol{K}_{I,:} \cdot \vec{\alpha}}{\vec{1}_{|I|}^\top \cdot \boldsymbol{K}_{I,I}^{-1} \cdot \vec{1}_{|I|}} \overset{!}{=} \nu$$

Finally, plugging this result into our solution for $\tilde{\alpha}_I$, we obtain:

$$\tilde{\alpha}_I = \boldsymbol{K}_{I,I}^{-1} \cdot (\boldsymbol{K}_{I,:} \cdot \vec{\alpha} + \frac{1 - \vec{1}_{|I|}^\top \cdot \boldsymbol{K}_{I,I}^{-1} \cdot \boldsymbol{K}_{I,:} \cdot \vec{\alpha}}{\vec{1}_{|I|}^\top \cdot \boldsymbol{K}_{I,I}^{-1} \cdot \vec{1}_{|I|}} \cdot \vec{1}_{|I|}) \tag{A.75}$$

Overall, we obtain Algorithm A.3.

Algorithm A.3 A variant of kernelized Orthogonal Matching Pursuit (k-OMP, D. Hofmann et al. 2014) which ensures affine combinations.

function K-OMP(A $M \times M$-kernel matrix $\boldsymbol{K}$, a coefficient vector $\vec{\alpha} \in \mathbb{R}^M$, a set $J \subseteq \{1, \ldots, M\}$, a number $m \in \mathbb{N}$.)
  Initialize $I \leftarrow \emptyset$.
  Initialize $\tilde{\alpha} \leftarrow \vec{0}_M$.
  while $|I| < m$ **do**
    $i = \text{argmax}_{i \in J} |\boldsymbol{K}_{\{i\},:} \cdot (\vec{\alpha} - \tilde{\alpha})|$.
    $I \leftarrow I \cup \{i\}, \quad J \leftarrow J \setminus \{i\}$.
    Set $\tilde{\alpha}_I$ according to Equation A.75.
  end while
  return $\tilde{\alpha}$.
end function

A.18 PROOF OF THEOREM 7.1

Recall the theorem we intend to prove.

Under the assumption of fixed $\gamma_{k|i}$, $\hat{Q}$ (Equation 7.6) is convex with respect to $\boldsymbol{H}$.

Further, if our source model is a slGMM, $\hat{Q}$ takes a unique optimum at $\boldsymbol{H} = \boldsymbol{W} \cdot \boldsymbol{\Gamma} \cdot \hat{\boldsymbol{X}}^{\dagger}$, where $\hat{\boldsymbol{X}} := (\hat{x}_1, \ldots, \hat{x}_N) \in \mathbb{R}^{n \times N}$, $\boldsymbol{W} := (\vec{\mu}_1, \ldots, \vec{\mu}_K) \in \mathbb{R}^{m \times K}$, $\boldsymbol{\Gamma}$ denotes the $K \times N$-dimensional matrix with the entries $\boldsymbol{\Gamma}_{k,j} = \gamma_{k|j}$, and $\hat{\boldsymbol{X}}^{\dagger}$ denotes the Moore-Penrose-Pseudoinverse of $\hat{\boldsymbol{X}}$.

Proof

Following standard matrix calculus conventions, we define the gradient $\nabla_{\boldsymbol{H}} \hat{Q}$ as the matrix with entries $\left(\nabla_{\boldsymbol{H}} \hat{Q}\right)_{r,s} := \frac{\partial}{\partial \boldsymbol{H}_{r,s}} \hat{Q}$. Further, following the suggestion of Fackler (2005), we define the Hessian $\nabla^2_{\boldsymbol{H}} \hat{Q}$ as a $(n \cdot m) \times (n \cdot m)$ dimensional matrix which contains the second derivatives of $\hat{Q}$ with respect to all pairs of entries in $\boldsymbol{H}$. Equivalently, $\nabla^2_{\boldsymbol{H}} \hat{Q}$ can be seen as the Hessian of $\hat{Q}$ with respect to a vector which contains all entries of $\boldsymbol{H}$ in concatenated form.

We obtain the following results for the gradient and Hessian of $\hat{Q}$ with respect to $\boldsymbol{H}$ (Fackler 2005; Petersen and Pedersen 2012).

$$\nabla_{\boldsymbol{H}} \hat{Q} = 2 \cdot \sum_{k=1}^{K} \boldsymbol{\Lambda}_k \cdot \sum_{j=1}^{N} \gamma_{k|j} \cdot (\boldsymbol{H} \cdot \hat{x}_j - \vec{\mu}_k) \cdot \hat{x}_j^{\top} \tag{A.76}$$

$$\nabla^2_{\boldsymbol{H}} Q(\boldsymbol{H}) = 2 \cdot \sum_{k=1}^{K} \boldsymbol{\Lambda}_k \otimes \left(\sum_{j=1}^{N} \gamma_{k|j} \cdot \hat{x}_j \cdot \hat{x}_j^{\top} \right) \tag{A.77}$$

where $\otimes$ denotes the Kronecker product of two matrices. Recall that $\boldsymbol{\Lambda}_k$ is a positive definite matrix and note that $\sum_{j=1}^{N} \gamma_{k|j} \cdot \hat{x}_j \cdot \hat{x}_j^{\top}$ is positive semi-definite due to its quadratic form. Further, positive semi-definite matrices are closed under the Kronecker product, addition, and the multiplication with positive scalars such that the Hessian is also positive semi-definite, which in turn shows that $\hat{Q}$ is convex with respect to $\boldsymbol{H}$ (Fackler 2005). This concludes the first part of the proof.

For the second part of the proof, we replace all $\boldsymbol{\Lambda}_k$ with $\boldsymbol{\Lambda}$ and set the gradient A.76 to zero.

$$2 \cdot \sum_{k=1}^{K} \boldsymbol{\Lambda} \cdot \sum_{j=1}^{N} \gamma_{k|j} \cdot (\boldsymbol{H} \cdot \hat{x}_j - \vec{\mu}_k) \cdot \hat{x}_j^{\top} = 0$$

$$\iff \quad \boldsymbol{\Lambda} \cdot \boldsymbol{H} \cdot \sum_{j=1}^{N} \hat{x}_j \cdot \hat{x}_j^{\top} = \boldsymbol{\Lambda} \cdot \sum_{k=1}^{K} \sum_{j=1}^{N} \gamma_{k|j} \cdot \vec{\mu}_k \cdot \hat{x}_j^{\top}$$

$$\iff \quad \boldsymbol{\Lambda} \cdot \boldsymbol{H} \cdot \hat{\boldsymbol{X}} \cdot \hat{\boldsymbol{X}}^{\top} = \boldsymbol{\Lambda} \cdot \boldsymbol{W} \cdot \boldsymbol{\Gamma} \cdot \hat{\boldsymbol{X}}^{\top}$$

$$\iff \quad \boldsymbol{H} = \boldsymbol{W} \cdot \boldsymbol{\Gamma} \cdot \hat{\boldsymbol{X}}^{\dagger} \tag{A.78}$$