GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN IN PUBLICA COMMODA
SEIT 1737

Matching Patterns with Variables in Approximate Settings

Dissertation

zur Erlangung des mathematisch-naturwissenschaftlichen Doktorgrades

“Doctor rerum naturalium”

der Georg-August-Universität Göttingen

im Promotionsprogramm

“PhD Programme in Computer Science (PCS)”

der Georg-August University School of Science (GAUSS)

vorgelegt von

Stefan Siemer

aus Hameln

Göttingen, 2023

Date of the Oral Examination: January 31, 2024

Acknowledgments

First and foremost, I hereby express my sincere gratitude and thank my supervisor, Florin Manea, for the excellent guidance and support throughout my studies. Florin is very knowledgeable in many areas of theoretical computer science, encouraging and helpful with new research ideas, and a highly cheerful person throughout my entire studies and everyday work. I am very grateful that he is swift in answering requests and always finds time to discuss research.

Secondly, I would like to thank my second supervisor, Carsten Damm. I have been working as a Tutor for Carsten since my Bachelor studies and enjoyed taking his advanced courses in theoretical computer science. Carsten also convinced me to pursue doctoral studies with the back then "new" professor at our institute, Florin Manea. Looking back, I am very grateful for all of these things. Further, I express my gratitude towards Jens Grabowski, who advised me very well with all the formal requirements as a member of the thesis advisory committee.

I also extend my many thanks to the entire working group. A special thank you goes to Maria Kosche, Tore Koß, and Paul Sarnighausen-Cahn as my future academic siblings, co-authors, and thesis proofreaders. I am thankful for the amicable and relaxed work environment I experienced with the previously mentioned group members and our master students Tina Ringleb, Timo Specht, and Maximilian Winkler, who work on the same floor. Speaking of people working on the same floor, I want to express my deepest gratitude to Henrik Brosenne, whom I closely collaborated with in teaching, as well as Heike Jachinke and Patricia Nitzke, who were always helping us regarding formal problems, such as traveling, room bookings, and much more, making it exceptionally convenient to deal with.

I am very grateful to have met many co-authors and collaborators in the last couple of years. First, I want to thank Pamela Fleischmann, Mitja Kulczynski, Dirk Nowotka, and Max Wiedenhöft from the University of Kiel for their excellent research cooperation. Secondly, I want to thank Vijay Ganesh and Zhengyang (John) Lu for the joint work on the *Z3Str** solvers, and I am deeply grateful that you invited me to your group in Waterloo (Canada) for a research stay. Further, I want to thank Joel D. Day and Markus L. Schmid as regular visitors and good collaborators of our group. It is my honor to extend my thanks to Paweł Gawrychowski, my most common co-author outside of the working group, whose immense knowledge of algorithms and data structures is truly inspiring.

I want to thank my family and friends. Let me start with Henrik Wiechers and Jonas Lueg, who have been my good companions from my early school days to our doctoral studies. Next, I want to mention Arvid Preuß and Niklas Stöcker, whom I am honored to call my friends since the very early days of my childhood. Let me continue with Bakhodir Ashirmatov, Kevin Freeman, and Christoph Rauterberg, who became my good friends during my studies in Göttingen. I want to thank everybody else I consider a part of my circle of friends and was not explicitly mentioned. Further, I want to give a big thank you to my parents, Elisabeth and Werner, who always supported me in every part of my life, my brothers Michael, Ralf, and Max, as well as my sister-in-law Frida and my lovely nephews Nils and Otto, for the good relationship and strong family bonds.

Last but not least, it is important for me to express my deepest gratitude to my partner, Linda, for her support in many parts of my life. Ultimately, I want to thank our cats, Amy and Lilly, for letting me write a few sentences now and then to finish this document.

Contributions

In this overview, I present chronologically my contributions to articles and projects during my doctoral studies. The articles Paper 2 [50], Paper 3 [78], Paper 6 [79], and Paper 7 [65] that make up the chapters of this thesis were chosen based on two criteria. First and most important, the proportion of my contribution to that project. The second criterion is the thematic connection of the articles to each other to form a coherent story.

Paper 1

Reference: P. Gawrychowski, M. Kosche, T. Koß, F. Manea, and S. Siemer. Efficiently Testing Simon's Congruence. In M. Bläser and B. Monmege, editors, *38th International Symposium on Theoretical Aspects of Computer Science, STACS 2021, March 16-19, 2021, Saarbrücken, Germany (Virtual Conference)*, volume 187 of *LIPIcs*, pages 34:1–34:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPICS.STACS.2021.34`

Description: We solved the problem of finding the largest k, for which two given words are k-Simon congruent, in optimal linear time.

Contribution: I contributed to the design of novel data structures and their usage in our algorithms, as well as to the general write-up of this paper.

Paper 2

Reference: J. D. Day, P. Fleischmann, M. Kosche, T. Koß, F. Manea, and S. Siemer. The edit distance to k-subsequence universality. In M. Bläser and B. Monmege, editors, *38th International Symposium on Theoretical Aspects of Computer Science, STACS 2021, March 16-19, 2021, Saarbrücken, Germany (Virtual Conference)*, volume 187 of *LIPIcs*, pages 25:1–25:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPICS.STACS.2021.25`

Description: We started to investigate for two words that are not $\sim_k$-congruent what is the minimal number of edit operations (edits distance) that we need to perform on the first word to obtain two $\sim_k$-congruent words? We showed combinatoric and algorithmic results on the particular case of the second word being a k-universal word with a compatible alphabet.

Contribution: In this paper, I was a main contributor for developing the data structures and algorithms regarding the changing of the universality index of words via the insertion and substitution operations. Further, I presented this work at *STACS 2021* in Saarbrücken (Germany) and several workshops. A video of my talk can be found on YouTube under this URL (`https://www.youtube.com/watch?v=YkRy9WYW8EQ&ab_channel=SaarlandInformaticsCampus`).

Paper 3

Reference: P. Gawrychowski, F. Manea, and S. Siemer. Matching Patterns with Variables Under Hamming Distance. In *46th International Symposium on Mathematical Foundations of Computer Science, MFCS 2021*, volume 202 of *LIPIcs*, pages 48:1–48:24, 2021. `doi:10.4230/LIPIcs.MFCS.2021.48`

Description: We provided our first generalized setting for the problem of matching patterns with variables by allowing mismatches between the target word and the pattern (under any substitution). The number of mismatches, known as the Hamming distance, must be either bound by an integer given as input (decision variant) or minimal (minimization variant) to be a valid substitution. We obtained algorithmic lower and upper bounds for various classes of patterns.

Contribution: I was a main contributor to identifying the problem described in this paper, the design of the algorithms, and their write-up in the paper. Further, I presented this work at *MFCS 2021* in Tallinn (Estonia) and several workshops.

Paper 4

Reference: M. Kosche, T. Koß, F. Manea, and S. Siemer. Absent subsequences in words. *Fundam. Informaticae*, 189(3-4):199–240, 2022. `doi:10.3233/FI-222159`

Description: We investigated the shortest and minimal (with respect to the subsequence relation) absent subsequences (i.e., words which are not a subsequence) in a word. This is an invited extended version of a paper presented at RP 2021.

Contribution: I contributed to the initial stages of this paper, in defining and sketching parts of the ideas which were later extended to the final algorithms presented there.

Paper 5

Reference: M. Kosche, T. Koß, F. Manea, and S. Siemer. Combinatorial algorithms for subsequence matching: A survey. In H. Bordihn, G. Horváth, and G. Vaszil, editors, *Proceedings 12th International Workshop on Non-Classical Models of Automata and Applications, NCMA 2022, Debrecen, Hungary, August 26-27, 2022*, volume 367 of *EPTCS*, pages 11–27, 2022. `doi:10.4204/EPTCS.367.2`

Description: This paper is a survey paper on our group's results on problems related to subsequences.

Contribution: I contributed to this survey paper by selecting and adjusting previous results and presenting them as part of a coherent overview of the results of our group on this topic.

Paper 6

Reference: P. Gawrychowski, F. Manea, and S. Siemer. Matching Patterns with Variables Under Edit Distance. In D. Arroyuelo and B. Poblete, editors, *String Processing and Information Retrieval - 29th International Symposium, SPIRE 2022, Concepción, Chile, November 8-10, 2022, Proceedings*, volume 13617 of *Lecture Notes in Computer Science*, pages 275–289. Springer, 2022. `doi:10.1007/978-3-031-20643-6\_20`

Description: This paper provided a natural follow-up to the setting from Paper 3 [78]. Instead of only considering mismatches (single letter substitutions) between the target word and the pattern, we extended the setting to allow for insertions and deletions of letters. The amount of these operations (commonly known as the edit distance) to get from the target word to the input pattern (under any substitution) must be either bound by an integer in the input (decision variant) or has to be minimal (minimization variant) in order to be a valid substitution. We obtained algorithmic lower and upper bounds for various classes of patterns.

Contribution: I was a main contributor to identifying the problem described in this paper, the design of the algorithms, and their write-up in the paper. Further, I presented this work at *SPIRE 2022* in Concepción (Chile) and in several workshops.

Paper 7

Reference: P. Fleischmann, S. Kim, T. Koß, F. Manea, D. Nowotka, S. Siemer, and M. Wiedenhöft. Matching Patterns with Variables Under Simon's Congruence. In O. Bournez, E. Formenti, and I. Potapov, editors, *Reachability Problems - 17th International Conference, RP 2023, Nice, France, October 11-13, 2023, Proceedings*, volume 14235 of *Lecture Notes in Computer Science*, pages 155–170. Springer, 2023. `doi:10.1007/978-3-031-45286-4\_12`

Description: This paper combined two settings we extensively studied in the group. On the one hand, we have Simon's congruence, which asks for the equivalence of the respective sets of subsequences (up to a specific length) of two words. On the other hand, we have the problem of matching patterns with variables, which was considered in an approximate setting under string metrics in Paper 3 [78] and Paper 6 [79]. When we combine these two concepts, we ask for a substitution of the variables in our pattern to reach a word that is k-Simon congruent to a target word. Contrary to the metrics, this gives us a relation that describes a similarity of the pattern (under a substitution) and our target word. This problem originated from a joint workshop of our group in Göttingen and the *Dependable Systems* group in Kiel, where the co-authors Pamela Fleischmann and Max Wiedenhöft proposed the problem.

Contribution: I contributed to the complexity results obtained in this paper and was a main contributor to the NP-hardness reductions.

Paper 8

Reference: D. Adamson, M. Kosche, T. Koß, F. Manea, and S. Siemer. Longest common subsequence with gap constraints. In A. E. Frid and R. Mercas, editors, *Combinatorics on Words - 14th International Conference, WORDS 2023, Umeå, Sweden, June 12-16, 2023, Proceedings*, volume 13899 of *Lecture Notes in Computer Science*, pages 60–76. Springer, 2023. doi:10.1007/978-3-031-33180-0_5

Description: We investigated the problem of computing the longest common subsequence with gap constraints of two input words. We analyzed various settings of gap constraints and provided efficient algorithms for computing the longest common subsequence in each. This paper was invited to the special issue dedicated to the best papers of WORDS 2023, to appear in Theory of Computing Systems.

Contribution: I contributed to this paper by identifying some of the discussed gap constraints models and designing initial algorithms for all these settings. These all became parts of the final algorithms. Further, I was a main contributor to the write-up of several sections, and I presented this work at *WORDS 2023* in Umeå (Sweden).

String-Solver at SMT-COMP 2023

Reference: S.-C. 2023. The international satisfiability modulo theories (smt) competition. URL: https://smt-comp.github.io/2023/

Description: We develop and work on extensions and improvements for the series of string solvers *Z3Str**, which are branches of Microsofts *SMT* solver *Z3*. The newest iteration *Z3Alpha* won a track in the section *QF_Strings* (Single Query Track) in the SMT competition mentioned above. This project has yet to produce a paper, but it is planned to write papers on this topic shortly.

Contribution: I contributed to this project by identifying novel approaches and implementing and testing these string-solving tactics within our solvers.

Abstract

In the literature dealing with patterns with variables, a word, also called string, is a sequence of terminal letters, while a pattern is a sequence of terminal and variable letters. The problem of deciding if there is a substitution for all the variables in a pattern, such that a target word is obtained, is the matching problem for patterns with variables. In many problems related to the processing of textual data, it is essential to model uncertainty in the text, such as, e.g., typos in handwritten texts or mutations in biological data. For this reason, I introduce and present in this thesis our proposals of several settings which model uncertainty in the context of matching patterns with variables and a series of algorithmic and complexity theoretic results developed in these settings. The first setting, approached in Chapter 3, is concerned with matching patterns with variables under Hamming distance, that is, deciding if there is a substitution for all variables in a pattern, such that a word is obtained that is not "too far" from the target word with respect to the Hamming distance. The same problem is analyzed in Chapter 4 for the Edit distance and in Chapter 5 for a similarity measure based on k-Simon's congruence. The final problem in Chapter 6 is concerned with the edit distance of a word to a k-universal word, i.e., a word which contains all possible words of length k as subsequences.

Contents

CHAPTER 1

Introduction

This Chapter is divided into four sections. In the first Section 1.1, I will introduce the original problem of matching patterns with variables, which is the central underlying concept for the topics discussed in this thesis. Further, in Section 1.2, I will provide related work and motivation to study the problems of matching patterns with variables and their approximate extensions. Following the motivation, I will present an overview of the extensions of the original problem in Section 1.3, which will be discussed in later chapters of this thesis. In the last Section 1.4, I will outline the structure of the thesis.

1.1 The original problem

A *pattern with variables* is a string consisting of *constant* or *terminal letters* from a finite set, the alphabet Σ (e.g., $\Sigma := \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$), and a potentially infinite set $\mathcal{X}$, the *variables* (e.g., $\mathcal{X} := \{x, y, x_1, x_2\}$) with $\Sigma \cap \mathcal{X} = \emptyset$. In other words, a pattern α is an element of $PAT_\Sigma := (\mathcal{X} \cup \Sigma)^+$. A pattern α is mapped by a *morphism* $h : PAT_\Sigma \to \Sigma^*$, henceforth called *substitution*, to a word by substituting the variables in α by arbitrary strings of terminal letters over Σ and leaving all the terminals in α unchanged. If all the variables from α can be mapped by a substitution h so that $h(\alpha) = w$, then h is a *solution* for which α matches w (see Example 1.1). The following decision problem is called the *(exact) matching problem*: given a pattern α and a word w, does a substitution h exist, such that it matches α to w ($h(\alpha) = w$)?

Exact Matching Problem: `Match`
Input: A pattern $\alpha \in PAT_\Sigma$, a word $w \in \Sigma^*$.
Question: Is there a substitution h with $h(\alpha) = w$?

Example 1.1. $\alpha = \mathtt{x_1x_1bbbx_2x_2}$, $w = \mathtt{aaaabbbbb}$

$$h(\mathtt{x_1}) = \mathtt{aa} \qquad h(\alpha) = \mathtt{aaaabbbbb}$$
$$h(\mathtt{x_2}) = \mathtt{b} \qquad w = \mathtt{aaaabbbbb}$$

1.2 Motivation and related work

The matching problem `Match` for patterns with variables appears in various theoretical and practical areas of computer science. In particular, `Match` is a restricted case of the satisfiability problem for *word equations*, where one is given two patterns α and β and is interested in finding a substitution h that maps both patterns to the same word (see, e.g., [123]). Therefore, `Match` is a *word equation* where one side is restricted to contain only terminal letters.

Word Equations:
Input: Two patterns $\alpha, \beta \in PAT_\Sigma$.
Question: Is there a substitution h with $h(\alpha) = h(\beta)$?

Example 1.2. $\alpha = \mathtt{ax_1abx_2abx_1b}$, $\beta = \mathtt{x_2bax_1aabbx_1}$

$$h(\mathtt{x_1}) = \mathtt{b} \qquad h(\alpha) = \mathtt{abababaabbb}$$
$$h(\mathtt{x_2}) = \mathtt{aba} \qquad h(\beta) = \mathtt{abababaabbb}$$

Patterns with variables (as a restricted case of word equations) frequently come up in the area of combinatorics on words (e.g., unavoidable patterns [124]), stringology (e.g., generalized function matching [9, 136]), language theory (e.g., pattern languages [10]), algorithmic learning theory (e.g., the theory of descriptive patterns for finite sets of words [155, 10, 58]), database theory (e.g., document spanners [69, 67, 56, 150, 105, 151]), or extended regular expressions with backreferences [38, 71, 66, 70]. There are several extensions and variations of matching patterns with variables and word equations. A prominent family of problems, that is intuitively speaking in between matching patterns and word equations, is the matching of patterns against formal languages, e.g., regular languages [26, 27, 25, 55, 125, 111].

Moreover, the problem of solving word equations is not only central to the area of combinatorics on words [123] and other theoretical areas but also frequently occurs in the more practical area of string solving [7, 83]. An intuitive definition of *string solving* is the process of reasoning algorithmically in logics over strings, string functions, and relations of strings. To be more precise, it describes the process of checking the satisfiability of first-order formulas concerning a logical theory over strings. The functions in this theory, also called *constraints*, include, e.g., linear arithmetic on the length of strings, concatenation and equality of strings, language-membership (e.g., regular languages), and many more (see Theory of Strings in [18]). The main question that is answered in string solving is: for a formula (word equations + constraints), does a substitution exist of the variables of that formula, such that it evaluates to true? In practical scenarios, the most common theories used to build string-solving problems are specified using the *SMT* (**S**atisfiability **M**odulo **T**heories) standard defined in [18]. The purpose of *SMT* is to provide a declarative approach not only to string solving (theory of strings) but also for, e.g., the theory on integers, reals, and arrays [18]. These formulas are then evaluated by SMT-solvers (also called string solvers, if they focus on the theory of strings), of which the most prominent ones are *Z3* [53], *CVC4* [19], *CVC5* [16] and *OSTRICH* [137]. In the Listing 1.1, a word equation example is written in the standard *SMT* language *SMT-LIB2*. The functions in the first four lines, e.g., (declare −fun X1 () String), declare string variables, e.g., $X1$ in this example. Afterwards, the two patterns A and B are defined using the equality function (=) and the string concatenation function (str .++). A word equation is formed at the end of the formula using the equality function to set A and B equal.

```
(declare-fun X1 () String)
(declare-fun X2 () String)
(declare-fun A () String)
(declare-fun B () String)
(assert (= A (str.++ "aab" (str.++ X1 "a"))))
(assert (= B (str.++ X2 (str.++ "ba" X1))))
(assert (= A B))
(check-sat)
```

Listing 1.1: The word equation $aabx_1a = x_2bax_1$ written in SMT-LIB2.

In the Listing 1.2, an example formula with more complex constraints is displayed. The functions (>=) and (str . len) are used to model a length constraint in order to make sure that $X1$ is larger than two. With (str . in_re) and (str . to_re), it is made sure that $X1$ is in the regular language that consists of only one word "aa". The core word equation $A = B$ in this formula is an example of a pattern matching problem, as B consists of only terminal letters.

```
(declare-fun X1 () String)
(declare-fun X2 () String)
(declare-fun A () String)
(assert (>= (str.len X1) 2))
(assert (str.in_re X2 (str.to_re "aa")))
(assert (= A (str.++ X2 (str.++ "ba" X1))))
(assert (= B "aabbbaaa"))
(assert (= A B))
(check-sat)
```

Listing 1.2: The formula $|x_1| \geq 2 \wedge x_2 \in \{aa\} \wedge x_2bax_1 = aabbbaaa$ written in SMT-LIB2.

The problems from our papers [78] (presented in Chapter 3) and [79] (presented in Chapter 4) originated from the idea of adding string distance functions as an alternative to the equality function in such SMT-formulas. It was then decided to focus first on the theoretical background of the restricted problem of *matching patterns with variables in approximate settings*.

Considering string problems under string distances is natural, as most real-world string-based data is prone to errors due to, e.g., mutations in genetic data or typos in handcrafted texts. Even though many concepts are already from the 50's and 60's (*Hamming distance* [85] and *edit distance* [121, 120]), it is a very active area of research with a long history of results as it can be seen in, e.g., the recent papers [42, 81, 80, 161], and the references therein, as well as classical results such as [8, 132, 114]. Many prominent stringology problems have been considered under the *edit distance*, e.g., [115, 44, 24, 131, 22, 43, 41, 133]. Closer to the problems presented in this thesis, the approximate matching problem was also considered for regular expressions, e.g., in [28, 132].

Another concept for modeling the lossy representation of strings is subsequences [159]. A string u is a *subsequence* (in the literature also called *scattered factor* or *subword*) of a string w if there exists a strictly increasing integer sequence $0 < i_1 < i_2 < \ldots < i_{|u|} \leq |w|$ with $w[i_j] = u[j]$ for all $j \in [|u|]$. Subsequences are also a heavily investigated topic in the area of word combinatorics, string algorithms, and combinatorial pattern matching, and are connected to other areas of computer science (see, e.g., in the Chapter *Subwords* by J. Sakarovitch and I. Simon of the standard textbook [123] or the survey [107] and the references therein). In theoretical computer science, one can often encounter subsequences and their generalizations; for instance, in logic of automata theory, subsequences are used in the context of piecewise testability [156, 157], in particular, to the height of piecewise testable languages [94, 95, 96], subword order [84, 113, 112], or downward closures [167]. In combinatorics on words, many concepts were developed around the idea of counting the occurrences of particular subsequences of a word, such as the k-binomial equivalence [144, 68, 119, 117], subword histories [153], and Parikh matrices [130, 146]. In the area of algorithms, subsequences appear, e.g., in classical problems such as the longest common subsequence [15, 33, 36], the shortest common supersequence [126], or the string-to-string correction [163].

In the 70's, Imre Simon initiated a line of research on subsequences that studies the set of all subsequences of a particular word length [156]. An intriguing concept from these studies was the relation $\sim_k$, which is nowadays called *Simon's congruence* [157, 123]. Let $\mathbb{S}_k(w)$ be the set of all subsequences of a given string w up to length $k \in \mathbb{N}_0$. Two strings w_1 and w_2 are *k-Simon congruent* ($w_1 \sim_k w_2$), iff $\mathbb{S}_k(w_1) = \mathbb{S}_k(w_2)$. Simon's congruence, which was originally introduced to study *piecewise testable events*, is now established in the study of piecewise testable languages, a special class of regular languages with applications in learning theory, databases theory, or linguistics (see, e.g., [96] and the references therein). More information and a broad overview of the work of Imre Simon can be found in the two survey papers [140, 141]

Next to many interesting combinatorial properties, Simon's congruence is a string similarity measure. The larger the maximal k for which two words are k-Simon congruent, the more similar the two words are [156]. For this reason, the computation of deciding for a given k whether two words are $\sim_k$ congruent, as well as the corresponding maximisation problem were the subject of a long line of research in the combinatorial pattern matching community [87, 73, 158, 160, 49, 62]. At last, these problems were solved in optimal linear time in [17, 76]. In recent research, an interesting algorithmic problem was tackled in the article [104], which is related to the problem of *matching patterns with variables under Simon's congruence* [65] (Chapter 5 of this thesis). In that paper, the authors present an efficient solution for the following problem: given two strings w, u and a natural number k, decide whether there exists a factor $w[i : j]$ with arbitrary i, j of w which is k-Simon congruent to u. In a framework that uses patterns with variables under Simon's congruence, this can be modeled with the pattern $\alpha = x_1 u x_2$ for variables x_1, x_2. Therefore, it is reasonable to investigate the general setting: given a pattern α and string w, does there exist a substitution h, such that $h(\alpha) \sim_k w$?

A specific congruence class defined by $\sim_k$ frequently arises in the literature: the class of k-subsequence *universal* words consists of words that contain all possible k-length words over a fixed alphabet as subsequences. The maximal k for which a word w is k-subsequence universal is denoted by the *universality index* $\iota(w) = k$. This particular class was first studied in [95, 97], and is related to the more general problem of fixed-length universality: for a given k and alphabet Σ, is Σ^k described/generated/accepted by a mechanism which describes/generates/accepts languages? Such problems were investigated in contexts related to and motivated by formal languages, automata theory, or combinatorics, where the notion of universality is central [17, 50, 63, 106, 2, 3, 152, 64]. In [142, 110, 77] and the references therein, the authors discuss many variants and results of the universality problem for various language-generating and accepting formalisms. The universality problem was considered for words [128, 52] and partial words [45, 82] with respect to their factors. More precisely, one is interested in finding, for a given integer ℓ, a word w over an alphabet Σ, such that each word of length ℓ over Σ occurs exactly once as a contiguous factor of w. The De Bruijn sequences [52] fulfill this property and have many applications in computer science or combinatorics [45, 82]. The notion of k-subsequence universal words is going to play an important role in proving a lower bound for *matching patterns with variables under Simon's congruence* in [65], as well as being the obvious central notion of the problem *the edit distance to k-subsequence universality* from [50] (Chapter 5 and 6 from this thesis).

Coming back to the study of patterns with variables. Unfortunately, the *(exact) matching problem* is in general already NP-complete [10]. Thus, all the problems based on solving the matching problem as a subroutine inherit this intractability. One of these problems is the computation of *descriptive patterns* for finite sets of words [10, 58], which is located in the area of algorithmic learning theory. These descriptive patterns are a prominent example of a language class that can be inferred from *positive data* (see the survey [155] and the references therein). Further, the new approach of *matching patterns with variables in approximate settings* can be used to test: given a learned language and an input word, compute how "far" apart the word is from the language. These use cases are motivation to identify tractable cases of the matching problem, as well as for the approximate setting. A natural approach to this task is to consider restricted classes of patterns. A thorough analysis of the complexity of the matching problem has provided several subclasses of patterns for which the matching problem is in P when some structural parameters of patterns are bounded by constants [143, 154, 60, 61, 59, 149]. Prominent examples in this direction are patterns with a bounded number of repeated variables, patterns with bounded scope coincidence degree [143], patterns with bounded locality [51], or patterns with a bounded treewidth [143]. The formal definitions of these parameters are given in Section 2.2, and corresponding efficient matching algorithms can be found in [59, 51, 143] and are overviewed in Section 2.3 of this thesis. These numerical parameters, which describe the structure of a class of patterns, also parameterize the complexity of the corresponding matching algorithms. More precisely, in all cases, if the value of the parameter equals k, the matching algorithm runs in $O(n^{ck})$ for some constant c, and the matching problem is $W[1]$-hard w.r.t. the parameter. A more general approach from [143] introduces the notion of treewidth of patterns and shows that the matching problem can be solved in $O(n^{2k+4})$ time for patterns with bounded treewidth k. The algorithms resulting from this general theory are less efficient than the specialized ones, while the matching problem remains $W[1]$-hard w.r.t. treewidth of patterns. For more details on this, I refer to the survey [127].

1.3 The extended matching problem

In the upcoming chapters of this thesis, I am going to discuss and present our results on an extended setting of the matching problem for patterns with variables: given a pattern α and a word w, decide if there exists a substitution h such that $h(\alpha)$ is similar to w, with respect to a similarity measure

(Hamming resp. edit distance in [78, 79], and Simon's congruence [65], as well as a comparison to the original problem, string equality for (exact) `Match`). This modification to the problem is called *(approximate) pattern matching*.

The immediate natural choices for similarity measures between strings are *string metrics*. A smaller distance between two strings indicates higher similarity between them. In these metric settings, one is given a pattern α, a word w, and (in most cases) a natural number Δ. Then, it is required to decide if there exists a (alternatively, to find the minimizing) substitution h such that $D(h(\alpha), w) \leq \Delta$, where D is a *distance function* (see problems below). Among the most prominent metrics on strings are the Hamming distance and the edit distance. In Chapter 3, I will present our results where D is the Hamming distance (published on the 46th International Symposium on Mathematical Foundations of Computer Science) [78]. In Chapter 4, I will present our results for D being the edit distance (published on the 29th International Symposium on String Processing and Information Retrieval) [79]. In both chapters, the matching problem is discussed for the general setting and for structurally restricted classes of patterns.

Approximate Matching Decision Problem under Distance D

Input: A pattern $\alpha \in PAT_\Sigma$, a word $w \in \Sigma^*$, an integer Δ.

Question: Is there a substitution h with $D(h(\alpha))w \leq \Delta$?

Approximate Matching Minimisation Problem under Distance D

Input: A pattern $\alpha \in PAT_\Sigma$, a word $w \in \Sigma^*$.

Question: Compute h that minimizes $D(h(\alpha))w$.

Other very popular measures to define similarity between two strings are the (k-)*abelian equivalence* [98, 99], *k-binomial equivalence* [144, 68, 118] or the previously mentioned Simon's congruence $\sim_k$ [157]. In all of these relations, a numerical parameter k indicates that two k-equivalent strings are more similar when k is larger. In Chapter 5, I will present our results on problems related to Simon's congruence. The first problem involves finding a substitution h, such as $h(\alpha) \sim_k w$. Alternatively, the problem of finding a substitution h to reach a target universality index k, i.e., $\iota(h(\alpha)) = k$ is considered. Lastly, in the case of word equations, it is examined how to find a substitution h for two patterns α and β, such that $h(\alpha) \sim_k h(\beta)$ (published at the 17th International Conference on Reachability Problems) [65].

Matching under Simon's Congruence: `MatchSimon`(α, w, k)

Input: Pattern $\alpha \in PAT_\Sigma$, word $w \in \Sigma^*$, an integer $k \in \mathbb{N}_0$.

Question: Is there a substitution h with $h(\alpha) \sim_k w$?

Matching a Target Universality: `MatchUniv`(α, k)
Input: Pattern $\alpha \in PAT_\Sigma$, an integer $k \in \mathbb{N}_0$.
Question: Is there a substitution h with $\iota(h(\alpha)) = k$?

Word Equations under Simon's Congruence: `WESimon`(α, β, k)
Input: Patterns $\alpha, \beta \in PAT_\Sigma$, an integer $k \in \mathbb{N}_0$.
Question: Is there a substitution h with $h(\alpha) \sim_k h(\beta)$?

In all of the considered settings, it is possible to simulate the classical (exact) `Match` problem by either setting the distance Δ to zero or the k of the $\sim_k$ relation to the length of our target word w. Therefore, all the presented results are generalizations of the `Match` problem. The results from these papers are compactly summarized in Chapter 7.

1.4 Structure of the thesis

First of all, note that the core of this thesis are the Chapters 3, 4, 5 and 6, which are slightly modified versions, but essentially unchanged core content from our published papers [78, 79, 65, 50], respectively, to all of which I contributed as a co-author. In most parts, I altered some of the notations and merged all their respective preliminaries and introductions into one section to provide a concise presentation of all those thematically well-connected results in one thesis. At the beginning of each Chapter, I will explicitly state the scope of my contribution to the respective publication.

In Chapter 2, I provide, in addition to the basic notations of this thesis, all necessary definitions, data structures, and methodology used across all chapters unless explicitly stated otherwise in the respective section.

In Chapter 3, I present our results from the paper *Matching Patterns with Variables under Hamming Distance*, that was published on the *46th International Symposium on Mathematical Foundations of Computer Science*. This chapter is concerned with upper and lower complexity bounds for the problems `HDMatch` and `MinHDMatch` for various pattern classes.

In Chapter 4, I present our results from the paper *Matching Patterns with Variables under Edit Distance*, which was published on the *29th International Symposium on String Processing and Information Retrieval*. This chapter is concerned with upper and lower complexity bounds for the problems `EDMatch` and `MinEDMatch` for various classes of pattern.

In Chapter 5, I present our results from the paper *Matching Patterns with Variables under Simon's congruence*, that was published on the *17th International Conference On Reachability Problems*. This chapter is concerned with upper and lower complexity bounds for the problems `MatchSimon`, `MatchUniv`, and `WESimon`.

In Chapter 6, I present our results from the paper *The Edit Distance to k-Subsequence Universality*, that was published on the *38th International Symposium on Theoretical Aspects of Computer Science*. This chapter leaves the context of matching patterns with variables but stays in the broader context of word-to-language distance problems. To be more precise, we studied the edit distance between a word w and the language $L_{u,k}$ of words which are $\sim_k$-equivalent to a word u. From this general setting, we efficiently solved the following nontrivial and well-motivated problem: given the word w and a number k, compute the minimum number of edit operations applied to w to obtain a k-universal word (w.r.t. the alphabet of w). An extra motivation and some individual notions regarding this problem are introduced in the respective overview Section 6.1 to illustrate the importance of this work.

Finally, I provide a general overview of the results and concluding remarks, and an outlook on future work and open problems in Chapter 7.

CHAPTER 2

Preliminaries

The preliminaries of this thesis provide the basic notations and essential concepts that will appear in the upcoming chapters. As the chapters are based on a series of thematically related papers, these preliminaries consist of a slightly adapted selection of all the crucial concepts from these papers in a compact, conflated manner. Hence, this chapter is based on the preliminaries of the following articles:

P. Gawrychowski, F. Manea, and S. Siemer. Matching Patterns with Variables Under Hamming Distance. In *46th International Symposium on Mathematical Foundations of Computer Science, MFCS 2021*, volume 202 of *LIPIcs*, pages 48:1–48:24, 2021. `doi:10.4230/LIPIcs.MFCS.2021.48`

P. Gawrychowski, F. Manea, and S. Siemer. Matching Patterns with Variables Under Edit Distance. In D. Arroyuelo and B. Poblete, editors, *String Processing and Information Retrieval - 29th International Symposium, SPIRE 2022, Concepción, Chile, November 8-10, 2022, Proceedings*, volume 13617 of *Lecture Notes in Computer Science*, pages 275–289. Springer, 2022. `doi:10.1007/978-3-031-20643-6\_20`

P. Fleischmann, S. Kim, T. Koß, F. Manea, D. Nowotka, S. Siemer, and M. Wiedenhöft. Matching Patterns with Variables Under Simon's Congruence. In O. Bournez, E. Formenti, and I. Potapov, editors, *Reachability Problems - 17th International Conference, RP 2023, Nice, France, October 11-13, 2023, Proceedings*, volume 14235 of *Lecture Notes in Computer Science*, pages 155–170. Springer, 2023. `doi:10.1007/978-3-031-45286-4\_12`

J. D. Day, P. Fleischmann, M. Kosche, T. Koß, F. Manea, and S. Siemer. The edit distance to k-subsequence universality. In M. Bläser and B. Monmege, editors, *38th International Symposium on Theoretical Aspects of Computer Science, STACS 2021, March 16-19, 2021, Saarbrücken, Germany (Virtual Conference)*, volume 187 of *LIPIcs*, pages 25:1–25:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPICS.STACS.2021.25`

2.1 General Notation

Let $\mathbb{N} = \{1, 2, \ldots\}$ be the set of natural numbers (*integers*) and $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$. Let $[n] = \{1, \ldots, n\}$ and $[m : n] = [n] \setminus [m-1]$, for $m, n \in \mathbb{N}$, with $m < n$. Let $\Sigma = [\sigma]$ be a finite alphabet of *terminal letters* or *symbols* (in most examples the letters $\{\mathtt{a}, \mathtt{b}, \ldots\}$ or simply integers $\{1, 2, \ldots\}$ are used). Let Σ^* be the set of all words (strings, sequences) under the concatenation of letters and ε the empty word. In an algebraic framework, the structure $(\Sigma^*, \cdot, \varepsilon)$ is the free monoid over Σ with concatenation $\cdot$ as the binary operation and ε the neutral element. The concatenation of k words $w_1, w_2, \ldots, w_k$ is written $\Pi_{i=1}^{k} w_i$. The set Σ^+ is defined as $\Sigma^* \setminus \{\varepsilon\}$. For $w \in \Sigma^*$, the length of w is defined by the number of letters of w and denoted as $|w|$. Further for an integer k, let $\Sigma^k = \{w \in \Sigma^* \mid |w| = k\}$ and $\Sigma^{\leq k} = \bigcup_{i=0}^{k} \Sigma^i$. The letter on position i of w, for $1 \leq i \leq |w|$, is denoted by $w[i]$. For $w \in \Sigma^+$ and $x, y, z \in \Sigma^*$, the word y is called a *factor* of w, if $w = xyz$; moreover, if $x = \varepsilon$ (respectively, $z = \varepsilon$), then y is called a *prefix* (respectively, *suffix*) of w. Let $w[i : j] = w[i] \cdots w[j]$ be the factor of w starting on position i and ending on position j; if $i > j$ then $w[i : j] = \varepsilon$. Similarly, for an array A, $A[i : j]$ denotes a subarray of A whose positions are indexed by the numbers in $[i : j]$.

Let $\mathcal{X} = \{x_1, x_2, x_3, \ldots\}$ be a set of *variables*. The set of terminal letters Σ and the set of variables $\mathcal{X}$ are disjoint $\Sigma \cap \mathcal{X} = \emptyset$. A pattern α is a word containing terminals and variables, i.e., an element of $PAT_\Sigma := (\mathcal{X} \cup \Sigma)^+$. The set of all patterns, over all terminal-alphabets, is denoted $PAT := \bigcup_\Sigma PAT_\Sigma$. Given a word or a pattern γ, for the smallest sets (w.r.t. inclusion) $B \subseteq \Sigma$ and $Y \subseteq \mathcal{X}$ with $\gamma \in (B \cup Y)^*$, define the set of terminal symbols in γ, denoted by $\mathtt{alph}(\gamma) = B$, and the set of variables of γ, denoted by $\mathtt{var}(\gamma) = Y$. For any symbol $t \in \Sigma \cup \mathcal{X}$ and $\alpha \in PAT_\Sigma$, $|\alpha|_t$ denotes the number of occurrences of t in α. For a pattern $\alpha = w_0 x_1 w_1 \ldots w_k x_k$ with $w_0, w_1, \ldots, w_k \in \Sigma^*$ and $x_1, x_2, \ldots, x_k \in \mathcal{X}$, the projection of α on the terminal alphabet Σ is denoted by $\mathtt{term}(\alpha) = w_0 w_1 \ldots w_k$ and the projection on the variable alphabet as the skeleton $\mathtt{skel}(\alpha) = x_1 x_2 \ldots x_k$.

A *substitution* of the variables from $\mathcal{X}$, $h : (\mathcal{X} \cup \Sigma)^* \to \Sigma^*$, is a *homomorphism* that acts as the identity on Σ and maps each variable of $\mathcal{X}$ to a (potentially empty) string over Σ. That is, $h(\mathtt{a}) = \mathtt{a}$ for all $\mathtt{a} \in \Sigma$ and $h(x) \in \Sigma^*$ for all $x \in \mathcal{X}$. A pattern α *matches* a string w over Σ w.r.t. a binary relation $\sim$ if there exists a substitution h that satisfies $h(\alpha) \sim w$.

If $\sim$ is the string equality $=$, the pattern α is said to *(exactly) match* the string w instead of saying that α *matches* w under $=$. Further, if it is obvious from the context, the substitution h can be omitted from the notion to get the shorter but less formal $\alpha \sim w$.

2.2 Pattern classes

A *pattern class* P is a subset of all patterns $PAT_\Sigma = (\mathcal{X} \cup \Sigma)^+$, such that all patterns $\alpha \in P$ have a structural property in common. Next to the more or less trivial numerical parameters, such as counting the variables or the number of distinct variables, more involved parameters are used to describe the structure of a pattern. One of these numerical parameters is the *scope coincidence* *degree* $\mathtt{scd}(\alpha)$ of a pattern α. The two definitions of scope and coincidence are required to define $\mathtt{scd}(\alpha)$. The scope of a variable $x \in \mathtt{var}(\alpha)$ is defined by $\mathtt{sc}_\alpha(x) = [i : j]$, where i is the leftmost and j the rightmost occurrence of x in α. The variables $x_1, \ldots, x_k \in \mathtt{var}(\alpha)$ coincide in α if $\bigcap_{i=1}^{k} \mathtt{sc}(x_i) \neq \emptyset$. Now, the scope coincidence degree of a pattern $\mathtt{scd}(\alpha)$ is the maximal number k of variables that coincide. This defines the class $\mathtt{kSCD}$, where all pattern α have a bounded $\mathtt{scd}(\alpha) \leq k$. Further, there is a similar notion of k-bounded patterns; these are all patterns that have their treewidth bounded by k if represented as an undirected graph that has all symbols of the pattern as nodes and edges between consecutive symbols as well as between equal variables (for a detailed definition see [143]). In the following examples, the scope of each variable is presented as an underline from the leftmost to the rightmost occurrence of each variable; hence, the scope coincidence degree of the respective pattern is intuitively given by the maximal amount of stacked lines at any position.

Another nontrivial numerical parameter of a pattern is the notion of *k-locality*. Let α be a pattern with p distinct variables. A *marking sequence* of α is an ordering $x_1 < x_2 < \ldots < x_p$ of all variables that is defined on $\mathtt{skel}(\alpha)$ by the following procedure: in step i mark all occurrences of the variable x_i in $\mathtt{skel}(\alpha)$. A pattern α is then called k-local iff there exists a marking sequence of $x_1 < x_2 < \ldots < x_p$ such that, for i from 1 to p, the variables marked in the first i steps of the marking of $\mathtt{skel}(\alpha)$ form at most k non-overlapping length-maximal factors. The class of all k-local patterns is denoted by $\mathtt{kLOC}$. See [51, 39] for an extended discussion and examples regarding k-locality.

In the following, I will define particular classes of interest and how they relate. Let me start with the class of unary patterns $\mathtt{1Var}$. A pattern is *unary* or a one-variable pattern if there exists one variable $x_1 \in \mathtt{var}(\alpha)$ and additonally $|\mathtt{var}(\alpha)| = 1$. Clearly, $\mathtt{1Var}$ is a subclass of $\mathtt{1SCD}$ and also of $\mathtt{1LOC}$. The Example 2.1 shows a unary pattern α_1.

Example 2.1.

$$\alpha_1 = \mathtt{abcba}\underline{x_1 x_1 x_1 \mathtt{ab} x_1 x_1}\mathtt{bab} \in \mathtt{1Var}$$

Another pattern class of interest are the regular patterns $\mathtt{Reg}$. A pattern $\alpha \in \mathtt{Reg}$ is *regular* if it can be written $\alpha = w_0 \prod_{i=1}^{M}(x_i w_i)$, with $w_i \in \Sigma^*$ and the variables named in a canonical way; or alternatively the pattern fulfils for all $x_i \in \mathtt{var}(\alpha)$ that $|\alpha|_{x_i} = 1$. It is clear that $\mathtt{Reg} \subset \mathtt{1LOC}$ as every variable scope is just its position. Furthermore, to obtain a generic marking sequence, canonically rename the variables and mark them ascending w.r.t to their index. With this marking sequence, it is implied that $\mathtt{Reg} \subset \mathtt{1SCD}$. One of the most prominent patterns from the class of $\mathtt{Reg}$ is $\alpha = x_1 w_1 x_2$. The matching problem for this pattern simulates the *(classical) pattern matching* problem of finding a word within a text (e.g., when pressing *Ctrl+F* on the computer). It can be efficiently solved by the well known *Knuth-Morris-Pratt* (*KMP*) or *Rabin-Karp* algorithm. The name *regular* pattern stems from the fact that every variable can be substituted by an arbitrary word Σ^*; hence yielding an equivalent regular expression, as seen for α_2 in the Example 2.2.

Example 2.2.

$$\begin{aligned} \alpha_2 &= \mathtt{ab}\underline{x_1}\mathtt{cb}\underline{x_2}\mathtt{aab}\underline{x_3}\mathtt{bab} && \in \mathtt{Reg} \\ &= \mathtt{ab}\Sigma^*\mathtt{cb}\Sigma^*\mathtt{aab}\Sigma^*\mathtt{bab} && \in \mathit{RegEx} \end{aligned}$$

Intuitively, when combining the two previous classes of patterns into one class of pattern, the resulting class is called one-repeated-variable patterns $\mathtt{1RepVar}$. To be more precise, α is from $\mathtt{1RepVar}$ if there exists at most one variable $x_1 \in \mathtt{var}(\alpha)$ with $|\alpha|_{x_1} > 1$ and $|\alpha|_{x_i} = 1$ for $i > 1$. Clearly, $\mathtt{1RepVar} \supset \mathtt{1Var}$ and $\mathtt{1RepVar} \supset \mathtt{Reg}$. Furthermore, for a pattern $\alpha \in \mathtt{1RepVar}$ the $\mathtt{scd}(\alpha)$ is at most 2. That is the case when the occurrences of the repeated variable wrap at least one other variable. The Example 2.3 shows a pattern α_3 from $\mathtt{1RepVar}$.

Example 2.3.

$$\alpha_3 = \mathtt{ab}\underline{x_1\underline{x_2}\mathtt{ab}\underline{x_3}x_1x_1}\mathtt{baab} \in \mathtt{1RepVar}$$

An interesting problem from stringology that can be modeled with the pattern class $\mathtt{1RepVar}$ is the detection of t-repetitions (e.g., squares) in a word. The matching problem for the pattern α_{t-rep} in the Example 2.4 models this problem.

Example 2.4.

$$\alpha_{t-rep} = x_1 x_2^t x_3 \in \mathtt{1RepVar}$$

The next class to consider are the non-cross patterns $\mathtt{NonCross}$. A pattern α is called non-cross iff it can be written as the concatenation of multiple $\mathtt{1Var}$-patterns with pairwise distinct variables. By a similar argument as in the case of $\mathtt{Reg}$, the variables can be marked generically in their order of first appearance from left to right to get that $\mathtt{NonCross} \subset \mathtt{1LOC}$. Further, the variables are not interleaved and do, therefore, not coincide, such that $\mathtt{NonCross} = \mathtt{1SCD}$. See the Example 2.5 for a non-cross pattern.

Example 2.5.

$$\alpha_4 = \mathtt{a}\underline{x_1x_1}\ \underline{x_2}\mathtt{ab}\underline{x_3x_3x_3x_3}\mathtt{bb}\underline{x_4} \in \mathtt{NonCross}$$

The listing below sums up all the important relations of the introduced pattern. These relations are important as, e.g., hardness results for matching problems on the more restricted cases immediately imply the hardness of the more general pattern.

- $\mathtt{1Var} \subset \mathtt{1RepVar}$
- $\mathtt{1Var} \subset \mathtt{NonCross} \subset \mathtt{1LOC}$
- $\mathtt{Reg} \subset \mathtt{1RepVar}$
- $\mathtt{Reg} \subset \mathtt{NonCross} \subset \mathtt{1LOC}$
- $\mathtt{NonCross} \setminus \mathtt{1RepVar} \neq \emptyset$ and $\mathtt{1RepVar} \setminus \mathtt{NonCross} \neq \emptyset$ (hence they are incomparable)

To finish this overview of pattern classes, let me provide extra pattern to show the separation and inclusion of these classes in Example 2.6.

Example 2.6.

$$\begin{aligned}
\alpha_5 &= \mathtt{ab}\underline{x_1\mathtt{ab}x_1x_1}\mathtt{baab} && \in \mathtt{1Var}\\
\alpha_6 &= \mathtt{ab}\underline{x_1\underline{x_2}\mathtt{ab}\underline{x_3}x_1x_1}\mathtt{baab} && \in \mathtt{1RepVar} \setminus \mathtt{NonCross}\\
\alpha_7 &= \mathtt{ab}\underline{x_1x_1}\ \underline{x_2}\mathtt{ab}\underline{x_3x_3x_3}\mathtt{bb} && \in \mathtt{NonCross} \setminus \mathtt{1RepVar}\\
\alpha_8 &= \mathtt{ab}\underline{x_1x_1}\mathtt{a}\underline{x_2}\mathtt{ab}\underline{x_3}\mathtt{bba} && \in \mathtt{NonCross} \cap \mathtt{1RepVar}
\end{aligned}$$

2.3 The matching problem

Let α be a pattern from a class of patterns $P \subset PAT_\Sigma = (\mathcal{X} \cup \Sigma)^+$ and w be a word from Σ^*. The corresponding decision problem is called *(exact) matching problem* $\mathtt{Match}_P$.

Exact Matching Problem for P: $\mathtt{Match}_P$
Input: A pattern $\alpha \in P$, with $|\alpha| = m$, a word w, with $|w| = n$.
Question: Is there a substitution h with $h(\alpha) = w$?

Let $\sim$ be a relation on two strings, e.g., a string metric or a similarity measure. If a substitution h exists such that $h(\alpha) \sim w$, α is said to match the word w under $\sim$.

In the following, I will provide known results from the literature on the *(exact) pattern matching* problem for a word w and a pattern α from various classes. All of these complexities are in the Word RAM model, which will be introduced later in Section 2.8. The Theorem 2.7 is on the matching for 1Var and is considered folklore. The target word's length immediately implies the variable's length, and the substitution is obtained from the variables first occurrence. All that is left to check is whether this is a valid substitution for all its other occurrences.

Theorem 2.7. *The problem* $\mathtt{Match}_{\mathtt{1Var}}$ *is solvable in linear time.*

Also, Theorem 2.8 on Reg is considered folklore as it greedily uses any linear time pattern matching algorithm (e.g., *KMP*) to find the first occurrence of each terminal word $w_1, \ldots, w_M$ from the pattern $\alpha = w_0 \prod_{i=1}^{M}(x_i w_i)$ in their respective order from left to right. For an algorithm see [59].

Theorem 2.8. *The problem* $\mathtt{Match}_{\mathtt{Reg}}$ *is solvable in linear time.*

One can find the results of the theorems below in the literature [59, 143, 51].

Theorem 2.9. *[59] The problem* $\mathtt{Match}_{\mathtt{1RepVar}}$ *is solvable in quadratic time.*

Theorem 2.10. *[59] The problem* $\texttt{Match}_{\texttt{kRepVar}}$ *is solvable in* $O\left(\frac{n^{2k}}{((k-1)!)^2}\right)$ *time.*

Theorem 2.11. *[59] The problem* $\texttt{Match}_{\texttt{NonCross}}$ *is solvable in* $O(pn \log n)$ *time, where p is the number of one-variable blocks (i.e., number of distinct variables).*

Theorem 2.12. *[59] The problem* $\texttt{Match}_{\texttt{kSCD}}$ *is solvable in* $O\left(\frac{pn^{2k}}{((k-1)!)^2}\right)$ *time, where p is the number of repetitive-variable blocks.*

Theorem 2.13. *[143] The problem* $\texttt{Match}_{k-bounded}$ *is solvable in* $O\left(n^{2k+4}\right)$ *time.*

Theorem 2.14. *[51] The problem* $\texttt{Match}_{\texttt{1LOC}}$ *is solvable in* $O\left(mn^2 log n\right)$ *time, where m is the length of the pattern.*

Theorem 2.15. *[51] The problem* $\texttt{Match}_{\texttt{kLOC}}$ *is solvable in* $O\left(mkn^{2k+1}\right)$ *time, where m is the length of the pattern.*

2.4 Hamming Distance

Definition 2.16. *For two words* $w_1, w_2 \in \Sigma^*$ *with equal length (*$|w_1| = |w_2|$*), the Hamming distance* $d_{\texttt{HAM}}(w_1, w_2) = \Delta$ *(*$\Delta \in \mathbb{N}_0$*) is defined as the number of mismatches between the two words. Alternatively, the Hamming distance is the amount of substitution operations needed to obtain* w_2 *from* w_1.

$$d_{\texttt{HAM}}(w_1, w_2) := \left|\{i \mid w_1[i] \neq w_2[i]\}\right|, \textit{for } 1 \leq i \leq |w_1| = |w_2|$$

The Example 2.17 illustrates the Hamming distance of two words.

Example 2.17. $w_1 = \texttt{abbcabaa}$ *and* $w_2 = \texttt{acbcaaaa}$ *with* $d_{\texttt{HAM}}(w_1, w_2) = \Delta = 2$.

$$\begin{array}{cccccccccc} w_1 & = & \texttt{a} & \texttt{b} & \texttt{b} & \texttt{c} & \texttt{a} & \texttt{b} & \texttt{a} & \texttt{a} \\ & & & \downarrow & & & & \downarrow & & \\ w_2 & = & \texttt{a} & \texttt{c} & \texttt{b} & \texttt{c} & \texttt{a} & \texttt{a} & \texttt{a} & \texttt{a} \end{array}$$

One can extend the Hamming distance to a pattern α and a word w as a word-to-language distance, defined by the smallest distance to any word from the language represented by the pattern.

$$d_{\texttt{HAM}}(\alpha, w) := \Delta = \min\{d_{\texttt{HAM}}(h(\alpha), w) \mid h \text{ is arbitrary substitution of } \alpha\}$$

This definition leads directly to the definition of the minimization matching problem.

Approximate Matching Minimisation Problem for P: $\texttt{MinHDMatch}_P$

Input: A pattern $\alpha \in P$, with $|\alpha| = m$, a word w, with $|w| = n$.

Question: Compute h that minimizes $d_{\texttt{HAM}}(h(\alpha), w)$.

Further, Δ can be given in the input to describe a decision problem, which asks for the existence of a substitution h that fulfills $d_{\mathtt{HAM}}(h(\alpha), w) \leq \Delta$. This decision variant can generally be used as a subroutine in combination with binary search (choice of Δ) to solve the minimization problem.

Approximate Matching Decision Problem for P: $\mathtt{HDMatch}_P$

Input: A pattern $\alpha \in P$, with $|\alpha| = m$, a word w, with $|w| = n$, an integer $\Delta \leq m$.

Question: Is there a substitution h with $h\ d_{\mathtt{HAM}}(h(\alpha), w) \leq \Delta$?

It is important to note that the length of $h(\alpha)$ is fixed by $|w|$ due to the definition of Hamming distance.

2.5 Edit Distance

Definition 2.18. *For two words $w_1, w_2 \in \Sigma^*$ with $|w_1| = n$ and $|w_2| = m$ (assume $n \geq m$), the* edit distance $d_{\mathtt{ED}}(w_1, w_2)$ *[121, 120] is defined as the minimal number of* edit operations *(insertions, deletions, substitutions) applied to w_1 to obtain w_2. Let $w_1, w_2, v_1, v_2 \in \Sigma^*$ and let $a, b \in \Sigma$. The insertion of a letter a transforms $w_1 = v_1v_2$ into $w_2 = v_1av_2$, and the deletion of a letter a transforms $w_1 = v_1av_2$ into $w_2 = v_1v_2$. Further, a substitution of the letter a by the letter b transforms $w_1 = v_1av_2$ into $w_2 = v_1bv_2$.*

The Example 2.19 shows the edit operations applied to a word to obtain a target word.

Example 2.19. *Given: $w_1 = \mathtt{abbab}$. Three edit operations are required to obtain $w_2 = \mathtt{baaba}$, hence the edit distance is $\Delta = 3$.*

$$\begin{array}{ccccccccc|l}
w_1 & = & \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{b} & - & & \downarrow \textit{deletion} \\
 & & - & \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{b} & - & & \downarrow \textit{substitution} \\
 & & - & \mathtt{b} & \mathtt{a} & \mathtt{a} & \mathtt{b} & - & & \downarrow \textit{insertion} \\
w_2 & = & - & \mathtt{b} & \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{a} & &
\end{array}$$

Recall the following basic handling of the edit distance when transforming the word u into w, given in our paper [79]. Assume that u is transformed into w by a sequence of edit operations γ (i.e., w is aligned to w by γ). We can assume without losing generality that the edits in γ are ordered left to right with respect to the position of u where they are applied. Then, for each factorization $u = u_1 \dots u_k$ of u, there exists a factorization $w = w_1 \dots w_k$ of w such that w_i is obtained from u_i when applying the edits of γ which correspond to the positions of u_i, for $i \in \{1, \dots, k\}$. Note that this factorization of w is not unique: assume that the insertions applied at the beginning of u correspond to positions of u_1, the insertions applied at the end of u correspond to positions of u_k, but the insertions applied between u_{i-1} and u_i can be split arbitrarily into two parts: when considering them in the order in which they occur in γ (so left to right w.r.t. the positions of u where they are applied) one can assume first to have a (possibly empty) set of insertions which correspond to positions of u_{i-1} and then a (possibly empty) set of insertions which correspond to positions of u_i. On the other hand, if $w = w_1 w_2$, one can uniquely identify the shortest prefix u_1 (respectively, the longest prefix u'_1) of u from which, when applying the edits of γ one obtains the prefix w_1 of w.

The known algorithms for calculating the edit distance $d_{\mathtt{ED}}(w_1, w_2)$ of words w_1, w_2 are aligning prefixes $w_1[1 : j]$ and $w_2[1 : \ell]$ via dynamic programming based on the following recursion. To make the recursion more readable, only the length of the prefix j (respectively ℓ) is written instead of $w_1[1 : j]$ (respectively $w_2[1 : \ell]$). The base cases to this recursion are given by $d_{\mathtt{ED}}(\epsilon, \ell) = \ell$ and $d_{\mathtt{ED}}(j, \epsilon) = j$ due to either deleting or inserting every letter.

$$d_{\mathtt{ED}}(j,\ell) = min\begin{cases} d_{\mathtt{ED}}(j-1,\ell)+1, & w_1[j] \text{ is deleted;}\\ d_{\mathtt{ED}}(j,\ell-1)+1, & w_2[\ell] \text{ is inserted;}\\ d_{\mathtt{ED}}(j-1,\ell-1)+1, & w_1[j] \text{ is substituted by } w_2[\ell], \text{ if } w_1[j] \neq w_2[\ell];\\ d_{\mathtt{ED}}(j-1,\ell-1), & w_1[j] \text{ is left unchanged, if } w_1[j] = w_2[\ell]; \end{cases}$$

The standard dynamic programming algorithm that follows from the recursion by saving $d_{\mathtt{ED}}(j, \ell)$ in a cell $M[j][\ell]$ of a $(n \times m)$-matrix runs in $O(n^2)$ time. Two other algorithms prune the search space of the recursion to relevant values depending on the distance Δ. These algorithms run in $O(n\Delta)$ and $O(n + \Delta^2)$ and improve the standard algorithm as $\Delta \leq n$ (substituting m letters and deleting $n - m$ letters). In fact, the conditional fine-grained lower bound of Arturs Backurs and Piotr Indyk [13] implies that, under standard complexity theoretic assumptions, there does not exist an algorithm polynomially faster than the rectangular algorithm.

Theorem 2.20. *Edit distance can not be computed in time $O(n^h \Delta^g)$ where $h + g = 2 - \epsilon$ with $\epsilon > 0$, unless the Orthogonal Vectors Conjecture fails.*

The edit distance can be extended to a pattern α and a word w as a word-to-language distance, defined by the smallest distance to any word from the language represented by the pattern.

$$d_{\mathtt{ED}}(\alpha, w) := \Delta = \min\{d_{\mathtt{ED}}(h(\alpha), w) \mid h \text{ is arbitrary substitution of } \alpha\}$$

Further, the decision and minimization matching problem can be defined in the same way as for the Hamming distance in the previous Section 2.4.

Approximate Matching Minimisation Problem for $\mathtt{MinEDMatch}_P$

Input: A pattern $\alpha \in P$, with $|\alpha| = m$, a word w, with $|w| = n$.

Question: Compute h that minimizes $d_{\mathtt{ED}}(h(\alpha), w)$.

Approximate Matching Decision Problem for $\texttt{EDMatch}_P$

Input: A pattern $\alpha \in P$, with $|\alpha| = m$, a word w, with $|w| = n$, an integer $\Delta \leq m$.

Question: Is there a substitution h with $d_{\texttt{ED}}(\alpha, w) \leq \Delta$?

While in the case of Hamming distance, the length of $h(\alpha)$ is fixed by $|w|$ due to the nature of the problem, this is no longer the case under edit distance. It is, however, easy to put an upper bound on Δ by considering the substitution that replaces all variables with the empty word. This guarantees that $\Delta \leq \max\{|\texttt{term}(\alpha)|, |w|\}$.

2.6 Subsequences

This section contains the important concepts about subsequences from our papers [65] and [50].

A word u is a *subsequence* of a word w if there exists a strictly increasing integer sequence $0 < i_1 < i_2 < \ldots < i_{|u|} \leq |w|$ with $w[i_j] = u[j]$ for all $j \in [|u|]$. If u is a subsequence of w, this is denoted by $u \preceq w$ and, for a given $k \in \mathbb{N}_0$, $\mathbb{S}_k(w) = \{u \in \Sigma^{\leq k} \mid u \preceq w\}$ is the set of all subsequences of w with length at most k (see Example 2.21).

Example 2.21.

$$\mathbb{S}_2(abaca) = \{a, b, c, aa, ab, ac, ba, bc, ca\}$$

Two words $w_1, w_2 \in \Sigma^*$ are called *Simon k-congruent* ($w_1 \sim_k w_2$) if $\mathbb{S}_k(w_1) = \mathbb{S}_k(w_2)$ [157]. Further, if for two words $w_1 \sim_k w_2$, this also implies $w_1 \sim_\ell w_2$ for $0 \leq \ell \leq k$ (smaller subsequences are therefore omitted in most examples). In the Example 2.22 , the words $w_1 = abacab$ and $w_2 = baacabba$ are Simon 2-congruent, and the words w_1 and $w_3 = baacabbac$ are not Simon 2-congruent.

Example 2.22.

$$\begin{aligned}
\mathbb{S}_2(w_1) = \mathbb{S}_2(abacab) &= \{aa, ab, ac, ba, bb, bc, ca, cb\}\\
\mathbb{S}_2(w_2) = \mathbb{S}_2(baacabba) &= \{aa, ab, ac, ba, bb, bc, ca, cb\}\\
\mathbb{S}_2(w_3) = \mathbb{S}_2(baacabbac) &= \{aa, ab, ac, ba, bb, bc, ca, cb, cc\}\\
\mathbb{S}_2(w_1) = \mathbb{S}_2(w_2) &\Rightarrow w_1 \sim_2 w_2\\
\mathbb{S}_2(w_1) \neq \mathbb{S}_2(w_3) &\Rightarrow w_1 \nsim_2 w_3
\end{aligned}$$

A word $w \in \Sigma^*$ is called *k-subsequence universal* (or k-universal for short) for some $k \in \mathbb{N}$ if $\mathbb{S}_k(w) = \Sigma^{\leq k}$; this means that $w \sim_k (1 \cdots \sigma)^k$. The largest $k \in \mathbb{N}_0$, such that w is k-universal, is called the *universality index* of w, denoted by $\iota(w)$. If the universality index of w has value k ($\iota(w) = k$), then w is also ℓ-universal for all $\ell \leq k$. Notice that k-universality is always w.r.t. a given alphabet Σ as the word $\mathtt{abcba}$ is 1-universal (or just universal) for $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ but it is not universal for $\Sigma \cup \{\mathtt{d}\}$.

Unless explicitly stated otherwise, it is always assumed that the universality is defined w.r.t. the alphabet of the considered word w, i.e., $\texttt{alph}(w)$. The notion of k-universality coincides with the term k-richness in some parts of the literature (e.g., [95, 96]). See the subsequences of the words $w_1 = abacab$ and $w_2 = abacabc$ with $\texttt{alph}(w_1) = \texttt{alph}(w_2) = \{a, b, c\}$ in Example 2.23.

Example 2.23.

$$\mathbb{S}_2(w_1) = \mathbb{S}_2(abacab) = \{aa, ab, ac, ba, bb, bc, ca, cb\}$$
$$\mathbb{S}_2(w_2) = \mathbb{S}_2(abacabc) = \{aa, ab, ac, ba, bb, bc, ca, cb, cc\}$$

Because $\mathbb{S}_1(w_1) = \{a, b, c\}$ it is clearly 1-universal but since $cc \notin \mathbb{S}_2(w_1)$ it is not 2-universal, hence $\iota(w_1) = 1$. For $\mathbb{S}_2(w_2)$, one can easily verify that it is 2-universal but not 3-universal, therefore $\iota(w_2) = 2$. The following Definition 2.24 introduces a unique factorization of words derived from the research of Hébrard [87].

Definition 2.24. *For $w \in \Sigma^*$ the* arch factorisation *of w is $w = \mathrm{ar}_w(1) \cdots \mathrm{ar}_w(k) r(w)$ for some $k \in \mathbb{N}_0$ where $\mathrm{ar}_w(i)$ is universal, the last letter of $\mathrm{ar}_w(i)$, namely $\mathrm{ar}_w(i)[|\,\mathrm{ar}_w(i)|]$, does not occur in $\mathrm{ar}_w(i)[1 : |\,\mathrm{ar}_w(i)| - 1]$ for all $i \in [1 : k]$, and $alph(r(w)) \subset \Sigma$. The words $\mathrm{ar}_w(i)$ are called* arches *of w, $r(w)$ is called* rest.

The Example 2.25 shows the arch factorization of the word $w = abcaccccabcaba$.

Example 2.25.

$$\underbrace{abc}_{\mathrm{ar}_w(1)} | \underbrace{accccab}_{\mathrm{ar}_w(2)} | \underbrace{cab}_{\mathrm{ar}_w(3)} | \underbrace{a}_{r(w)}$$

The following immediate Theorem 2.26, based on the work of Simon [157], completely characterizes the set of k-subsequence universal words based on Hebrard's arch factorization.

Theorem 2.26. *The word $w \in \Sigma^*$ is k-universal if and only if there exist the words v_i, with $i \in [1 : k]$, such that $v_1 \cdots v_k = w$ and $alph(v_i) = \Sigma$ for all $i \in [1 : k]$.*

Using Theorem 2.26, the following Theorem 2.27 shows that the arches and, therefore, the universality index of a word can be computed in linear time w.r.t. the word length (the computational model is defined in Section 2.8).

Theorem 2.27. *Let w be a word, with $|w| = n$, $\texttt{alph}(w) = \Sigma$, and $\Sigma = \{1, 2, \ldots, \sigma\}$. The arch factorization, and therefore $\iota(w)$, can be computed in linear time $O(n)$.*

More precisely, the decomposition of w into arches $w = u_1 \cdots u_k$ can be computed by a greedy approach as follows:

- u_1 is the shortest prefix of w with $\texttt{alph}(u_1) = \Sigma$, or $u_1 = w$ if there is no such prefix;
- if $u_1 \cdots u_i = w[1 : t]$, for some $i \in [1 : k]$ and $t \in [1 : n]$, compute u_{i+1} as the shortest prefix of $w[t + 1 : n]$ with $\texttt{alph}(u_{i+1}) = \Sigma$, or $u_{i+1} = w[t + 1 : n]$ if there is no such prefix.

In order to introduce a slightly modified version of the arch factorization used in Chapter 5, the following notions are required. To access the first occurrence of a letter $\texttt{a} \in \Sigma$ after a position $i \in [|w|]$ in a word $w \in \Sigma^*$, define the *X-ranker* as a mapping [165]:

$$\texttt{X} : \Sigma^* \times ([|w|] \cup \{0, \infty\}) \times \Sigma \to [|w|] \cup \{\infty\}$$

with,

$$(w, i, \texttt{a}) \mapsto \min(\{j \in [i + 1 : |w|] \mid w[j] = \texttt{a}\} \cup \{\infty\}).$$

Notice that a lookup table for all possible X-ranker evaluations for some given $w \in \Sigma^*$ can be computed in linear time in $|w|$, where each item can be accessed in constant time [62, 17]. In the special case of $\texttt{X}(w, 0, \texttt{a})$, call this occurrence of $\texttt{a}$ the *signature letter* $\texttt{a}$ of w, for all $\texttt{a} \in \texttt{alph}(w)$. A *permutation* γ of an alphabet Σ is a word in Σ^σ with $\texttt{alph}(\gamma) = \Sigma$. Based on this new notation, Definition 2.28 introduces an equivalent variant of the arch factorization.

Definition 2.28. *The* arch factorisation *of a word $w \in \Sigma^*$ is defined as $w = \texttt{ar}_1(w) \cdots \texttt{ar}_k(w)\texttt{r}(w)$ for some $k \in \mathbb{N}_0$ such that there exists a sequence $(i_j)_{j \leq k}$ with $i_0 = 0$, $i_j = \max\{\texttt{X}(w, i_{j-1}, \texttt{a}) \mid \texttt{a} \in \Sigma\}$ for all $j \geq 1$, $\texttt{ar}_j(w) = w[i_{j-1} + 1 : i_j]$ whenever $1 \leq i_j < \infty$, and $\texttt{r}(w) = w[i_j : |w|]$, if $i_{j+1} = \infty$.[87]*

In Definition 2.29 the arches and rest as well as the universality index of $w \in \Sigma^*$ is specifically defined for individual $\texttt{a} \in \texttt{alph}(w)$ (cf. the arch jumping functions introduced in [152]). That is the arch factorization and universality index for the suffix of w that starts after the first occurrence of $\texttt{a}$ in w.

Definition 2.29. *Let $w \in \Sigma^*$, $\texttt{a} \in \texttt{alph}(w)$, and $j \in [\iota(w)]$. The arches of signature letters are defined by $\texttt{ar}_{\texttt{a},j}(w) = \texttt{ar}_j(w[\texttt{X}(w, 0, \texttt{a}) + 1 : |w|])$ and $\texttt{r}_{\texttt{a}}(w) = \texttt{r}(w[\texttt{X}(w, 0, \texttt{a}) + 1 : |w|])$. The* universality index of $\texttt{a}$ *is $\iota_{\texttt{a}}(w) = \iota(w[X(w, 0, \texttt{a}) + 1 : |w|])$. The last index with respect to w of $\texttt{ar}_{\texttt{a},j}(w)$ is defined as* $\texttt{arEnd}_{\texttt{a},j}(w) = \texttt{X}(w, 0, \texttt{a}) + \sum_{i=1}^{j} |\texttt{ar}_{\texttt{a},i}(w)|$.

The *marginal sequence* of a word w is introduced in Definition 2.1. This is a breadth-first ordering of σ parallel arch factorizations, each starting after a signature letter of the word. A *marginal sequence* is used to search for the smallest substrings of w that allow the completion of the rest of specific prefixes of w to full arches.

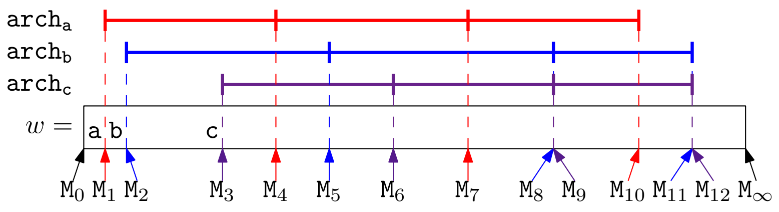


Figure 2.1: The marginal sequence of a word. (Figure by my co-author Sungmin Kim)

Definition 2.30. *Let $w \in \Sigma^*$ and γ be a permutation of Σ such that $\mathtt{X}(w, 0, \gamma[i])$ is increasing w.r.t. $i \in [\sigma]$. From the arches for signature letters, the* marginal sequence *of integers of $w \in \Sigma^*$ is defined inductively by $\mathtt{M}_0(w) = 0$, $\mathtt{M}_i(w) = \mathtt{X}(w, 0, \gamma[i])$ for all $i \in [\sigma]$, and $\mathtt{M}_{i\sigma+j}(w) = \mathtt{arEnd}_{\gamma[j],i}(w)$ for $j \in [\sigma]$, $i \in [\iota_{\gamma[j]}(w)]$. Let $\mathtt{M}_\infty(w) = |w|$ denote the last element of the sequence.*

The sequence is called *marginal* because, for $j \in [\sigma]$, $w[\mathtt{M}_{i\sigma+j-1}(w) + 1 : \mathtt{M}_{i\sigma+j}(w)]$ is the smallest prefix p of $w[\mathtt{M}_{i\sigma+j-1}(w) + 1 : |w|]$ such that $\iota_{\gamma[j]}(w[1 : \mathtt{M}_{i\sigma+j-1}(w)]p) = i$. Note that the marginal sequence $\mathtt{M}_i(w)$ is non-decreasing. Definition 2.31 is a slight variation of the *subsequence universality signature* $\mathtt{s}(w)$ introduced in [152].

Definition 2.31. *1. For $w \in \Sigma^*$, the* subsequence universality signature *$\mathtt{s}(w)$ of w is defined as the 3-tuple $(\gamma, \mathcal{K}, \mathcal{R})$ with a permutation γ of $\mathtt{alph}(w)$, where $\mathtt{X}(w, 0, \gamma[i]) > \mathtt{X}(w, 0, \gamma[j]) \Leftrightarrow i > j$ (γ consists of the letters of $\mathtt{alph}(w)$ in order of their first appearance in w) and two arrays $\mathcal{K}$ and $\mathcal{R}$ of length σ with $\mathcal{K}[i] = \iota_{\gamma[i]}(w)$ and $\mathcal{R}[i] = \mathtt{alph}(\mathtt{r}_{\gamma[i]}(w))$ for all $i \in [|\mathtt{alph}(w)|]$. For all $i \in [\sigma] \setminus \mathtt{alph}(w)$, let $\mathcal{R}[i] = \Sigma$ and $\mathcal{K}[i] = -\infty$.*
2. Conversely, for a permutation γ' of Σ, an integer array $\mathcal{K}'$ and an alphabet array $\mathcal{R}'$ both of length σ, the tuple $(\gamma', \mathcal{K}', \mathcal{R}')$ is a valid *signature if there exists a word w that satisfies $\mathtt{s}(w) = (\gamma,' \mathcal{K}', \mathcal{R}')$.*

Note that, for $k_i = \iota_{\gamma[i]}(w)$, it holds that $\mathcal{R}[i] = \mathtt{alph}(w[\mathtt{M}_{k_i\sigma+i}(w) + 1 : \mathtt{M}_\infty(w)])$, since $\mathtt{r}_{\gamma[i]}(w) = w[\mathtt{M}_{k_i\sigma+i}(w) + 1 : \mathtt{M}_\infty(w)]$.

2.7 General data structures

This section will provide an overviewof the general data structures used in this thesis.

The first data structure reports the *Longest Common Prefix* (LCP) of two suffixes of a word. This data structure is essential for many stringology algorithms and is important in this thesis.

Definition 2.32. *Given a word w, of length n, the LCP-data structure for w can be constructed in $O(n)$-time. This data structure returns for two indices i, j in $O(1)$-time the length of the longest common prefix of the suffixes $w[i : n]$ and $w[j : n]$.*

$$\mathsf{LCP}_w(i, j) = max\{|v| \mid v \text{ is a prefix of both } w[i : n] \text{ and } w[j : n]\}$$

For more details see [100, 101] and the references therein. Now, given two words w_1, w_2, of length n and m, one can construct in $O(n + m)$-time a data structure which returns in $O(1)$-time the value $\mathtt{LCP}(w_1[i : n], w_2[j : m])$, the length of the longest common prefix of the strings $w_1[i : n]$ and $w_2[j : m]$ for all j and i. In other words, $\mathtt{LCP}(w_1[i : n], w_2[j : m]) = max\{|v| \mid v$ is a prefix of both $w_1[i : n]$ and $w_2[j : m]\}$. This is achieved by constructing $LCP_{w_1w_2}$-data structures for the word w_1w_2, as above, and a slightly adjusted definition.

$$\mathtt{LCP}(w_1[i : n], w_2[j : m]) = \min(\mathtt{LCP}_{w_1w_2}(i, n + j), n - i)$$

If one wants to consider suffixes instead of prefixes, the *Longest Common Suffix* (LCS) data structure can be build analogously [100, 101]. The Example 2.33 below shows a *LCP* query on the word $w = banana$.

Example 2.33.

$$\begin{aligned} w[1 : n] &= banana \\ w[2 : n] &= ana|na \\ w[4 : n] &= ana \\ LCP_w(2, 4) &= 3 \end{aligned}$$

Secondly, the *Interval Union-Find data structure* [72, 90] is introduced in Definition 2.34.

Definition 2.34. *Let $V = [1 : n]$ and S a set with $S \subseteq V$. The elements of $S = \{s_1, \ldots, s_p\}$ are called borders and are ordered $0 = s_0 < s_1 < \ldots < s_p < s_{p+1} = n + 1$ where s_0 and s_{p+1} are generic borders. For each border s_i, $V(s_i) = [s_{i-1} + 1 : s_i]$ is defined as an induced interval. Now, $P(S) := \{V(s_i) \mid s_i \in S\}$ gives an ordered partition of the set V. The* interval Union-Find structure *maintains the partition $P(S)$ under the operations:*

- *For $u \in V$,* `find`(u) *returns $s_i \in S \cup \{n + 1\}$ such that $u \in V(s_i)$. In other words, all elements in the interval $V(s_i)$ are represented by or have the representative s_i.*
- *For $u \in S$,* `union`(u) *updates the partition $P(S)$ to $P(S \setminus \{u\})$. That is, if $u = s_i$, then replace the intervals $V(s_i)$ and $V(s_{i+1})$ by the single interval $[s_{i-1} + 1 : s_{i+1}]$ and update the partition so that further* `find` *and* `union` *operations can be performed.*

The usage of the interval Union-Find data structure will be as follows: describing the intervals stored initially in the data structure, and then the unions are made between adjacent intervals. Further, the `find` operation is adjusted so that it returns both borders of the interval containing the searched value and additional satellite data associated with that interval. Lemma 2.35 was shown in [72, 90].

Lemma 2.35. *Given an* Interval Union-Find data structure *defined as above, the initialisation of the structure followed by a sequence of $m \in O(n)$ union and find operations can be executed in overall $O(n)$ time and space.*

Finally, in Definition 2.36 the *Range Minimum Query* data structure (RMQ) [21], constructed for an array of elements, efficiently reports the minimal element in subarrays.

Definition 2.36. *Let A be an array with n elements from a well-ordered set. The* range minimum query *data structure* RMQ_A *is defined for the array A:* $\mathrm{RMQ}_A(i, j) = \arg\min\{A[t] \mid t \in [i : j]\}$*, for $i, j \in [1 : n]$. That is,* $\mathrm{RMQ}_A(i, j)$ *is the position of the smallest element in the subarray $A[i : j]$; if there are multiple positions containing this smallest element,* $\mathrm{RMQ}_A(i, j)$ *is the leftmost of them. (When it is clear from the context, the subscript A is left out).*

Lemma 2.37 provides the main result from [21].

Lemma 2.37. *Let A be an array with n integer elements. One can preprocess A in $O(n)$ time and produce data structures allowing to answer in constant time* range minimum queries $\mathrm{RMQ}_A(i, j)$*, for any $i, j \in [1 : n]$.*

2.8 Computational Model

The *computational model* that was used in our papers [78, 79, 65, 50] and, therefore, in the chapters of this thesis is the standard Word RAM model with memory words of logarithmic size. This is a standard computational model in algorithm design in which, for an input of size n, the memory consists of memory words consisting of $\Theta(\log n)$ bits. Basic operations (including arithmetic and bitwise Boolean operations) on memory words take constant time, and any memory word can be accessed in constant time. Numbers larger than n, with ℓ bits, are represented in $\Theta(\ell / \log n)$ memory words, and working with them takes time proportional to the number of memory words on which they are represented. In most of the problems, it is assumed that a word w and a pattern α, with $|w| = n$ and $|\alpha| = m$, are given over a terminal-alphabet $\Sigma = \{1, 2, \ldots, \sigma\}$, with $|\Sigma| = \sigma \leq n + m$ (in Chapter 5 a constant sized alphabet is assumed). The variables are chosen from the set $\{x_1, \ldots, x_m\}$ and can be encoded as integers between $n + 1$ and $n + m$. These assumptions yield that all the processed words are sequences of integers (called letters or symbols), each fitting in $O(1)$ memory words. This is a common assumption in string algorithms: the input is said to be over *an integer alphabet*. For a more detailed general discussion on this computational model, cf. [48].

CHAPTER 3

Matching Patterns with Variables under Hamming Distance

and the LaTeX code of the article [74] was used to reproduce it here.

This chapter is based on article [78] (see reference below) and the LaTeX code of this article was used to reproduce it here. The Introduction and Preliminaries of this paper are worked into Chapter 1 and Chapter 2. Further, the notations are adjusted to match the other chapters and provide a uniform notation across this thesis.

Reference: P. Gawrychowski, F. Manea, and S. Siemer. Matching Patterns with Variables Under Hamming Distance. In *46th International Symposium on Mathematical Foundations of Computer Science, MFCS 2021*, volume 202 of *LIPIcs*, pages 48:1–48:24, 2021. `doi:10.4230/LIPIcs.MFCS.2021.48`

Description: We provided our first generalized setting for the problem of matching patterns with variables by allowing mismatches between the target word and the pattern (under any substitution). The number of mismatches, known as the Hamming distance, must be either bound by an integer given as input (decision variant) or minimal to be a valid substitution (minimization variant). We obtained algorithmic lower and upper bounds for various classes of patterns.

Contribution: I was a main contributor to identifying the problem described in this paper, the design of the algorithms, and their write-up in the paper. Further, I presented this work at *MFCS 2021* in Tallinn (Estonia) and several workshops.

3.1 Overview

In this chapter, we extend the study of patterns which can be matched efficiently to the case of approximate matching: we allow mismatches between the word w and the image of α under a substitution h. More precisely, we consider two problems. In the decision problem $\mathtt{HDMatch}_P$ we are interested in deciding, for a given pattern α from a class P, a given word w, and a non-negative integer Δ whether there exists a variable-substitution h such that the word $h(\alpha)$ has at most Δ mismatches to the word w; in other words, $d_{\mathtt{HAM}}(h(\alpha), w) \leq \Delta$. Alternatively, we consider the corresponding minimisation problem $\mathtt{MinHDMatch}_P$ of computing $d_{\mathtt{HAM}}(\alpha, w) = \min\{d_{\mathtt{HAM}}(h(\alpha), w) \mid h \text{ is a substitution of the variables in } \alpha\}$.

With the definition of the Hamming Distance (Section 2.4) and the pattern matching problems for families of patterns $P \subseteq PAT$ (Section 2.2) we can give the formal problem description. In the first problem, we allow for a certain distance Δ between the image $h(\alpha)$ of α under a substitution h and the target word w instead of searching for an exact matching.

Approximate Matching Decision Problem for P: $\mathtt{HDMatch}_P$
Input: A pattern $\alpha \in P$, with $|\alpha| = m$, a word w, with $|w| = n$, an integer $\Delta \leq m$.
Question: Is there h with $d_{\mathtt{HAM}}(h(\alpha), w) \leq \Delta$?

In the second problem, we are interested in finding the substitution h such that the number of mismatches between $h(\alpha)$ and the target word w is minimal, over all possible choices of h.

Approximate Matching Minimisation Problem for P: $\mathtt{MinHDMatch}_P$
Input: A pattern $\alpha \in P$, with $|\alpha| = m$, a word w, with $|w| = n$.
Question: Compute h that minimizes $d_{\mathtt{HAM}}(h(\alpha), w)$.

Example 3.1. $\alpha = \mathtt{x}_1\mathtt{x}_1\mathtt{bab}\mathtt{x}_2\mathtt{x}_2$, $w = \mathtt{baaababbb}$, $\Delta = 1$

$$h(\mathtt{x}_1) = \mathtt{aa} \qquad h(\alpha) = \boldsymbol{a}\mathtt{aaababbb}$$
$$h(\mathtt{x}_2) = \mathtt{b} \qquad w = \boldsymbol{b}\mathtt{aaababbb}$$

When analysing the number of mismatches between $h(\alpha)$ and w we need to argue about the number of mismatches between corresponding factors of $h(\alpha)$ and w, i.e., the factors occurring between the same positions i and j in both words. To simplify the presentations, for a substitution h that maps a pattern α to a word of the same length as w, we will call the factors $h(\alpha)[i : j]$ and $w[i : j]$ aligned under h. We omit h when it is clear from the context. Moreover, saying that we align a factor $\alpha[i : j]$ to a factor $w[i' : j']$ with a minimal number of mismatches, we mean that we are looking for a substitution h such that $|h(\alpha)| = |w|$, $h(\alpha[i : j])$ is aligned to $w[i' : j']$ under h, and the resulting number of mismatches between $h(\alpha[i : j])$ and $w[i' : j']$ is minimal w.r.t. all other choices for the substitution h.

We make some preliminary remarks. Firstly, in all the problems we consider here, we can assume that the pattern α starts and ends with variables, i.e., $\alpha = x\alpha' y$, with α' pattern and x and y variables. Indeed, if this would not be the case, we could simply reduce the problems by considering them for inputs α' and the word w' obtained by removing from w the prefix and suffix aligned, respectively, to the maximal prefix of α which contains only terminals and the maximal suffix of α which contains only terminals. Clearly, in the case of the exact-matching problem the respective prefixes (suffixes) of w and α must match exactly, while in the case of the approximate-matching problems one needs to account for the mismatches created by these prefixes and suffixes. So, from now on, we will work under the assumption that the patterns we try to align to words start and end with variables.

Secondly, solving $\texttt{Match}_P$ is equivalent to solving $\texttt{HDMatch}_P$ for $\Delta = 0$. Also, in a general framework, $\texttt{MinHDMatch}_P$ can be solved by combining the solution of the decision problem $\texttt{HDMatch}_P$ with a binary search on the value of Δ. Given that the distance between α and w is at most $n = |w|$, one needs to use the solution for $\texttt{HDMatch}_P$ a maximum of $log\ n$ times in order to find the exact distance between α and w. Sometimes this can be done even more efficiently, as shown in Theorem 3.5. On the other hand, solving $\texttt{MinHDMatch}_P$ leads directly to a solution for $\texttt{HDMatch}_P$. For that reason, some results (especially related to hardness) for the minimisation problem follow trivially by solving the decision variant and are not explicitly mentioned.

Our Contribution. Our results are also summarized in Table 7.1 in Section 7.1. We obtained results for the problems $\texttt{HDMatch}_P$ and $\texttt{MinHDMatch}_P$ (introduced above) for a series of classes P of patterns for which the matching problem `Match` can be solved in polynomial time (see Section 2.2 and 2.3). Interestingly, for `Reg` we obtain matching upper and conditional lower bounds. As regular patterns are, in fact, a particular case of regular expressions, it is worth mentioning that, due to the rectangular conditional lower bounds from [12] on exact regular expression matching, it is not to be expected that the general case of matching regular-expressions under Hamming distance can be solved as efficiently as the case of regular patterns. Regarding patterns with repeated variables, we note that while in the case when the number of repeated variables, the scope coincidence degree, or the treewidth was bounded by a constant, polynomial-time algorithms for the exact matching problem were obtained. This does not hold in our approximate setting, unless P=NP. Only the locality measure has the same behaviour as in the case of exact matching: $\texttt{HDMatch}_{\texttt{kLOC}}$ and $\texttt{MinHDMatch}_{\texttt{kLOC}}$ can still be solved in polynomial time for constant k. In the simpler case of `1RepVar`-patterns, the locality corresponds to the number of x-blocks, so, if this is bounded by a constant, the two problems we consider can be solved in polynomial time. Our upper and lower bounds for patterns with repeated variables are based on a connection to the Median String and Consensus Patterns problems [122, 57, 29, 37].

3.2 Matching Regular Patterns with Mismatches

In this section we consider $\texttt{HDMatch}_{\texttt{Reg}}$ and $\texttt{MinHDMatch}_{\texttt{Reg}}$.

As a reminder, a pattern α is *regular* if $\alpha = w_0 \prod_{i=1}^{M}(x_i w_i)$, with $w_i \in \Sigma^*$. The class of regular patterns is denoted by $\texttt{Reg}$ (for more details see Section 2.2). For example, the pattern $\alpha_0 = \texttt{ab}x\texttt{ab}yz\texttt{baab}$, with $\texttt{var}(\alpha) = \{x, y, z\}$ is in $\texttt{Reg}$.

As mentioned already, a solution for $\texttt{HDMatch}_{\texttt{Reg}}$ with distance $\Delta = 0$ is a solution to $\texttt{Match}_{\texttt{Reg}}$. The latter problem can be solved in $O(n)$ by a greedy approach, as shown in Section 2.3. As noted in the previous Section 3.1, we can assume that $w_0 = w_M = \varepsilon$, so $\alpha = (\prod_{i=1}^{M-1} x_i w_i)x_M$. Thus, we identify the last occurrence $w[\ell + 1 : \ell + |w_{M-1}|]$ of w_{M-1} in w, assign the string $w[\ell + |w_{M-1}| + 1 : n]$ to x_M, and then recursively match the pattern $\alpha = (\prod_{i=1}^{M-2} x_i w_i)x_{M-1}$ to $w[1 : \ell]$.

In the following, we propose a solution for $\texttt{MinHDMatch}_{\texttt{Reg}}$ which generalizes this approach. Further, we will show a matching lower bound for any algorithm solving $\texttt{MinHDMatch}_{\texttt{Reg}}$.

3.2.1 Efficient solutions for $\texttt{HDMatch}_{\texttt{Reg}}$ and $\texttt{MinHDMatch}_{\texttt{Reg}}$

An equivalent formulation of $\texttt{MinHDMatch}_{\texttt{Reg}}$ is to find factors $w[\ell_i+1 : \ell_i+|w_i|]$, with $1 \le i \le M-1$, such that $\sum_{i=1}^{M-1} d_{\texttt{HAM}}(w_i, w[\ell_i+1 : \ell_i+|w_i|])$ is minimal and $\ell_i+|w_i|+1 \le \ell_{i+1}$, for all $i \in \{1, \dots, M-2\}$. In other words, we want to find the $M - 1$ factors $w[\ell_i + 1 : \ell_i + |w_i|]$, with i from 1 to $M - 1$, such that these factors occur one after the other without overlapping in w, they correspond (in order, from left to right) to the words w_i, for i from 1 to $M - 1$, and the total sum of mismatches between $w[\ell_i + 1 : \ell_i + |w_i|]$ and w_i, added up for i from 1 to $M - 1$, is minimal.

To approach this problem we need the following data-structures-preliminaries.

Given a word w, of length n, we can construct in $O(n)$-time *longest common suffix*-data structures which allow us to return in $O(1)$-time the value $LCS_w(i, j) = max\{|v| \mid v$ is a suffix of both $w[1 : i]$ and $w[1 : j]\}$. See the *LCP* data structure 2.7 or [100, 101] and the references therein. Given a word w, of length n, and a word u, of length m, we can construct in $O(n + m)$-time data structures which allow us to return in $O(1)$-time the value $LCS_{w,u}(i, j) = max\{|v| \mid v$ is a suffix of both $w[1 : i]$ and $u[1 : j]\}$. This is achieved by constructing LCS_w-data structures for wu, as above, and noting that $LCS_{w,u}(i, j) = \min(LCS_w(i, n + j), j)$.

The following two lemmas are based on the data structures defined above and the technique called kangaroo-jump [114].

Lemma 3.2. *Let w and u, with $|w| = |u| = n$, be two words and δ a non-negative integer. Assume that, in a preprocessing phase, we have constructed $LCS_{w,u}$-data structures. We can compute $\min(\delta + 1, d_{\texttt{HAM}}(u, w))$ using $\delta + 1$ $LCS_{w,u}$ queries, so in $O(\delta)$ time.*

Proof. Let $a = b = m$ and $d = 0$. While $a > 0$ and $d \leq \delta$ execute the following steps. Compute $h = LCS_{w,u}(a, b)$. If $h < b$, then increment d by 1, set $a \leftarrow a - h - 1$ and $b \leftarrow b - h - 1$, and start another iteration of the while-loop. If $h = b$, then set $b \leftarrow 0$ and exit the while-loop.

It is not hard to note that before each iteration of the while loop it holds that $d = d_{\mathtt{HAM}}(w[a+1 : m], u[b+1 : m])$. When the while loop is finished, $d = \min(d_{\mathtt{HAM}}(w[i-m+1 : i], u[1 : m]), \delta + 1)$. In each iteration we first identify the length h of the longest common suffix of $w[1 : a]$ and $u[1 : b]$. Then, we jump over this suffix, as it causes no mismatches, and have either traversed completely the words w and u (and we do not need to do anything more), or we have reached a mismatch between w and u, on position $a - h = b - h$. In the latter case, we count this mismatch, jump over it, and repeat the process (but only if the number of mismatches is still at most δ). So, in other words, we go through the mismatches of w and u, from right to left, and jump from one to the next one using $LCS_{w,u}$ queries. If we have more than δ mismatches, we do not count all of them, but stop as soon as we have met the $(\delta + 1)^{th}$ mismatch. Accordingly, the algorithm is correct. Clearly, we only need $\delta + 1$ $LCS_{w,u}$-queries and the time complexity of this algorithm is $O(\delta)$, once the $LCS_{w,u}$-data structures are constructed. □

Lemma 3.3. *Given a word w, with $|w| = n$, a word u, with $|u| = m < n$, and a non-negative integer δ, we can compute in $O(n\delta)$ time the array $D[m : n]$ with $n - m + 1$ elements, where $D[i] = \min(\delta + 1, d_{\mathtt{HAM}}(w[i-m+1 : i], u))$.*

Proof. We first construct, in linear time, the $LCS_{w,u}$-data structures for the input words. Note that the $LCS_{w,u}$-data structure can be directly used as $LCS_{w[i:i+m-1],u}$ data structure, for all $i \leq n - m + 1$.

Then, for each position i of w, with $i \leq m$, we use Lemma 3.2 to compute, in $O(\delta)$ time the value $d = \min(d_{\mathtt{HAM}}(u, w[i-m+1 : i]), \delta + 1)$. We then set $D[i] \leftarrow d$. By the correctness of Lemma 3.2, we get the correctness of this algorithm. Clearly, its time complexity is $O(n\delta)$. □

The following result is the main technical tool of this section.

Theorem 3.4. $\mathtt{HDMatch}_{\mathtt{Reg}}$ *can be solved in $O(n\Delta)$ time. For an accepted instance w, α, Δ of* $\mathtt{HDMatch}_{\mathtt{Reg}}$ *we also compute $d_{\mathtt{HAM}}(\alpha, w)$ (which is upper bounded by Δ).*

Proof. Assume $\alpha = \prod_{i=1}^{M-1}(x_i w_i) x_M$ and let $\alpha_\ell = \prod_{i=\ell}^{M-1}(x_i w_i) x_M$, for $\ell \in \{1, \ldots, M-1\}$.

A first observation is that the problem can be solved in a standard way by dynamic programming in $O(nm)$ time.

We only give the main idea behind this approach. We can compute the minimum number of mismatches $T[i][j]$ which can be obtained when aligning the suffix of length i of w to the suffix of length j of α, for all $i \leq n$ and $j \leq m$. Clearly, $T[i][j]$ can be computed based on the values $T[i+1][j+1]$ and, if $\alpha[j]$ is a variable, $T[i+1][j]$. The full technicalities of this standard approach are easy to obtain so we do not go into further details.

We present a more efficient approach below.

Our efficient algorithm starts with a preprocessing phase, in which we compute $LCS_{w,u}$-data structures, where $u = \prod_{i=\ell}^{M-1} w_i$. This allows us to retrieve in constant time answers to LCS_{w,w_i}-queries, for $1 \leq i \leq M-1$.

In the main phase of our algorithm, we compute an $(M-1) \times \Delta$ matrix $Suf[\cdot][\cdot]$, where, for $\ell \leq M-1$ and $d \leq \Delta$, we have $Suf[\ell][d] = g$ if and only if $w[g..n]$ is the shortest suffix of w with $d_{\mathtt{HAM}}(\alpha_\ell, w[g:n]) \leq d$.

Once more, we note that the elements of $Suf[\cdot][\cdot]$ can be computed by a relatively straightforward dynamic programming approach in $O(nM\Delta)$ time. But, the strategy we present here is more efficient than that.

In our algorithm, we first use Lemma 3.3 to compute $Suf[M-1][\cdot]$ in $O(n\Delta)$ time. We simply run the algorithm of that lemma on the input strings w and w_{M-1} and the integer Δ. We obtain an array $D[\cdot]$, where $D[i] = \min(\Delta+1, d_{\mathtt{HAM}}(w[i-|w_{M-1}|+1 : i], w_{M-1}))$. We now go with j from $|w_{M-1}|$ to n and, if $D[j] \leq \Delta$, we set $Suf[M-1][D[j]] = j - |w_{M-1}| + 1$. It is clear that $h = Suf[M-1][d]$ will be the starting position of the shortest suffix $w[h:n]$ of w such that $d_{\mathtt{HAM}}(w_{M-1}x_M, w[h:n]) \leq d$. Thus, $Suf[M-1][\cdot]$ was correctly computed, and the time needed to do so is $O(n\Delta)$.

Further, we describe how to compute $Suf[\ell][\cdot]$ efficiently, based on $Suf[\ell+1][\cdot]$ (for ℓ from $M-2$ down to 1). We use the following approach. We go through the positions i of w from right to left and maintain a queue Q. When i is considered, Q stores all elements d such that $Suf[\ell][d]$ was not computed yet until reaching that position, but $i < Suf[\ell+1][d]$. Accordingly, the fact that d is in Q means that with a suitable alignment of w_ℓ ending on position i, we could actually find an alignment with $\leq d$ mismatches of α_ℓ with $w[i-|w_\ell|+1 : n]$: when Q contains $d, \ldots, d-t$, for some $t \geq 0$, an alignment of w_ℓ to $w[i-|w_\ell|+1 : i]$ with $\leq t$ mismatches would lead to an alignment of α_ℓ with $w[i-|w_\ell|+1 : n]$ with $\leq d$ mismatches by extending the alignment of $\alpha_{\ell+1}$ to $w[Suf[\ell+1][d-t] : n]$. The values d present in Q at some point are ordered increasingly (the older values are larger), the array $Suf[\ell+1][\cdot]$ is also monotonically increasing, and, as $Suf[\ell][d]$ cannot be set before $Suf[\ell][d']$, for any d and d' such that $d' < d$, the queue Q is actually an interval of integers $[new : old]$, where new is the newest element of Q, and old the oldest one. When we consider position i of the word, if the alignment of w_ℓ ending on position i causes t mismatches, then to be able to set a value $Suf[\ell][d]$, with $d \in Q$, we need to have that $Suf[\ell+1][d-t] > i$. As $Suf[\ell+1][d] > Suf[\ell+1][d-t]$ and $d \in Q$, this means that $d-t \in Q$, so the number of mismatches t must be strictly upper bounded by

$|Q|$, in order to be useful. Accordingly, when considering position i, we compute the number $t \leftarrow \min\{d_{\mathtt{HAM}}(w_\ell, w[i-|w_\ell|+1 : i]), |Q|\}$, and if $t < |Q|$ we set $Suf[\ell][d] \leftarrow i-|w_\ell|+1$ for all d such that $d-t \in Q$; we also eliminate all these elements d from the queue. Before considering a new position i, we check if $i = Suf[\ell+1][new-1]$, and, if yes, we insert $new-1$ in Q and update $new \leftarrow new-1$.

This computation of $Suf[\ell][\cdot]$ is implemented in the following algorithm:

1. Initialization: We maintain a queue Q, which initially contains only the Δ.
   Let $new \leftarrow \Delta$ (this is the top element of the queue).

2. Iteration: $i \leftarrow Suf[\ell+1][\Delta]-1$ while $i \geq |w_\ell|$ we execute the steps a, b, and c:

   (a) Using Lemma 3.2 we compute $t \leftarrow \min(d_{\mathtt{HAM}}(w_l, w[i-|w_\ell|+1 : i]), |Q|)$.

   (b) If $t < |Q|$, we remove from Q all elements d, such that $d-t \geq new$, and set, for each of them, $Suf[\ell][d] \leftarrow i-|w_\ell|+1$. Keep a pointer on the smallest d denoted d_{min}. If Q is empty continue with (c), otherwise continue with (d).

   (c) If $Suf[\ell+1][d_{min}] \neq 0$ then insert d_{min} in Q and set $i \leftarrow Suf[\ell+1][d_{min}]$. Else set $i \leftarrow 0$ and exit the loop. Go to (e).

   (d) If $Suf[\ell+1][top-1] = i$ then we insert $top-1$ in Q and $top \leftarrow top-1$. Else, if $Suf[\ell+1][top-1] = 0$ then set $i \leftarrow 0$ and exit the loop.

   (e) Decrement $i--$

3. Filling-in the remaining positions: Set all the positions of $Suf[\ell][\cdot]$ which were not filled during the above while-loop to 0.

The matrix $Suf[\cdot][\cdot]$ is computed correctly by the above algorithm, as it can be shown by the following inductive argument.

To show that $Suf[\ell][\cdot]$ is computed correctly by our algorithm, under the assumption that $Suf[\ell+1][\cdot]$ was correctly computed, we make several observations.

Firstly, it is clear that $Suf[\ell+1][d] \leq Suf[\ell+1][d+1]$. Secondly, when computed correctly, $Suf[\ell][d]$ should be the rightmost position g of w such that $d_{\mathtt{HAM}}(w[g : n], w_\ell) = t \leq d$ and $Suf[\ell+1][d-t] \geq g+|w_\ell|$. Clearly, if $Suf[\ell][d+1] \neq 0$, then $Suf[\ell][d] < Suf[\ell][d+1]$.

Regarding the algorithm described in the main part of the Chapter, it is important to observe that the queue Q is ordered increasingly (i.e., the newer is an element in Q, the smaller it is) and the elements of Q form an interval $[new : old]$.

Now, let us show the correctness of the algorithm.

Let d be a non-negative integer, $d \leq \Delta$. Assume that our algorithm sets $Suf[\ell][d] = g$, with $g > 0$.

This means that d was removed from the queue in step 2.b when the for-loop was executed for $i = g + |w_\ell| - 1$. The reason for this removal was that $d_{\texttt{HAM}}(w[g : g + |w_\ell| - 1], w_\ell) = t \leq |Q| - 1$. Hence, in this step we have removed exactly those elements δ such that $new \leq \delta - t$. Accordingly, we also have that $new \leq d - t$ holds. Let $g' = Suf[\ell + 1][new]$. We thus have $g' > i = g + |w_\ell| - 1$, $d_{\texttt{HAM}}(\alpha_{\ell+1}, w[g' : n]) \leq new$, and $d_{\texttt{HAM}}(w_\ell x_\ell, w[g : g' - 1]) = t$. Putting this all together, we get that $d_{\texttt{HAM}}(\alpha_\ell, w[g : n]) \leq new + t \leq d$.

Now, assume for the sake of a contradiction, that there exists $g'' > g$ such that $d_{\texttt{HAM}}(\alpha_\ell, w[g'' : n]) \leq d$, i.e., $w[g : n]$ is not the shortest suffix s of w such that $d_{\texttt{HAM}}(\alpha_\ell, s) \leq d$. In this case, there exists d'' such that $g'' + |w_\ell| - 1 < Suf[\ell + 1][d'']$ and $d'' + d_{\texttt{HAM}}(w[g'' : g'' + |w_\ell| - 1], w_\ell) \leq d$. Because d is in Q when $i = g + |w_\ell| - 1$ is reached in the for-loop, then d must also be in Q when $i'' = g'' + |w_\ell| - 1$ is reached in the for-loop, because $i < i'' < Suf[\ell + 1][d''] \leq Suf[\ell + 1][d]$. In fact, as $Suf[\ell+1][d] \geq Suf[\ell+1][d''] > i''$, it follows that d'' must also be in Q when i'' is reached. Thus, $q \geq d - d''$ and, as we have seen above, $d - d'' \geq d_{\texttt{HAM}}(w[g'' : g'' + |w_\ell| - 1], w_\ell)$. Moreover, if new'' is the element on the top of the queue when i'' is reached, we have that $new'' \leq d''$. Hence, $new'' + d_{\texttt{HAM}}(w[g'' : g'' + |w_\ell| - 1], w_\ell) \leq d'' + d_{\texttt{HAM}}(w[g'' : g'' + |w_\ell| - 1], w_\ell) \leq d$. Therefore, when i'' was reached, all the conditions needed to remove d from Q and set $Suf[\ell][d] \leftarrow g''$ were met. We have reached a contradiction with our assumption that $g'' > g$.

In conclusion, if our algorithm sets $Suf[\ell][d] = g$, with $g > 0$, then $w[g : n]$ is the shortest suffix of w such that $d_{\texttt{HAM}}(w[g : n], w_\ell) \leq d$. By an analogous argument as the one used above in our proof by contradiction, we can show that if our algorithm sets $Suf[\ell][d] = 0$ then there does not exist any suffix $w[g : n]$ of w such that $d_{\texttt{HAM}}(w[g : n], w_\ell) \leq d$.

This means that our algorithm computing $Suf[\cdot][\cdot]$ is correct.

To finalize the proof of the theorem, we note that, after computing the entire matrix $Suf[\cdot][\cdot]$, we can accept the instance w, α, Δ of $\texttt{HDMatch}_{\texttt{Reg}}$ if and only if there exists $d \leq \Delta$ such that $Suf[1][d] \neq 0$. Moreover, $d_{\texttt{HAM}}(\alpha, w) = \min(\{d \mid Suf[1][d] \neq 0\} \cup \{+\infty\})$.

In the following we show that this algorithm works in $O(n\Delta)$ time. We will compute the complexity of this algorithm using amortized analysis. Firstly, we observe that the complexity of the algorithm is proportional to the total number of LCS_{w,w_ℓ}-queries we compute in step 2.a, for each $\ell \leq M$ or, in other words, over all executions of the algorithm. Now, we observe that when position i of w is considered (for a certain ℓ), we do $|Q|$ many LCS_{w,w_ℓ}-queries. So, this means that we do one query per each current element of Q (and none if $|Q| = 0$). Thus, the number of queries corresponding to each pair (ℓ, d) which appears in Q at some point equals the number of positions considered between the step when it was inserted in Q and the step when it was removed from Q. This means $O(Suf[\ell + 1][d] - Suf[\ell][d])$ queries corresponding to (ℓ, d). Summing this up for a fixed d and ℓ from 1 to $M - 2$ we obtain that the overall number of queries corresponding to a fixed δ is $O(Suf[M-1][d]) = O(n)$. Adding this up for all $d \leq \Delta$, we obtain that the number of LCS-queries performed in our algorithm is $O(n\Delta)$. So, together with the complexity of the initialization of $Suf[M-1][\cdot]$, the complexity of this algorithm is $O(n\Delta)$.

This algorithm outperforms the other two algorithms solving $\texttt{MinHDMatch}_{\texttt{Reg}}$ which we mentioned, and, for $\Delta = 0$, it is a reformulation of the greedy algorithm solving $\texttt{Match}_{\texttt{Reg}}$. □

Now it is not hard to show the following result by exponential search.

Theorem 3.5. $\texttt{MinHDMatch}_{\texttt{Reg}}$ *can be solved in $O(n\Phi)$ time, where $\Phi = d_{\texttt{HAM}}(\alpha, w)$.*

Proof. We use the algorithm of Theorem 3.4 for $\Delta = 2^i$, for increasing values of i starting with 1 and repeating until the algorithm returns a positive answer and computes $\Phi = d_{\texttt{HAM}}(\alpha, w)$. The algorithm is clearly correct. Moreover, the value of i which was considered last is such that $2^{i-1} < \Phi \leq 2^i$. So $i = \lceil \log_2 \Phi \rceil$, and the total complexity of our algorithm is $O(n \sum_{i=1}^{\lceil \log_2 \Phi \rceil} 2^i) = O(n\Phi)$. □

3.2.2 Lower Bounds for $\texttt{HDMatch}_{\texttt{Reg}}$ and $\texttt{MinHDMatch}_{\texttt{Reg}}$.

In order to show that $\texttt{MinHDMatch}_{\texttt{Reg}}$ and $\texttt{HDMatch}_{\texttt{Reg}}$ cannot be solved by algorithms running polynomially faster than the algorithms from Theorems 3.4 and 3.5, we will reduce the Orthogonal Vectors problem `OV` [32] to $\texttt{HDMatch}_{\texttt{Reg}}$. The overall structure of our reduction is similar to the one used for establishing hardness of computing edit distance [13, 35] or LCS [36], however we needed to construct gadgets specific to our problem. We recall the `OV` problem.

Orthogonal Vectors: `OV`

Input:	Two sets U, V consisting each of n vectors from $\{0,1\}^d$, where $d \in \omega(\log n)$.
Question:	Do vectors $u \in U, v \in V$ exist, such that u and v are orthogonal, i.e., for all $1 \leq k \leq d$, $v[k]u[k] = 0$ holds?

In general, for a vector $u = (u[1], \ldots, u[d]) \in \{0,1\}^d$, the bits $u[i]$ are called coordinates. It is clear that, for input sets U and V as in the above definition, one can solve `OV` trivially in $O(n^2 d)$ time. The following conditional lower bound is known for `OV`.

Lemma 3.6 (`OV`-Conjecture)**.** `OV` *can not be solved in $O(n^{2-\epsilon} d^c)$ for any $\epsilon > 0$ and constant c, unless the Strong Exponential Time Hypothesis (SETH) fails.*

See [32, 166] and the references therein for a detailed discussion regarding conditional lower bounds related to OV. In this context, we can show the following result.

Theorem 3.7. $\texttt{HDMatch}_{\texttt{Reg}}$ *can not be solved in $O(|w|^h \Delta^g)$ time (or in $O(|w|^h |\alpha|^g)$ time) with $h + g = 2 - \epsilon$ for some $\epsilon > 0$, unless the* `OV`*-Conjecture fails.*

Proof. We reduce $\mathtt{OV}$ to $\mathtt{MinHDMatch}_{\mathtt{Reg}}$. For this, we consider an instance of $\mathtt{OV}$: $U = \{u_1, \dots, u_n\}$ and $V = \{v_1, \dots, v_n\}$, with $U, V \subset \{0,1\}^d$. We transform this $\mathtt{OV}$-instance into a $\mathtt{HDMatch}_{\mathtt{Reg}}$-instance (α, w, Δ), where $\Delta = n(d+1) - 1$. More precisely, we ensure that for the respective $\mathtt{HDMatch}_{\mathtt{Reg}}$-instance, there exists a way to replace the variables with strings leading to exactly $n(d+1)$ mismatches between the image of α and w if and only if no two vectors u_i and v_j are orthogonal. But, if there exists at least one orthogonal pair of vectors u_i and v_j, there also exists a way to replace the variables of α such that the resulting string has strictly less than $n(d+1)$ mismatches to w. Both $|w|$ and $|\alpha|$ are in $O(nd)$, and can be built in $O(nd)$ time. The reduction consists of three main steps. First we will present a gadget for encoding the single coordinates of vectors u_i and v_i from U and V, respectively. Then we will show another gadget to encode a full vector of each respective set. And, finally, we will show how to assemble these gadgets of the vectors from set U into the word w and from V into α.

First gadget. Let $u_i = (u_i[1], u_i[2], \dots, u_i[d]) \in U, v_j = (v_j[1], v_j[2], \dots, v_j[d]) \in V$ and let k be a position of these vectors. We define the following gadgets:

$$A'(i_k) = \begin{cases} \mathtt{001}, & \text{if } u_i[k] = 0. \\ \mathtt{100}, & \text{if } u_i[k] = 1. \end{cases} \qquad B'(j_k) = \begin{cases} \mathtt{000}, & \text{if } v_j[k] = 0. \\ \mathtt{011}, & \text{if } v_j[k] = 1. \end{cases}$$

Note that, when aligned, the pair of strings $(A'(i_k), B'(j_k))$ produces exactly one mismatch if and only if $u_i[k] \cdot v_j[k] = 0$; otherwise it produces three mismatches. So, $A'(i_k)$ and $B'(j_k)$ encode the single coordinates of u_i and v_j respectively.

Further, we construct a gadget $X' = \mathtt{010}$ that produces always one mismatch if aligned to any of the strings $B'(j_k)$ corresponding to coordinates $v_j[k]$. See also Figure 3.1.

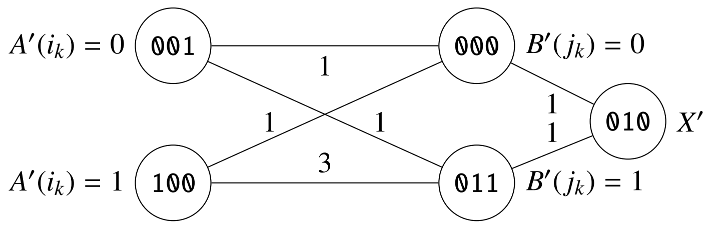


Figure 3.1: Gadgets for the encoding of single coordinates of the vectors. On each edge we wrote the number of mismatches between the strings in the nodes connected by that edge.

Second gadget. The gadget $A(i)$ encodes the vector u_i, for $1 \le i \le n$, while the gadget $B(j)$ encodes the vector v_j, for $1 \le j \le n$. We construct these gadgets such that aligning $B(j)$ to $A(i)$ with a minimum number of mismatches yields exactly d mismatches, if the two corresponding vectors are orthogonal, and exactly $d+1$ mismatches, otherwise. Moreover, we show that any other alignment of the gadgets $B(j)$ with other factors of w yields more mismatches.

In order to assemble the gadgets $A(i)$ and $B(j)$, for $1 \leq i, j \leq n$, we extend the terminal alphabet by three new symbols $\{\mathtt{a}, \mathtt{b}, \#\}$, as well as use two fresh variables x_j, y_j for each vector v_j. The gadgets $A(i)$, for all i, and, respectively, the gadgets $B(j)$, for all j, consist of the concatenation of the coordinate gadgets $A'(i_k)$ and, respectively, $B'(j_k)$ from left to right, in ascending order of k. Each two such consecutive gadgets $A'(i_k)$ and $A'(i_{k+1})$ (respectively, $B'(j_k)$ and $B'(j_{k+1})$) are separated by ###. We prepend to $A(i)$ the string $\mathtt{bba}$ and append the string $\mathtt{bbb}X$, where $X = (X'\#\#\#)^{d-1}X'$. In the case of $B(j)$, we prepend $x_j\mathtt{bba}$ and append y_j. The full gadgets $A(i)$ and $B(j)$ are defined as follows.

- $A(i) = \mathtt{bba}A'(i_1)\#\#\#A'(i_2)\#\#\# \ldots A'(i_d)\mathtt{bbb}X$
- $B(j) = x_j\mathtt{bba}B'(j_1)\#\#\#B'(j_2)\#\#\# \ldots B'(j_d)y_j$.

For simplicity of the exposure, let $B'(j) = \mathtt{bba}B'(j_1)\#\#\#B'(j_2)\#\#\# \ldots \#\#\#B'(j_d)$.

Note that $|A(i)|$ is the same for all i, so we can define $M = |A(i)|$.

Final assemblage. To define the word w, we use a new terminal \$. The word w is:

- $w = \$^M A(1)\$^M A(2)\$^M \ldots A(n)\$^M A(1)\$^M A(2) \ldots \$^M A(n)\$^M$

To define α, we use two new fresh variables x and y. The pattern α is:

- $\alpha = x\$^M B(1)\$^M B(2)\$^M \ldots \$^M B(n)\$^M y$.

The correctness of the reduction. We show that there exists a way to align α with w with $< n(d+1)$ mismatches if and only if a pair of orthogonal vectors $u_i \in U$ and $v_j \in V$ exists. Otherwise, there exists an alignment of α to w with exactly $n(d+1)$ mismatches.

To formally prove that the reduction fulfills this requirement, we proceed as follows.

A general idea: the repetition of the gadgets $A(i)$ in the word w guarantees that, if needed, a pair of gadgets $A(i)$ and $B(j)$, corresponding to the vectors $u_i \in U$ and, respectively, $v_j \in V$, can be aligned. More precisely, we can align $B'(j)$ to $\mathtt{bba}A'(i_1)\#\#\# \ldots A'(i_d)$. The variables x, y and x_j, y_j, for $j \in \{1, \ldots, n\}$, act as spacers: they allow us to align a string $B'(j)$ to the desired factor of w. This kind of alignment is enough for our purposes, as we only need to find one orthogonal pair of vectors, not all of them; however, we need enough space in w for the factors of α occurring before and after $B'(j)$, thus the repetition of the $A(i)$ gadgets.

We now analyse how a factor $B'(j)$ can be aligned to a factor of w. The main idea is to show that if there are no orthogonal vectors, then any alignment of $B'(j)$ to a factor of w creates at least $d+1$ mismatches. Otherwise, we can align it with d mismatches only.

Case 1: $B'(j)$ is aligned to a factor $w[i : h]$ of w which starts with \$. Then the prefix bba of $B'(j)$ causes at least two mismatches, as the first b in bba is aligned to a \$ letter, while the a is aligned to either a b letter (from a bba factor) or a \$ letter. The rest of $B'(j)$ causes, overall, at least d mismatches, one per each group $B'(j_k)$. So, in this case, we have at least $d + 2$ mismatches caused by $B'(j)$.

Case 2: $B'(j)$ is aligned a factor $w[i : h]$ of w which ends with \$. Then, its prefix bba cannot be aligned to a factor bba of w. So, the a of the prefix bba of $B'(j)$ produces one mismatch, while the suffix $B'(j_d)$ causes at least 2 mismatches. The rest of $B'(j)$ causes at least $d - 1$ mismatches, one per each remaining group $B'(j_k)$. So, in this case, we have again at least $d + 2$ mismatches caused by $B'(j)$.

Case 3: $B'(j)$ is aligned exactly to the factor bba$A'(i_1)$###$\ldots A'(i_d)$ and u_i and v_j are orthogonal, then $B'(j)$ causes exactly d mismatches.

Case 4: $B'(j)$ is aligned exactly to the factor bba$A'(i_1)$###$\ldots A'(i_d)$ and u_i and v_j are not orthogonal, then $B'(j)$ causes at least $d + 2$ mismatches.

Case 5: $B'(j)$ is aligned exactly to the factor bbb$X$, then $B'(j)$ causes $d + 1$ mismatches.

Case 6: $B'(j)$ is aligned to a factor starting strictly inside bba$A'(i_1)$###$\ldots A'(i_d)$, then the prefix bba of $B'(j)$ cannot be aligned to a factor bba of w, so it causes at least two mismatches (from the alignment of ba). The rest of $B'(j)$ causes at least d mismatches, one per each group $B'(j_k)$. So, overall, $B'(j)$ causes at least $d + 2$ mismatches in this case.

To ease the understanding, cases 3 and 4 are illustrated in the following table: when aligning $A(i)$ to $B(j)$, to obtain the desired number of mismatches, we can match the parts of $A(i)$ to the parts of $B(j)$ as described in this table in the two cases 3. and 4.

Gadget	I	II	III	IV	mismatches
$A(i) =$	ε	bba$A'(i_1)$###$\ldots$###$A'(i_d)$	bbbX' ###$\ldots$###X'	ε	
3. $B(j) =$	x_j	bba$B'(j_1)$###$\ldots$###$B'(j_d)$	y_j	ε	d (in II)
4. $B(j) =$	ε	x_j	bba$B'(j_1)$###$\ldots$###$B'(j_d)$	y_j	$d + 1$ (in IV)

Wrapping up, there are no other ways than those described in cases 1-6 above in which $B'(j)$ can be aligned to a factor of w. In particular, in order to reach an alignment with at most $n(d + 1) - 1$ mismatches, at least one $B'(j)$ should be aligned to a factor of w such that it only causes d mismatches (as in case 3). Thus, in that case we would have a pair of orthogonal vectors. Conversely, if there exist u_i and v_j which are orthogonal and $i \geq j$, then we can align $B'(j)$ to the occurrence of bba$A'(i_1)$###$\ldots A'(i_d)$ from the first $A(i)$ and all the other gadgets $B'(\ell)$ to factors bbb$X$, and obtain a number of $n(d + 1) - 1$ mismatches. Note that such an alignment is possible as there exist at least $j - 1$ factors bbb$X$ before the first $A(i)$ and at least n more occurrences of bbb$X$ after it; moreover the variables x_ℓ and y_ℓ can be used to align as desired the strings $B'(v_\ell)$ to the respective bbb$X$ factors of w. If there exist u_i and v_j which are orthogonal and $i < j$, then we can align $B'(j)$ to the occurrence of $\mathtt{bba}A'(i_1)^{\#\#\#}A'(i_2)^{\#\#\#}\ldots A'(i_d)$ from the second $A(i)$ and all the other gadgets $B'(\ell)$ to factors $\mathtt{bbb}X$, and obtain again a number of $n(d+1)-1$ mismatches. This is possible for similar reasons to the ones described above.

This shows that our reduction is correct. The instance of $\mathtt{OV}$ defined by U and V contains two orthogonal vectors if and only the instance of $\mathtt{HDMatch}_{\mathtt{Reg}}$ defined by w, α, and $\Delta = n(d+1)-1$ can be answered positively. Moreover, the instance of $\mathtt{HDMatch}_{\mathtt{Reg}}$ can be constructed in $O(nd)$ time and we have that $|w|, |\alpha|, \Delta \in \Theta(nd)$.

Assume now that there exists a solution of $\mathtt{HDMatch}_{\mathtt{Reg}}$ running in $O(|w|^g|\alpha|^h)$ with $g+h = 2-\epsilon$ for some $\epsilon < 0$. This would lead to a solution for $\mathtt{OV}$ running in $O(nd + (nd)^{2-\epsilon})$, a contradiction to the $\mathtt{OV}$-conjecture. Similarly, if there exists a solution of $\mathtt{HDMatch}_{\mathtt{Reg}}$ running in $O(|w|^g\Delta^h)$ with $g+h = 2-\epsilon$ for some $\epsilon < 0$, then there exists a solution for $\mathtt{OV}$ running in $O(nd + (nd)^{2-\epsilon})$, a contradiction to the $\mathtt{OV}$-conjecture. This proves our statement. □

Remark 3.8. *An immediate consequence of the previous theorem is that* $\mathtt{MinHDMatch}_{\mathtt{Reg}}$ *can not be solved in* $O(n^h d_{\mathtt{HAM}}(\alpha, w)^g)$ *time (or in* $O(|w|^h|\alpha|^g)$ *time) with* $h+g = 2-\epsilon$ *for some* $\epsilon > 0$*, unless the* $\mathtt{OV}$*-Conjecture fails. Thus, as* $d_{\mathtt{HAM}}(\alpha, w) \le |\alpha|$*,* $\mathtt{MinHDMatch}_{\mathtt{Reg}}$ *and* $\mathtt{HDMatch}_{\mathtt{Reg}}$ *cannot be solved polynomially faster than our algorithms, unless the* $\mathtt{OV}$*-Conjecture fails.*

3.3 Matching Patterns with Repeated Variables

In Section 3.2 we have shown that if no variable occurs more than once in the input pattern α, then the problems $\mathtt{HDMatch}$ and $\mathtt{MinHDMatch}$ can be solved in polynomial time. Let us now consider patterns where variables are allowed to occur more than once, i.e., patterns with repeated variables. To recall pattern classes and their properties and relations, a detailed discussion can be found in the Sections 2.2 and 2.3.

If α is a pattern and $x \in \mathtt{var}(\alpha)$, then an x-block is a factor $\alpha[i:j]$ such that $\alpha[i:j] \in \mathtt{1Var}$ with $\mathtt{var}(\alpha[i:j]) = x$ and it is length-maximal with this property: it cannot be extended to the right or to the left without introducing a variable different from x.

The next lemma is fundamental for the results of this section.

Lemma 3.9. *Given a set of words* $w_1, \ldots, w_p \in \Sigma^m$*, we can find in* $O(|\Sigma| + mp)$ *a* median string *for* $\{w_1, \ldots, w_p\}$*, i.e. a string* w *such that* $\sum_{j=1}^{p} d_{\mathtt{HAM}}(w_i, w)$ *is minimal.*

Proof. We will use an array C with Σ elements, called counters, indexed by the letters of Σ, and all initially set to 0. For each i between 1 and m, we count how many times each letter of Σ occurs in the multi-set $\{w_1[i], w_2[i], \ldots, w_p[i]\}$ using C. Let $w[i]$ be the most frequent letter of this multi-set. After computing $w[i]$, we reset the counters which were changed in this iteration, and repeat the algorithm for $i+1$. After going through all values of i, we return the word $w = w[1]w[2]\dots w[m]$ as the answer to the problem. The correctness of the algorithm is immediate, while its complexity is clearly $O(|\Sigma| + mp)$. □

The typical use of this lemma is the following: we identify the factors of w to which a repeated variable is aligned, and then compute the optimal assignment of this variable. Based on this, the following theorem can now be shown.

Theorem 3.10. $\texttt{MinHDMatch}_{\texttt{1Var}}$ *and* $\texttt{HDMatch}_{\texttt{1Var}}$ *can be solved in* $O(n)$ *time.*

Proof. It is enough to show how to solve $\texttt{MinHDMatch}_{\texttt{1Var}}$.

Recall that we were given a word w, of length n, and a pattern α, of length m. Let x be the single variable that occurs in α and, for simplicity, we denote by m_x the number of occurrences of x in α, i.e., $m_x = |\alpha|_x$. Thus, $\alpha = \prod_{i=1}^{m_x}(v_{i-1}x)v_{m_x}$, where $v_i \in \Sigma^*$ for all $i \in \{1,\dots,m_x\}$.

Let $m' = m - m_x$ be the number of terminal symbols of α. It is clear that x should be mapped to a string of length $\ell = \frac{n-m'}{m_x}$. If ℓ is not an integer, there exists no string u which can be obtained from α by substituting x with a terminal-word such that $|u| = |w|$ and $d_{\texttt{HAM}}(u,w)$ is finite. So, let us assume ℓ is an integer.

Now we know that we want to compute a string u which can be obtained from α by substituting x with a terminal-word u_x of length exactly ℓ. Moreover, $u = \prod_{i=1}^{m_x}(v_{i-1}u_x)v_{m_x}$. We define the factors $w_1,\dots,w_{m_x}$ of w such that $w_i = w[a_i+1 : a_i+\ell_x]$ and $a_i = |\prod_{j=1}^{i-1}(v_{i-1}u_x)v_i|$. These are the factors that would align to the occurrences of u_x when aligning u with w. As the factors v_i always create the same number of mismatches to the corresponding factors of w, irrespective on the choice of u_x, we need to choose u_x such that $\sum_{j=1}^{m_x} d_{\texttt{HAM}}(w_i,u_x)$ is minimal. For this, we can use Lemma 3.9, and compute u_x in $O(|\Sigma| + m_x\ell_x)$ time. As it is our assumption that $|\Sigma| \le n$, we immediately get that u_x can be computed in $O(n)$ time. So u can be computed in $O(n)$ time. To solve $\texttt{MinHDMatch}_{\texttt{1Var}}$, we simply return $d_{\texttt{HAM}}(u,w)$, and this can be again computed in linear time. □

By a standard dynamic programming approach, we use the previous result to obtain a polynomial-time solution for $\texttt{MinHDMatch}_{\texttt{NonCross}}$ based on the solution for $\texttt{MinHDMatch}_{\texttt{1Var}}$ (in the statement, $p = |\texttt{var}(\alpha)|$).

Theorem 3.11. $\texttt{MinHDMatch}_{\texttt{NonCross}}$ *and* $\texttt{HDMatch}_{\texttt{NonCross}}$ *can be solved in* $O(n^3p)$ *time.*

Proof. It is enough to show how to solve $\texttt{MinHDMatch}_{\texttt{NonCross}}$. Once more, we were given a word w, of length n, and a pattern α, of length m. Assume $\texttt{var}(\alpha) = \{x_1,\dots,x_p\}$, and we have $\alpha = \beta_1\beta_2\cdots\beta_p$, where β_{2i+1} is an x_{2i+1}-block, for all i such that $1 \le 2i+1 \le m$, and $\texttt{var}(\beta_{2i}) = \{x_{2i}\}$, for all i such that $1 < 2i \le m$. Let $\alpha_\ell = \beta_1\cdots\beta_\ell$, for $\ell \ge 1$.

The idea of our algorithm is the following.

For ℓ from 1 to p, we define $Dist[j][\ell] = d_{\mathtt{HAM}}(\alpha_\ell, w[1 : j])$ for all prefixes $w[1 : j]$ of w. This matrix can be computed by dynamic programming.

For $\ell = 1$, we can use Theorem 3.10 to compute each element $Dist[j][1]$ in linear time. So, $Dist[\cdot][1]$ is computed in $O(n^2)$ time.

Consider now the case when $\ell > 1$ and assume we have computed the array $Dist[\cdot][\ell - 1]$. For a position j of the word w, we compute $Dist[j][\ell] = \min\{Dist[j'][\ell-1]+d_{\mathtt{HAM}}(\beta_\ell, w[j'+1 : j]) \mid j' \leq j\}$, where $d_{\mathtt{HAM}}(\beta_\ell, w[j' + 1 : j])$ is computed, once more, by Theorem 3.10. It is clear that computing each element $Dist[j][\ell]$ as described above is correct, and that this computation takes $O(n^2)$ time.

Therefore, we can compute all elements of the matrix $Dist[\cdot][\cdot]$ in $O(n^3 p)$ time. We return $Dist[n][p]$ as the answer to $\mathtt{MinHDMatch}_{\mathtt{NonCross}}$. □

The results presented so far show that $\mathtt{MinHDMatch}_P$ and $\mathtt{HDMatch}_P$ can be solved in polynomial time, as long as we do not allow interleaved occurrences of variables in the patterns of the class P. We now consider the case of `1RepVar`-patterns, the simplest class of patterns which permits interleaved occurrences of variables.

For simplicity, in the results regarding `1RepVar` we assume that the variable which occurs more than once in the input pattern is denoted by x.

Theorem 3.12. *$\mathtt{MinHDMatch}_{\mathtt{1RepVar}}$ and $\mathtt{HDMatch}_{\mathtt{1RepVar}}$ can be solved in $O(n^{k+2}m)$ time, where k is the number of x-blocks in the input pattern α.*

Proof. Once more, we only show how $\mathtt{MinHDMatch}_{\mathtt{1RepVar}}$ can be solved. The result for the problem $\mathtt{HDMatch}_{\mathtt{1RepVar}}$ follows then immediately.

In $\mathtt{MinHDMatch}_{\mathtt{1RepVar}}$, we are given a word w, of length n, and a pattern α, of length m, which, as stated above, has exactly k x-blocks. Thus $\alpha = \prod_{i=1}^{k}(\gamma_{i-1}\beta_i)\gamma_k$, where the factors β_i, for $i \in \{1, \ldots, k\}$, are the x-blocks of α. It is easy to observe that $\mathtt{var}(\gamma_i) \cap \mathtt{var}(\gamma_j) = \emptyset$, for all i and j, and $\gamma = \gamma_0\gamma_1 \cdots \gamma_k$ is a regular pattern.

When aligning α to w we actually align each of the patterns γ_j and β_i, for $0 \leq j \leq k$ and $1 \leq i \leq k$, to respective factors of the word w. Moreover, the factors to which these patterns are respectively aligned are completely determined by the length ℓ of the image of x, and the starting positions h_i of the factors aligned to the patterns β_i, for $1 \leq i \leq k$. Knowing the length ℓ of the image of x, we can also compute, for $1 \leq i \leq k$, the length ℓ_i of β_i, when x is replaced by a string of length ℓ. In this case, γ_0 is aligned $u_0 = w[1..h_1 - 1]$ and, for $1 \leq i \leq k$, β_i is aligned to $w_i = w[h_i : h_i + \ell_i - 1]$ and γ_i is aligned $u_i = w[h_{i-1} + \ell_{i-1} : h_i - 1]$. Thus, $\beta_1 \cdots \beta_k$ matches $w_1 \cdots w_k$ and we can use Theorem 3.10 to determine $d_{\mathtt{HAM}}(\beta_1 \cdots \beta_k, w_1 \cdots w_k)$ (or, in other words, determine the string u_x that should replace x in order to realize this Hamming distance). Further, we can use Theorem 3.5 to compute $d_{\mathtt{HAM}}(\gamma_i, u_i)$, for all $i \in \{0, \ldots, k\}$. Adding all these distances up, we obtain a total distance $D_{\ell,h_1,\ldots,h_k}$; this value depends on $\ell, h_1, \ldots, h_k$.

So, we can simply iterate over all possible choices for $\ell, h_1, \ldots, h_k$ and find $d_{\mathtt{HAM}}(\alpha, w)$ as the minimum of the numbers $D_{\ell,h_1,\ldots,h_k}$.

By the explanations above, it is straightforward that the approach is correct: we simply try all possibilities of aligning α with w. The time complexity is, for each choice of $\ell, h_1, \ldots, h_k$, $O(\sum_{i=1}^{k} |w_i|) \subseteq O(n)$ for the part corresponding to the computation of the optimal alignment between the factors β_i and the words w_i, and $O(\sum_{i=0}^{k} |u_i| d_{\mathtt{HAM}}(\gamma_i, u_i)) \subseteq O(nm)$ for the part corresponding to the computation of the optimal alignment between the factors γ_i and the words u_i. So, the overall complexity of this algorithm is $O(n^{k+2}m)$. □

We can also show the following more general result.

Theorem 3.13. $\mathtt{MinHDMatch}_{\mathtt{kLOC}}$ *and* $\mathtt{HDMatch}_{\mathtt{kLOC}}$ *can be solved in* $O(n^{2k+2}m)$ *time.*

Proof. We only present the solution for $\mathtt{MinHDMatch}_{\mathtt{kLOC}}$ (as it trivially works in the case of $\mathtt{HDMatch}_{\mathtt{kLOC}}$ too).

Let us note that, by the results in [51], we can compute a marking sequence of α in $O(m^{2k}k)$ time. So, after such a preprocessing phase, we can assume that we have a word w, a k-local pattern α (with p variables) with a witness marking sequence $x_1 \leq \ldots \leq x_p$ for the k-locality of α, and we want to compute $d_{\mathtt{HAM}}(\alpha, w)$.

Generally, the main idea behind matching $\mathtt{kLOC}$-patterns is that when looking for possible ways to align such a pattern α to a word w we can consider the variables in the order given by the marking sequence, and, when reaching variable x_i, we try all possible assignments for x_i. The critical observation here is that after each such assignment of a new variable, we only need to keep track of the way the $t \leq k$ length-maximal factors of α, which contain only marked variables and terminals, match (at most) $t \leq k$ factors of w.

We will use this approach in our algorithm for $\mathtt{MinHDMatch}_{\mathtt{kLOC}}$.

The first step of this algorithm is the following. We go through α and identify all x_1-blocks: $\beta_{1,1}, \ldots, \beta_{1,j_1}$. Because α is k-local, we have that $j_1 \leq k$. For each $2j_1$-tuple $(i_1, \ldots, i_{2j_1})$ of positions of w, we compute the minimum number of mismatches if we align (simultaneously) the patterns β_g to the factors $w[i_{2g-1} : i_{2g}]$, for g from 1 to j_1, respectively. This reduces to finding an assignment for x_1 which aligns optimally the patterns $\beta_{1,g}$ to the respective factors, and can be done in $O(n)$ time using Theorem 3.10. For each $2j_1$-tuple $(i_1, \ldots, i_{2j_1})$ of positions of w, we denote by

$M_1(i_1, \ldots, i_{2j_1})$ the minimum number of mismatches resulting from the (simultaneous) alignment of the patterns $\beta_{1,g}$ to the factors $w[i_{2g-1} : i_{2g}]$, for g from 1 to j_1, respectively. Clearly, M_1 can be seen as a j_1-dimensional array.

Assume that after $h \geq 1$ steps of our algorithm we have computed the factors $\beta_{h,1}, \ldots, \beta_{h,j_h}$ of α, which are length-maximal factors of α which only contain the variables $x_1, \ldots, x_h$ and terminals (i.e., extending them to the left or right would introduce a new variable x_ℓ with $\ell > h$); as α is k-local, we have $j_h \leq k$. Moreover, for each $2j_h$-tuple $(i_1, \ldots, i_{2j_h})$ of positions of w, we have computed $M_h(i_1, \ldots, i_{2j_h})$, the minimum number of mismatches if we align (simultaneously) the patterns $\beta_{h,g}$ to the factors $w[i_{2g-1} : i_{2g}]$, for g from 1 to j_h, respectively. M_h is implemented as a j_h dimensional array, and this assumption clearly holds after the first step.

We now explain how step $h + 1$ is performed.

1. We compute the factors $\beta_{h+1,1}, \ldots, \beta_{h+1,j_{h+1}}$ of α, which are length-maximal factors of α which only contain the variables $x_1, \ldots, x_{h+1}$ and terminals (i.e., extending them to the left or right would introduce a new variable x_ℓ with $\ell > h + 1$). Clearly, $\beta_{h+1,r}$ is either an x_{h+1}-block or it has the form $\beta_{h+1,r} = \gamma_{r,0}\beta_{h,a_r}\gamma_{r,1} \cdots \beta_{r,a_r+b_r}\gamma_{r,b_r+1}$ where the patterns $\gamma_{r,t}$ contain only the variable x_{h+1} and terminals and extending $\beta_{h+1,r}$ to the left or right would introduce a new variable x_ℓ with $\ell > h + 1$.

2. We initialize the values $M_{h+1}(i_1, \ldots, i_{2j_{h+1}}) \leftarrow \infty$, for each $2j_{h+1}$-tuple $(i_1, \ldots, i_{2j_{h+1}})$ of positions of w.

3. For each $\ell \leq n$ (where ℓ corresponds to the length of the image of x_{h+1}) and each $2j_h$-tuple $(i_1, \ldots, i_{2j_h})$ of positions of w such that $M_h(i_1, \ldots, i_{2j_h})$ is finite do the following:

   (a) We compute the tuple $(i'_1, \ldots, i'_{2j_{h+1}})$ such that $\beta_{h+1,g}$ is aligned to the factor $w[i'_{2g-1} : i'_{2g}]$, for g from 1 to j_{h+1}, respectively. This can be computed based on the fact that the factors $\beta_{h,g}$ are aligned to the factors $w[i_{2g-1} : i_{2g}]$, for g from 1 to j_h, respectively, and the image of x_{h+1} has length ℓ.

   (b) We compute the factors of w aligned to x_{h+1} in the alignment computed in the previous line. Then, we can use the algorithm from Theorem 3.10 and the value of $M_h(i_1, \ldots, i_{2j_h})$ to compute an assignment for x_{h+1} which aligns optimally the patterns $\beta_{h+1,g}$ to the corresponding factors of w.

   (c) If the number of the mismatches in this alignment is smaller than the current value of $M_{h+1}(i'_1, \ldots, i'_{2j_{h+1}})$, we update $M_{h+1}(i'_1, \ldots, i'_{2j_{h+1}})$.

This dynamic programming approach is clearly correct. In $M_{h+1}(i_1, \dots, i_{2j_{h+1}})$ we have the optimal alignment of the patterns $\beta_{h+1,1}, \dots, \beta_{h+1,j_{h+1}}$ to $w[i_1 : i_2], \dots, w[i_{2j_{h+1}-1} : i_{2j_{h+1}}]$. As far as the complexity is concerned, the lines 1, 3.a, 3.b, 3.c can be implemented in linear time, while the for-loop is iterated $O(n^{2k+1})$ times. Line 2 takes $O(n^{2k})$ times. The whole computation in step $h+1$ of the algorithm takes, thus, $O(n^{2k+1})$ time.

Now, we execute the procedure described above for h from 2 to m, and, in the end, we compute the array M_m. The answer to our instance of the problem $\texttt{MinHDMatch}_{\texttt{kLOC}}$ is $M_m(1, n)$. The overall time complexity needed to perform this computation is $O(mn^{2k+1})$ time. □

Note that NonCross-patterns are 1-local, while the locality of an 1RepVar-pattern is upper bounded by the number of x-blocks. However, the algorithms we obtained in those particular cases are more efficient than the ones which follow from Theorem 3.13.

The fact that Lemma 3.9 is used as the main building block for our results regarding $\texttt{HDMatch}_P$ and $\texttt{MinHDMatch}_P$ for $P \in \{\texttt{1RepVar}, \texttt{kLOC}\}$, suggests that these problems could be closely related to the following well-studied problem [122, 57, 29, 37].

Consensus Patterns: CP

Input: k strings $w_1, \dots, w_k \in \Sigma^\ell$, integer $m \in \mathbb{N}$ with $m \le \ell$, an integer $\Delta \le mk$.

Question: Do the strings s, of length m, and $s_1, \dots, s_k$, factors of length m of each $w_1, \dots, w_k$, respectively, exist, such that $\sum_{i=1}^{k} d_{\texttt{HAM}}(s_i, s) \le \Delta$?

Exploiting this connection, and following the ideas of [122], we can show the following theorem. In this theorem we restrict to the case when the input word w of $\texttt{MinHDMatch}_{\texttt{1RepVar}}$ is over $\Sigma = \{1, \dots, \sigma\}$ of constant size σ.

Theorem 3.14. *For each constant $r \ge 3$, there exists an approximation algorithm with run-time $O(n^{r+3})$ for $\texttt{MinHDMatch}_{\texttt{1RepVar}}$ whose output distance is at most:*

$$\min\left\{2, \left(1 + \frac{4\sigma - 4}{\sqrt{e}(\sqrt{4r+1} - 3)}\right)\right\} d_{\texttt{HAM}}(\alpha, w)$$

Proof. We first note that there exists a relatively simple algorithm solving $\texttt{MinHDMatch}_{\texttt{1RepVar}}$ such that the output distance is no more than $2d_{\texttt{HAM}}(\alpha, w)$ (which also works for integer alphabets).

Indeed, assume that we have a substitution h for which $d_{\texttt{HAM}}(h(\alpha), w) = d_{\texttt{HAM}}(\alpha, w)$. Assume that the repeated variable x is mapped by h to a string u and the t occurrences of x are aligned, under h, to the factors $w_1, w_2, \dots, w_t$ of w. Now, let w_i be such $d_{\texttt{HAM}}(u, w_i) \le d_{\texttt{HAM}}(u, w_j)$ for all $j \ne i$. Let us consider now the substitution h' which substitutes x by w_i and all the other variables exactly as h did. We claim that $d_{\texttt{HAM}}(h'(\alpha), u) \le 2d_{\texttt{HAM}}(h(\alpha), u)$. It is easy to see that $d_{\texttt{HAM}}(h'(\alpha), w) - d_{\texttt{HAM}}(h(\alpha), w) = \sum_{j=i}^{t}(d_{\texttt{HAM}}(w_i, w_j) - d_{\texttt{HAM}}(u, w_j)) \le \sum_{j=i}^{t}(d_{\texttt{HAM}}(w_i, u) + d_{\texttt{HAM}}(u, w_j) - d_{\texttt{HAM}}(u, w_i))$

(where the last inequality follows from the triangle inequality for the Hamming distance). Thus, $d_{\texttt{HAM}}(h'(\alpha), w) - d_{\texttt{HAM}}(h(\alpha), w) \leq \sum_{j=i}^{t} d_{\texttt{HAM}}(w_i, u) \leq \sum_{j=i}^{t} d_{\texttt{HAM}}(w_j, u) \leq d_{\texttt{HAM}}(h(\alpha), u)$. So our claim holds.

A consequence of the previous observation is that there exists a substitution h' that maps x to a factor of w and produces a string $h'(\alpha)$ such that $d_{\texttt{HAM}}(h'(\alpha), u) \leq 2d_{\texttt{HAM}}(\alpha, u)$. So, for each factor u of w, we x by u in α to obtain a regular pattern α', then use Theorem 3.5 to compute $d_{\texttt{HAM}}(\alpha', w)$. We return the smallest value $d_{\texttt{HAM}}(\alpha', w)$ achieved in this way. Clearly, this is at most $2d_{\texttt{HAM}}(\alpha, u)$. The complexity of this algorithm is $O(n^4)$, as it simply uses the quadratic algorithm of Theorem 3.5 for each factor of w.

We will now show how this algorithm can be modified to produce a value closer to $d_{\texttt{HAM}}(\alpha, w)$, while being less efficient.

The algorithm consists of the following main steps:

1. For $\ell \leq n/r$ and r factors $u_1, \dots, u_r$ of length ℓ of w do the following:
   (a) Compute $u_{u_1,\dots,u_r}$ the median string of $u_1, \dots, u_r$ using Lemma 3.9.
   (b) Let α' be the regular pattern obtained by replacing x by $u_{u_1,\dots,u_r}$ in α.
   (c) Compute the distance $d_{u_1,\dots,u_r} = d_{\texttt{HAM}}(\alpha', w)$ using Theorem 3.5.
2. Return the smallest distance $d_{u_1,\dots,u_r}$ computed in the loop above.

Clearly, for $r = 1$ the above algorithm corresponds to the simple algorithm presented in the beginning of this proof. Let us analyse its performance for an arbitrary choice of r.

The complexity is easy to compute: we need to consider all possible choices for ℓ and the starting positions of $u_1, \dots, u_r$. So, we have $O(n^{r+1})$ possibilities to select the non-overlapping factors $u_1, \dots, u_r$ of length ℓ of w. The computation done inside the loop can be performed in $O(n^2)$ time. So, overall, our algorithm runs in $O(n^{r+3})$ time.

Now, we want to estimate how far away from $d_{\texttt{HAM}}(\alpha, w)$ is the value this algorithm returns. In this case, we will make use of the fact that the input terminal-alphabet is constant. We follow closely (and adapt to our setting) the approach from [122].

Firstly, a notation. In step 1.b of the algorithm above, we align α' to w with a minimal number of mismatches. In this alignment, let $d'_{u_1,\dots,u_r}$ be the total number of mismatches caused by the factors $u_{u_1,\dots,u_r}$ which replaced the occurrences of the variable x in α.

Now, assume that we have a substitution h for which $d_{\mathtt{HAM}}(h(\alpha), w) = d_{\mathtt{HAM}}(\alpha, w) = d_{opt}$. Assume also that the repeated variable x is mapped by h to a string u_{opt} of length L and the t occurrences of x are aligned, under h, to the factors $w_1, w_2, \dots, w_t$ of w. Let d'_{opt} be the number of mismatches caused by the alignment of the images of the t occurrences of x under h to the factors $w_1, w_2, \dots, w_t$. Finally, let $\rho = 1 + \frac{4\sigma-4}{\sqrt{e}(\sqrt{4r+1}-3)}$.

Note that, for $\ell = L$, $u_1, \dots, u_r$ correspond to a set of randomly chosen numbers $i_1, \dots, i_r$ from $\{1, \dots, n\}$: their starting positions. We will show in the following that $E\left[d'_{u_1,\dots,u_r}\right] \le \rho d'_{opt}$. If this inequality holds, then we can apply the probabilistic method: there exists at least a choice of $u_1, \dots, u_r$ of length L such that $d'_{u_1,\dots,u_r} \le \rho d'_{opt}$. As we try all possible lengths ℓ and all variants for choosing $u_1, \dots, u_r$ of length ℓ, we will also consider the choice of $u_1, \dots, u_r$ of length L such that $d'_{u_1,\dots,u_r} \le \rho d'_{opt}$, and it is immediate that, for that, for the respective $u_1, \dots, u_r$ we also have that $d_{u_1,\dots,u_r} \le \rho d_{opt}$. Thus, the value returned by our algorithm is at most ρd_{opt}.

So, let us show the inequality $E\left[d'_{u_1,\dots,u_r}\right] \le \rho d_{opt}$.

For $\mathtt{a} \in \Sigma$, let $f_j(\mathtt{a}) = |\{i \mid 1 \le i \le t, w_i[j] = \mathtt{a}\}|$. Now, for an arbitrary string s of length L, we have that $\sum_{i=1}^{t} d_{\mathtt{HAM}}(w_i, s) = \sum_{j=1}^{L}(t - f_j(s[j]))$. So, for $s = u_{opt}$ we get $\sum_{i=1}^{t} d_{\mathtt{HAM}}(w_i, u_{opt}) = \sum_{j=1}^{L}(t - f_j(u_{opt}[j]))$, and for $s = u_{u_1,\dots,u_r}$ we have that $d'_{opt} = \sum_{i=1}^{t} d_{\mathtt{HAM}}(w_i, u_{u_1,\dots,u_r}) = \sum_{i=j}^{L}(t - f_j(u_{u_1,\dots,u_r}[j]))$.

Therefore, $E\left[d'_{u_1,\dots,u_r}\right] = E\left[\sum_{j=1}^{L}(t - f_j(u_{u_1,\dots,u_r}[j]))\right] = \sum_{j=1}^{L} E\left[t - f_j(u_{u_1,\dots,u_r}[j])\right]$.

Consequently, $E\left[d'_{u_1,\dots,u_r} - d'_{opt}\right] = \sum_{j=1}^{L}(E\left[t - f_j(u_{u_1,\dots,u_r}[j])\right] - t + f_j(u_{opt}[j]))$.

That is, $E\left[d'_{u_1,\dots,u_r} - d'_{opt}\right] = \sum_{j=1}^{L} E\left[f_j(u_{opt}[j]) - f_j(u_{u_1,\dots,u_r}[j])\right]$.

By Lemma 7 of [122], we have that $E\left[f_j(u_{opt}[j]) - f_j(u_{u_1,\dots,u_r}[j])\right] \le (\rho - 1)(t - f_j(u_{opt}[j]))$.

Hence, $E\left[d'_{u_1,\dots,u_r} - d'_{opt}\right] \le (\rho - 1)\sum_{j=1}^{L}(t - f_j(u_{opt}[j])) = (\rho - 1)d'_{opt}$.

So, we indeed have that $E\left[d'_{u_1,\dots,u_r}\right] \le \rho d'_{opt}$.

In conclusion, the statement of the theorem holds. □

It remains open whether other algorithmic results related to CP (such as those from, e.g., [30, 31, 129]) apply to our setting too.

In the following we show two hardness results which explain why the algorithms in Theorems 3.12 and 3.14 are interesting.

Theorem 3.15. $\mathtt{HDMatch}_{\mathtt{1RepVar}}$ *is* $W[1]$*-hard w.r.t. the number of x-blocks.*

Proof. We reduce CP to $\mathtt{HDMatch}_{\mathtt{1RepVar}}$, such that an instance of CP with k different input strings is mapped to an instance of $\mathtt{HDMatch}_{\mathtt{1RepVar}}$ with $k + 1$ x-blocks (where x is the repeated variable), each containing exactly one occurrence of x.

Hence, we consider an instance of CP which consists of k strings $w_1, \dots w_k \in \Sigma^\ell$ of length ℓ and two integer m, Δ defining the length of the target factors and the number of allowed mismatches, respectively.

The instance of $\texttt{HDMatch}_{\texttt{1RepVar}}$ which we construct consists of a text w and a pattern α, such that α contains $k+1$ x-blocks, each with exactly one occurrence of x, and is of polynomial size w.r.t. the size of the CP-instance. Moreover, the number of mismatches allowed in this instance of $\texttt{HDMatch}_{\texttt{1RepVar}}$ is $\Delta' = m + \Delta$. That is, if there exists a solution for the CP-instance with Δ allowed mismatches, then, and only then, we should be able to find a solution of the $\texttt{HDMatch}_{\texttt{1RepVar}}$-instance with $\Delta + m$ mismatches.

The construction of the $\texttt{MinHDMatch}_{\texttt{1RepVar}}$ is realized in such a way that the word w encodes the input strings, while α creates the mechanism for selecting the string s and corresponding factors $s_1, \dots, s_k$. The general idea is that x should be mapped to s, and the factors to which the occurrences of x are aligned should correspond to the strings $s_1, \dots, s_k$.

The structure of the word w and that of the pattern α ensure that, in an alignment of α with w which cannot be traced back to a admissible solution for the CP-instance (that is, the occurrences of x are not aligned to factors of length m of the words $w_1, \dots, w_k$ or x is not mapped to a string of length m) we have at least $M \gg \Delta'$ mismatches, hence it cannot lead to a positive answer for the constructed instance of $\texttt{HDMatch}_{\texttt{1RepVar}}$.

The reduction consists of three main steps. Firstly, we present a pair of gadgets to encode the relation of the strings w_i and their factors s_i, for i from 1 to k. Then, we present a second pair of gadgets, which ensures that, in a positive solution of $\texttt{HDMatch}_{\texttt{1RepVar}}$, the variable x can only be mapped to a string of length m, corresponding to the string s. Finally, we show how to assemble these gadgets into the input word w and the input pattern α for $\texttt{HDMatch}_{\texttt{1RepVar}}$.

First pair of gadgets. We introduce the new letters $\{\mathtt{a}, \mathtt{b}\}$, not contained in the input alphabet of the CP-instance, as well as the variable x and two fresh variables y_i, z_i, for each i form 1 to k. We construct the following two gadgets for each input string w_i with $1 \le i \le k$.

- A gadget to be included in w: $\mathtt{g_i} = \mathtt{w_i} \overbrace{a^M b^M \dots a^M b^M}^{\mathtt{M}}$.
- A gadget to be included in α: $\mathtt{f_i} = \mathtt{y_i x z_i} \overbrace{a^M b^M \dots a^M b^M}^{\mathtt{M}}$.

These gadgets allows us to align the i^{th} occurrence of x to an arbitrary factor of the word w_i, for i from 1 to k.

Second pair of gadgets. In this case, we use three new letters $\{\mathtt{c}, \mathtt{d}, \$\}$ which are not contained in the input alphabet of CP. Also, let $M = (k\ell)^2$. We define two new gadgets.

- A gadget to be included in w: $\mathtt{A_w} = \overbrace{c^M d^M \dots c^M d^M}^{\mathtt{M}} \$^m$.
- A gadget to be included in α: $\mathtt{A}_\alpha = \overbrace{c^M d^M \dots c^M d^M}^{\mathtt{M}} x$.

These gadgets enforce that, in an alignment of α and w, the variable x is mapped to a string of length m, at the cost of exactly m extra mismatches. Note that, because $\Delta \leq km$, we have that $M \gg \Delta$.

Final assemblage. The word w and the pattern α are defined as follows.

- $w = \mathtt{g}_1\mathtt{g}_2 \dots \mathtt{g_k}\mathtt{A_w}$ and $\alpha = \mathtt{f}_1\mathtt{f}_2 \dots \mathtt{f_k}\mathtt{A}_\alpha$.

To wrap up, the instance of $\mathtt{MinHDMatch_{1RepVar}}$ is defined by $w, \alpha, \Delta + m$.

The correctness of the reduction. We will show that our reduction is correct by a detailed case analysis. We consider an alignment of α and w with minimal number of mismatches, and we make the following observations.

A. Firstly, if every g_i is aligned to f_i, for i from i to k, it is immediate that x is mapped to a string of length m, as the last occurrence of x will be aligned to the $\$^m$ suffix of w. Thus, the total number of mismatches between α and w in an alignment with a minimum number of mismatches is upper-bounded by $(k + 1)m$.

B. Secondly, we assume, for the sake of a contradiction, that the length of the image of x is not m. If $|x| > m$ (respectively, $|x| < m$) then the prefix $(\mathtt{c^M d^M})^\mathtt{M}$ of A_α is aligned to a factor of w which starts strictly to the left of (respectively, to the right of) the first position of the prefix $(\mathtt{c^M d^M})^\mathtt{M}$ of A_w. It is not hard to see that this causes at least M mismatches. Indeed, in the case when $|x| > m$, if the factor $(\mathtt{c^M d^M})^\mathtt{M}$ of α is aligned to a factor that starts at least M position to the left of the factor $(\mathtt{c^M d^M})^\mathtt{M}$ of w, the conclusion is immediate; if the factor $(\mathtt{c^M d^M})^\mathtt{M}$ starts less then M positions to the left of the factor $(\mathtt{c^M d^M})^\mathtt{M}$ of w, then each group c^M in α will be aligned to a factor of w that includes at least a $\mathtt{d}$ letter, so we again reach the conclusion. In the case when $|x| < m$, then, again, each group c^M in α will be aligned to a factor of w that includes at least a $\mathtt{d}$ letter, so the alignment leads to at least M mismatches.
So, we can assume from now on that x is mapped to a string of length m. This also implies that A_α and A_w are aligned, so we will largely neglect them from now on.

C. Thirdly, we assume that there exists i such that $|h(y_i)| + |h(z_i)| \neq |w_i| - m$. Let $j = \min\{i \leq k \mid |h(y_i)| + |h(z_i)| \neq |w_i| - m\}$. Then the suffixes $(\mathtt{a^M b^M})^\mathtt{M}$ of g_j and f_j do not align perfectly to each other. If $|h(y_j)| + |h(z_j)| < |w_i| - m$, then the suffix $(\mathtt{a^M b^M})^\mathtt{M}$ of f_j is aligned to a factor of w which starts inside w_j. This immediately causes at least M mismatches, as each group $\mathtt{a^M}$ will overlap to a group of which contains at least one $\mathtt{b}$ letter. If $|h(y_j)| + |h(z_j)| > |w_i| - m$, then the suffix $(\mathtt{a^M b^M})^\mathtt{M}$ of f_j is aligned to a factor of w which starts strictly to the right of the factor w_j. However, because $M = (k\ell)^2 \gg k\ell$, and f_j and g_j are followed by the same

number of factors $(\mathtt{a}^{\mathtt{M}}\mathtt{b}^{\mathtt{M}})^{\mathtt{M}}$ (until the factors A_α and A_w are reached), the factor corresponding to the suffix $(\mathtt{a}^{\mathtt{M}}\mathtt{b}^{\mathtt{M}})^{\mathtt{M}}$ of f_j cannot start more than $k\ell$ positions to the right of w_j. It is then immediate that this factor $(\mathtt{a}^{\mathtt{M}}\mathtt{b}^{\mathtt{M}})^{\mathtt{M}}$ of f_j will cause at least M mismatches: each group $\mathtt{a}^{\mathtt{M}}$ will overlap to a group of which contains at least one $\mathtt{b}$ letter.
So, from now on we can assume that the factors $(\mathtt{a}^{\mathtt{M}}\mathtt{b}^{\mathtt{M}})^{\mathtt{M}}$ of g_j and f_j are aligned.

D. At this point, it is clear that in each alignment of α and w which fulfils the conditions described in items B and C: the variable x is mapped to a string of length m, and its first k occurrences are aligned to factors of the words $w_1, \dots, w_k$. We will now show that for each alignment of α and w in which the image of x contains a \$ symbol and fulfills the conditions above, there exists an alignment of α and w with at most the same number of mismatches, in which the image of x does not contain a \$ symbol and, once more, fulfills the conditions B and C. Assume that in our original alignment x is mapped to a string u_x of length m such that $u_x[i] = \$$. Let $u_1, \dots, u_k$ be the factors of $w_1, \dots, w_k$, respectively, to which the first occurrences of the variable x are aligned. Consider the string u'_x which is obtained from u_x by simply replacing the \$ symbol on position i by $u_1[i]$. And then consider the alignment of α and w which is obtained from the original alignment by changing the image of x to u'_x instead of u_x. When compared to the original alignment, the new alignment has an additional mismatch caused by the occurrence of x aligned to $\$^m$, but at least one less mismatch caused by the alignments of the first k occurrences of x. Indeed, in the original alignment, the i^{th} position of u_x was a mismatch to the i^{th} position of any string $u_1, \dots, u_k$, but now at least the i^{th} positions of w_1 and u'_x coincide. This shows that our claim holds. A similar argument shows that for any alignment in which x is mapped to a string containing other letters than the input letters from the $\mathtt{CP}$-instance there exits an alignment in which x is mapped to a string containing only letters from the $\mathtt{CP}$-instance.
Hence, from now on we can assume that the factors $(\mathtt{a}^{\mathtt{M}}\mathtt{b}^{\mathtt{M}})^{\mathtt{M}}$ of g_j and f_j are aligned and that the image of x has length m and is over the input alphabet of $\mathtt{CP}$-instance.

Based on the observations A-D, we can show that the reduction has the desired properties. If the $\mathtt{CP}$-instance admits a solution $s, s_1, \dots, s_k$ which causes a number of mismatches less or equal to Δ, then we can produce an alignment of α to w as follows. We map x to s and, for i from 1 to k, we map x_i and y_i to the prefix of w_i occurring before s_i and, respectively, the suffix of w_i occurring after s_i. This leads to $\Delta + m$ mismatches between α and w, so the input $(w, \alpha, \Delta + m)$ of $\mathtt{HDMatch}_{\mathtt{1RepVar}}$ is accepted. Conversely, if we have an alignment of α and w with at most $\Delta + m$ mismatches, then we have an alignment with the same number of mismatches which fulfills the conditions summarized at the end of item D above. Hence, we can define s as the image of x in this alignment, and the strings $s_1, \dots, s_k$ as the factors of w aligned to the first k occurrences of x from α. Clearly, for i between 1 and k, s_i is a factor of w_i. As m mismatches of the alignment were caused by the alignment of the last x to $\$^m$, we get that $\sum_{i=1}^{k} d_{\mathtt{HAM}}(s, s_i) \leq \Delta$. Thus, the instance of $\mathtt{CP}$ is accepted.

This concludes the proof of the correctness of our reduction. As M is clearly of polynomial size w.r.t. the size of the $\mathtt{CP}$-instance, it follows that both w and α are of polynomial size $O(kM^2)$. Therefore, the instance of $\mathtt{MinHDMatch}_{\mathtt{1RepVar}}$ can be computed in polynomial time, and our entire reduction is done in polynomial time. Moreover, we have shown that the instance $(w, \alpha, \Delta + M)$ of $\mathtt{MinHDMatch}_{\mathtt{1RepVar}}$ is answered positively if and only if the original instance of $\mathtt{CP}$ is answered positively.

Finally, as the number of x blocks in α is $k+1$, where k is the number of input strings in the instance of $\mathtt{CP}$, and $\mathtt{CP}$ is $W[1]$-hard with respect to this parameter, it follows that $\mathtt{MinHDMatch}_{\mathtt{1RepVar}}$ is also $W[1]$-hard when the number of k-blocks in α is considered as parameter. This completes our proof. □

It is worth noting that the pattern α constructed in the reduction above is $k-1$-local (and not k-local): a witness marking sequence is $z_1 < y_2 < z_2 < y_3 < \ldots < z_{k-1} < y_k < x < y_1 < z_k$. Thus, $\mathtt{HDMatch}_{\mathtt{1RepVar}}$ is W[1]-hard w.r.t. locality of the input pattern as well. Also, it is easy to see that $\mathtt{scd}(\alpha) = 2$, and, by the results of [143], this shows that the treewidth of the pattern α, as defined in the same paper, is at most 3. Thus, even for classes of patterns with constant $\mathtt{scd}$, number or repeated variables, or treewidth, the problems $\mathtt{HDMatch}_P$ and $\mathtt{MinHDMatch}_P$ can become intractable.

In Theorem 3.14 we have shown that $\mathtt{MinHDMatch}_{\mathtt{1RepVar}}$ admits a polynomial time approximation scheme (for short, PTAS). We will show in the following that it does not admit an efficient PTAS (for short, EPTAS), unless $FPT = W[1]$. This means that there is no PTAS for $\mathtt{MinHDMatch}_{\mathtt{1RepVar}}$ such that the exponent of the polynomial in its running time is independent of the approximation ratio.

To show this, we consider an optimisation variant of the problem $\mathtt{CP}$, denoted $\mathtt{minCP}$. In this problem, for k strings $w_1, \ldots, w_k \in \Sigma^\ell$ of length ℓ and an integer $m \in \mathbb{N}$ with $m \leq \ell$, we are interested in the smallest non-negative integer Δ for which there exist strings s, of length m, and $s_1, \ldots, s_k$, factors of length m of each $w_1, \ldots, w_k$, respectively, such that $\sum_{i=1}^{k} d_{\mathtt{HAM}}(s_i, s) = \Delta$. In [29], it is shown that $\mathtt{minCP}$ has no EPTAS unless $FPT = W[1]$. We can use this result and the reduction from the Theorem 3.15 to show the following result.

Theorem 3.16. $\mathtt{MinHDMatch}_{\mathtt{1RepVar}}$ *has no EPTAS unless* $FPT = W[1]$.

Proof. Assume, for the sake of a contradiction, that $\mathtt{MinHDMatch}_{\mathtt{1RepVar}}$ has an EPTAS. That is, for an input word w and an $\mathtt{1RepVar}$-pattern α, there exists a polynomial time algorithm which returns as answer to $\mathtt{MinHDMatch}_{\mathtt{1RepVar}}$ a value $\delta' \leq (1+\epsilon)d_{\mathtt{HAM}}(\alpha, w)$, and the exponent of the polynomial in its running time is independent of ϵ.

An algorithm for `minCP` would first implement the reduction in Theorem 3.15 to obtain a word w and a pattern α. Then it uses the EPTAS for $\texttt{MinHDMatch}_{\texttt{1RepVar}}$ to approximate the distance between α and w with approximation ratio $(1 + \frac{\epsilon}{2m})$. Assuming that this EPTAS returns the value D, the answer returned by this algorithm for the `minCP` problem is $D - m$.

As explained in the proof of Theorem 3.15, it is easy to see that the distance between the word w and the pattern α constructed in the respective reduction is $m + \Delta$, if Δ is the answer to the instance of the `minCP` problem. Thus, the value D returned by the EPTAS for $\texttt{MinHDMatch}_{\texttt{1RepVar}}$ fulfils $m + \Delta \leq D \leq (1 + \frac{\epsilon}{2m})(m + \Delta)$. So, we have $\Delta \leq D - m \leq \frac{\epsilon}{2} + (1 + \frac{\epsilon}{2m})\Delta$. We get that $\Delta \leq D - m \leq (1 + \frac{\epsilon}{2m} + \frac{\epsilon}{2\Delta})\Delta \leq (1 + \epsilon)\Delta$. So, indeed, $D - m$ would be a $(1 + \epsilon)-$approximation of Δ.

Therefore, this would yield an EPTAS for `minCP`. This is a contradiction to the results reported in [29], where it was shown that such an EPTAS does not exist, unless $FPT = W[1]$. This concludes our proof. □

A conclusion of all the results as well as an outlook on future work is presented in Section 7.

CHAPTER 4

Matching Patterns with Variables under Edit Distance

This chapter is based on article [79] (see reference below) and the LaTeX code of this article was used to reproduce it here. The Introduction and Preliminaries of this paper are worked into Chapter 1 and Chapter 2. Further, the notations are adjusted to match the other chapters and provide a uniform notation across this thesis.

Reference: P. Gawrychowski, F. Manea, and S. Siemer. Matching Patterns with Variables Under Edit Distance. In D. Arroyuelo and B. Poblete, editors, *String Processing and Information Retrieval - 29th International Symposium, SPIRE 2022, Concepción, Chile, November 8-10, 2022, Proceedings*, volume 13617 of *Lecture Notes in Computer Science*, pages 275–289. Springer, 2022. `doi: 10.1007/978-3-031-20643-6\_20`

Description: This paper provided a natural follow-up to the setting from [78] presented in Chapter 3. Instead of only considering mismatches (single letter substitutions) between the target word and the pattern, we extended the setting to allow for insertions and deletions of letters. The amount of these operations (commonly known as the edit distance) to get from the target word to the input pattern (under any substitution) must be either bound by an integer in the input (decision variant) or has to be minimal in order to be a valid substitution (minimization variant). We obtained algorithmic lower and upper bounds for various classes of patterns.

Contribution: I was a main contributor to identifying the problem described in this paper, the design of the algorithms, and their write-up in the paper. Further, I presented this work at *SPIRE 2022* in Concepción (Chile) and in several workshops.

4.1 Overview

In this chapter, we study patterns that can be matched efficiently under edit distance: we allow insertion, deletions, substitutions between the word w and the image of α under a substitution h. More precisely, we consider two problems. In the decision problem $\mathtt{EDMatch}_P$ we are interested in deciding, for a given pattern α from a class P, a given word w, and a non-negative integer Δ whether there exists a variable-substitution h such that the word $h(\alpha)$ is at most Δ edit operations from the word w; in other words $d_{\mathtt{ED}}(h(\alpha), w) \leq \Delta$. Alternatively, we consider the corresponding minimisation problem $\mathtt{MinEDMatch}_P$ of computing $d_{\mathtt{ED}}(\alpha, w) = \min\{d_{\mathtt{ED}}(h(\alpha), w) \mid h \text{ is a substitution of the variables in } \alpha\}$.

With the definition of the edit distance (see Section 2.5) and the pattern matching problems for families of patterns $P \subseteq PAT$ (see Section 2.2) we can give the formal problem description. In the first problem, we allow for a certain distance Δ between the image $h(\alpha)$ of α under a substitution h and the target word w instead of searching for an exact matching. In the second problem, we are interested in finding the substitution h such that the edit distance between $h(\alpha)$ and the target word w is minimal, over all possible choices of h.

Approximate Matching Decision Problem for $\mathtt{EDMatch}_P$
Input: A pattern $\alpha \in P$, with $|\alpha| = m$, a word w, with $|w| = n$, an integer $\Delta \leq m$.
Question: Is there h with $d_{\mathtt{ED}}(\alpha, w) \leq \Delta$?

Approximate Matching Minimisation Problem for $\mathtt{MinEDMatch}_P$
Input: A pattern $\alpha \in P$, with $|\alpha| = m$, a word w, with $|w| = n$.
Question: Compute h that minimizes $d_{\mathtt{ED}}(\alpha, w)$.

Example 4.1. $\alpha = \mathtt{x}_1\mathtt{x}_1\mathtt{bab}\mathtt{x}_2\mathtt{x}_2$, $w = \mathtt{aaababbb}$, $\Delta = 1$. *In this instance we require either an insertion of an* a *in front of w or a deletion of an* a *in front of h*(α).

$$h(\mathtt{x}_1) = \mathtt{aa} \qquad h(\alpha) = \boldsymbol{a}\mathtt{aaababbb}$$
$$h(\mathtt{x}_2) = \mathtt{b} \qquad w = \mathtt{aaababbb}$$

Firstly, we consider the class of regular patterns, and show that $\mathtt{EDMatch}_{\mathtt{Reg}}$ and $\mathtt{MinEDMatch}_{\mathtt{Reg}}$ can be solved in $O(n\Delta)$ time (where, for $\mathtt{MinEDMatch}$, Δ is the computed result); a matching conditional lower bound follows from the literature [14]. This is particularly interesting because the problem of computing $d_{\mathtt{ED}}(\alpha, w)$ for $\alpha = w_0x_1w_1 \ldots x_kw_k$ can be seen as the problem of computing the minimal edit distance between any string in which $w_1, \ldots, w_k$ occur, without overlaps, in this exact order and the word w.

Secondly, we show that, unlike the case of matching under Hamming distance (Chapter 3), $\texttt{EDMatch}_P$ becomes $W[1]$-hard already for P being the class of unary patterns, with respect to the number of occurrences of the single variable. So, interestingly, the problem of matching patterns with variables under edit distance is computationally hard for all the classes (that we are aware of) of structurally restricted patterns with polynomial exact matching problem, as soon as at least one variable is allowed to occur an unbounded number of times.

To complement the results presented in this work, we note that, for the classes of patterns considered in [59, 51, 143, 78], which admit polynomial-time exact matching algorithms, one can straightforwardly adapt those algorithms to work in polynomial time in the case of matching under edit distance, when a constant upper bound k_1 on the number of occurrences of each variable exists. The complexity of these algorithms is usually $O(n^{f(k_1,k_2)})$, for a polynomial function f and for k_2 being a constant upper bound for the value of the structural parameter considered when defining these classes (locality, scope coincidence degree, treewidth, etc.). If no restriction is imposed on the structure of the pattern, `Match` (and, as such, the matching under both Hamming and edit distances) is NP-hard even if there are at most two occurrences of each variable [60].

As a first general remark, based on the Section 2.5 on edit distance, the following theorem follows.

Theorem 4.2. *$\texttt{EDMatch}_{PAT}$ and $\texttt{MinEDMatch}_{PAT}$ can be solved in $O(n^{2k_2+k_1})$ time, where k_1 is the maximum number of occurrences of any variable in the input pattern α and k_2 is the total number of occurrences of variables in α.*

Proof. We only give the proof for $\texttt{MinEDMatch}_{PAT}$.

Assume the input pattern is $\alpha = u_0 x_1 u_1 \dots x_{k_2} u_{k_2}$ from PAT_Σ, where x_i is a variable, for $i \in \{1, \dots, k_2\}$, and $w_i \in \Sigma^*$ a terminal word, for $i \in \{0, \dots, k_2\}$. Note that there might be the case that $x_i = x_j$ for some $i \neq j$, as there are no restrictions on the structure of the pattern α.

We make several observations.

Let h be a substitution of the variables from α, such that $h(x_i) = t_i$, for $i \in \{1, \dots, k_2\}$. Then, $h(\alpha) = u_0 t_1 u_1 \dots t_{k_2} u_{k_2}$. When computing the edit distance $d_{\mathtt{ED}}(h(\alpha), w)$, one obtains a factorization of $w = w_0 w'_1 w_1 \dots w'_{k_2} w_{k_2}$ such that the optimal sequence of edits transforming $h(\alpha)$ into w transforms u_i into w_i, for $i \in \{0, \dots, k_2\}$, and t'_i into w_i, for $i \in \{1, \dots, k_2\}$.

Now, let $V_x = \{i \in \{1, \dots, k_2\} \mid x_i = x\}$ and assume h is a substitution of the variables from α such that $d_{\mathtt{ED}}(h(\alpha), w)$ is minimal w.r.t. all possible substitutions of the variables of α. Moreover, let $h(x_i) = t_i$ for $i \in \{1, \dots, k_2\}$. As before, there exists a factorization of $w = w_0 w'_1 w_1 \dots w'_{k_2} w_{k_2}$ such that the optimal sequence of edits transforming $h(\alpha)$ into w transforms u_i into w_i, for $i \in \{0, \dots, k_2\}$, and t'_i into w_i, for $i \in \{1, \dots, k_2\}$. In this case, from the fact that h is optimal, it is immediate that $h(x) = s_x$ where s_x is the median string of $\{w'_i \mid i \in V_x\}$.

Based on these observations, we can use the following algorithm solving $\mathtt{MinEDMatch}_{PAT}$.

For each $x \in \mathtt{var}(\alpha)$, define $V_x = \{i \in \{1,\dots,k_2\} \mid x_i = x\}$. For each factorization f of $w = w_0 w'_1 w_1 \dots w'_{k_2} w_{k_2}$ and for each variable x: compute the median string s_x of $\{w'_i \mid i \in V_x\}$; define the substitution h_f which maps x to s_x for all x; compute the edit distance $d_{\mathtt{ED}}(h_f(\alpha), w)$. After considering each possible factorization f, return the substitution h_f for which $d_{\mathtt{ED}}(h_f(\alpha), w)$ is minimal.

In the above algorithm, to compute the median string of $\{w'_i \mid i \in V_x\}$, we use the algorithm of [147]. This algorithm runs in $O(\ell_x^{|V_x|})$, where $\ell_x = \max\{|w'_i| \mid i \in V_x\}$. Therefore, the running time of our algorithm can be upper bounded by $O(n^{2k_2} n^{k_1})$, so also by $O(n^{2k_2+k_1})$. □

As mentioned in the Introduction, the result of the previous Theorem can be improved if we consider the two problems for classes of patterns with restricted structure, where we obtain algorithms whose complexity depends on the structural parameter associated to that class, rather than the total number of occurrences of variables.

4.2 Matching Regular Patterns under Edit Distance

The first main result of our paper is about the class of regular patterns. A detailed discussion on the various pattern families can be found in Section 2.2. As a reminder, a pattern α over the terminal alphabet Σ is regular ($\alpha \in \mathtt{Reg}$) if $\alpha = w_0(\Pi_{i=1}^{k} x_i w_i)$ where, for $i \in \{1,\dots,k\}$, $w_i \in \Sigma^*$ and x_i is a variable, and $x_i \neq x_j$ for all $i \neq j$.

4.2.1 Efficient solutions for $\mathtt{EDMatch}_{\mathtt{Reg}}$ and $\mathtt{MinEDMatch}_{\mathtt{Reg}}$

We can show the following theorem.

Theorem 4.3. $\mathtt{EDMatch}_{\mathtt{Reg}}$ *can be solved in* $O(n\Delta)$ *time. For an accepted instance* w, α, Δ *of* $\mathtt{EDMatch}_{\mathtt{Reg}}$ *we also compute* $d_{\mathtt{ED}}(\alpha, w)$ *(which is at most* Δ*).*

Proof. **Preliminaries and setting.** We begin with an observation. For $\alpha = w_0(\Pi_{i=1}^{k} x_i w_i)$, we can assume w.l.o.g. that $w_i \in \Sigma^+$ for all $i \leq k$ as otherwise we would have neighboring variables that could be replaced by a single variable; thus, $k \leq |\mathtt{term}(\alpha)|$. For example, the pattern $\alpha = \mathtt{ab}x_1\mathtt{ab}x_2x_3\mathtt{baab}$, with $\mathtt{var}(\alpha) = \{x_1, x_2, x_3\}$ is in $\mathtt{Reg}$, and is equivalent (in terms of the words that it matches) to the pattern $\alpha' = \mathtt{ab}x_1\mathtt{ab}x_2\mathtt{baab}$, with the additional information that any substitution of x_2 in a match of α' can be split at any position to give substitutions for x_2 and x_3 in a match of α. To avoid some corner cases, we can assume w.l.o.g. that α and w start with the same terminal symbol (this can be achieved by adding a fresh letter \$ in front of both α and w). While not fundamental, these simplifications make the exposure of the following algorithm easier to follow.

Before starting the presentation of the algorithm, we note that a solution for $\texttt{EDMatch}_{\texttt{Reg}}$ with distance $\Delta = 0$ is a solution to $\texttt{Match}_{\texttt{Reg}}$ and can be solved in $O(n)$ by a greedy approach (Section 2.3). Further, the special case $x_1 w_1 x_2$ can be solved by an algorithm due to Landau and Vishkin [115] in $O(n\Delta)$ time. In the following, we are going to achieve the same complexity for the general case of $\texttt{EDMatch}_{\texttt{Reg}}$ by extending the ideas of this algorithm to accommodate the existence of an unbounded number of pairwise-distinct variables.

One important idea which we use in the context of computing the edit distance between an arbitrary regular pattern and a word is to interpret each regular variable as an arbitrary amount of "free" insertions on that position, where "free" means that they will not be counted as part of the actual distance (in other words, they do not increase this distance). Indeed, we can see that the factor which substitutes a variable should always be equal to the factor to which it is aligned (after all the edits are performed) from the target word, hence does not add anything to the overall distance (and, therefore, it is "free"). As such, this factor can be seen as being obtained via an arbitrary amount of letter insertions. Now, using this observation, it is easier to design an $O(nm)$-time algorithm which computes the edit distance between the terminal words $\beta = \texttt{term}(\alpha)$ (instead of the pattern α) and w with the additional property that, for the positions $F_g = |(\Pi_{i=0}^{g} |w_i|)|$ for $0 \le g \le k-1$, we have that the insertions done between positions $\beta[F_g]$ and $\beta[F_g + 1]$ when editing β to obtain w do not count towards the total edit distance between β and w. For simplicity, we denote the set $\{F_g | 0 \le g \le k-1\}$ by F, we set $F_k = +\infty$, and note that $|\beta| = m - k$ (so $\beta \in \Theta(m)$).

The description of our algorithm is done in two phases. We first explain how $\texttt{EDMatch}_{\texttt{Reg}}$ can be solved by dynamic programming in $O(nm)$ time. Then, we refine this approach to an algorithm which fulfills the statement of the theorem.

When presenting our algorithms, we refer to an alignment of prefixes $\beta[1:j]$ of β and $w[1:\ell]$ of w, which simply means editing $\beta[1:j]$ to obtain $w[1:\ell]$.

First phase: a classical dynamic programming solution. We define the $(|\beta|+1) \times (n+1)$ matrix $D[\cdot][\cdot]$, where $D[j][\ell]$ is the edit distance between the prefixes $\beta[1:j]$, with $0 \le j \le |\beta|$, and $w[1:\ell]$, with $0 \le \ell \le n$, with the additional important property that the insertions done between positions $\beta[F_g]$ and $\beta[F_g + 1]$, for $F_g \le j$, are not counted in this distance (they correspond to variables in the pattern α). As soon as this matrix is computed, we can retrieve the edit distance between α and w from the element $D[m-k][n]$. Clearly, now the instance (α, w, Δ) of $\texttt{EDMatch}_{\texttt{Reg}}$ is answered positively if and only if $D[m-k][n] \le \Delta$. So, let us focus on an algorithm computing this matrix.

The elements of the matrix $D[\cdot][\cdot]$ can be computed by dynamic programming. The base cases are $D[j][0] = j$, for all $j \le \beta$ and $D[0][\ell] = \ell$. In the case of computing $D[0][\ell]$, we simply insert all the letters of $w[1:\ell]$ in $\beta[1:0] = \varepsilon$, while in the case of $D[j][0]$ we are deleting all letters from $\beta[1:j]$ (and, if we refer to the edits in α, where we also have variables, then we substitute all the variables of the prefix of α which corresponds to $\beta[1:j]$ by the empty word, as well).

The rest of the elements of $D[\cdot][\cdot]$ are now computed according to two cases.

Firstly, we consider the computation of $D[j][\ell]$ for $j \notin F$. In this case, we cannot use the aforementioned free insertions, so the element $D[j][\ell]$ is computed as in the case of computing the usual edit distance between two strings.

$$D[j][\ell] = min \begin{cases} D[j-1][\ell]+1, & \beta[j] \text{ is deleted in the alignment of } \beta[1:j] \\ & \text{to } w[1:\ell]; \\ D[j][\ell-1]+1, & w[\ell] \text{ is inserted after position } j \text{ of } \beta \text{ in} \\ & \text{the alignment of } \beta[1:j] \text{ to } w[1:\ell]; \\ D[j-1][\ell-1]+1, & \beta[j] \text{ is substituted by } w[j] \text{ in} \\ & \text{the alignment of } \beta[1:j] \text{ to } w[1:\ell]; \\ D[j-1][\ell-1], & \beta[j] \text{ is left unchanged in the alignment} \\ & \text{of } \beta[1:j] \text{ to } w[1:\ell] \text{ because } \beta[j]=w[\ell]. \end{cases}$$

The more interesting case is when $j \in F$ and we can use free insertions. Naturally, our starting point is still represented by the four possible cases based on which we computed $D[j][\ell]$ when $j \notin F$. However, the case corresponding to the insertion of $w[\ell]$ to extend an alignment of $\beta[1:j]$ and $w[1:\ell-1]$ to an alignment of $\beta[1:j]$ and $w[1:\ell]$ can now be obtained by a free insertion, instead of an insertion of cost 1. This brings us to the main difference between the two cases. In this case, an alignment between $\beta[1:j]$ and $w[1:\ell]$ can be obtained as follows. We first obtain an alignment of $\beta[1:j]$ to some prefix $w[1:\ell-k]$ of w and then use free insertions to append $w[\ell-k+1:\ell]$ to the edited pattern, and, as such, obtain $w[1:\ell]$. But, this also means that we first obtain an alignment of $\beta[1:j]$ to some prefix $w[1:\ell-k]$ of w and then use free insertions to append $w[\ell-k+1:\ell-1]$ to the edited pattern, and, as such, obtain an alignment of the pattern to $w[1:\ell-1]$, and then insert (again, without counting this towards the edit distance) $w[\ell]$ to obtain $w[\ell-k+1:\ell]$. Thus, in this case, an alignment between $\beta[1:j]$ and $w[1:\ell]$ which uses free insertions corresponding to the position $j \in F$ is obtained from an alignment between $\beta[1:j]$ and $w[1:\ell-1]$ followed by an additional free insertion. We obtain, as such, the following recurrence relation for $D[j][\ell]$, when $j \in F$:

$$D[j][\ell] = min \begin{cases} D[j-1][\ell]+1, & \beta[j] \text{ is deleted;} \\ D[j-1][\ell-1]+1, & \beta[j] \text{ is substituted by } w[\ell], \text{ if } \beta[j] \neq w[\ell]; \\ D[j-1][\ell-1], & \beta[j] \text{ is left unchanged, if } \beta[j]=w[\ell]; \\ D[j][\ell-1], & w[\ell] \text{ is inserted after position } j, \text{ for free.} \end{cases}$$

Using the two recurrence relation above, we can compute the elements of the matrix D by dynamic programming (for j from 0 to $m-k$, for ℓ from 0 to n) in $O(nm)$ time. Moreover, by tracing back the computation of $D[m-k][n]$, we obtain a path consisting of elements of the matrix, leading from $D[0][0]$ to $D[m-k][n]$, which encodes the edits needed to transform β into w. An edge between $D[j-1][\ell]$ and $D[j][\ell]$ corresponds to the deletion of $\beta[j]$; and edge between $D[j-1][\ell-1]$ and $D[j][\ell]$ corresponds to a substitution of $\beta[j]$ by $w[\ell]$, or to the case where $\beta[j]$ and $w[\ell]$ are left unchanged, and will be aligned in the end. Moreover, an edge between $D[j][\ell-1]$ and $D[j][\ell]$ corresponds to an insertion of $w[\ell]$ after position j in β; this can be a free insertion too (and part of the image of a variable of α), but only when $j \in F$.

A listing of an algorithm computing $D[\cdot][\cdot]$ is given in Figure 4.1.

```
Input: $w, \alpha$
Output: minimal edit distance between $\alpha$ and $w$ in $O(nm)$
compute $\beta$ and the set $F$;
for $j \leftarrow [0$ to $|\beta|]$ do
    $D[j][0] \leftarrow j$;
end
for $\ell \leftarrow [0$ to $n]$ do
    $D[0][\ell] \leftarrow \ell$;
end
$g \leftarrow 0$
for $j \leftarrow [0$ to $|\beta|]$ do
    if $j = F_g$ then
        for $\ell \leftarrow [0$ to $n]$ do
            $D[j][\ell] \leftarrow min \begin{cases} D[j][\ell-1], & \text{free insertion} \\ D[j-1][\ell]+1, & \text{deletion} \\ D[j-1][\ell-1]+1, & \text{substitution} \\ D[j-1][\ell-1], & \text{if } w[\ell]=\beta[j] \end{cases}$
        end
        $g \leftarrow g+1$
    else
        for $\ell \leftarrow [0$ to $n]$ do
            $D[j][\ell] \leftarrow min \begin{cases} D[j][\ell-1]+1, & \text{insertion} \\ D[j-1][\ell]+1, & \text{deletion} \\ D[j-1][\ell-1]+1, & \text{substitution} \\ D[j-1][\ell-1], & \text{if } w[\ell]=\beta[j] \end{cases}$
        end
    end
end
return $D[|\beta|][n]$
```

Figure 4.1: Algorithm to compute $D[\cdot][\cdot]$ in $O(nm)$ time.

This concludes the first phase of our proof.

Second phase: a succinct representation and more efficient computation of the dynamic programming table. In the second phase of our proof, we will focus on how to solve $\texttt{EDMatch}_{\texttt{Reg}}$ more efficiently. The idea is to avoid computing all the elements of the matrix $D[\cdot][\cdot]$, and compute, instead, only the relevant elements of this matrix, following the ideas of the algorithm by Landau and Vishkin [115]. The main difference between the setting of that algorithm (which can be directly used to compute the edit distance between two terminal words or between a word w and a pattern α of the form xuy, xu, or uy, where x and y are variables and u is a terminal word) and ours is that, in our case, the diagonals of the matrix $D[\cdot][\cdot]$ are not non-decreasing (when traversed in increasing order of the rows intersected by the respective diagonal), as we now also have free insertions which may occur at various positions in β (not only at the beginning and end). This is a significant complication, which we will address next.

The main idea of the optimization done in this second phase is that we could actually compute and represent the matrix $D[\cdot][\cdot]$ more succinctly, by only computing and keeping track of at most Δ relevant elements on each diagonal of this matrix, where relevant means that we cannot explicitly rule out the existence of a path leading from $D[0][0]$ to $D[m-k][n]$ which goes through that element.

For the clarity of exposure, we recall that the diagonal d of the matrix $D[\cdot][\cdot]$ is defined as the array of elements $D[j][\ell]$ where $\ell - j = d$ (ordered in increasing order w.r.t. the first component j), where $-|\beta|+1 \leq d \leq n$. Very importantly, for a diagonal d, we have that if $D[j][j+d] \leq D[j+1][j+1+d]$ then $D[j+1][j+1+d] - D[j][j+d] \leq 1$; however, it might also be the case that $D[j][j+d] > D[j+1][j+1+d]$, when $D[j+1][j+1+d]$ is obtained from $D[j+1][j+d]$ by a free insertion.

Analysis of the diagonals, definition of $M_d[\delta]$ and its usage. Now, for each diagonal d, with $-|\beta|+1 \leq d \leq n$, and $\delta \leq \Delta$, we define $M_d[\delta] = \max\{j \mid D[j][j+d] = \delta$, and $D[j'][j'+d] > \delta$ for all $j' > j\}$ (by convention, $M_d[\delta] = -\infty$, if $\{j \mid D[j][j+d] = \delta$, and $D[j'][j'+d] > \delta$ for all $j' > j\} = \emptyset$). That is, $M_d[\delta]$ is the greatest row where we find the value δ on the diagonal d and, moreover, all the elements appearing on greater rows on that diagonal are strictly greater than δ (or $M_d[\delta] = -\infty$ if such a row does not exist).

Note that if a value δ appears on diagonal d and there exists some j' such that $D[j][j+d] \geq \delta$ for all $j \geq j'$, then, due to the only relations which may occur between two consecutive elements of d, we have that $M_d[\delta] \neq -\infty$. In particular, if a value δ appears on diagonal d then $M_d[\delta] \neq -\infty$ if and only if $D[|\beta|][|\beta|+d] \geq \delta$. Consequently, if there exists $k > 0$ such that $M_d[\delta-k] = |\beta|$ then $M_d[\delta] = -\infty$.

In general, all values $M_d[\delta]$ which are equal to $-\infty$ are not relevant to our computation. To understand which other values $M_d[\delta]$ are not relevant for our algorithm, we note that if there exist some $k > 0$ and $s \geq 0$ such that $M_{d+s}[\delta-k] = |\beta|$ then it is not needed to compute $M_{d-g}[\delta+h]$, for any $g, h \geq 0$, at all, as any path going from $D[0][0]$ to $D[|\beta|][n]$, which corresponds to an optimal sequence of edits, does not go through $D[M_{d-g}[\delta+h]][M_d[\delta+h]+d]$. If $s = 0$, then it is already clear that $M_d[\delta] = -\infty$, and we do not need to compute it. If $s \geq 1$, it is enough to show our claim for $h = 0$ and $g = 0$. Indeed, assume that the optimal sequence of edits transforming β into w corresponds to a path from $D[0][0]$ to $D[|\beta|][n]$ going through $D[M_d[\delta]][M_d[\delta] + d]$. By the fact that $M_d[\delta]$ is the largest j for which $D[j][j + d] \leq \delta$, we get that this path would have to intersect, after going through $D[M_d[\delta]][M_d[\delta]+d]$, the path from $D[0][0]$ to $D[M_{d+s}[\delta-k]][M_{d+s}[\delta-k]+d+s] = D[|\beta|][|\beta|+d+s]$ (which goes only through elements $\leq \delta - k$). As $k > 0$, this is a contradiction, as the path from $D[0][0]$ to $D[|\beta|][n]$ going through $D[M_d[\delta]][M_d[\delta] + d]$ goes only through elements $\geq \delta$ after going through $D[M_d[\delta]][M_d[\delta] + d]$. So, $M_d[\delta]$ is not relevant if there exist $k > 0$ and $s > 0$ such that $M_{d+s}[\delta - k] = m$.

Once all relevant values $M_d[\delta]$ are computed, for d diagonal and $\delta \leq \Delta$, we simply have to check if $M_{n-|\beta|}[\delta] = |\beta|$ (i.e., $D[|\beta|][n] = \delta$) for some $\delta \leq \Delta$. So, we can focus, from now on, on how to compute the relevant elements $M_d[\delta]$ efficiently. In particular, all these elements are not equal to $-\infty$.

Towards an algorithm: understanding the relations between elements on consecutive diagonals. Let us now understand under which conditions $D[j][\ell] = \delta$ holds, as this is useful to compute $M_d[\delta]$. In general, this means that there exists a path leading from $D[0][0]$ to $D[j][\ell]$ consisting only in elements with value $\leq \delta$, and which ends with a series of edges belonging to the diagonal $d = \ell - j$, that correspond to substitutions or to letters being left unchanged. In particular, if all the edges connecting $D[j'][j'+d]$ and $D[j][\ell]$ on this path correspond to unchanged letters, then $\beta[j' : j]$ is a common prefix of $\beta[j' : |\beta|]$ and $w[j'+d : n]$. Looking more into details, there are several cases when $D[j][\ell] = \delta$.

If $j \notin F$ and $\beta[j] \neq w[\ell]$, then $D[j-1][\ell-1] \geq \delta - 1$ and $D[j-1][\ell] \geq \delta - 1$ and $D[j][\ell-1] \geq \delta - 1$ and at least one of the previous inequalities is an equality (i.e., one of the following must hold: $D[j][\ell - 1] = \delta - 1$ or $D[j - 1][\ell - 1] = \delta - 1$ or $D[j - 1][\ell] = \delta - 1$). If $j \notin F$ and $\beta[j] = w[\ell]$, then $D[j - 1][\ell - 1] \geq \delta$ and $D[j - 1][\ell] \geq \delta - 1$ and $D[j][\ell - 1] \geq \delta - 1$ and at least one of the previous inequalities is an equality.

If $j \in F$ and $\beta[j] \neq w[\ell]$, then $D[j-1][\ell-1] \geq \delta-1$ and $D[j-1][\ell] \geq \delta-1$ and $D[j][\ell-1] \geq \delta$ and at least one of the previous inequalities is an equality. If $j \in F$ and $\beta[j] = w[\ell]$ then $D[j-1][\ell-1] \geq \delta$ and $D[j-1][\ell] \geq \delta-1$ and $D[j][\ell-1] \geq \delta$ and at least one of the previous inequalities is an equality.

Moving forward, assume now that $M_d[\delta] = j \neq -\infty$. This means that $D[j][\ell] = \delta$, and $D[j''][j'' + d] > \delta$ for all $j'' > j$. By the observations above, there exists $j' \leq j$ such that $D[j'][j' + d] = \delta$ and the longest common prefix of $\beta[j' : |\beta|]$ and $w[j' + d : n]$ has length $j - j' + 1$, i.e., it equals $\beta[j' : j]$. The last part of this statement means that once we have aligned $\beta[1 : j']$ to $w[1 : j' + d]$, we can extend this alignment to an alignment of $\beta[1 : j]$ to $w[1 : j + d]$ by simply leaving the symbols of $\beta[j' + 1 : j]$ unchanged.

Let us see now what this means for the elements of diagonals d, $d + 1$, and $d - 1$.

Firstly, we consider the diagonal d. Here we have that $j' \geq M_d[\delta - 1] + 1$. Note that if $\delta - 1$ appears on diagonal d then $M_d[\delta - 1] \neq -\infty$.

Secondly, we consider the diagonal $d + 1$. Here, for all rows ℓ with $j' \leq \ell \leq j$, we have that $D[\ell - 1][\ell + d] \geq \delta - 1$ and $D[j'' - 1][j'' + d] > \delta - 1$, for all j'' with $|\beta| \geq j'' > j$. Therefore, if $\delta - 1$ appears on diagonal $d + 1$, either $D[m][m + d + 1] \leq d - 1$ or $M_{d+1}[\delta - 1] \neq -\infty$ and $M_d[\delta - 1] + 1 \leq j$.

Finally, we consider the diagonal $d - 1$. Here, for all rows ℓ with $j' \leq \ell \leq j$, we have that $D[\ell][\ell + d - 1] \geq \delta - 1$ and $D[j''][j'' + d - 1] \geq \delta$, for all j'' with $m \geq j'' > j$. Thus, either all elements on the diagonal $d - 1$ are $\geq \delta$, or $\delta - 1$ occurs on diagonal $d - 1$ and $M_{d-1}[\delta - 1] \neq -\infty$. In the second case, when $M_{d-1}[\delta - 1] \neq -\infty$, we have that $j \geq M_{d-1}[\delta - 1]$ as, otherwise, we would have that $D[M_{d-1}[\delta - 1]][M_{d-1}[\delta - 1] + d] \leq \delta$ and $M_{d-1}[\delta - 1] > j$, a contradiction.

Still on diagonal $d - 1$, if δ occurs on it, then $M_d[\delta] \neq -\infty$ holds. So, for $g \leq k - 1$ with $F_g \leq M_{d-1}[\delta] < F_{g+1}$, we have that $F_g \leq M_d[\delta]$. Indeed, otherwise we would have two possibilities. If the path connecting $D[0][0]$ to $D[M_{d-1}[\delta]][M_{d-1}[\delta] + d - 1]$ via elements $\leq d$ intersects row F_g on $D[F_g][F_g + d']$ for some $d' \leq d$, then $D[F_g][F_g + d] \leq D[F_g][F_g + d'] \leq \delta$ and $F_g > j$, a contradiction. If the path connecting $D[0][0]$ to $D[M_{d-1}[\delta]][M_{d-1}[\delta] + d - 1]$ via elements $\leq d$ intersects row F_g on $D[F_g][F_g + d']$ for some $d' > d$, then the respective path will also intersect diagonal d on a row $> j$ before reaching $M_{d-1}[\delta]$, a contradiction with the fact that j is the last row on diagonal d where we have an element $\leq \delta$.

So, for $M_d[\delta]$ to be relevant, we must have $D[|\beta|][|\beta| + d + 1] \geq \delta$ (so there exists no $k > 0$ such that $M_{d+1}[\delta - k] = |\beta|$). In this case, if $M_d[\delta] = j$, then the following holds. The path (via elements $\leq d$) from $D[0][0]$ to $D[j][j + d]$ goes through an element $D[g][g + d'] = \delta - 1$. If the last such element on the respective path is on diagonal d, then it must be $M_d[\delta - 1]$. If it is on diagonal $d - 1$, then either $g = M_{d-1}[\delta - 1]$ (and then the path moves on diagonal d via an edge corresponding to an insertion) or $g < M_{d-1}[\delta - 1]$ (and then the path moves on diagonal d via an edge corresponding to an insertion); in this second case, we could replace the considered path by a path connecting $D[0][0]$ to $D[M_{d-1}[\delta - 1]][M_{d-1}[\delta - 1] + d - 1]$ (via elements $\leq \delta - 1$), which then moves on diagonal d via an edge corresponding to an insertion, and continues along that diagonal (with edges corresponding to letters left unchanged). If $D[g][g + d']$ is on diagonal $d + 1$ (i.e., $d' = d + 1$) then, just like in the previous case, we can simply consider the path connecting $D[0][0]$ to $D[M_{d+1}[\delta - 1]][M_{d+1}[\delta - 1] + d + 1]$ (via elements $\leq \delta - 1$), which then moves on diagonal d via an edge corresponding to a deletion, and then continues along diagonal d (with edges corresponding to letters left unchanged). If $D[g][g + d']$ is on none of the diagonals $d - 1, d, d + 1$ then we reach diagonal d by edges corresponding to free insertions from some diagonal $d'' < d$. The respective path also intersects diagonal $d - 1$ (when coming from d'' to d by free insertions), so diagonal $d - 1$ contains δ and $M_{d-1}[\delta] \neq \infty$, and we might simply consider as path between $D[0][0]$ and $D[j][j + d]$ the path reaching diagonal $d - 1$ on position $D[F_g][F_g + d - 1]$ (via elements $\leq \delta$), where

$F_g \leq M_{d-1}[\delta] < F_{g+1}$, which then moves on diagonal d by an edge corresponding to a free insertion, and then continues along d (with edges corresponding to letters left unchanged, as F_g is greater or equal to the row where the initial path intersected diagonal d). This analysis covers all possible cases.

Computing $M_d[\delta]$. Therefore, if $M_d[\delta]$ is relevant (and, as such, $M_d[\delta] \neq -\infty$), then $M_d[\delta]$ can be computed as follows. Let g be such that $F_g \leq M_{d-1}[\delta] < F_{g+1}$ (and $g = -1$ and $F_g = -\infty$ if $M_{d-1}[\delta] = -\infty$). Let $H = \max\{M_{d-1}[\delta - 1], F_g, M_d[\delta - 1] + 1, M_{d+1}[\delta - 1] + 1\}$ (as explained, in the case we are discussing, at least one of these values is not $-\infty$). Then we have that $j \geq H$ and the longest common prefix of $\beta[H + 1 : |\beta|]$ and $w[H + d + 1 : n]$ is exactly $\beta[H + 1 : j]$ (or we could increase j). So, to compute $j = M_d[\delta]$, we compute H and then we compute the longest common prefix $\beta[H + 1 : j]$ of $\beta[H + 1 : |\beta|]$ and $w[H + d + 1 : n]$.

In general, $M_d[\delta]$ is not relevant either because there exists some $s \geq 0$ and $\delta' < \delta$ such that $M_{d+s}[\delta'] = |\beta|$ or because all elements of diagonal d are strictly greater than δ. In the second case, we note that all values $M_{d-1}[\delta-1]$, F_g, $M_d[\delta-1]$, and $M_{d+1}[\delta-1]$ must be $-\infty$ (as otherwise the diagonal d would contain an element equal to δ), so our computation of $M_d[\delta]$ returns $-\infty$ (which is correct).

Now, based on these observations, we can see a way to compute the relevant values $M_d[\delta]$, for $-|\beta| \leq d \leq n$ and $\delta \leq \Delta$ (without computing the matrix D).

We first construct the word β and longest common prefix data structures for the word βw, allowing us to compute $\mathsf{LCP}(\beta[h : |\beta|], w[h + d : n])$, the length of the longest common prefix of $\beta[h : |\beta|]$ and $w[h + d : n]$ for all h and d.

Then, we will compute the values of $M_d[0]$ for all diagonals d. Basically, we need to identify, if it exists, a path from $D[0][0]$ to $D[M_d[0]][M_d[0] + d]$ which consists only of edges corresponding to letters left unchanged, or to free insertions. By an analysis similar to the one done above, we can easily show that $M_0[0]$ is $\mathsf{LCP}(\beta[1 : |\beta|], w[1 : n])$ (which is ≥ 1, by our assumptions). Further, $M[d][0] = -\infty$ for $d < 0$ and, for $d \geq 0$, $M_d[0] = F_g + \mathsf{LCP}(\beta[F_g + 1 : |\beta|], w[F_g + 1 + d : n])$, where $F_g \in F$ is such that $F_g \leq M_{d-1}[0] < F_{g+1}$ ($M_d[0] = -\infty$ if such an element F_g does not exist).

Further, for δ from 1 to Δ we compute all the values $M_d[\delta]$, in order for d from $-|\beta| + 1$ to n. We first compute the largest diagonal d' such that $M_{d'}[\delta - k] = |\beta|$, for some $k > 0$. We will only compute $M_d[\delta]$, for d from $d' + 1$ to n. For each such diagonal d, we compute g such that $F_g \leq M_{d-1}[\delta] < F_{g+1}$ and $H = \max\{M_{d-1}[\delta - 1], F_g, M_d[\delta - 1] + 1, M_{d+1}[\delta - 1] + 1\}$. Then we set $M_d[\delta]$ to be $H + \mathsf{LCP}(\beta[H + 1 : |\beta|], w[H + d + 1 : n]) - 1$.

Conclusions. This algorithm, which computes all relevant values $M_d[\delta]$,can be implemented in $O((n + m)\Delta)$ time. We first use a linear time algorithm for the computation of the longest common prefix data structures for β and w (see Section 2.7). Secondly, we use an auxiliary array G of size $|\beta|+1$, which stores for each positive integer $i \leq \beta$ the value $G[i] = \max\{g \mid F_g \leq i\}$, and can be computed in linear time. This allows us to efficiently retrieve the values F_g. Finally, while computing the values $M_d[\delta]$, for d and δ, we can maintain the value d' of the greatest diagonal such that there exist k with $M_{d'}[\delta - k] = |\beta|$: when we are done with computing all the values $M_d[\delta - 1]$, for all d, we simply check if we need to update d' because we might have found some $d'' > d'$ for which $M_{d''}[\delta - 1] = |\beta|$.

Computing $M_d[\delta]$.

The algorithm for computing the relevant values $M_d[\Delta]$ and how these are used to solve Problem $\texttt{EDMatch}_{\texttt{Reg}}$ is given in Figure 4.2.

```
**Input:** $w, \alpha$
**Output:** minimal edit distance between $\alpha$ and $w$ in $O((n+m)\Delta)$
construct $\beta$;
construct $F$;
init $g \leftarrow -1$;
construct $LCP_{\beta,w}$;
for $d \leftarrow [-|\beta|$ to $0]$ do
    $M_d[0] \leftarrow -\infty$;
end
$M_0[0] \leftarrow LCP(\beta[1:|\beta|], w[1:n])$;
compute $g$ such that $F_g \leq M_0[0] < F_{g+1}$ ($g \leftarrow -1$ if $F_0 > M_0[d]$);
if $g = -1$ then
    for $d \leftarrow [1$ to $n]$ do
        $M_d[0] \leftarrow -\infty$;
    end
else
    for $d \leftarrow [1$ to $n]$ do
        $M_d[0] \leftarrow F_g + \mathsf{LCP}(\beta[F_g + 1 : |\beta|], w[F_g + 1 + d : n])$;
        update $g$ such that $F_g \leq M_d[0] < F_{g+1}$;
    end
end
$g \leftarrow -1$;
compute $d' = \min\{d \leq n \mid M_d[0] = m\}$;
for $\delta \leftarrow [1$ to $\Delta]$ do
    for $d \leftarrow [d'+1$ to $n]$ do
        update $g$ such that $F_g \leq M_{d-1}[\delta] < F_{g+1}$;
        $H \leftarrow max \begin{cases} M_{d-1}[\delta-1], & \text{diagonal below} \\ F_g, & \text{for } F_g \text{ with } F_g \leq M_{d-1}[\delta] < F_{g+1} \\ M[d][\delta-1]+1, & \text{same diagonal} \\ M[d+1][\delta-1]+1, & \text{diagonal above} \end{cases}$ ;
        $M[d][\delta] \leftarrow H + \mathsf{LCP}(\beta[H+1 : |\beta|], w[H+d+1 : n]) - 1$;
        if $(d = n - |\beta|) \wedge (M[d][\delta] = |\beta|)$ then
            return $\delta$;
        end
    end
    maintain $d' = \min\{d'' \leq |\beta| \mid M_{d''}[\delta - s] = |\beta|$ for some $s \geq 0\}$;
end
return No solution with $\Delta$ edit operations.;
```

Figure 4.2: Algorithm to compute the relevant values of M in $O((n+m)\Delta)$ time.

□

The following result now follows by exponential search.

Theorem 4.4. `MinEDMatch`$_{\texttt{Reg}}$ *can be solved in $O(n\Phi)$ time, where $\Phi = d_{\mathtt{ED}}(\alpha, w)$.*

Proof. We use the algorithm of Theorem 4.3 for $\Delta = 2^i$, for increasing values of i starting with 1 and repeating until the algorithm returns a positive answer and computes $\Phi = d_{\mathtt{ED}}(\alpha, w)$. The algorithm is clearly correct. Moreover, the value of i which was considered last is such that $2^{i-1} < \Phi \leq 2^i$. So $i = \lceil \log_2 \Phi \rceil$, and the total complexity of our algorithm is $O(n \sum_{i=1}^{\lceil \log_2 \Phi \rceil} 2^i) = O(n\Phi)$. □

4.2.2 Lower Bounds for `EDMatch`$_{\texttt{Reg}}$ and `MinEDMatch`$_{\texttt{Reg}}$

The upper bounds reported in Theorems 4.3 and 4.4 are complemented by the following lower bound, known from the literature [14]. Firstly, we recall the `OV` problem.

Orthogonal Vectors (for short, `OV`)	
Input:	Two sets U, V consisting each of n vectors from $\{0,1\}^d$, where $d \in \omega(\log n)$.
Question:	Do vectors $u \in U, v \in V$ exist, such that u and v are orthogonal, i.e., for all $1 \leq k \leq d$, $v[k]u[k] = 0$ holds?

It is clear that, for input sets U and V as in the above definition, one can solve `OV` trivially in $O(n^2 d)$ time. The following conditional lower bound is known.

Lemma 4.5 (`OV`-Conjecture)**.** *OVC `OV` can not be solved in $O(n^{2-\epsilon} d^c)$ for any $\epsilon > 0$ and constant c, unless the Strong Exponential Time Hypothesis (SETH) fails.*

See [32, 166] and the references therein for a detailed discussion regarding conditional lower bounds related to OV. In this context, the following result is an immediate consequence of [14, Thm. 3].

Theorem 4.6. `EDMatch`$_{\texttt{Reg}}$ *can not be solved in time $O(|w|^h \Delta^g)$ (or $O(|w|^h |\alpha|^g)$) where $h + g = 2 - \epsilon$ with $\epsilon > 0$, unless the Orthogonal Vectors Conjecture fails.*

It is worth noting that the lower bound from Theorem 4.6 already holds for very restricted regular patterns, i.e., for $\alpha = xuy$, where u is a string of terminals and x and y are variables. Interestingly, a similar lower bound (for such restricted patterns) does not hold in the case of the Hamming distance, covered in [78].

4.3 Matching Patterns with Repeated Variables

Our second main result addresses another class of restricted patterns (Section 2.2). To this end, we consider the class of unary (or one-variable) patterns `1Var`, which is defined as follows: $\alpha \in$ `1Var` if there exists $x \in X$ such that `var`$(\alpha) = \{x\}$. An example of unary pattern is $\alpha_1 = \mathtt{ab}x\mathtt{ab}xx\mathtt{baab}$.

4.3.1 Lower Bounds for Unary Pattern

Now we are going to show the following theorem.

Theorem 4.7. $\texttt{EDMatch}_{\texttt{1Var}}$ *is* $W[1]$*-hard w.r.t. the number of occurrences of the single variable* x *of the input pattern* α.

Before starting the proof of Theorem 4.7 we need the following technical lemma.

Lemma 4.8. *Let* \$ *and* # *be two letters and let* S, g, *and* ℓ *be integers. If* $g \geq 0$, $2g \leq S$, $\frac{S}{2} \leq \ell - g$, *and* $\ell \leq S$ *then:*

1. $d_{\texttt{ED}}(\$^g(\$^S\#^S)^{S-1}\$^S\#^\ell, (\$^S\#^S)^S) = g + (S - \ell)$;
2. $d_{\texttt{ED}}(\$^\ell\#^S(\$^S\#^S)^{S-1}\#^g, (\$^S\#^S)^S) = g + (S - \ell)$.

Proof. We only show the first claim, as the second follows identically (as it is symmetrical).

Firstly, it is clear that $g + (S - \ell)$ edits suffice to transform $\$^g(\$^S\#^S)^{S-1}\$^S\#^\ell$ into $(\$^S\#^S)^S$.

Now, we will show that we cannot transform $\$^g(\$^S\#^S)^{S-1}\$^S\#^\ell$ into $(\$^S\#^S)^S$ with fewer than $S - \ell + g$ edits.

Note that from $\frac{S}{2} \leq \ell - g$ we get $\ell \geq g + (S - \ell)$. So, the suffix $\#^S$ of $(\$^S\#^S)^S$ must be obtained by a series of edits from a suffix of the suffix $\$^S\#^\ell$ of $\$^g(\$^S\#^S)^{S-1}\$^S\#^\ell$. This means that at least $S - \ell$ edits must be performed in the respective suffix to obtain S symbols #. This leaves us with at most g edits remaining to obtain $(\$^S\#^S)^{S-1}\$^S$. In particular, this means that the prefix $\$^S\#^S$ of $(\$^S\#^S)^{S-1}\$^S$ must be obtained from a prefix of the prefix $\$^g\$^S\#^S$ of $\$^g(\$^S\#^S)^{S-1}\$^S\#^\ell$. As, in the best case, g \$ symbols need to be substituted or removed, it follows that we need to use g edits to obtain the prefix $\$^S\#^S$ of $(\$^S\#^S)^S$. As such, we already had to use $g + (S - \ell)$ edits to transform $\$^g(\$^S\#^S)^{S-1}\$^S\#^\ell$ into $(\$^S\#^S)^S$, so it cannot be done with fewer edits. The conclusion follows. □

Proof. **Preliminaries.** We begin by recalling the following problem:

Median String: MS	
Input:	k strings $w_1, \dots, w_k \in \Sigma^*$ and an integer Δ.
Question:	Does there exist a string s such that $\sum_{i=1}^{k} d_{\texttt{ED}}(w_i, s) \leq \Delta$? (The string s for which $\sum_{i=1}^{k} d_{\texttt{ED}}(w_i, s)$ is minimum is called the median string of the strings $\{w_1, \dots, w_k\}$.)

Without loss of generality, we can assume that $\Delta \leq \sum_{i=1}^{k} |w_i|$ as, otherwise, the answer is clearly yes (for instance, for $s = \varepsilon$ we have that $\sum_{i=1}^{k} d_{\texttt{ED}}(w_i, \varepsilon) \leq \sum_{i=1}^{k} |w_i|$). Similarly, we can assume that $|s| \leq \Delta + \max\{|w_i| \mid i \in \{1, \dots, k\}\}$.

In [135] it was shown that MS is NP-complete even for binary input strings and W[1]-hard with respect to the parameter k, the number of input strings.

Reduction: intuition and definition. We will reduce MS to $\mathtt{EDMatch}_{\mathtt{1Var}}$, such that an instance of MS with k input strings is mapped to an instance of $\mathtt{EDMatch}_{\mathtt{1Var}}$ with exactly k occurrences of the variable x (the single variable occurring in the pattern).

Thus, we consider an instance of MS which consists in the k binary strings $w_1, \ldots, w_k \in \{0,1\}^*$ and the integer Δ. As mentioned above, we can assume that in this instance $\Delta \leq \sum_{i=1}^{k} |w_i|$.

The instance of $\mathtt{EDMatch}_{\mathtt{1Var}}$ which we construct consists of a word w and a pattern α, such that α contains exactly k occurrences of a variable x, and both strings are of polynomial size w.r.t. the size of the MS-instance. Moreover, the bound on the $d_{\mathtt{ED}}(\alpha, w)$ defined in this instance equals Δ. That is, if there exists a solution for the MS-instance such that $\sum_{i=1}^{k} d_{\mathtt{ED}}(w_i, s) \leq \Delta$, then, and only then, we should be able to find a solution of the $\mathtt{EDMatch}_{\mathtt{1Var}}$-instance with $d_{\mathtt{ED}}(\alpha, w) \leq \Delta$.

The construction of the $\mathtt{EDMatch}_{\mathtt{1Var}}$ instance is realized in such a way that the word w encodes the k input strings, conveniently separated by some long strings over $\{\$, \#\}$ (where $\$, \#$ are two fresh symbols), while α can be obtained from w by simply replacing each of the words w_i by a single occurrence of the variable x. Intuitively, in this way, for $d_{\mathtt{ED}}(\alpha, w)$ to be minimal, x should be mapped to the median string of $\{w_1, \ldots, w_k\}$.

We can now formally define the reduction.

For the k binary strings $w_1, \ldots w_k \in \{0,1\}^*$defining the instance of MS, let $S = 6(\sum_{i=1}^{k} |w_i|)$; clearly $S \geq 6\Delta$. Let now $w = w_1(\$^S\#^S)^S w_2(\$^S\#^S)^S \ldots w_k(\$^S\#^S)^S$ and $\alpha = \left(x(\$^S\#^S)^S\right)^k$.

Reduction: correctness. We prove first the correctness of the reduction, that is, the following claim: the instance of MS defined by $w_1, \ldots, w_k$ and Δ is answered positively if and only if the instance of $\mathtt{EDMatch}_{\mathtt{1Var}}$ defined by w, α, Δ is answered positively.

Assume first that the instance of MS defined by $w_1, \ldots, w_k$ and Δ is answered positively. Then, it is immediate to see that $d_{\mathtt{ED}}(\alpha, w) \leq \Delta$. Indeed, let $w' = \left(s(\$^S\#^S)^S\right)^k$ be the word obtained from α by replacing x with the median string s of $w_1, \ldots, w_k$. Then, clearly, $d_{\mathtt{ED}}(w', w) \leq \Delta$.

Now, assume that the instance of $\mathtt{EDMatch}_{\mathtt{1Var}}$ defined by w, α, Δ is answered positively. This means that there exists some word $t \in \{0,1,\$,\#\}^*$ such that $d_{\mathtt{ED}}(u, w) \leq \Delta$ for $u = \left(t(\$^S\#^S)^S\right)^k$.

Therefore, there exists an optimal (w.r.t. length) sequence of edits γ which transforms u into w, such that the length of γ is at most Δ. As explained in the preliminaries, we can assume that the edits in the sequence γ are ordered increasingly by the position of u to which they are applied (i.e., left to right). Our road plan is to show that if such a sequence of edits γ exists, then there exists a sequence δ of edits of equal length (so also optimal) transforming u into w, such that the edits rewrite the i^{th} occurrence of the factor t in w into w_i, for i from 1 to k, and leave the rest of the string u unchanged.

Let u_1 be the shortest prefix of u from which we obtain the prefix $w_1(\$^S\#^S)^S$ of w when applying the edits of γ. Clearly, $|w_1(\$^S\#^S)^S| - S \leq |u_1| \leq |w_1(\$^S\#^S)^S| + S$ (as the overall distance between u and w is upper bounded by $\Delta \leq S$). Let now u_1' be the longest prefix of u_1 from which we obtain w_1 when applying the edits of γ, and let $u_1 = u_1'u_1''$. Clearly, the edits of γ transform u_1'' into $(\$^S\#^S)^S$. We are now performing a case analysis.

Case 1: $|u_1'| \leq |t|$.

Case 1.1: $u_1'' = v(\$^S\#^S)^S s$, where $v, s \in \{0, 1, \#, \$\}^*$ and v is a suffix of t and s a prefix of $(t(\$^S\#^S)^S)^{k-1}$. As $|u_1''| = 2S^2 + |v| + |s|$, then at least $|v| + |s|$ edits are needed to transform u_1'' into $(\$^S\#^S)^S$. We can modify γ such that these operations are deletions of all symbols of v and s, and obtain a new sequence of edits γ'.

Case 1.2: $u_1'' = v(\$^S\#^S)^{S-1}s$, where $v \in \{0, 1, \#, \$\}^*$ is a suffix of t and $s = \$^S\#^\ell$ for some $\ell \geq S - \Delta$. We thus have $t = u_1'v$ and $|v| \leq 2\Delta$ (because $\left||u_1''| - 2S^2\right| \leq \Delta$). Further, when applying the operations of γ, after all the edits in u_1 were performed, we obtain $w_2(\$^S\#^S)^S \dots w_k(\$^S\#^S)^S$ from $\#^{S-\ell}(t(\$^S\#^S)^S)^{k-1}$ optimally. Hence, from u_1'' we obtain $(\$^S\#^S)^S$ so, after performing the p edits corresponding to positions of v (excluding the potential insertions on positions occurring to the right of the last symbol of v), we must edit them into \$ letters, so we must obtain a string $\$^g(\$^S\#^S)^{S-1}\$^S\#^\ell$ for some $0 \leq g \leq 2\Delta$. It is immediate that $p + g \geq |v|$ (as when counting the p edit operations, we count the symbols which were deleted from v, while all the symbols which were substituted in v correspond to distinct positions of $\$^g$). Now, by Lemma 4.8, since $g \leq 2\Delta$, $S - \ell \leq \Delta$, and $S \geq 6\Delta$, we get that the minimum number of edits needed to transform $\#^g(\$^S\#^S)^{S-1}\#^S\$^\ell$ into $(\$^S\#^S)^S$ is $g + (S - \ell)$. So, to transform u_1'' into $(\$^S\#^S)^S$ we use $p + g + S - \ell \geq |v| + S - \ell$ edits. We can, therefore, modify γ to obtain a new sequence of edits γ', which has at most the same length as γ, in which we first apply all the edit operations from γ to u_1', then we delete all symbols of v, then we simply leave $(\$^S\#^S)^S$ alone, then we insert $\#^{S-\ell}$ after $(\$^S\#^S)^S$, and we continue by editing $\#^{S-\ell}(t(\$^S\#^S)^S)^{k-1}$ into $w_2(\$^S\#^S)^S \dots w_k(\$^S\#^S)^S$ exactly as in γ. Clearly, we have just replaced $p + g + S - \ell$ operations in γ by $|v| + S - \ell$ edits to obtain γ'. As γ was of optimal length, and $p + g + S - \ell \geq |v| + S - \ell$, we have that γ' must be of optimal length too.

Case 2: $|u_1'| > |t|$. Then $u_1' = t\$^{S-\ell}$, for some ℓ such that $0 < S - \ell \leq \Delta$.

Case 2.1: $u_1'' = \$^\ell\#^S(\$^S\#^S)^{S-2}\$^S\#^{S-g}$ for some g such that $(S - \ell) + g \leq \Delta$. Moreover, when considering the sequence γ, we have that $\#^g(t(\$^S\#^S)^S)^{k-1}$ is transformed into $w_2(\$^S\#^S)^S \dots w_k(\$^S\#^S)^S$ optimally after the edits in u_1 are performed. As $|u_1''| = 2S^2 - g - (S - \ell)$, then at least $S - \ell + g$ edits are needed to transform u_1'' into $(\$^S\#^S)^S$. Now we can modify γ as follows. We first note that, in γ, the suffix $\$^{S-\ell}$ of u_1' has to be completely rewritten to obtain w_1 (as w_1 does not contain \$ symbols). Therefore, we transform t into w_1 by simulating the edits performed in the suffix $\$^{S-\ell}$ by only applying insertions after the last symbol of t (instead of substitutions in $\$^{S-\ell}$ we do insertions, the insertions are done as before, and the deletions from $\$^{S-\ell}$ are not needed anymore); the number of these insertions is at most as big as the number of initial edits applied to the suffix $\$^{S-\ell}$ of u_1'. Then, the factor $(\$^S\#^S)^S$ following the first t in u is not edited, as it corresponds to the identical factor of w which follows w_1, and then we insert after the first factor $(\$^S\#^S)^S$ of u a factor $\#^g$, with g insertions, and then continue editing $\#^g(t(\$^S\#^S)^S)^{k-1}$ to obtain $w_2(\$^S\#^S)^S \ldots w_k(\$^S\#^S)^S$ as in γ. The resulting sequence γ' of edits is at least $S-\ell$ edits shorter than γ, with $S-\ell > 0$. As γ was optimal, this is a contradiction, so this case is not possible.

Case 2.2: $u_1'' = \$^\ell\#^S(\$^S\#^S)^{S-1}s$ where $0 < S-\ell \leq \Delta$ and $s \in \{0,1,\#,\$\}^*$ is a prefix of $(t(\$^S\#^S)^S)^{k-1}$. In this case, in γ, we have that the suffix u' of u occurring after u_1 is transformed into $w_2(\$^S\#^S)^S \ldots w_k(\$^S\#^S)^S$ optimally after the edits in u_1 are performed. Now, the suffix s is transformed, by p edits into $\#^g$ for some $g \leq 2\Delta$, and we have $p+g \geq |v|$ (similarly to the Case 1.2). By Lemma 4.8, as in Case 1.2, we get that the minimum number of edits needed to transform $\$^\ell\#^S(\$^S\#^S)^{S-1}\#^g$ into $(\$^S\#^S)^S$ is $g+(S-\ell)$. So, overall, the number of edits needed to transform $\$^\ell\#^S(\$^S\#^S)^{S-1}s$ into $(\$^S\#^S)^S$ is $(S-\ell)+g+p \geq (S-\ell)+|s|$. Therefore, we can modify γ as follows to obtain a new optimal sequence of edits γ'. As in Case 2.1 we simulate the edits in the suffix $\$^{S-\ell}$ of u_1' by insertions. Then, the factor $(\$^S\#^S)^S$ is left unchanged. Then we simply delete the letters of s, and we continue by editing u' as in γ to obtain $w_2(\$^S\#^S)^S \ldots w_k(\$^S\#^S)^S$. Clearly, in γ' we have at least $S-\ell$ edits less than in γ, with $S-\ell > 0$. As γ was optimal, his is a contradiction, so this case is also not possible.

This concludes our case analysis.

In all possible cases (1.1 and 1.2), in the newly obtained sequence γ' of edits, which has the same optimal length as γ, we have that the prefix t of u is transformed into w_1 by a sequence of edits γ_1' (which ends with the deletion of the suffix v of t), the first factor $(\$^S\#^S)^S$ of u is then trivially transformed (by an empty sequence of edits) into the first factor $(\$^S\#^S)^S$ of w, and then $(t(\$^S\#^S)^S)^{k-1}$ is transformed into $w_2(\$^S\#^S)^S \ldots w_k(\$^S\#^S)^S$ by an optimal sequence of edits γ_2' (which starts, in Case 1.1, the deletion of the prefix s of $(t(\$^S\#^S)^S)^{k-1}$ or, in Case 1.2, with the insertion of a factor $\#^{S-\ell}$ before $(t(\$^S\#^S)^S)^{k-1}$).

Now, we can apply the same reasoning, inductively, to the optimal sequence of edits γ_2' which transforms $(t(\$^S\#^S)^S)^{k-1}$ into $w_2(\$^S\#^S)^S \ldots w_k(\$^S\#^S)^S$, and, we will ultimately obtain that there exists an optimal sequence of edits δ which transforms u into w by transforming the i^{th} factor t of u into w_i, for all i from 1 to k, and leaving the rest of the symbols of u unchanged. As the length of δ is at most Δ, this means that for the string t we have $\sum_{i=1}^k d_{\mathtt{ED}}(t, w_i)\ leq\Delta$, so the instance defined by $w_1, \ldots, w_k$ and Δ of MS can be answered positively. This concludes the proof of our claim and, as such, the proof of the correctness of our reduction.

Conclusion. The instance of $\mathtt{EDMatch}_{\mathtt{1Var}}$ (i.e., α, w, Δ) is of polynomial size w.r.t. the size of the MS-instance. Therefore, the instance of $\mathtt{MinEDMatch}_{\mathtt{1RepVar}}$ can be computed in polynomial time, and our entire reduction is done in polynomial time. Moreover, we have shown that the instance

(w, α, Δ) of $\texttt{EDMatch}_{\texttt{1Var}}$ is answered positively if and only if the original instance of $\texttt{MS}$ is answered positively. Finally, as the number of occurrences of the variable x blocks in α is k, where k is the number of input strings in the instance of $\texttt{MS}$, and $\texttt{MS}$ is $W[1]$-hard with respect to this parameter, it follows that $\texttt{EDMatch}_{\texttt{1Var}}$ is also $W[1]$-hard when the number of occurrences of the variable x in α is considered as parameter. This completes the proof of our theorem. □

4.3.2 Solution for Unary Pattern

A simple corollary of Theorem 4.2 is the following:

Corollary 4.9. $\texttt{EDMatch}_{\texttt{1Var}}$ *and* $\texttt{MinEDMatch}_{\texttt{1Var}}$ *can be solved in* $O(n^{3|\alpha|_x})$ *time, where* x *is the single variable occurring in* α.

Clearly, finding a polynomial time algorithm for $\texttt{EDMatch}_{\texttt{1Var}}$, for which the degree of the polynomial does not depend on $|\alpha|_x$, would be ideal. Such an algorithm would be, however, an FPT-algorithm for $\texttt{EDMatch}_{\texttt{1Var}}$, parameterized by $|\alpha|_x$, and, by Theorem 4.7 and common parameterized complexity assumptions, the existence of such an algorithm is unlikely. This makes the straightforward result reported in Corollary 4.9 relevant, to a certain extent.

A conclusion of all the results as well as an outlook on future work is presented in Section 7.

CHAPTER 5

Matching Patterns with Variables under Simon's Congruence

This chapter is based on article [65] (see reference below) and the LaTeX code of this article was used to reproduce it here. The Introduction and Preliminaries of this paper are worked into Chapter 1 and Chapter 2. Further, the notations are adjusted to match the other chapters and provide a uniform notation across this thesis.

Reference: P. Fleischmann, S. Kim, T. Koß, F. Manea, D. Nowotka, S. Siemer, and M. Wiedenhöft. Matching Patterns with Variables Under Simon's Congruence. In O. Bournez, E. Formenti, and I. Potapov, editors, *Reachability Problems - 17th International Conference, RP 2023, Nice, France, October 11-13, 2023, Proceedings*, volume 14235 of *Lecture Notes in Computer Science*, pages 155–170. Springer, 2023. `doi:10.1007/978-3-031-45286-4\_12`

Description: This paper combined two settings we extensively studied in the group. On the one hand, we have Simon's congruence, which asks for the equivalence of the respective sets of subsequences (up to a specific length) of two words. On the other hand, we have the problem of matching patterns with variables, which was considered in an approximate setting under string metrics in Chapter 3 [78] and Chapter 4 [79]. When we combine these two concepts, we ask for a substitution of the variables in our pattern to reach a word that is k-Simon congruent to a target word. Contrary to the metrics, this gives us a relation that describes a similarity of the pattern (under a substitution) and our target word. This problem originated from a joint workshop of our group in Göttingen and the *Dependable Systems* group in Kiel, where the co-authors Pamela Fleischmann and Max Wiedenhöft proposed the problem.

Contribution: I contributed to the complexity results obtained in this paper and was a main contributor of the NP-hardness reductions.

5.1 Overview

In this chapter, we study the matching of patterns under Simon's congruence: we want the word w and the image of the pattern α under a substitution h to have the same set of subsequences of a given length k. Further, we consider the problem of finding a substitution h for all the variables in the pattern α, such that applying this substitution yields a *k-subsequence universal* word.

Matching under Simon's Congruence: $\texttt{MatchSimon}(\alpha, w, k)$	
Input:	Pattern α, $\lvert\alpha\rvert = m$, word w, $\lvert w\rvert = n$, and number $k \in [n]$.
Question:	Is there a substitution h with $h(\alpha) \sim_k w$?

Matching a Target Universality: $\texttt{MatchUniv}(\alpha, k)$	
Input:	Pattern α, $\lvert\alpha\rvert = m$, and $k \in \mathbb{N}_0$.
Question:	Is there a substitution h with $\iota(h(\alpha)) = k$?

In this problem, $\iota(w)$ (the universality index of w) is the largest integer ℓ for which w is ℓ-subsequence universal. Note that $\texttt{MatchUniv}$ can be formulated in terms of $\texttt{MatchSimon}$: the answer to $\texttt{MatchUniv}(\alpha, k)$ is yes if and only if the answer to $\texttt{MatchSimon}(\alpha, (1 \cdots \sigma)^k, k)$ is yes and the answer to $\texttt{MatchSimon}(\alpha, (1 \cdots \sigma)^{k+1}, k + 1)$ is no. However, there is an important difference: for $\texttt{MatchUniv}$ we are not explicitly given the target word w, whose set of k-length subsequences we want to reach; instead, we are given the number k which represents the target set more compactly (using only $\log k$ bits).

In the problems introduced above, we attempt to match (or reach), starting with a pattern α, the set of subsequences defined by a given word w (given explicitly or implicitly). We are also going to extend $\texttt{MatchSimon}$ to the problem of solving word equations under $\sim_k$, defined as follows.

Word Equations under Simon's Congruence: $\texttt{WESimon}(\alpha, \beta, k)$	
Input:	Patterns α, β, $\lvert\alpha\rvert = m$, $\lvert\beta\rvert = n$, and $k \in [m + n]$.
Question:	Is there a substitution h with $h(\alpha) \sim_k h(\beta)$?

Before we go into the results of this work let us state the following assumptions. The problems addressed in this work deal with matching patterns to words under Simon's congruence $\sim_k$. For these problems, the input consists of patterns, words, and a number k. In general, we assume that each letter of Σ appears at least once, in at least one of the input patterns or words. Therefore, for input pattern α and word w we assume that $\Sigma = \texttt{term}(\alpha) \cup \texttt{alph}(w)$. Hence, σ is upper bounded by the total length of the input words and patterns. Similarly, the total number of variables occurring in the input patterns is upper bounded by the total length of these patterns. However, in this section and in [65], although the number of variables is not restricted, we assume that σ is a constant, i.e.,

$\sigma \in O(1)$. Clearly, the complexity lower bounds proven in this setting for the analysed problems are stronger while the upper bounds are weaker than in the general case, when no restriction is placed on σ. Note, however, that $\sigma \in O(1)$ is not an unusual assumption, being used in, e.g., [62].

In Section 5.2 we show that `MatchUniv` is NP-complete, and also present a series of structurally restricted classes of patterns, for which it can be solved in polynomial time. In Section 5.3, we approach `MatchSimon` and show that it is also NP-complete; some other variants of this problem, both tractable and intractable, are also discussed. Finally, in Section 5.4, we discuss `WESimon` and its variants, and characterise their computational complexity.

5.2 Complexity of `MatchUniv`

In this section, we discuss the `MatchUniv` problem. In this problem, we are given a pattern α and a natural number $k \leq n$, and we want to check the existence of a substitution h with $\iota(h(\alpha)) = k$. Note that $\iota(h(\alpha)) = k$ means both that $h(\alpha)$ is k-universal and that it is not $(k+1)$-universal. A slightly relaxed version of the problem, where we would only ask for $h(\alpha)$ to be k-universal is trivial (and, therefore, not interesting): the answer, in that case, is always positive, as it is enough to map one of the variables of α to $(1 \cdots \sigma)^k$. The main result of this section is that `MatchUniv` is NP-complete. Because the subproofs are long and based on involved ideas, we split them into two individual sections to make them easier to follow. Based on Lemma 5.2 in Section 5.2.1 we get NP-hardess and from Lemma 5.7 in Section 5.2.2 we get the containment in NP, such that the following theorem trivially follows from the upcoming two sections.

Theorem 5.1. `MatchUniv` *is* NP-*complete.*

5.2.1 `MatchUniv` is NP-hard

To show that `MatchUniv`(α, k) is NP-hard, we reduce `3CNFSAT` (3-satisfiability in conjunctive normal form) to `MatchUniv`(α, k). We provide several gadgets allowing us to encode a `3CNFSAT`-instance φ as an `MatchUniv`-instance (α, k). Finally, we show that we can find a substitution h for the instance (α, k), such that $\iota(h(\alpha)) = k$, if and only if φ is satisfiable. We begin by recalling `3CNFSAT`.

3-Satisfiability for formulas in conjunctive normal form, `3CNFSAT`.

Input: Clauses $\varphi := \{c_1, c_2, \ldots, c_m\}$, where $c_j = (y_j^1 \vee y_j^2 \vee y_j^3)$ for $1 \leq j \leq m$, and y_j^1, y_j^2, y_j^3 from a finite set of boolean variables $X := \{x_1, x_2, \ldots, x_n\}$ and their negations $\bar{X} := \{\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n\}$.

Question: Is there an assignment for X, which satisfies all clauses of φ?

It is well-known that `3CNFSAT` is NP-complete (see [102, 74] for a proof). With this result at hand, we can prove the following lower bound.

Lemma 5.2. `MatchUniv` *is* NP-*hard.*

Proof. We reduce `3CNFSAT` to `MatchUniv`(α, k). Let us consider an instance of `3CNFSAT`: formula φ given by m clauses $\varphi := \{c_1, c_2, \ldots c_m\}$ over n variables $X := \{x_1, x_2, \ldots x_n\}$ (for simplicity in notation we define $N = n+m$). We map this `3CNFSAT` instance to an instance (α, k) of `MatchUniv`(α, k) with $k = 5n + m + 2$, the alphabet $\Sigma := \{\mathtt{0}, \mathtt{1}, \#, \$\}$ and the variable set $\mathcal{X} := \{\mathtt{z}_1, \mathtt{z}_2, \ldots \mathtt{z}_n, \mathtt{u}_1, \mathtt{u}_2, \ldots \mathtt{u}_n\}$. More precisely, we want to show that there exists a substitution h to replace all the variables in α with constant words, such that $\iota(h(\alpha)) = 5n + m + 2$, if and only if the boolean formula φ is satisfiable. Our construction can be performed in polynomial time and is of polynomial size with respect to N. To present this construction, we will go through its building blocks, the so-called gadgets.

Before we start with these gadgets, let us introduce a renaming function for the variables $\rho : X \cup \bar{X} \to \mathcal{X}$ with $\rho(x_i) = \mathtt{z}_\mathtt{i}$ and $\rho(\bar{x}_i) = \mathtt{u}_\mathtt{i}$. Also, a substitution h which maps α to a string of universality index $5n + m + 2$ is called valid in the following.

The binarisation gadgets. We use the following gadgets to make the image of variables $\mathtt{z}_\mathtt{i}$ and $\mathtt{u}_\mathtt{i}$ under a valid substitution be strings over $\{0, 1\}$. Recall that we have the alphabet $\Sigma := \{\mathtt{0}, \mathtt{1}, \#, \$\}$ and the set of variables $\mathcal{X} := \{\mathtt{z}_1, \mathtt{z}_2, \ldots, \mathtt{z}_n, \mathtt{u}_1, \mathtt{u}_2, \ldots, \mathtt{u}_n\}$.

At first, we construct the gadget $\pi_\# = (\mathtt{z}_1\mathtt{z}_2 \cdots \mathtt{z}_n\mathtt{u}_1\mathtt{u}_2 \cdots \mathtt{u}_n\mathtt{0}\,\mathtt{1}\,\$)^{\mathtt{N}^6}\,\#$, as shown in Figure 5.1. We observe that for all possible substitutions h, we have two cases for the universality of the image of this gadget. On the one hand, assume that any of the variables is substituted under h by a string that contains a #. Then, the universality index of the image of this gadget will be $\iota(h(\pi_\#)) = k'$ with $k' \geq N^6 > k$, which is too big for a valid substitution. On the other hand, when all the variables are substituted under h by strings that do not contain #, this gadget is mapped to a string which is exactly one arch because there is only one # at its very end. Thus, under a valid substitution h, the images of the variables $\mathtt{z}_\mathtt{i}$ and $\mathtt{u}_\mathtt{i}$ do not contain #. Note also that, in the arch factorisation of such a string ($h(\pi_\#)$, where h is a valid substitution) we have one arch and no rest.

The gadget $\pi_\$ = (\mathtt{z}_1\mathtt{z}_2 \cdots \mathtt{z}_n\mathtt{u}_1\mathtt{u}_2 \cdots \mathtt{u}_n\mathtt{0}\,\mathtt{1}\,\#)^{\mathtt{N}^6}\,\$$ is constructed analogously and can be seen in Figure 5.2. This enforces that under a valid substitution h, the images of the variables $\mathtt{z}_\mathtt{i}$ and $\mathtt{u}_\mathtt{i}$ do not contain $.

In conclusion, the gadgets $\pi_\#$ and $\pi_\$$ enforce that under a valid substitution h, the image of the variables $\mathtt{z}_\mathtt{i}$ and $\mathtt{u}_\mathtt{i}$ contains only 0 and 1, i.e., they are binary strings.

$$\pi_\# = (\ \underbrace{\mathtt{z}_1\mathtt{z}_2 \cdots \mathtt{z}_n\ \ \mathtt{u}_1\mathtt{u}_2 \cdots \mathtt{u}_n}_{\text{No \# allowed}}\ \ \mathtt{0}\,\mathtt{1}\,\$\ )^{\mathtt{N}^6}\ \#$$

Figure 5.1: If any of the variables is substituted by a string that contains a #, then this gadget would add at least N^6 arches, which is already greater than the target universality $k = 5n + m + 2$.

$$\pi_{\$} = (\underbrace{\mathtt{z}_1\mathtt{z}_2\cdots\mathtt{z}_n\ \ \mathtt{u}_1\mathtt{u}_2\cdots\mathtt{u}_n\ \ \mathtt{0}1\#\ }_{\text{No \$ allowed}})^{N^6}\ \ \$$$

Figure 5.2: If any of the variables is substituted by a string that contains a $\$$, then this gadget would add at least N^6 arches, which is already greater than the target universality $k = 5n + m + 2$.

The Boolean gadgets. We use the following gadgets to force the image of each $\mathtt{z_i}$ and $\mathtt{u_i}$ to be either in $\mathtt{0}^*$ or 1^*. Intuitively, mapping a variable $\mathtt{z_i}$ (respectively, $\mathtt{u_i}$) to a string of the form 0^+ corresponds to mapping x_i (respectively, $\bar{x}_i$) to the Boolean value false (respectively, true). Similarly, mapping one of these string-variables to a string from 1^+ means mapping the corresponding boolean variable to true. For a beginning, these gadgets just have to enforce that the image of any string-variable does not contain both $\mathtt{0}$ and 1. We construct the gadget $\pi^{\mathtt{z}}_{\mathtt{i}}$ (respectively $\pi^{\mathtt{u}}_{\mathtt{i}}$) for every string-variable $\mathtt{z_i}$ (respectively, $\mathtt{u_i}$), according to Figure 5.3. More precisely, for all $i \in [n]$, we define two gadgets $\pi^{\mathtt{z}}_{\mathtt{i}} = (\mathtt{z_i}\ \$\,\#)^{N^6}\,\mathtt{1001}\,\$\,\#$ and $\pi^{\mathtt{u}}_{\mathtt{i}} = (\mathtt{u_i}\ \$\,\#)^{N^6}\,\mathtt{1001}\,\$\,\#$.

We now analyse the possible images of $\pi^{\mathtt{z}}_{\mathtt{i}} = (\mathtt{z_i}\ \$\,\#)^{N^6}\,\mathtt{1001}\,\$\,\#$ under various substitutions h. There are three ways in which $\mathtt{z_i}$ can be mapped to a string by h. Firstly, if the image of $\mathtt{z_i}$ contains both $\mathtt{0}$ and 1, then for the universality index of the image of $\pi^{\mathtt{z}}_{\mathtt{i}}$ under the respective substitution is $\iota(h(\pi^{\mathtt{z}}_{\mathtt{i}})) \geq N^6 > k$; such a substitution cannot be valid. Secondly, if the image of $\mathtt{z_i}$ is a string from $\mathtt{0}^*$, then the universality of this gadget is exactly $\iota(h(\pi^{\mathtt{z}}_{\mathtt{i}})) = 2$ as shown in Figure 5.4. As a third option, if the image of $\mathtt{z_i}$ is a string from 1^*, then the universality of this gadget is exactly $\iota(h(\pi^{\mathtt{z}}_{\mathtt{i}})) = 2$ as shown in Figure 5.5. As for the binarisation gadgets, in the arch factorisation of a string $h(\pi^{\mathtt{z}}_{\mathtt{i}})$, where h is a valid substitution, we have exactly two arches (and no rest). A similar analysis can be performed for the gadgets $\pi^{\mathtt{u}}_{\mathtt{i}} = (\mathtt{u_i}\ \$\,\#)^{N^6}\,\mathtt{1001}\,\$\,\#$. In conclusion, the gadgets $\pi^{\mathtt{z}}_{\mathtt{i}}$ and $\pi^{\mathtt{u}}_{\mathtt{i}}$ enforce that under a valid substitution h, the image of the variables $\mathtt{z_i}$ and $\mathtt{u_i}$ contains either only 0s or only 1s (or is empty).

$$\pi^{\mathtt{z}}_{\mathtt{i}} = \ (\ \underbrace{\mathtt{z_i}\ \ \$\,\#}_{\text{No } \mathtt{0} \text{ and } 1 \text{ together allowed}}\ )^{N^6}\ \ 1\ \ \mathtt{0}\ \ \mathtt{0}1\,\$\,\#$$

Figure 5.3: If any of the variables in the gadget $\pi^{\mathtt{z}}_{\mathtt{i}}$ (analogous for $\pi^{\mathtt{u}}_{\mathtt{i}}$) is substituted by a string that contains both a 0 and a 1, then this gadget would add N^6 arches, which is already greater than the target universality $k = 5n + m + 2$.

$$\pi^{\mathtt{z}}_{\mathtt{i}} = \ (\ \underbrace{\mathtt{z_i}\ \ \$\,\#\ )^{N^6}\ \ 1}_{\text{arch if } \mathtt{z_i} \in \mathtt{0}^+}\underbrace{\ \ \mathtt{0}\ \ \mathtt{0}1\,\$\,\#}_{}$$

Figure 5.4: If any of the variables $\pi^{\mathtt{z}}_{\mathtt{i}}$ (analogous for $\pi^{\mathtt{u}}_{\mathtt{i}}$) consits of only 0's, this gadget would add 2 arches per variable.

$$\pi_{\mathtt{i}}^{\mathtt{z}} = \underbrace{(\ \mathtt{z_i}\ \$\#\ )^{N^6}\ 1\ \mathbb{0}}_{\text{arch if } \mathtt{z_i} \in 1^*}\underbrace{\mathbb{0}1\$\#}$$

Figure 5.5: If any of the variables $\pi_{\mathtt{i}}^{\mathtt{z}}$ (analogous for $\pi_{\mathtt{i}}^{\mathtt{u}}$) consits of only 1's, this gadget would add 2 arches per variable.

The complementation gadgets. The role of these gadgets is to enforce the property that $\mathtt{z_i}$ and $\mathtt{u_i}$ are not both in $\mathbb{0}^+$ or not both in 1^+, for all $i \in [n]$. We construct the gadget $\xi_{\mathtt{i}} = \$\,\mathtt{z_i}\mathtt{u_i}\,\#$, for every $i \in [n]$, according to Figure 5.6. Let us now analyse the image of these gadgets under a valid substitution ($\pi_\#$ and $\pi_\$$ are mapped to exactly one arch each, and $\pi_{\mathtt{i}}^{\mathtt{z}}$ and $\pi_{\mathtt{i}}^{\mathtt{u}}$ are mapped to exactly two arches each). In this case, we observe that $\xi_{\mathtt{i}}$ is mapped to exactly one complete arch ending on the rightmost symbol # if and only if the image of one of the variables $\mathtt{z_i}$ and $\mathtt{u_i}$ has at least one 0 and the image of the other one has at least one 1. Further, let us consider the concatenation of two consecutive such gadgets $\xi_{\mathtt{i}}\xi_{\mathtt{i+1}}$ and assume that both $\mathtt{z_i}$ and $\mathtt{u_i}$ are mapped to strings over the same letter or at least one of them is mapped to the empty word. In that case, the first arch must close to the right of the \$ letter in ξ_{i+1}, hence $\xi_i\xi_{i+1}$ could not contain two arches. Thus, the concatenation of the gadgets $\xi_1 \cdots \xi_n$ is mapped to a string which has exactly n arches if and only if each gadget ξ_i is mapped to exactly one arch, which holds if and only if the image of one of the variables $\mathtt{z_i}$ and $\mathtt{u_i}$ has at least one 0 and the image of the other one has at least one 1. When assembling together all the gadgets, we will ensure that, in a valid substitution, this property holds: $\mathtt{z_i}$ and $\mathtt{u_i}$ are mapped to repetitions of different letters.

$$\xi_{\mathtt{i}} = \ \$\ \ \mathtt{z_i}\ \ \mathtt{u_i}\ \ \#$$

Figure 5.6: In order for $\xi_{\mathtt{i}}$ to contribute an arch, one of z_i and u_i has to be replaced by only 1's while the other must consist of only 0's.

The clause gadgets. Let $c_j = (y_j^1 \vee y_j^2 \vee y_j^3)$ be a clause, with $y_j^1, y_j^2, y_j^3 \in X \cup \bar{X}$. We construct the gadget $\delta_{\mathtt{j}}$ for every clause c_j as $\$\,\mathbb{0}\rho(\mathtt{y_j^1})\rho(\mathtt{y_j^2})\rho(\mathtt{y_j^3})\,\#$, as shown in Figure 5.7. Now, by all of the properties discussed for the previous gadgets, we can analyse the possible number of arches contained in the image of this gadget under a valid substitution. Firstly, note that if at least one of the variables $\rho(\mathtt{y_j^1}), \rho(\mathtt{y_j^2}), \rho(\mathtt{y_j^3})$ is mapped to a string containing at least one 1, then this gadget will contain exactly one arch ending on its rightmost symbol #. Now consider the concatenation of two consecutive such gadgets $\delta_j\delta_{j+1}$, and assume that all the variables in δ_j are substituted by only 0s. In this case, the first arch must end to the right of the \$ symbol in δ_{j+1}, hence the string to which $\delta_{\mathtt{j}}\delta_{\mathtt{j+1}}$ is mapped could not contain two arches. The same argument holds if we look at the concatenation of the last complementation gadget and the first clause gadget, e.g. $\xi_n\delta_1$.

Thus, the concatenation of the gadgets $\delta_1 \cdots \delta_m$ is mapped to a string which has exactly m arches if and only if each gadget δ_i is mapped to exactly one arch. This holds if and only if at least one of the string-variables occurring in δ_i is mapped to a string of 1s. When assembling together all the gadgets, we will ensure that at least one of the variables occurring in each gadget δ_i, for all $i \in [m]$, is mapped to a string of 1s in a valid substitution.

$$\delta_{\mathtt{j}} = \quad \mathtt{\$} \quad \mathtt{0} \quad \rho(\mathtt{y}_{\mathtt{j}}^1) \quad \rho(\mathtt{y}_{\mathtt{j}}^2) \quad \rho(\mathtt{y}_{\mathtt{j}}^3) \quad \mathtt{\#}$$

Figure 5.7: In order for $\delta_{\mathtt{j}}$ to contribute an arch, at least one of $\rho(\mathtt{y}_{\mathtt{j}}^1)$, $\rho(\mathtt{y}_{\mathtt{j}}^2)$ and $\rho(\mathtt{y}_{\mathtt{j}}^3)$ has to be replaced by only 1's.

Final Assemblage. We finish the construction of the pattern α by concatenating all the gadgets. That is, $\alpha = \pi_{\#}\pi_{\$}\pi_1^{\mathtt{z}}\pi_1^{\mathtt{u}}\pi_2^{\mathtt{z}}\pi_2^{\mathtt{u}} \cdots \pi_{\mathtt{n}}^{\mathtt{z}}\pi_{\mathtt{n}}^{\mathtt{u}}\xi_1\xi_2 \cdots \xi_{\mathtt{n}}\delta_1\delta_2 \cdots \delta_{\mathtt{m}}$, as shown in Figure 5.8.

$$\alpha = \underbrace{\pi_{\#}}_{1} \; \underbrace{\pi_{\$}}_{1} \; \underbrace{\pi_1^{\mathtt{z}}\pi_1^{\mathtt{u}}\pi_2^{\mathtt{z}}\pi_2^{\mathtt{u}} \cdots \pi_{\mathtt{n}}^{\mathtt{z}}\pi_{\mathtt{n}}^{\mathtt{u}}}_{4\mathtt{n}} \; \underbrace{\xi_1\xi_2 \cdots \xi_{\mathtt{n}}}_{\mathtt{n}} \; \underbrace{\delta_1\delta_2 \cdots \delta_{\mathtt{m}}}_{\mathtt{m}}$$

Figure 5.8: The concatenation of all gadgets and their respective amount of arches we expect, if we can find a substitution h with $\iota(h(\alpha)) = 5n + m + 2$.

The correctness of the reduction. We show that there exists a substitution h of the string variables of α with $\iota(h(\alpha)) = 5n + m + 2$ (i.e., a valid substitution) if and only if we can find an assignment for all Boolean-variables occurring in φ that satisfy all clauses $c_j \in \varphi$.

Let us first show that if there is a satisfying assignment for Boolean-variables of φ which makes the formula true, then there exists a substitution h of the string-variables of α such that $\iota(h(\alpha)) = 5n + m + 2$. In this case, we can give a canonical substitution h with $h(\rho(x_i)) = 1$ and $h(\rho(\bar{x}_i)) = 0$ if x_i is assigned true, and $h(\rho(x_i)) = 0$ and $h(\rho(\bar{x}_i)) = 1$ if x_i is assigned false. We can easily verify, by the definition of the gadgets, that under this substitution we have $\iota(h(\alpha)) = 5n + m + 2$. Indeed, in the images of each gadget $\pi_{\#}, \pi_{\$}, \pi_{\mathtt{i}}^{\mathtt{z}}, \xi_i$ and δ_i we have exactly one arch, ending on the last symbol of the respective strings, while in the image of each gadget $\pi_{\mathtt{i}}^{\mathtt{z}}$ under this substitution there will be exactly two arches, again ending on their last positions.

Conversely, we want to show that if we have a substitution h of the string-variables such that $\iota(h(\alpha)) = 5n + m + 2$, then there must be a satisfying assignment of the Boolean-variables for φ. The general idea is the following. We assume that we have a substitution of the string variables and compute the arch factorisation greedily and look at the properties enforced by the individual gadgets, as discussed above. Assume first, towards a contradiction, that the image of some variable $\mathtt{z_i}$ contains both 0 and 1 or that it contains # or \$. Then, as explained, the number of arches of the image of $\pi_{\#}\pi_{\$}\pi_1^{\mathtt{z}}\pi_1^{\mathtt{u}}\pi_2^{\mathtt{z}}\pi_2^{\mathtt{u}} \cdots \pi_{\mathtt{n}}^{\mathtt{z}}\pi_{\mathtt{n}}^{\mathtt{u}}$ will blow up to a value greater than $5n+m+2$, a contradiction. The same reasoning holds for the variables $\mathtt{u_i}$. Therefore, each variable $\mathtt{z_i}$ is mapped to a string from

$\mathtt{0}^* \cup \mathtt{1}^*$, and the same holds for the variables $\mathtt{u_i}$. It follows that $h(\pi_{\#}\pi_{\$}\pi_1^{\mathtt{z}}\pi_1^{\mathtt{u}}\pi_2^{\mathtt{z}}\pi_2^{\mathtt{u}} \cdots \pi_{\mathtt{n}}^{\mathtt{z}}\pi_{\mathtt{n}}^{\mathtt{u}})$ contributes exactly $4n + 2$ arches to the arch factorisation of $h(\alpha)$, and the last arch of this factorisation (when identified greedily, from left to right) ends on the last letter of $\pi_{\mathtt{n}}^{\mathtt{u}}$ (which is a # symbol). By this last property, we are guaranteed that we can look at the suffix $h(\xi_1\xi_2 \cdots \xi_{\mathtt{n}}\delta_1\delta_2 \cdots \delta_{\mathtt{m}})$ of our pattern's image under h separately, as no arch from the prefix $h(\pi_{\#}\pi_{\$}\pi_1^{\mathtt{z}}\pi_1^{\mathtt{u}}\pi_2^{\mathtt{z}}\pi_2^{\mathtt{u}} \cdots \pi_{\mathtt{n}}^{\mathtt{z}}\pi_{\mathtt{n}}^{\mathtt{u}})$ extends in it. More precisely, this allows us to consider the subproblem of analysing h under the assumption that $\iota(h(\xi_1\xi_2 \cdots \xi_{\mathtt{n}}\delta_1\delta_2 \cdots \delta_{\mathtt{m}})) = m + n$, and, moreover, each string variable is mapped to strings from $\mathtt{0}^* \cup \mathtt{1}^*$. In this subproblem, we have $n + m$ \$ symbols in the pattern and we can not introduce new \$ symbols in the image of the string-variables. Therefore, every \$ symbol needs to be in exactly one arch. Now, as discussed when introducing the complementation and clause gadgets, we have to have the following properties, as otherwise we would have at least two \$ symbols in the same arch and would only get to $k' < m + n$ arches overall. Firstly, one of each $h(\rho(x_i))$ and $h(\rho(\bar{x}_i))$ has to consist only of $\mathtt{0}$s while the other consists only of $\mathtt{1}$s, and both of them should have length at least 1. Secondly, at least one of each $\rho(\mathtt{y}_{\mathtt{j}}^1)$, $\rho(\mathtt{y}_{\mathtt{j}}^2)$ and $\rho(\mathtt{y}_{\mathtt{j}}^3)$ has to be substituted by a string from $\mathtt{1}^+$.

Given these properties, we can construct a satisfying assignment of the Boolean-variables from φ by setting a variable to be true if and only if their corresponding string-variable is mapped to a string from $\mathtt{1}^*$. As $h(\rho(x_i))$ and $h(\rho(\bar{x}_i))$ are mapped to strings over distinct alphabets, we get that x_i and $\bar{x}_i$ will be assigned distinct truth values. Moreover, at least one of each $\rho(\mathtt{y}_{\mathtt{j}}^1)$, $\rho(\mathtt{y}_{\mathtt{j}}^2)$ and $\rho(\mathtt{y}_{\mathtt{j}}^3)$ has to be substituted by a string from $\mathtt{1}^+$, so at least one variable per clause is assigned to true. Therefore, this assignment makes φ true.

This concludes our proof, and shows that $\mathtt{MatchUniv}(\alpha, k)$ is NP-hard. □

5.2.2 `MatchUniv` is in NP

In the following we show that $\mathtt{MatchUniv}(\alpha, k)$ is in NP. One natural approach is to guess the images of the variables occurring in the input pattern α under a substitution h and check whether or not $\iota(h(\alpha))$ is indeed k. However, it is difficult to bound the size of the images of the variables of α under h in terms of the size of α and $\log k$ (the size of our input), since the strings we look for may be exponentially long. For example, consider the pattern $\alpha = X_1$: the length of the shortest k-universal string is $k\sigma$ [17], which is already exponential in $\log k$. Therefore, we consider guessing only the subsequence universality signatures for the image of each variable under the substitution. We show that it is sufficient to guess $|\mathtt{var}(\alpha)|$ subsequence universality signatures, one for each variable, instead of the actual images of the variables under a substitution h using the following proposition by Schnoebelen and Veron [152].

Proposition 5.3 ([152]). *For $u, v \in \Sigma^*$, we can compute $\mathsf{s}(uv)$, given the subsequence universality signatures $\mathsf{s}(u) = (\gamma_u, \mathcal{K}_u, \mathcal{R}_u)$ and $\mathsf{s}(v) = (\gamma_v, \mathcal{K}_v, \mathcal{R}_v)$ of each string, in time polynomial in $|\mathtt{alph}(uv)|$ and $\log t$, where t is the maximum element of $\mathcal{K}_u$ and $\mathcal{K}_v$.*

Once we have guessed the subsequence universality signatures of all variables in $\mathtt{var}(\alpha)$ under substitution h, we can compute $\iota(h(\alpha))$ in the following way. We first compute the subsequence universality signature of the maximal prefix of α that does not contain any variables. We then incrementally compute the subsequence universality signature of prefixes of the image of α. Let $\alpha = \alpha_1\alpha_2$, where we already have $\mathtt{s}(h(\alpha_1))$ from induction. If $\alpha_2[1]$ is a variable, we compute $\mathtt{s}(h(\alpha_1\alpha_2[1]))$ from $\mathtt{s}(h(\alpha_1))$ and the guessed subsequence universality signature for variable $\alpha_2[1]$, using Proposition 5.3. Otherwise, we take the maximal prefix w of α_2 that does not consist of any variables. We first compute $\mathtt{s}(w)$ and then compute $\mathtt{s}(h(\alpha_1 w))$ using Proposition 5.3. Once we have $\mathtt{s}(h(\alpha)) = (\gamma, \mathcal{K}, \mathcal{R})$, we compute $\iota(h(\alpha)) = \mathcal{K}[\sigma] + 1$. Note that the whole process can be done in a polynomial number of steps in $|\alpha|$, $\log k$, and σ due to Proposition 5.3, provided that the signatures are of polynomial size.

Thus, we now measure the encoding size of a subsequence universality signature and, as such, the overall size of the certificate for `MatchUniv` that we guess. We can use $\sigma!$ bits to encode a permutation γ of a subset of Σ. An integer between 1 and $\sigma - 1$ requires $\log\sigma$ bits. Naively, $\mathcal{R}$ requires $(2^\sigma)^\sigma$ bits because there can be 2^σ choices for each item. Finally, in the framework of our problem, note that $\mathcal{K}[1] - \mathcal{K}[|\gamma|] \leq 1$ by Schnoebelen and Veron [152], and that the values of $\mathcal{K}[i]$ are non-increasing in i. Therefore, we can encode $\mathcal{K}$ as a tuple (l, k') where $k' = \max\{\mathcal{K}[i] \mid 1 \leq i \leq |\gamma|\} \leq k$ and $l = |\{i \in [|\gamma|] \mid \mathcal{K}[i] = k'\}|$. This encoding scheme requires at most $\log\sigma + \log k$ bits. Summing up, the overall space required to encode a certificate that consists of $|\mathtt{var}(\alpha)|$ subsequence universality signatures takes at most $(1 + \sigma! + (2^\sigma)^\sigma + \log\sigma + \log k)|\mathtt{var}(\alpha)|$ bits. This is polynomial in the size of the input and the number of variables, because we assume a constant-sized alphabet, i.e. $\sigma \in O(1)$.

It remains to design a deterministic polynomial algorithm that tests the validity of the guessed subsequence universality signature. Assume that we have guessed the 3-tuple $(\gamma, \mathcal{K}, \mathcal{R})$. We claim that there are only constantly many strings we need to check to decide whether or not $(\gamma, \mathcal{K}, \mathcal{R})$ is a valid subsequence universality signature - allowing us a brute-force approach. Lemma 5.4 allows us to "pump down" strings with universality index greater than $(2^\sigma)^\sigma$, which is a constant.

Lemma 5.4. *The tuple $(\gamma, \mathcal{K}_1, \mathcal{R})$ is a valid subsequence universality signature iff there exists $w \in \Sigma^*$ with $\iota(w) \leq (2^\sigma)^\sigma$, $\mathtt{s}(w) = (\gamma, \mathcal{K}_2, \mathcal{R})$, and $\mathcal{K}_1[t] - myKarr_2[t] = c \in \mathbb{N}_0$ for all $t \in [|\gamma|]$.*

Proof. **Only if part.** Let $\mathtt{s}(w) = (\gamma, \mathcal{K}_2, \mathcal{R})$ with $\iota(w) \geq 1$. Then, we have $\mathtt{s}(\gamma^c w) = (\gamma, \mathcal{K}_1, \mathcal{R})$ where $\mathcal{K}_1[t] = \mathcal{K}_2[t] + c$ for all $t \in [\sigma]$ and all $c \in \mathbb{N}_0$.

If part. Let u be a string with $\mathtt{s}(u) = (\gamma, \mathcal{K}_1, \mathcal{R})$. If $\iota(u) \leq (2^\sigma)^\sigma$, the statement is already true. Otherwise, we have $\iota(u) > (2^\sigma)^\sigma$. Consider two integers i and j $(i < j)$, which are multiples of σ. Assume that $\mathtt{alph}(u[\mathtt{M}_{i+l}(u) + 1 : \mathtt{M}_{i+l+1}(u)]) = \mathtt{alph}(u[\mathtt{M}_{j+l}(u) + 1 : \mathtt{M}_{j+l+1}(u)])$ for $l \in [\sigma - 1] \cup \{0\}$. Note that the endpoints of the arches after $\mathtt{M}_{j+\sigma}(u)$ depend exactly on the set of characters $\texttt{alph}(w[\texttt{M}_{j+l}(u)+1 : \texttt{M}_{j+l+1}(u)])$ from $l = 0$ to $\sigma - 1$, and the suffix $u[\texttt{M}_{j+\sigma}(u)]$. Therefore, we can remove $u[\texttt{M}_{i+\sigma}(u) + 1 : \texttt{M}_{j+\sigma}(u)]$ without altering γ or $\mathcal{R}$. This argument is illustrated in Figure 5.9.

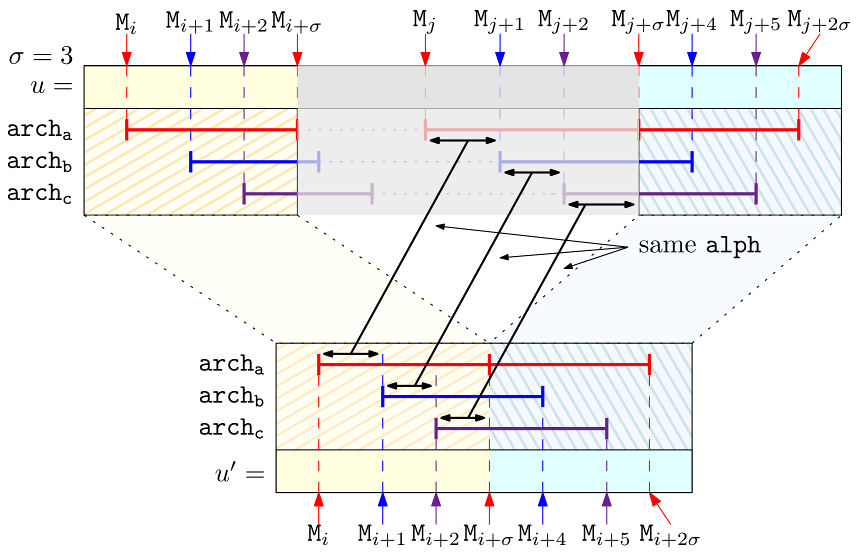


Figure 5.9: (Figure by Sungmin Kim) We can safely remove the substring $u[\texttt{M}_{i+\sigma}(u) + 1 : \texttt{M}_{j+\sigma}(u)]$ to obtain u' with the same γ and $\mathcal{R}$, and all $\mathcal{K}$ values are lower by $\frac{j-i}{\sigma}$.

Now, let $\mathcal{K}_2$ be the array of integers obtained by subtracting $\frac{j-i}{\sigma}$ from all values in $\mathcal{K}_1$. Then, $\texttt{s}(u[1 : \texttt{M}_{i+\sigma}(u)]u[\texttt{M}_{j+\sigma}(u) + 1 : |u|]) = (\gamma, \mathcal{K}_2, \mathcal{R})$. Since there can be naïvely 2^σ choices of $\texttt{alph}(u[\texttt{M}_{i+l}(u) + 1 : \texttt{M}_{i+l+1}(u)])$ for each $l \in [\sigma - 1] \cup \{0\}$, any string with $\iota(u) > (2^\sigma)^\sigma$ is guaranteed to have integers i and j that satisfy the above conditions by the pigeonhole principle. Therefore, we will reach a string w with $\iota(w) \leq (2^\sigma)^\sigma$ and $\texttt{s}(w) = (\gamma, \mathcal{K}_2, \mathcal{R})$, where $\mathcal{K}_1[t] - \mathcal{K}_2[t] = c$ for some non-negative constant c and all $t \in [\sigma]$ if we repeatedly apply the same argument to remove arches.

□

Lemma 5.4 limits the search space for the candidate string corresponding to a tuple $(\gamma, \mathcal{K}, \mathcal{R})$ by mapping valid subsequence universality signatures to subsequence universality signatures for strings with universality index at most $(2^\sigma)^\sigma$. Therefore, we need to investigate those strings where there are up to $\sigma \cdot (1 + (2^\sigma)^\sigma) + 1$ terms in its marginal sequence. The following lemma bounds the length of the substring between two consecutive marginal sequence terms in such a string. The conclusion of this line of thought follows then, in Corollary 5.6.

Lemma 5.5. *For a given string w, let $w = uvx$ where $v = w[\texttt{M}_i(w) + 1 : \texttt{M}_{i+1}(w)] \neq \varepsilon$, and $u = w[1 : \texttt{M}_i(w)]$, and $x = w[\texttt{M}_{i+1}(w) + 1 : |w|]$ for some integer $i \geq 1$. For a permutation v' of $\texttt{alph}(v)$ that ends with $v[|v|]$, we have $\texttt{s}(uvx) = \texttt{s}(uv'x)$.*

Proof. Since v is between two consecutive marginal sequence terms, all arches for any signature letter in uvx must end at a position no more than $|u|$ or at a position no less than $|uv|$ Arches that end at a position no more than $|u|$ in uvx will end at the same positions in $uv'x$ because they only depend on u. Suppose that an arch for signature letter $\mathtt{a}$ starts at a position no more than $|u|$ and ends at a position no less than $|uv|$ in uvx. Let i be the minimum non-negative integer that allows $\mathtt{alph}(\mathtt{r_a}(u)vx[1:i]) = \Sigma$. If $i \geq 1$, we have $\mathtt{alph}(\mathtt{r_a}(u)v) \neq \Sigma$. Since $\mathtt{alph}(\mathtt{r_a}(u)v) = \mathtt{alph}(\mathtt{r_a}(u)v')$, the minimum integer i' that allows $\mathtt{alph}(\mathtt{r_a}(u)v'x[1:i']) = \Sigma$ is equal to i. On the other hand, if $i = 0$, then we have $\mathtt{alph}(\mathtt{r_a}(u)v) = \Sigma$ and $\mathtt{alph}(\mathtt{r_a}(u)v[1:|v|-1]) \neq \Sigma$. Since we have $v'[|v'|] = v[|v|]$ and $\mathtt{alph}(v'[1:|v'|-1]) = \mathtt{alph}(v[1:|v|-1])$, the arch ends exactly at $|uv'|$ in $uv'x$. The number of arches that continues afterwards and the corresponding letters in the rest are thus equal for uvx and $uv'x$. Finally, even if v is in the rest of the arch factorization for a signature letter $\mathtt{a}$, we still have $\mathtt{alph}(\mathtt{r_a}(uvx)) = \mathtt{alph}(\mathtt{r_a}(uv'x))$ because $\mathtt{alph}(v) = \mathtt{alph}(v')$. □

Corollary 5.6. *The tuple $(\gamma, \mathcal{K}, \mathcal{R})$ is a valid subsequence universality signature if and only if there exists a string w of length at most $\sigma \cdot (\sigma \cdot (1 + (2^\sigma)^\sigma) + 1)$ and a constant $c \in \mathbb{N}_0$ that satisfies $\mathtt{s}(w) = (\gamma, \mathcal{K} - c, \mathcal{R})$.*

We can now show the following result.

Lemma 5.7. $\mathtt{MatchUniv}(\alpha, k)$ *is in* NP.

Proof. Follows from Proposition 5.3 and Corollary 5.6. Firstly, for a guessed sequence of universality signatures $(\gamma_x, \mathcal{K}_x, \mathcal{R}_x)$, for $x \in \mathtt{var}(\alpha)$, we check their validity. For that, we enumerate all strings of length up to the constant $\sigma \cdot (\sigma \cdot (1 + (2^\sigma)^\sigma) + 1)$ over Σ and see if there exist strings w_x such that $\mathtt{s}(w_x) = (\gamma_x, \mathcal{K}_x - c_x, \mathcal{R}_x)$ for some constant $c_x \leq k$. Since σ is constant, this takes polynomial time. We then use Proposition 5.3 to check if the guessed signatures lead to an assignment h of the variables such that $\iota(h(\alpha)) = k$, as already explained. Since we have a polynomial size bound on the certificate and a deterministic verifier that runs in polynomial time, we obtain that $\mathtt{MatchUniv}(\alpha, k)$ is in NP. □

5.2.3 Tractable classes of pattern in `MatchUniv`

Further, we describe two classes of patterns, defined by structural restrictions on the input patterns, for which `MatchUniv` can be solved in polynomial time.

Proposition 5.8. a) $\mathtt{MatchUniv}(\alpha, k)$ *is in* P *when there exists a variable that occurs only once in α. As such,* $\mathtt{MatchUniv}(\alpha, k)$ *is in* P *for the class of regular patterns (see, e.g., [59] and the references therein), where each variable occurs only once.* b) $\mathtt{MatchUniv}(\alpha, k)$ *is in* P *when* $|\mathtt{var}(\alpha)|$ *is constant.*

Proof. a) Let x be the variable that occurs only once in α. Then, we can uniquely rewrite $\alpha = \alpha_1 x \alpha_2$. We will successively define three substitutions h_1, h_2, h_3, all of which map variables that are not x to the empty string, i.e., $h_1(x') = h_2(x') = h_3(x') = \varepsilon$ for all $x' \in \mathcal{X} \setminus \{x\}$. Now, let $h_1(x) = \varepsilon$ as well. We claim that $k \geq \iota(h_1(\alpha))$ if and only if $\texttt{MatchUniv}(\alpha, k)$ is true. For any substitution h, we have $\iota(h_1(\alpha)) \leq \iota(h(\alpha))$ because $h_1(\alpha) \leq h(\alpha)$. Therefore, the problem is false if $k < \iota(h_1(\alpha))$. Moreover, if $k = \iota(h_1(\alpha))$, the problem is true by definition. Now, assume $k > \iota(h_1(\alpha))$ and let $h_2(x)$ be a permutation of $\Sigma \setminus \texttt{r}(h_2(\alpha_1))$. Then, $\iota(h_2(\alpha)) = \iota(h_2(\alpha_1)) + 1 + \iota(h_2(\alpha_2))$, because $\texttt{r}(h_2(\alpha_1 x)) = \varepsilon$. Note that we either have $\iota(h_2(\alpha)) = \iota(h_1(\alpha))$ or $\iota(h_1(\alpha)) + 1$. Finally, for an integer $i = k - \iota(h_2(\alpha))$, let $h_3(x) = h_2(x)\gamma^i$ where γ is a permutation of Σ. We now have $\iota(h_3(\alpha)) = \iota(h_3(\alpha_1)h_2(x)) + i + \iota(h_3(\alpha_2)) = i + \iota(h_2(\alpha))$.

Thus, for any $k \geq \iota(h_1(\alpha))$, there exists a substitution h such that $k = \iota(h(\alpha))$. We therefore compute $\iota(h_1(\alpha))$, the universality index of the image, and then return true if and only if $\iota(h_1(\alpha)) \leq k$, which can be done in polynomial time. □

b) The subsequence universality signature $\texttt{s}(h(x_i))$ of the image of some variable $x_i \in \texttt{var}(\alpha)$ under substitution h consists of three items, a permutation γ_i of a subset of Σ, an array $\mathcal{K}_i$ of σ integers, and an array $\mathcal{R}_i$ of σ subsets of Σ. Recall from the size estimation of such a universality signature that $\mathcal{K}_i$ can be represented with two integers l_i and k_i, where $\mathcal{K}_i[j] = k_i$ for all $j \in [l_i]$ and $\mathcal{K}_i[j] = k_i - 1$ for all $j \in [l_i + 1 : |\gamma_i|]$. Note that there are $\sum_{j=0}^{\sigma} j!$ choices for γ, at most σ choices for l_i, and $(2^\sigma)^\sigma$ choices for $\mathcal{R}$. Therefore, if we treat k_i as a variable whose value should be determined, we can enumerate for all possible assignments of γ_i, l_i, and $\mathcal{R}_i$ for all $i \in [|\texttt{var}(\alpha)|]$ in constant time under the assumption that σ and $|\texttt{var}(\alpha)|$ are constant.

Now, for a fixed set of γ_is, l_is, and $\mathcal{R}_i$s, we find the minimum value k'_i of k_i that validates $(\gamma_i, \mathcal{K}_i, \mathcal{R}_i)$ as a subsequence universality signature by enumerating all strings up to length $\sigma \cdot (\sigma \cdot (1 + (2^\sigma)^\sigma) + 1)$ using Corollary 5.6. If no such k'_i exists, we move on to the next set of γ_is, l_is, and $\mathcal{R}_i$s. Since Lemma 5.4 allows us to add an arbitrary number of arches while not altering γ_i and $\mathcal{R}_i$, we first assume that the number of additional arches is zero and compute how many more arches we need for $h(\alpha)$ to reach a universality index of k. We compute this number by counting the number of arches through an arch factorization on α. Specifically, for each rewriting $\alpha_1 x_i \alpha_2$, we compute the minimal $j \in [|\gamma|]$ that allows $\texttt{alph}(\texttt{r}(h(\alpha_1))) \cup \texttt{alph}(\gamma[1 : j]) = \Sigma$. If no such j exists, then we simply compute $\texttt{alph}(\texttt{r}(h(\alpha_1 x_i))) = \texttt{alph}(\texttt{r}(h(\alpha_1))) \cup \texttt{alph}(\gamma)$ without incrementing the number of arches. Then, if $j \leq l_i$, we add k'_i arches to the total arch count. If $j > l_i$, we add $k'_i - 1$ arches instead. Finally, we continue the arch factorization process with $\texttt{alph}(\texttt{r}(h(\alpha_1 x_i))) = \mathcal{R}_i[j]$. This way, we can compute the minimum number of arches the image of α can have for a fixed set of γ_is, l_is, and $\mathcal{R}_i$s.

Let d be the total number of additional arches we need for the image of α to reach a universality index of k. Note that an additional arch for each variable x_i will contribute to $|\alpha|_{x_i}$ more arches in the image of α. However, if the subsequence universality signature features γ_i with $\texttt{alph}(\gamma_i) \neq \Sigma$, then we cannot add any more arches for each variable. Let $I = \{i \in [|\texttt{var}(\alpha)|] \mid |\gamma_i| = \sigma\}$. Now, the problem boils down to finding how many additional arches we need for the image of each variable x_i with $i \in I$ while making the universality index of the image of α exactly k. Let d_i be the number of additional arches for each occurrence of variable x_i with $i \in I$. We can solve for d_is the following system of linear inequalities:

$$\begin{aligned} &\sum_{i \in I} |\alpha|_{x_i} d_i \le d, \\ &\sum_{i \in I} -|\alpha|_{x_i} d_i \le -d, \\ &-d_i \le 0 \ \forall i \in I \end{aligned}$$

Note that the first two inequalities imply $\sum_{i=1}^{|\texttt{var}(\alpha)|} |\alpha|_{x_i} d_i = d$ and the last $|\texttt{var}(\alpha)|$ inequalities enforce positive values for each d_i. If there is an integer solution for the system, then we can assign d_i more arches for the image of x_i, and the universality index of the image of α will be exactly k. Because $|\texttt{var}(\alpha)|$ is a constant, this system can be solved in time polynomial in $\log H$, where H is the maximum between k and the greatest coefficient of a variable in the above system (in absolute value) [93]. $\square$

5.3 Complexity of `MatchSimon`

Further, we discuss the `MatchSimon` problem. In the case of `MatchSimon`, we are given a pattern α, a word w, and a natural number $k \le n$, and we want to check the existence of a substitution h with $h(\alpha) \sim_k w$. The first result is immediate: `MatchSimon` is NP-hard, because `MatchSimon`$(\alpha, w, |w|)$ is equivalent to `Match`(α, w), and `Match` is NP-complete.

Lemma 5.9. `MatchSimon` *is* NP*-hard.*

Proof. We note that `MatchSimon`$(\alpha, w, |w|)$ is equivalent to the NP-complete `Match`(α, w). $\square$

To understand why this results followed much easier than the corresponding lower bound for `MatchUniv`, we note that in `MatchSimon` we only ask for $h(\alpha) \sim_k w$ and allow for $h(\alpha) \sim_{k+1} w$, while in `MatchUniv` $h(\alpha)$ has to be k-universal but not $(k+1)$-universal. So, in a sense, `MatchSimon` is not strict, while `MatchUniv` is strict. So, we can naturally consider the following problem.

Matching under Strict Simon's Congruence: `MatchStrictSimon`(α, w, k)

Input:	Pattern α, $|\alpha| = m$, word w, $|w| = n$, and $k \in [n]$.
Question:	Is there a substitution h with $h(\alpha) \sim_k w$ and $h(\alpha) \nsim_{k+1} w$?

Adapting the reduction from Lemma 5.2, we can show that `MatchStrictSimon` is NP-hard.

Lemma 5.10. `MatchStrictSimon` *is* NP-*hard.*

Proof. We refer to the notations from Lemma 5.2. We use the same reduction from `3CNFSAT` and note that α can either be mapped to a string $h(\alpha)$ with $\iota(h(\alpha)) \leq 5n+m+2$ (with equality only if the input instance of `3CNFSAT` is satisfiability) or to a string $h(\alpha)$ with $\iota(h(\alpha)) \leq (n+m)^6$. Therefore, consider the instance of `MatchStrictSimon` with input the pattern α constructed in the reduction, $w = (\mathtt{10}\,\$\,\#)^{5n+m+3}$, and $k = 5n+m+2$. Clearly, there exists a substitution h with $h(\alpha) \sim_k w$ and $h(\alpha) \nsim_{k+1} w$ if and only if there exists a substitution h with $\iota(h(\alpha)) = k$. Such a substitution exists if and only if the given instance of `3CNFSAT` is satisfiable. □

We can also show an NP-upper bound: it is enough to consider as candidates for the images of the variables under the substitution h only strings of length $O((k+1)^\sigma)$; longer strings can be replaced with shorter, $\sim_k$-congruent ones, which have the same impact on the sets $\mathbb{S}_k(h(\alpha))$. The following holds.

Theorem 5.11. `MatchSimon` *and* `MatchStrictSimon` *are* NP-*complete.*

Proof. By Lemmas 5.9 and 5.10, it is enough to show that both problems are in NP.

We make some observations first. Note that $\mathbb{S}_k(w_1w_2) = \Sigma^{\leq k} \cap \mathbb{S}_k(w_1)\mathbb{S}_k(w_2)$. Thus, for a pattern α and two substitutions h_1 and h_2 where $h_1(x) \sim_k h_2(x)$ for all variables $x \in \mathcal{X}$, we have $w \sim_k h_1(\alpha)$ if and only if $w \sim_k h_2(\alpha)$. Moreover, Kim et al. [103] showed that, for a given string w, the length of the shortest string in the set $\{u \in \Sigma^* \mid u \sim_k w\}$ is at most $\binom{k+\sigma}{\sigma} \leq k^\sigma$.

Based on these observations, we can now give NP-algorithms for both problems. We note that these problems reduce to the (exact) pattern matching problem when $k \geq |w|$. For `MatchSimon`, if $k \geq |w|$, we answer `MatchSimon`(α, w, k) positively if and only if α matches w. For `MatchStrictSimon`, if $k \geq |w|$, we always answer `MatchStrictSimon`(α, w, k) negatively. Indeed, if there exists h such that $h(\alpha) \sim_k w$, then $h(\alpha) \sim_{|w|} w$. It follows that $h(\alpha) = w$ and $h(\alpha) \sim_{k+1} w$, as well; the answer to `MatchStrictSimon`(α, w, k) should therefore be no.

Hence, from now on, we can assume that $k < |w|$.
Let us consider first the problem `MatchStrictSimon`. From the observations we have made at the beginning of this proof, and taking into account that we need to consider strings congruent under $\sim_{k+1}$, we can conclude that there exists a substitution h such that $h(\alpha) \sim_k w$ and $h(\alpha) \nsim_{k+1} w$ if and only if there exists such a substitution h where the length of the image of each variable is $(k+1)^\sigma$.

Since k is at most $|w|$ and σ is a constant, we only need to test certificates of polynomial length which encode the substitution. The verifier can then simply substitute the variables in the pattern, which will yield a string of length at most $(k+1)^\sigma|\alpha|$. Now, we can compute the largest ℓ for which

$h(\alpha) \sim_\ell w$ in $O(|h(\alpha)| + |w|) = O((|w| + 1)^\sigma|\alpha| + |w|)$ time [76], which is polynomial in the size of the input, under the assumption that σ is constant. If $\ell = k$, then we answer the respective instance positively. Therefore, the problem is in NP.

A similar argument holds for `MatchSimon` (but, in that case, it is enough to look for substitutions where the image of the variables is at most k^σ, as we only deal with $\sim_k$). On the other hand, note that the same argument cannot be applied to `MatchUniv`, because there is no bound on the size of k. This makes the size of the certificate, k^σ, exponentially large in $\log k$, which is the size of the encoding for a binary representation of k. □

Finally, note that `MatchSimon` and `MatchStrictSimon` are in P when the input pattern is regular.

Proposition 5.12. *If α is a regular pattern, then both of the problems* `MatchSimon`(α, w, k) *and* `MatchStrictSimon`(α, w, k) *are in* P.

Proof. We consider the problem `MatchSimon`(α, w, k). We assume that α is a regular pattern $\alpha = w_0x_1w_1 \cdots x_\ell w_\ell$, where, for $i \in [\ell]$, x_i is a variable and, for $i \in [\ell] \cup \{0\}$, w_i is a string of constants. The language $L(\alpha)$ of all words which can be obtained by replacing the variables of α by constant strings is regular, and we can construct in polynomial time a non-deterministic finite automaton N_α accepting it (it is the automaton accepting the language described by the regular expression $w_0\Sigma^*w_1 \cdots \Sigma^*w_\ell$). Now, using the results of [103], we can construct in polynomial time (when the size of the input alphabet σ is constant) a deterministic finite automaton $D_{w,k}$ accepting the words which are $\sim_k$ equivalent to w. Now, we simply check if there is a word accepted by both these automata (N_α and $D_{w,k}$), which can be done in polynomial time. We return the answer to this check as the answer to `MatchSimon`(α, w, k).

Further, we consider the problem `MatchStrictSimon`(α, w, k). Just as before, we construct the NFA N_α and the DFA $D_{w,k}$. Moreover, we construct the DFA $D_{w,k+1}$ and its complement $D'_{w,k+1}$ (which accepts the words which are not $\sim_{k+1}$ equivalent to w). Now, we see if there is a word accepted by N_α and $D_{w,k}$ and $D'_{w,k+1}$. Clearly, all steps can be done in polynomial time. We return the answer to this check as the answer to the problem `MatchStrictSimon`(α, w, k). □

5.4 Complexity of `WESimon`

In this section, we address the `WESimon` problem, where we are given two patterns α and β, and a natural number $k \leq n$, and we want to check the existence of a substitution h with $h(\alpha) \sim_k h(\beta)$. The first result is immediate: this problem is NP-hard because `MatchSimon`, which is a particular case of `WESimon`, is NP-hard.

To show that the problem is in NP, we need a more detailed analysis. If $k \leq |\alpha| + |\beta|$, the same proof as for the NP-membership of `MatchSimon` works: it is enough to look for substitutions of the variables with the image of each variable having length at most k^{σ}, and this is polynomial in the size of the input. If $k > |\alpha| + |\beta|$, and $\beta = w$ contains no variable, then this is an input for `MatchSimon` with k greater than the length of the input word w, and we have seen previously how this can be decided. Finally, if both α and β contain variables, then the problem is trivial, irrespective of k: the answer to any input is positive, as we simply have to map all variables to $(1 \cdots \sigma)^k$ and obtain two $\sim_k$-congruent words. Therefore, we have the following result.

Theorem 5.13. `WESimon` *is* NP-*complete.*

To avoid the trivial cases arising in the above analysis for `WESimon`, we can also consider a stricter variant of this problem:

Word Equations under Strict Simon's Congruence: `WEStrictSimon`(α, β, k)

Input:	Patterns α, β, $|\alpha| = m, \beta = n$, and $k \in [m + n]$.
Question:	Is there a substitution h with $h(\alpha) \sim_k h(\beta)$ and $h(\alpha) \not\sim_{k+1} h(\beta)$?

Differently from `WESimon`, we can show that this problem is NP-hard, even in the case when both sides of the pattern contain variables.

Lemma 5.14. `WEStrictSimon` *is* NP-*hard, even if both patterns contain variables.*

Proof. We refer to the notations from Lemma 5.2. We use the same reduction from `3CNFSAT` and note that α can either be mapped to a string $h(\alpha)$ with $\iota(h(\alpha)) \leq 5n + m + 2$ (with equality only if the input instance of `3CNFSAT` is satisfiability) or to a string $h(\alpha)$ with $\iota(h(\alpha)) \leq (n + m)^6$. Therefore, consider the instance of `WEStrictSimon` with input the pattern α constructed in the reduction, the second pattern $\beta = (\mathtt{10\,\$\,\#})^{5n+m+3}x$, where x is a fresh string-variable, and $k = 5n + m + 2$. Clearly, there exists a substitution h with $h(\alpha) \sim_k h(\beta)$ and $h(\alpha) \not\sim_{k+1} h(\beta)$ if and only if there exists a substitution h with $\iota(h(\alpha)) = k$. Such a substitution exists if and only if the given instance of `3CNFSAT` is satisfiable. □

Regarding the membership in NP: if k is upper bounded by a polynomial function in $|\alpha| + |\beta|$ (or, alternatively, if k is given in unary representation), then the fact that `WEStrictSimon` is in NP follows as in the case of `MatchStrictSimon`. The case when k is not upper bounded by a polynomial in $|\alpha| + |\beta|$ remains open. We can show the following theorem.

Theorem 5.15. `WEStrictSimon` *is* NP-*complete, for* $k \leq \mathit{poly}(|\alpha|, |\beta|)$.

A conclusion of all the results as well as an outlook on future work is presented in Section 7.

CHAPTER 6

The Edit Distance to k-Subsequence Universality

This chapter is based on article [50] (see reference below) and the LaTeX code of this article was used to reproduce it here. The Introduction and Preliminaries of this paper are worked into Chapter 1 and Chapter 2. Further, the notations are adjusted to match the other chapters and provide a uniform notation across this thesis.

Reference: J. D. Day, P. Fleischmann, M. Kosche, T. Koß, F. Manea, and S. Siemer. The edit distance to k-subsequence universality. In M. Bläser and B. Monmege, editors, *38th International Symposium on Theoretical Aspects of Computer Science, STACS 2021, March 16-19, 2021, Saarbrücken, Germany (Virtual Conference)*, volume 187 of *LIPIcs*, pages 25:1–25:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPICS.STACS.2021.25`

Description: We started to investigate for two words that are not $\sim_k$-congruent; what is the minimal number of edit operations (edits distance) that we need to perform on the first word to obtain two $\sim_k$-congruent words? We showed combinatoric and algorithmic results on the particular case of the second word being a k-universal word with a compatible alphabet.

Contribution: In this paper, I was a main contributor for developing the data structures and algorithms regarding changing the universality index of words via the insertion and substitution operations. Further, I presented this work at *STACS 2021* in Saarbrücken (Germany) and several workshops. A video of my talk can be found on YouTube under this URL (`https://www.youtube.com/watch?v=YkRy9WYW8EQ&ab_channel=SaarlandInformaticsCampus`).

6.1 Overview

As described in the introduction and preliminaries (Chapter 1 and 2), asymptotically optimal algorithms are known for deciding whether two words w and u are $\sim_k$-congruent. Thus, similarly to the case of other relations on strings (e.g., [11, 23]), it is natural to ask, for two words w and u, which are not $\sim_k$-congruent, what is the minimal number of edit operations (edits, for short) that we need to perform on them in order to obtain two $\sim_k$-congruent words. The edits we consider are the usual letter-insertion, -deletion, -substitution, and the scenario we assume is the following: we edit one word only (say w) and attempt to reach, with a minimal number of edits, an intermediate word which has the same subsequences of length k as the second input word (namely u). This formulation is essentially an instance of a word-to-language edit distance problem, in which we wish to compute the distance between w and the language $L_{u,k}$ of words which are $\sim_k$-congruent to u. It is well documented that word-to-language edit distance problems, alongside the classical word-to-word and also the language-to-language variants, are well motivated and have consequently been well studied (see, e.g., [162, 139, 86, 34, 92, 46, 47]).

In our case, the languages $L_{u,k}$ are regular. In particular, for a given subsequence v of length k of u, we can easily construct a DFA recognising the language of all words containing v as a subsequence. Consequently, a finite automaton accepting $L_{u,k}$ can be obtained as a boolean combination of these DFAs. In fact, for a positive integer k, the set of all languages which can be written as the union of several languages $L_{u,k}$, where u are words of a finite set, is the class of k-piecewise testable languages [156, 157, 123]. Therefore, if we take as input the word w and the language $L_{u,k}$ given as an automaton $A_{u,k}$ with q states, we can solve our distance problem in time $O(|w|q^2)$ [162, 6]. However, this is not necessarily efficient, since even when $A_{u,k}$ is a minimal NFAs accepting $L_{u,k}$, the number q of states can be exponential in the size of the alphabet (and hence in the length of u), see appendix of [50]. Consequently, if we consider the input to be (w, u, k) rather than $(w, A_{u,k})$, the exact complexity remains unclear. We can, however, guarantee inclusion in NP as we can trivially rewrite w into the shortest word $u \in L_{u,k}$ using at most $|w| + |u|$ edits.

It is also worth pointing out that the order of w and u in the input matters: the number of edits necessarily applied to w order to reach a word w' such that $w' \sim_k u$ holds, is not generally equal to the number of edits needed to apply on u in order to reach a word u' such that $u' \sim_k w$. Consider, for example, the words $w = aba$, $u = aaabbbaaa$, and $k = 2$. We need one insertion to transform w into $abab$, which is $\sim_2$-congruent to u, but we need two deletions to transform u into $aaabaaa$, which is $\sim_2$-congruent to w. An intuitive explanation for this is that w is closer w.r.t. the edit distance to the set $L_{u,k}$ of words which are $\sim_k$-congruent to u than u is to the set $L_{w,k}$ of words which are $\sim_k$-congruent to w, and we only need to edit each of our words until it reaches the word which is closest to them from the respective sets.

Essentially, we are considering the word-to-language edit distance problem for regular languages (in fact, piecewise testable languages) which admits a particularly succinct representation: a single word u. One way to generalise this is to consider the edit distance from a word to the closure of a given language under $\sim_k$. The problem remains decidable when considering the closures of regular or context-free languages (the regular case can be solved in nondeterministic polynomial time when k is a constant). On the other hand, we have already mentioned how taking the closure under $\sim_k$ can result in an exponential blow-up in the size of the representation of the language. Going in the other direction, one of the most natural restrictions is to consider only words u over an alphabet Σ for which all length-k subsequences over Σ occur, called *k-subsequence universal words* (called, for short, k-universal words) w.r.t. the alphabet Σ. This case is also among the ones for which the corresponding automata for $L_{u,k}$ may be exponentially large, remaining thus non-trivial. This restriction forms the focus of our work.

The focus of our work. In some cases, the problem introduced above admits an input-specification where the target language is defined in a way which is both easier-to-use and more succinct. One of these cases is the already mentioned language of k-subsequence universal words w.r.t. an alphabet $\Sigma = \{1, \ldots, \sigma\}$. While this language can be defined by a word $(1 \cdot 2 \cdots \sigma)^k$ of length $k\sigma$ or by an NFA with $\Theta(2^\sigma)$ states, it can also be simply specified by the number k and the alphabet Σ (or even only the size of this alphabet).

The main contribution of our work, described below, is the study of the following problem: given a word w and a number k, compute the minimum number of edits we need to apply to w in order to obtain a k-universal word w.r.t. $\mathrm{alph}(w)$ (see Section 6.6 for a discussion on why the alphabet Σ used in the definition of universality is chosen here to be the set $\mathrm{alph}(w)$ of letters occurring in the input word w). As such, we are interested in the edit distance from the input word w to the set of k-universal words w.r.t. $\mathrm{alph}(w)$. We give a series of efficient algorithms showing how to solve this problem.

This investigation seems interesting to us as the language of k-universal words plays an interesting role in the overall picture described in Section 1.2.

Our results. The maximum k for which a word w is k-universal is called *the universality index* of w, and denoted $\iota(w)$. Firstly, we note that when we want to increase the universality index of a word by edits, it is enough to use only insertions. Similarly, when we want to decrease the universality index of a word, it is enough to consider deletions. So, to measure the edit distance to the class of k-subsequence universal words, for a given k, it is enough to consider either insertions or deletions. However, changing the universality of a word by substitutions (both increasing and decreasing it) is interesting in itself as one can see the minimal number of substitutions needed to transform a word w into a k-universal word as the *Hamming distance* [85] between w and the set of k-universal words. Thus, we consider all these operations independently and propose efficient algorithms computing the minimal number of insertions, deletions, and substitutions, respectively, needed to apply to a given word w in order to reach the class of k-universal words (w.r.t. the alphabet of w), for a given k. The time needed to compute these numbers is $O(nk)$ in the case of deletions and substitutions, as well as in the case of insertions when $k \leq n$ (for larger values of k it is just the time complexity of computing $k\sigma - n$, which is the value of the distance in that case). These algorithms are presented in the Section 6.3, and work in optimal linear time for constant k.

These algorithms are based, like most edit distance algorithms, on a dynamic programming approach. However, implementing such an approach within the time complexities stated above does not seem to follow directly from the known results on the word-to-word or word-to-language edit distance. In particular, we do not explicitly construct any k-universal word nor any representation (e.g., automaton or grammar) of the set of k-universal words, when computing the distance from the input word w to this set. Rather, we obtain the k-universal word which is closest w.r.t. edit distance to w as a byproduct of our algorithms. In our approach, we first develop (Section 6.2) several efficient data structures (most notably Lemma 6.5). Then (Section 6.3), for each of the considered operations, we make several combinatorial observations, allowing us to restrict the search space of our algorithms, and creating a framework where our data structures can be used efficiently.

Finally (in Section 6.5), we give algorithms running in $(n \log^{O(1)} \sigma)$-time computing the minimum number of insertions (respectively, substitutions) we need to apply to w in order to obtain a k-universal word, with $k > \iota(w)$. These algorithms rely heavily on the fact that computing the edit distance to k-universality can be reformulated, in this case, as computing the path of length k of minimum weight in a weighted DAG with the Monge property. In particular, these algorithms provide optimal linear-time solutions for our problem in the case of increasing the universality-index of words over constant-size alphabets.

6.2 Problem specific Toolbox

In the first part of this section, we present data structures which will be decisive in obtaining efficient solutions for the approached problems. In the second part of the section, we give examples for the aforementioned data structures. Our running example will be the word $w = \texttt{bananaban}$, on which we illustrate some of the notions we define here.

6.2.1 Algorithms and Data Structures

For a word w over an alphabet Σ, a position j of w, and a letter $a \in \Sigma$ which occurs in $w[1 : j]$, let $\text{last}_j[a] = \max\{i \leq j \mid w[i] = a\}$, the last position where a occurs before j; if a does not occur in $w[1 : j]$ or for $j = 0$, then, by convention, $\text{last}_j[a] = |w| + 1$. Let $S_j = \{\text{last}_j[a] \mid a \in \text{alph}(w[1 : j])\}$. If i, j are two positions of w, let $\Delta(i, j)$ be the number of distinct letters occurring in $w[i : j]$, i.e., $\Delta(i, j) = |\text{alph}(w[i : j])|$; if $i > j$, then $\Delta(i, j) = 0$. For a position i of w, and a letter $a \in \Sigma$, let $d_i[a] = \Delta(\text{last}_i[a], i)$.

Lemma 6.1. *Let w be a word, with $|w| = n$, alph$(w) = \Sigma$, and $\Sigma = \{1, 2, \ldots, \sigma\}$. We can compute in $O(n)$ the values $\Delta(1, \ell)$, for all $\ell \in [1 : n]$.*

Proof. We define an array $C[1 : \sigma]$, whose elements are initialised with 0 and $f = 0$. Now, we will traverse the positions of the word left to right. When we reach position ℓ, we do the following. If $C[w[\ell]] = 0$, then we set $C[w[\ell]] = 1$ and we increment f by 1. We set $\Delta(1, \ell) = f$. □

The pseudocode for this algorithm is given in Algorithm 1.

Algorithm 1: Calculation of $\Delta(1, i)$ for all $i \in [1 : n]$

```
Input  : word $w$, alphabet $\Sigma$
Output : $\Delta(i - \sigma + 1, i)$ for $i \in [\sigma : n]$
// initialization
int $f \leftarrow 0$; int $n \leftarrow |w|$; int $\sigma \leftarrow |\Sigma|$;
int $C[1 : \sigma] = 0$;
for int $i \leftarrow 1$ to $n$ do
    if $C[w[i]] = 0$ then
        $f \leftarrow f + 1$;
    end
    $C[w[i]] \leftarrow C[w[i]] + 1$;
    $\Delta(1, i) \leftarrow f$;
end
```

Lemma 6.2. *Let w be a word, with $|w| = n$, alph$(w) = \Sigma$, and $\Sigma = \{1, 2, \ldots, \sigma\}$. We can compute in $O(n)$ the values $\Delta(i - \sigma + 1, i)$, for all $i \in [\sigma : n]$.*

Proof.
We define an array $C[1 : \sigma]$, whose elements are initialised with 0 and $f = 0$. Now, we will traverse the positions of the word left to right as shown in 2. When we reach position i, we do the following. If $C[w[i]] = 0$, we increment f by 1 since we saw a new letter. Then, $C[w[i]]$ is incremented by 1. When $i = \sigma$, we set $\Delta(1, i) = f$. When $i > \sigma$, we decrement $C[w[i - \sigma]]$ by one, and if the value of $C[w[i - \sigma]]$ is now 0, meaning that it does not occur in $w[i - \sigma + 1 : i]$, we decrement f by 1. Finally, $\Delta(i - \sigma + 1, i)$ is set to f. □

The pseudocode for this algorithm is given in Algorithm 2.

Lemma 6.3. *Let w be a word, with $|w| = n$, alph$(w) = \Sigma$, and $\Sigma = \{1, 2, \ldots, \sigma\}$. We can compute in $O(n)$ time $\text{last}_{j\sigma+1}[a]$ and $d_{j\sigma+1}[a]$, for all $a \in \Sigma$ and all integers $1 \leq j \leq (n - 1)/\sigma$.*

Proof. We define an array $C[1 : \sigma]$, whose elements are initialised with ∞, a variable $f = 0$, as well as a doubly linked list R, which will have at most σ elements from $[1 : \sigma]$, and is empty at the beginning, and an array $P[1 : \sigma]$ of pointers to the elements of R (initially they are all set to an undefined value ∞).

Algorithm 2: Calculation of $\Delta(i-\sigma+1, i)$ for all $i \in [\sigma : n]$

```
Input  : word $w$, alphabet $\Sigma$
Output : $\Delta(i-\sigma+1, i)$ for $i \in [\sigma : n]$
// initialization
int $f \leftarrow 0$; int $n \leftarrow |w|$; int $\sigma \leftarrow |\Sigma|$;
int $C[1 : \sigma] = 0$;
for int $i \leftarrow 1$ to $n$ do
    if $C[w[i]] = 0$ then
        $f \leftarrow f + 1$;
    end
    $C[w[i]] \leftarrow C[w[i]] + 1$;
    if $i = \sigma$ then
        $\Delta(1, i) \leftarrow f$;
    else if $i > \sigma$ then
        $C[w[i-\sigma]] \leftarrow C[w[i-\sigma]] - 1$;
        if $C[w[i-\sigma]] = 0$ then
            $f \leftarrow f - 1$;
        end
        $\Delta(i-\sigma+1, i) \leftarrow f$;
    end
end
```

Now, we will traverse the positions of the word w left to right. So, we consider each position i of w, for i from 1 to $\left\lfloor \frac{n-1}{\sigma} \right\rfloor \sigma + 1$ (the largest number of the form $j\sigma + 1$, smaller or equal to n).

For position i, we do the following three steps.

1. If $C[w[i]] = \infty$ then we increment f by 1. Both when $C[w[i]] = \infty$ or $C[w[i]] \neq \infty$ we set $C[w[i]] = i$. Now, $C[a]$ is the last occurrence of the letter $a \in \Sigma$ in the word $w[1 : i]$, for all $a \in \Sigma$ (or ∞ if a does not occur in $w[1 : i]$). Also, f is the number of elements of C which are not equal to ∞, so the number of distinct letters we have seen in $w[1 : i]$.

2. If $w[i]$ does not occur in R (tested by checking if $P[w[i]] = \infty$ or not), insert $w[i]$ at the end of R (and set $P[w[i]]$ to point at the node where $w[i]$ occurs, which we have just created). If $w[i]$ occurs in R, remove $w[i]$ from R (the place where $w[i]$ is stored in R is $P[w[i]]$), insert $w[i]$ at the end of R, and update $P[w[i]]$ to point to this node of R. Now R contains in order (from the front to the end) the f distinct letters occurring in $w[1 : i]$ ordered increasingly by the position of their last occurrence, and for each letter a occurring in $w[1 : i]$, $P[a]$ is a pointer to the node containing a of R.

3. If $i = j\sigma + 1$ for some j, we need to run the following two special steps. Firstly, we define the array $\text{last}_{j\sigma+1}[\cdot]$, by setting $\text{last}_{j\sigma+1}[a] = C[a]$ for all $a \in \Sigma$. Secondly, we set $g = f$; then, for each element e in R, in the order in which these elements occur when traversing R left to right, we set $d_{j\sigma+1}[e] = g$ and decrement g by 1.

It is not hard to see that the arrays $\text{last}_{j\sigma+1}[\cdot]$ and $d_{j\sigma+1}[\cdot]$ are correctly computed for all $j \le (n-1)/\sigma$. The overall time needed to maintain the array C is linear, while computing each of the arrays $\text{last}_{j\sigma+1}[1:\sigma]$ and $d_{j\sigma+1}[1:\sigma]$, for $j \le n/\sigma$, takes $O(\sigma)$ time. However, as we only need to compute such arrays $O(n/\sigma)$ times, the overall time needed to compute them is $O(n)$. So, the statement holds. □

The pseudocode for this algorithm is given in Algorithm 3.

Algorithm 3: Calculation of $\text{last}_{j\sigma+1}$, $d_{j\sigma+1}$

```
Input   : word $w$, alphabet $\Sigma$
Output : $\text{last}_{j\sigma+1}$, $d_{j\sigma+1}$ for $j \le (n-1)/\sigma$
// initialization
int $f = 0$; int $n \leftarrow |w|$; int $\sigma \leftarrow |\Sigma|$;
int $C[1:\sigma] = \infty$; doubly linked list $R$; pointer array $P[1:n]$;
for $i = 1$ to $\left\lfloor \frac{n-1}{\sigma} \right\rfloor \sigma + 1$ do
    // step one
    if $C[w[i]] == \infty$ then
        $f \leftarrow f + 1$;
    end
    $C[w[i]] \leftarrow i$;
    // step two
    if $P[w[i]] \ne null$ then
        remove the element in R at pointer $P[w[i]]$;
    end
    add $w[i]$ to the right end of $R$;
    $P[w[i]] \leftarrow$ adress of $R[w[i]]$;
    // step three
    if $i \mod \sigma = 1$ then
        $j \leftarrow (i-1)/\sigma$;
        foreach $a \in \Sigma$ do
            $\text{last}_{j\sigma+1}[a] \leftarrow C[a]$;
        end
        int $g \leftarrow f$;
        foreach $e \in R$ do
            $d_{j\sigma+1}[e] = g$;
            $g \leftarrow g - 1$;
        end
    end
end
```

For w = bananaban, we have $|w| = 9$ and $\sigma = 3$. In Lemma 6.1 we compute $\Delta(1,1) = 1$, $\Delta(1,2) = 2$, and $\Delta(1,\ell) = 3$ for $\ell \in [3:9]$. In Lemma 6.2 we compute $\Delta(1,3) = 3$, $\Delta(2,4) = \Delta(3,5) = \Delta(4,6) = 2$, $\Delta(5,7) = 3$, $\Delta(6,8) = 2$, and $\Delta(7,9) = 3$. In Lemma 6.3 we compute the arrays $\text{last}_4[\cdot]$ and $\text{last}_7[\cdot]$. We get: $\text{last}_4[\texttt{a}] = 4$, $\text{last}_4[\texttt{b}] = 1$, $\text{last}_4[\texttt{n}] = 3$, and $\text{last}_7[\texttt{a}] = 6$, $\text{last}_7[\texttt{b}] = 7$, $\text{last}_7[\texttt{n}] = 5$. Therefore, $S_4 = \{1,3,4\}$, $S_7 = \{5,6,7\}$, and $d_4[\texttt{a}] = 1$, $d_4[\texttt{b}] = 3$, $d_4[\texttt{n}] = 2$, $d_7[\texttt{a}] = 2$, $d_7[\texttt{b}] = 1$, $d_7[\texttt{n}] = 3$.

For a word w and a position i of w, let $\text{univ}[i] = \max\{j \mid w[j:i] \text{ is universal}\}$. That is, for the position i we compute the shortest universal word ending on that position. If there is no universal word ending on position i we set $\text{univ}[i] = 0$.

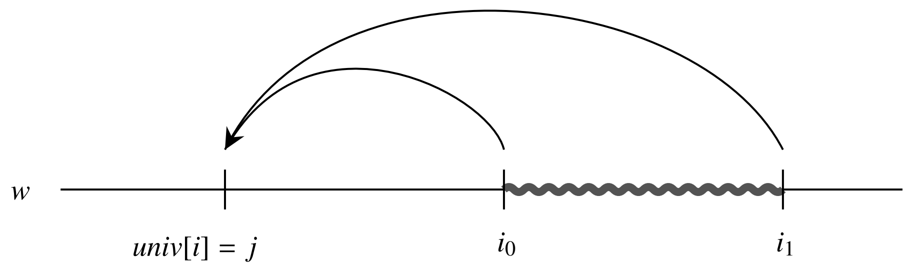


Figure 6.1: The set of elements with $\mathrm{univ}[i] = j$ forms an interval: $L_j = [i_0 : i_1]$.

Further, if $n = |w|$, let $V_w = \{\mathrm{univ}[i] \mid 1 \leq i \leq n\}$. In V_w we collect the starting positions of the shortest universal words ending at each position of the word w. Now, for $j \in V_w$, let $L_j = \{i \mid \mathrm{univ}[i] = j\}$; in other words, we group together the positions i of w for which the shortest universal word ending on i starts on some position j. Note that $L_0 = \{i \mid w[1 : i] \text{ is not universal}\}$, i.e., the positions of w where no universal word ends.

Several observations are immediate: for $i \in L_j$, $i' \in L_{j'}$, we have $i \leq i'$ if and only if $j \leq j'$. As each position i of w belongs to a set L_j, for some $j \in V_w$, we get that $\{L_j \mid j \in V_w\}$ is a partition of $[1 : n]$ into intervals. Furthermore, $w[i] \neq w[j]$ for all $i \in L_j$ and $j \neq 0$: if $w[i]$ would be the same as $w[j]$ then $w[j+1 : i]$ would also be a universal word, so i would not be in L_j. Also, if $i = \max(L_j)$ for some $j > 0$ then $w[i+1] = w[j]$. Indeed, there exists $j' \in [j+1 : i]$ such that $w[j' : i+1]$ is universal. But $w[j]$ does not occur in $w[j' : i]$, so $w[j] = w[i+1]$ must hold.

Further, we define for all positions i of w the value $\mathrm{freq}[i] = |w[1 : i]|_{w[i]}$, the number of occurrences of $w[i]$ in $w[1 : i]$. Also, let $T[i] = \min\{|w[i+1 : n]|_a \mid a \in \Sigma\}$, for $i \in [0, n-1]$, be the least number of occurrences of a letter in $w[i+1 : n]$; set $T[n] = 0$.

Lemma 6.4. *Let w be a word, with $|w| = n$, $alph(w) = \Sigma$, and $\Sigma = \{1, 2, \ldots, \sigma\}$. We can compute in $O(n)$ time the following data structures:* 1. *the array* $\mathrm{univ}[\cdot]$*;* 2. *the set V_w and the lists L_j, for all $j \in V_w \setminus \{0\}$;* 3. *the array* $\mathrm{freq}[\cdot]$*;* 4. *the array* $T[\cdot]$*;* 5. *the values* $\mathrm{last}_{j-1}[w[i]]$*, for all $j \in V_w$ and all $i \in L_j$;* 6. *the values* $\mathrm{last}_{i-1}[w[i]]$*, for all $i \in [2 : n]$.*

Proof. We present first an algorithm for items 1 and 2, then an algorithm for items 3 and 4, and finally an algorithm for items 5 and 6.

Algorithm for 1,2. (pseudocode: Algorithm 4) To compute univ, V_w, and the lists L_j we will use a *two pointer-strategy*. That is, we go through the positions of w from n to 1 with two pointers a and b. Initially, $a = n$ and $b = n + 1$. We also define an array C with σ elements, all initially set to 0, and a variable f, initialised with 0. The elements of the array univ are initialised with 0.

Now we repeat the following three-step procedure until b is 0.

In the first step, we execute the following loop. While $b > 1$ and $f < \sigma$, we do the following: decrement b by 1, increment $C[w[b]]$ by 1 and if $C[w[b]] = 1$ then increment f by 1, too.

In the second step, when the loop finished, if $f < \sigma$ and $b = 1$, then set $b = 0$. If $f = \sigma$, then we add b to V_w.

In the third step, while $f = \sigma$, we do the following steps. First, set $\text{univ}[a] = b$ and store a in L_b. Decrement $C[w[a]]$ by 1. Now, if $C[w[a]] = 0$, then decrement f by 1, too. Finally, decrement a by 1.

The idea is relatively simple: we start with the factor $w[a]$ and then try to extend it to the left (i.e., produce the factors $w[b : a]$ with $b \leq a$) while keeping track in f how many different letters of Σ we met (i.e., $f = \Delta[b : a]$), and their respective counts in the array $C[\cdot]$. As soon as we have seen all letters (i.e., $f = \sigma$), we know that $w[b : a]$ is the shortest universal word ending on a. Thus, we can store b in V_w, and set $\text{univ}[a] = b$. Now, we try to identify all the universal words starting on b, ending to the left of a (their respective ending positions and a are exactly the elements of L_b). This is done similarly: we move the pointer a now to the left one position at a time, and see which letters of Σ are removed from the word $w[b : a]$, using the array C where we counted the letter-occurrences. As soon as we have a letter that occurs 0 times in $w[b : a]$ we stop, as $w[b : a]$ is no longer universal. We then repeat the procedure: move b to the left first till we again have that $w[b : a]$ is universal, then move a to the left till $w[b : a]$ is no longer universal, and so on. By the observations we made on the structure of the lists L_j, this approach is clearly correct.

The complexity is linear, as each pointer visits each position of w once.

Algorithm for 3,4. (pseudocode: Algorithm 5) To compute $\text{freq}[\cdot]$ we use a straightforward strategy. We use an array $C[\cdot]$ with σ elements, initially set to 0. Then, for i from 1 to n we increment $C[w[i]]$ by 1 and set $\text{freq}[i] = C[w[i]]$. This clearly takes linear time. At the end of this traversal of the word, $C[a]$ is the number of occurrences of a in w. We will show now how the array T is computed. Let x be the letter of Σ such that $C[x]$ is the smallest value of C and $m = C[x]$. We set $T[0] = m$. Now, for i from 1 to $n - 1$ we do the following three steps. Firstly, we decrement $C[w[i]]$ by 1. Secondly, if $C[w[i]] < m$ then we set $m = C[w[i]]$ and $x = w[i]$. Thirdly, we set $T[i] = m$. It is immediate that T is correctly computed and that the computation takes linear time.

Algorithm for 5,6. (pseudocode: Algorithm 6) For the computation of all the values $\text{last}_{j-1}[w[i]]$, for all $j \in V_w$ and all $i \in L_j$, and the values $\text{last}_{i-1}[w[i]]$, for all $i \in [2 : n]$, we do the following. We use an array $L[\cdot]$ with σ elements, initially set to $n + 1$. Then, for i from 0 to n we do the following two steps. If $i > 0$, set $\text{last}_{i-1}[w[i]] = L[w[i]]$ and $L[w[i]] = i$. If $i + 1 \in V_w$, then we go through the elements e of list L_{i+1} and set $\text{last}_i[w[e]] = L[w[e]]$. This takes linear time, as the time needed to execute the iteration of the loop for some i is either $O(1)$ if $i + 1 \notin V_w$ or $1 + O(|L_{i+1}|)$ if $i + 1 \in V_w$. This adds up to $O(\sum_{j\in V_w} |L_j|) = O(n)$.

Algorithm 4: Calculation of *univ*, V_w, L_j

```
Input  : word $w$, alphabet $\Sigma$
Output : $univ$, $V_w$, $L_j$ for $j \leq \sigma$
// initialization
int $f \leftarrow 0$; int $a \leftarrow n$; int $b \leftarrow n + 1$; int $n \leftarrow |w|$; int $\sigma \leftarrow |\Sigma|$;
int $C[1 : \sigma] = 0$, univ$[1 : n] = 0$;
while $b > 0$ do
    // step one
    while $f < \sigma$ and $b > 1$ do
        $b \leftarrow b - 1$;
        if $C[w[b]] = 0$ then
            $f \leftarrow f + 1$;
        end
        $C[w[b]] \leftarrow C[w[b]] + 1$;
    end
    // step two
    if $f = \sigma$ then
        $V_w$.add($b$);
    else if $f < \sigma$ and $b = 1$ then
        $b \leftarrow 0$;
    end
    // step three
    while $f = \sigma$ do
        $univ[a] = b$;
        $L_b$.add($a$);
        $C[w[a]] \leftarrow C[w[a]] - 1$;
        if $C[w[a]] = 0$ then
            $f \leftarrow f - 1$;
        end
        $a \leftarrow a - 1$;
    end
end
```

It is important to note here that each last$_i[\cdot]$ is implemented as a list (implemented statically) with exactly one element if $i + 1 \notin V_w$ (i.e., last$_i[w[i + 1]]$) and exactly $|L_{i+1} \cup \{w[i + 1]\}|$ elements (i.e., last$_i[w[i + 1]]$ and last$_i[w[j]]$ with $j \in L_{i+1}$). In the second case, the list is implemented as an array, indexed by the the letters in $L_{i+1} \cup \{w[i + 1]\}$.

This concludes our proof. □

Consider again $w = \texttt{bananaban}$. In Lemma 6.4 we compute the following values. Firstly, univ[1] = univ[2] = 0, univ[ℓ] = 1 for $\ell \in [3 : 6]$, univ[7] = univ[8] = 5, univ[9] = 7. Thus, $V_w = \{0, 1, 5, 7\}$ and $L_0 = [1 : 2]$, $L_1 = [3 : 6]$, $L_5 = [7 : 8]$, $L_7 = [9 : 9]$. Secondly, freq[1] = 1, freq[2] = freq[3] = 1, freq[4] = freq[5] = 2, freq[6] = 3, freq[7] = 2, freq[8] = 4, freq[9] = 3. Moreover, $T[0] = 2$, $T[\ell] = 1$ for $\ell \in [1 : 6]$, and $T[\ell] = 0$ for $\ell \in [7 : 9]$. Then, for $j = 1$, we have last$_0[a] = 0$, for $a \in \{\texttt{a}, \texttt{b}, \texttt{n}\}$; for $j = 5$, we have last$_4$[b] = 1 and last$_4$[a] = 4; for $j = 7$, we have last$_6$[n] = 5. Finally, last$_0$[b] = 10, last$_1$[a] = 10, last$_2$[n] = 10, last$_3$[a] = 2, last$_4$[n] = 3, last$_5$[a] = 4, last$_6$[b] = 1, last$_7$[a] = 6, last$_8$[n] = 5.

Algorithm 5: Calculation of $freq$ and T

```
Input  : word $w$, alphabet $\Sigma$
Output : $freq$, $T$
// initialization
int $n \leftarrow |w|$; int $\sigma \leftarrow |\Sigma|$;
int $C[1:\sigma] = 0$; int $freq[n]$; int $T[n]$;
for $i \leftarrow 1$ to $n$ do
    $C[w[i]] \leftarrow C[w[i]] + 1$;
    $freq[i] = C[w[i]]$;
end
int $x$;
int $m \leftarrow n + 1$;
for $i \leftarrow 1$ to $\sigma$ do
    if $C[w[i]] < m$ then
        $m \leftarrow C[w[i]]$;
        $x \leftarrow w[i]$;
    end
end
$T[0] \leftarrow m$;
for $i \leftarrow 1$ to $n - 1$ do
    // step one
    $C[w[i]] \leftarrow C[w[i]] - 1$;
    // step two
    if $C[w[i]] < m$ then
        $m \leftarrow C[w[i]]$;
        $x \leftarrow w[i]$;
    end
    // step three
    $T[i] \leftarrow m$;
end
```

The main idea behind proving Lemmas 6.1, 6.3, and 6.4 is to traverse the word w left to right (or, respectively, right to left) and maintain the number of occurrences, as well as the last occurrence, of each letter in the prefix (respectively, suffix) of w that we have visited so far. For Lemma 6.2, we only consider a sliding window of size σ which traverses the word left-to-right, while maintaining similar data as before, but only for the content of the window. In all cases, this requires linear time and enables us to construct the desired data structures.

Together with the string-processing data structures we defined above and in section 2.7, we build the following general technical data structures lemma. This lemma (combined with some combinatorial observations) will be used to speed up some of our dynamic programming algorithms.

In this lemma we process a list A which initially has σ elements, and in which we insert, in successive steps, σ new elements, by appending them always at the same end. For simplicity, we can assume that the list A is a sequence with 2σ elements (denoted $A[i]$, with $i \in [1 : 2\sigma]$), out of which the last σ are initially undefined. The i^{th} insertion would, consequently, mean setting $A[\sigma + i]$ to the actual value that we want to insert in the list A. In our lemma we will also repeatedly perform an operation which decrements the values of some elements of the list A. However, we will not require to be able to explicitly access, after every operation, all the elements of the list (so we will not need to retrieve the values $A[i]$). Consequently, we will not maintain explicitly the

Algorithm 6: Calculation of $\text{last}_i[w[j]]$ for $i+1 \in V_w$ and $j \in L_{i+1}$, and $\text{last}_i[w[i+1]]$ for $i \leq n$.

```
Input  : word $w$, alphabet $\Sigma$, $V_w$, lists $L_j$ for $j \in V_w$
Output : values $\text{last}_{j-1}[w[i]]$ for all $j \in V_w$ and $i \in L_j$, values $\text{last}_{i-1}[w[i]]$ for all $i \leq n$
// initialization
int $n \leftarrow |w|$; int $\sigma \leftarrow |\Sigma|$;
int $L[1:\sigma] = n+1$;
for $i \leftarrow 0$ to $n-1$ do
    if $i+1 \in V_w$ then
        foreach $e \in L_{i+1}$ do
            $\text{last}_i[w[e]] \leftarrow L[w[e]]$;
        end
    end
    if $i < n$ then
        $\text{last}_i[w[i+1]] \leftarrow L[w[i+1]]$;
        $L[w[i+1]] \leftarrow i+1$;
    end
end
```

value of all the elements of A (that is, we will not update the elements affected by decrements). We are only interested in being able to retrieve (by value and position), at each moment, the smallest element and the last element of A. Thus, throughout the computation, we only maintain a subset of important elements of A, including the aforementioned two. We can now state our result, whose proof is based on Lemma 2.35.

Lemma 6.5. *Let A be a list with σ elements (natural numbers) and let $m = \sigma$. We can execute (in order) the sequence of σ operations $o_1, \ldots, o_\sigma$ on A in overall $O(\sigma)$ time, where o_i consists of the following three steps, for $i \in [1:m]$:*

1. *Return $e = \arg\min\{A[i] \mid i \in [1:m]\}$ and $A[e]$.*
2. *For some $j_i \in [1:m]$, decrement all elements $A[j_i], A[j_i+1], \ldots, A[m]$ by 1.*
3. *For some natural number x_i, append the element x_i to A (i.e., set $A[m+1]$ to x_i), and increment m by 1 (i.e., set m to $m+1$).*

Proof. Firstly, we will *run a preprocessing* of A.

We begin by defining recursively a finite sequence of positions as follows:

- a_1 is the rightmost position of A on which $\min\{A[i] \mid i \in [1:\sigma]\}$ occurs;
- for $i \geq 2$, if $a_{i-1} < \sigma$, then a_i is the rightmost position on which $\min\{A[i] \mid i \in [a_{i-1}+1:\sigma]\}$ occurs;
- for $i \geq 2$, if $a_{i-1} = \sigma$, then we can stop, our sequence will have $i-1$ elements.

Let p be the number of elements in the sequence defined above, i.e., our sequence is $a_1, \ldots, a_p$. For convenience, let $a_0 = 0$. Then the sequence $a_1, \ldots, a_p$ fulfils the following properties:

- $a_p = \sigma$ and $a_i > a_{i-1}$, for all $i \in [1 : p]$;
- $A[a_i] > A[a_{i-1}]$ for all $i \in [2 : p]$;
- for all $i \in [1 : p]$, we have $A[a_i] < A[t]$, for all $t \in [a_i + 1 : \sigma]$;
- for all $i \in [1 : p]$, we have $A[a_i] \leq A[t]$, for all $t \in [a_{i-1} + 1 : a_i]$.

By definition, for $i \in [1 : p]$ we have $A[a_i] = \min\{A[t] \mid t \in [a_{i-1} + 1 : \sigma]\}$, $A[a_i] < \min\{A[t] \mid t \in [a_i + 1 : \sigma]\}$, and $a_1 = \min\{A[i] \mid i \in [1 : \sigma]\}$. Clearly, we have $a_p = \sigma$.

The positions $a_1, \dots, a_p$ can be computed in linear time $O(\sigma)$, in reversed order. As we do not know from the beginning the value of p, we will compute a sequence $b_1, b_2, \dots$ of positions as follows. We start with $b_1 = \sigma$, $t = \sigma - 1$, and $i = 2$. Then, while $t \geq 1$ we do the following case analysis. If $A[t] < b_{i-1}$, then set $b_i = t$, increment i by 1, and decrement t by 1. Otherwise, if $A[t] \geq b_{i-1}$, just decrement t by 1. It is straightforward that this process takes $O(\sigma)$ time, and, when we have finished it, the number i is exactly the number p, and $a_i = b_{p-i+1}$.

Another observation is that, for $a_0 = 0$, the intervals $[a_{i-1} + 1, a_i]$, for $i \in [1, p]$, define a partition of the interval $[1 : \sigma]$ into p intervals. Therefore, we can define a partition of the interval $[1 : 2\sigma]$ into the intervals $[a_{i-1} + 1 : a_i]$, for $i \in [1, p]$, and $[t : t]$, for $t \in [\sigma + 1 : 2\sigma]$. Thus, we construct in linear time, according to Lemma 2.34, an interval union-find data structure for the interval $[1 : 2\sigma]$, as induced by the intervals $[1 : a_1]$, $[a_1 + 1 : a_2], \dots [a_{p-1}, a_p]$, $[\sigma + 1 : \sigma + 1]$, $[\sigma + 2 : \sigma + 2], \dots$ $[2\sigma : 2\sigma]$.

Let us now take $m = \sigma$ (and assume the convention $A[0] = 0$). We associate as satellite data to each interval $[x : y]$ with $y \leq m$ from our interval union-find data structure the value $A[y] - A[x - 1]$.

This entire preprocessing takes clearly $O(\sigma)$ time.

In order to explain how the operations are implemented, we assume as invariant that the following properties are fulfilled before o_i is executed, for $i \in [1 : \sigma]$:

- A contains m elements;
- all intervals $[x : y]$ with $y > m$ from our interval union-find data structure are singletons (i.e., $x = y$);
- for each interval $[x : y]$ with $y \leq m$, we have the associated satellite data $A[y] - A[x - 1]$;
- for each interval $[x : y]$ with $y \leq m$, we have that $A[y] \leq A[t]$ for $t \in [x : m]$ and $A[y] < A[t]$ for $t \in [y + 1 : m]$;
- we have stored in a variable ℓ the value $A[m]$.

This clearly holds after the preprocessing step, so before executing o_1.

Let us now explain *how the operation o_i is executed.*

The first step of o_i is to return $e = \min\{A[i] \mid i \in [1 : m]\}$ and i_e the rightmost position of the list A such that $A[i_e] = e$. We execute $\texttt{find}(1)$ to return the first interval $[1 : i_e]$ stored in our interval union-find data structure; $A[i_e]$ is the satellite data associated to this interval (by convention, $A[i_e] - A[1-1] = A[i_e] - A[0] = A[i_e]$). The fact that the invariant property holds shows that i_e is correctly computed.

The second step of o_i is to decrement all elements $A[j_i], A[j_i+1], \ldots, A[m]$ by 1, for some $j_i \in [1 : m]$. We will make no actual change to the elements of the list A, as this would be too inefficient, but we might have to change the state of the union-find data structure, as well as the satellite data associated to some intervals of this structure.

So, let $[x : y]$ be the interval containing j_i, returned by $\texttt{find}(j_i)$, and also assume first that $x \neq 1$.

According to the invariant, $A[j_i] \geq A[y]$ and $A[y] > A[x-1]$. After decrementing the elements $A[j_i], A[j_i+1], \ldots, A[m]$ by 1, the difference $A[t] - A[t']$ is exactly the same as before, for all $t, t' \in [j_i : m]$. In consequence, the relative order between the elements of the suffix $A[j_i : m]$ of the list A is preserved. Also, for all $t \in [x : j_i - 1]$, we have now $A[t] > A[y]$ (before decrementing $A[y]$ we had only $A[t] \geq A[y]$). However, the difference $A[y] - A[x-1]$ is now decreased by 1. If it stays strictly positive, we just update the satellite data of the respective interval (by decrementing it accordingly by 1). If $A[y] - A[x-1] = 0$, then we make the $\texttt{union}$ of the interval $[z : x-1]$ (returned by $\texttt{find}(x-1)$) and $[x : y]$ to obtain the new interval $[z : y]$. Its satellite data is $A[y] - A[z-1] = A[x-1] - A[z-1]$, so the same as the satellite data that was before associated to $[z : x-1]$. The invariant is clearly preserved, as, even after decrementing it, $A[y]$ (which is now equal to $A[x-1]$) is strictly greater than $A[z-1]$, strictly smaller than $A[t]$, for $t \in [y+1 : m]$, and smaller than or equal to $A[t]$, for $t \in [z : y]$.

If the interval containing j_i is $[1 : y]$, then we just update the satellite data of the respective interval by decrementing it by 1.

The third step of o_i is to append the element x_i to A (i.e., set $A[m+1] = x_i$), for some natural number x_i, and increment m by 1.

We implement this as follows. Let $t = m$ and $q = A[m]$ (this value is stored and maintained using the variable ℓ). While $t \geq 1$ do the following. Let $[z, t]$ be the interval returned by $\texttt{find}(t)$; we have $q = A[t]$. If $q \geq x_i$, make the union of $[z : t]$ and $[t+1 : m+1]$; update $q = q-(A[t]-A[z-1]) = A[z-1]$ (using the satellite data $A[t] - A[z-1]$ associated to $[z, t]$), update $t = z-1$, and reiterate the loop. If $q < x_i$, exit the loop. After this, we set m to $m+1$ and $\ell = x_i$.

It is not hard to see that after running this third step, so before executing operation o_{i+1}, the invariant is preserved.

Performing operation o_i takes an amount of time proportional to the sum of the number of `union` and the number of `find` operations executed during its three steps. By Lemma 2.35, this means that executing all operations $o_1, \ldots, o_\sigma$ takes in total at most $O(\sigma)$ time. □

6.2.2 Examples

These examples are based on (and supposed to be a companion in understanding) the algorithms and proofs.

Most of the algorithms which we exemplify in this section use a temporary array C to keep track of the letters occurring in w, but in slightly different ways. We will explain in each case what is the semantic of the elements in the array $C[\cdot]$.

For convenience, we assume $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{n}\}$ for the examples instead of $\Sigma = \{1, 2, 3\}$. Let $w =$ `bananaban` and thus $n = 9$ and $\sigma = 3$.

In Lemma 6.1 we want to compute $\Delta(1,1), \ldots, \Delta(1,9)$. Therefore, we traverse the word from left to right, i.e. from $\ell = 1$ to $\ell = 9$ and we maintain an array C of length 3 as well as a counter f. In this lemma, when reaching position i of the word, $C[a] = 1$ if and only if $|w[1:i]|_a \neq 0$, for $a \in \{\mathtt{a}, \mathtt{b}, \mathtt{n}\}$. This results in the following computation:

	b	a	n	a	n	a	b	a	n
ℓ	1	2	3	4	5	6	7	8	9
$C[\mathtt{a}]$	0	1	1	1	1	1	1	1	1
$C[\mathtt{b}]$	1	1	1	1	1	1	1	1	1
$C[\mathtt{n}]$	0	0	1	1	1	1	1	1	1
f	1	2	3	3	3	3	3	3	3
$\Delta(1,\ell)$	1	2	3	3	3	3	3	3	3

.

Notice that we have $\sigma = 3$. Hence for Lemma 6.2, we only consider $i \in [3:9]$ and we want to compute $\Delta(1,3)$, $\Delta(2,4)$, $\Delta(3,5)$, $\Delta(4,6)$, $\Delta(5,7)$, $\Delta(6,8)$, and $\Delta(7,9)$. In this case, when processing position i, $C[a] = |w[i-\sigma+1, i]|_a$, for $a \in \{\mathtt{a}, \mathtt{b}, \mathtt{n}\}$.

	b	a	n	a	n	a	b	a	n
i	1	2	3	4	5	6	7	8	9
$C[\mathtt{a}]$	0	1	1	2	1	2	1	2	1
$C[\mathtt{b}]$	1	1	1	0	0	0	1	1	1
$C[\mathtt{n}]$	0	0	1	1	2	1	1	0	1
f	1	2	3	2	2	2	3	2	3
$\Delta(1,i)$	1	2	3	3	3	3	3	3	3
$\Delta(i-\sigma+1,i)$	-	-	3	2	2	2	3	2	3

In Lemma 6.3 we determine for all $j \leq \frac{n-1}{\sigma} = \frac{8}{3}$ the values $\text{last}_{j\sigma+1}[a]$ and $d_{j\sigma+1}[a]$ for all $a \in \Sigma$. The way the array $C[\cdot]$ is used in this case is a bit different: when processing position i, $C[a]$ is the position of the last occurrence of a in $w[1:i]$, for $a \in \{\mathtt{a}, \mathtt{b}, \mathtt{n}\}$.

	b	a	n	a	n	a	b
i	1	2	3	4	5	6	7
$C[\mathtt{a}]$	∞	2	2	4	4	6	6
$C[\mathtt{b}]$	1	1	1	1	1	1	7
$C[\mathtt{n}]$	∞	∞	3	3	5	5	5
f	1	2	3	3	3	3	3
R	(b)	(b, a)	(b, a, n)	(b, n, a)	(b, a, n)	(b, n, a)	(n, a, b)
$P[\cdot]$	$(\infty, 1, \infty)$	$(2, 1, \infty)$	$(2, 1, 3)$	$(3, 1, 2)$	$(2, 1, 3)$	$(3, 1, 2)$	$(2, 3, 1)$
$\text{last}_i[\cdot]$				$[4, 1, 3]$			$[6, 7, 5]$
$d_i[\cdot]$				$[1, 3, 2]$			$[2, 1, 3]$

In the table above, $P[a]$ is a pointer to the position of the list R where $a \in \{\mathtt{a}, \mathtt{b}, \mathtt{n}\}$ is stored.

Finally we have a look at the algorithms for Lemma 6.4. For the first algorithm (Algorithm 4) we get the following table. We show the state of the arrays C after each iteration of the while-loop from `step one`. In this case, after each iteration of the while-loop from `step one`, $C[x]$ stores the number of occurrences of x in $w[b:a]$, for $x \in \{\mathtt{a}, \mathtt{b}, \mathtt{n}\}$. With k we simply count how many times the while-loop from `step one` was executed:

k	1	2	3	4	5	6	7	8	9	10
a	9	9	9	8	8	7	6	6	6	6
b	9	8	7	6	5	5	4	3	2	1
$C[\mathtt{a}]$	0	1	1	2	2	1	2	2	3	3
$C[\mathtt{b}]$	0	0	1	1	1	1	0	0	0	1
$C[\mathtt{n}]$	1	1	1	0	1	1	1	2	2	2
f	1	2	3	2	3	3	2	2	2	3
$V_{\mathtt{bananaban}}$			$\{7\}$	$\{7\}$	$\{7, 5\}$	$\{7, 5\}$	$\{7, 5\}$	$\{7, 5\}$	$\{7, 5\}$	$\{7, 5, 1\}$.
univ[9]			7	7	7	7	7	7	7	7
univ[8]					5	5	5	5	5	5
univ[7]						5	5	5	5	5
univ[6]										1
L_7			$\{9\}$	$\{9\}$	$\{9\}$	$\{9\}$	$\{9\}$	$\{9\}$	$\{9\}$	$\{9\}$
L_5					$\{8\}$	$\{8, 7\}$	$\{8, 7\}$	$\{8, 7\}$	$\{8, 7\}$	$\{8, 7\}$
L_1										$\{6\}$

The while-loop in `step three` is now used and we will obtain $\text{univ}[5] = \text{univ}[4] = \text{univ}[3] = 1$, and $5, 4, 3$ are all added in L_1. So $L_1 = \{3, 4, 5, 6\}$. The rest of the values in the univ array are left as initialized, namely 0, and $L_0 = \{1, 2\}$.

For the second algorithm (Algorithm 5), when reaching position i of the word, $C[a] = |w[1:i]|_a$, for $a \in \{\mathtt{a}, \mathtt{b}, \mathtt{n}\}$. Thus, we compute:

	b	a	n	a	n	a	b	a	n
i	1	2	3	4	5	6	7	8	9
$C[\mathtt{a}]$	0	1	1	2	2	3	3	4	4
$C[\mathtt{b}]$	1	1	1	1	1	1	2	2	2
$C[\mathtt{n}]$	0	0	1	1	2	2	2	2	3
freq[·]	1	1	1	2	2	3	2	4	3

.

For the computation of T set $x = \mathtt{b}$ and $m = 2$ (as determined by the fact that $C[\mathtt{b}]$ is the minimum in C). This implies $T[0] = 2$. Thus, we get for the computation of T:

i	1	2	3	4	5	6	7	8
$C[\mathtt{a}]$	4	3	3	2	2	1	1	0
$C[\mathtt{b}]$	1	1	1	1	1	1	0	0
$C[\mathtt{n}]$	3	3	2	2	1	1	1	1
m	1	1	1	1	1	1	0	0
x	b	b	b	b	b	b	b	b
T	1	1	1	1	1	1	0	0

.

Note that $T[9]$ is also set to 0. Now, in each step, when considering the letter a, we decrement $C[a]$ by 1, and then compute again the minimum of C.

For the last algorithm (Algorithm 6) we get, with $V_{\mathtt{bananaban}} = \{7, 5, 1\}$, $L_1 = \{3, 4, 5, 6\}$ and $w[3] = \mathtt{n}$, $w[4] = \mathtt{a}$, $w[5] = \mathtt{n}$, $w[6] = \mathtt{a}$, $L_5 = \{7, 8\}$ and $w[7] = \mathtt{b}$ and $w[8] = \mathtt{a}$, $L_7 = \{9\}$ and $w[9] = \mathtt{n}$:

i		0	1	2	3	4	5	6	7	8
last$_i$[·]		[10,10,10]	[10,-,-]	[-,-,10]	[2,-,-]	[4,1,3]	[4,-,-]	[-,1,5]	[6,-,-]	[-,-,5]
$L[\mathtt{a}]$	10	10	2	2	4	4	6	6	8	8
$L[\mathtt{b}]$	10	1	1	1	1	1	1	7	7	7
$L[\mathtt{n}]$	10	10	10	3	3	5	5	5	5	9

In the table above, $\text{last}_i[\cdot] = (\text{last}_i[\mathtt{a}], \text{last}_i[\mathtt{b}], \text{last}_i[\mathtt{n}])$. This representation is chosen for the ease of understanding. However, note that each $\text{last}_i[\cdot]$ is implemented as a list with exactly one element if $i + 1 \notin V_w$ (i.e., $\text{last}_i[w[i + 1]]$) and exactly $\min\{|\Sigma|, |L_{i+1}| + 1\}$ elements (i.e., $\text{last}_i[w[i + 1]]$ and $\text{last}_i[w[j]]$ for $j \in L_{i+1}$) if $i + 1 \in V_w$.

6.3 Edit Distance to k-universality

We are interested in computing the minimal number of edits we need to apply to a word w, with $|w| = n$, $\mathrm{alph}(w) = \Sigma$, with universality index $\iota(w)$, so that it is transformed into a word with universality index k, w.r.t. the same alphabet Σ. The edits considered are insertion, deletion, substitution, and the number we want to compute can be seen as the *edit distance* between w and the set of k-universal words over Σ.

However, if we want to obtain a k-universal word with $k > \iota(w)$, then it is enough to consider only insertions. Indeed, deleting a letter of a word can only restrict the set of subsequences of the respective word, while in this case we are interested in enriching it. Substituting a letter might make sense, but it can be simulated by an insertion: assume one wants to substitute the letter $\mathtt{a}$ on position i of a word w by a $\mathtt{b}$. It is enough to insert a $\mathtt{b}$ next to position i, and the set of subsequences of w is enriched with all the words that could have appeared as subsequences of the word where $\mathtt{a}$ was actually replaced by $\mathtt{b}$. We might have some extra words in the set of subsequences, which would have been eliminated through the substitution, but it does not affect our goal of reaching k-universality.

If we want to obtain a word with universality index k, for $k < \iota(w)$, then it is enough to consider only deletions. Assume that we have a sequence of edits that transforms the word w into a word w' with universality index k. Now, remove all the insertions of letters from that sequence. The word w'' we obtain by executing this new sequence of operations clearly fulfils $\iota(w'') \leq \iota(w')$. Further, in the new sequence, replace all substitutions with deletions. We obtain a word w''' with a set of subsequences strictly included in the one of w'', so with $\iota(w''') \leq \iota(w'')$. As each deletion changes the universality index by at most 1, it is clear that (a prefix of) this new sequence of deletions witnesses a shorter sequence of edits which transforms w into a word of universality index k.

So, to increase the universality index of a word it is enough to use insertions and to decrease the universality index of a word it is enough to use deletions. Nevertheless, one might be interested in what happens if we only use substitutions. In this way, we can both decrease and increase the universality index of a word. Moreover, one can see the minimal number of substitutions needed to transform w into a k-universal word as the Hamming distance between w and the set of k-universal words. We will discuss each of these cases separately.

6.3.1 Changing the k-universality with Insertions

Theorem 6.6. *Let w be a word, with $|w| = n$, $alph(w) = \Sigma$, and $\Sigma = \{1, 2, \ldots, \sigma\}$. Let $k \geq \iota(w)$ be an integer. We can compute the minimal number of insertions needed to apply to w in order to obtain a k-universal word (w.r.t. Σ) in $O(nk)$ time if $k \leq n$ and $O(T(n, \sigma, k))$ time otherwise, where $T(n, \sigma, k)$ is the time needed to compute $k\sigma - n$.*

Proof. **Case 1.** Let us assume first that $k \leq n$. We structured our proof in such a way that the idea of the solution, as well as the actual computation steps, and the arguments supporting their correctness are clearly marked.

General approach. We want to transform the word w into a k-universal word with a minimal number of insertions. Assume that the word we obtain this way is w', and $|w'| = m$. Thus, w' has a prefix $w'[1 : m']$ which is k-universal, but $w'[1 : m' - 1]$ is not k-universal. Moreover, $w'[1 : m']$ is obtained from a prefix $w[1 : \ell]$ of w, and $w'[m' + 1 : m] = w[\ell + 1 : n]$. Indeed, any insertion done to obtain $w'[m' + 1 : m]$ can be simply omitted and still obtain a k-universal word from w, with a lower number of insertions.

Consequently, it is natural to compute the minimal number of insertions needed to transform $w[1 : \ell]$ into a t-universal word, for all $\ell \leq n$ and $t \leq k$. Let $M[\ell][t]$ denote this number. By the same reasoning as above, transforming (with insertions) $w[1 : \ell]$ into a t-universal word means that there exists a prefix $w[1 : \ell']$ of $w[1 : \ell]$ which is transformed into a $(t - 1)$-universal word and $w[\ell' + 1 : \ell]$ is transformed into a 1-universal word. Clearly, the number of insertions needed to transform $w[\ell' + 1 : \ell]$ into a 1-universal word is $\sigma - \Delta(\ell' + 1, \ell)$, i.e., the number of distinct letters not occurring in $w[\ell' + 1 : \ell]$. As we are interested in the minimal number of insertions needed to transform $w[1 : \ell]$ into a t-universal word, we need to find a position ℓ' such that the total number of insertions needed to transform $w[1 : \ell']$ into a $(t - 1)$-universal word and $w[\ell' + 1 : \ell]$ into a 1-universal word is minimal.

Algorithm - initial idea. So, for $\ell \in [1 : n]$ and $t \in [1 : k]$, $M[\ell][t]$ is the minimal number of insertions needed to make $w[1 : \ell]$ t-universal. By the explanations above, we get the following recurrence $M[\ell][t] = \min\{M[\ell'][t - 1] + (\sigma - \Delta(\ell' + 1, \ell)) \mid \ell' \leq \ell\}$. Clearly, $M[\ell][1] = \sigma - \Delta(1, \ell)$. Also, it is immediate to note that $M[\ell][t] \geq M[\ell''][t]$ for all $\ell \leq \ell''$. Indeed, transforming a word into a t-universal word can always be done with at most as many insertions as those used in transforming any of its prefixes into a t-universal word.

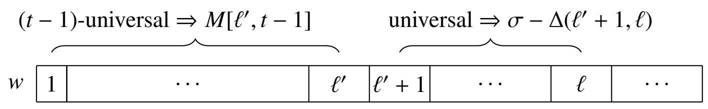

$(t-1)$-universal $\Rightarrow M[\ell', t-1]$
universal $\Rightarrow \sigma - \Delta(\ell'+1, \ell)$
w
1
$\cdots$
ℓ'
$\ell'+1$
$\cdots$
ℓ
$\cdots$

Figure 6.2: Illustration of the formula developed for the computation of $M[\ell][t]$.

We now want to compute the elements of matrix M. Before this, we produce the data structures of Lemma 6.3 (and we use the notations from its framework). That is, we compute in $O(n)$ time $\text{last}_{j\sigma+1}[a]$ and $d_{j\sigma+1}[a] = \Delta(\text{last}_{j\sigma+1}[a], j\sigma + 1)$, for all $a \in \Sigma$ and all $j \leq \frac{(n-1)}{\sigma}$.

By Lemma 6.1, we can compute the values $M[\ell][1]$, for all $\ell \in [1 : n]$ in $O(n)$ time. However, a direct computation of the values $M[\ell][t]$, for $t > 1$, according to the recurrence above is not efficient. Implemented directly, it requires $O(n^2k)$ time; using an efficient structure (e.g., interval trees) for computing the various minima leads to an $O(nk \log n)$-time solution; exploiting the algebraic properties of M (related to the Monge property [4]) leads to an $O(nk \log \sigma / \log\log \sigma)$-time solution. We will describe a more efficient solution.

A useful observation. Assume that to transform $w[1:\ell]$ into a t-universal word we transform $w[1:\ell']$ into a $(t-1)$-universal word and $w[\ell'+1:\ell]$ into a 1-universal word. The number of insertions needed to do this is $M[\ell'][t-1] + (\sigma - \Delta(\ell'+1, \ell))$. If $w[\ell'+1]$ occurs twice in $w[\ell'+1:\ell]$, then $M[\ell'][t-1] + (\sigma - \Delta(\ell'+1, \ell)) \geq M[\ell'+1][t-1] + (\sigma - \Delta(\ell'+2, \ell))$. Thus, we can rewrite our recurrence in the following way, using the framework of Lemma 6.3: $M[\ell][t] = \min\{M[\ell'][t-1] + (\sigma - \Delta(\ell'+1, \ell)) \mid \ell'+1 \in S_\ell \cup \{\ell+1\}\}$ (recall the definition of $S_\ell = \{\text{last}_\ell[a] \mid a \in \text{alph}(w[1:\ell])\}$ from Section 6.2).

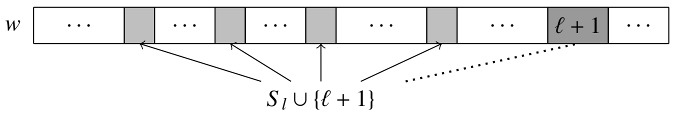


Figure 6.3: Only the positions $\ell'+1 \in \{\text{last}_\ell[a] \mid a \in \text{alph}(w[1:\ell])\} \cup \{\ell+1\} = S_\ell \cup \{\ell+1\}$ are needed to compute $M[\ell][t]$ by dynamic programming. These positions are depicted here in grey.

Once more, a brief analysis can be done. Using directly this observation leads to an $O(nk\sigma)$-time algorithm for our problem; an implementation based on, e.g., interval trees runs in $O(nk \log \sigma)$-time. In the following we see that a faster solution exists.

In fact, in the efficient version of our algorithm we will use a slightly weaker formula, where the minimum is computed for all elements $\ell'+1$ from a set $S'_\ell \cup \{\ell+1\}$, instead of the set $S_\ell \cup \{\ell+1\}$, where S'_ℓ is a superset of size at most 2σ of S_ℓ defined as follows. If $\ell = j\sigma + i$, for some $j \leq (n-1)/\sigma$ and $i \in [1:\sigma]$, then $S'_\ell = \begin{cases} S_\ell & \text{if } i = 1, \\ S_{j\sigma+1} \cup \{j\sigma+2, \ldots, j\sigma+i\} & \text{if } i \in [2:\sigma]. \end{cases}$

Algorithm - the efficient variant. Using the observation above, together with Lemma 6.5, we can compute the elements of the matrix M efficiently using dynamic programming.

So, let us consider a value $t \geq 2$. Assume that we have computed the values $M[\ell][t-1]$, for all $\ell \in [1:n]$. We now want to compute the values $M[\ell][t]$, for all $\ell \in [1:n]$. The main idea in doing this efficiently is to split the computation of the elements on column $M[\cdot][t]$ of the matrix M in phases. In phase j we compute the values $M[j\sigma+1][t], M[j\sigma+2][t], \ldots, M[(j+1)\sigma][t]$, for $j \leq (n-1)/\sigma$.

We now consider some j, with $0 \leq j \leq (n-1)/\sigma$. We want to apply Lemma 6.5, so we need to define the list A of size σ. This is done as follows.

We will maintain an auxiliary array $\mathrm{pos}[\cdot]$ with σ elements. Moreover, the element $A[i]$, for each i, is accompanied by two satellite information: a position of w and the letter found on that position. For a from 1 to σ, if $d_{j\sigma+1}[a] = \sigma - i$ for some $i < \sigma$ then we set $A[i+1] = M[\mathrm{last}_{j\sigma+1}[a]-1][t-1]+i$ and $\mathrm{pos}[a] = i+1$; the satellite data of $A[i+1]$ is the pair $(\mathrm{last}_{j\sigma+1}[a], a)$. If, for some letter a, $\mathrm{last}_{j\sigma+1}[a] = n+1$ and $d_{j\sigma+1}[a] = 0$ (i.e., a does not occur in $w[1 : j\sigma+1]$) we set $\mathrm{pos}[a] = 0$.

Intuitively, one can see the elements of A as triples: $(A[e], \mathrm{last}_{j\sigma+1}[a], a)$ where $A[e] = M[\mathrm{last}_{j\sigma+1}[a]-1][t-1]+e-1$, with $e \in [1 : \sigma]$ and $a \in \Sigma$. More precisely, let $a_d, a_{d-1}, \dots, a_1$ be the letters of Σ that occur in $w[1 : j\sigma+1]$, ordered such that $\mathrm{last}_{j\sigma+1}[a_e] < \mathrm{last}_{j\sigma+1}[a_f]$ if and only if $e > f$. At this point, we have defined only the last d elements of A and, for $i \in [1 : d]$, the element on position $\sigma - i + 1$ is $A[\sigma-i+1] = M[\mathrm{last}_{j\sigma+1}[a_i]-1][t-1] + (\sigma - i)$ and has the satellite data $(\mathrm{last}_{j\sigma+1}[a_i], a_i)$. Also, $\mathrm{pos}[a_i] = \sigma - i + 1$. The first $\sigma - d$ elements of A are set to ∞; as convention, applying arithmetic operations to ∞ leaves it unchanged.

We set m to σ and define (and apply) a sequence of operations $o_1, \dots, o_\sigma$ as in Lemma 6.5.

An invariant: We want to ensure that the list A fulfils the following invariant properties before the execution of each operation o_i.

- For $e \in [1 : d]$, the triple on position $\sigma - e + 1$ of A is: $(M[\mathrm{last}_{j\sigma+1}[a_e]-1][t-1]+(\sigma-\Delta(\mathrm{last}_{j\sigma+1}[a_e], j\sigma+i)), \mathrm{last}_{j\sigma+1}[a_e], a_e)$. That is, $A[\sigma-e+1] = M[\mathrm{last}_{j\sigma+1}[a_e]-1][t-1] + (\sigma - \Delta(\mathrm{last}_{j\sigma+1}[a_e], j\sigma+i))$.
- For $g \in [1 : i-1]$, the triple on position $\sigma + g$ of A is: $(M[j\sigma+g][t-1] + (\sigma - \Delta(j\sigma+g+1, j\sigma+i)), j\sigma+g+1, w[j\sigma+g+1])$. That is $A[\sigma+g] = M[j\sigma+g][t-1] + (\sigma - \Delta(j\sigma+g+1, j\sigma+i))$.
- $\mathrm{pos}[a]$ is the position of the rightmost position i storing a triple $(A[i], \ell, a)$.

That is, the list A contains all the values $M[\ell][t-1] + (\sigma - \Delta(\ell+1, j\sigma+i))$, for $\ell + 1 \in S_{j\sigma+1} \cup \{j\sigma+2, \dots, j\sigma+i\}$, and $\mathrm{pos}[a]$ indicates the rightmost position of the list A where we store a value $M[\ell][t-1] + (\sigma - \Delta(\ell+1, j\sigma+i))$ with $w[\ell+1] = a$. A consequence of this is that $A[\mathrm{pos}[a]] = M[\mathrm{last}_{j\sigma+i}[a]-1][t-1] + (\sigma - \Delta(\mathrm{last}_{j\sigma+i}[a], j\sigma+i))$.

The invariant clearly holds for $i = 1$.

Algorithm - application of Lemma 6.5. In o_i, we extract the minimum q of A. Then set $M[j\sigma+i][t] = \min\{q, M[j\sigma+i][t-1]+\sigma\}$. We decrement by 1 all elements of A on the positions $\mathrm{pos}[a]+1, \mathrm{pos}[a]+2, \dots, m$, where $a = w[j\sigma+i+1]$. Then, we append to A the element $M[j\sigma+i][t-1] + (\sigma-1)$, with the satellite data $(j\sigma+i+1, a)$, which implicitly increments m by 1, and set $\mathrm{pos}[a] = m$.

Claim 1. The invariant holds after operation o_i.

Proof of Claim 1. We now need to show that the invariant is preserved after this step. If $a = w[j\sigma + i + 1]$ then the number of distinct letters occurring after each position $g > \text{last}_{j\sigma+i}[a]$ in $w[1 : j\sigma + i]$ is exactly one smaller than the number of distinct letters occurring after g in $w[1 : j\sigma + i + 1]$. This means that $M[g - 1][t - 1] + (\sigma - \Delta(g, j\sigma + i + 1))$ is one smaller than $M[g - 1][t - 1] + (\sigma - \Delta(g, j\sigma + i))$. Consequently, all values occurring on positions greater than $\text{pos}[a]$ in the list A, which stored some values $M[g - 1][t - 1] + (\sigma - \Delta(g, j\sigma + i + 1))$ with $g > \text{last}_{j\sigma+i}[a]$, should be decremented by 1. Also, the number of distinct letters occurring after each position $g \leq \text{last}_{j\sigma+i}[a]$ in $w[1 : j\sigma + i]$ is exactly the same as number of distinct letters occurring after g in $w[1 : j\sigma + i + 1]$. Thus, all values occurring on positions smaller or equal to $\text{pos}[a]$ in the list A, which stored some values $M[g - 1][t - 1] + (\sigma - \Delta(g, j\sigma + i + 1))$ with $g \leq \text{last}_{j\sigma+i}[a]$, should stay the same. So, the invariant holds for the first $\sigma + i - 1$ positions of A. After appending $M[j\sigma + i][t - 1] + (\sigma - 1)$ to A and incrementing m, then the invariant holds for the position $\sigma + i$ (which is also the last position) of A too, so the invariant still holds for all positions of A.

Furthermore, the only position of the pos array that needs to be updated after operation o_i is $\text{pos}[a]$, and it needs to be set to the new value of m. This is exactly what we do. □

Claim 2. $M[j\sigma + i][t]$ is correctly computed, for all $i \in [1 : \sigma]$.

Proof of Claim 2. According to the invariant, before executing operation o_i, A contains the values $M[\ell][t-1]+(\sigma-\Delta(\ell+1, j\sigma+i))$, for $\ell+1 \in S_{j\sigma+1}$, and $M[j\sigma+g][t-1]+(\sigma-\Delta(j\sigma+g+1, j\sigma+i))$, for $g \in [1 : i - 1]$. As $S'_{j\sigma+i} = S_{j\sigma+1} \cup \{j\sigma + g + 1 \mid g \in [1 : i - 1]\}$ is a superset of size at most 2σ of $S_{j\sigma+i}$, we obtain that $M[j\sigma + i][t]$ is correctly computed as the minimum between the smallest value in A and $M[j\sigma + i][t - 1] + \sigma$. □

Algorithm - the result of applying Lemma 6.5. After executing the σ operations $o_1, \ldots, o_\sigma$, we have computed the values $M[j\sigma + 1][t], M[j\sigma + 2][t], \ldots, M[(j + 1)\sigma][t]$ correctly. We can move on to phase $j + 1$ and repeat this process.

The result and complexity. The minimal number of insertions needed to make w k-universal is, according to the observations we made, correctly computed as $M[n][k]$.

By Lemma 6.5, computing $M[j\sigma + 1][t], M[j\sigma + 2][t], \ldots, M[(j + 1)\sigma][t]$ takes $O(\sigma)$ for each j. Overall, computing the entire column $M[\cdot][t]$ takes $O(n)$ time. We do this for all $t \leq k$, so we use $O(nk)$ time in total to compute all elements of M. This concludes Case 1.

Case 2. If $k > n$, we return $k\sigma - n$. We need, in all cases, $k\sigma - n$ insertions to obtain a word of length $k\sigma$ from w. This is also sufficient: we first use $n(\sigma - 1)$ insertions to transform w into $(1 \cdots \sigma)^n$; then, by $(k - n)\sigma$ insertions, we further transform it into $(1 \cdots \sigma)^k$. So the time needed to solve our problem, in this case, is the time needed to compute $k\sigma - n$. □

Note that, if k is in $O(c^n)$ for constant c, then $T(n, \sigma, k) \in O(n \log \sigma)$. Hence, in that case, our algorithm runs in $O(n \log \sigma)$ time. If $k \in O(1)$ our algorithm runs in optimal $O(n)$ time.

Algorithm 7: The efficient algorithm from Case 1 of Theorem 6.6 (on insertions).

```
Input  : word $w$, alphabet $\Sigma$, int $k$
Output : minimal number of insertions
// initialization
int $n \leftarrow |w|$; int $\sigma \leftarrow |\Sigma|$; int $d \leftarrow 0$;
int $M[n][k]$;
// initialise first column of $M$
for $l = 1$ to $n$ do
    $M[l][1] \leftarrow \sigma - \Delta(1, l)$;
end
// efficient variant
for $t = 2$ to $k$ do
    // $\leq (n-1)/\sigma$ phases
    for $j = 0$ to $\lceil (n-1)/\sigma \rceil$ do
        int $A[\sigma][3]$ (list of triples including satellite data); int pos$[\sigma]$;
        for $a = 1$ to $\sigma$ do
            if $d_{j\sigma+1}[a] > 0$ then
                int $i \leftarrow \sigma - d_{j\sigma+1}[a]$; $d \leftarrow d + 1$;
                $A[i+1][1] \leftarrow M[\mathrm{last}_{j\sigma+1}[a] - 1][t-1] + i$;
                pos$[a] \leftarrow i + 1$;
                // satellite data for $A[i+1]$
                $A[i+1][2] \leftarrow \mathrm{last}_{j\sigma+1}[a]$;
                $A[i+1][3] \leftarrow a$;
            end
            if $d_{j\sigma+1}[a] = 0$ and $\mathrm{last}_{j\sigma+1}[a] = n + 1$ then
                pos$[a] = 0$;
            end
        end
        // exactly $d$ letters of $\Sigma$, occur in $w[1 : j\sigma + 1]$
        // (ordered increasingly by their last occurrence):
        // $A[\sigma - d + 1][3], A[\sigma - d + 2][3], \ldots, A[\sigma - 1][3], A[\sigma-][3]$.
        set all elements in $A[1 : \sigma - d]$ to $\infty$ (they cannot be changed);
        $m = \sigma$;
        // apply sequence of operations as in Theorem 6.5
        for $i = 1$ to $\sigma$ do
            $q \leftarrow$ minimum of $A$;
            $M[j\sigma + i][t] \leftarrow \min\{q, M[j\sigma + i][t-1] + \sigma\}$;
            $a = w[j\sigma + i + 1]$;
            decrement positions pos$[a] + 1$, pos$[a] + 2, \ldots, m$ by 1;
            append $M[j\sigma + i][t-1] + (\sigma - 1)$ to $A$ (i.e., set $A[m+1][1]$ to this value);
            // and add satellite data
            $A[m+1][2] \leftarrow j\sigma + i + 1$;
            $A[m+1][3] \leftarrow a$;
            $m \leftarrow m + 1$;
            pos$[a] \leftarrow m$;
        end
    end
end
return $M[n][k]$;
```

6.3.2 Changing the k-universality with Deletions

Theorem 6.7. *Let w be a word, with $|w| = n$, $alph(w) = \Sigma$, and $\Sigma = \{1, 2, \ldots, \sigma\}$. Let k be an integer with $k \leq \iota(w) \leq n/\sigma$. We can compute in $O(nk)$ time the minimal number of deletions needed to obtain a word of universality index k (w.r.t. Σ) from w.*

Proof. General approach. The case $k = 0$ is trivial. We just need to count how many times each letter occurs and then remove the letter that occurs the least number of times. This takes $O(n)$ time. So let us assume that $k > 0$. To simplify the presentation, we will call a word u a weak-p-universal word if u is p-universal and by deleting the last letter of u we obtain a word of universality index $p - 1$.

Assume that w' is a k-universal word that can be obtained by applying the sequence of deletions of minimal length to w. Clearly, w' is a subsequence of w, and, by the decomposition defined in the context of Theorem 2.27, there exist the arches $w'_1, \ldots, w'_k$, all of universality index exactly 1, and w'_{k+1}, of universality index 0, such that $w' = w'_1 \cdots w'_k w'_{k+1}$. Moreover, each arch w'_i, with $i \in [1 : k]$, is a weak-1-universal word.

It follows that actually each of the words w'_i is a subsequence of w too. So we will try to identify the factors $w[i_{j-1} + 1 : i_j]$, with $j \in [1 : k + 1]$ and $i_0 = 0$, from which w'_j is obtained by deletions, with the important condition that the last letter of $w[i_{j-1} + 1 : i_j]$ is not deleted (so the last letters of w'_j and $w[i_{j-1} + 1 : i_j]$ coincide).

Note that, to obtain the word $w'_1 \cdots w'_k w'_{k+1}$ from w with a minimal number of deletions, we need to find a position i_k of w such that $w'_1 \cdots w'_k$ is obtained from $w[1 : i_k]$ and w'_{k+1} from $w[i_k + 1 : n]$, with the restrictions that $w[i_k]$ is not deleted and the overall number of deletions made in this process is the smallest (over all possible choices of the position i_k). If we now have i_k, we can then search for i_{k-1}: we need to find a position of w such that $w'_1 \cdots w'_{k-1}$ is obtained from $w[1 : i_{k-1}]$ and w'_k from $w[i_{k-1} + 1 : i_k]$, with the similar restrictions that $w[i_{k-1}]$ is not deleted and the overall number of deletions made in this process is the smallest (over all possible choices of the position i_k). And then we continue with i_{k-2}, i_{k-3}, and so on.

Thus, it seems natural to consider and solve the following type of subproblems: what is the minimal number of deletions we need to apply to transform a prefix $w[1 : i]$ into a weak-p-universal word v, without deleting $w[i]$.

Algorithm - initial idea. We first compute all the data structures defined in Lemma 6.4.

We define the $n \times k$ matrix N, where $N[i][p]$ is the minimal number of deletions we need to apply to $w[1 : i]$, without deleting $w[i]$, to obtain a weak-p-universal word v from it (for $1 \leq i \leq n$ and $1 \leq p \leq k$). If $w[1 : i]$ is not p-universal, $N[i][p]$ will be set to ∞.

To compute this matrix, we note that if $N[i][1] \neq \infty$, then $N[i][1] = \text{freq}[i] - 1$. Indeed, in order to produce from $w[1 : i]$ a weak-1-universal word, which fulfils the conditions stated above, we have to delete all occurrences of the letter $w[i]$ except for the one on position i.

In general, the minimal number of deletions needed to transform a factor $w[i : j]$ of universality index at least 1 (so with $i \leq \text{univ}[j]$) into a weak-1-universal word v, such that $w[j]$ is not deleted, is $|w[i : j]|_{w[j]} - 1$.

Accordingly, we can define the elements of the matrix N as $N[i][p] = \min\{N[i'][p-1] + |w[i'+1 : i]|_{w_i} \mid i' \in [1 : \text{univ}[i] - 1]\}$, for $i \in [1 : n], p \in [2 : k]$.

A straightforward implementation of the above formula is not efficient, so we will explore alternative and more efficient ways to compute $N[i][p]$.

Algorithm - an efficient implementation. We will show how the elements of the column $N[\cdot][p]$ can be computed efficiently for a given $p \geq 2$, assuming that we have computed them for $N[\cdot][p-1]$.

Firstly, we define the data structures of Lemma 2.37 (section 2.7) allowing us to answer range minimum queries for the column $N[\cdot][p-1]$ of N, denoted by RMQ_{p-1} in the following. Note that the query $\text{RMQ}_{p-1}(j, i)$ returns, for some $j < i$, the position k with $k \in [j : i]$, such that $w[1 : k]$ is the prefix of w, ending between j and i, which can be transformed into a weak-$(p-1)$-universal word with the least number of deletions (among all other prefixes ending between i and j), such that its last letter $w[k]$ is not deleted.

We define an auxiliary array $M'[\cdot]$ with n elements, where if $i \geq (p-1)\sigma$, we have $M'[i] = \min\{N[j][p-1] + |w[j+1 : i]|_{w[i]} \mid j < i\}$, and if $i < (p-1)\sigma$, we have $M'[i] = \infty$ (that is, a large enough value).

Intuitively, $M'[i]$ is the minimal number of deletions we need to make in $w[1 : i]$ in order to obtain a word $v'v''$, where v' is a weak-$(p-1)$-universal word and v'' is a word with universality index 0 which contains no occurrence of $w[i]$.

We observe that the values $M'[i]$ can be computed as follows. Firstly, we set $M'[i] = \infty$ if $i < (p-1)\sigma$. Then, for $i \geq (p-1)\sigma$, let $\ell = \text{last}_{i-1}[w[i]]$.

Claim 1. We claim that $M'[i] = 1 + \min\{M'[\ell], N[\text{RMQ}_{p-1}(\ell+1, i-1)][p-1]\}$, if $\text{last}_{i-1}[w[i]] \neq n+1$, or $M'[i] = \infty$ otherwise.

Proof of Claim 1. The case when $w[i]$ does not occur in $w[1 : i-1]$ is trivial. The first part of the claim holds because the minimal number of deletions we need to make in $w[1 : i]$ in order to obtain a word $v'v''$, where v' is a weak-$(p-1)$-universal word and v'' is a word with universality index 0 which contains no occurrence of $w[i]$ is:

- either the minimal number of deletions we need to apply to $w[1 : \ell]$ in order to obtain a word $v'_0 v''_0$, where v'_0 is a weak-$(p-1)$-universal word and v'' is a word with universality index 0 which contains no occurrence of $w[\ell] = w[i]$, and then remove the last occurrence of $w[i]$ from $w[1 : i]$,
- or the minimal number of deletions we need to apply to $w[1 : i]$ in order to obtain a word $v'v''$, where v' is a weak-$(p-1)$-universal word obtained from a prefix $w[1 : j]$, with $j > \ell$, by deleting some letters, but not $w[j]$, and v'' is a word with universality index 0 which contains no occurrence of $w[i]$, obtained from $w[j+1 : i]$ by the single deletion of $w[i]$.

This **concludes the proof of Claim 1.**

Computing M' takes linear time. Now, we can use the array M' to compute the values stored in N. Assume that we want to compute $N[i][p]$.

Let $j = \text{univ}[i]$. If $j = 0$, then set $N[i][p] = \infty$. Assume in the following that $j \neq 0$. Clearly, $i \in L_j$. Therefore, the structures we computed with Lemma 6.4 provide the value $t = \text{last}_{j-1}[w[i]]$. If $t = n + 1$ (i.e., $w[i]$ does not occur in $w[1 : j - 1]$), and because $p \geq 2$, we set $N[i][p] = \infty$ (we cannot transform the word $w[1 : i]$ into a p-universal word by deletions when it only contains one occurrence of $w[i]$). If $t < n + 1$, let $r = \text{RMQ}_{p-1}(t + 1, j - 1)$.

Claim 2. We claim that the following holds $N[i][p] = \min\{M'[t] + |w[t + 1 : i]|_{w[i]} - 1, N[r][p - 1] + |w[r + 1 : i]|_{w[i]} - 1\}$.

Proof of Claim 2. Indeed, this is true because in order to transform (with a minimal number of deletions) $w[1 : i]$ into a weak-p-universal word without deleting $w[i]$ we can

- either transform (with a minimal number of deletions) a prefix $w[1 : t']$ of $w[1 : t]$ into a weak-$(p - 1)$-universal word and remove all the occurrences of $w[i]$ from $w[t' + 1 : t]$, and, then, all occurrences of $w[i]$ from $w[t + 1 : i]$, except $w[i]$,
- or we transform (again with a minimal number of deletions) a prefix $w[1 : t']$ of $w[1 : j]$, with $t' > t$ and $j = \text{univ}(i)$, into a weak-$p - 1$-universal word, and then remove all the occurrences of $w[i]$ from $w[t' + 1 : i]$, except $w[i]$.

This **concludes the proof of Claim 2.**

Let us now note that $M'[t] + |w[t + 1 : i]|_{w[i]} - 1 = M'[t] + \text{freq}[i] - \text{freq}[t] - 1$ (because $w[i] = w[t]$). Also, $N[r][p-1] + |w[r+1 : i]|_{w[i]} - 1 = N[r][p-1] + |w[t+1 : i]|_{w[i]} - 1 = N[r][p-1] + \text{freq}[i] - \text{freq}[t] - 1$ because $w[i]$ does not occur in $w[t + 1 : r]$ (as $w[i]$ does not occur between $w[t + 1 : j - 1]$).

This gives us a way to compute each $N[i][p]$ in constant time (once $M'[\cdot]$ was computed).

Thus, computing the entire column $N[\cdot][p]$ takes overall linear time $O(n)$ (including here the computation of the array M').

Consequently, in total, we can compute the elements of the matrix N in $O(nk)$ time.

Collecting the results. We are not done yet, as it is not clear which is the minimal number of deletions we need in order to transform w into a k-universal word.

Recall that Lemma 6.4 also computes the array $T[\cdot]$ with $T[i] = \min\{|w[i + 1 : n]|_a \mid a \in \Sigma\}$, for $i \in [0 : n]$.

Now, the minimal number of deletions we need in order to transform w into a k-universal word is clearly $\min\{N[i][k] + T[i] \mid 1 \leq i \leq n\}$: we check which is the minimal number of deletions we need in order to both transform a prefix $w[1 : i]$ into a weak-p-universal word, without deleting $w[i]$, and the word $w[i + 1 : n]$ into a word with universality index 0.

Complexity. According to the above, the answer returned by our algorithm can be computed in $O(n)$ time, after the matrix N was computed. So, overall, the time complexity of the algorithm is $O(nk)$.

The correctness of this approach follows from the observations we made during the explanation of the algorithm. So, the statement follows. □

The idea of this proof is the following. Assume that w' is a word of universality index k obtained via the sequence of deletions of minimal length from w. Clearly, w' is a subsequence of w, and, by the decomposition defined in Theorem 2.27, there exist $w'_1, \dots, w'_k$, all of universality index exactly 1, and w'_{k+1}, of universality index 0, such that $w' = w'_1 \cdots w'_k w'_{k+1}$. It follows that each of the words w'_i is a subsequence of w too. So we will try to identify each subsequence $w'_1 \cdots w'_p$ for $p \leq k$ and the shortest factor $w[1 : i]$ from which it is obtained. To this end, we define the matrix N, where $N[i][p]$ is the minimal number of deletions we need to apply to $w[1 : i]$, without deleting $w[i]$, to obtain a word v from it, with $\iota(v) = p$ and $\iota(v[1 : |v| - 1]) = p - 1$ (for $i \in [1 : n]$ and $p \in [1 : k]$). If $\iota(w[1 : i]) \geq 1$, then $N[i][1] = |w[1 : i]|_{w[i]} - 1$, as we have to delete all occurrences of $w[i]$ from $w[1 : i]$, except the one on position i. Then, $N[i][p] = \min\{N[j][p - 1] + |w[j + 1 : i]|_{w[i]} - 1 \mid j < i$ such that $\iota(w[j + 1 : i]) \geq 1\}$. This gives a dynamic programming algorithm for computing N. Using additional data structures extending the standard Range Minimum Queries structures (see Lemma 2.37), we can compute the elements of N in $O(nk)$ time. To show the statement, we return $\min\{N[i, k] + T[i] \mid 1 \leq i \leq n\}$, using the array T of Lemma 6.4.

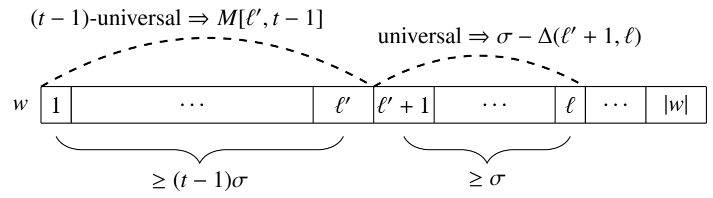
6.3.3 Changing the k-universality with Substitutions

Theorem 6.8. *Let w be a word, with $|w| = n$, alph$(w) = \Sigma$, and $\Sigma = \{1, 2, \dots, \sigma\}$. Let k be an integer $0 \leq k \leq \lfloor \frac{n}{\sigma} \rfloor$. We can compute the minimal number of substitutions needed to apply to w in order to obtain a k-universal word (w.r.t. Σ) in $O(nk)$ time.*

Proof. Recall that $\iota(w)$ is the initial universality index of w. We will distinguish between the cases $\iota(w) < k$ and $\iota(w) > k$. While the former allows for an argumentation similar to Theorem 6.6 for insertions, the latter will fall back to the Theorem 6.7 for deletions.

Case 1. Let us assume first that $\iota(w) < k$.

General approach. At a high level, the algorithm and data structures we use here are similar to those used in the case of changing the universality of a word by insertions, described in Theorem 6.6 (the finer details are, however, different). As in the respective algorithm, we will compute $M[\ell][t]$ the minimal number of substitutions one needs to apply to $w[1 : \ell]$ in order to make it t-universal, for all $\ell \in [1 : n]$ and all $t \in [1 : k]$. Clearly, to edit $w[1 : \ell]$ into a t-universal word using substitutions, we first create a $(t-1)$-universal word from a prefix $w[1 : \ell']$ of $w[1 : \ell]$, and then a 1-universal word from $w[\ell' + 1 : \ell]$. As in the case of insertions, the number of substitutions used in this process has to be minimal among all the numbers we obtain when choosing ℓ' in all possible ways. The main differences are that, in the case of substitutions, we need to have that $|w[\ell' + 1 : \ell]| \geq \sigma$, or we would not be able to obtain a 1-universal word from $w[\ell' + 1 : \ell]$, and $|w[1 : \ell']| \geq (t-1)\sigma$.



$(t-1)$-universal $\Rightarrow M[\ell', t-1]$
universal $\Rightarrow \sigma - \Delta(\ell' + 1, \ell)$
w
1
$\cdots$
ℓ'
$\ell' + 1$
$\cdots$
ℓ
$\cdots$
$|w|$
$\geq (t-1)\sigma$
$\geq \sigma$

Figure 6.4: Illustration of the formula developed for the computation of $M[\ell][t]$.

Algorithm - initial idea. As described informally above, we compute the $n \times k$ matrix $M[\cdot][\cdot]$, where $M[\ell][t]$ denotes the minimal number of substitution needed to transform $w[1 : \ell]$ into a t-universal word, for all $\ell \in [1 : n]$ and $t \in [1 : k]$. We will achieve this using dynamic programming. By the remarks we made above, it is not hard to see that:

$$M[\ell][t] = \min\{M[\ell'][t-1] + (\sigma - \Delta(\ell' + 1, \ell)) \mid \sigma(t-1) < \ell' + 1 \leq \ell - \sigma + 1\}.$$

In order to compute M efficiently, we will run the algorithms from Lemmas 6.1 and 6.3. These will provide us with the data structures $\text{last}_{j\sigma+1}[\cdot]$, and $d_{j\sigma+1}[\cdot]$, for all j, as well as all $M[\cdot][1]$ in $O(n)$ time. More precisely, $M[\ell][1] = \sigma - \Delta(1, \ell)$, if $\ell \geq \sigma$, and $M[\ell][1] = \infty$, if $\ell < \sigma$. As in the case of insertions, the direct computation of $M[\ell][t]$ would be too costly. Therefore, we will need further observations.

Observations. Assume that the letter on position $\ell' + 1 < \ell - \sigma + 1$ in w, namely $w[\ell' + 1]$, occurs at least twice in $w[\ell' + 1 : \ell]$. Then $(\sigma - \Delta(\ell' + 1, \ell)) = (\sigma - \Delta(\ell' + 2, \ell))$. Thus, it clearly follows that $M[\ell'][t-1] + (\sigma - \Delta(\ell' + 1, \ell)) \geq M[\ell' + 1][t-1] + (\sigma - \Delta(\ell' + 2, \ell))$. So, once more, we only need to consider in our recurrence that $\ell' + 1$ is the rightmost occurrence of some letter inside the factor $w[1 : \ell]$. The observation made above does not include the case when $\ell' + 1 = \ell - \sigma + 1$, so we will just add this position in the set of the relevant positions for our recurrence too.

As we did in the proof of Theorem 6.6, we can now rewrite our recurrence in a way that will enable us to apply Lemma 6.3. For $\mathfrak{S}_\ell = (S_\ell \cap [(t-1)\sigma : \ell - \sigma]) \cup \{\ell - \sigma + 1\}$:

$$M[\ell][t] = \min\{M[\ell'][t-1] + (\sigma - \Delta(\ell' + 1, \ell)) \mid \ell' + 1 \in \mathfrak{S}_\ell\}.$$

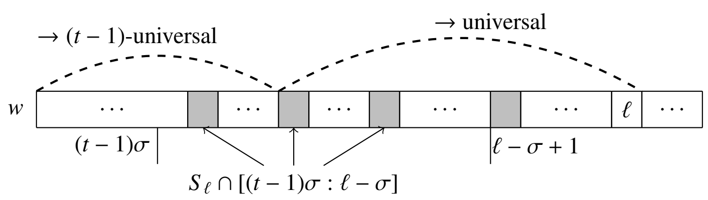


Figure 6.5: Illustration of $\mathfrak{S}_\ell$ and how it is used to compute $M[\ell][t]$. Here we only select from $S_\ell = \{\mathrm{last}_\ell[a] \mid a \in \mathrm{alph}(w[1:\ell])\}$ the elements which are in $[(t-1)\sigma : l-\sigma]$. To obtain $\mathfrak{S}_\ell$ we also have to consider the position $\ell - \sigma + 1$. The elements of $\mathfrak{S}_\ell$ are depicted with grey in the figure.

Algorithm - the efficient variant. Using these observations in combination with Lemma 6.5, we can compute the elements of the matrix M efficiently. The algorithm is similar to the one from Case 1 of Theorem 6.6. Firstly, by Lemma 6.2, we can compute the values $\Delta(i-\sigma+1, i)$, for all i, in $O(n)$ time.

So, let us consider a value $t \geq 2$. Assume that we have computed the values $M[\ell][t-1]$, for all $\ell \in [1:n]$. We now want to compute the values $M[\ell][t]$, for all $\ell \in [1:n]$. The main idea in doing this efficiently is to split the computation of the elements on column $M[\cdot][t]$ of the matrix M in phases. In phase j we compute the values $M[j\sigma+1][t], M[j\sigma+2][t], \ldots, M[(j+1)\sigma][t]$, for $j \leq (n-1)/\sigma$.

Now we consider some j, with $0 \leq j \leq (n-1)/\sigma$. We want to apply Lemma 6.5, so we need to define the list A of size σ. This is done as follows.

We will keep an auxiliary array pos[·] with σ elements. Moreover, the element on each position i of A, namely $A[i]$, will be accompanied by two satellite data: a position of w and the letter on that position. Now, for a such that $\mathrm{last}_{j\sigma+1}[a] \in \mathfrak{S}_{j\sigma+1} \setminus \{(j-1)\sigma+2\}$, we have that $d_{j\sigma+1}[a] = \sigma - i$ for some $i \in [0 : \sigma-2]$ (as $w[\mathrm{last}_{j\sigma+1}[a] : j\sigma+1]$ contains at least a and $w[(j-1)\sigma+2]$). We set $A[i+1] = M[\mathrm{last}_{j\sigma+1}[a]-1][t-1] + i$ and $\mathrm{pos}[a] = i+1$; the satellite data of A is the pair $(\mathrm{last}_{j\sigma+1}[a], a)$. We also set $A[\sigma] = M[(j-1)\sigma+1][t-1] + (\sigma - \Delta((j-1)\sigma+2, j\sigma+1))$ and $\mathrm{pos}[w[(j-1)\sigma+2]] = \sigma$; the satellite data of A is the pair $((j-1)\sigma+2, w[(j-1)\sigma+2])$. If, for some letter a, $\mathrm{last}_{j\sigma+1}[a] = n+1$ (i.e., a does not occur in $w[1 : j\sigma+1]$), we simply set $\mathrm{pos}[a] = 0$.

The elements of A which are not defined above will store ∞ (i.e., a large enough value, at least $(k+1)\sigma+1$); for simplicity, if we apply any arithmetic operation to ∞, we get ∞. We also set m to σ. Now we are in the position of defining and applying a sequence of operations $o_1, \ldots, o_\sigma$ from Lemma 6.5.

Algorithm - application of Lemma 6.5. In o_i, we extract the minimum q of A. Then set $M[j\sigma+i][t] = q$. We decrement by 1 all elements of A on the positions $\text{pos}[a]+1, \text{pos}[a]+2, \ldots, m$, where $a = w[(j-1)\sigma+i+2]$. Then, we append to A the element $M[(j-1)\sigma+i+1][t-1] + (\sigma - \Delta((j-1)\sigma+i+2, j\sigma+i+1))$, with the satellite data $(j\sigma+i+2, a)$ (and implicitly increment m by 1), and set $\text{pos}[a] = m$.

Algorithm - the result of applying Lemma 6.5. One can show exactly as in the case of Theorem 6.6 that after executing the σ operations $o_1, \ldots, o_\sigma$, we have computed the values $M[j\sigma+1][t]$, $M[j\sigma+2][t], \ldots, M[(j+1)\sigma][t]$ correctly. We can move on to phase $j+1$ and repeat this process.

The result. The minimal number of substitutions needed to make w k-universal is correctly computed as $M[n][k]$.

Complexity. By Lemma 6.5, computing $M[j\sigma+1][t], M[j\sigma+2][t], \ldots, M[(j+1)\sigma][t]$ takes $O(\sigma)$ for each j. Overall, computing the entire column $M[\cdot][t]$ takes $O(n)$ time. We do this for all $t \leq k$, and we obtain $O(nk)$ time in total to compute all elements of the matrix M.

Case 2. Now let us assume that $\iota(w) > k$.

General approach. We will show that the minimal number of substitutions needed to obtain a k-universal word from w equals the minimal number of deletions needed to obtain a k-universal word from w, and then use the algorithm from Theorem 6.7 to compute it.

The proof. First of all, the comments made in the opening of Section 6.3 explain why the minimal number of substitutions needed to obtain a k-universal word from w is lower bounded by the minimal number of deletions needed to obtain a k-universal word from w. Indeed, in a transformation of w, of universality index $\iota(w)$, into a word w', of universality index k, using s substitutions, we can replace all these substitutions by deletions and get a word w'' that has universality index lower or equal to k. Thus reaching a k-universal word from w requires s deletions or less.

To see that, in fact, the minimal number of substitutions needed to obtain a k-universal word from w equals the minimal number of deletions needed to obtain a k-universal word from w, we proceed as follows.

Let $ar_w(1)\cdots ar_w(\iota(w))r_w$ be the arch factorisation of w. Let w' be a k-universal word that is obtained from w using a minimal number of deletions, and let $w' = \text{ar}_{w'}(1)\cdots\text{ar}_{w'}(k)r_{w'}$ be its arch factorisation. Clearly, w' is a subsequence of w, so we can identify the list of deleted positions of w, as well as the position i_j of w which corresponds to the last symbol of $\text{ar}_{w'}(j)$ for $j \in [1:k]$; that is, $\text{ar}_{w'}(1)\cdots\text{ar}_{w'}(j)$ is obtained using the respective deletions from $w[1:i_j]$. Let $i_0 = 0$. Now, we can associate the deleted positions to the arches of w' in the following way: if i is a position of w such that $w[i]$ was deleted and $i \in [i_{j-1}+1 : i_j]$, then i is associated to $\text{ar}_{w'}(j)$.

Now, in the case of substitutions, instead of deleting letters of w, we will replace any deleted letter $w[i]$ of w by a letter different from the last letter of $\mathrm{ar}_{w'}(j)$, namely $\mathrm{ar}_{w'}(j)[|\mathrm{ar}_{w'}(j)|]$, where $\mathrm{ar}_{w'}(j)$ is the arch of w' associated to position i. Let w'' be the word obtained in this way. By Definition 2.28 w'' has an arch factorisation with exactly k archs (the j^{th} arch of this factorisation ends with the letters $\mathrm{ar}_{w'}(j)[|\mathrm{ar}_{w'}(j)|]$ from the corresponding arch of w').

This shows that the minimal number of substitutions needed to obtain a k-universal word from w is lower or equal to the minimal number of deletions needed to obtain a k-universal word from w, and, as we have also shown the opposing inequality, these numbers must be equal.

With this, the analysis of both cases is finished, and it follows that the statement of the theorem holds. □

Note that, while substitutions and deletions can be used similarly to decrease the universality index of a word, we always need at least as many substitutions as insertions to increase it. To see that this inequality can also be strict, note that one insertion is enough to make aabb 2-universal, but we need two substitutions to achieve the same result.

6.4 Space Efficient Implementation

We presented a series of algorithms computing the minimal number of edits one needs to apply to a word w in order to reach k-subsequence universality. In fact, one can extend our algorithms and, using additional $O(k|\mathrm{alph}(w)|)$ time, we can effectively construct a k-universal word which is closest to w, with respect to the edit distance. All our algorithms can be implemented in linear space using a technique called *Hirschberg's trick* [88].

Remark 6.9. *Assume $k \leq n$. In the proofs of Theorems 6.6 and 6.8 and, respectively, in the proof of Theorem 6.7 the main part is computing the matrices M and respectively N. This requires $O(nk)$ time and space. However, as computing the columns $M[\cdot][p]$ and $N[\cdot][p]$ only requires knowing the values on columns $M[\cdot][p-1]$ and, respectively, $N[\cdot][p-1]$, we can reduce the space consumption to $O(n)$. So, if we are only interested in computing the minimal number of insertions, deletions, or substitutions required to transform w into a k-universal word, $O(n)$ space and $O(nk)$ time are enough.*

The case $k > n$ is only relevant when we want to compute the minimal number of insertions required to transform w into a k-universal word. As explained in the proof of Theorem 6.6, this can be computed in $O(n^2)$ time to which we add the time needed to compute $\sigma(k-n)$. Similarly, by the observations made above, we can compute the minimal number of insertions required to transform w into a k-universal word in $O(n)$ space (the computation of the matrix M) to which we add the space needed to compute $\sigma(k-n)$. □

Theorem 6.10. *Let w be a word, with $|w| = n$, alph$(w) = \Sigma$, and $\Sigma = \{1, 2, \ldots, \sigma\}$. Let $k \neq \iota(w)$ be an integer. We can construct one of the k-universal words which are closest to w w.r.t. edit distance in $O(kn)$ time, if $k \leq n$, and $O(n^2 + k\sigma)$ time, otherwise. The space needed for this construction is $O(n + k\sigma)$*

Proof. The initial algorithm. We first explain how one of the k-universal words which are closest to w w.r.t. edit distance can be constructed in $O(kn)$ time, without fulfilling the space complexity restriction. We will split the discussion in two cases.

Case 1. $k > \iota(w)$. In this case, we only need insertions to produce one of the k-universal words which are closest to w w.r.t. edit distance. So we will use the algorithm in Theorem 6.6 to construct this word. We assume that we use the same notations as in the proof of the respective theorem. In the referenced algorithm we compute, for each position ℓ of w and each $t \leq \min\{n, k\}$ a value j_ℓ such that

$$M[\ell][t] = M[j_\ell][t-1] + (\sigma - \Delta(j_\ell + 1, \ell) = \min\{M[\ell'][t-1] + (\sigma - \Delta(\ell' + 1, \ell)) \mid \ell' \leq \ell\}.$$

We define $Sol[\ell][t] \leftarrow j_\ell$.

Let us assume first $k \leq n$. Define the sequence $i_{k-1} = Sol[n][k]$ and, for $j \in [2, k-1]$, $i_{j-1} = Sol[i_j][j]$. Let $i_0 = 0$ and $i_k = n$. It is not hard to see that for the decomposition $w = v_1 \cdots v_k$, where, for $j \in [1 : k-1]$, $v_j = w[i_{j-1} + 1 : i_j]$, the following holds: $\sum_{j \in [1:k]} (\sigma - \Delta(i_{j-1} + 1, i_j)) = M[n][k]$. In other words, if we insert the minimal number of letters in each of the words $v_1, \ldots, v_k$ such that they become universal, then we obtain one of the k-universal words which are closest to w w.r.t. the edit distance.

Clearly, the sequence of words $v_1 = [1 : i_1]$, $v_2 = [i_1 + 1 : i_2]$, ...,$v_k = [i_{k-1} + 1 : i_k]$ can be computed in linear time $O(n)$ once we have the matrix Sol.

Now, to compute a k-universal word obtained from w by making each of the words $v_1, \ldots, v_k$ universal with a minimal number of insertions, we do the following. For each $i \in [1 : k]$, by traversing the word v_i left to right we can identify in $O(|v_i| + \sigma)$ the subset V_i of Σ containing the letters which do not occur in v_i (e.g., using a counting vector like in the proof of Lemma 6.1). We produce a word u_i from v_i by appending the letters from V_i at the end of v_i; this takes $O(|v_i| + \sigma)$ time. Then we concatenate the words $u_1, \ldots, u_k$ and obtain a word u which is k-universal and the number of insertions needed to obtain u from w is $M[n][k]$. The total time needed to produce u is $O(k\sigma)$.

Let us now assume that $k > n$. Just like above, we obtain an n-universal word u from w. Then we concatenate at the end of u the word $(1 \cdot 2 \cdots \cdot \sigma)^{k-n}$. In this way, we obtain a word u' which is k-universal and the number of insertions needed to obtain u from w is minimal, i.e., $M[n][k] + (k-n)\sigma$. The total time needed to produce u' is $O(n^2 + k\sigma)$.

This concludes the analysis of Case 1.

Case 2. $k < \iota(w)$. In this case, we only need deletions to produce one of the k-universal words which are closest to w w.r.t. edit distance. So we will use the algorithm in Theorem 6.7 to construct this word. We assume that we use the same notations as in the proof of the respective theorem. In the algorithm described in the proof of Theorem 6.7 we compute, for $i \in [1 : n], p \in [2 : k]$, a value j_i such that

$$N[i][p] = N[j_i][p-1] + |w[j_i + 1 : i]|_{w_i} = \min\{N[i'][p-1] + |w[i' + 1 : i]|_{w_i} \mid i' \in [1 : \text{univ}[i] - 1]\}.$$

We define $Sol[i][p] \leftarrow j_i$. We have $Sol[i][1] = 0$.

Finally, to return the minimal number of deletions needed to transform w into a k-universal word, we compute $m = \arg\min\{N[i][k] + T[i] \mid 1 \leq i \leq n\}$.

We now define the sequence $i_k = m$, $i_{j-1} = Sol[i_j][j]$, with $j \geq 2$, and $i_0 = 1$. Let $i_{k+1} = n$. It is not hard to see that for the decomposition $w = v_1 \cdots v_k v_{k+1}$, where, for $j \in [1 : k+1]$ and $v_j = w[i_{j-1} + 1 : i_j]$, the following holds: $\sum_{i\in[1:k]}(|v_i|_{v_i[|v_i|]} + 1) + |v_k|_{w[m]} = \min\{N[i][k] + T[i] \mid 1 \leq i \leq n\}$.

Clearly, the sequence of words $v_1, \ldots, v_k, v_{k+1}$ can be computed in linear time $O(n)$ once we have the matrix Sol. To compute a k-universal word obtained from w by making each of the words $v_1, \ldots, v_k$ universal with a minimal number of deletions, we just remove from v_i all the occurrences of their last letter (i.e., $v_i[|v_i|]$), except the rightmost one. This takes $O(n)$ time, and we can output the word obtained by this procedure as one of the k-universal words which are closest w.r.t. edit distance to w.

A space efficient implementation. We will only discuss in detail how the case when $n \geq k > \iota(w)$ is implemented, as all the other cases can be approached in exactly the same manner.

We know by Remark 6.9 that the matrix M can be computed in linear space $O(n)$. However, it is unclear how we can compute the sequence $i_0, i_1, \ldots, i_k$ in linear space. In particular, in the approach described above we explicitly need to compute and store all the elements in the matrix.

Fortunately, there exists a standard way to deal with this problem (known as *Hirschberg's trick* [88]).

We will need to define more formally the problem that we want to solve in this framework.

We want to solve the problem $\mathcal{P}(w)$ which requires, for the input word w of length n, to compute the smallest number m of insertions needed to transform w into a k-universal word **and** the sequence of positions $i_0 = 1, i_1, \ldots, i_{k-1}, i_k = n$ such that we can transform each of the words $w[i_{j-1} : i_j]$, with $j \in [1 : k]$, into a universal word, using in total exactly m insertions.

The solution of $\mathcal{P}(w)$ is the following.

Firstly, we will compute the value m as described in the proof of Theorem 6.6, using only linear space as described in Remark 6.9, but, alongside m, we will also compute the value $i_{\lfloor k/2 \rfloor}$.

This can be done by computing (the columns of) an additional matrix H, simultaneously with (the columns of) the matrix M. Recall that computing $M[\ell][t]$ is based on identifying a position j_ℓ such that $M[\ell][t] = M[j_\ell][t-1] + (\sigma - \Delta(j_\ell + 1, \ell)$. Then $H[\ell][t]$ is defined as follows:

$$H[\ell][t] = \begin{cases} \infty & \text{if } t < \lfloor k/2 \rfloor, \\ H[j_\ell][t-1] & \text{if } t > \lfloor k/2 \rfloor, \\ j & \text{if } t = \lfloor k/2 \rfloor. \end{cases}$$

Intuitively, given that $M[\ell][t]$ is the minimal number of insertions needed to transform $w[1:\ell]$ into a t-universal word, then $H[\ell][t]$ is the ending position of the prefix of $w[1:\ell]$ which was transformed in a $\lfloor k/2 \rfloor$-universal word when the respective sequence of length $M[\ell][t]$ of insertions is applied to $w[1:\ell]$.

Clearly, to compute the elements of the column $H[\cdot][t]$ of the matrix H we only need column $H[\cdot][t-1]$ (and the columns $M[\cdot][t]$ and $M[\cdot][t-1]$ of matrix M). So, we can compute $H[n][k]$ in linear space and $O(nk)$ time. Also, it is not hard to see that $H[n][k]$ is exactly the value $i_{\lfloor k/2 \rfloor}$. In fact, there may be more solutions to our problem $\mathcal{P}(w,k)$, so $H[n][k]$ corresponds to the position $i_{\lfloor k/2 \rfloor}$ in one of these solutions.

To compute the rest of the values $i_0, \ldots, i_{\lfloor k/2 \rfloor - 1}, i_{\lfloor k/2 \rfloor + 1}, \ldots, i_k$ we proceed in a divide and conquer manner. We solve $\mathcal{P}(w[1:i_{\lfloor k/2 \rfloor}], \lfloor k/2 \rfloor)$ and obtain the sequence $i_0, \ldots, i_{\lfloor k/2 \rfloor - 1}, i_{\lfloor k/2 \rfloor}$, and then we solve $\mathcal{P}(w[i_{\lfloor k/2 \rfloor} + 1 : n], \lceil k/2 \rceil)$ and we obtain the sequence $i_{\lfloor k/2 \rfloor}, i_{\lfloor k/2 \rfloor + 1}, \ldots, i_k$. Then we simply return $i_0, \ldots, i_{\lfloor k/2 \rfloor - 1}, i_{\lfloor k/2 \rfloor}, i_{\lfloor k/2 \rfloor + 1}, \ldots, i_k$ and the value $M[n][k]$ as a solution to $\mathcal{P}(w,k)$.

The correctness of the algorithm is based on the following simple remark: if $w[1:n]$ can be transformed into a k-universal word by a sequence of m insertions, such that in the respective sequence of insertions transforms $w[1:i_{\lfloor k/2 \rfloor}]$ into a $\lfloor k/2 \rfloor$-universal word, then the following hold:

- $w[1:i_{\lfloor k/2 \rfloor}]$ can be transformed into a $\lfloor k/2 \rfloor$-universal word by p insertions;
- $w[i_{\lfloor k/2 \rfloor} + 1 : n]$ can be transformed into a $\lceil k/2 \rceil$-universal word by $m - p$ insertions.

Thus, solving $\mathcal{P}(w,k)$ can be reduced to computing $i_{\lfloor k/2 \rfloor}$ and solving recursively $\mathcal{P}(w[1:i_{\lfloor k/2 \rfloor}], \lfloor k/2 \rfloor)$ and $\mathcal{P}(w[i_{\lfloor k/2 \rfloor} + 1 : n], \lceil k/2 \rceil)$.

The time complexity $T(n,k)$ of solving $\mathcal{P}(w,k)$ is then $T(n,k) = O(nk) + T(i_{\lfloor k/2 \rfloor}, \lfloor k/2 \rfloor) + T(n - i_{\lfloor k/2 \rfloor}, \lceil k/2 \rceil)$. It is easy to show that $T(n,k) \in O(nk)$.

Assume that the algorithm computing $H[n][k]$ and $M[n][k]$ needs cn space for some constant c.

One can show by induction on $n + k$ that the space complexity $S(n, k)$ of solving $\mathcal{P}(w, k)$ is upper-bounded by dn, where $d \geq c$ is a constant, $n = |w|$, and $k \leq n$. The case $n + k = 2$ is trivial. Assume that this is true for $n + k \leq r$. We show that it is true for $n + k = r + 1$. To solve $\mathcal{P}(w, k)$ we first compute the values $M[n][k]$ and $H[n][k]$ in $cn \leq dn$ space, for some constant c. The main observation we make here is that the space we used in this computation can then be reused. Then we solve $\mathcal{P}(w[1 : i_{\lfloor k/2 \rfloor}], \lfloor k/2 \rfloor)$ using $di_{\lfloor k/2 \rfloor} \leq dn$ space (which is actually reused space). Finally, we solve $\mathcal{P}(w[i_{\lfloor k/2 \rfloor} + 1 : n], \lfloor k/2 \rfloor)$ using $d(n - i_{\lfloor k/2 \rfloor}) \leq dn$ space (which is, once more, reused space). Thus, $S(n, k)$ is upper bounded by dn.

This concludes our proof, and shows the statement of the theorem for the case of increasing the universality index of a word w by insertions to a value k with $k \leq |w|$.

We immediately get that in the case of increasing the universality index of a word w by insertions to a value k with $k > |w|$ we need $O(n)$ space to reach n-universality, and the rest of the construction can be done trivially in $O(k\sigma)$ space (we just need to write down the output).

The case of decreasing the universality index of a word w by deletions to a value $k < \iota(w)$ can be treated in an identical way (using the same divide-and-conquer trick). □

6.5 Extensions on the results

The algorithms we presented work in a general setting: the processed words are over an integer alphabet. It seems natural to ask whether faster solutions for inputs over an alphabet of constant size (e.g., binary alphabets) exist. To this end, we state the following two results in the case of insertions as well as increasing the universality by subsitutions in the following section. For readability we will present the utilized data structure in a separate section afterwards.

6.5.1 Efficient Implementation for small Alphabets

Theorem 6.11. *Let w be a word, with $|w| = n$, $alph(w) = \Sigma$, and $\Sigma = \{1, 2, \ldots, \sigma\}$. Let k be an integer $\iota(w) < k$. We can compute the minimal number of insertions needed to apply to w in order to obtain a k-universal word (w.r.t. Σ) in $(n \log^{O(1)} \sigma)$-time.*

Proof. We first define a weighted directed acyclic graph G with the nodes $0, 1, \ldots, n$ and directed edges (i, j) with $i < j$.

Let $\omega(i, j) = \sigma - \Delta(i + 1, j)$ (i.e., the number of letter of Σ which do not appear in $w[i + 1 : j]$) be the weight of the edge (i, j).

Let us show that the number of insertions needed to transform w into a k-universal word equals the weight of a minimum weight k-link path in G (i.e., a path with k edges in the DAG G, starting in node 0, as defined in [20, 5, 148]).

Let the minimum weight k-link path of G be $(0, i_1), (i_1, i_2), \ldots, (i_{k-1}, i_k)$. It has weight $W = \omega(0, i_1) + \omega(i_1, i_2) + \cdots + \omega(i_{k-1}, i_k) = \sigma - \Delta(1, i_1) + \sigma - \Delta(i_1 + 1, i_2) + \cdots + \sigma - \Delta(i_{k-1}, i_k)$. It is straightforward that by doing $\omega(i_{t-1}, i_t)$ insertions we can transform $w[i_{t-1} + 1 : i_t]$ into a 1-universal word (for $0 \leq t \leq k$, and $i_0 = 0$). Thus, with W insertions we can transform w into a word w' with $\iota(w') \geq k$.

Let U be the minimum number of insertions needed to transform w into a word u with $\iota(u) = k$, and let I be the respective sequence of insertions. Assume $U < W$. Consider the arch-factorisation of u into k-archs $u = u_1 u_2 \cdots u_k u_{k+1}$, where $\mathrm{alph}(u_{k+1})$ is a strict subset of Σ. Then, there exists $1 \leq j_1 < j_2 < \ldots < i_{k-1} < i_k \leq n$ such that $w[j_{t-1} + 1 : j_t]$ is transformed using exactly $\sigma - \Delta(j_{t-1} + 1, j_t)$ insertions from the sequence I into u_t, where $1 \leq t \leq k$ and $j_0 = 0$. Consequently, the k-link path $(0, j_1), (j_1, j_2), \ldots, (j_{k-1}, j_k)$ will have weight $U < V$, a contradiction.

In conclusion, the number of insertions needed to transform w into a k-universal word equals the weight of a minimum weight k-link path in G.

Moreover, we can show that G fulfils the concave Monge property. That is, we want to show that $\omega(i, j) + \omega(i + 1, j + 1) < \omega(i + 1, j) + \omega(i, j + 1)$ holds for all $0 < i + 1 < j < n$.

To show that $\omega(i, j) + \omega(i + 1, j + 1) < \omega(i + 1, j) + \omega(i, j + 1)$ means to show that $\sigma - \Delta[i + 1, j] + \sigma - \Delta[i + 2, j + 1] \leq \sigma - \Delta[i + 1, j + 1] + \sigma - \Delta[i + 2, j]$ holds for all for all $0 < i + 1 < j < n$. This can be done easily by case analysis, analysing whether the letter $w[i + 1]$ is contained in $\mathrm{alph}(w[i + 2..j])$.

If $w[i + 1]$ is contained in $\mathrm{alph}(w[i + 2..j])$ then $\sigma - \Delta[i + 1, j] = \sigma - \Delta[i + 2, j]$ and $\sigma - \Delta[i + 2, j + 1] = \sigma - \Delta[i + 1, j + 1]$ and we get that the desired inequality is in fact an equality, so it is true.

If $w[i + 1]$ is not contained in $\mathrm{alph}(w[i + 2..j])$ then $\sigma - \Delta[i + 1, j] = \sigma - \Delta[i + 2, j] - 1$ and $\sigma - \Delta[i + 2, j + 1] - 1 \leq \sigma - \Delta[i + 1, j + 1]$ and we get, once more, the desired inequality.

This shows that G fulfils the concave Monge property.

We now move on to our algorithm. We first construct in $n \log^{O(1)} \sigma$-time the data structures from Lemma 6.13. With these structures, we do not need to construct the graph G explicitly. We can, however, retrieve in $O(\log \sigma / \log \log \sigma)$ the weight of any edge (i, j) of the graph G.

Further, as the DAG G fulfils the concave Monge property), then the minimum weight k-link path in G can be computed in $O(n \log T(\log \sigma / \log \log \sigma))$-time, using the algorithms of [20, 5], where T is the maximum weight of an edge in G.

Adding this up together, the minimum weight k-link path in G can be computed in $n \log^{O(1)} \sigma + O(n \log \sigma (\log \sigma / \log \log \sigma)) = n \log^{O(1)} \sigma$ time. Consequently, the number of insertions needed to transform w into a k-universal word can be computed in $n \log^{O(1)} \sigma$ time. □

Theorem 6.12. *Let w be a word, with $|w| = n$, $alph(w) = \Sigma$, and $\Sigma = \{1, 2, \ldots, \sigma\}$. Let k be an integer $\iota(w) < k \leq \lfloor \frac{n}{\sigma} \rfloor$. We can compute the minimal number of substitutions needed to apply to w in order to obtain a k-universal word (w.r.t. Σ) in $(n \log^{O(1)} \sigma)$-time.*

Proof. As in Theorem 6.11, we define the weighted directed acyclic graph G with the nodes $0, 1, \ldots, n$ and directed edges (i, j) with $i < j$. However, the weight of the edges are defined differently this time. Let $\omega(i, j) = \sigma - \Delta(i + 1, j)$ (i.e., the number of letter of Σ which do not appear in $w[i + 1 : j]$) be the weight of the edge (i, j) if $|w[i + 1 : j]| \geq \sigma$, and $\omega(i, j) = \infty$ if $|w[i + 1 : j]| < \sigma$.

The rest of the proof is identical to that of Theorem 6.11. □

We conclude that, if $\sigma \in O(1)$, then the algorithms of Theorem 6.11 and 6.12 run in optimal linear time $O(n)$.

6.5.2 Universality Queries

We consider the following problem, which we will call *Distinct Letter Counting*: Given a word w, over an alphabet Σ of size σ, construct data structures allowing us to answer queries $\Delta(i, j)$: *"How many distinct letters occur in $w[i..j]$?"*. This problem is, in fact, a reformulation of the problem of *1D-Coloured Range Counting* [116, 134]. We will give an efficient solution for it, which has a query-answering time similar to that from [134], and, moreover, has also an efficient data structure construction-time.

For our result, we will use a well known variant of the *2D-Range Counting* problem [138, 91], which asks to construct data structures for n given points in the two dimensional plane, all having integer coordinates, so that we can answer queries $D(a, b, c, d)$: *"How many input points are located in the rectangle with the lower-left corner (a, b) and upper right-corner (c, d)?"*, where $a \leq c$ and $b \leq d$. Note that the coordinates of the points in this problem do not have to be in any relation to n. For instance, they are not necessarily bounded by $O(n^\ell)$ for some constant ℓ, as it is the case to the integers used as letters of the input word in *Distinct Letter Counting*.

Pătraşcu showed in [138] that a static data structure of size $n \log^{O(1)} n$ for *2D-Range Counting* requires $\Omega(\log n/(\log \log n))$ time per query (in the cell probe model, so also in our computational model). Chan and Pătraşcu [40] showed the following upper bounds: data structures for *2D-Range Counting* can be constructed in $O(n \sqrt{\log n})$ time (and space) such that each query is answered in $O(\log n/(\log \log n))$ time. Based on this, we can show that the following theorem holds.

Theorem 6.13. *Given a word w over an alphabet Σ, with $|\Sigma| = \sigma$ and $|w| = n$, we can construct in $(n \log^{O(1)} \sigma)$-time data structures for* Distinct Letter Counting *with input w, allowing us to answer each query in $O(\log \sigma/(\log \log \sigma))$ time.*

Proof. For simplicity, let us assume that there exists k such that $n = 2^k$.

In a preprocessing phase that takes linear time we compute an array storing $\lfloor \log_2 i \rfloor$, for all $i \in [n]$. As such, we can assume that $\lfloor \log_2 i \rfloor$ can be computed in $O(1)$ time in the rest of this algorithm.

Let p be a number such that $2^{p-1} < \sigma \le 2^p$. In this context, we will assume for simplicity that $p < k$; this can be achieved by processing as input word ww instead of w, for instance.

We now describe the first phase of our algorithm.

For each of the positions $j = i2^p$, with $i \in [2^{k-p} - 1]$ we can construct in $O(\frac{n\sigma}{2^p}) = O(n)$ time (following the same ideas as in the proof of Lemma 6.4 or of Theorem 2.27) the list containing the rightmost occurrences $x_{\mathtt{a}}$ of every letter $\mathtt{a} \in \Sigma$ in $w[1..i2^p]$, as well as the list containing the leftmost occurrences $y_{\mathtt{a}}$ of every letter $\mathtt{a} \in \Sigma$ in $w[i2^p + 1..n]$. The construction of these lists consists in traversing once the word left to right while maintaining the list of the last seen occurrence of each letter, to obtain the $x_{\mathtt{a}}$ values for each of the considered positions, and then a similar traversal right to left to obtain the values $y_{\mathtt{a}}$ for all considered positions. So, this takes linear time.

We now describe the second phase of our algorithm. In this phase we associate a *2D-Range Counting* structure to the pairs of factors of length 2^ℓ occurring around position $i2^\ell$, for $i \in [\frac{n}{2^\ell}]$ and ℓ between p and k.

For ℓ from $k - 1$ down to p, and for each position $j = i2^\ell$ with $i \in [\frac{n}{2^\ell} - 1]$, we do the following. Select from the list of rightmost occurrences $x_{\mathtt{a}}$ of every letter in $w[1..i2^p]$, that we computed for the position $i2^\ell$, the list of occurrences contained in $w[(i-1)2^\ell + 1..i2^\ell]$. For the letters $\mathtt{a}$ that do not occur in $w[(i-1)2^\ell + 1..i2^\ell]$ we set $x_{\mathtt{a}} = n + 1$. Select from the list containing the leftmost occurrences $y_{\mathtt{a}}$ of every letter $\mathtt{a} \in \Sigma$ in $w[i2^\ell + 1..n]$ the list of occurrences contained in $w[i2^\ell + 1..(i+1)2^\ell]$. For the letters $\mathtt{a}$ that do not occur in $w[i2^\ell + 1..(i+1)2^\ell]$ we set $y_{\mathtt{a}} = 0$. This can be done in $O(\sigma)$ time. For each letter $\mathtt{a} \in \Sigma$, if either $x_{\mathtt{a}} \neq n + 1$ or $y_{\mathtt{a}} \neq 0$, we add the point $(x_{\mathtt{a}}, y_{\mathtt{a}})$ to a set of points S_{i2^ℓ} (which is initially empty). Then, we construct data structures for answering *2D-Range Counting* queries for S_{i2^ℓ}.

This process takes $\sigma \log^{O(1)} \sigma$ time, for each position $i2^\ell$, with $i \in [\frac{n}{2^\ell} - 1]$ and $p \le \ell \le k - 1$. In total it takes

$$\sum_{p \le \ell \le k-1} \frac{n\sigma \log^{O(1)} \sigma}{2^\ell} \in \left(\frac{n\sigma \log^{O(1)} \sigma}{2^p}\right) = (n \log^{O(1)} \sigma) \text{ time.}$$

We now describe the third phase of our algorithm. In this phase we associate a *2D-Range Counting* structure to the pairs of factors of length 2^ℓ occurring around position $i2^\ell$, for $i \in [\frac{n}{2^\ell}]$ and $\ell < p$.

For this phase of our algorithm we will use two arrays $C[\cdot]$ and $D[\cdot]$ with σ elements each (indexed by the letters of Σ). All the values stored in C and D are initially null.

Finally, for ℓ from $p-1$ down to 0, and for each position $j = i2^\ell$ with $i \in [\frac{n}{2^\ell} - 1]$, we compute the list of rightmost occurrences $x_\mathtt{a}$ of every letter $\mathtt{a} \in \Sigma$ inside $w[(i-1)2^\ell + 1..i2^\ell]$; we also compute the list of the leftmost occurrences $y_\mathtt{a}$ of every letter $\mathtt{a} \in \Sigma$ inside $w[i2^\ell + 1..(i+1)2^\ell]$. If a occurs in $w[(i-1)2^\ell + 1..i2^\ell]$ we set $C[\mathtt{a}] = x_\mathtt{a}$. If a occurs in $w[i2^\ell + 1..(i+1)2^\ell]$ set $D[\mathtt{a}] = y_\mathtt{a}$. Moreover, if $C[\mathtt{a}]$ stores a position in $w[(i-1)2^\ell + 1..i2^\ell]$ then we add $(x_\mathtt{a}, y_\mathtt{a})$ to the set S_{i2^ℓ} (which is initially empty); otherwise, we add $(n+1, y_\mathtt{a})$ to S_{i2^ℓ}. Similarly, if a occurs in $w[(i-1)2^\ell + 1..i2^\ell]$ and $D[\mathtt{a}]$ does not store a position of $w[i2^\ell + 1..(i+1)2^\ell]$ we add $(x_\mathtt{a}, 0)$ to S_{i2^ℓ}. This entire process, including the construction of S_{i2^ℓ}, can be done in $O(2^\ell)$ time. We construct *2D-Range Counting* structures for S_{i2^ℓ}. The time needed for this is $O(2^\ell \text{poly}(\ell))$.

Running this process for all positions $i2^\ell$, with $i \in [\frac{n}{2^\ell} - 1]$ and $\ell \in [p-1]_0$, takes $O(n \log \sigma)$ (the construction of the sets of points for all the positions) to which we add the total time needed to construct the *2D-Range Counting* data structures. For a fixed ℓ this takes $O((\frac{n}{2^\ell} - 1)2^\ell \text{poly}(\ell)) = O(n\text{poly}(\ell))$; so the time needed to construct these structures for all $\ell \leq p-1$ is $(n \log^{O(1)} \sigma)$ (as ℓ ranges from 1 to $p - 1 \in O(\log \sigma)$).

At this point we have constructed all the data structures that we need. The total time used is $(n \log^{O(1)} \sigma)$.

To answer a query we proceed as follows. Assume we have to answer $\Delta(i, j)$.

First, we want to see which of the constructed *2D-Range Counting* data structures we will use for this. We need to identify an ℓ such that there exists r with $(r-1)2^\ell + 1 \leq i \leq r2^\ell$ and $r2^\ell + 1 \leq j \leq (r+1)2^\ell]$. It is easy to see that we can take ℓ to be the most significant bit where $i-1$ and $j-1$ differ. We compute $b = (i-1)\text{xor}(j-1)$ and let $\ell = \lfloor \log_2 b \rfloor$. Then compute $r = \lfloor (i-1)/2^\ell \rfloor + 1$. So we will use the *2D-Range Counting* structure computed for the position $r2^\ell$ (for which we will answer queries $D(\cdot, \cdot, \cdot, \cdot)$.

Now answer a query $\Delta(i, j)$ with $i \leq 2^{k-1}$ and $j > 2^{k-1}$, it is enough to return $D(i, 0, n+1, n+1) + D(0, 0, n+1, j) - D(i, 0, n+1, j)$, i.e., count by the inclusion-exclusion principle how many of the points $(x_\mathtt{a}, y_\mathtt{a})$ of S_k have $x_\mathtt{a} \geq i$ or $y_\mathtt{a} \leq j$, or, in other words, how many distinct letters of Σ occur in $w[i..j]$. □

6.6 Considerations on the computational model

In this section we will discuss some theoretical modifications of the presented problems.

General algorithmic framework. It is interesting to see whether we can extend our results for more general input alphabets, so when we drop the assumption that if the input word is w, then w is over the alphabet $\Sigma = \{1, \ldots, \sigma\}$ with $\sigma \leq |w|$. In the next paragraph we will follow the similar discussion made in [76].

A more general computational model sometimes used in string algorithms assumes that the input is over general ordered alphabets (see [108, 109, 75] and the references therein). More precisely, the input is a sequence of elements from a totally ordered set $\mathcal{U}$ (i.e., string over $\mathcal{U}$). The operations allowed in this model are those of the standard Word RAM model, with one important restriction: the elements of the input cannot be directly accessed nor stored in the memory used by the algorithms; instead, we are only allowed to *compare* (w.r.t. the order in $\mathcal{U}$) any two elements of the input, and the answer to such a comparison-query is retrieved in $O(1)$ time. In this model, it holds that sorting the elements of an input sequence requires at least $\Omega(n \log n)$ comparisons. An implementation of our algorithms, where we first sort the letters of the input word, map them to words over $\{1, \ldots, n\}$, and then use the same strategies as the ones described for the case of integer alphabets, would require some additional $O(n \log n)$ computational time, due to the sorting.

In fact, one cannot hope to go under $\Omega(n \log n)$ comparisons in the respective model of computation. Indeed, in this framework, it also holds that testing the equality of two sets of size $O(n)$ requires $\Omega(n \log n)$ comparisons [54]. We can show the following lower bounds.

Theorem 6.14. *Let w be a word, with $|w| = n$, alph$(w) = \Sigma$, and universality index $\iota(w)$. Let k be an integer with $n \geq k$. Computing the minimal number of insertions (respectively, deletions, or substitutions) needed to transform w into a k-universal word requires $\Omega(n \log n)$ comparisons (so $\Omega(n \log n)$ time as well).*

Proof. Let $S = \{s_1, \ldots, s_n\}$ and $T = \{t_1, \ldots, t_m\}$ two sets, with $m \leq n$. We define the alphabet $\Sigma = S \cup T \cup \{\$, \#\}$, where the letters \$ and # do not occur in S. We define the word

$$w = \$s_1 \cdots s_n\#\$t_1 \cdots t_m\#.$$

We want to show that $S = T$ if and only if the number of insertions needed to transform w into a 2-universal word is 0.

The left to right implication is trivial. The right to left implication is also easy to show. If w is 2-universal, then it has two arches. These arches must be $\$s_1 \cdots s_n\#$ and $\$t_1 \cdots t_m\#$ (otherwise one would need to insert one of the separators). This means that the letters $s_1, \ldots, s_n$ are the same as $t_1, \ldots, t_m$. So $S = T$.

Thus, to check the equality $S = T$ we can compute the minimal number of insertions needed to make w 2-universal. Thus, this requires at least $\Omega(|w| \log |w|)$ comparisons. As $|w| = n + m + 4 \in O(n)$, the statement follows.

We can similarly show that $S = T$ if and only if the minimal number of substitutions needed to transform w into a 2-universal word is 0. Similarly to the case of insertions, the lower bound is easily obtained now.

Finally, we can show that $S \neq T$ if and only if the minimal number of deletions needed to transform $w' = s_1 \cdots s_n t_1 \cdots t_m$ into a 0-universal word (w.r.t. $\mathrm{alph}(w')$) is exactly 1. Indeed, each element s_i occurs exactly once in S and each element t_j occurs exactly once in T. So, each letter s_i (respectively, t_j) may occur at most twice in w' (if it is contained in both S and T). Clearly, if there is a letter that occurs exactly once, then the minimal number of deletions needed to transform $w' = s_1 \cdots s_n t_1 \cdots t_n$ into a 0-universal word is 1, but also this letter occurs only in S or only in T. So $T \neq S$. If all letters occur twice, then the minimal number of deletions needed to transform $w' = s_1 \cdots s_n t_1 \cdots t_n$ into a 0-universal word is 2, and $T = S$.

Thus, to check the equality $S = T$ we can compute the minimal number of deletions needed to make w' 0-universal. Thus, this requires at least $\Omega(|w| \log |w|)$ comparisons as well. As $|w| = n + m \in O(n)$, the statement follows. □

So, having faster algorithms in this model of computation requires finding better methods than the dynamic programming approach we used. The approaches in Theorem 6.11 and 6.12 are, for the case when we want to increase the universality index of a word, optimal in this model of computation (up to polylog-factors).

In an intermediate model, we can assume that the input is a sequence of elements from a totally ordered set $\mathcal{U}$ (i.e., string over $\mathcal{U}$) whose elements can be stored in a constant number of memory words. The operations allowed in this model are those of the standard Word RAM model. So, in other words, the letters are from $[1 : n^d]$ for some constant d, if the input word has length n. An implementation of our dynamic programming algorithms, where we first sort the letters of the input word, map them to words over $\{1, \ldots, n\}$, and then use the same strategies as the ones described for the case of integer alphabets, runs in exactly the same complexity as stated in the main part of this work, as a set of n numbers from $[1 : n^d]$, where d is a constant, can be sorted in $O(n)$ time using Radix-sort.

Algorithm 8: The efficient algorithm from Theorem 6.7 (on deletions).

```
Input   : word $w$, alphabet $\Sigma$, int $k$
Output : minimal number of deletions
// initialization
int $n \leftarrow |w|$; int $\sigma \leftarrow |\Sigma|$;
int $N[n][k] = \infty$;
// initialise first column of $N$
for $i = 1$ to $n$ do
    if $\Delta(1, i) = \sigma$ (i.e., $w[1:i]$ is 1-universal) then
        $N[i][1] \leftarrow \mathrm{freq}[i] - 1$;
    end
end
// efficient variant
for $p = 2$ to $k$ do
    int $M'[n]$;
    for $i = 1$ to $(p-1)\sigma$ do
        $M'[i] \leftarrow \infty$;
    end
    for $i = (p-1)\sigma$ to $n$ do
        int $l \leftarrow \mathrm{last}_{i-1}[w[i]]$;
        if $l = n + 1$ then
            $M'[i] \leftarrow \infty$;
        else
            int $r \leftarrow \mathrm{RMQ}_{p-1}(l+1, i-1)$;
            $M'[i] \leftarrow 1 + \min\{M'[l], N[r][p-1]\}$;
        end
    end
    // compute $N[\cdot][p]$ using $M'$
    for $i = 1$ to $n$ do
        int $j \leftarrow \mathrm{univ}[i]$;
        if $j = 0$ then
            $N[i][p] \leftarrow \infty$;
        else
            int $t \leftarrow \mathrm{last}_{j-1}[w[i]]$;
            if $t = n + 1$ then
                $N[i][p] \leftarrow \infty$;
            else
                int $r \leftarrow \mathrm{RMQ}_{p-1}(t+1, j-1)$;
                $N[i][p] \leftarrow \min\{M'[t] + \mathrm{freq}[i] - \mathrm{freq}[t] - 1, N[r][p-1] + \mathrm{freq}[i] - \mathrm{freq}[t] - 1\}$;
            end
        end
    end
end
return $\min\{N[i][k] + T[i] \mid 1 \leq i \leq n\}$;
```

Algorithm 9: The efficient algorithm from Case 1 of Theorem 6.8 (on substitutions).

Input : word w, alphabet Σ, int k
Output : minimal number of substitutions

```
// initialisation
int $n \leftarrow |w|$; int $\sigma \leftarrow |\Sigma|$;
int $M[n][k] = \infty$;
// initialise first column of $M$
for $l = \sigma$ to $n$ do
    $M[l][1] \leftarrow \sigma - \Delta(1, l)$;
end
// efficient variant
for $t = 2$ to $k$ do
    // $\leq (n-1)/\sigma$ phases
    for $j = 0$ to $(n-1)/\sigma$ do
        int $A[\sigma][3]$ (list of triples including satellite data); int pos$[\sigma]$;
        for $a = 1$ to $\sigma$ do
            // $S_\ell = \{\mathrm{last}_\ell[a] \mid a \in \mathtt{alph}(w[1:\ell])\}$
            // $\mathfrak{S}_\ell = (S_\ell \cap [(t-1)\sigma : \ell - \sigma]) \cup \{\ell - \sigma + 1\}$
            if $a \in \mathfrak{S}_{j\sigma+1} \setminus \{(j-1)\sigma + 2\}$ then
                int $i \leftarrow \sigma - d_{j\sigma+1}[a]$;
                $A[i+1][1] \leftarrow M[\mathrm{last}_{j\sigma+1}[a] - 1][t-1] + i$;
                pos$[a] \leftarrow i + 1$;
                // satellite data for $A[i+1]$
                $A[i+1][2] \leftarrow \mathrm{last}_{j\sigma+1}[a]$;
                $A[i+1][3] \leftarrow a$;

                $A[\sigma][1] \leftarrow M[(j-1)\sigma + 1][t-1] + (\sigma - \Delta((j-1)\sigma + 2, j\sigma + 1))$;
                pos$[w[(j-1)\sigma + 2]] \leftarrow \sigma$;
                // satellite data for $A[\sigma]$
                $A[\sigma][2] \leftarrow (j-1)\sigma + 2$;
                $A[\sigma][3] \leftarrow w[(j-1)\sigma + 2]$;
            end
            if last$_{j\sigma+1}[a] = n + 1$ then
                pos$[a] = 0$;
            end
        end
        // apply sequence of operations as in Theorem 6.5
        for $i = 1$ to $\sigma$ do
            $q \leftarrow$ minimum of $A$;
            $M[j\sigma + i][t] \leftarrow q$;
            $a = w[(j-1)\sigma + i + 2]$;
            decrement positions pos$[a] + 1$, pos$[a] + 2, \ldots, m$ by 1;
            append $M[(j-1)\sigma + i + 1][t-1] + (\sigma - \Delta((j-1)\sigma + i + 2, j\sigma + i + 1))$ to $A$ (that is, set $A[m+1][1]$ to that value);
            // and add satellite data
            $A[m+1][2] \leftarrow j\sigma + i + 2$;
            $A[m+1][3] \leftarrow a$;
            $m \leftarrow m + 1$;
            pos$[a] \leftarrow m$;
        end
    end
end
return $M[n][k]$;
```

CHAPTER 7

Conclusion

In this last Chapter, I will summarize the results of the findings from the previous chapters and give a brief outlook on future work and open problems.

7.1 Results Summary

In this section, I will summarise all the findings from the papers presented in the previous chapters. In Table 7.1 are all results from Chapter 3, i.e., the paper *Matching Patterns with variables under Hamming Distance* [78]. In the leftmost column is the name of the class of patterns that was considered in the respective row (see Section 2.2). In the second column, all the known results from the literature on *(exact) matching* $\texttt{Match}_P$ are displayed for comparison purposes (see Section 2.3). The last two columns list the results for *(approximate) matching* under Hamming distance in the decision variant ($\texttt{HDMatch}_P$) and the minimization variant ($\texttt{MinHDMatch}_P$). The result for $\texttt{HDMatch}_{\texttt{Reg}}$ can be found in Theorem 3.4 (upper bound) and Theorem 3.7 (lower bound). The corresponding result in the case of $\texttt{HDMatch}_{\texttt{Reg}}$ stems from Theorem 3.5. In the row of 1Var are the results of Theorem 3.10 and directly below that Theorem 3.11 for the class NonCross. The upper bound of $\texttt{HDMatch}_{\texttt{1RepVar}}$ is from Theorem 3.12 and the lower bound from Theorem 3.15. In the case of $\texttt{MinHDMatch}_{\texttt{1RepVar}}$, there is also a *PTAS* as shown in Theorem 3.14 and no *EPTAS* as shown in Theorem 3.16. Interestingly, in the case of $\texttt{MinHDMatch}_{\texttt{kLOC}}$ shown in Theorem 3.13, there is no immediate NP-hardness result for a constant value of its parameter k, as it is the case for all the other classes as an immediate corollary of Theorem 3.15.

Class P	$\texttt{Match}_P(w, \alpha)$	$\texttt{HDMatch}_P(w, \alpha, \Delta)$	$\texttt{MinHDMatch}_P(w, \alpha)$
Reg	$O(n)$ [folklore]	$O(n\Delta)$ matching cond. lower bound	$O(nd_{\texttt{HAM}}(\alpha, w))$ matching cond. lower bound
1Var ($\texttt{var}(\alpha) = \{x\}$)	$O(n)$ [folklore]	$O(n)$	$O(n)$
NonCross	$O(nm \log n)$ [59]	$O(n^3 p)$	$O(n^3 p)$
1RepVar k=# x-blocks	$O(n^2)$ [59]	$O(n^{k+2}m)$ W[1]-hard w.r.t. k	$O(n^{k+2}m)$, PTAS W[1]-hard w.r.t. k no EPTAS (if $FPT \neq W[1]$)
kLOC	$O(mkn^{2k+1})$ [51] W[1]-hard w.r.t. k	$O(n^{2k+2}m)$ W[1]-hard w.r.t. k	$O(n^{2k+2}m)$ W[1]-hard w.r.t. k no EPTAS (if $FPT \neq W[1]$)
kSCD	$O(m^2 n^{2k})$ [59] W[1]-hard w.r.t. k	NP-hard for $k \geq 2$	NP-hard for $k \geq 2$
kRepVar	$O(n^{2k})$ [59] W[1]-hard w.r.t. k	NP-hard for $k \geq 1$	NP-hard for $k \geq 1$
k-bounded treewidth	$O(n^{2k+4})$ [143] W[1]-hard w.r.t. k	NP-hard for $k \geq 3$	NP-hard for $k \geq 3$

Table 7.1: The results from Chapter 3 are listed in columns 3 and 4. Note that: $|w| = n$, $|\alpha| = m$, $|\texttt{var}(\alpha)| = p$.

In Table 7.2 below are the findings of Chapter 4, i.e., the paper *Matching Patterns with Variables under Edit Distance*. Since there is no difference in the results of $\texttt{EDMatch}_P$ and $\texttt{MinEDMatch}_P$, I omitted the $\texttt{MinEDMatch}_P$ column and added the $\texttt{HDMatch}_P$ column from the previous Table 7.1

for better comparison of these results. The findings from Chapter 4 can be found in the rightmost column. The first result is based on Theorem 4.3 for the upper bound and Theorem 4.6 for the lower bound. All other entries in that column are the direct results or corollaries of Theorem 4.7. That is because in all of these pattern classes there might exist a variable that occurs an unbounded number of times. Therefore, the hardness result proven for `1Var` extends to the other pattern classes that do not take the overall number of occurrences into account but rather the number of continuous blocks of a variable or how they are arranged, i.e., interleaved.

Class P	$\texttt{Match}_P(w, \alpha)$	$\texttt{HDMatch}_P(w, \alpha, \Delta)$	$\texttt{EDMatch}_P(w, \alpha, \Delta)$
`Reg`	$O(n)$ [folklore]	$O(n\Delta)$, matching cond. lower bound	$O(n\Delta)$, matching cond. lower bound
`1Var` ($\texttt{var}(\alpha) = \{x\}$)	$O(n)$ [folklore]	$O(n)$	$O(n^{3\|\alpha\|_x})$ W[1]-hard w.r.t. $\|\alpha\|_x$
`NonCross`	$O(nm \log n)$ [59]	$O(n^3 p)$	NP-hard
`1RepVar` k=# x-blocks	$O(n^2)$ [59]	$O(n^{k+2}m)$ W[1]-hard w.r.t. k	NP-hard for $k \geq 1$
`kLOC`	$O(mkn^{2k+1})$ [51] W[1]-hard w.r.t. k	$O(n^{2k+2}m)$ W[1]-hard w.r.t. k	NP-hard for $k \geq 1$
`kSCD`	$O(m^2 n^{2k})$ [59] W[1]-hard w.r.t. k	NP-hard for $k \geq 2$	NP-hard for $k \geq 1$
`kRepVar`	$O(n^{2k})$ [59] W[1]-hard w.r.t. k	NP-hard for $k \geq 1$	NP-hard for $k \geq 1$
k-bounded treewidth	$O(n^{2k+4})$ [143] W[1]-hard w.r.t. k	NP-hard for $k \geq 3$	NP-hard for $k \geq 1$

Table 7.2: The results from Chapter 4 are in column 4. The results in column 3 are from Chapter 3. Note that: $|w| = n$, $|\alpha| = m$, $|\texttt{var}(\alpha)| = p$.

From Chapter 5, the following results hold for the problem of *matching patterns with variables under Simon's congruence*. The problem `MatchUniv` is NP-complete due to Theorem 5.1 and in P when a variable occurs only once or a constant bounds the overall amount of variables. Further, the problems `MatchSimon` and `MatchStrictSimon` are NP-complete as shown in Theorem 5.11 and in P when considering the class `Reg`. If the setting is extended to word equations, it was proven that `WESimon` is NP-complete (Theorem 5.13) and `WEStrictSimon` is NP-complete, for $k \leq |\alpha| + |\beta|$ (Theorem 5.15).

In Chapter 6, the problem of computing the *edit distance to k-subsequence universality* was split into the independent computable cases of using insertions in the case when $k \geq \iota(w)$, deletions when $k \leq \iota(w)$, and substitutions for any of the two cases. All of these scenarios can be solved in $O(nk)$ time which follows from Theorems 6.6, 6.7, and 6.8. Note that in the case of insertions, this term might be dominated by the time to compute the value of the product $k\sigma - n$. Further, it is possible to construct a k-universal word with fewest edits with $O(n + k\sigma)$ space in $O(kn)$ time, if $k \leq n$, and $O(n^2 + k\sigma)$ time, otherwise (Theorem 6.10). If $\iota(w) < k$, it is also possible to compute the minimal number of insertions or substitutions needed to obtain a k-universal word in $(n \log^{O(1)} \sigma)$ (Theorems 6.11 and 6.12).

7.2 Open Problems and Future Work

Now that I have summarised the results of this thesis, it remains to state what is left to do from the presented topics and give an outlook on related future work. For that, I would like to present some open problems that my co-authors and I are happy to work and cooperate on and give a broader look at my future work in this area. Let me start with the latter. As stated in the introduction (see Section 1.2), some of these more theoretical problems arose while working on string solving. Currently, my colleagues and I are working on expanding and improving the string solvers *Z3Str3, Z3Str4* and *Z3Alpha*, which are branches from *Microsofts* general *SMT* solver *Z3*. The latter even won a track in this year's string solving competition *SMT-COMP 2023* [1]. Optimally, some of the results presented in this thesis could be incorporated into a practical scenario within this solver. Alternatively to implementing it directly within a big solver (e.g., *Z3str4*), the writing of a smaller string solver that implements some of the theoretical research ideas on real (for this case modified) benchmarks of the competition could be a good next goal. Outside of directly implementing the ideas from this thesis, I would like to investigate other problems related to stringology and string solving, some of which can be found in the following paragraphs.

While the results from Chapter 3, 4, and 5 seem to cover a vast amount of results, there are still some extensions to the work of *matching patterns with variables in approximate settings*. First and foremost, let me name the obvious extensions to consider other string metrics and similarity measures. Suitable candidates for this are the *Dynamic Time Warping Distance* [145], (k-)abelian equivalence [98, 99], or the k-binomial equivalence[144, 68, 118]. Further, it would be interesting to identify new classes of patterns that appear in practice, e.g., data mining existing benchmarks. These classes must not necessarily be as general as the classes discussed in this thesis but could also describe very restricted kinds of patterns, e.g., a constantly bounded number of variables. In the case of Hamming distance, if one considers `Reg` with only two variables, the algorithms from [80, 161] are already faster than the general lower bounds from Theorem 3.7. Investigating enumeration algorithms for the presented settings would also be interesting, which produce a stream of valid substitutions with as little delay between outputs as possible (see survey [164] and references therein).

Furthermore, it would be interesting to complete and strengthen the results from Tables 7.1 and 7.2, in the sense that either the algorithms or the conditional lower bounds are improved. Exciting candidates for these improvements are the following. Firstly, the fine-grained complexity of computing the median string under edit distance for k [89] was shown for inputs over unbounded alphabets; it is interesting to see if the bounds also hold for alphabets of constant size. Moreover, it would be interesting to then extend these results to $\texttt{EDMatch}_{\texttt{1Var}}$. Secondly, it would be good to improve the corresponding upper bound of Theorem 4.9 to the one of the median string algorithm by Sankoff [147]. The last problem that I want to work on (among many other possible open problems), in the context of Hamming and edit distance, is an improved upper and lower bound for $\texttt{HDMatch}_{\texttt{NonCross}}$. Then, in the context of `MatchUniv` and `MatchSimon`, it seems interesting to consider the problem in a parameterized setting, e.g., under nonconstant alphabets of size σ or the number of variables. The conjecture for these cases are $W[1]$-hardness results in both cases [65].

Finally, it seems interesting to investigate some of the applications for *matching patterns with variables* outside of string solving in this new setting. In the area of algorithmic learning theory for some pattern classes, a pattern can be inferred from positive data [154]. As mentioned in the introduction, the first application is to test for an inferred (learned) pattern and a given word: is the word from this pattern language, and if not, how “far” is it? Other applications could be improving the learning process itself or developing cluster algorithms for a given set of strings.

Before moving away from the results of Hamming distance and Edit distance, I would like to make a small excursion to a paper I recently read. In the paper *Faster Approximate Pattern Matching: A Unified Approach* [43], the authors present a computational meta-model for various settings in pattern matching. That is, if one can describe an algorithm with a specific set of meta-operations, called *PILLAR*, then as a consequence of this model, multiple results in various computational models follow directly. Among these are next to the classical word *RAM* model (used in this thesis), a compressed setting, a quantum computing setting, and a dynamic string setting. The following operations make up the *PILLAR* model [43]:

1. $\texttt{Extract}(\texttt{w}, \texttt{l}, \texttt{r})$ retrieves the substring $w[l : r]$ from w.

2. $\texttt{LCP}(\texttt{w}_1, \texttt{w}_2)$ computes the length of the Longest Common Prefix of w_1 and w_2.

3. $\texttt{LCS}(\texttt{w}_1, \texttt{w}_2)$ computes the length of the Longest Common Suffix of w_1 and w_2.

4. $\texttt{IPM}(\texttt{u}, \texttt{w})$ computes indices of starting positions of exact occurrences of u in w.

5. $\texttt{Access}(\texttt{w}, \texttt{i})$ retrieves the character $w[i]$.

6. $\texttt{Length}(\texttt{w})$ returns the length $|w|$ of w.

I conjecture that these operations are sufficient to describe the algorithms in Chapter 3 and 4. Hence, the results would be extended into multiple computational settings. A deeper investigation of this model and its applicability to our results will be done in the near future.

Lastly, I want to state some open problems related to the topics of *the edit distance to k-subsequence universality* from Chapter 6. In the current setting of the problem, a word w' with $\iota(w') = k$ is obtained from a given word *w w.r.t. the alphabet* $\mathtt{alph}(w)$. If the setting is adjusted in a way that the word w' shall be k-universal w.r.t to $\mathtt{alph}(w')$, then the premise of only using deletions to get to a smaller k and insertions to get to a larger k does not hold anymore. The word `bananan` can be edited with 2 insertions to be 3-universal (e.g., `banbananb`), and also with one deletion (e.g., `ananan`). This setting could also be even more generalized by providing the alphabet as input to the algorithm. These modifications seem to be a possible extension of the presented work. Finally, let me restate the original goal of the research on this topic as the last open problem that I will mention: given two words w and u and an integer k, compute efficiently the edit distance from w to the language defined by all words that have the same set of subsequences $\mathbb{S}_k(u)$.

Bibliography

[1] S.-C. 2023. The international satisfiability modulo theories (smt) competition. URL: https://smt-comp.github.io/2023/.

[2] D. Adamson. Ranking and unranking k-subsequence universal words. In A. E. Frid and R. Mercas, editors, *Combinatorics on Words - 14th International Conference, WORDS 2023, Umeå, Sweden, June 12-16, 2023, Proceedings*, volume 13899 of *Lecture Notes in Computer Science*, pages 47–59. Springer, 2023. doi:10.1007/978-3-031-33180-0_4.

[3] D. Adamson, M. Kosche, T. Koß, F. Manea, and S. Siemer. Longest common subsequence with gap constraints. In A. E. Frid and R. Mercas, editors, *Combinatorics on Words - 14th International Conference, WORDS 2023, Umeå, Sweden, June 12-16, 2023, Proceedings*, volume 13899 of *Lecture Notes in Computer Science*, pages 60–76. Springer, 2023. doi:10.1007/978-3-031-33180-0_5.

[4] A. Aggarwal, M. M. Klawe, S. Moran, P. W. Shor, and R. E. Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2:195–208, 1987. doi:10.1007/BF01840359.

[5] A. Aggarwal, B. Schieber, and T. Tokuyama. Finding a minimum-weight k-link path graphs with the concae monge property and applications. *Discret. Comput. Geom.*, 12:263–280, 1994. doi:10.1007/BF02574380.

[6] C. Allauzen and M. Mohri. Linear-space computation of the edit-distance between a string and a finite automaton. *CoRR*, abs/0904.4686, 2009. URL: http://arxiv.org/abs/0904.4686, arXiv:0904.4686.

[7] R. Amadini. A survey on string constraint solving. *ACM Comput. Surv.*, 55(2):16:1–16:38, 2023. doi:10.1145/3484198.

[8] A. Amir, M. Lewenstein, and E. Porat. Faster algorithms for string matching with k mismatches. *J. Algorithms*, 50(2):257–275, 2004. doi:10.1016/S0196-6774(03)00097-X.

[9] A. Amir and I. Nor. Generalized function matching. *J. Discrete Algorithms*, 5:514–523, 2007. doi:10.1016/j.jda.2006.10.001.

[10] D. Angluin. Finding patterns common to a set of strings. *J. Comput. Syst. Sci.*, 21(1):46–62, 1980. doi:10.1016/0022-0000(80)90041-0.

[11] L. A. K. Ayad, C. Barton, and S. P. Pissis. A faster and more accurate heuristic for cyclic edit distance computation. *Pattern Recognit. Lett.*, 88:81–87, 2017. doi:10.1016/J.PATREC.2017.01.018.

[12] A. Backurs and P. Indyk. Which regular expression patterns are hard to match? In *Proc. 57th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2016*, pages 457–466, 2016. doi:10.1109/FOCS.2016.56.

[13] A. Backurs and P. Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). *SIAM J. Comput.*, 47(3):1087–1097, 2018. doi:10.1145/2746539.2746612.

[14] A. Backurs and P. Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). *SIAM J. Comput.*, 47(3):1087–1097, 2018. doi:10.1137/15M1053128.

[15] R. A. Baeza-Yates. Searching subsequences. *Theor. Comput. Sci.*, 78(2):363–376, 1991. doi:10.1016/0304-3975(91)90358-9.

[16] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar. cvc5: A versatile and industrial-strength SMT solver. In D. Fisman and G. Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022. doi:10.1007/978-3-030-99524-9_24.

[17] L. Barker, P. Fleischmann, K. Harwardt, F. Manea, and D. Nowotka. Scattered factor-universality of words. In N. Jonoska and D. Savchuk, editors, *Developments in Language Theory - 24th International Conference, DLT 2020, Tampa, FL, USA, May 11-15, 2020, Proceedings*, volume 12086 of *Lecture Notes in Computer Science*, pages 14–28. Springer, 2020. doi:10.1007/978-3-030-48516-0_2.

[18] C. Barrett, P. Fontaine, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2016.

[19] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011. `doi:10.1007/978-3-642-22110-1\_14`.

[20] W. W. Bein, L. L. Larmore, and J. K. Park. The d-edge shortest-path problem for a monge graph. *UNT Digital Library*, 7 1992. URL: `https://www.osti.gov/biblio/10146169`.

[21] M. A. Bender and M. Farach-Colton. The lca problem revisited. In G. H. Gonnet and A. Viola, editors, *LATIN 2000: Theoretical Informatics*, pages 88–94, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[22] G. Bernardini, H. Chen, G. Loukides, N. Pisanti, S. P. Pissis, L. Stougie, and M. Sweering. String Sanitization Under Edit Distance. In *31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020)*, volume 161 of *LIPIcs*, pages 7:1–7:14, 2020. `doi:10.4230/LIPIcs.CPM.2020.7`.

[23] G. Bernardini, H. Chen, G. Loukides, N. Pisanti, S. P. Pissis, L. Stougie, and M. Sweering. String sanitization under edit distance. In I. L. Gørtz and O. Weimann, editors, *31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020, June 17-19, 2020, Copenhagen, Denmark*, volume 161 of *LIPIcs*, pages 7:1–7:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. `doi:10.4230/LIPICS.CPM.2020.7`.

[24] G. Bernardini, N. Pisanti, S. P. Pissis, and G. Rosone. Approximate pattern matching on elastic-degenerate text. *Theor. Comput. Sci.*, 812:109–122, 2020. `doi:10.1016/j.tcs.2019.08.012`.

[25] M. Berzish, J. D. Day, V. Ganesh, M. Kulczynski, F. Manea, F. Mora, and D. Nowotka. String theories involving regular membership predicates: From practice to theory and back. In T. Lecroq and S. Puzynina, editors, *Combinatorics on Words - 13th International Conference, WORDS 2021, Rouen, France, September 13-17, 2021, Proceedings*, volume 12847 of *Lecture Notes in Computer Science*, pages 50–64. Springer, 2021. `doi:10.1007/978-3-030-85088-3\_5`.

[26] M. Berzish, J. D. Day, V. Ganesh, M. Kulczynski, F. Manea, F. Mora, and D. Nowotka. Towards more efficient methods for solving regular-expression heavy string constraints. *Theor. Comput. Sci.*, 943:50–72, 2023. `doi:10.1016/J.TCS.2022.12.009`.

[27] M. Berzish, M. Kulczynski, F. Mora, F. Manea, J. D. Day, D. Nowotka, and V. Ganesh. An SMT solver for regular expressions and linear arithmetic over string length. In A. Silva and K. R. M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV*

2021, Virtual Event, July 20-23, 2021, Proceedings, Part II, volume 12760 of *Lecture Notes in Computer Science*, pages 289–312. Springer, 2021. doi:10.1007/978-3-030-81688-9_14.

[28] P. Bille and M. Farach-Colton. Fast and compact regular expression matching. *Theor. Comput. Sci.*, 409(3):486–496, 2008. doi:10.1016/j.tcs.2008.08.042.

[29] C. Boucher, C. Lo, and D. Lokshantov. Consensus patterns (probably) has no EPTAS. In *Proc. 23rd Annual European Symposium, ESA*, volume 9294 of *LNCS*, pages 239–250, 2015. doi:10.1007/978-3-662-48350-3_21.

[30] B. Brejová, D. G. Brown, I. M. Harrower, A. López-Ortiz, and T. Vinar. Sharper upper and lower bounds for an approximation scheme for consensus-pattern. In *Proc. 16th Annual Symposium Combinatorial Pattern Matching, CPM 2005*, volume 3537 of *LNCS*, pages 1–10, 2005. doi:10.1007/11496656_1.

[31] B. Brejová, D. G. Brown, I. M. Harrower, and T. Vinar. New bounds for motif finding in strong instances. In *Proc. 17th Annual Symposium Combinatorial Pattern Matching, CPM 2006*, volume 4009 of *LNCS*, pages 94–105, 2006. doi:10.1007/11780441_10.

[32] K. Bringmann. Fine-grained complexity theory (tutorial). In *Proc. 36th International Symposium on Theoretical Aspects of Computer Science, STACS 2019*, volume 126 of *LIPIcs*, pages 4:1–4:7, 2019. doi:10.4230/LIPIcs.STACS.2019.4.

[33] K. Bringmann and B. R. Chaudhury. Sketching, streaming, and fine-grained complexity of (weighted) LCS. In S. Ganguly and P. K. Pandya, editors, *38th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2018, December 11-13, 2018, Ahmedabad, India*, volume 122 of *LIPIcs*, pages 40:1–40:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICS.FSTTCS.2018.40.

[34] K. Bringmann, F. Grandoni, B. Saha, and V. V. Williams. Truly sub-cubic algorithms for language edit distance and rna-folding via fast bounded-difference min-plus product. In I. Dinur, editor, *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*, pages 375–384. IEEE Computer Society, 2016. doi:10.1109/FOCS.2016.48.

[35] K. Bringmann and M. Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In *Proc. 56th IEEE Annual Symposium on Foundations of Computer Science, FOCS*, pages 79–97, 2015. doi:10.1109/FOCS.2015.15.

[36] K. Bringmann and M. Künnemann. Multivariate fine-grained complexity of longest common subsequence. In *Proc. 29th ACM-SIAM Symposium on Discrete Algorithms, SODA 2018*, pages 1216–1235. SIAM, 2018. `doi:10.1137/1.9781611975031.79`.

[37] L. Bulteau and M. L. Schmid. Consensus strings with small maximum distance and small distance sum. *Algorithmica*, 82(5):1378–1409, 2020. `doi:10.1007/s00453-019-00647-9`.

[38] C. Câmpeanu, K. Salomaa, and S. Yu. A formal study of practical regular expressions. *Int. J. Found. Comput. Sci.*, 14:1007–1018, 2003. `doi:10.1142/S012905410300214X`.

[39] K. Casel, J. D. Day, P. Fleischmann, T. Kociumaka, F. Manea, and M. L. Schmid. Graph and string parameters: Connections between pathwidth, cutwidth and the locality number. In *Proc. 46th International Colloquium on Automata, Languages, and Programming, ICALP 2019*, volume 132 of *LIPIcs*, pages 109:1–109:16, 2019. `doi:10.4230/LIPIcs.ICALP.2019.109`.

[40] T. M. Chan and M. Puatracscu. Counting inversions, offline orthogonal range counting, and related problems. In M. Charikar, editor, *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pages 161–173. SIAM, 2010. `doi:10.1137/1.9781611973075.15`.

[41] P. Charalampopoulos, T. Kociumaka, and S. Mozes. Dynamic String Alignment. In *31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020)*, volume 161 of *LIPIcs*, pages 9:1–9:13, 2020. `doi:10.4230/LIPIcs.CPM.2020.9`.

[42] P. Charalampopoulos, T. Kociumaka, and P. Wellnitz. Faster approximate pattern matching: A unified approach. In *Proc. 61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020*, pages 978–989, 2020. `doi:10.1109/FOCS46700.2020.00095`.

[43] P. Charalampopoulos, T. Kociumaka, and P. Wellnitz. Faster approximate pattern matching: A unified approach. In S. Irani, editor, *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020*, pages 978–989. IEEE, 2020. `doi:10.1109/FOCS46700.2020.00095`.

[44] P. Charalampopoulos, T. Kociumaka, and P. Wellnitz. Faster pattern matching under edit distance. *CoRR*, abs/2204.03087, 2022. `arXiv:2204.03087`, `doi:10.48550/arXiv.2204.03087`.

[45] H. Z. Q. Chen, S. Kitaev, T. Mütze, and B. Y. Sun. On universal partial words. *Discret. Math. Theor. Comput. Sci.*, 19(1), 2017. `doi:10.23638/DMTCS-19-1-16`.

[46] H. Cheon and Y. Han. Computing the shortest string and the edit-distance for parsing expression languages. In N. Jonoska and D. Savchuk, editors, *Developments in Language Theory - 24th International Conference, DLT 2020, Tampa, FL, USA, May 11-15, 2020, Proceedings*, volume 12086 of *Lecture Notes in Computer Science*, pages 43–54. Springer, 2020. `doi:10.1007/978-3-030-48516-0\_4`.

[47] H. Cheon, Y. Han, S. Ko, and K. Salomaa. The relative edit-distance between two input-driven languages. In P. Hofman and M. Skrzypczak, editors, *Developments in Language Theory - 23rd International Conference, DLT 2019, Warsaw, Poland, August 5-9, 2019, Proceedings*, volume 11647 of *Lecture Notes in Computer Science*, pages 127–139. Springer, 2019. `doi:10.1007/978-3-030-24886-4\_9`.

[48] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on strings*. Cambridge University Press, 2007. `doi:10.1017/CBO9780511546853`.

[49] M. Crochemore, B. Melichar, and Z. Troníček. Directed acyclic subsequence graph - overview. *J. Discrete Algorithms*, 1(3-4):255–280, 2003. `doi:10.1016/S1570-8667(03)00029-7`.

[50] J. D. Day, P. Fleischmann, M. Kosche, T. Koß, F. Manea, and S. Siemer. The edit distance to k-subsequence universality. In M. Bläser and B. Monmege, editors, *38th International Symposium on Theoretical Aspects of Computer Science, STACS 2021, March 16-19, 2021, Saarbrücken, Germany (Virtual Conference)*, volume 187 of *LIPIcs*, pages 25:1–25:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPICS.STACS.2021.25`.

[51] J. D. Day, P. Fleischmann, F. Manea, and D. Nowotka. Local patterns. In *Proc. 37th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2017*, volume 93 of *LIPIcs*, pages 24:1–24:14, 2017. `doi:10.4230/LIPIcs.FSTTCS.2017.24`.

[52] N. G. de Bruijn. A combinatorial problem. *Koninklijke Nederlandse Akademie v. Wetenschappen*, 49:758–764, 1946. URL: `https://api.semanticscholar.org/CorpusID:63276705`.

[53] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings.*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[54] D. P. Dobkin and R. J. Lipton. On the complexity of computations under varying sets of primitives. *J. Comput. Syst. Sci.*, 18(1):86–91, 1979. `doi:10.1016/0022-0000(79)90054-0`.

[55] A. Draghici, C. Haase, and F. Manea. Semënov arithmetic, affine vass, and string constraints. *CoRR*, abs/2306.14593, 2023. `arXiv:2306.14593`, `doi:10.48550/ARXIV.2306.14593`.

[56] R. Fagin, B. Kimelfeld, F. Reiss, and S. Vansummeren. Document spanners: A formal approach to information extraction. *J. ACM*, 62(2):12:1–12:51, 2015. `doi:10.1145/2699442`.

[57] M. R. Fellows, J. Gramm, and R. Niedermeier. On the parameterized intractability of motif search problems. *Comb.*, 26(2):141–167, 2006. `doi:10.1007/s00493-006-0011-4`.

[58] H. Fernau, F. Manea, R. Mercas, and M. L. Schmid. Revisiting Shinohara's algorithm for computing descriptive patterns. *Theor. Comput. Sci.*, 733:44–54, 2018. `doi:10.1016/j.tcs.2018.04.035`.

[59] H. Fernau, F. Manea, R. Mercas, and M. L. Schmid. Pattern matching with variables: Efficient algorithms and complexity results. *ACM Trans. Comput. Theory*, 12(1):6:1–6:37, 2020. `doi:10.1145/3369935`.

[60] H. Fernau and M. L. Schmid. Pattern matching with variables: A multivariate complexity analysis. *Inf. Comput.*, 242:287–305, 2015. `doi:10.1016/j.ic.2015.03.006`.

[61] H. Fernau, M. L. Schmid, and Y. Villanger. On the parameterised complexity of string morphism problems. *Theory Comput. Syst.*, 59(1):24–51, 2016. `doi:10.1007/s00224-015-9635-3`.

[62] L. Fleischer and M. Kufleitner. Testing Simon's congruence. In I. Potapov, P. G. Spirakis, and J. Worrell, editors, *43rd International Symposium on Mathematical Foundations of Computer Science, MFCS 2018, August 27-31, 2018, Liverpool, UK*, volume 117 of *LIPIcs*, pages 62:1–62:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. `doi:10.4230/LIPICS.MFCS.2018.62`.

[63] P. Fleischmann, S. B. Germann, and D. Nowotka. Scattered factor universality - the power of the remainder. *CoRR*, abs/2104.09063, 2021. URL: `https://arxiv.org/abs/2104.09063`, `arXiv:2104.09063`.

[64] P. Fleischmann, J. Höfer, A. Huch, and D. Nowotka. α-β-factorization and the binary case of Simon's congruence, 2023. `arXiv:2306.14192`.

[65] P. Fleischmann, S. Kim, T. Koß, F. Manea, D. Nowotka, S. Siemer, and M. Wiedenhöft. Matching Patterns with Variables Under Simon's Congruence. In O. Bournez, E. Formenti, and I. Potapov, editors, *Reachability Problems - 17th International Conference, RP 2023, Nice, France, October 11-13, 2023, Proceedings*, volume 14235 of *Lecture Notes in Computer Science*, pages 155–170. Springer, 2023. `doi:10.1007/978-3-031-45286-4\_12`.

[66] D. D. Freydenberger. Extended regular expressions: Succinctness and decidability. *Theory of Comput. Syst.*, 53:159–193, 2013. doi:10.1007/s00224-012-9389-0.

[67] D. D. Freydenberger. A logic for document spanners. *Theory Comput. Syst.*, 63(7):1679–1754, 2019. doi:10.1007/s00224-018-9874-1.

[68] D. D. Freydenberger, P. Gawrychowski, J. Karhumäki, F. Manea, and W. Rytter. Testing k-binomial equivalence. *CoRR*, abs/1509.00622, 2015. URL: http://arxiv.org/abs/1509.00622, arXiv:1509.00622.

[69] D. D. Freydenberger and M. Holldack. Document spanners: From expressive power to decision problems. *Theory Comput. Syst.*, 62(4):854–898, 2018. doi:10.1007/s00224-017-9770-0.

[70] D. D. Freydenberger and M. L. Schmid. Deterministic regular expressions with back-references. *J. Comput. Syst. Sci.*, 105:1–39, 2019. doi:10.1016/j.jcss.2019.04.001.

[71] J. E. F. Friedl. *Mastering regular expressions - understand your data and be more productive: for Perl, PHP, Java, .NET, Ruby, and more (3. ed.)*. O'Reilly, 2006. URL: http://www.oreilly.de/catalog/regex3/index.html.

[72] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *J. Comput. Syst. Sci.*, 30(2):209–221, 1985. doi:10.1016/0022-0000(85)90014-5.

[73] E. Garel. Minimal separators of two words. In A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, editors, *Combinatorial Pattern Matching, 4th Annual Symposium, CPM 93, Padova, Italy, June 2-4, 1993, Proceedings*, volume 684 of *Lecture Notes in Computer Science*, pages 35–53. Springer, 1993. doi:10.1007/BFB0029795.

[74] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[75] P. Gawrychowski, T. Kociumaka, W. Rytter, and T. Walen. Faster longest common extension queries in strings over general alphabets. In R. Grossi and M. Lewenstein, editors, *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016, June 27-29, 2016, Tel Aviv, Israel*, volume 54 of *LIPIcs*, pages 5:1–5:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPICS.CPM.2016.5.

[76] P. Gawrychowski, M. Kosche, T. Koß, F. Manea, and S. Siemer. Efficiently Testing Simon's Congruence. In M. Bläser and B. Monmege, editors, *38th International Symposium on Theoretical Aspects of Computer Science, STACS 2021, March 16-19, 2021, Saarbrücken, Germany (Virtual Conference)*, volume 187 of *LIPIcs*, pages 34:1–34:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.STACS.2021.34.

[77] P. Gawrychowski, M. Lange, N. Rampersad, J. O. Shallit, and M. Szykula. Existential length universality. In C. Paul and M. Bläser, editors, *37th International Symposium on Theoretical Aspects of Computer Science, STACS 2020, March 10-13, 2020, Montpellier, France*, volume 154 of *LIPIcs*, pages 16:1–16:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICS.STACS.2020.16.

[78] P. Gawrychowski, F. Manea, and S. Siemer. Matching Patterns with Variables Under Hamming Distance. In *46th International Symposium on Mathematical Foundations of Computer Science, MFCS 2021*, volume 202 of *LIPIcs*, pages 48:1–48:24, 2021. doi:10.4230/LIPIcs.MFCS.2021.48.

[79] P. Gawrychowski, F. Manea, and S. Siemer. Matching Patterns with Variables Under Edit Distance. In D. Arroyuelo and B. Poblete, editors, *String Processing and Information Retrieval - 29th International Symposium, SPIRE 2022, Concepción, Chile, November 8-10, 2022, Proceedings*, volume 13617 of *Lecture Notes in Computer Science*, pages 275–289. Springer, 2022. doi:10.1007/978-3-031-20643-6_20.

[80] P. Gawrychowski and P. Uznanski. Optimal trade-offs for pattern matching with k mismatches. *CoRR*, abs/1704.01311, 2017. arXiv:1704.01311.

[81] P. Gawrychowski and P. Uznanski. Towards unified approximate pattern matching for hamming and l_1 distance. In *Proc. 45th International Colloquium on Automata, Languages, and Programming, ICALP 2018*, volume 107 of *LIPIcs*, pages 62:1–62:13, 2018. doi:10.4230/LIPIcs.ICALP.2018.62.

[82] B. Goeckner, C. Groothuis, C. Hettle, B. Kell, P. Kirkpatrick, R. Kirsch, and R. W. Solava. Universal partial words over non-binary alphabets. *Theor. Comput. Sci.*, 713:56–65, 2018. doi:10.1016/J.TCS.2017.12.022.

[83] M. Hague. Strings at MOSCA. *ACM SIGLOG News*, 6(4):4–22, 2019. doi:10.1145/3373394.3373396.

[84] S. Halfon, P. Schnoebelen, and G. Zetzsche. Decidability, complexity, and expressiveness of first-order logic over the subword ordering. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*, pages 1–12. IEEE Computer Society, 2017. doi:10.1109/LICS.2017.8005141.

[85] R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160, 1950. doi:10.1002/j.1538-7305.1950.tb00463.x.

[86] Y. Han, S. Ko, and K. Salomaa. The edit-distance between a regular language and a context-free language. *Int. J. Found. Comput. Sci.*, 24(7):1067–1082, 2013. doi:10.1142/S0129054113400315.

[87] J. Hébrard. An algorithm for distinguishing efficiently bit-strings by their subsequences. *Theor. Comput. Sci.*, 82(1):35–49, 1991. `doi:10.1016/0304-3975(91)90170-7`.

[88] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343, 1975. `doi:10.1145/360825.360861`.

[89] G. Hoppenworth, J. W. Bentley, D. Gibney, and S. V. Thankachan. The Fine-Grained Complexity of Median and Center String Problems Under Edit Distance. In *28th Annual European Symposium on Algorithms (ESA 2020)*, volume 173 of *LIPIcs*, pages 61:1–61:19, 2020. `doi:10.4230/LIPIcs.ESA.2020.61`.

[90] H. Imai and T. Asano. Dynamic segment intersection search with applications. In *25th Annual Symposium on Foundations of Computer Science, West Palm Beach, Florida, USA, 24-26 October 1984*, pages 393–402. IEEE Computer Society, 1984. `doi:10.1109/SFCS.1984.715940`.

[91] J. F. JáJá, C. W. Mortensen, and Q. Shi. Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In R. Fleischer and G. Trippen, editors, *Algorithms and Computation, 15th International Symposium, ISAAC 2004, Hong Kong, China, December 20-22, 2004, Proceedings*, volume 3341 of *Lecture Notes in Computer Science*, pages 558–568. Springer, 2004. `doi:10.1007/978-3-540-30551-4\_49`.

[92] R. Jayaram and B. Saha. Approximating language edit distance beyond fast matrix multiplication: Ultralinear grammars are where parsing becomes hard! In I. Chatzigiannakis, P. Indyk, F. Kuhn, and A. Muscholl, editors, *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, volume 80 of *LIPIcs*, pages 19:1–19:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. `doi:10.4230/LIPICS.ICALP.2017.19`.

[93] H. W. L. Jr. Integer programming with a fixed number of variables. *Math. Oper. Res.*, 8(4):538–548, 1983. `doi:10.1287/MOOR.8.4.538`.

[94] P. Karandikar, M. Kufleitner, and P. Schnoebelen. On the index of Simon's congruence for piecewise testability. *Inf. Process. Lett.*, 115(4):515–519, 2015. `doi:10.1016/J.IPL.2014.11.008`.

[95] P. Karandikar and P. Schnoebelen. The height of piecewise-testable languages with applications in logical complexity. In J. Talbot and L. Regnier, editors, *25th EACSL Annual Conference on Computer Science Logic, CSL 2016, August 29 - September 1, 2016, Marseille, France*, volume 62 of *LIPIcs*, pages 37:1–37:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. `doi:10.4230/LIPICS.CSL.2016.37`.

[96] P. Karandikar and P. Schnoebelen. The height of piecewise-testable languages and the complexity of the logic of subwords. *Log. Methods Comput. Sci.*, 15(2), 2019. doi:10.23638/LMCS-15(2:6)2019.

[97] P. Karandikar and P. Schnoebelen. The height of piecewise-testable languages and the complexity of the logic of subwords. *Log. Methods Comput. Sci.*, 15(2), 2019. doi:10.23638/LMCS-15(2:6)2019.

[98] J. Karhumäki, A. Saarela, and L. Q. Zamboni. On a generalization of abelian equivalence and complexity of infinite words. *J. Comb. Theory, Ser. A*, 120(8):2189–2206, 2013. doi:10.1016/J.JCTA.2013.08.008.

[99] J. Karhumäki, A. Saarela, and L. Q. Zamboni. Variations of the morse-hedlund theorem for *k*-abelian equivalence. *Acta Cybern.*, 23(1):175–189, 2017. doi:10.14232/ACTACYB.23.1.2017.11.

[100] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proc. 30th International Colloquium Automata, Languages and Programming, ICALP 2003*, volume 2719 of *LNCS*, pages 943–955, 2003. doi:10.1007/3-540-45061-0_73.

[101] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006. doi:10.1145/1217856.1217858.

[102] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Proceedings of a symposium on the Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972. doi:10.1007/978-1-4684-2001-2_9.

[103] S. Kim, Y. Han, S. Ko, and K. Salomaa. On Simon's congruence closure of a string. *Theor. Comput. Sci.*, 972:114078, 2023. doi:10.1016/J.TCS.2023.114078.

[104] S. Kim, S. Ko, and Y. Han. Simon's Congruence Pattern Matching. In S. W. Bae and H. Park, editors, *33rd International Symposium on Algorithms and Computation, ISAAC 2022, December 19-21, 2022, Seoul, Korea*, volume 248 of *LIPIcs*, pages 60:1–60:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.ISAAC.2022.60.

[105] S. Kleest-Meißner, R. Sattler, M. L. Schmid, N. Schweikardt, and M. Weidlich. Discovering event queries from traces: Laying foundations for subsequence-queries with wildcards and gap-size constraints. In *25th International Conference on Database Theory, ICDT 2022*, volume 220 of *LIPIcs*, pages 18:1–18:21, 2022. doi:10.4230/LIPIcs.ICDT.2022.18.

[106] M. Kosche, T. Koß, F. Manea, and S. Siemer. Absent subsequences in words. *Fundam. Informaticae*, 189(3-4):199–240, 2022. doi:10.3233/FI-222159.

[107] M. Kosche, T. Koß, F. Manea, and S. Siemer. Combinatorial algorithms for subsequence matching: A survey. In H. Bordihn, G. Horváth, and G. Vaszil, editors, *Proceedings 12th International Workshop on Non-Classical Models of Automata and Applications, NCMA 2022, Debrecen, Hungary, August 26-27, 2022*, volume 367 of *EPTCS*, pages 11–27, 2022. `doi:10.4204/EPTCS.367.2`.

[108] D. Kosolobov. Computing runs on a general alphabet. *Inf. Process. Lett.*, 116(3):241–244, 2016. `doi:10.1016/J.IPL.2015.11.016`.

[109] D. Kosolobov. Finding the leftmost critical factorization on unordered alphabet. *Theor. Comput. Sci.*, 636:56–65, 2016. `doi:10.1016/J.TCS.2016.04.037`.

[110] M. Krötzsch, T. Masopust, and M. Thomazo. Complexity of universality and related problems for partially ordered nfas. *Inf. Comput.*, 255:177–192, 2017. `doi:10.1016/J.IC.2017.06.004`.

[111] M. Kulczynski, K. Lotz, D. Nowotka, and D. B. Poulsen. Solving string theories involving regular membership predicates using SAT. In O. Legunsen and G. Rosu, editors, *Model Checking Software - 28th International Symposium, SPIN 2022, Virtual Event, May 21, 2022, Proceedings*, volume 13255 of *Lecture Notes in Computer Science*, pages 134–151. Springer, 2022. `doi:10.1007/978-3-031-15077-7\_8`.

[112] D. Kuske. The subtrace order and counting first-order logic. In H. Fernau, editor, *Computer Science - Theory and Applications - 15th International Computer Science Symposium in Russia, CSR 2020, Yekaterinburg, Russia, June 29 - July 3, 2020, Proceedings*, volume 12159 of *Lecture Notes in Computer Science*, pages 289–302. Springer, 2020. `doi:10.1007/978-3-030-50026-9\_21`.

[113] D. Kuske and G. Zetzsche. Languages ordered by the subword order. In M. Bojanczyk and A. Simpson, editors, *Foundations of Software Science and Computation Structures - 22nd International Conference, FOSSACS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, volume 11425 of *Lecture Notes in Computer Science*, pages 348–364. Springer, 2019. `doi:10.1007/978-3-030-17127-8\_20`.

[114] G. M. Landau and U. Vishkin. Efficient string matching in the presence of errors. In *Proc. 26th Annual Symposium on Foundations of Computer Science, FOCS 1985*, pages 126–136, 1985. `doi:10.1109/SFCS.1985.22`.

[115] G. M. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10(2):157–169, 1989. URL: `https://www.sciencedirect.com/science/article/pii/0196677489900102`.

[116] K. G. Larsen and F. van Walderveen. Near-optimal range reporting structures for categorical data. In S. Khanna, editor, *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 265–276. SIAM, 2013. `doi:10.1137/1.9781611973105.20`.

[117] M. Lejeune, J. Leroy, and M. Rigo. Computing the k-binomial complexity of the thue-morse word. *J. Comb. Theory, Ser. A*, 176:105284, 2020. `doi:10.1016/J.JCTA.2020.105284`.

[118] M. Lejeune, M. Rigo, and M. Rosenfeld. The binomial equivalence classes of finite words. *Int. J. Algebra Comput.*, 30(07):1375–1397, 2020. `doi:10.1142/S0218196720500459`.

[119] J. Leroy, M. Rigo, and M. Stipulanti. Generalized pascal triangle for binomial coefficients of words. *Adv. Appl. Math.*, 80:24–47, 2016. `doi:10.1016/J.AAM.2016.04.006`.

[120] V. Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission*, 1:8–17, 1965.

[121] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics. Doklady*, 10:707–710, 1965. URL: `https://api.semanticscholar.org/CorpusID:60827152`.

[122] M. Li, B. Ma, and L. Wang. Finding similar regions in many sequences. *J. Comput. Syst. Sci.*, 65(1):73–96, 2002. `doi:10.1006/jcss.2002.1823`.

[123] M. Lothaire. *Combinatorics on Words*. Cambridge University Press, 1997. `doi:10.1017/CBO9780511566097`.

[124] M. Lothaire. *Algebraic Combinatorics on Words*. Cambridge University Press, 2002. `doi:10.1017/CBO9781107326019`.

[125] K. Lotz, A. Goel, B. Dutertre, B. Kiesl-Reiter, S. Kong, R. Majumdar, and D. Nowotka. Solving string constraints using SAT. In C. Enea and A. Lal, editors, *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II*, volume 13965 of *Lecture Notes in Computer Science*, pages 187–208. Springer, 2023. `doi:10.1007/978-3-031-37703-7\_9`.

[126] D. Maier. The complexity of some problems on subsequences and supersequences. *J. ACM*, 25(2):322–336, Apr. 1978.

[127] F. Manea and M. L. Schmid. Matching patterns with variables. In *Proc. 12th International Conference Combinatorics on Words, WORDS 2019*, volume 11682 of *LNCS*, pages 1–27, 2019. `doi:10.1007/978-3-030-28796-2_1`.

[128] B. M. H. Martin and M. H. Martin. A problem in arrangements. *Bulletin of the American Mathematical Society*, 40:859–864, 1934. URL: `https://api.semanticscholar.org/CorpusID:123567918`.

[129] D. Marx. Closest substring problems with small distances. *SIAM J. Comput.*, 38(4):1382–1410, 2008. `doi:10.1137/060673898`.

[130] A. Mateescu, A. Salomaa, and S. Yu. Subword histories and parikh matrices. *J. Comput. Syst. Sci.*, 68(1):1–21, 2004. `doi:10.1016/J.JCSS.2003.04.001`.

[131] T. Mieno, S. P. Pissis, L. Stougie, and M. Sweering. String sanitization under edit distance: Improved and generalized. In P. Gawrychowski and T. Starikovskaya, editors, *32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021, July 5-7, 2021, Wrocław, Poland*, volume 191 of *LIPIcs*, pages 19:1–19:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPICS.CPM.2021.19`.

[132] E. W. Myers and W. Miller. Approximate matching of regular expressions. *Bull. Math. Biol.*, 51(1):5–37, 1989. `doi:10.1007/BF02458834`.

[133] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001. `doi:10.1145/375360.375365`.

[134] Y. Nekrich. Efficient range searching for categorical and plain data. *ACM Trans. Database Syst.*, 39(1):9:1–9:21, 2014. `doi:10.1145/2543924`.

[135] F. Nicolas and E. Rivals. Hardness results for the center and median string problems under the weighted and unweighted edit distances. *J. Discrete Algorithms*, 3(2-4):390–415, 2005. `doi:10.1016/j.jda.2004.08.015`.

[136] S. Ordyniak and A. Popa. A parameterized study of maximum generalized pattern matching problems. *Algorithmica*, 75(1):1–26, 2016. `doi:10.1007/S00453-015-0008-8`.

[137] OSTRICH. Ostrich - an smt solver for string constraints. URL: `https://github.com/uuverifiers/ostrich/`.

[138] M. Pătraşcu. Lower bounds for 2-dimensional range counting. In D. S. Johnson and U. Feige, editors, *Proceedings of the 39th Annual ACM Symposium on Theory of Computing, San Diego, California, USA, June 11-13, 2007*, pages 40–46. ACM, 2007. `doi:10.1145/1250790.1250797`.

[139] G. Pighizzini. How hard is computing the edit distance? *Inf. Comput.*, 165(1):1–13, 2001. `doi:10.1006/INCO.2000.2914`.

[140] J. Pin. The consequences of imre simon's work in the theory of automata, languages, and semigroups. In M. Farach-Colton, editor, *LATIN 2004: Theoretical Informatics, 6th Latin American Symposium, Buenos Aires, Argentina, April 5-8, 2004, Proceedings*, volume 2976 of *Lecture Notes in Computer Science*, page 5. Springer, 2004. `doi:10.1007/978-3-540-24698-5\_4`.

[141] J. Pin. The influence of Imre Simon's work in the theory of automata, languages and semigroups. *Semigroup Forum*, 98:1–8, 2019. `doi:10.1007/s00233-019-09999-8`.

[142] N. Rampersad, J. Shallit, and Z. Xu. The computational complexity of universality problems for prefixes, suffixes, factors, and subwords of regular languages. *Fundam. Inf.*, 116(1-4):223–236, Jan. 2012.

[143] D. Reidenbach and M. L. Schmid. Patterns with bounded treewidth. *Inf. Comput.*, 239:87–99, 2014. `doi:10.1016/j.ic.2014.08.010`.

[144] M. Rigo and P. Salimov. Another generalization of abelian equivalence: Binomial complexity of infinite words. *Theor. Comput. Sci.*, 601:47–57, 2015. `doi:10.1016/J.TCS.2015.07.025`.

[145] H. Sakoe and S. Chiba. Dynamic programming algorithm optimization for spoken word recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 26(1):43–49, 1978. `doi:10.1109/TASSP.1978.1163055`.

[146] A. Salomaa. Connections between subwords and certain matrix mappings. *Theor. Comput. Sci.*, 340(1):188–203, 2005. `doi:10.1016/J.TCS.2005.03.024`.

[147] D. Sankoff. Minimal mutation trees of sequences. *SIAM Journal on Applied Mathematics*, 28(1):35–42, 1975. `doi:10.1137/0128004`.

[148] B. Schieber. Computing a minimum-weight k-link path in graphs with the concave monge property. In K. L. Clarkson, editor, *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1995. San Francisco, California, USA*, pages 405–411. ACM/SIAM, 1995. URL: `http://dl.acm.org/citation.cfm?id=313651.313774`.

[149] M. L. Schmid. A note on the complexity of matching patterns with variables. *Inf. Process. Lett.*, 113(19):729–733, 2013. `doi:10.1016/j.ipl.2013.06.011`.

[150] M. L. Schmid and N. Schweikardt. A purely regular approach to non-regular core spanners. In *Proc. 24th International Conference on Database Theory, ICDT 2021*, volume 186 of *LIPIcs*, pages 4:1–4:19, 2021. `doi:10.4230/LIPIcs.ICDT.2021.4`.

[151] M. L. Schmid and N. Schweikardt. Document spanners - A brief overview of concepts, results, and recent developments. In *PODS '22: International Conference on Management of Data*, pages 139–150. ACM, 2022. doi:10.1145/3517804.3526069.

[152] P. Schnoebelen and J. Veron. On arch factorization and subword universality for words and compressed words. In A. E. Frid and R. Mercas, editors, *Combinatorics on Words - 14th International Conference, WORDS 2023, Umeå, Sweden, June 12-16, 2023, Proceedings*, volume 13899 of *Lecture Notes in Computer Science*, pages 274–287. Springer, 2023. doi:10.1007/978-3-031-33180-0_21.

[153] S. Seki. Absoluteness of subword inequality is undecidable. *Theor. Comput. Sci.*, 418:116–120, 2012. doi:10.1016/J.TCS.2011.10.017.

[154] T. Shinohara. Polynomial time inference of extended regular pattern languages. In E. Goto, K. Furukawa, R. Nakajima, I. Nakata, and A. Yonezawa, editors, *RIMS Symposium on Software Science and Engineering, Kyoto, Japan, 1982, Proceedings*, volume 147 of *Lecture Notes in Computer Science*, pages 115–127. Springer, 1982. doi:10.1007/3-540-11980-9_19.

[155] T. Shinohara and S. Arikawa. Pattern inference. In K. P. Jantke and S. Lange, editors, *Algorithmic Learning for Knowledge-Based Systems, GOSLER Final Report*, volume 961 of *Lecture Notes in Computer Science*, pages 259–291. Springer, 1995. doi:10.1007/3-540-60217-8_13.

[156] I. Simon. *Hierarchies of events with dot-depth one - Ph.D. thesis.* University of Waterloo, 1972.

[157] I. Simon. Piecewise testable events. In H. Barkhage, editor, *Automata Theory and Formal Languages, 2nd GI Conference, Kaiserslautern, May 20-23, 1975*, volume 33 of *Lecture Notes in Computer Science*, pages 214–222. Springer, 1975. doi:10.1007/3-540-07407-4_23.

[158] I. Simon. Words distinguished by their subwords (extended abstract). In *Proc. WORDS 2003*, volume 27 of *TUCS General Publication*, pages 6–13, 2003.

[159] S. S. Sturrock. Time warps, string edits, and macromolecules – the theory and practice of sequence comparison. david sankoff and joseph kruskal. isbn 1-57586-217-4. *Genetics Research*, 76(3):327–329, 2000. doi:10.1017/S0016672300219320.

[160] Z. Tronícek. Common subsequence automaton. In J. Champarnaud and D. Maurel, editors, *Implementation and Application of Automata, 7th International Conference, CIAA 2002, Tours, France, July 3-5, 2002, Revised Papers*, volume 2608 of *Lecture Notes in Computer Science*, pages 270–275. Springer, 2002. doi:10.1007/3-540-44977-9_28.

[161] P. Uznanski. Recent advances in text-to-pattern distance algorithms. In *Proc. 16th Conference on Computability in Europe, CiE 2020*, volume 12098 of *LNCS*, pages 353–365, 2020. doi:10.1007/978-3-030-51466-2_32.

[162] R. A. Wagner. Order-n correction for regular languages. *Commun. ACM*, 17(5):265–268, 1974. doi:10.1145/360980.360995.

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

[164] K. Wasa. Enumeration of enumeration algorithms, 2016. arXiv:1605.05102.

[165] P. Weis and N. Immerman. Structure theorem and strict alternation hierarchy for fo^2 on words. *Log. Methods Comput. Sci.*, 5(3), 2009. URL: http://arxiv.org/abs/0907.0616.

[166] R. Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theor. Comput. Sci.*, 348(2-3):357–365, 2005. doi:10.1016/j.tcs.2005.09.023.

[167] G. Zetzsche. The complexity of downward closure comparisons. In I. Chatzigiannakis, M. Mitzenmacher, Y. Rabani, and D. Sangiorgi, editors, *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*, volume 55 of *LIPIcs*, pages 123:1–123:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPICS.ICALP.2016.123.