Pattern Matching with Variables: Efficient Algorithms and Complexity Results*

HENNING FERNAU, Fachbereich IV – Abteilung Informatikwissenschaften, Universität Trier
FLORIN MANEA, Göttingen University, Institute of Computer Science
ROBERT MERCAŞ, Loughborough University, Department of Computer Science
MARKUS L. SCHMID, Fachbereich IV – Abteilung Informatikwissenschaften, Universität Trier

A pattern α (i. e., a string of variables and terminals) matches a word w, if w can be obtained by uniformly replacing the variables of α by terminal words. The respective matching problem, i. e., deciding whether or not a given pattern matches a given word, is generally NP-complete, but can be solved in polynomial-time for restricted classes of patterns. We present efficient algorithms for the matching problem with respect to patterns with a bounded number of repeated variables and patterns with a structural restriction on the order of variables. Furthermore, we show that it is NP-complete to decide, for a given number k and a word w, whether w can be factorised into k distinct factors. As an immediate consequence of this hardness result, the injective version (i. e., different variables are replaced by different words) of the matching problem is NP-complete even for very restricted clases of patterns.

CCS Concepts: • **Theory of computation** → **Problems, reductions and completeness**; Parameterized complexity and exact algorithms; • **Mathematics of computing** → *Discrete mathematics*;

Additional Key Words and Phrases: Combinatorial pattern matching, combinatorics on words, patterns with variables, NP-complete string problems

ACM Reference format:
Henning Fernau, Florin Manea, Robert Mercaş, and Markus L. Schmid. 2019. Pattern Matching with Variables: Efficient Algorithms and Complexity Results. *ACM Trans. Comput. Theory* 1, 1, Article 1 (January 2019), 38 pages.
DOI: 10.1145/3369935

1 INTRODUCTION

In the context of this work, a *pattern* is a string that consists of *terminal symbols* (e. g., a, b, c) and *variables* (e. g., x_1, x_2, x_3). The terminal symbols are treated as constants, while the variables are to be uniformly replaced by strings over the set of terminals (i. e., different occurrences of the same variable are replaced by the same string); thus, a pattern is mapped to a terminal word. For example, $x_1 \mathtt{ab} x_1 x_2 \mathtt{c} x_2 x_1$ can be mapped to acabaccaaccaaac and babbacab by the replacements $(x_1 \to \mathtt{ac}, x_2 \to \mathtt{caa})$ and $(x_1 \to \mathtt{b}, x_2 \to \mathtt{a})$, respectively.

Due to their simple definition, the concept of patterns (and how they map to words) emerges in various areas of theoretical computer science, such as language theory (pattern languages [2]),

*This work represents an extended version of the paper "Pattern Matching with Variables: Fast Algorithms and New Hardness Results" presented at STACS 2015 and published in LIPICS with http://dx.doi.org/10.4230/LIPIcs.STACS.2015.302

learning theory (inductive inference [2, 13, 32, 34], PAC-learning [25]), combinatorics on words (word equations [23, 31], unavoidable patterns [30]), pattern matching (generalised function matching [1, 33]), database theory (extended conjunctive regular path queries [5]), and we can also find them in practice in the form of extended regular expressions with backreferences [6, 18, 19], used in programming languages like Perl, Java, Python, etc.

In all these different applications, the main purpose of patterns is to express combinatorial pattern matching questions. For instance, searching for a word w in a text t can be expressed as testing whether the pattern xwy can be mapped to t and testing whether a word w contains a k-repetition is equivalent to testing whether the pattern xy^kz can be mapped to w. Not only problems of testing whether a given word contains a regularity or a motif of a certain form can be expressed by patterns, but also problems asking whether a word can be factorised in a specifically restricted manner can be modelled in this way. For instance, asking whether x^2y^3 can be mapped to w is equivalent to asking whether the word w can be factorised in two equal factors followed by three equal factors.

It is also easy to see that such abstract combinatorial questions can have natural applications as well, e. g., in tasks of information retrieval where recurring factors need to be identified that are not a priori length-bounded (note that this goes beyond the capability of regular expressions).

Unfortunately, deciding whether a given pattern can be mapped to a given word, the *matching problem*, is NP-complete [2], which naturally severely limits the practical application of patterns. In fact, there are only few applications of patterns for which this problem does not play a central role and, furthermore, some computational tasks on patterns that have no apparent connection to the matching problem turn out to implicitly solve it anyway (e. g., this is the case for the task of computing so-called descriptive patterns for finite sets of words [2, 15]). A comprehensive multivariate analysis of the complexity of the matching problem [16, 17] demonstrates that the NP-completeness also holds for strongly restricted variants of the problem. On the other hand, some subclasses of patterns are known for which the matching problem is in P (this is obviously the case if the number of different variables in the patterns is bounded by a constant, but there are also more sophisticated structural parameters of patterns that can be exploited in order to solve the matching problem efficiently [35–37, 39]). Unfortunately, the existing polynomial time algorithms for these classes serve the mere purpose of proving containment in P; thus, they cannot be considered efficient in a practical sense. Therefore, for some of the known restrictions that yield polynomial-time solvability, we present better algorithms. While we consider our algorithms to be advanced and non-trivial, their running times have still an exponential dependency on certain parameters of patterns and, therefore, are acceptable only for strongly restricted classes of patterns. However, as can be concluded from the parameterised hardness results of [17], these exponential dependencies seem necessary under common complexity theoretical assumptions.

In some applications of patterns it might be necessary to require the mapping of variables to be injective (i. e., different variables are substituted by different objects), e. g., this is the case in the detection of duplications in program code (see [3]). From a more general point of view, this injective version of the matching problem asks whether a word can be factorised in a certain way, such that some specific factors are not allowed to coincide. The special version of this problem where each two factors must be different has been investigated in [9] and is motivated by the problem of self-assembly of short DNA fragments into larger sequences, which is crucial for gene synthesis (see references in [9]). We show the NP-completeness of the following natural combinatorial factorisation problem: given a number k and a word w, can w be factorised into at least k distinct factors? Besides the general insight into the hardness of computing a factorisation with distinct factors, this result also implies that even for the trivial patterns $x_1x_2 \cdots x_k$ the matching problem becomes NP-complete if we require injectivity. Thus, the injective variant of the matching problem can be considered much harder; in particular, all the structural restrictions of patterns that are known to yield polynomial-time solvability of the matching problem fail if injectivity is required.

This paper is the full version of the extended abstract [14] presented at STACS 2015 and it contains all the omitted proofs and technical details. The organisation is as follow. The next section, Section 2, contains basic definitions and then, in Section 3, we give an overview of our results. In Section 4, we develop our algorithms for the matching problem and, in Section 5, we present the hardness result mentioned above. Finally, we give some conclusions in Section 6.

2 BASIC DEFINITIONS

For detailed definitions regarding combinatorics on words we refer to [29].

We denote our *alphabet* by Σ, the *empty word* by ε, the set of all non-empty words over Σ by Σ^+, the set of all words over Σ by Σ^*, and the *length* of a word w by $|w|$. $(\Sigma^*, \cdot, \varepsilon)$ is the free monoid over Σ with *concatenation* as its binary operation, written $\cdot$. For $w \in \Sigma^*$ and every integers i, j with $1 \leq i \leq j \leq |w|$, let $w[i..j] = w[i] \cdots w[j]$, where $w[k]$ represents the *letter on position* k and $1 \leq k \leq |w|$. A *period* of w is any positive integer p for which $w[i] = w[i+p]$, for all defined positions. Moreover, in this case, w is said to be p-periodic. Its *minimal period* is denoted by $per(w)$ and represents the smallest period of w. For example, $w = \mathsf{abacabacabacabacab}$ has periods 8 and 4; in particular, $per(w) = 4$. A word w is called periodic if $per(w) \leq \frac{|w|}{2}$.

The *concatenation* of k words $w_1, w_2, \ldots, w_k$ is written $\Pi_{i=1,k} w_i$. If $w = w_i$ for all integers i with $1 \leq i \leq k$, this represents the kth *power* of w, denoted by w^k; here, w is a *root* of w^k. We can further extend the notion of a power of a word by saying that $w = w[1..per(w)]^{\frac{|w|}{per(w)}}$. We say that w is *primitive* if it cannot be expressed as a power of exponent ℓ of any root, where ℓ is an integer with $\ell > 1$. Conversely, if $w = v^\ell$ for some integer $\ell > 1$, then w is also called a *repetition*. The infinite repetition $vvv \cdots$ of some word v is denoted v^ω.

For any word $w \in \Sigma^+$ with $w = xyz$, we say that y is a *factor* of w. If x is empty, then y is also a *prefix* of w, while when z is empty, then y is also a *suffix*. Whenever we have a factor both as a prefix and as a suffix, the factor is said to be a *border* of the word. Furthermore, every word $u = yzx \in \Sigma^+$ is a *conjugate* of w. Note that, if w is primitive, so is every conjugate of it. If w has prefix v, i. e., $w = vu$, then we also use $v^{-1}w$ in order to denote the suffix u that remains after removing the prefix v from w.

For a word $w \in \Sigma^+$, a *factorisation* of w is a tuple $p = (u_1, u_2, \ldots, u_k) \in (\Sigma^+)^k$, for $k \geq 1$, with $w = u_1 u_2 \cdots u_k$. For every i, $1 \leq i \leq k$, $p(i) = u_i$ is called a *factor of* p, or simply a p*-factor*. The set of factors of p is defined as $\mathsf{sf}(p) = \{u_1, u_2, \ldots, u_k\}$ and its size as $\mathsf{s}(p) = k$. The factorisation p is *unique* if all its factors are distinct, i. e., $\mathsf{s}(p) = |\mathsf{sf}(p)|$. For the sake of readability, we sometimes represent a factorisation $(u_1, u_2, \ldots, u_k)$ in the form $u_1 \mid u_2 \mid \ldots \mid u_k$.

Example 2.1. Let $w = \mathsf{abacbaabaa}$ be a word over $\Sigma = \{\mathsf{a}, \mathsf{b}\}$. Then $p = (\mathsf{a}, \mathsf{ba}, \mathsf{cba}, \mathsf{a}, \mathsf{ba}, \mathsf{a})$ is a factorisation of w with its third factor $p(3) = \mathsf{cba}$. Furthermore, p is not unique, since $|\mathsf{sf}(p)| = |\{\mathsf{a}, \mathsf{ba}, \mathsf{cba}\}| \neq \mathsf{s}(p) = 6$, or, in other words, there exists a repeated factor, e. g., $p(2) = p(5) = \mathsf{ba}$. On the other hand, $q = (\mathsf{ab}, \mathsf{a}, \mathsf{c}, \mathsf{ba}, \mathsf{abaa})$ is a unique factorisation for w of size 5.

Let $X = \{x_1, x_2, x_3, \ldots\}$ and call every $x \in X$ a *variable*. For a finite alphabet Σ of *terminals* with $\Sigma \cap X = \emptyset$, we define $\textsc{Pat}_\Sigma = (X \cup \Sigma)^+$ and $\textsc{Pat} = \bigcup_\Sigma \textsc{Pat}_\Sigma$. Every $\alpha \in \textsc{Pat}$ is a *pattern* and every $w \in \Sigma^*$ is a (*terminal*) *word*. Given a word or a pattern v, for the smallest sets $B \subseteq \Sigma$ and $Y \subseteq X$ with $v \in (B \cup Y)^*$, we denote $\mathrm{alph}(v) = B$ (i. e., the set of terminal symbols) and $\mathrm{var}(v) = Y$ (i. e., the set of variables). For any $x \in \Sigma \cup X$ and $\alpha \in \textsc{Pat}_\Sigma$, $|\alpha|_x$ denotes the number of occurrences of x in α; for the sake of convenience, we set $|\alpha|_x = 0$ for every symbol x not in $\Sigma \cup X$. By $\mathrm{rvar}(\alpha)$, we denote the set of *repeated* variables, i. e., $\mathrm{rvar}(\alpha) = \{x \in \mathrm{var}(\alpha) \mid |\alpha|_x \geq 2\}$. For a pattern α, we say that $w = \alpha[i..i + |w|]$ is a maximal terminal factor of α if $\alpha[i - 1]$ and $\alpha[i + |w| + 1]$ are either not defined, or are variables.

A *substitution* (*for* α) is a mapping $h : \mathrm{var}(\alpha) \to \Sigma^+$. For every $x \in \mathrm{var}(\alpha)$, we say that x *is substituted by* $h(x)$ and $h(\alpha)$ denotes the word obtained by substituting every occurrence of a variable x in α by $h(x)$ and leaving the terminals unchanged. If, for all $x, y \in \mathrm{var}(\alpha)$, $x \neq y$ implies $h(x) \neq h(y)$, then h is *injective*. We say that the pattern α *matches* $w \in \Sigma^+$ if $h(\alpha) = w$ for some substitution $h : \mathrm{var}(\alpha) \to \Sigma^+$.

Next, we formally define the problems of matching patterns with variables. To this end, let $P \subseteq \textsc{Pat}$ be a *family of patterns*. The *matching problem* (for P) is defined as follows.

Match for P	
Instance:	Pattern $\alpha \in P$ and word w.
Question:	Is there a substitution h with $h(\alpha) = w$?

If $P = \textsc{Pat}$, i. e., the full class of all patterns, we just use the term *matching problem* (or just Match, for short).[1]

As an example, consider the pattern $\beta = x_1 \mathsf{a} x_2 \mathsf{b} x_2 x_1 x_2$ and the terminal words $u = \mathsf{bacbabbbbacbb}$ and $v = \mathsf{abaabbababab}$. Both (β, u) and (β, v) are positive instances of Match, as witnessed by the substitutions h with $h(x_1) = \mathsf{bacb}$, $h(x_2) = \mathsf{b}$ and g with $g(x_1) = g(x_2) = \mathsf{ab}$, respectively. On the other hand, it can be easily seen that β does not match the word $w = \mathsf{cbaabbbbab}$, since any possible substitution maps x_1 to a word starting with c, but there is only one occurrence of c in w.

The *injective* variant of the matching problem (for P) is defined as follows.

inj-Match for P	
Instance:	Pattern $\alpha \in P$ and word w.
Question:	Is there an *injective* substitution h with $h(\alpha) = w$?

Coming back to our example from above, we observe that (β, u) is also a positive instance of inj-Match, since h is an injective substitution. On the other hand, g is obviously not injective and it can be easily verified, that (β, v) is in fact a negative instance of inj-Match, i. e., there is no injective substitution that maps β to v.

Next, we introduce several interesting families of patterns. A pattern α is *regular* if, for every $x \in \mathrm{var}(\alpha)$, we have $|\alpha|_x = 1$, and the class of regular patterns is denoted by $\textsc{Pat}_{\mathrm{reg}}$. For any integer $k \geq 1$, a *k-variable* pattern is a pattern α that satisfies $|\mathrm{var}(\alpha)| \leq k$ and a *k-repeated-variable* pattern is a pattern α that satisfies $|\mathrm{rvar}(\alpha)| \leq k$ (recall that $\mathrm{rvar}(\alpha)$ is the set of repeated variables as defined above). For every integer $k \geq 0$, $\textsc{Pat}_{\mathrm{var}\leq k}$ and $\textsc{Pat}_{\mathrm{rvar}\leq k}$ denote the set of k-variable patterns and k-repeated-variable patterns, respectively. Obviously, $\textsc{Pat}_{\mathrm{reg}} = \textsc{Pat}_{\mathrm{rvar}\leq 0}$.

For every $y \in \mathrm{var}(\alpha)$, the *scope of y in α* is defined by $\mathrm{sc}_\alpha(y) = \{i, i + 1, \ldots, j\}$, where i is the leftmost and j the rightmost occurrence of y in α. The scopes of some variables $y_1, y_2, \ldots, y_k \in \mathrm{var}(\alpha)$ *coincide in* α if $\bigcap_{1\leq i\leq k} \mathrm{sc}_\alpha(y_i) \neq \emptyset$. By $\mathrm{scd}(\alpha)$, we denote the *scope coincidence degree* (scd for short) of α, which is the maximum number of variables in α such that their scopes coincide. For every $k \geq 1$, let $\textsc{Pat}_{\mathrm{scd}\leq k}$ denote the set of patterns α with $\mathrm{scd}(\alpha) \leq k$. By definition, $\textsc{Pat}_{\mathrm{scd}\leq 1}$ coincides with the class of *non-cross* patterns (see [39]), which we denote by $\textsc{Pat}_{\mathrm{nc}}$.

The *one-variable blocks* in a pattern are maximal contiguous blocks of occurrences of the same variable. A pattern α with m one-variable blocks can be written as $\alpha = w_0 \Pi_{i=1,m}(z_i^{k_i} w_i)$ with $z_i \in \mathrm{var}(\alpha)$ for $i \in \{1, 2, \ldots, m\}$ and $z_i \neq z_{i+1}$, whenever $w_i = \varepsilon$ for $i \in \{1, 2, \ldots, m - 1\}$. The number of one-variable blocks is a natural complexity measure that we will consider.

[1]There exist variants of the matching problem where substitutions can also *erase* variables by mapping them to ε. In this work, we are not concerned with this variant of the problem.

Example 2.2. By definition, $\mathtt{a}x_1\mathtt{b}x_2\mathtt{c}x_3\mathtt{bca}x_4$ is regular; $\mathtt{a}x_1\mathtt{b}x_1x_2\mathtt{a}x_2x_3x_3\mathtt{b}x_4$ is non-cross, whereas $x_1\mathtt{b}x_1x_2\mathtt{b}x_3x_4\mathtt{b}x_2$ is not. For $\alpha = x_1x_2x_1x_2x_3x_1x_2x_3$, the scopes are $\mathrm{sc}_\alpha(x_1) = \{1, 2, \dots, 6\}$, $\mathrm{sc}_\alpha(x_2) = \{2, 3, \dots, 7\}$ and $\mathrm{sc}_\alpha(x_3) = \{5, 6, \dots, 8\}$; thus, the scopes of all three variables coincide, which means $\mathrm{scd}(\alpha) = 3$. On the other hand, for $\beta = x_1x_2x_1x_2x_3x_2x_3x_3$, $\mathrm{sc}_\alpha(x_1) \cap \mathrm{sc}_\alpha(x_2) \neq \emptyset$, $\mathrm{sc}_\alpha(x_2) \cap \mathrm{sc}_\alpha(x_3) \neq \emptyset$, but $\mathrm{sc}_\alpha(x_1) \cap \mathrm{sc}_\alpha(x_3) = \emptyset$; thus, $\mathrm{scd}(\beta) = 2$. The pattern $\alpha = x_1x_2x_2\mathtt{a}x_2x_2x_2x_3\mathtt{a}x_3x_2x_2x_3x_3$ has 7 one-variable blocks: $x_1, x_2x_2, x_2x_2x_2, x_3, x_3, x_2x_2, x_3x_3$.

The computational model we use in this work is the standard unit-cost RAM with logarithmic word size. Also, all logarithms appearing in our time complexity evaluations are in base 2.

For the sake of generality, we assume that whenever we are given as input a word $w \in \Sigma^*$ of length n, the symbols of w are in fact integers from $\{1, 2, \dots, n\}$ (i.e., $\Sigma = \mathrm{alph}(w) \subseteq \{1, 2, \dots, n\}$), and w is seen as a sequence of integers. This is a common assumption in the area of algorithmics on words (see, e.g., the discussion in [24]). Clearly, our reasoning holds canonically for constant alphabets, as well.

For a length n word w we can build in $O(n)$ time the *suffix array* structure, as well as data structures allowing us to retrieve in constant time the length of the *longest common prefix* of any two suffixes $w[i..n]$ and $w[j..n]$ of w, denoted $LCP_w(i, j)$ (the subscript w is omitted when there is no danger of confusion). Such structures are called *LCP* data structures in the following. For details, see, e.g., [21, 24], and the references therein. Similarly, we can build structures allowing us to retrieve in constant time the length of the longest common suffix of any two prefixes $w[1..i]$ and $w[1..j]$ of w, denoted $LCS_w(i, j)$.

3 SUMMARY OF OUR RESULTS

The classical and parametrised complexity of the matching problem for patterns have been recently investigated and are well understood (see [7, 16, 17, 33]). The most prominent subclasses of patterns for which it can be solved in polynomial time are the classes of patterns with a bounded number of (repeated) variables (in this regard, note that the database example in Section 1 had only one repeated variable), of regular patterns, of non-cross patterns, and of patterns with a bounded scope coincidence degree (see [2, 36, 39]). However, as mentioned in the introduction, the respective algorithms are rather poor considering their running times. For example, for patterns with a bounded number k of variables, the matching problem can be solved in $O(\frac{mn^{k-1}}{(k-1)!})$, where m and n are the lengths of the pattern and the word (see [22]). For patterns with a scope coincidence degree of at most k, an $O(mn^{2(k+3)}(k + 2)^2)$ time algorithm is given in [36], where m and n are the lengths of the pattern and the word, respectively, and the proof that the matching problem for non-cross patterns is in P (see [39]) leads to an $O(n^4)$-time algorithm. Hence, we consider the following problem worth investigating.

Problem 1. *Let K be a class of patterns for which the matching problem can be solved in polynomial time. Find an efficient algorithm that solves the matching problem for K.*

The main class of patters we consider is that of patterns with bounded scope coincidence degree. Our first result in this setting concerns patterns where the scope coincidence degree is bounded by 1, or, in other words, non-cross patterns. In that case we show that we can decide whether a pattern α having m one-variable blocks matches a word w of length n in $O(mn \log n)$ time; this is an important improvement over the previously available $O(n^4)$ algorithm. Our algorithm is based on a general dynamic programming approach, and it tries to find, for certain prefixes of the pattern, the prefixes of the word that match them. While the general approach is rather simple, the details of the efficient implementation of this approach require a detailed combinatorial analysis of the possible matches. For instance, as a byproduct of our approach to the matching problem for $\mathrm{PAT}_{\mathrm{nc}}$, we obtain a stringology result that extends in a non-trivial manner a major result from [10], showing how the primitively rooted squares contained in a word of length n can be listed optimally in $O(n \log n)$. Our result shows that, given a word w of length n and a word v with length shorter than n, the word w contains $O(n \log n)$ factors of the form uvu with uv primitive, and all these factors can be found in $O(n \log n)$ time. Again, this result is optimal, as it can be seen just by looking at the original case of primitively rooted squares, or factors of the form uvu with uv primitive and $v = \varepsilon$.

When considering general patterns with bounded scope coincidence degree, we show, using a similar dynamic programming approach, that the matching problem for $\mathrm{PAT}_{\mathrm{scd} \leq k}$ is solvable in $O(\frac{m^2 n^{2k}}{k!(k-1)!})$ time, where n is the length of the input word and m is, again, the number of one-variable blocks occurring in the pattern. One should note that in this case we were not able to use all the combinatorial insights shown for non-cross patterns (thus, the $\log n$ factor is replaced by an n factor in the evaluation of the time complexity), but, still, our algorithm is significantly faster than the previously known solution.

Another class of patterns we consider is $\mathrm{PAT}_{\mathrm{rvar} \leq k}$ of patterns with at most k repeated variables. For the basic case of $k = 1$ we obtain that the matching problem can be solved in $O(n^2)$ time, where n is the length of the input word. Our algorithm is based on a non-trivial processing of the suffix array of the input word. Further, we use this result to show that the matching problem for the general class of patterns $\mathrm{PAT}_{\mathrm{rvar} \leq k}$ is solvable in $O(\frac{n^{2k}}{((k-1)!)^2})$ time. Note that our algorithm is better than the one that could have been obtained by using the fact that patterns with at most k repeated variables have the scope coincidence degree bounded by $k + 1$, and then directly applying our previous algorithm solving the matching problem for $\mathrm{PAT}_{\mathrm{scd} \leq k+1}$.

The classes of non-cross patterns and of patterns with a bounded scope coincidence degree or with a bounded number of repeated variables are of special interest, since for them we can compute so-called descriptive patterns (see [2, 39]) in polynomial time. A pattern α is *descriptive* (with respect to, say, non-cross patterns) for a finite set S of words if it can generate all words in S and there exists no other non-cross pattern that describes the elements of S in a better way. Computing a descriptive pattern, which is NP-complete in general, means to infer a pattern common to a finite set of words, with applications for inductive inference of pattern languages (see [32]). For example, our algorithm for computing non-cross patterns can be used in order to obtain an algorithm that computes a descriptive non-cross pattern in time $O(\sum_{w \in S}(m^2|w| \log |w|))$, where m is the length of a shortest word of S (see [15] for details).

Our algorithms, except the ones for the basic cases of non-cross patterns and patterns with only one repeated variable, still have an exponential dependency on the number of repeated variables or the scope coincidence degree. Therefore, only for very low constant bounds on these parameters can these algorithms be considered efficient. Naturally, finding a polynomial time algorithm for which the degree of the polynomial does not depend on the number of repeated variables would be desirable. However, such an algorithm would also be a fixed parameter algorithm for the matching problem parameterised by the number of repeated variables and in [17] it has been shown that this parameterised problem is $W[1]$-hard. This means that the existence of such an algorithm is very unlikely. Furthermore, since the number of repeated variables gives also an upper bound for the scope coincidence degree, the mentioned $W[1]$-hardness result carries over to the case where the scope coincidence degree is a parameter and therefore it is just as unlikely to find an algorithm that is not exponential in the scope coincidence degree. This observation justifies the exponential dependency of our algorithms on the number of repeated variables and the scope coincidence degree.

As mentioned in the introduction, in certain settings it makes sense to require the mapping of variables to words to be injective. The current state of knowledge regarding the complexity of the matching problem suggests that this difference has no substantial impact; although, in [16] it is shown that MATCH is still NP-complete if the alphabet size and the length of the words the variables are mapped to are bounded, whereas it is in P if we additionally require injectivity. In contrast to this, we prove the following result, which gives strong evidence that inj-MATCH is generally much harder than the non-injective version.

THEOREM 3.1. *The following problem is* NP*-complete: given a word w and a number k, is it possible to factorise w into at least k distinct factors?*

Consequently, the injective matching problem is NP-complete even for the trivial patterns $x_1 x_2 \cdots x_k$, which means that, under the assumption P $\neq$ NP, for *all* the above mentioned classes of patterns no polynomial time algorithms for the injective matching problem exist. In addition to this negative result for the matching problem, we also gain an important insight regarding the more general problem of factorising a string into distinct factors, which, as mentioned in the introduction, is motivated by computational biology. In [9], it is shown that it is NP-complete to factorise a string into distinct factors with a bound on the length of the factors and, in this regard, our result shows that the NP-completeness is preserved if the length bound is dropped and instead we have a lower bound on the number of factors.

In the following two main sections of this paper, we first present algorithmic results for the matching problem, while turning to mostly complexity-theoretic results on the injective variant in the section following thereafter.

4 EFFICIENT ALGORITHMS FOR THE MATCHING PROBLEM

In this section we propose efficient algorithms for the matching problem for several classes of patterns. On the one hand, we look in Subsection 4.1 at classes where the number of repeated variables is bounded: first we consider the basic class of regular patterns, where no variable is repeated, and then investigate the class of patterns in which exactly k variables are repeated. On the other hand, we look at classes with bounded scope coincidence degree. A basic class in this case is the one of non-cross patterns, where the upper bound of the scope coincidence degree is 1 (Subsection 4.2); we then move on and analyse the general case of patterns having the upper bound of the scope coincidence degree equal to k (Subsection 4.3).

As a general aspect, note that whenever we deal with a matching problem, if the size of the pattern α is greater than the length of the word w, then the problem is solvable in $O(|w|)$ in the negative (we can at the beginning verify the lengths of each sequence and stop after $O(|w|)$ steps). Therefore, for the remainder of the paper we will always assume that $|\alpha| < |w|$. Also, when talking about the running times of algorithms, we can hence measure the input length by $|w|$. Moreover, since for every instance of the matching problem we are presented with a word and a pattern as input, one can in a preprocessing step construct in linear time (relative to the length of the word) the suffix array, *LCP*, and *LCS* data structures associated to the word, the pattern, or the concatenation of these two. This will ensure, e.g., the fact that testing the equality of factors of the pattern and the word can be done in constant time, testing whether a factor has a certain period, etc., within our main algorithms.

4.1 Patterns with a bounded number of repeated variables

For the class PAT_{reg} of regular patterns (note that this class coincides with the class of patterns with no repeated variable) we can show the following result.

THEOREM 4.1. *The matching problem for PAT_{reg} is solvable in linear time.*

Algorithm 1 Solving the matching problem for PAT_{reg} in linear time

```
Input: regular pattern $\alpha = w_0\Pi_{i=1,m}(x_i w_i)$ with $m \geq 1$, $x_i \in \text{var}(\alpha)$ and $w_i \in \Sigma^*$, and a word
    $w \in \Sigma^*$ with $n = |w|$
Output: yes or no according to whether $\alpha$ matches $w$
1: if $w_0$ is not a prefix of $w$ then return no
2: $s := 1, j := |w_0|$
3: while $s < m$ do
4:     if $w_s$ does not occur in $w[j+2..n]$ then return no
5:     let $j'$ be the start position of the leftmost occurrence of $w_s$ in $w[j+2..n]$
6:     $j := j' + |w_s| - 1, s := s + 1$
7: end while
8: if $w_m$ is a suffix of $w[j+2..n]$ then return yes else return no
```

Proof. We claim that Algorithm 1 solves the matching problem for PAT_{reg} in linear time. The correctness of this algorithm follows from the observation that there is a substitution h with $h(\alpha) = w$ if and only if there is a substitution g, such that, for every s with $1 \leq s \leq m$, $g(w_0\Pi_{i=1,s}x_i w_i)$ is the shortest prefix of w that matches the pattern $w_0\Pi_{i=1,s}x_i w_i$ (the *if* direction of this statement is trivial, while the *only if* direction follows by induction). Next, we show that Algorithm 1 can be implemented such that it has a linear running-time. To this end, we observe that if we perform the search for w_s in $w[j+2..n]$ with the Knuth-Morris-Pratt (KMP) algorithm, then every iteration of the while-loop requires time $O(|w_s| + |w[j+2..j'+|w_s|-1]|)$, which, over all iterations, sums up to $O(|w|)$. Since all other tasks can obviously be carried out in time $O(|w|)$, the overall running-time is $O(|w|)$. □

We stress that for constant alphabets our result is similar to that in [39]. However, that result uses string matching strategies based on finite automata, thus the O-denotation used to express its complexity hides a factor depending on $|\Sigma|$. For our algorithm, this is not the case: the complexity we get for solving the matching problem does not depend at all on the size of the terminal alphabet. Moreover, our algorithm can be easily adapted to the case when allowing variables to map to the empty word, a scenario neglected otherwise in this paper.

This would be also true for the following algorithm that solves the matching problem for $\text{PAT}_{\text{rvar}\leq 1}$ in cubic time: *For all assignments of factors w_i of w to the repeated variable x, run the previously obtained linear-time algorithm for the matching problem for the regular pattern obtained from the pattern α by substituting x by w_i.* In the following, we show how to further improve this simple algorithm towards quadratic time complexity.

Theorem 4.2. *The matching problem for $\text{PAT}_{\text{rvar}\leq 1}$ is solvable in quadratic time.*

Proof. Let us consider a pattern α in which exactly one variable, denoted by x, occurs more than once (if no such variable exists, then the statement follows as in the algorithm solving the matching problem for regular patterns, see Theorem 4.1).

The main idea behind the algorithm used to obtain this result is to find an assignment for the repeating variable x from the input pattern α, such that all terminal factors are well placed within the word, and then fill up (using a linear pattern matching algorithm to correctly align the maximal terminal factors of the pattern inside the word) the remaining spaces with the help of the rest of the variables from the pattern, since they occur only once.

Let again $n = |w|$. We split the discussion in cases: the *simple cases* and the *involved case*.

Simple cases. If $\alpha = w'x\beta$ with $w' \in \Sigma^*$, then α matches w if and only if α' matches w, where $\alpha' \in \textsc{Pat}_{\mathrm{reg}}$ is obtained from α by substituting x with some factor $w[|w'| + 1..|w'| + i]$, for some i with $1 \leq i \leq |w| - |w'|$. Hence, according to our solution for the matching problem for regular patterns, it is decidable whether α matches w in time $O(n^2)$. Obviously, the same holds when $\alpha = \beta x w'$ with $w' \in \Sigma^*$.

Involved Case. Assume that we are no longer in any of the simple cases discussed above. The rest of the proof deals with the more involved case, namely when $\alpha = \beta x \gamma x \delta$, where β and δ contain at least one variable each and $|\beta|_x + |\delta|_x = 0$ (in the following, we denote this property by †). Let us denote the number of occurrences of x in α by $N = |\alpha|_x$. We factorise α as

$$\alpha = w_0 \Pi_{i=1,m}(\beta_i w_i \gamma_i w'_i)\beta_{m+1} w_{m+1},$$

such that $m \leq N$ and

- $w_0, w_{m+1}, w_j, w'_j \in \Sigma^*$, where $1 \leq j \leq m$,
- β_j starts and ends with a variable and $|\beta_j|_x = 0$, where $1 \leq j \leq m + 1$, and
- γ_j starts and ends with variable x and $\gamma_j \in (\Sigma \cup \{x\})^+$, where $1 \leq j \leq m$.

For legibility, we write $\alpha_j = w_0 \Pi_{i=1,j}(\beta_i w_i \gamma_i w'_i)$, where $1 \leq j \leq m$.

It is easy to see that each pattern α with property † has a unique factorisation of this form, which can be computed in $O(|\alpha|)$ time. Intuitively, by defining the patterns γ_i, we identified the length-maximal factors of α that contain only the variable x and terminals, and start and end with x. Now, while trying to find an assignment of the variables of α so that this pattern matches w, we can assign values to the variables occurring in the factors β_i independently, but we need to check that the factors of w corresponding to the patterns γ_i induce the same assignment of each occurrence of the variable x.

We first run a series of preprocessing steps, denoted with § *Preprocessing* in the following.

§ *Preprocessing 1.* We construct for the words w and $w\alpha$ the suffix arrays, the *LCP* and the *LCS* data structures. It is clear that for any terminal factor u of α and position i of w we can now test in constant time whether u starts or ends at position i in w.

§ *Preprocessing 2.* Further, we try to find the possible matches for the factors β_j of α that do not contain the variable x. To this end, we define the $n \times (m + 1)$ matrix $M[\cdot][\cdot]$ with $M[i][j] = \ell$ if and only if $w[i..\ell]$ is the shortest factor starting on position i of w that is matched by β_j. As β_j starts and ends with variables that occur only once in α, then $w[i'..\ell']$ matches β_j for all $i' \leq i$ and $\ell' \geq M[i][j]$.

The matrix M can be computed by Algorithm 2, which we explain next. For a maximal terminal factor u of α, Line 2 identifies all its occurrences and requires time $O(|w|)$ (by using the KMP algorithm) and, by using the starting positions of the occurrences computed in Line 2, the for-loop in Lines 3 to 5 requires time $O(|w|)$, as well, to compute, for each $i \leq n$, the starting position of the first occurrence of u in $w[i..n]$. Consequently, the whole preprocessing of Lines 1 to 6 requires time $O(|\alpha|n)$. It can be easily verified that Lines 9 to 14 (i. e., computing a single entry $M[i][j]$) require time $O(|\beta_j|)$. Essentially, the algorithm follows the same greedy approach as in the algorithm matching regular patterns. Consequently, computing all entries $M[i][\cdot]$ is done in time $O(|\alpha|)$ and therefore the overall time spent for computing M is $O(n|\alpha|)$.

We can now move on to discuss the main algorithm.

§ *Main Algorithm.* Analogous to Algorithm 1, the algorithm for the matching problem for $\textsc{Pat}_{\mathrm{rvar}\leq 1}$ is based on the following observation: for every j with $1 \leq j \leq m$, there exists a substitution h with $h(\alpha_j) = w[1..p]$ and $h(x) = v$ if and only if there exists a substitution g with $g(x) = v$, $g(\alpha_j) = w[1..p]$, and $g(\alpha_{j-1})$ is the shortest prefix of w that matches α_{j-1} with respect to a

Algorithm 2 Compute matrix M – § *Preprocessing 2*

```
Input: pattern $\alpha$, and a word $w \in \Sigma^*$ with $n = |w|$
Output: matrix $M$
 1: for every maximal terminal factor $u$ of $\alpha$ do
 2:     compute all starting positions of occurrences of $u$ in $w$
 3:     for $i = 1, 2, \ldots, n$ do
 4:         compute $d_{u,i} = \min\{j \mid i \leq j \leq n, u = w[j..j + |u| - 1]\}$
 5:     end for
 6: end for
 7: for $i = 1, 2, \ldots, n$ do
 8:     for $j = 1, 2, \ldots, m + 1$ do
 9:         let $u_1, u_2, \ldots, u_s$ be the maximal terminal factors occurring in $\beta_j$
10:         $g_1 := d_{u_1, i+1}$
11:         for $h = 2, \ldots, s$ do
12:             $g_h := d_{u_h, g_{h-1}+|u_{h-1}|+1}$
13:         end for
14:         $M[i][j] := g_s + |u_s| + 1$
15:     end for
16: end for
```

substitution that maps x to v. Therefore, for a fixed v, checking whether there exists a substitution h with $h(\alpha) = w$ and $h(x) = v$ can be done by Algorithm 3.

Algorithm 3 Check whether there exists h with $h(\alpha) = w$ and $h(x) = v$

```
Input: pattern $\alpha$, word $w \in \Sigma^*$ with $n = |w|$, factor $v$ of $w$
Output: yes or no according to whether $\alpha$ matches $w$
1: if $w$ starts with $w_0$ then set $p_0 := |w_0|$; else return no
2: for $i = 1, 2, \ldots, m - 1$ do
3:     $p_i := p_{i-1} + M[p_{i-1} + 1][i]$
4:     find the starting position $p'_i$ of the leftmost occurrence of $w_i h(\gamma_i) w'_i$ in $w[p_i + 1..n]$
5:     if $w_i h(\gamma_i) w'_i$ does not occur in $w[p_i + 1..n]$ then set $p'_i = n + 1$
6:     if $p'_i \leq n$ then $p_i := p'_i + |w_i h(\gamma_i) w'_i| - 1$; else return no.
7: end for
8: $p_m := p_{m-1} + M[p_{m-1} + 1][m]$
9: if $w_{m+1}$ is a suffix of $w[p_m + 1..n]$ then return yes; else return no.
```

In order to solve the matching problem for $\text{PAT}_{\text{rvar}\leq 1}$ as sketched above, we would have to try all factors of w as possible images for x, which leads to a cubic running-time. Indeed, for each such factor, and for all $i \leq m - 1$ we do the following. In Line 3 we check if β_i occurs at position $p_{i-1} + 1$ in constant time. Then we need to run a linear-time pattern matching algorithm to see where $w_i h(\gamma_i) w'_i$ occurs first, to the left of the shortest factor starting after p_{i-1} and matching β_i.

To show our claim, it remains to explain how this general strategy can be implemented in quadratic time. More precisely, we are looking for an efficient exploration of the possible assignments of the values v to x, so that Line 4 of Algorithm 3 can also be executed more efficiently. This is described in the pseudocode of Algorithms 4 and 5, while the details are explained in the rest of this proof.

Algorithm 4 Check whether there exists h with $h(\alpha) = w$

```
Input: pattern $\alpha$ with $N = |\alpha|_x$, word $w \in \Sigma^*$ with $n = |w|$
Output: yes or no according to whether $\alpha$ matches $w$
 1: for $\ell = 1, 2, \ldots, n$ do
 2:     partition the suffix array of $w$ into clusters, by grouping together suffixes whose LCP $\geq \ell$
 3:     let $C_1, \ldots, C_p$ be the clusters with at least $N$ elements
 4:     for $i = 1, 2, \ldots, p$ do
 5:         Let $v$ be the common prefix of length $\ell$ of the suffixes of $C_i$
 6:         Check if there exists $h$ with $h(\alpha) = w$ and $h(x) = v$ using Algorithm 5 for $C_i$ and $v$.
 7:         if yes then return yes
 8:     end for
 9: end for
10: return no
```

§ *An Efficient Implementation of the Main Algorithm.* The idea is that, for every integer ℓ from 1 to n, we check if there exists a substitution mapping α to w, where x is mapped to a factor of length ℓ. Let us fix such an ℓ. In linear time, we can partition the suffix array of w in several *clusters* (i.e., blocks of consecutive positions in the suffix array that are not extendable to the left or right) such that the suffixes contained in one cluster are sharing a common prefix of length at least ℓ (note that this can be done by moving through the suffix array and performing *LCP* queries with respect to each two consecutive suffixes). It is important to note that no matter to what factor v of length ℓ the variable x is mapped, if the image of each γ_i under this mapping is indeed a factor of w, then they all occur as prefixes of some suffixes contained in the same cluster from the ones defined above, with at least N (the number of occurrences of x in α) elements, i. e., exactly the cluster where the suffixes share a common prefix starting with v, and there is a sufficient number of them to cover all occurrences of x. Moreover, in linear time, by traversing once the suffix array, we order (with radix sort) the suffixes in all clusters increasingly w.r.t. their starting position. Now, not only that all the images of the patterns γ_i under the aforementioned mapping occur as prefixes of suffixes from the same cluster, but the images of the γ_i occur in order of their appearance in α within the cluster.

So, once ℓ is fixed, we also fix a cluster of at least N suffixes sharing a prefix of length at least ℓ. Now, we try to find a substitution that maps α to w such that x is mapped to v, the common prefix for the cluster. This process is described in pseudo-code in Algorithm 5.

We start with $j = 1$, $p = |w_0| + 1$ and a token placed on the first element of the cluster. Let $s = M[p][j]$ (i.e., $w[p..s]$ is the shortest prefix of $w[p..n]$ that is an image of β_j); initially, $s = M[|w_0| + 1][1]$ (i.e., $w[1..s]$ is the shortest prefix of w that is the image of $w_0\beta_1$). We now traverse the cluster, left to right, from the position pointed by the token and moving the token accordingly, until we find the first suffix starting after position $s + |w_j|$ and preceded by w_j (checked in $O(1)$ time by an *LCS* query). Assume we found such a suffix $w[r..n]$. We want to check whether it starts with the image of $\gamma_j w'_j$, where $\gamma_j = (\Pi_{i=1,q_j}(xu_{i,j}))x$ for some q_j and maximal terminal factors $u_{i,j}$. The maximum integer h with $(\Pi_{i=1,h}(vu_{i,j}))v$ a prefix of $w[r..n]$ is found in time $O(h)$ by $2h + 1$ *LCP* queries, and we store $pos = r$.

If $h = q_j$ and $(\Pi_{i=1,q_j}(vu_{i,j}))v$ is followed by w'_j, then we found the leftmost occurrence corresponding to $w_j\gamma_j w'_j$ in w to the right of s, where x is mapped to v; if p' is the last position of this occurrence, then we identified the shortest prefix $w[1..p']$ of w that can be the image of α_j in a

Algorithm 5 Check whether there exists h with $h(\alpha) = w$ and $h(x) = v$, with v a common prefix of all suffixes in a cluster C

```
Input: pattern $\alpha$, word $w \in \Sigma^*$ with $n = |w|$, cluster $C$, word $v \in \Sigma^*$ a common prefix of the suffixes of $C$
Output: yes or no according to whether $h(\alpha) = w$ for a substitution with $h(x) = v$
 1: assume $C$ contains the positions $r_1 < \ldots < r_u$ of $w$ (corresponding to the suffixes $w[r_i..n]$ of $w$, with $1 \le i \le u$
 2: Set $t = 1$ (position of the token), $p = |w_0| + 1, j = 1$ (we search for matches for $\gamma_j$)
 3: while $j \le m$ do
 4:   assume $\gamma_j = (\Pi_{i=1,q_j}(xu_{i,j}))x$
 5:   set $h = 0, pos = 1$ ($h$ tracks the longest prefix of $\gamma_j$ matched so far, and $pos$ the position where this prefix occurs), $s = M[p][j]$, $d = 0$ (tracks whether $\gamma_j$ was completely matched)
 6:   while $d = 0$ do
 7:     set $f = 0$ (used to search the first $w[r_t..n]$ is preceded by $w_j$)
 8:     while $f = 0$ do
 9:       if $r_t - |w_j| > s$ and $w[r_t - |w_j|..r_t - 1] = w_j$ then set $f = 1$
10:       set $t = t + 1$
11:       if $t > u$ then set $f = 2$
12:     end while
13:     if $f = 2$ then return no
14:     find the maximum $h' > h$ such that $(\Pi_{i=1,h'}(vu_{i,j}))v$ is a prefix of $w[r_t..n]$; this is done by $2(h' - h) + 1$ LCP-queries: check that LCP of $w[pos..n]$ and $w[r_t..n]$ is at least $|(\Pi_{i=1,h}(vu_{i,j}))v|$, then that the prefix $(\Pi_{i=1,h+g}(vu_{i,j}))v$ of $w[r_t..n]$ is followed by $u_{h+g+1}$ and then by $v$ for $g$ from 0 to $h' - h - 1$
15:     if we found $h' > h$ in the previous step then update $h = h'$ and $pos = r_t$
16:     let $\ell = |(\Pi_{i=1,h}(vu_{i,j}))v|, p' = r_t + \ell + |w'_j| - 1$
17:     if $h = q_j$ and $w[r_t + \ell..p'] = w'_j$ then set $p = p' + 1, j = j + 1, d = 1$ ($w[r_t..p']$ matches $\gamma_j$)
18:   end while
19: end while
20: return yes.
```

substitution that maps x to v. We take now $p = p' + 1$, increase j by one and restart the procedure if $j \le m$ still holds.

If $h < q_j$, or $h = q_j$ and w'_j does not follow $(\Pi_{i=1,h}(vu_{i,j}))v$, then we keep traversing the cluster, and, moving the token, we look for a suffix that is preceded by w_i and shares a prefix with $w[r..n]$ at least as long as $(\Pi_{i=1,h}(vu_{i,j}))v$. As soon as we found one, say $w[r'..n]$, we find the maximum integer $h' > h$ such that $(\Pi_{i=1,h'}(vu_{i,j}))v$ is its prefix. Finding h' takes at most $2(h' - h) + 1$ *LCP*-queries, as we only need to first check if $(\Pi_{i=1,h}(vu_{i,j}))v$ occurs as a prefix of $w[r'..n]$ (and for this we do an LCP-comparison between $w[r'..n]$ and the suffix $w[pos..n]$ of the cluster where we already found $(\Pi_{i=1,h}(vu_{i,j}))v$), and then check if all factors of the concatenation $(\Pi_{i=h+1,h'}(vu_{i,j}))v$ occur consecutively after the prefix $(\Pi_{i=1,h}(vu_{i,j}))v$ of $w[r'..n]$. If $h' = q_j$ and $(\Pi_{i=1,h'}(vu_{i,j}))v$ is followed by w'_j, then we proceed as above; otherwise, we continue the traversal of the cluster with $w[r'..n]$ in the role of $w[r..n]$ and h' in the role of h. Thus, finding the right h within a cluster of size k, when it exists, will not require more than $\sum_{i=1}^{k-1} 2(h_{i+1} - h_i) = 2h_k - 2h_1 \le 2k$ *LCP*-queries.

If finally we have $j = m + 1$, then we found the shortest prefix $w[1..p_m]$ of w that is an image of α_m while mapping x to v, and we can proceed as previously described to decide whether there exists a substitution that maps α to w. The processing of a cluster clearly takes $O(k+N) = O(k)$, where $k \geq N$ is its size.

If we cannot find the prefix $w[1..p']$ of w that is the image of α_j in a substitution that maps x to v within the current cluster, then the image of x was chosen wrongly, and we should try another cluster (thus choose another value v for x) defined for the same length, or even another length. As such, we might repeat the above procedure for all clusters of size at least N. As the total number of elements in these clusters is upper bounded by $n - \ell$ for the fixed ℓ, it is clear that their total processing takes $O(n)$ time.

We have to consider all possible values for ℓ. The time spent for each one is $O(n)$, so, in total our algorithm needs $O(n^2)$ time to decide whether there exists a substitution that maps α to w. The correctness follows from the explanations above. □

To solve the matching problem for patterns with at most k repeated variables, we choose the images (starting and ending positions in w) of $k-1$ of the k repeated variables, and then get a pattern with only one repeated variable. Further, we apply Theorem 4.2 on this pattern. As a general result, we will use that $\binom{n}{k} \in O(\frac{n^k}{k!})$.

Theorem 4.3. *The matching problem for* $\text{PAT}_{\text{rvar}\leq k}$ *is solvable in* $O\left(\frac{n^{2k}}{((k-1)!)^2}\right)$ *time, where* n *is the length of the input word.*

Proof. Let $w \in \Sigma^+$ with $|w| = n$ and let $\alpha \in \text{PAT}_{\text{rvar}\leq k}$. Let x be the leftmost repeating variable and let $x_1, x_2, \ldots, x_{k-1}$ be the other repeating variables of α, indexed in the order they occur. Our idea is to first generate all possible images for the variables x_i, where $1 \leq i \leq k-1$, and, secondly, for each such assignment, to try to find a suitable image for x such that the substitution maps α to w.

Let us first look at the part where we generate the images of the variables $x_1, x_2, \ldots, x_{k-1}$. We first choose $k-1$ numbers $j_1, j_3, \ldots, j_{2k-3}$ from $\{1, \ldots, n\}$ and then $k-1$ numbers $j_2, j_4, \ldots, j_{2k-2}$ from $\{1, \ldots, n\} \setminus \{j_1, j_3, \ldots, j_{2k-3}\}$. With $i_{2\ell-1} = \min\{j_{2\ell-1}, j_{2\ell}\}$ and $i_{2\ell} = \max\{j_{2\ell-1}, j_{2\ell}\}$ for $\ell = 1, \ldots, k-1$, we obtain $k-1$ many non-empty factors $w_\ell = w[i_{2\ell-1}..i_{2\ell}]$. Consequently, there are less than $\frac{n^{2k-2}}{((k-1)!)^2}$ ways of choosing these factors.

Now, for each possible substitution for the variables $\{x_1, x_2 \ldots, x_{k-1}\}$, we produce a pattern α' where just x is repeated. Further, we can use the algorithm in the proof of Theorem 4.2 to check in $O(n^2)$ time whether the substitution can be extended (i. e., by an image for x), such that α' is mapped to w.

Clearly, this decides the existence of a substitution mapping α to w. The running time is bounded by $O\left(n^2 \frac{n^{2k-2}}{((k-1)!)^2}\right) = O\left(\frac{n^{2k}}{((k-1)!)^2}\right)$. □

4.2 Non-cross patterns

Next, we present an algorithm for matching non-cross patterns, which, in a similar way as the algorithm for patterns with only one repeated variables was extended to patterns with a bounded number of variables, shall then be extended to patterns with a bounded scope coincidence degree. The following combinatorial results are well known (see, e. g., [12]).

Lemma 4.4 ([10]). *Let u_1, u_2, and u_3 be primitive words, with $|u_1| < |u_2| < |u_3|$ and u_i^2 prefixes (suffixes) of a word w, for every i with $1 \leq i \leq 3$. Then $2|u_1| < |u_3|$. As a consequence, we have $|\{u|u$ primitive, u^2 prefix (respectively, suffix) of $w\}| \leq 2\log|w|$.*

Assume that $w \in \Sigma^*$ is of length n. For each i with $1 \leq i \leq n$ we define the set

$$P_i = \{u \mid u \text{ is a primitive word such that } u^2 \text{ is a suffix of } w[1..i]\}.$$

Lemma 4.4 shows that $|P_i| \leq 2 \log n$ for every i with $1 \leq i \leq n$. Generally, we can represent the elements of P_i in various efficient manners (e. g., for each $u \in P_i$ it is enough to store its length).

Lemma 4.5 ([10]). *Let $w \in \Sigma^*$ be a word of length n. We can compute in $O(n \log n)$ time all the sets P_i associated to w, where $i \in \{1, 2, \ldots, n\}$.*

Note that in [10] there are examples of words of length n for which $\sum_{i \leq n} |P_i| \in \Theta(n \log n)$.
We recall the theorem of Fine and Wilf (also called periodicity lemma).

Theorem 4.6 (Fine and Wilf [29]). *If $\alpha \in u\{u, v\}^*$ and $\beta \in v\{u, v\}^*$ have a common prefix of length at least $|u| + |v| - gcd(|u|, |v|)$, then $u, v \in \{t\}^+$ for a word t.*

A rather direct corollary of Theorem 4.6 is the following:

Theorem 4.7 ([29]). *Let $\alpha \in \Sigma^*$ be a word having two periods p and q. If $|\alpha| \geq p + q - \gcd(p, q)$, then α has also the period $\gcd(p, q)$.*

A standard application of Theorem 4.7 is the following. If we have the equality $u^\gamma u' = v^\delta v'$ for some words u, v, with u' a prefix of u and v' a prefix of v, and $\gamma, \delta \geq 2$, then we can conclude that there exists a word t such that $u, v \in \{t\}^*$. Indeed, the word $\alpha = u^\gamma u' = v^\delta v'$ has as periods both $p = |u|$ and $q = |v|$, and length $|\alpha| \geq \max\{2p, 2q\} \geq p + q - \gcd(p, q)$. Then, α has also the period $\gcd(p, q)$. This divides both p and q, so there exists a word t (prefix of α) such that $u, v \in \{t\}^*$ and $|t| = \gcd(p, q)$.
By similar reasons, we get also that if $u'u^\gamma = v'v^\delta$ holds for some words u, v, with u' a suffix of u and v' a suffix of v, and $\gamma, \delta \geq 2$, then we can conclude that there exists a word t such that $u, v \in \{t\}^*$. Like before, $\alpha = u'u^\gamma = v'v^\delta$ has as periods both $p = |u|$ and $q = |v|$. So, α has also the period $\gcd(p, q)$. Reading the period from the end of α towards its beginning, we get that there exists a word t (suffix of α) such that $u, v \in \{t\}^*$ and $|t| = \gcd(p, q)$.
The next result states a necessary condition for the primitivity of a word.

Theorem 4.8. *[29] If u is a primitive word, then the only occurrences of u as a factor of uu are as prefix or suffix.*

Next, we extend the results of Lemmas 4.4 and 4.5. Instead of primitively rooted squares, we consider words of the form uvu for some fixed word v, with uv primitive (or, equivalently, with vu primitive).

Proposition 4.9. *For a fixed word v, let $u_1vu_1, u_2vu_2, u_3vu_3$ be prefixes (suffixes) of a word w with $|u_1| < |u_2| < |u_3|$ and u_iv primitive, for every i with $1 \leq i \leq 3$. Then $3|u_1|/2 < |u_3|$. As a consequence, we have $|\{uvu | uv$ primitive, uvu prefix (respectively, suffix) of $w\}| \in O(\log |w|)$.*

Proof. Let us assume that $|u_3| \leq 3|u_1|/2$. Clearly, we get that $|u_2| \leq 3|u_1|/2$ and $|u_3| \leq 3|u_2|/2$, as well. In the following, we will show that this assumption leads to a contradiction (note that our line of reasoning is completely illustrated in Fig. 1).
As u_1 is a prefix of u_2, looking at the alignment between the second u_1 from u_1vu_1 with the prefix u_1 of the second u_2 from u_2vu_2, since $|u_2| \leq \frac{3|u_1|}{2}$ we get that the period of u_1 is less than $\frac{|u_1|}{2}$ and, therefore, u_1 is periodic. Indeed, the second u_2, which starts with u_1, occurs in u_2vu_2 on position $|u_2| + |v| + 1$, while the factor of u_2vu_2 aligned to the second u_1 of u_1vu_1 occurs in u_2vu_2 on position $|u_1| + |v| + 1$. The difference between these positions, which gives the period of u_1, is $|u_2| - |u_1| \leq |u_1|/2$. Thus, we have that $u_1 = t^\alpha t'$, for an integer α with $\alpha \geq 2$ and t' a prefix of t, for some primitive word t (with $|t| = per(u_1)$).
Let us also note that, by the same reasoning as above, the prefix u_1 of the second u_2 from u_2vu_2 starts at least $|t|$ symbols to the right of the starting position of the factor of u_2vu_2 aligned to the

second u_1 of u_1vu_1. This means $|u_2v| + 1 - (|u_1v| + 1) = |u_2| - |u_1| \geq |t|$. In a similar way we get $|u_3| - |u_2| \geq |t|$.

Since t is primitive, as to not get a contradiction with Theorem 4.8, there exists an integer δ with $\delta > 0$ such that $u_1vt^{\delta} = u_2v$. Indeed, u_2 starts with u_1, so it starts with a factor t. Also, the overlap between the factor of u_2vu_2 aligned to the second u_1 of u_1vu_1 and the second u_2 of u_2vu_2 has length at least $2|u_1| + |v| - |u_2| - |v| = 2|u_1| - |u_2| \geq |u_1|/2 \geq |t|$. So, if u_2v would not have the form u_1vt^{δ} then the prefix t of the second u_2 would occur inside a factor t^2 of the factor of u_2vu_2 aligned to the second u_1 of u_1vu_1, yielding a contradiction to the primitivity of t.

Moreover, the same holds with respect to u_3v, i. e., there exists an integer ψ with $\psi > 0$ such that $u_1vt^{\psi} = u_3v$. Hence, v is a suffix of vt^{ψ}, which implies that $v = t''t^{\beta}$ where t'' is a suffix of t and β is an integer with $\beta \geq 0$.

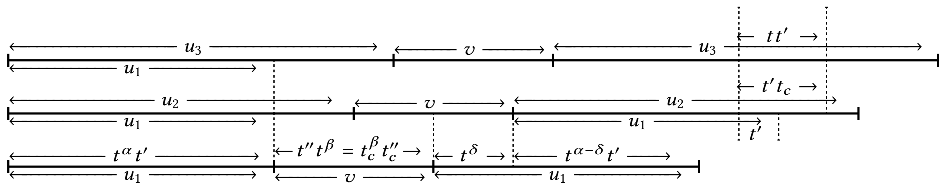

Fig. 1. Left alignment of three factors u_1vu_1, u_2vu_2, and u_3vu_3 with u_1v, u_2v, and u_3v primitive, such that $|u_1| < |u_2| < |u_3|$ and $|u_3| \leq \frac{3|u_1|}{2}$.

We can assume that $|u_1v|$ is not divisible by $|t|$ since otherwise $t't'' = t$, so u_1v, is a repetition, which contradicts the primitivity of u_1v. This means that $v = t_c^{\beta}t''$ where t_c is a conjugate of t, obtained by rotating t by $|t''|$ positions.

We now check what factor occurs in u_2vu_2 and u_3vu_3 at position $|u_2| + |v| + |u_1| - |t'| + 1$. In u_2vu_2 we have $t't_c$ (look after the first occurrence of u_1 in u_2.) Indeed, $|u_2| - |u_1| \geq |t|$, so at least one full factor of length $|t|$ occurs in u_2 after its prefix u_1.

In u_3vu_3, at position $|u_2| + |v| + |u_1| - |t'| + 1$, starts a factor tt'. Indeed, $|u_2| + |v| + |u_1| - |t'| + 1 \geq |u_3| + |v| + 1$, because $|u_3| \leq 3|u_1|/2$ and $|t'| < |t| \leq |u_1|/2$. Also, we know that $u_1vt^{\psi} = u_3v$, so $|u_2| + |v| + |u_1| - |t'| - |u_3v| = |u_2| + |u_1| - |u_3| - |t'|$ is both upper bounded by $|u_1| - |t| - |t'|$ and divisible by t. It follows that, at position $|u_2| + |v| + |u_1| - |t'| + 1$ we have a factor of u_1, occurring with an offset which is both upper bounded by $|u_1| - |tt'|$ and also a multiple of $|t|$ from the beginning of u_1. Each such factor starts with tt' so our claim follows. Hence, $t't_c = tt'$ holds.

Since $vu_1[1..|u_2| - |u_1|] = u_2[|u_1| + 1..|u_2|]v$ and $v = t''t^{\beta} = t_c^{\beta}t''$ with t and t_c primitive, we can conclude that $vt^{\gamma} = t_c^{\gamma}v$, where $\gamma = \frac{|u_2|-|u_1|}{|t|}$ (note that γ must be an integer). In particular, it follows that $(t'v)t^{\gamma} = t't_c^{\gamma}v$ and, since $t't_c = tt'$, we can conclude that $(t'v)t^{\gamma} = t^{\gamma}(t'v)$. Consequently, the words $t'v$ and t satisfy the conditions of the periodicity lemma (Theorem 4.7); thus, we can conclude that $t'v$ and t are powers of the same word. This leads to a contradiction, as $u_1v = t^{\alpha}t'v$, and if t and $t'v$ are powers of the same word we would obtain that u_1v is a repetition. □

Consider two words $w, v \in \Sigma^*$ with $|w| = n$. Following the case of the primitively rooted squares, for every i with $i \leq n$ we define the set

$$R_i^v = \{u \mid uvu \text{ is a suffix of } w[1..i] \text{ with } uv \text{ primitive}\}.$$

Again, R_i^v can be stored efficiently by the lengths of the words it contains. By Proposition 4.9, $\sum_{i=1}^{n} |R_i^v| \in O(n \log n)$. Moreover, as uvu with uv primitive is just a primitively rooted square when

$v = \varepsilon$, it follows that for certain values of v, we have $\sum_{i=1}^{n} |R_i^v| \in \Theta(n \log n)$. Note that, in the following, if $v = \varepsilon$, then $R_{v,i} = P_i$

Our next result extends in a non-trivial manner the result of Lemma 4.5. However, we continue first with the description of some data structures and remarks that constitute the basis of this result. The following observation is straightforward, but nevertheless useful later on.

Remark 1. *For any word w of length n and a suffix $w[i..j]$ of $w[1..j]$, where j is an integer with $j \leq n$, the maximum positive integer ℓ such that $w[i..j]^{\ell}$ is a suffix of $w[1..j]$ is $\ell = \left\lfloor \frac{LCS_w(j, i-1)}{j-i+1} \right\rfloor + 1$.*

The most important data structure that we shall use in the remainder of this section is the dictionary of basic factors (DBF, for short, see [11]). The DBF of a word w is a data structure that associates labels to the factors of the form $w[i..i + 2^k - 1]$ (called basic factors), for integers i, k with $0 \leq k \leq \lfloor \log(n) \rfloor$ and $1 \leq i \leq n - 2^k + 1$, such that two identical factors get the same label and we can retrieve the label of such a factor in $O(1)$ time. The dictionary of basic factors of a word of length n is constructed in $O(n \log n)$ time (see [11]). Generally, for every integer k the DBF contains an array $A_k[\cdot]$, such that $A_k[i]$ is the label of $w[i..i + 2^k - 1]$.

We observe that a basic factor $w[i..i + 2^k - 1]$ occurs in a factor $w[j..j + 2^{k+1} - 1]$ either at most twice or the positions where it occurs form an arithmetic progression of ratio $per(w[i..i + 2^k - 1])$ (see [26]). Hence, the occurrences of $w[i..i + 2^k - 1]$ in $w[j..j + 2^{k+1} - 1]$ can be represented in a compact manner, i. e., either by the two starting positions, or by the starting position of the progression and its ratio. Moreover, the occurrences of $w[i..i + 2^k - 1]$ in $w[i..i + 2^{k+2} - 1]$ can also be represented in a compact manner, since they are given by the occurrences of the basic factor $w[i..i + 2^k - 1]$ in the basic factors $w[i..i + 2^{k+1} - 1]$, $w[i + 2^k..i + 2^k + 2^{k+1} - 1]$ and $w[i + 2^{k+1}..i + 2^{k+2} - 1]$. According to this fact it is clear that one can preprocess the DBF of w even more, so that we can answer in constant time the following queries:

(♣) "Given positive integers i and k, return the compact representation of the occurrences of $w[i..i + 2^k - 1]$ in $w[i..i + 2^{k+2} - 1]$."

Moreover, fixing some word v, we can also produce in $O(n \log n)$ time a data structure answering the following type of queries in $O(1)$ time:

(♠) "Given positive integers i and k, return the compact representation of the occurrences of $w[i..i + 2^k - 1]$ in $w[i - |v| - 2^{k+1}.. i - |v| - 1]$."

The following lemma follows directly from Theorem 8.1 of [27]. There it is shown that one can construct, for a given word w of length n, in $O(n)$ time, a data structure allowing to decide in constant time whether some factor of w is periodic, and, if yes, to compute its shortest period. To test then if some factor of the given word is primitive it is enough to test whether that factor is periodic and its shortest period divides its length, using the aforementioned data structure.

Lemma 4.10. *[27] We can produce in $O(n)$ time a data structure that allows us to test the primitivity of each factor of a word w of length n in $O(1)$ time.*

An obvious consequence of Lemma 4.4 is that if we consider all primitively rooted squares starting at some position i with root lengths between ℓ and 2ℓ, then there are at most two of them, since the root length of a third one must exceed 2ℓ. We seperately state this for a special case, which shall be applied in the next proof (note that in the following lemmas the fact that we can compute the squares in $O(n \log n)$ is a consequence of Lemma 4.5).

Lemma 4.11. *Let $w[i..j] = v$ be a factor of a word w of length n with $per(w[i..j]) \leq \frac{j-i+1}{5}$. There are at most two primitively rooted squares z_1^2 and z_2^2 which are prefixes of $w[i..n]$ such that*

$$j - i + 1 - per(w[i..j]) \leq |z_1| < |z_2| \leq j - i + 1.$$

Finding these squares for each occurrences of v takes $O(n \log n)$.

Lemma 4.12. *Let $w[i..j] = v$ be a factor of a word w of length n and j' a position such that $|w[j+1..j']| \leq \frac{j-i+1}{5}$. There are at most two primitively rooted squares z_1^2 and z_2^2 which are prefixes of $w[i..n]$ such that*

$$j - i + 1 \leq |z_1| < |z_2| \leq j' - i + 1.$$

Finding these squares for each occurrence of v takes $O(n \log n)$.

Now we are ready to state our extension of Lemma 4.5.

Proposition 4.13. *Given two words $w, v \in \Sigma^*$ with w of length n, we can compute in $O(n \log n)$ time all the sets R_i^v associated to w, for all i with $1 \leq i \leq n$.*

Proof. We begin by constructing for the word wv the suffix array, and *LCP* and *LCS* data structures. Furthermore we also construct the above DBF data structure for w, together with its extensions (♣) and (♠), and, due to Lemma 4.10, a data structure that allows us to check whether a factor of w is primitive in constant time (in the following, we shall perform primitivity checks without mentioning specifically that they only require constant time). We note that this preprocessing requires time $O(n \log n)$.

In the following, let $v' = v[1..per(v)]$, i. e., $v = (v')^\sigma v''$ for an integer σ and a prefix v'' of v'. We first assume that v' is small, more precisely, $|v'| < \frac{|v|}{6}$, which implies that there might be many occurrence of v in w (case A). The case $|v'| \geq \frac{|v|}{6}$ already bounds the number of occurrences of v in w and is somehow simpler; we shall discuss it later on (case B).

The general task that needs to be performed can be described as follows: we have to find all occurrences $w[i..i+|v|-1]$ of v in w that can be extended to both sides with the same factor u such that uv is primitive (we denote such a factor u as *extension* (*with respect to* i)), or, more formally, for each i with $w[i..i+|v|-1] = v$, we have to find all j with $w[i-j..i+|v|-1+j] = uvu$ and uv primitive. Obviously, doing this in a naive way is too time consuming. The crucial improvement of our procedure is due to the fact that, for an i with $w[i..i+|v|-1] = v$, we analyse, for all integers k with $0 \leq k \leq \lfloor \log(n) \rfloor$, all the possible candidates for extensions u that satisfy $2^k \leq |u| \leq 2^{k+1}$ simultaneously (we shall denote such extensions by the term k-*extension* (*with respect to* i)). Obviously, for a fixed k, a compact representation of these candidates can be efficiently retrieved due to the previously computed DBF data structure with its extensions (♣) and (♠).

Case A**:** $|v'| < \frac{|v|}{6}$.

In the following, let i, $1 \leq i \leq n$, be such that $w[i..i+|v|-1] = v$ and let k, $0 \leq k \leq \lfloor \log(n) \rfloor$, be arbitrarily chosen. We shall show how we can find all the k-extensions with respect to i in time proportional to their number. Consequently, by doing this for all occurrences of v in w (which can be easily found by the *LCP* data structure for wv) and all k, $0 \leq k \leq \lfloor \log(n) \rfloor$, we find all the elements of the sets R_i^v in time proportional to their total number, which, by Proposition 4.9, is $O(n \log n)$.

We note that all k-extensions have the prefix $t = w[i+|v|..i+|v|+2^k-1]$ and start inside the factor $w[i-2^{k+1}..i-1]$; thus, a compact representation of all starting positions of the basic factor t in $w[i-2^{k+1}..i-1]$, which is provided by the DBF data structure, contains the starting positions of all k-extensions and is therefore a suitable set of possible candidates for k-extensions. We only have to identify the actual k-extensions among these candidates and we shall next explain how this can be done efficiently.

Firstly, we consider the special case when there are at most two occurrences of t in $w[i-2^{k+1}..i-1]$, say $w[i_1..i_1+2^k-1]$ and $w[i_2..i_2+2^k-1]$. We can now easily check whether these candidates are actual k-extensions as follows: for $\ell \in \{1,2\}$, check whether $w[i_\ell..i-1]$ occurs at position $i+|v|$ (with an *LCP* query) and if so, whether $w[i..i+|v|-1]w[i_\ell..i-1]$ is primitive or not.

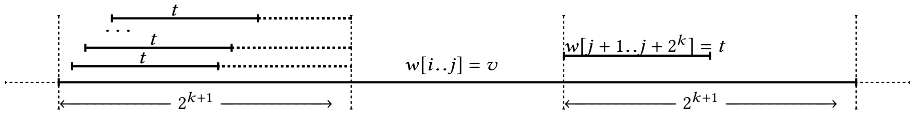

Fig. 2. Occurrences of t in $w[i-2^{k+1}..i-1]$ indicate the start positions of candidates for k-extensions (note that here $j = i+|v|-1$).

Next, we assume that the occurrences of t in $w[i-2^{k+1}..i-1]$ are given as an arithmetic progression (for an illustration see Fig. 2), which starts on position i_0 and has ratio p. Then $w[i_0..i-1] = x^\alpha x'y'$, where $|x| = p$, x' is a prefix of x with $|x'| < |x|$, and $x^\alpha x'$ is a word of period p that cannot be further extended to the right without breaking the period, i. e., if $y' \neq \varepsilon$, then $y'[1] \neq x[|x'|+1]$. In the following, we consider the two cases $y' \neq \varepsilon$ and $y' = \varepsilon$ separately and we shall start with the former, which is the simpler one.

Case $A.1$**:** $y' \neq \varepsilon$.
In this case, clearly, every k-extension with respect to i is not $|x|$-periodic. Let β be such that $x^\beta x''$ is the longest $|x|$-periodic prefix of $w[i+|v|..n]$, with x'' a prefix of x. If $\beta > \alpha$, then we do not have any k-extension with respect to i. If we would have one, then it would either be $|x|$-periodic (and be a factor of $x^\beta x''$, a contradiction), or it should start with $x^\beta x''$ which is a contradiction to the fact that every factor of $w[i_0..i-1]$ starts with at most x^α and $\alpha < \beta$. So, $\beta \leq \alpha$. By a reasoning similar to the above, we get that $x' = x''$ (or, again, the periodic prefixes of the two sides of the k-extension would not match) and that the extension should be $x^\beta x'y'$. We check whether $x^\beta x'y'$ is a suffix of $w[1..i-1]$ and a prefix of $w[i+|v|..n]$, and if $x^\beta x'y'v$ is primitive (in constant time, using the data structures we built). If yes, we found the single correct k-extension. (Case $A.1$) ⋄
Case $A.2$**:** $y' = \varepsilon$.
In this case, $w[i_0..i-1] = x^\alpha x'$. This implies that the candidates for k-extensions are the words $x^\gamma x'$, with $\gamma \leq \alpha$ and $\gamma \geq |t|/|x|$ (as at least one t should be contained in u). We shall check which of these are actual k-extensions by first determining (in constant time) all the values γ, for which $vx^\gamma x'$ is *not* primitive, i. e., $vx^\gamma x' = z^s$ for some primitive word z. Then we can output the values of $\gamma \geq 1$ for which $vx^\gamma x'$ is primitive in time proportional to their number.

As stated in the preliminaries, a word is primitive (respectively, non-primitive) if and only if all its conjugates are also primitive (respectively, non-primitive). Therefore, instead of looking for the values of γ such that $vx^\gamma x'$ is not primitive, we look for the values of γ such that $x^\gamma x'v$ is not primitive.

Recall that $v = (v')^\sigma v''$, where $|v'| = per(v)$. Then we can also write $v = v_2(v_1)^\sigma$ where v_1 is a primitive suffix of v with $|v_1| = per(v)$, by just reading the period of v from right to left. Therefore, we are interested in the values $\gamma \leq \alpha$ such that $x^\gamma x'v_2(v_1)^\sigma$ is not primitive.

First we check in constant time whether $x^\gamma x'v_2(v_1)^\sigma$ is primitive or not for $|t|/|x| \leq \gamma \leq 5$. This can be done in constant time, using the data structures we constructed. So, from now on we only look for $\gamma \geq \max\{|t|/|x|, 6\}$ such that $x^\gamma x'v_2(v_1)^\sigma$ is not primitive.

Accordingly, in the following we identify under which conditions can the word $x^\gamma x'v_2(v_1)^\sigma$ be written as z^s with z primitive.

Let us first assume that $x^\gamma x'v_2(v_1)^\sigma = z^s$ with $x \neq z \neq v_1$. If $s \geq 4$, then either $x^\gamma x'$ shares with z^s a prefix of length at least $|x|+|z|$ or $v_2(v_1)^\sigma$ shares with z^s a suffix of length $|z|+|v_1|$. In any case, we can apply Theorem 4.7 to get that one of x, v_1, or z is not primitive, a contradiction. Therefore, $s \leq 3$. If $s = 3$, then $|z|/2 \geq \max\{|x|, |v_1|\}$, as both γ and σ are at least 6. If $|x^\gamma x'| \geq 3|z|/2$ then we can apply Theorem 4.7 to $x^\gamma x'$ and z^3, that share a prefix of length at least $3|z|/2 \geq |z| + |x|$, and get that z is not primitive, a contradiction. If $|v_2(v_1)^\sigma| \geq 3|z|/2$ then we can apply Theorem 4.7 to $v_2(v_1)^\sigma$ and z^3, that share a suffix of length at least $3|z|/2 \geq |z| + |v_1|$, and get that z is not primitive, again a contradiction. So $s \neq 3$. If $s = 2$ the analysis is finer.

If $|x^\gamma x'| \geq |z| + |x|$, we can apply again Theorem 4.7 to $x^\gamma x'$ and z^2 and get that z is not primitive; similarly, if $|v_2(v_1)^\sigma| \geq |z| + |v_1|$. So, in those cases we reach contradictions. Therefore, we can assume $|x^\gamma x'| \leq |z| + |x|$ and $|v_2(v_1)^\sigma| \leq |z| + |v_1|$. Let us now assume that $|v_2(v_1)^\sigma| \leq |x^\gamma x'|$ (the other case can be treated identically). Then, it follows that $z = x^{\gamma'} x''$, where γ' is either γ or $\gamma - 1$ and x'' is a prefix of x. Equivalently, $z = x_c'' x_c^{\gamma'}$, where x_c is a conjugate of x. Also, z has the suffix $v_2(v_1)^\sigma$ (since $x^\gamma x' v_2(v_1)^\sigma = z^2$ and we assume that $|v_2(v_1)^\sigma| \leq |x^\gamma x'|$), and, moreover, we have on the one hand that $|v_2(v_1)^\sigma| = 2|z| - |x^\gamma x'| \geq 2|z| - (|z| + |x|) = |z| - |x| \geq 4|x|$ and on the other hand that $|v_2(v_1)^\sigma| \geq 6|v_1|$. Therefore, we can apply Theorem 4.7 to $z = x_c'' x_c^{\gamma'}$ and its suffix $v_2(v_1)^\sigma$, and get that $x_c = v_1$ or either x_c or v_1 would not be primitive (which would yield a contradiction). Because $(\sigma + 1)|v_1| \geq |v_2(v_1)^\sigma| \geq |z| - |x| \geq (\gamma' - 1)|x|$ we get $\sigma + 1 \geq \gamma' - 1$. As $x^{\gamma'+1} \geq |z| \geq \sigma|v_1|$, we get that $\gamma' + 1 \geq \sigma$. That is, $\sigma + 2 \geq \gamma' \geq \sigma - 1$. This formula gives us four variants for γ', and for each of them we have two respective variants for γ, such that $x^\gamma x' v_2(v_1)^\sigma$ could be a square. We try in constant time each variant, and store the values of γ for which $x^\gamma x' v_2(v_1)^\sigma$ is indeed a square, thus not primitive.

We now consider the case $x^\gamma x' v_2(v_1)^\sigma = z^s$ with $x = z$. It follows that $v_2(v_1)^\sigma = z'' z^r$ with $x' z'' = x = z$. If $r \geq 2$ we can apply Theorem 4.7 and get that either $z = v_1$ or z is not primitive (a contradiction). In the case $z = x = v_1$ it follows that $x^\gamma x' v_2(v_1)^\sigma$ is not primitive if and only if $x' v_2 = x$ (which can be checked in constant time), and if this holds then $x^\gamma x' v_2(v_1)^\sigma$ is not primitive for all γ. If $r = 1$ and $|z''| \geq |v_1|$, then we can still apply Theorem 4.7 to get that z is not primitive, a contradiction. The only remaining case is $r = 1$ and $|z''| < |v_1|$. In this case, as we know x and x', so also z'', we can determine in $O(1)$ time, by *LCP*-queries, whether $v = z'' x$. If yes, then, once again, $x^\gamma x' v_2(v_1)^\sigma$ is not primitive for all γ.

The last case, $x^\gamma x' v_2(v_1)^\sigma = z^s$ with $z = v_1$ is rather similar. This time, we get that $x^\gamma x' = z^r z'$, with $z' v_2 = v_1 = z$. If $r \geq 2$ we once again get that $z = x$, and the analysis is done as above. If $r = 1$ and $|z'| \geq |x|$ we get that z is not primitive, a contradiction. Finally (and a bit different to the above), if $r = 1$ and $|z'| < |x|$, then $x^\gamma x'$ must be equal to $v_1 z''$ (whose length is known). Thus, as we know $|x|$ and $|x'|$, we get the value of γ. We check by an *LCP* query whether $x^\gamma x' = v_1 z''$, and, if yes, we store that $x^\gamma x' v$ is not primitive.

By the explanations above, it follows that this analysis takes constant time. Once it is completed, we output the values γ for which $x^\gamma x' v$ (and, equivalently, $v x^\gamma x'$) is primitive. (Case *A*.2) ⋄

Case *B*: $|v'| \geq \frac{|v|}{6}$.

It remains to consider the case that the period of v is large, i. e., $|v'| \geq \frac{|v|}{6}$. In this case, the number of occurrences of v in w is at most $\frac{6n}{|v|}$, which is $O(\frac{n}{|v|})$. For each occurrence $v = w[i..i + |v| - 1]$, we can first try naively all the factors $u = w[i + |v|..j]$ with $|u| \leq 4|v|$ and check whether $u = w[i - |u|..i - 1]$ and vu is primitive. In this way, we get all extensions u with the property $|u| \leq 4|v|$. In order to compute the remaining extensions, we proceed similarly to case A.

More precisely, for an i with $w[i..i + |v| - 1] = v$, we analyse, for all integers k with $\log(4|v|) \leq k \leq \lfloor \log(n) \rfloor$, all the possible candidates for extensions u around $w[i..i + |v| - 1]$ that satisfy $2^k \leq |u| \leq 2^{k+1}$ simultaneously. Just like before, let $t = w[j + 1..j + 2^k]$, where $j = i + |v| - 1$. Clearly, t is a prefix of the factor u we look for. Using the DBF we retrieve the occurrences of t in

$w[i-2^{k+1}..i-1]$. If they are a constant number only, we can check each of them to see if it leads to a valid extension uvu with uv primitive. So let us assume that, in fact, they form an arithmetic progression with ratio p and that x is the prefix of length p of t. Let i_0 be the position where the first t in that progression occurs. Then $w[i_0..i-1] = x^\alpha x'y'$, where x' is a prefix of x with $|x'| < |x|$, and $x^\alpha x'$ is a word of period p that cannot be further extended to the right without breaking the period, i. e., if $y' \neq \varepsilon$, then $y'[1] \neq x[|x'|+1]$. If $y' \neq \varepsilon$, we follow the same analysis as in case A.1 above. So let us consider next the case $y' = \varepsilon$.

Just like in case A.2 of the previous analysis, we want to identify the values $|t|/|x| \leq \gamma \leq \alpha$ such that $x^\gamma x'v$ is primitive. Once again, we actually determine (in constant time) the values γ such that $x^\gamma x'v$ is not primitive and output the others. For $\gamma \leq 5$ we proceed just as in the case A.2 above: check whether each such $x^\gamma x'v$ is primitive in constant time. If $\gamma \geq 6$, we try to see whether $x^\gamma x'v = z^s$ for some word z and exponent s. Now, as we only look for the case when $|u| \geq 4|v|$, we have that $x^\gamma x'$ should be at least as long as $4|v|$. Therefore, if $s \geq 4$, $x^\gamma x'$ has z^2 as a prefix. We can once again apply Theorem 4.7 and get that x should be equal to z. Now, $x^\gamma x'v = z^s$, with $s > 3$, if and only if $v = x''z^r$ for some r and $x'x'' = x$. Clearly, if this holds, then we have that $x^\gamma x'v$ is a power of z for all values of γ. If $s = 3$, then $|z|/2 \geq |x|$ and $x^\gamma x'$ and z^2 have a common prefix of length at least $3|z|/2$. Like before, we could apply Theorem 4.7 and get a contradiction to the primitivity of z. Finally, if $s = 2$ we proceed as follows. If $x^\gamma x'$ would be longer than $|z| + |x|$ we could once more apply Theorem 4.7 and get the contradiction. So, we look for some z which is longer than $x^{\gamma-1}x'$. This also means that $|v| + |x| > |z|$. So $|v| \geq |x|(\gamma - 2)$ and $|x|(\gamma+1) \geq |v|$. We get that $|v|/|x| + 2 \geq \gamma \geq |v|/|x| - 1$. We now check for which such values of γ is the word $x^\gamma x'v$ indeed a primitively rooted square. In the end we determined all values of γ for which $x^\gamma x'v$ is not primitive, and we can output the others.

This analysis takes constant time, and the time needed to output all γ such that $x^\gamma x'v$ is primitive takes time proportional to their number. We repeat this for all k with $\log(4|v|) \leq k \leq \lfloor \log(n) \rfloor$ and obtain all factors uvu with uv (and vu) primitive.

From the explanations above, it follows that the time needed to do this is upper bounded by the preprocessing time and the size of the output. So, our algorithm runs in $O(n \log n)$ time. □

Observe now that when we construct for a word w the sets P_i, as in Lemma 4.5, we can actually store in them the elements ordered according to their lengths (this is implemented at construction time). Accordingly, each P_i is an array where $P_i[k]$, for some k with $k \leq 2\log n$, stores the length of the k^{th} element of the ordered set P_i. Therefore, for the simplicity of the exposition, we will say that a suffix t of $w[1..i]$ is in P_i if, in fact, $|t| \in P_i$. Obviously, these two ways are equivalent, but our implementation of the sets P_i is more efficient, as we just store in them numbers that can be represented in one memory word (lengths), instead of actual strings.

Lemma 4.14. *If two occurrences of a primitively rooted square t^2, with $|t| = \ell$, end on positions i and j of some word, and $P_i[k] = \ell$, for some $k \leq |P_i|$, then $P_j[k] = \ell$.*

Proof. A word x^2 with x primitive and $|x| < \ell$ is a suffix of $w[1..i]$ if and only if it is a suffix of $w[i-\ell+1..i]^2$ if and only if it is a suffix of $w[1..j]$. □

In the following we show a result regarding the periodicity of instances of patterns.

Lemma 4.15. *Let t be a primitive word, let $\alpha \in (\Sigma \cup \{x\})^*$ be a one-variable pattern with $\alpha = \alpha' xvx$, where v is a prefix of t^ω, let $\ell = \lfloor \frac{|v|}{|t|} \rfloor$, and let $w \in \Sigma^+$ with $vw = t^{\ell+2}$. Moreover, for every $u \in \Sigma^+$, define the substitution $h_u : \{x\} \to \Sigma^+$ by $h(x) = u$. Then one of the following statements holds:*

(1) *if $h_w(\alpha)$ is $|t|$-periodic then $h_{wt^k}(\alpha)$ is $|t|$-periodic for every $k \geq 0$;*

(2) *if $h_w(\alpha)$ is not $|t|$-periodic then, for every word u ending with t, there exists at most one value $k \geq 0$ such that $h_{wt^k}(\alpha)$ is a suffix of u.*

Moreover, in the second case, the value k can be determined in $O(m)$ time, provided that we have LCP data structures for $w\alpha$, where m is the number of one-variable blocks occurring in α.

Proof. For the rest of the proof, assume $\alpha = v_p x v_{p-1} x \cdots v_1 x v x$. Moreover, for every i, $1 \leq i \leq p$, let $\alpha_i = v_i x v_{i-1} x \cdots v_1 x v x$ be the suffix of α starting with v_i and let $w_i = v_i w v_{i-1} w \cdots v_1 w v w$ be its image under h_w, that is $w_i = h_w(\alpha_i)$; furthermore, let $w_0 = vw$.

Case 1: $h_w(\alpha)$ is $|t|$-periodic.
As $vw = t^{\ell+2}$, we get that $v = t^{\ell} t'$ and $w = t'' t$ where $t't'' = t$. We shall show by induction, for every i, $1 \leq i \leq p-1$, that w_i is a power of t. As base of the induction, we note that, by assumption, w_0 is a power of t. Now let $i \geq 1$ and let us assume that w_{i-1} is a power of t. This means that $ww_{i-1} = t'' t^s$, for some $s \geq 2$. Now if $w_i = v_i w w_{i-1}$ is not a power of t, then v_i does not have the form $t^j t'$, for some j, which implies that the suffix t of w would be contained inside a t^2 factor (induced by the periodicity), a contradiction to the primitivity of t by Theorem 4.8. Thus, w_i is a power of t. In particular, this implies that, for all i with $1 \leq i \leq p-1$, $v_i w$ is a power of t and $v_p w$ is a suffix of $t^j t' t'' t$. Using this structural information on the words v_i and v, we can clearly see that also $h_{wt^k}(\alpha)$ is $|t|$-periodic, for every $k \geq 0$. Hence, Point 1 of the lemma holds.

Case 2: $h_w(\alpha)$ is not $|t|$-periodic.
Let u be some word ending in t with u_t being its longest suffix that is $|t|$-periodic. If $h_w(\alpha)$ is not $|t|$-periodic, then there either exists d such that w_d is not $|t|$-periodic and ww_{d-1} is $|t|$-periodic (†) or there exists d such that w_d is $|t|$-periodic and ww_d is not $|t|$-periodic (‡). If property (†) applies, then, by a similar argument as in the above case, $h_{wt^k}(\alpha_d)$ remains not $|t|$-periodic, while $h_{wt^k}(x\alpha_{d-1})$ is $|t|$-periodic, for all $k \geq 0$. In other words, when assigning to x a factor $t''t^k$, the period $|t|$ of $h_{wt^k}(\alpha)$, read from right to left, breaks inside v_d. Thus, there is only one possible candidate for k with the property that $h_{wt^k}(\alpha)$ is a suffix of u, namely the greatest integer for which $h_{wt^k}(x\alpha_{d-1})$ is at most as long as u_t and $h_{wt^k}(\alpha_d)$ is longer than u_t. That is, the points where the periods $|t|$ of $h_{wt^k}(\alpha)$ and u break must align. Analogously, if we have property (‡), then the period breaks this time inside a w, and a similar analysis can be performed. The only possible candidate for k is now the greatest integer for which $h_{wt^k}(\alpha_d)$ is at most as long u_t and $h_{wt^k}(x\alpha_d)$ is longer than u_t. Consequently, Point 2 of the lemma holds.

In order to find k efficiently, we can first find t' such that $v = v^{\ell} t'$ and check which is the smallest d such that v_d has not the form $t^j t'$; note that if x^2 is a factor of α then t' must be the empty word, so the factor v_d we look for will be exactly the rightmost v_i-factor which is not a power of t. Generally, finding v_d can be done in $O(m)$ time using *LCP* queries by just considering each of the v_i-factors and checking whether it starts with t, has period $|t|$, and $|v_i| - \lceil \frac{|v_i|}{|t|} \rceil |t| = |t'|$. Once we found this factor v_d, we know exactly which is the $|t|$-periodic suffix of $h_w(\alpha)$: it either starts inside v_d or in the w preceding it. In the first case, property (†) from above holds. In the second one, property (‡) holds. We can check in $O(1)$ time in which case we are. More precisely, if v_d is $|t|$-periodic and is a suffix of $t^j t'$ for some j, then (‡) holds, otherwise (†) holds. Let u_t be the suffix of u which is $|t|$-periodic. In the case when (†) holds, we compute k such that $h_{wt^k}(\alpha_d)$ is at most as long as $|u_t|$ and $h_{wt^k}(v_d\alpha_d)$ is longer than $|u_t|$, and this is our candidate k. In the case when (‡) holds, we compute k such that $h_{wt^k}(v_d\alpha_d)$ is at most as long as $|u_t|$ and $h_{wt^k}(\alpha_{d+1})$ is longer than $|u_t|$, and this is our candidate k. These computations can be done in $O(1)$ time, because they just require solving an inequality with the unknown k; for instance, in the first case, we compute the maximum k such that $|h_{wt^k}(v_d\alpha_d)| = |t||k|(d+1) + |v_d v_{d-1} \cdots v_1 v| \geq |u_t| \geq |t||k|(d+1) + |v_{d-1} \cdots v_1 v| = |h_{wt^k}(\alpha_d)|$. Once this k is found, we have determined exactly the suffix of u which is the image of x, and, as we have *LCP* structures for $u\alpha$, we can check from right to left whether $h_{wt^k}(\alpha)$ really is a suffix of u, in $O(m)$ time. This concludes the proof of our lemma. □

We are now ready to solve the matching problem for non-cross patterns. Notice that, as mentioned in Section 3, an $O(n^4)$ algorithm is presented in [39], which is automaton-based, but a dynamic programming algorithm, running in $O(mn^3)$ time for a word w of length n and a pattern of length m, is also straightforward. This can be achieved, e. g., by creating some data structures that allow us to compare factors of w in constant time (e.g., *LCP* data structures) and then by simply filling in the following table (notice that for non-cross patterns, when reading the pattern from left to right, there is always one *current variable*):

$$T[i,j,k,\ell] = \begin{cases} 1 & \text{if } \alpha[1..i] \text{ matches } w[1..j] \text{ such that} \\ & \text{the current variable is assigned } w[k..\ell], \\ 0 & \text{else.} \end{cases}$$

Next, we are considerably improving on these simple algorithms.

Theorem 4.16. *The matching problem for* Pat_{nc} *is solvable in* $O(mn\log n)$ *time, where* w *is the input word of length* n *and* m *is the number of one-variable blocks occurring in the pattern.*

Proof. Let $\alpha \in \text{Pat}_{\text{nc}}$ with $\text{var}(\alpha) = \{x_1, x_2, \ldots, x_s\}$ be our pattern. We construct the *LCP* data structure for the word $w\alpha$ and the sets P_i as in Lemma 4.5, as well as all the sets R_j^v for all positions j of w and v a maximal factor of terminals of α, as in Lemma 4.13. This preprocessing can be clearly implemented in $O(mn\log n)$.

Clearly, α can be factorised as $\alpha = w_0\Pi_{i=1,s}(\alpha_i w_i)$, where, for every i, $1 \le i \le s$, $\text{var}(\alpha_i) = \{x_i\}$, α_i starts and ends with x_i, and w_i is a maximal terminal factor. For every k, $0 \le k \le s-1$, we set $\beta_k = w_0\Pi_{i=1,k}(\alpha_i w_i)$. Let p_i denote the number of one-variable blocks from α_i.

We use a dynamic programming approach to test whether α matches w. More precisely, for each i with $1 \le i \le s$, we identify all j with $1 \le j \le n$, such that $\beta_{i-1}\alpha_i$ matches $w[1..j]$.

The general idea is the following. Once we identified all ways to match $\beta_{i-2}\alpha_{i-1}$ to a factor $w[1..j']$, we check, in constant time for each such j', whether w_{i-1} matches a string $w[j'+1..j'']$. After this step we identified all j'' such that β_{i-1} matches $w[1..j'']$. We then try to see, for all possible j, whether α_i matches a factor $w[j''+1..j]$ such that β_{i-1} matches $w[1..j'']$.

With a non-trivial quadratic time preprocessing [28], one could check this matching in constant time. This would lead to a matching algorithm for non-cross patterns running in $O(n^2m)$ time. However, a faster strategy exists and we present it in the following.

Based on the combinatorial toolbox we developed, we can first find some *basic assignments* for the variable x_i occurring in the pattern α_i. Suppose that we want to check whether $\beta_{i-1}\alpha_i$ matches $w[1..j]$. Assume, for instance, that α_i ends with $x_i v x_i$, for some maximal factor of terminals v. Here, we see, in the first case, how x_i can be assigned to a suffix of $w[1..j]$ such that vx_i is mapped to a primitive word (so that $\beta_{i-1}\alpha_i$ is mapped to $w[1..j]$), and, in the second case, for each primitively rooted square t^2 ending on position j, how x_i can be assigned to a suffix of $w[1..j]$ such that vx_i is mapped to a repetition of t of minimum exponent and $\beta_{i-1}\alpha_i$ is mapped to $w[1..j]$. There are $O(\log n)$ possibilities to be tried in these two cases, for each j. Then, by a second dynamic programming, we try to increase the exponent of t for some of the basic assignments of vx_i, i.e., *to extend* the respective assignments of vx_i, while still ensuring the match between $\beta_{i-1}\alpha_i$ and $w[1..j]$. In this way, we identify longer matches of α_i as suffix of $w[1..j]$ (while β_{i-1} matches shorter prefixes of $w[1..j]$), for all j, in overall time $O(p_i n\log n)$.

Before we explain the details of the algorithm, we state two claims, which explain all the ways α_i may be mapped to a suffix of $w[1..j]$, and essentially prove the correctness of our dynamic programming algorithm. The presentation is structured following a case distinction on the form of α_i.

In the following, we assume that for some i, $1 \leq i \leq s-1$, a substitution $h : \{x_1, x_2, \ldots, x_{i-1}\} \to \Sigma^*$ is already defined and j, $1 \leq j \leq n$, is arbitrarily chosen. Moreover, by h_z with $z \in \Sigma^+$, we denote the substitution h extended by $h(x_i) = z$.

Claim 1. Assume that

$$\alpha_i = x_i.$$

Then, we can find a word $v_i \in \Sigma^+$ such that $w[1..j] = h_{v_i}(\beta_{i-1}\alpha_i)$ if and only if one of the following holds:

i) $w[1..j-1] = h(\beta_{i-1})$ in this case, we set $v_i = w[j]$.

ii) $w[1..j-1] = h(\beta_{i-1})v'_i$ and, in this case, we set $v_i = v'_i w[j]$, with $v'_j \in \Sigma^+$.

The claim follows immediately. Note that, in the cases described by this claim, as $w[1..j] = h(\beta_{i-1})v_i$ and $\alpha_i = x$, we get trivially that $w[1..j] = h_{v_i}(\beta_{i-1}\alpha_i)$.

Claim 2. Assume that $|\alpha_i|_{x_i} \geq 2$, i.e.,

$$\alpha_i = \alpha'_i x_i v x_i, \text{ for some } v \in \Sigma^* \text{ and } \alpha'_i \in (\Sigma \cup \{x_i\})^*.$$

Then, we can find a word $v_i \in \Sigma^+$ such that $w[1..j] = h_{v_i}(\beta_{i-1}\alpha_i)$ if and only if one of the following holds, with j' denoting $j - |h_{v_i}(\alpha_i)|$:

i) $v_i \in R^v_j$ (so vv_i is primitive), $w[1..j'] = h(\beta_{i-1})$, and $h_{v_i}(\alpha_i) = w[j'+1..j]$.

ii) $vv_i = t^{\ell+k}$ for some word $t \in P_j$, where $\ell = \lfloor \frac{|v|}{|t|} \rfloor$ and k determined as follows:

a) $k = 1$, if $h_{v^{-1}t^{\ell+1}}(\alpha_i)$ is a suffix of $w[1..j]$, $w[1..j'] = h(\beta_{i-1})$, and $h_{v_i}(\alpha_i) = w[j'+1..j]$.

b) if $h_{v^{-1}t^{\ell+2}}(\alpha_i)$ is $|t|$-periodic, then k can be any integer greater than 1 such that $w[1..j'] = h(\beta_{i-1})$, $w[j'+1..j]$ is $|t|$-periodic, and $h_{v_i}(\alpha_i) = w[j'+1..j]$.

c) if $h_{v^{-1}t^{\ell+2}}(\alpha_i)$ is not $|t|$-periodic, then k is the only integer greater than 1 such that $h_{v^{-1}t^{\ell+k}}(\alpha_i)$ is a suffix of $w[1..j]$, $w[1..j'] = h(\beta_{i-1})$, and $h_{v_i}(\alpha_i) = w[j'+1..j]$.

The if-direction of the Claim is obvious. If, for some word v_i, we have $w[1..j] = h_{v_i}(\beta_{i-1}\alpha_i)$, then either $v_i \in R^v_j$, which implies that case *i*) holds, or $v_i \notin R^v_j$, which means that $vv_i = t^{\ell+k}$ for some word $t \in P_j$ and case *ii*) applies. Moreover, if $k = 1$, then we are in case *ii*)a), while for $k > 1$ Lemma 4.15 implies that either case *ii*)b) or *ii*)c) holds.

Now we describe our dynamic programming algorithm. Assume we identified the prefixes $w[1..j]$ of w that can be the image of $w_0\Pi_{k=1,i-2}(\alpha_k w_k)\alpha_{i-1}$. Initially, only the prefix of length 0 of w matches the prefix of length 0 of α. Using the *LCP* data structures built for $w\alpha$ we can test in $O(1)$ time for each j, such that $w[1..j]$ of w can be the image of $w_0\Pi_{k=1,i-2}(\alpha_k w_k)\alpha_{i-1}$, whether w_i occurs on position $j+1$. After this linear search, we get the prefixes $w[1..j]$ of w that can be the image of $w_0\Pi_{k=1,i-1}(\alpha_k w_k)$. Next, we try to see which of these prefixes can be extended with a factor matching α_i.

Following, our claims, we do a case analysis on the form of α_i. Recall that each part of our analysis has two phases: in the first one we find some basic assignments for x_i, the variable occurring in α_i, while in the second phase we try to extend these basic assignments to more complex ones. Also, we denote by $h_{v_i}(\alpha_i)$ the image of α_i under the substitution that replaces x_i by v_i.

If $\alpha_i = x_i$ then in the first phase we conclude that $w[1..j]$ can be the image of $w_0\Pi_{k=1,i-1}(\alpha_k w_k)\alpha_i$ if $w[1..j-1]$ can be the image of $w_0\Pi_{k=1,i-1}(\alpha_k w_k)$. So, in this case, the basic assignments of x_i consist in a single letter. This phase is based on Claim 1.*i*). In the second phase, we extend these assignments following Claim 1.*ii*). More precisely, for j from 1 to n, we decide that $w[1..j]$ can be the image of $w_0\Pi_{k=1,i-1}(\alpha_k w_k)\alpha_i$ if $w[1..j-1]$ can be the image of $w_0\Pi_{k=1,i-1}(\alpha_k w_k)\alpha_i$. That is, when matching $w[1..j-1]$ and $w_0\Pi_{k=1,i-1}(\alpha_k w_k)\alpha_i$ we assigned some factor v'_i to x_i, and now we append $w[j]$ to v'_i, and assign the value $v_i = v'_i w[j]$ to x_i, and obtain a matching between $w[1..j]$ and $w_0\Pi_{k=1,i-1}(\alpha_k w_k)\alpha_i$.

The analysis in the case above takes $O(n)$ time for each α_i that consists in a single variable.

If $\alpha_i = \alpha'_i xvx$ we follow Claim 2. In the first phase, we note that $w[1..j]$ can be the image of $w_0\Pi_{k=1,i-1}(\alpha_k w_k)\alpha_i$ if there exists $t \in R^v_j$ such that $w[1..j-|h_t(\alpha_i)|]$ can be the image of $w_0\Pi_{k=1,i-1}(\alpha_k w_k)$ and $h_t(\alpha_i)$ is a suffix of $w[1..j]$. This is one basic assignment that we will not try to extend. Similarly, and still in the first phase, we also look for assignments v_i of x_i that make vv_i a repetition. So, we consider $t \in P_j$ with $P_j[d] = t$, and analyse the case when vx is mapped to a repetition of t. We further take $\ell = \lfloor \frac{|v|}{|t|} \rfloor$ and we will have that v should be a prefix of $t^{\ell+1}$, as vx is assumed to be mapped to a power of t. So, assume v is indeed a prefix of $t^{\ell+1}$, of length at least $\ell|t|$ and let $v_i = v^{-1}t^{\ell+1}$. For each j, we can check in $O(p_i)$ time as in the proof of Lemma 4.15, if $h_{v_i}(\alpha_i)$ is a suffix of $w[1..j]$, and, if yes, whether $w[1..j-|h_{v_i}(\alpha_i)|]$ can be the image of $w_0\Pi_{k=1,i-1}(\alpha_k w_k)$. If also the last check returns true, we obtained a matching between $w[1..j]$ and $w_0\Pi_{k=1,i-1}(\alpha_k w_k)\alpha_i$. Again, this gives us a basic assignment for x_i that we will not extend in the following. More important, we take $v_i = v^{-1}t^{\ell+2}$ and check, again in $O(p_i)$ time as in Lemma 4.15, if $h_{v_i}(\alpha_i)$ is $|t|$-periodic. If not, we determine as in the respective lemma the unique value k' such that $h_{v'_i}(\alpha_i)$ is a suffix of $w[1..j]$, with $v'_i = v^{-1}t^{\ell+k'}$, and if this k' exists we check whether $w[1..j-|h_{v'_i}(\alpha_i)|]$ can be the image of $w_0\Pi_{k=1,i-1}(\alpha_k w_k)$; if the answer to this check is positive, we store this match. If $h_{v_i}(\alpha_i)$ is $|t|$-periodic, we check whether it is a suffix of $w[1..j]$, for each j, and, if yes, whether $w[1..j-|h_{v_i}(\alpha_i)|]$ can be the image of $w_0\Pi_{k=1,i-1}(\alpha_k w_k)$. If also the last check returns true, we obtained a matching between $w[1..j]$ and $w_0\Pi_{k=1,i-1}(\alpha_k w_k)\alpha_i$ and store in a matrix that there exists a match between $w[1..j]$ and $w_0\Pi_{k=1,i-1}(\alpha_k w_k)\alpha_i$ with vv_i a power of $P_j[d] = t$ (so, essentially, we store j, i, and d). This last assignment of x_i, and the matching it defines, will be extended in the second phase. The checks needed to implement this matching take $O(p_i \log n)$ time for each j from 1 to n.

Now, we move to the second phase, where we try to extend some of the assignments determined in the first step. Assume we want to check, for j from 1 to n, whether $w[1..j]$ can be the image of $w_0\Pi_{k=1,i-1}(\alpha_k w_k)\alpha_i$ with vx mapped to a power of some $t \in P_j$, with $P_j[d] = t$. Let $e = |\alpha_i|_{x_i}$. We check whether we concluded (in the first phase, or in previous iterations of this phase) that $w[1..j-e|t|]$ can be the image of $w_0\Pi_{k=1,i-1}(\alpha_k w_k)\alpha_i$, such that vx_i is mapped to a power of $P_{j-e|t|}[d]$, whether the image of α_i under this substitution is $|t|$-periodic, and whether $P_{j-e|t|}[d] = t$. If both checks are true, we conclude and store that $w[1..j]$ can be the image of $w_0\Pi_{k=1,i-1}(\alpha_k w_k)\alpha_i$ with vx mapped to a power of $P_j[d] = t$ (again, we store j, i, and d). This follows Claim 2.*ii*). That is, when matching $w[1..j-e|t|]$ and $w_0\Pi_{k=1,i-1}(\alpha_k w_k)\alpha_i$ we assigned some factor $v'_i = v^{-1}t^{\ell+k}$ to x_i, and now we append t to v'_i, and assign $v_i = v'_i t$ to x_i, and obtain a matching between $w[1..j]$ and $w_0\Pi_{k=1,i-1}(\alpha_k w_k)\alpha_i$. The time complexity of the processing we do in this second phase is $O(p_i \log n)$ for each j from 1 to n.

The two claims we made show how one can decide as described above whether $w[1..j]$ can be the image of β_i, knowing the prefixes of w which are the image of β_{i-1}. We do not describe this dynamic programming in detail (i.e., we do not give the code), as it is just a simple (yet, tedious) exercise.

We stress again that for an efficient implementation, one needs to construct the sets P_j, for all j with $1 \leq j \leq n$, and $R_j^{w_{p_i,i}}$, for all j with $1 \leq j \leq n$, and i such that $\alpha_i = x_i^{\ell_{0,i}}\Pi_{k=1,p_i}(w_{k,i}x_i^{\ell_{k,i}})$, as well as *LCP* data structures for $w\alpha$. But all these preprocessing steps can be done in less than $O(mn \log n)$ time.

To conclude, we can find, for all i from 1 to s, in $O(np_i \log n)$ time all the positions j such that $w_0\Pi_{k=1,i-1}(\alpha_k w_k)\alpha_i$ matches $w[1..j]$, where p_i is the number of one-variable blocks of α_i. Hence we have that α matches to w if there exists a position j such that $w_0\Pi_{k=1,s-1}(\alpha_k w_k)\alpha_\ell$ matches $w[1..j]$ and $w[j+1..n] = w_\ell$. The total time is, clearly, $O\left(\left(\sum_{i=1,s} p_i\right) n \log n\right)$. This shows that the matching problem for non-cross patterns can be solved in $O(mn \log n)$ time. □

4.3 Patterns with bounded scope coincidence degree

We now move on to the general case of patterns with bounded scope coincidence degree. The matching problem for $\text{PAT}_{\text{scd}\le k}$ can be still solved by a dynamic programming approach.

THEOREM 4.17. *The matching problem for* $\text{PAT}_{\text{scd}\le k}$ *is solvable in* $O\left(\frac{mn^{2k}}{((k-1)!)^2}\right)$ *time, where* w *is the input word of length* n *and* m *is the number of one-variable blocks occurring in the pattern.*

PROOF. Let $\alpha = w_0\Pi_{i=1,m}(y_i^{k_i} w_i)$ (where the y_i's are variables and the w_i's are maximal terminal factors) be a pattern with $\text{scd}(\alpha) \le k$; we want to decide whether α matches w. Note that we do not necessarily have $y_i \ne y_j$ for $i \ne j$.

By definition, a variable y is called active at a position ℓ of α if y occurs both in $\alpha[1..\ell]$ and in $\alpha[\ell..|\alpha|]$ (if y occurs on position ℓ, then it is active at position ℓ, even if this is its only occurrence); in other words, a variable y is active on each position contained in its scope, and only on those positions.

For every j with $1 \le j \le m$, we denote $\alpha_j = w_0\Pi_{i=1,j}(y_i^{k_i} w_i)$ and $\ell_j = |\alpha_j|$. For all j with $j \le m$, we produce a list of the active variables at position ℓ_j; this takes $O(|\alpha|m)$ time. We also build *LCP*-data structures for $w\alpha$. This allows us, for instance, to check whether a certain factor w_i occurs at a position ℓ of w in $O(1)$ time.

We define the $n \times m$ matrix $M[\cdot][\cdot]$ such that $M[i][j]$ contains a representation of all the possible substitutions for the active variables at position ℓ_j in substitutions mapping α_j to $w[1..i]$. We first explain how this representation is defined, and then show how this matrix is computed.

Firstly, let us note that if at most $k-1$ variables are active at position ℓ_j, then we store them by the starting and ending positions of their images, in the order of their occurrence in the pattern; in this way, the positions we store are also ordered, as the images of the variables also occur in the same order as the variables. Hence, we need to store a list of $2k-2$ ordered indices less than i; therefore, we may have to store in $M[i][j]$ at most $\binom{i}{k-1}^2$ different lists, each containing the indices corresponding to a substitution.

Secondly, if k variables are active at position ℓ_j, then one of the active variables is y_j. Moreover, for the image of y_j we do not need to store the ending position. Once we know the starting position of the image of the block $y_j^{k_j}$, say i', we can get its ending position by noting that the image of α_j is $w[1..i]$ and this image ends with the image of $y_j^{k_j} w_j$. So, the image of y_j occurs between i' and $i' + \frac{i-|w_j|-i'+1}{k_j} - 1$. Therefore, when exactly k variables are active at position ℓ_j we only need to store $2k-1$ indices: the starting and ending positions of the images of all the active variables except y_j, in order of their occurrence, and the starting position of the image of $y_j^{k_j}$. This means that $M[i][j]$ stores at most $\binom{i}{k-1}\binom{i}{k}$ distinct lists of $2k-1$ indices.

To summarise, when k variables are active at ℓ_j we need to store $2k-1$ indices in $M[i][j]$, and, thus, there are $\binom{i}{k}\binom{i}{k-1}$ possible lists that may be the actual list stored on this position of the matrix. When at most $k-1$ variables are active at ℓ_j, the lists stored in $M[i][j]$ contain at most $2k-2$ indices; so there exist at most $\binom{i}{k-1}^2$ lists that may appear as $M[i][j]$. This holds because an upper bound on the number of such lists is obtained by considering all the possibilities of choosing the numbers on the odd positions (starting positions) and the numbers on the even positions of the list (ending positions of the images of the variables), independently.

Next, we address the question of how to represent efficiently such collections of lists. Assume that we want to store a collections of lists, each containing p indices $i_1, i_2, \ldots, i_p$ between 1 and i, such that $i_{2h-1} \leq i_{2h}$, for $1 \leq h \leq \frac{p}{2}$, and $i_{2h} < i_{2h+1}$, for $1 \leq h \leq \frac{p-1}{2}$.

We construct a tree (called (i,p)-tree in the following) with $p+1$ levels $0, 1, \ldots, p$. On the level 0 we have the root labelled with 0. Under this root, we insert in the tree, one bye one, as paths, all the possible lists of exactly p indices between 1 and i that fulfil the condition above, in their natural lexicographical order on $\mathbb{N}^p$. When inserting a new list, we basically add to the tree a new path starting with the root and whose further nodes are labelled with the indices of the list, in increasing order. If needed, we add new nodes and edges to the tree: we first traverse the prefix of the list that already appears as a path in the tree, and, from the node we reached in this way, we insert a new path labelled with the rest of the indices in the list. For simplicity, we keep in a node an array of pointers to its children, in increasing order of their labels. More precisely, if a node is labelled with r and it is on an odd level, then its children will be labelled with $r, r+1, r+2, \ldots$; consequently, the j^{th} pointer in the array points to the child labelled with $r+j-1$. If a node is labelled with r and it is on an even level, then its children are labelled with $r+1, r+2, \ldots$; thus, the j^{th} pointer in the array points to the child labelled with $r+j$. Moreover, it is not hard to see that if a node is on the odd level $p-\ell$, then the maximum label of its children is $i - \lceil(\ell-1)/2\rceil$; if a node is on the even level $p-\ell$, then the maximum label of its children is $i - \lceil \ell/2 \rceil$. The time needed to construct this tree is upper bounded by the number of paths the tree contains. Thus, this upper bound is $O(\binom{i}{k}^2)$ if $p = 2k$ or $O(\binom{i}{k}\binom{i}{k-1})$ if $p = 2k-1$.

It is worth noting that there is a bijective correspondence between the leafs of an (i,p)-tree and the lists of exactly p indices that fulfil the condition stated before the definition of (i,p)-trees; basically, a leaf corresponds to the labels of the nodes along a path from the root to that leaf in the tree (without the root), and vice versa.

Clearly, to store a collection I of lists containing p indices between 1 and i like above, we can use such an (i,p)-tree where we just mark in it the leaf corresponding to each list in I (so, we have an (i,p)-tree with some marked leaves instead of the collection I). With this representation, we can test whether a list is in the collection I in $O(p)$ time (traverse the path whose labels correspond to the list and check if the leaf at the end of this path is marked), we can insert or delete a list in $O(p)$ time (traverse the path whose labels correspond to the list and mark or, respectively, unmark, the leaf at the end of this path). By keeping a linked list of the leaves, we can identify the marked leaves in time proportional to the number of leaves of an (i,p)-tree, so in $O(\binom{i}{p}^2)$ time. Given a leaf, we can retrieve the list that defines it by following the path from that leaf to the root, in $O(p)$ time (provided that we store also child to father links in our tree). For simplicity, the root of a tree is said to be also marked if at least one of the leaves is marked. As an example, see Figure 3.

So, coming back to our matrix, each $M[i][j]$ is initialised as an empty (i,s)-tree, where $s = 2p$ when there are $p \leq k-1$ active variables at position ℓ_j in α or $s = 2k-1$ when there are exactly k active variables at position ℓ_j in α. We just have to explain how $M[i][j]$ is computed efficiently.

Firstly, $M[i][1]$ is obtained in $O(n^3)$ as follows. For each integer i, we try all possibilities of choosing the image of y_1 as a factor of $w[|w_0|+1..i]$. Each of them is saved in the corresponding $(i,2)$ tree. The leaves and the root are marked accordingly.

Now, we move on to computing $M[i][j]$, assuming that we already computed $M[\cdot][j-1]$. Basically, for an i' with $i' \leq n$, looking at $M[i'][j-1]$ we retrieve the substitution for the variables that are active at position ℓ_{j-1}. Now, if y_j is one of them, we just have to check whether the image of

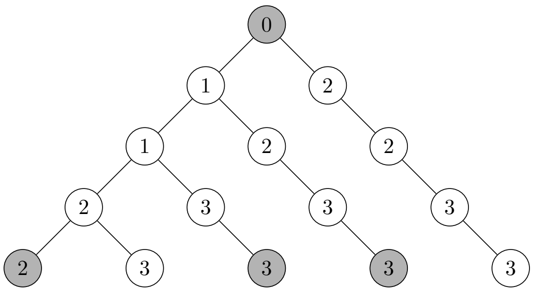

Fig. 3. A $(3, 4)$-tree storing the collection $I = \{(1, 1, 2, 2), (1, 1, 3, 3), (1, 2, 3, 3)\}$. The leaves corresponding to the paths $(0, 1, 1, 2, 2)$, $(0, 1, 1, 3, 3)$, and, respectively, $(0, 1, 2, 3, 3)$ are marked (represented by grey circles); the root of the respective tree is also marked, as it contains marked leaves.

$(y_j)^{k_j} w_j$ occurs at position $i' + 1$ (which is done in constant time by LCP-queries). If yes, and there are k active variables, then the positions storing the image of the last variable in our lists may have changed; hence, we update the list in $O(k)$ (that is, delete the starting and, unless $y_j = y_{j-1}$, ending position that denoted the image of y_j, and then append to the list the ending position of y_{j-1} – as its starting position was already in the list – as well as i' as the starting position of $(y_j)^{k_j} w_j$, as explained before). If the list contained less than k variables, we just leave it as it is. In both cases, the new list is inserted in the tree $M[i][j]$, where i is the ending position of the image of $(y_j)^{k_j} w_j$.

We just have to deal with the case when y_j was not one of the active variables. In this case, it is a new variable, that becomes active now. As a first step, we assume that the image of y_j is $w[i' + 1]$ and save accordingly the images of the active variables in a tree $C[i' + 1][j]$ (these trees are auxiliary, and are empty before $M[\cdot][j]$ are computed); basically, at this step we did not check if w_j can follow the image of y_j, nor did we find all possible images for y_j, but only those where y_j is mapped to a single letter. After we finish this process for all values of i', we continue. Now, for each index $i'' < n$, considered in increasing order, if $C[i''][j]$ is non-empty, then we insert all of its images into $C[i'' + 1][j]$, just saving (if there are less than k active variables at position ℓ_j) a new ending position for y_j as $i'' + 1$. In this phase, we just represented the cases where y_j is mapped to more than one letter. The process is simple: due to the order in which we consider the indices i'', when computing $C[i'' + 1][j]$ we will have in $C[i''][j]$ all the possibilities of mapping y_j to a string that ends on position i''; then we just extend them with a letter. As said, when we are done with this, $C[i''][j]$ contains all ways of assigning values to the variables active at position ℓ_j such that $\alpha_{j-1} y_j$ is mapped to $w[1..i'']$. Then, for each i'' with $i'' \leq n$ and each list in $C[i''][j]$ we check whether $y_j^{k_j - 1} w_j$ is a prefix of $w[i''..n]$, and, if yes, insert the list in the tree $M[i][j]$, where i is the ending position of $y_j^{k_j - 1} w_j$. The usage of the trees $C[\cdot][j]$ is justified as they store first images for the variables $y_1, y_2, \ldots, y_j$ that map only $w_0 \Pi_{i=1,j-1}(y_i^{k_i} w_i) y_j$ to prefixes of w, then we look for alignments of $w_0 \Pi_{i=1,j-1}(y_i^{k_i} w_i) y_j^{k_j}$ with prefixes of w, and finally we store in M the correct images of the variables that map α_j to prefixes of w.

The total time needed to construct the trees stored in the matrix M is clearly upper bounded by $O(kmn\binom{n}{k-1}\binom{n}{k}) = O(\frac{mn^{2k}}{((k-1)!)^2})$.

We conclude that α matches w if and only if $M[n][m]$ is not empty. □

5 THE INJECTIVE VARIANT OF THE MATCHING PROBLEM

So far, we presented a series of upper bounds for the time needed to solve Match for various restricted classes of patterns. In this section, we take a closer look at inj-Match, i. e., the injective variant of the matching problem for patterns. In particular, we investigate the question whether our (efficient) algorithms for Match can be adapted to inj-Match. In this regard, we first note the following result:

Proposition 5.1. inj-*Match can be solved in time* $O(\frac{mn^{k-1}}{(k-1)!})$*, where n is the length of the word, m the length of the pattern and k its number of distinct variables.*

Proof. Our algorithm extends the ideas of [22]. We denote by w our input word of length n and by α the given pattern.

Let us assume that the variables occurring in α are $x_1, x_2, \ldots, x_k$ such that the first occurrence of the variable x_i in α is to the left of the first occurrence of x_j, for every $1 \le i < j \le k$. Let α_i be the prefix of α ending with the first occurrence of x_i. In the first phase of our algorithm we will try all possible lengths for the images of $\alpha_1, \alpha_2, \ldots, \alpha_{k-1}$ in an assignment of the variables. As $|\alpha_i| < |\alpha_{i+1}|$ for all $1 \le i \le k-2$, we have $\binom{n}{k-1} \le \frac{n^{k-1}}{(k-1)!}$ possible combinations for these lengths.

Let us now consider such a choice $(\ell_1, \ell_2, \ldots, \ell_{k-1})$, where ℓ_i is the length of the image of α_i, for $1 \le i \le k-1$. It is immediate that we can compute from ℓ_1 in $O(|\alpha_1|)$ the actual factor of w to which x_1 is mapped. Then, knowing the image of x_1, we can compute from ℓ_2 in $O(|\alpha_2|)$ the actual factor of w to which x_2 is mapped, because α_2 contains only the variables x_1 and x_2, and some constants. Generally, knowing the images of $x_1, x_2, \ldots, x_{i-1}$, we can compute from ℓ_i in $O(|\alpha_i|)$ the actual factor of w to which x_i is mapped, because α_i contains only the variables $x_1, x_2, \ldots, x_{i-1}$ occurring multiple time, one occurrence of x_i, and some constant factors. Finally, after knowing the images of $x_1, x_2, \ldots, x_{k-1}$ we get in $O(m)$ the image of x_k. The total time needed to compute these images is $O(m + \sum_{i=1,k-1} |\alpha_i|) = O(m)$.

In the second phase of our algorithm (the injectivity check), we consider separately two cases: $k \le 3$ and $k > 3$. In the first case, we can check in $O(1)$ using *LCP*-data structures whether the images of the k variables are distinct. In the second case, we can build before the first phase, in a preprocessing step, a data structure that associates to each factor $w[i..j]$ of w the position i' of its first occurrence in w (which may be to the left of i). We also initialise an $n \times n$ matrix of $A[i'][\ell]$, initially with all elements 0. Now, after executing the first phase of the computation, described above, for i from 1 to k we increment the position $A[s][\ell]$ of the matrix A, where s is the first occurrence of the factor of w to which x_i is mapped, and ℓ is its length. This takes $O(m)$. The assignment of the variables is injective if and only if all the incremented positions have in the end value 1. After this, we set exactly those positions that were considered in this phase back to 0 (this takes $O(m)$ time, again).

Our algorithm is easily checked to be correct. In the case of $k \le 3$ it runs in $O(\frac{mn^{k-1}}{(k-1)!})$, while for $k > 3$ it runs in $O(\frac{mn^{k-1}}{(k-1)!} + n^2) = O(\frac{mn^{k-1}}{(k-1)!})$, which concludes our proof. □

The question arises whether it is possible to efficiently solve inj-Match for the restricted classes of patterns (i. e., regular and non-cross patterns as well as patterns with a bounded scope coincidence degree) for which efficient algorithms for Match exist. In this regard, we can prove the rather strong negative result that inj-Match is NP-complete for any class of patterns that contains the trivial class $\{x_1 x_2 \ldots x_n \mid n \ge 1\}$ of patterns (note that this is the case for the above mentioned classes).

In the non-injective case, for a fixed pattern $\alpha = x_1 x_2 \ldots x_n$, the set of words matching α has a trivial structure, since it is the set of all words of length at least n. For the injective variant, on the other hand, this set contains exactly the words that have a unique factorisation with n factors; thus, solving the injective matching problem for such trivial patterns is equivalent to the following decision problem:

Unique Factorisation (UF)
Instance: A word w and an integer k with $1 \leq k \leq |w|$.
Question: Does there exist a unique factorisation of w with size of at least k?

The problem of computing unique factorisations (also called *equality-free* factorisations) has been investigated in the literature in different contexts. In Condon et al. [8, 9], the problem of computing unique factorisations with factors of bounded length has been investigated, which is motivated by self-assembly of DNA sequences, while the paper [38] provides a more general (and also parameterised) complexity analysis. The hardness of computing a unique factorisation with only palindromes as factors is shown by Banai et al. [4].

We shall now prove that the problem UF is in fact NP-complete, which directly yields the above mentioned hardness results for the matching problem. We establish the NP-hardness of UF by a reduction from the following well-known problem:

3-Dimensional Matching (3D-Match)
Instance: An integer $\ell \geq 1$ and a set $S \subseteq \{(p, q, r) \mid 1 \leq p \leq \ell < q \leq 2\ell < r \leq 3\ell\}$.
Question: Does there exist $S' \subseteq S$ with $|S'| = \ell$ such that, for all $(p, q, r), (p', q', r') \in S'$, $(p, q, r) \neq (p', q', r')$ implies $p \neq p'$, $q \neq q'$ and $r \neq r'$?

An instance of 3D-Match is a set S of triples, the three components of which carry values from $\{1, 2, \ldots, \ell\}$, $\{\ell + 1, \ell + 2, \ldots, 2\ell\}$ and $\{2\ell + 1, 2\ell + 2, \ldots, 3\ell\}$, respectively. A *solution* for (S, ℓ) is a selection of ℓ triples such that no two of them coincide in any component. Hence, for every $i \in \{1, 2, 3\}$, if we collect all the ith components of the ℓ triples of a solution, then we get exactly the set $\{(i - 1)\ell + 1, (i - 1)\ell + 2, \ldots, i\ell\}$. For the NP-completeness of 3D-Match see [20].

We define a mapping g from 3D-Match to UF. Let (S, ℓ) be an instance of 3D-Match, where $\ell \geq 1$ and $S = \{s_1, s_2, \ldots, s_k\}$ with $s_i = (p_i, q_i, r_i)$, for $1 \leq i \leq k$. The input word $g(S, \ell)$ for UF shall be a word over the alphabet

$$\Sigma = \{\mathsf{a}, ¢_i, \$_i, \mathsf{b}_{i,j}, \%_{i,j}, l, \#_l, \#_0 \mid 1 \leq i \leq k, 1 \leq j \leq 4, 1 \leq l \leq 3\ell\},$$

and is defined as follows. Let $v = v_1 v_2 \cdots v_k$, where, for every integer i with $1 \leq i \leq k$,

$$v_i = ¢_i\, p_i\, \mathsf{a}\, \mathsf{b}_{i,1}\, \mathsf{b}_{i,2}\, q_i\, \mathsf{a}\, \mathsf{b}_{i,3}\, \mathsf{b}_{i,4}\, r_i\, \mathsf{a}\, \$_i.$$

Let $\widehat{u} = 1\,\#_1 \cdots \#_{3\ell-2}\,(3\ell - 1)\,\#_{3\ell-1}\,(3\ell)\,\#_{3\ell}$ and $\overline{u} = \overline{u}_1 \overline{u}_2 \cdots \overline{u}_k$, where, for every integer i with $1 \leq i \leq k$,

$$\overline{u}_i = \mathsf{b}_{i,1}\, \%_{i,1}\, \mathsf{b}_{i,2}\, \%_{i,2}\, \mathsf{b}_{i,3}\, \%_{i,3}\, \mathsf{b}_{i,4}\, \%_{i,4}\,.$$

Finally, $u = \mathsf{a}\,\#_0\,\widehat{u}\,\overline{u}$, $w = uv$, $\widehat{\ell} = 7\ell + 6(k - \ell) + |u|$ and $g(S, \ell) = (w, \widehat{\ell})$. This concludes the definition of the mapping g. In the following, let (S, ℓ) be a fixed instance of 3D-Match and $(w, \widehat{\ell}) = g(S, \ell)$.

We now explain the mapping g in an intuitive way. The actual information of the 3D-Match instance is exclusively encoded in the suffix v of w, whereas the purpose of the prefix u is to enforce a certain structure of unique factorisations (note that u is just a list that contains each symbol from $\Sigma \setminus \{¢_i, \$_i, \mid 1 \leq i \leq k\}$ exactly once). Every l with $1 \leq l \leq 3\ell$, i. e., the elements of the triples, is used as an individual symbol; thus, every triple $s_i = (p_i, q_i, r_i)$ of S can be represented by $v_i = ¢_i\, p_i\, \mathsf{a}\, \mathsf{b}_{i,1}\, \mathsf{b}_{i,2}\, q_i\, \mathsf{a}\, \mathsf{b}_{i,3}\, \mathsf{b}_{i,4}\, r_i\, \mathsf{a}\, \$_i$, where the factors $p_i\mathsf{a}$, $q_i\mathsf{a}$ and $r_i\mathsf{a}$ represent the single components. Each of the remaining symbols $¢_i$, $\$_i$, $\mathsf{b}_{i,j}$, for $1 \leq j \leq 4$, has exactly one occurrence in w; thus, every factor that contains one of these will necessarily be distinct. Hence, the factors

$p_i\mathtt{a}$, $q_i\mathtt{a}$ and $r_i\mathtt{a}$ are the only ones that may coincide in v_i and some v_j, for $i \neq j$, and this is only the case if the triples s_i and s_j contain common elements.

We now define two special factorisations of the factors v_i, where $1 \leq i \leq k$. The factorisation $¢_i p_i \mid \mathtt{ab}_{i,1} \mid \mathtt{b}_{i,2} q_i \mid \mathtt{ab}_{i,3} \mid \mathtt{b}_{i,4} r_i \mid \mathtt{a}\$_i$ is called *safe* and the factorisation $¢_i \mid p_i\mathtt{a} \mid \mathtt{b}_{i,1}\mathtt{b}_{i,2} \mid q_i\mathtt{a} \mid \mathtt{b}_{i,3}\mathtt{b}_{i,4} \mid r_i\mathtt{a} \mid \$_i$ is called *unsafe*. The safe factorisation contains only distinct factors, whereas the factors $p_i\mathtt{a}$, $q_i\mathtt{a}$ and $r_i\mathtt{a}$ of the unsafe factorisation may also occur in the unsafe factorisation of some v_j, where $1 \leq j \leq k$ and $i \neq j$; thus, the situation that s_i and s_j have common elements translates into the situation that the unsafe factorisations of v_i and v_j have common factors.

If $\{s_{t_1}, s_{t_2}, \dots, s_{t_\ell}\}$ is a solution of (S, ℓ), then we can factorise all v_{t_i}, for $1 \leq i \leq \ell$, into the unsafe factorisation, all other v_j, where $j \notin \{t_1, t_2, \dots, t_\ell\}$, into the safe factorisation, and the prefix u into $|u|$ individual factors. This yields a factorisation of w with $|u| + 7\ell + 6(k - \ell) = \widehat{\ell}$ factors and its uniqueness follows from the fact that $\{s_{t_1}, s_{t_2}, \dots, s_{t_\ell}\}$ is a solution of (S, ℓ) and that the symbols from u do not occur as single factors in v.

Lemma 5.2. *If (S, ℓ) has a solution, then there is a unique factorisation of w of size at least $\widehat{\ell}$.*

Proof. We assume that $\{s_{t_1}, s_{t_2}, \dots, s_{t_\ell}\}$ is a solution of (S, ℓ). We define a factorisation f of w in the following way. Every single symbol of u is an f-factor and, for every integer i, with $1 \leq i \leq k$ and $i \in \{t_1, t_2, \dots, t_\ell\}$, we factorise v_i into the unsafe factorisation

$$\underbrace{¢_i}\ \underbrace{p_i\mathtt{a}}\ \underbrace{\mathtt{b}_{i,1}\mathtt{b}_{i,2}}\ \underbrace{q_i\mathtt{a}}\ \underbrace{\mathtt{b}_{i,3}\mathtt{b}_{i,4}}\ \underbrace{r_i\mathtt{a}}\ \underbrace{\$_i}\,,$$

and, for every integer i, with $1 \leq i \leq k$ and $i \notin \{t_1, t_2, \dots, t_\ell\}$, we factorise v_i into the safe factorisation

$$\underbrace{¢_i p_i}\ \underbrace{\mathtt{ab}_{i,1}}\ \underbrace{\mathtt{b}_{i,2} q_i}\ \underbrace{\mathtt{ab}_{i,3}}\ \underbrace{\mathtt{b}_{i,4} r_i}\ \underbrace{\mathtt{a}\$_i}\,.$$

The factorisation f has exactly $|u| + 7\ell + 6(k - \ell) = \widehat{\ell}$ factors and it only remains to show that it is unique.

In u, every symbol has exactly one occurrence, which means that if one of the factors $f(i)$, where $1 \leq i \leq |u|$, is repeated, then it must occur as a factor of v. However, the only f-factors of v of length 1 are the factors c_i and $\$_i$, for every $i \in \{t_1, t_2, \dots, t_\ell\}$, which do not occur in u. Hence, all f-factors of length 1 are distinct factors. Furthermore, the f-factors $\mathtt{b}_{i,1}\mathtt{b}_{i,2}$ and $\mathtt{b}_{i,3}\mathtt{b}_{i,4}$, for every $i \in \{t_1, t_2, \dots, t_\ell\}$, as well as the f-factors $¢_i p_i$, $\mathtt{ab}_{i,1}$, $\mathtt{b}_{i,2} q_i$, $\mathtt{ab}_{i,3}$, $\mathtt{b}_{i,4} r_i$ and $\mathtt{a}\$_i$, for every $i \notin \{t_1, t_2, \dots, t_\ell\}$, all contain a symbol from $\{¢_i, \$_i, \mathtt{b}_{i,1}, \mathtt{b}_{i,2}, \mathtt{b}_{i,3}, \mathtt{b}_{i,4}\}$ and therefore are distinct factors. The only remaining f-factors are $p_i\mathtt{a}$, $q_i\mathtt{a}$ and $r_i\mathtt{a}$, for every $i \in \{t_1, t_2, \dots, t_\ell\}$. If there exists an integer $i \in \{t_1, t_2, \dots, t_\ell\}$, such that $p_i\mathtt{a}$ is a repeated f-factor, then there must exist an integer $i' \in \{t_1, t_2, \dots, t_\ell\}$ with $i \neq i'$ and $p_i\mathtt{a} = p_{i'}\mathtt{a}$ (note that $p_i\mathtt{a} = q_{i'}\mathtt{a}$ or $p_i\mathtt{a} = r_{i'}\mathtt{a}$ is impossible, since $1 \leq p_i \leq \ell$, $\ell + 1 \leq q_{i'} \leq 2\ell$ and $2\ell + 1 \leq r_{i'} \leq 3\ell$), which means $p_i = p_{i'}$. This implies that s_i and $s_{i'}$ coincide in a component, which is a contradiction to the fact that $\{s_{t_1}, s_{t_2}, \dots, s_{t_\ell}\}$ is a solution of (S, ℓ). Hence, all f-factors $p_i\mathtt{a}$, with $i \in \{t_1, t_2, \dots, t_\ell\}$, are distinct factors and in exactly the same way we can show that all f-factors $q_{t_i}\mathtt{a}$ and $r_{t_i}\mathtt{a}$, where $1 \leq i \leq \ell$, are distinct factors, as well. Therefore the factorisation f defined above is unique, which concludes the proof. □

Proving the converse of Lemma 5.2 is more difficult and here the prefix u of w will be crucial. The idea is to first show that if there exists a unique factorisation of w of size $\widehat{\ell}$, then there also exists one with at least the same size such that

(1) no factor overlaps[2] the boundaries between u and v, or between some v_i and v_{i+1}, where $1 \leq i \leq k - 1$, and

[2]A factor $f(i)$, where $1 \leq i \leq k$, *overlaps positions j and $j + 1$*, where $1 \leq j \leq |w| - 1$, if both of these positions belong to the factor $f(i)$, i. e., $|f(1)f(2)\cdots f(i-1)| < j < |f(1)f(2)\cdots f(i-1)f(i)|$.

(2) u is split into $|u|$ factors.

Property (1) can be achieved by simply splitting the factors that may overlap the critical positions; this does only increase the number of factors and the uniqueness of the factorisations is guaranteed by the fact that the new factors must contain symbols with only one occurrence in w.

Lemma 5.3. *If w has a unique factorisation f of size at least $\widehat{\ell}$, then, for some $\widehat{\ell'}$ with $\widehat{\ell'} \geq \widehat{\ell}$, there exists a unique factorisation f' of w of size at least $\widehat{\ell'}$, such that no f'-factor overlaps positions $|u|$ and $|u|+1$ or positions $|uv_1v_2 \cdots v_i|$ and $|uv_1v_2 \cdots v_i| + 1$, where $1 \leq i \leq k-1$.*

Proof. Let f be a unique factorisation of w. If, for some p, where $1 \leq p \leq \widehat{\ell}$, the factor $f(p)$ overlaps positions $|u|$ and $|u|+1$, then $f(p) = \pi_1 \, \%_{k,4} \, ¢_1\pi_2$ for some (possibly empty) factors π_1 and π_2. We can now transform f into a factorisation f' by splitting the factor $f(p) = \pi_1 \, \%_{k,4} \, ¢_1\pi_2$ into the factors $\pi_1 \, \%_{k,4}$ and $¢_1\pi_2$, respectively. Since the symbols $\%_{k,4}$ and $¢_1$ have only one occurrence in w, we can conclude that f' is a unique factorisation of size $(\widehat{\ell}+1)$. Since every v_i starts with $¢_i$ and ends with $\$_i$, we can apply exactly the same construction in order to resolve all possible overlaps of positions $|uv_1v_2 \cdots v_i|$ and $|uv_1v_2 \cdots v_i| + 1$, for every i such that $1 \leq i \leq k-1$. These constructions do not decrease the number of factors; thus, the statement of the lemma follows. □

Property (2) requires a more careful argument. If u is not split into $|u|$ factors, then in u there exists a factor π of size at least 2. If we simply cut off the first symbol of π, then we can only increase the number of factors, but it may happen that one of the new factors is now repeated, destroying the uniqueness of the factorisation. However, it turns out that at most one of these two factors can be repeated, and if one of these factors is repeated, then it has size 1 and must have a second occurrence in v. We can then show that this repeated factor necessarily has a neighbouring factor in v to which it can be joined without violating the distinctness of this factor.

Lemma 5.4. *If w has a unique factorisation f of size at least $\widehat{\ell}$, then, for some $\widehat{\ell'}$ with $\widehat{\ell'} \geq \widehat{\ell}$, w has a unique factorisation f' of size at least $\widehat{\ell'}$, such that every single symbol of u is an f'-factor.*

Proof. First, by applying Lemma 5.3, we transform f into a unique factorisation f'' of size $\overline{\ell}$ that satisfies the properties stated in Lemma 5.3. If f'' splits u into $|u|$ factors of size 1, then we are done. So we assume that this is not the case and we show how f'' can be transformed into a unique factorisation f', such that every single symbol of u is an f'-factor.

Since f'' does not split u into $|u|$ factors (and since no f''-factor overlaps positions $|u|$ and $|u|+1$), there exists an integer p, where $1 \leq p \leq |u|-1$, such that $|f''(p)| \geq 2$ and $|f''(1)f''(2)\cdots f''(p)| \leq |u|$. We define $f''(p) = x\pi$, where x is a single symbol and π is some non-empty factor. We shall first consider the case when $|\pi| \geq 2$ (i. e., $|f''(p)| > 2$).

We modify f'' by splitting the factor $f''(p)$ into the two factors x and π, which increases the number of total factors by 1. It is not possible that the factor π is repeated (with respect to this modified factorisation), since $|\pi| \geq 2$ and every factor of u of length at least 2 contains a symbol $\#_i$, for $0 \leq i \leq 3\ell+1$, or $\%_{i,j}$, for $1 \leq i \leq k$ and $1 \leq j \leq 4$, which have only a single occurrence in w. It is possible, however, that the factor x is repeated, i. e., there is an occurrence of x in some v_i, for $1 \leq i \leq k$, which is an f''-factor (note that x is not repeated in u). This particularly implies that $x \in \{\mathsf{a}, p_i, q_i, r_i, \mathsf{b}_{i,1}, \mathsf{b}_{i,2}, \mathsf{b}_{i,3}, \mathsf{b}_{i,4}\}$. We note that, by the structure of v_i, there is a symbol $y \in \{\mathsf{b}_{i,1}, \mathsf{b}_{i,2}, \mathsf{b}_{i,3}, \mathsf{b}_{i,4}, ¢_i, \$_i\}$, such that the factor x is the left neighbour of an f''-factor $y\delta$ or the right neighbour of an f''-factor δy. Since these two cases can be dealt with analogously, we only consider the case that x is the left neighbour of a factor $y\delta$. We now modify the factorisation by appending x to the factor $y\delta$ and we claim that the new factor $xy\delta$ is unique (with respect to the modified factorisation). This is obvious if $y \in \{¢_i, \$_i\}$, since these symbols have only one occurrence in w. If $y \notin \{¢_i, \$_i\}$, then $y \in \{\mathsf{b}_{i,1}, \mathsf{b}_{i,2}, \mathsf{b}_{i,3}, \mathsf{b}_{i,4}\}$ and, since y has only one occurrence in v, this implies that if $xy\delta$ is repeated, then it is repeated only in u. This is a contradiction, since $|xy\delta| \geq 2$ and every factor of u of size at least 2 contains a symbol $\#_i$, for $0 \leq i \leq 3\ell + 1$, or $\%_{i,j}$, for $1 \leq i \leq k$ and $1 \leq j \leq 4$. Consequently, $xy\delta$ is a distinct factor and the modified factorisation has exactly the same number of factors as f''.

Next, we consider the case when $|\pi| = 1$ (i. e., $|f''(p)| = 2$). Again, we split the factor $f''(p)$ into two factors x and π, which increases the number of total factors by 1. Furthermore, since either x or π is a symbol $\#_i$, for $0 \leq i \leq 3\ell + 1$, or $\%_{i,j}$, for $1 \leq i \leq k$ and $1 \leq j \leq 4$, we can conclude that at most one of the two factors x and π can be repeated. Hence, we can join this repeated factor with a neighbour as before, which results again in a unique factorisation with the same number of factors.

By repeating the construction described above, we can change the factorisation f'' into the unique factorisation f' of size $\widehat{\ell'}$ with $\widehat{\ell'} \geq \overline{\ell} \geq \widehat{\ell}$, such that every single symbol of u is an f'-factor. □

Now the structure of the factorisation ensured by Lemmas 5.3 and 5.4 allows us to argue that the only factors of size 1 in v are of the form $¢_i$ or $\$_i$, since otherwise they would be repeated in u. Consequently, every v_i is split in at most 7 factors and if so, the factorisation of v_i must be the unsafe factorisation. In order to get at least $\widehat{\ell}$ factors, at least ℓ of the v_i must be factorised into the unsafe factorisation, which corresponds to a solution of the 3D-Match instance.

Lemma 5.5. *If there exists a unique factorisation of w of size at least $\widehat{\ell}$, then (S, ℓ) has a solution.*

Proof. Let f be a unique factorisation of w of size $\widehat{\ell}$. By Lemmas 5.3 and 5.4, there exists a unique factorisation f' of w of size $\widehat{\ell'}$, for some $\widehat{\ell'} \geq \widehat{\ell}$, such that no f'-factor overlaps positions $|u|$ and $|u| + 1$, or positions $|uv_1v_2 \cdots v_i|$ and $|uv_1v_2 \cdots v_i| + 1$, where $1 \leq i \leq k - 1$, and every single symbol of u is an f'-factor.

If a single symbol x of some v_i, where $1 \leq i \leq k$, is an f'-factor, then $x \in \{¢_i, \$_i\}$. This is due to the fact that if $x \notin \{¢_i, \$_i\}$, then x has an occurrence in u and since every single symbol of u is an f'-factor, this means that x is a repeated f'-factor. This directly implies that, for every i with $1 \leq i \leq k$, it is not possible that f' splits v_i in more than 7 factors and, furthermore, if f' splits v_i in 7 factors, then this is done in the following way:

$$\underbrace{¢_i}\ \underbrace{p_i\mathsf{a}}\ \underbrace{\mathsf{b}_{i,1}\mathsf{b}_{i,2}}\ \underbrace{q_i\mathsf{a}}\ \underbrace{\mathsf{b}_{i,3}\mathsf{b}_{i,4}}\ \underbrace{r_i\mathsf{a}}\ \underbrace{\$_i}\ .$$

We assume that f' splits exactly m of the v_i, where $1 \leq i \leq k$, into 7 factors and the remaining $k - m$ of the v_i, where $1 \leq i \leq k$, into 6 or less factors. Hence, f' splits w into at most $|u| + 7m + 6(k - m)$ factors. If $m < \ell$, then

$$|u| + 7m + 6(k - m) < |u| + 7\ell + 6(k - \ell) = \widehat{\ell},$$

which is a contradiction, since f' is a factorisation of size $\widehat{\ell'}$ and $\widehat{\ell'} \geq \widehat{\ell}$. Thus, $\ell \leq m$. Now let $t_1, t_2, \ldots, t_m$ with $1 \leq t_i \leq k$ for $1 \leq i \leq m$, be such that f' splits every v_{t_i} with $1 \leq i \leq m$ in 7 factors. As explained above, this implies that $p_{t_i}\mathsf{a}$, $q_{t_i}\mathsf{a}$ and $r_{t_i}\mathsf{a}$ are f'-factors, for every i with $1 \leq i \leq m$. Consequently, for every i and j, where $1 \leq i < j \leq m$, we have $p_{t_i}\mathsf{a} \neq p_{t_j}\mathsf{a}$, $q_{t_i}\mathsf{a} \neq q_{t_j}\mathsf{a}$ and $r_{t_i}\mathsf{a} \neq r_{t_j}\mathsf{a}$, which means that $(s_{t_1}, s_{t_2}, \ldots, s_{t_\ell})$ is a solution for (S, ℓ). □

In order to conclude the NP-completeness of UF, it only remains to observe that the problem is clearly in NP (we can check the uniqueness of a guessed factorisation in polynomial time) and that the reduction defined above can be carried out in polynomial-time.

Theorem 5.6. *UF is* NP*-complete.*

Since a word w matches the pattern $x_1 x_2 \dots x_n$ if and only if w has a unique factorisation of size *exactly* n, but $(w, n) \in$ UF if and only if w has a unique factorisation of size *at least* n, it is important to note that if a word has a unique factorisation of size k, then it also has a unique factorisation of size k', for all k' with $1 \leq k' \leq k$. This is due to the fact that the uniqueness of a factorisation is preserved if we join a longest factor with one of its neighbours. Together with Theorem 5.6, this implies the following negative result with respect to the injective variant of the matching problem for patterns.

Corollary 5.7. *inj-Match is* NP*-complete for* $\mathit{PAT}_{\text{reg}}$, PAT_{nc}, $\mathit{PAT}_{\text{rvar}\leq k}$, *and* $\mathit{PAT}_{\text{scd}\leq k}$, *for every* k *with* $k \geq 1$.

A downside of our reduction to prove Theorem 5.6 is that it requires an unbounded alphabet and it is open whether UF is NP-complete for fixed alphabets.[3] Consequently, it is not implied that the injective matching problem for the classes of patterns mentioned in Corollary 5.7 is still NP-complete if the alphabet is fixed. However, for fixed alphabets of size 5, we can at least show that the injective matching problem remains NP-complete for the class of non-cross patterns and hence for patterns with a bounded scope coincidence degree.

To this end, we again devise a reduction g from 3D-Match, but now directly to inj-Match. In order to define g, let (S, ℓ) be an instance of 3D-Match, where $\ell \geq 1$ and $S = \{s_1, s_2, \dots, s_k\}$ with $s_i = (p_i, q_i, r_i)$, for $1 \leq i \leq k$. Next, we define a word w over the alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}, \$, ¢, \#\}$ and a pattern α which uses the variables $x_{i,j}$, for $1 \leq i \leq \ell$ and $1 \leq j \leq 3$, and y_i, z_j, for $1 \leq i \leq \ell + 1$ and $1 \leq j \leq 2\ell + 2$. We first define the factors

$$\begin{aligned} \beta_i &= x_{i,1}^2 x_{i,2}^2 x_{i,3}^2, && 1 \leq i \leq \ell\,, \\ u_i &= (\mathtt{a}^{p_i}\mathtt{b})^2(\mathtt{a}^{q_i}\mathtt{b})^2(\mathtt{a}^{r_i}\mathtt{b})^2, && 1 \leq i \leq k\,, \\ \overline{\#}_i &= (\#¢^1\#¢^2\# \cdots \#¢^i\#)^m, && 1 \leq i \leq 2k+2\,, \end{aligned}$$

where $m = \max\{2k + 1, 3\ell\} + 1$. Then, in order to form α and w, these factors are combined in the following way:

$$\begin{aligned} \alpha &= z_1^m\, y_1\, z_2^m\, \beta_1\, z_3^m\, y_2\, z_4^m\, \beta_2\, z_5^m\, y_3\, z_6^m\, \beta_3 \cdots \beta_\ell\, z_{2\ell+1}^m\, y_{\ell+1}\, z_{2\ell+2}^m\,, \\ w &= \overline{\#}_1\, \$^1\, \overline{\#}_2\, u_1\, \overline{\#}_3\, \$^3\, \overline{\#}_4\, u_2\, \overline{\#}_5\, \$^5\, \overline{\#}_6\, u_3 \cdots u_k\, \overline{\#}_{2k+1}\, \$^{2k+1}\, \overline{\#}_{2k+2}\,. \end{aligned}$$

Finally, we set $g(S, \ell) = (\alpha, w)$ and, in the following, let (S, ℓ) be a fixed instance of 3D-Match and $(\alpha, w) = g(S, \ell)$.

Let us give an intuitive explanation of this reduction. The triples (p_i, q_i, r_i) of S are encoded as strings $u_i = (\mathtt{a}^{p_i}\mathtt{b})^2(\mathtt{a}^{q_i}\mathtt{b})^2(\mathtt{a}^{r_i}\mathtt{b})^2$, i. e., the integer components are now encoded in unary and every component is represented as a square. Since the factors β_i of α are necessarily mapped to factors of w that are three consecutive squares, any substitution mapping α to w maps every β_i to some u_j; thus, selecting ℓ many of the u_j (the single occurrence variables y_i make sure that any u_j can be selected). If the substitution is injective, then the selected u_j must correspond to a solution for the 3D-Match instance and the factors $\overline{\#}_i$ and $\$^j$ ensure that if the 3D-Match instance has a solution, then also a substitution can select the corresponding u_j from w in an injective way.

We now first prove that any substitution mapping α to w can be interpreted as selecting ℓ many of the factors u_i.

Lemma 5.8. *If* $h(\alpha) = w$ *for some substitution* h, *then there exist* $t_1, t_2, \dots, t_\ell$ *with* $1 \leq t_1 < t_2 < \dots < t_\ell \leq k$, *such that, for every* i *with* $1 \leq i \leq \ell$, $h(\beta_i) = u_{t_i}$.

[3]As shown in [9], the variant where we require the factorisation to have short factors instead of a large size is NP-complete also for fixed alphabets.

Proof. We assume that $h(\alpha) = w$, where h is a substitution. Let j, where $1 \leq j \leq \ell$, be arbitrarily chosen. We recall that $m = \max\{2k + 1, 3\ell\} + 1$, which implies that the only factors of the form v^m in w are exactly the factors $\overline{\#}_i$, where $1 \leq i \leq 2k + 2$. Hence, $h(z^m_{2j}) = \overline{\#}_r$ and $h(z^m_{2j+1}) = \overline{\#}_{r'}$, for some r and r' with $1 \leq r < r' < 2k + 2$. This implies that

$$h(z^m_{2j}\,\beta_j\,z^m_{2j+1}) = \overline{\#}_r\,\pi_r\,\overline{\#}_{(r+1)}\,\pi_{(r+1)}\,\overline{\#}_{(r+2)}\,\cdots\overline{\#}_{(r'-1)}\,\pi_{(r'-1)}\,\overline{\#}_{r'}\,,$$

where, for every i with $r \leq i \leq r' - 1$, the factor π_i is either $u_{\frac{i}{2}}$ or $\$^i$. Furthermore, since $\beta_j = x^2_{j,1}x^2_{j,2}x^2_{j,3}$, we have that $h(\beta_j)$ is the concatenation of 3 squares, i. e.,

$$h(\beta_j) = \pi_r\,\overline{\#}_{(r+1)}\,\pi_{(r+1)}\,\overline{\#}_{(r+2)}\,\cdots\overline{\#}_{(r'-1)}\,\pi_{(r'-1)} = v_1v_1v_2v_2v_3v_3\,.$$

We first consider the case where $r + 1 < r'$. Since the factors $\overline{\#}_i$, for every i with $r + 1 \leq i \leq r' - 1$, are not repeated in w, we can conclude that these factors cannot be completely contained in one of the roots v_1, v_2 or v_3. We now assume that $\pi_r = \$^r$. Since π_r is not of even length, we can conclude that there is an integer i with $1 \leq i \leq 3$ such that v_i starts in π_r, but does not end in π_r, i. e., it overlaps the boundary between the factors π_r and $\overline{\#}_{(r+1)}$. Furthermore, as explained above, $\overline{\#}_{(r+1)}$ cannot be contained in v_i; thus, $v_i = v'_iv''_i$, where v'_i is a suffix of π_r and v''_i is a prefix of $\overline{\#}_{(r+1)}$. This implies that in $h(\beta_j)$ there occurs a factor $v'_iv''_iv'_iv''_i$ with $v'_i \in \{\$\}^+$ and $v''_i \in \{\#, ¢\}^+$, which is a contradiction, since such a factor does not occur in w. If, on the other hand, $\pi_r = u_{\frac{r}{2}}$, then it follows from the fact that $u_{\frac{r}{2}}$ is neither a square nor the concatenation of two squares, that there must exist an integer i with $1 \leq i \leq 3$ such that v_i overlaps the boundary between the factors π_r and $\overline{\#}_{(r+1)}$, which implies that in $h(\beta_j)$ there occurs a factor $v'_iv''_iv'_iv''_i$ with $v'_i \in \{\mathsf{a}, \mathsf{b}\}^+$ and $v''_i \in \{\#, ¢\}^+$, which again is a contradiction.

Consequently, $r + 1 = r'$, which implies that $h(\beta_j) = \pi_r$ and since $|\$^r|$ is odd and $|h(\beta_j)|$ is even, we can conclude that $\pi_r = u_{\frac{r}{2}}$. Hence, there must exist $t_1, t_2, \cdots, t_\ell$ with $1 \leq t_1 < t_2 < \ldots < t_\ell \leq k$, such that, for every integer i with $1 \leq i \leq \ell$, $h(\beta_i) = u_{t_i}$, which concludes the proof. □

We are now ready to prove that g is a reduction from 3D-Match to the injective matching problem for non-cross patterns.

Lemma 5.9. *Let (S, ℓ) be an instance of 3D-Match and let $(\alpha, w) = g(S, \ell)$. Then (S, ℓ) has a solution if and only if there exists an injective substitution h with $h(\alpha) = w$.*

Proof. In this proof, we shall use the following notation. For any substitution h, any pattern α and any set V of variables that occur in α, we say that h is *injective with respect to* V if, for every $x, y \in V$, $x \neq y$ implies $h(x) \neq h(y)$. Obviously, h is injective if and only if it is injective with respect to the set of all variables that occur in α.

We start with the *only if* direction and assume that $\{s_{t_1}, s_{t_2}, \ldots, s_{t_\ell}\}$ is a solution for (S, ℓ). We define an injective substitution h in the following way. For every i with $1 \leq i \leq \ell$, we define $h(x_{i,1}) = \mathsf{a}^{p_{t_i}}\mathsf{b}$, $h(x_{i,2}) = \mathsf{a}^{q_{t_i}}\mathsf{b}$ and $h(x_{i,3}) = \mathsf{a}^{r_{t_i}}\mathsf{b}$. Thus, for every i with $1 \leq i \leq \ell$, $h(\beta_i) = u_{t_i}$ and, since $\{s_{t_1}, s_{t_2}, \ldots, s_{t_\ell}\}$ is a solution, h is injective with respect to $\{x_{i,j} \mid 1 \leq i \leq \ell, 1 \leq j \leq 3\}$. We extend h in such a way that $h(z_1) = \#¢^1\#$, $h(z^m_{2l+2}) = \#¢^1\#¢^2\#\cdots\#¢^{2k+2}\#$ and, for every i with $1 \leq i \leq \ell + 1$, we have $h(z^m_{2i}) = \#¢^1\#¢^2\#\cdots\#¢^{2t_i}\#$ and $h(z^m_{2i+1}) = \#¢^1\#¢^2\#\cdots\#¢^{2t_i+1}\#$. This particularly implies that $h(z^m_1) = \overline{\#}_1$, $h(z^m_{2l+2}) = \overline{\#}_{2k+2}$ and, for every i with $1 \leq i \leq \ell + 1$, we have $h(z^m_{2i}) = \overline{\#}_{2t_i}$ and $h(z^m_{2i+1}) = \overline{\#}_{2t_i+1}$. Furthermore, since the words $\#¢^1\#¢^2\#\cdots\#¢^i\#$, $1 \leq i \leq 2k + 2$, are pairwise different, h is injective with respect to $\{z_i \mid 1 \leq i \leq 2\ell + 2\}$. Finally, we extend h in such a way that the single occurrence variables y_i, for $1 \leq i \leq \ell + 1$, are mapped to the parts between the $h(z_{2i}^m \beta_i z_{2i+1}^m) = \overline{\#}_{2t_i} u_{t_i} \overline{\#}_{2t_i+1}$ factors, i. e.,

$$h(y_1) = \$^1 \, \overline{\#}_2 \, u_1 \, \overline{\#}_3 \cdots \overline{\#}_{2(t_1-1)} \, u_{t_1-1} \, \overline{\#}_{2t_1-1} \, \$^{2t_1-1} \, ,$$

$$h(y_{\ell+1}) = \$^{2t_\ell+1} \, \overline{\#}_{2(t_\ell+1)} \, u_{t_\ell+1} \, \overline{\#}_{2t_\ell+3} \cdots \overline{\#}_{2k} \, u_k \, \overline{\#}_{2k+1} \, \$^{2k+1} \, ,$$

and, for every i with $2 \leq i \leq \ell$,

$$h(y_i) = \$^{2t_i+1} \, \overline{\#}_{2t_i+2} \, u_{t_i+1} \, \overline{\#}_{2t_i+3} \cdots u_{t_{i+1}-1} \, \overline{\#}_{2(t_{i+1}-1)+1} \, \$^{2(t_{i+1}-1)+1} \, .$$

Since all of these words $h(y_i)$, for $1 \leq i \leq \ell + 1$, contain at least one of the factors $\$^{2j-1}$, where $1 \leq j \leq k + 1$, we can conclude that they are pairwise different and, thus, h is injective with respect to $\{y_i \mid 1 \leq i \leq \ell + 1\}$. Moreover, $h(\alpha) = w$ holds and, since all variables $x_{i,j}$, with $1 \leq i \leq \ell$ and $1 \leq j \leq 3$, are mapped to words over $\{\mathtt{a}, \mathtt{b}\}$, all variables z_i, with $1 \leq i \leq 2\ell + 2$, are mapped to words over $\{\#, ¢\}$ and all variables y_i, with $1 \leq i \leq \ell + 1$, are mapped to words that contain at least one occurrence of \$, we can conclude that h is injective with respect to all variables that occur in α. This concludes the proof of the *only if* direction.

In order to prove the *if* direction, we assume that $h(\alpha) = w$ for some injective substitution h. By Lemma 5.8, we can conclude that there exist $t_1, t_2, \cdots, t_\ell$ with $1 \leq t_1 < t_2 < \ldots < t_\ell \leq k$, such that, for every i with $1 \leq i \leq \ell$, $h(\beta_i) = u_{t_i}$. This directly implies that, for every i with $1 \leq i \leq \ell$, we get $h(x_{i,1}) = \mathtt{a}^{p_{t_i}}\mathtt{b}$, $h(x_{i,2}) = \mathtt{a}^{q_{t_i}}\mathtt{b}$ and $h(x_{i,3}) = \mathtt{a}^{r_{t_i}}\mathtt{b}$. From the fact that h is injective, we can now easily conclude that $\{s_{t_1}, s_{t_2}, \ldots, s_{t_\ell}\}$ is a solution for (S, ℓ). More precisely, if, for some i and i' with $1 \leq i < i' \leq \ell$, we have $p_{t_i} = p_{t_{i'}}$, $q_{t_i} = q_{t_{i'}}$ or $r_{t_i} = r_{t_{i'}}$, then this implies $h(x_{i,1}) = h(x_{i',1})$, $h(x_{i,2}) = h(x_{i',2})$ or $h(x_{i,3}) = h(x_{i',3})$, respectively, which contradicts the injectivity of h. □

Since the mapping g can be computed in polynomial time, we conclude the following.

Theorem 5.10. inj-*Match is* NP-*complete for* $\textsc{Pat}_{\mathrm{nc}}$ *and* $\textsc{Pat}_{\mathrm{scd}\leq k}$, *for every* k *with* $k \geq 1$, *even for a constant alphabet of size at least* 5.

6 CONCLUSIONS AND OPEN PROBLEMS

For some of the classes of patterns with variables for which it is known that the matching problem can be solved in polynomial-time, we have presented efficient algorithms. The algorithms for patterns with a bounded number of repeated variables and for patterns with a bounded scope coincidence degree have an exponential dependency on these parameters, which, suggested by the $W[1]$-hardness of the respective parameterised problems, can probably not be avoided. However, for rather small bounds on the parameters, i. e., for patterns with only one repeated variable and patterns with a scope coincidence degree of one (i. e., non-cross patterns), our algorithms achieve rather efficient running-times and require novel and sophisticated word combinatorial insights. As mentioned in the introduction, these algorithms can also be used in order to efficiently compute descriptive patterns of finite sets of words.

This algorithmic part is complemented by showing that it is very unlikely that comparitively efficient algorithms (or any polynomial-time algorithms) exist for the injective variant of the matching problem, where different variables must be replaced by different strings. More precisely, the injective variant of the matching problem is NP-complete even for regular patterns (and, as an immediate consequence, also for patterns with a bounded number of repeated variables or a bounded scope coincidence degree (including non-cross patterns)). However, as a caveat, we stress that we were only able to show this result with respect to unbounded alphabets. For the case of constant alphabets (of size at least 5), we obtained the slightly weaker result that the injective variant of the matching problem is NP-complete even for non-cross patterns (so also for patterns with a bounded scope coincidence degree). It is an open problem whether the injective variant of the matching problem for regular patterns (or some other class of patterns with a bounded number of repeated variables) can be efficiently solved for fixed alphabets.

As further research, it might be worthwhile to find efficient algorithms for other classes of patterns for which polynomial-time matching is generally possible (note that the meta-theorem of [36] is a powerful tool for identifying such pattern classes). Another extension of our results is the *erasing* matching problem, where variables can also be substituted by the empty word. On the one hand, our efficient algorithms can be extended canonically to the erasing matching problem as well, without any change in the complexity. On the other hand, it is unclear whether also in the erasing case the requirement for injectivity[4] makes the matching problem harder (in the sense demonstrated in Section 5). More precisely, the erasing matching problem can also be solved in polynomial-time for regular patterns, non-cross patterns etc., but Theorem 5.6 does not imply that requiring injectivity makes the matching problem hard; in fact, while computing a unique factorisation of size at least n for a word w is equivalent to solving the injective matching problem for w and pattern $x_1x_2 \dots x_n$, the latter task becomes trivial, if variables can be erased.

Since patterns with variables have their origin in learning theory and formal languages, it might be worthwhile to extend the concept of injectivity accordingly. More precisely, to a given pattern α with terminal alphabet Σ, we can associate its pattern language $L(\alpha)$, which is the set of all words over Σ that match α. While the learnability as well as language theoretical properties of these pattern languages are well-investigated, *injective pattern languages*, i. e., the set of all words that match the pattern in an injective way, are, to the best of our knowledge, never considered so far.

7 ACKNOWLEDGEMENTS

The work of Florin Manea was funded by the DFG grant MA 5725/1-2 (project number 218587403).

We thank the anonymous reviewers for their valuable feedback, which improved the exposition of the paper.

REFERENCES

[1] Amihood Amir and Igor Nor. 2007. Generalized function matching. *Journal of Discrete Algorithms* 5 (2007), 514–523. Issue 3.

[2] Dana Angluin. 1980. Finding patterns common to a set of strings. *J. Comput. System Sci.* 21 (1980), 46–62.

[3] Brenda S. Baker. 1996. Parameterized Pattern Matching: Algorithms and Applications. *J. Comput. System Sci.* 52 (1996), 28–42.

[4] Hideo Bannai, Travis Gagie, Shunsuke Inenaga, Juha Kärkkäinen, Dominik Kempa, Marcin Piatkowski, and Shiho Sugimoto. 2018. Diverse Palindromic Factorization is NP-Complete. *International Journal of Foundations of Computer Science* 29, 2 (2018), 143–164.

[5] Pablo Barceló, Leonid Libkin, Anthony W. Lin, and Peter T. Wood. 2012. Expressive Languages for Path Queries over Graph-Structured Data. *ACM Transactions on Database Systems* 37 (2012).

[6] Cezar Câmpeanu, Kai Salomaa, and Sheng Yu. 2003. A formal study of practical regular expressions. *International Journal of Foundations of Computer Science* 14 (2003), 1007–1018.

[7] Raphaël Clifford, Aram Wettroth Harrow, Alexandru Popa, and Benjamin Sach. 2009. Generalised Matching. In *Proceedings of the 16th International Symposium on String Processing and Information Retrieval, SPIRE 2009, Saariselkä, Finland, August 25-27, 2009*. 295–301.

[8] Anne Condon, Ján Manuch, and Chris Thachuk. 2008. Complexity of a Collision-Aware String Partition Problem and Its Relation to Oligo Design for Gene Synthesis. In *Computing and Combinatorics, 14th Annual International Conference, COCOON 2008, Dalian, China, June 27-29, 2008, Proceedings*. 265–275.

[9] Anne Condon, Ján Manuch, and Chris Thachuk. 2015. The complexity of string partitioning. *Journal of Discrete Algorithms* 32 (2015), 24–43.

[10] Maxime Crochemore. 1981. An Optimal Algorithm for Computing the Repetitions in a Word. *Inform. Process. Lett.* 12, 5 (1981), 244–250.

[4]Injectivity means, in the erasing case, that only variables that are not erased must be substituted by different strings.

[11] Maxime Crochemore and Wojciech Rytter. 1991. Usefulness of the Karp-Miller-Rosenberg Algorithm in Parallel Computations on Strings and Arrays. *Theoretical Computer Science* 88, 1 (1991), 59–82.

[12] Maxime Crochemore and Wojciech Rytter. 1995. Squares, Cubes, and Time-Space Efficient String Searching. *Algorithmica* 13, 5 (1995), 405–425.

[13] Thomas Erlebach, Peter Rossmanith, Hans Stadtherr, Angelika Steger, and Thomas Zeugmann. 2001. Learning one-variable pattern languages very efficiently on average, in parallel, and by asking queries. *Theoretical Computer Science* 261 (2001), 119–156.

[14] Henning Fernau, Florin Manea, Robert Mercas, and Markus L. Schmid. 2015. Pattern Matching with Variables: Fast Algorithms and New Hardness Results. In *32nd International Symposium on Theoretical Aspects of Computer Science, STACS 2015, March 4-7, 2015, Garching, Germany.* 302–315.

[15] Henning Fernau, Florin Manea, Robert Mercas, and Markus L. Schmid. 2018. Revisiting Shinohara's algorithm for computing descriptive patterns. *Theoretical Computer Science* 733 (2018), 44–54.

[16] Henning Fernau and Markus L. Schmid. 2015. Pattern matching with variables: A multivariate complexity analysis. *Information and Computation* 242 (2015), 287–305.

[17] Henning Fernau, Markus L. Schmid, and Yngve Villanger. 2015. On the Parameterised Complexity of String Morphism Problems. *Theory of Computing Systems* (2015).

[18] Dominik D. Freydenberger. 2013. Extended Regular Expressions: Succinctness and Decidability. *Theory of Computing Systems* 53 (2013), 159–193.

[19] Jeffrey E. F. Friedl. 2006. *Mastering Regular Expressions* (third ed.). O'Reilly, Sebastopol, CA.

[20] Michael R. Garey and David S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA.

[21] Dan Gusfield. 1997. *Algorithms on strings, trees, and sequences: computer science and computational biology.* Cambridge University Press, New York, NY, USA.

[22] Oscar H. Ibarra, Ting-Chuen Pong, and Stephen M. Sohn. 1995. A note on parsing pattern languages. *Pattern Recognition Letters* 16 (1995), 179–182.

[23] Juhani Karhumäki, Wojciech Plandowski, and Filippo Mignosi. 2000. The Expressibility of Languages and Relations by Word Equations. *J. ACM* 47 (2000), 483–505.

[24] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. 2006. Linear work suffix array construction. *J. ACM* 53 (2006), 918–936. Issue 6.

[25] Michael J. Kearns and Leonard Pitt. 1989. A Polynomial-Time Algorithm for Learning k-Variable Pattern Languages from Examples. In *Proceedings of the Second Annual Workshop on Computational Learning Theory, COLT 1989, Santa Cruz, CA, USA, July 31 - August 2, 1989.* 57–71.

[26] Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. 2012. Efficient Data Structures for the Factor Periodicity Problem. In *Proceedings of the 19th International Symposium on String Processing and Information Retrieval, SPIRE 2012, Cartagena de Indias, Colombia, October 21-25, 2012.* 284–294.

[27] Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. 2015. Internal Pattern Matching Queries in a Text and Applications. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015.* 532–551.

[28] Dmitry Kosolobov, Florin Manea, and Dirk Nowotka. 2017. Detecting One-Variable Patterns. In *Proceedings of the 24th International Symposium on String Processing and Information Retrieval, SPIRE 2017, Palermo, Italy, September 26-29, 2017.* 254–270.

[29] M. Lothaire. 1997. *Combinatorics on Words.* Cambridge University Press.

[30] M. Lothaire. 2002. *Algebraic Combinatorics on Words.* Cambridge University Press, Cambridge, New York, Chapter 3.

[31] Alexandru Mateescu and Arto Salomaa. 1994. Finite Degrees of Ambiguity in Pattern Languages. *RAIRO Informatique Théoretique et Applications* 28 (1994), 233–253.

[32] Yen K. Ng and Takeshi Shinohara. 2008. Developments from enquiries into the learnability of the pattern languages from positive data. *Theoretical Computer Science* 397 (2008), 150–165.

[33] Sebastian Ordyniak and Alexandru Popa. 2016. A Parameterized Study of Maximum Generalized Pattern Matching Problems. *Algorithmica* 75 (2016), 1–26.

[34] Daniel Reidenbach. 2008. Discontinuities in pattern inference. *Theoretical Computer Science* 397 (2008), 166–193.

[35] Daniel Reidenbach and Markus L. Schmid. 2010. A Polynomial Time Match Test for Large Classes of Extended Regular Expressions. In *Proceedings of the 15th International Conference on Implementation and Application of Automata, CIAA 2010, Winnipeg, MB, Canada, August 12-15, 2010.* 241–250.

[36] Daniel Reidenbach and Markus L. Schmid. 2014. Patterns with bounded Treewidth. *Information and Computation* 239 (2014), 87–99.

[37] Markus L. Schmid. 2013. A note on the complexity of matching patterns with variables. *Inform. Process. Lett.* 113, 19-21 (2013), 729–733.

[38] Markus L. Schmid. 2016. Computing equality-free and repetitive string factorisations. *Theoretical Computer Science* 618 (2016), 42–51.

[39] Takeshi Shinohara. 1982. Polynomial Time Inference of Pattern Languages and Its Application. In *Proceedings of 7th IBM Symposium on Mathematical Foundations of Computer Science, MFCS*. 191–209.