Pattern Matching with Variables: Fast Algorithms and New Hardness Results

Henning Fernau¹, Florin Manea², Robert Mercaș², and Markus L. Schmid¹

1 Fachbereich IV – Abteilung Informatikwissenschaften, Universität Trier, D-54286 Trier, Germany, {Fernau, MSchmid}@uni-trier.de
2 Kiel University, Department of Computer Science, D-24098 Kiel, Germany, {flm, rgm}@informatik.uni-kiel.de

Abstract

A pattern (i.e., a string of variables and terminals) maps to a word, if this is obtained by uniformly replacing the variables by terminal words; deciding this is \( \mathcal{NP} \)-complete. We present efficient algorithms¹ that solve this problem for restricted classes of patterns. Furthermore, we show that it is \( \mathcal{NP} \)-complete to decide, for a given number \( k \) and a word \( w \), whether \( w \) can be factorised into \( k \) distinct factors; this shows that the injective version (i.e., different variables are replaced by different words) of the above matching problem is \( \mathcal{NP} \)-complete even for very restricted cases.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, F.4.3 Formal Languages

Keywords and phrases combinatorial pattern matching, combinatorics on words, patterns with variables, \( \mathcal{NP} \)-complete string problems

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

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, \( x_1abx_1x_2cx_2x_1 \) can be mapped to \( acabacaccaaccaacac \) and \( babbacab \) by the replacements \( (x_1 \rightarrow ac, x_2 \rightarrow caa) \) and \( (x_1 \rightarrow b, x_2 \rightarrow a) \), 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]), learning theory (inductive inference [2, 25, 27, 10], PAC-learning [20]), combinatorics on words (word equations [18, 24], unavoidable patterns [23]), pattern matching (generalised function matching [1, 26]), database theory (extended conjunctive regular path queries [4]), and we can also find them in practice in the form of extended regular expressions with backreferences [5, 14], 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 \( t \); testing whether a word \( w \) contains
Pattern Matching with Variables: Fast Algorithms and Hardness Results

A $k$-repetition is equivalent to testing whether the pattern $xy^kz$ can be mapped to $w$, etc. 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.

Unfortunately, deciding whether a given pattern can be mapped to a given word, the matching problem, is $\mathcal{NP}$-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 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, 11]). A comprehensive multivariate analysis of the complexity of the matching problem [12, 13] demonstrates that the $\mathcal{NP}$-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 $\mathcal{P}$ (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 [28, 29]).

Unfortunately, the existing polynomial time algorithms for these classes are fairly basic and they serve the mere purpose of proving containment in $\mathcal{P}$; thus, they cannot be considered efficient in a practical sense. Therefore, we present here better algorithms for the known polynomial variants of the matching problem. 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 [13], 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 programme 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 [7] 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 [7]). We show the $\mathcal{NP}$-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 $\mathcal{NP}$-complete if we require injectivity; thus, in terms of complexity, a clear borderline between the injective and the non-injective versions of the matching problem is established.

This paper is organised as follows. The next section contains basic definitions and then we give an overview of all our results in Section 3. In Section 4, we develop our algorithms for the matching problem and, in Section 5, we present the hardness result mentioned above.

## 2 Basic Definitions

For detailed definitions regarding combinatorics on words we refer to [22]. We denote our alphabet by $\Sigma$, the empty word by $\varepsilon$, and the length of a word $w$ by $|w|$. For $w \in \Sigma^*$ and
each $1 \leq i \leq j \leq |w|$, let $w[i,j] = w[i] \cdots w[j]$, where $w[k]$ represents the letter on position $k$, for $1 \leq k \leq |w|$. The catenation of $k$ words $w_1, \ldots, w_k$ is written $\Pi_{i=1,k} w_i$. If $w = w_i$ for all $1 \leq i \leq k$, this represents the $k$th power of $w$, denoted by $w^k$; here, $w$ is a root of $w^k$. We say that $w$ is primitive if it cannot be expressed as a power $\ell > 1$ of any root.

For any $w \in \Sigma^*$, a factorisation of $w$ is a tuple $p = (u_1, u_2, \ldots, u_k) \in (\Sigma^+)^k$, $k \in \mathbb{N}$, with $w = u_1 u_2 \cdots u_k$. Every word $u_i$, $1 \leq i \leq k$, is called a factor (of $p$) or simply $p$-factor. Let $p = (u_1, u_2, \ldots, u_k)$ be an arbitrary factorisation. We define its set of factors as $\sf(p) = \{u_1, u_2, \ldots, u_k\}$ and its size as $|s(p)| = k$. A factorisation $p$ is unique if every factor is distinct, i.e., $s(p) = |s(p)|$. For every $1 \leq i \leq |s(p)|$, $p(i) = u_i$ denotes the $i$th factor of $p$. As an example, we consider the factorisation $p = (a, b, cba, a, ba)$ of the word $w = ababcbababba$. We note that $\sf(p) = \{a, ba, cba\}$ and $|s(p)| = 6$. For the sake of readability, we sometimes represent a factorisation $(a, ba, cba, a, ba, a)$ in the form $a \mid ba \mid cba \mid a \mid ba \mid a$.

Let $X = \{x_1, x_2, x_3, \ldots\}$ and call every $x \in X$ a variable. For a finite alphabet of terminals $\Sigma \cap X = \emptyset$, we define $\operatorname{Pat}_{\Sigma} = (X \cup \Sigma)^+$ and $\operatorname{Pat} = \bigcup_{\Sigma} \operatorname{Pat}_{\Sigma}$. Every $\alpha \in \operatorname{Pat}$ is a pattern and every $w \in \Sigma^*$ is a (terminal) word. Given a sequence $v$, a word or pattern, for the smallest sets $B \subseteq \Sigma$ and $Y \subseteq X$ with $v \in (B \cup Y)^*$, we denote $\operatorname{alph}(v) = B$ and $\operatorname{var}(v) = Y$.

For every $x \in \var(\alpha) \cup \operatorname{alph}(\alpha)$, $|\alpha|_x$ denotes the number of occurrences of $x$ in $\alpha$.

A substitution (for $\alpha$) is a mapping $h : \var(\alpha) \to \Sigma^*$. For every $x \in \var(\alpha)$, 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 $\alpha$ by $h(x)$ and leaving the terminals unchanged. If, for all $x, y \in \var(\alpha)$, $x \neq y$ implies $h(x) \neq h(y)$, then $h$ is injective. As an example, we consider the pattern $\beta = x_1 a x_2 b x_1 x_2$ and the words $u = \operatorname{bacbabababab} \quad \operatorname{v} = \operatorname{baababababab}$. It can be verified that $h(\beta) = u$, where $h(x_1) = \operatorname{bacb}$, $h(x_2) = b$ and $g(\beta) = v$, where $g(x_1) = g(x_2) = \operatorname{ab}$. Furthermore, $h$ is injective, $g$ is not and $\beta$ cannot be mapped to $v$ by an injective substitution.

The matching problem, denoted by MATCH, is to decide for a given pattern $\alpha$ and word $w$, whether there exists a substitution $h$ with $h(\alpha) = w$. By $\operatorname{inj-MATCH}$, we denote the variant of the matching problem where the substitution needs to be injective.

For any $P \subseteq \operatorname{Pat}$, the matching problem for $P$ is the matching problem, where the input patterns are from $P$.

A pattern $\alpha$ is regular if, for every $x \in \var(\alpha)$, $|\alpha|_x = 1$, and the class of regular patterns is denoted by $\operatorname{Pat}_{\operatorname{reg}}$. For any $k \in \mathbb{N}$, a $k$-variable pattern is a pattern $\alpha$ that satisfies $|\var(\alpha)| \leq k$ and a pattern $\beta$ with $|\{x \in \var(\beta) \mid |\beta|_x \geq 2\}| \leq k$ is a $k$-repeated-variable pattern. For every $k \in \mathbb{N}$, $\operatorname{Pat}_{\var|\leq k}$ and $\operatorname{Pat}_{\var|\leq k}$ denote the set of $k$-variable patterns and $k$-repeated-variable patterns, respectively. Let $\alpha$ be a pattern. For every $y \in \var(\alpha)$, the scope of $y$ in $\alpha$ is defined by $\operatorname{sc}_\alpha(y) = \{i, i+1, \ldots, j\}$, where $i$ is the leftmost and $j$ the rightmost occurrence of $y$ in $\alpha$. The scopes of some variables $y_1, y_2, \ldots, y_k \in \var(\alpha)$ coincide in $\alpha$ if $\bigcap_{1 \leq i \leq k} \operatorname{sc}_\alpha(y_i) \neq \emptyset$. By $\operatorname{scd}(\alpha)$, we denote the scope coincidence degree (scd for short) of $\alpha$, which is the maximum number of variables in $\alpha$ such that their scopes coincide. For example, the scopes of all variables coincide in $\alpha_1 = x_1 x_2 x_1 x_2 x_1 x_2 x_3$, but the scopes of $x_1$ and $x_3$ do not coincide in $\alpha_2 = x_1 x_2 x_1 x_2 x_3 x_2 x_3$; thus, $\operatorname{scd}(\alpha_1) = 3$ and $\operatorname{scd}(\alpha_2) = 2$.

For every $k \in \mathbb{N}$, let $\operatorname{Pat}_{\operatorname{scd} \leq k}$ denote the set of patterns $\alpha$ with $\operatorname{scd}(\alpha) \leq k$. By definition, $\operatorname{Pat}_{\operatorname{scd} \leq 1}$ coincides with the class of non-cross patterns (see [29]), which we denote by $\operatorname{Pat}_{\operatorname{nc}}$.

The one-variable blocks in a pattern are contiguous blocks of occurrences of the same variable. For instance, the number of one-variable blocks in $\alpha = x_1 x_2 x_2 a x_2 x_2 x_2 x_3 x_2 x_2 x_3 x_2 x_3 x_2$ is 7. A pattern $\alpha$ with $m$ one-variable blocks can be written as $\alpha = w_0 \Pi_{i=1,m} (z_i^{|\alpha|_i} w_i)$ with $z_i \in \var(\alpha)$ for $i \in \{1, \ldots, m\}$ and $z_i \neq z_{i+1}$, whenever $w_i = \varepsilon$ for $i \in \{1, \ldots, m-1\}$. 

\footnote{There exist variants of the matching problem where substitutions can also \textit{erase} variables by mapping them to $\varepsilon$. In this work, we are not concerned with this variant of the problem.}
3 Summary of Our Results

The classical and parametrised complexity of the matching problem for patterns has been recently investigated and is well understood (see [6, 12, 13, 26]). 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, of regular patterns, of non-cross patterns and of patterns with a bounded scope coincidence degree (see [2, 29, 28]). 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 \( \mathcal{O}(\frac{n^{k-1}m}{(k-1)!}) \), where \( m \) and \( n \) are the lengths of the pattern and the word (see [17]). For patterns with a scd of at most \( k \), an \( \mathcal{O}(mn^{2(k+3)}(k + 1)^2) \) time algorithm is given in [28], 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 \( \mathcal{P} \) (see [29]) leads to an \( \mathcal{O}(n^4) \)-time algorithm. Hence, we consider the following problem worth investigating.

\( \uparrow \) 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 patterns 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 \( \alpha \) having \( m \) one-variable blocks matches a word \( w \) of length \( n \) in \( \mathcal{O}(nm \log n) \) time; this is an important improvement over the previously available \( \mathcal{O}(n^4) \) 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 \( \text{PAT}_{\text{nc}} \), we obtain a stringology result that extends in a non-trivial manner a major result from [8], showing how the primitively rooted squares contained in a word of length \( n \) can be listed optimally in \( \mathcal{O}(n \log n) \). Our result shows that given a word \( w \) of length \( n \) and a word \( v \) shorter than \( n \), then \( w \) contains \( \mathcal{O}(n \log n) \) factors of the form \( uvu \) with \( uv \) primitive, and all these factors can be found in \( \mathcal{O}(n \log n) \) 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 \( \text{PAT}_{\text{scd} \leq k} \) is solvable in \( \mathcal{O}\left(\frac{n^{2k}m}{((k-1)!)^2}\right) \) 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 \( \text{PAT}_{\text{var} \leq k} \) of patterns with at most \( k \) repeated variables. For the basic case \( k = 1 \) we obtain that the matching problem is solved in \( \mathcal{O}(n^2) \) time. 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 \( \text{PAT}_{\text{var} \leq k} \) is solvable in \( \mathcal{O}\left(\frac{n^{2k}}{((k-1)!)^2}\right) \) time, where \( n \) is the length of the input word. 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 scd bounded by \( k + 1 \), and then direct applying our previous algorithm solving the matching problem for \( \text{PAT}_{\text{scd} \leq k+1} \).
The classes of non-cross patterns and of patterns with a bounded scd or with a bounded number of repeated variables are of special interest, since for them we can compute so-called descriptive patterns (see [2, 29]) in polynomial time. A pattern \( \alpha \) 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 \( \mathcal{NP} \)-complete in general, means to infer a pattern common to a finite set of words, with applications for inductive inference of pattern languages (see [25]). 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 [11] 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 scd. 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 [13] 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 scd, the mentioned \( W[1] \)-hardness result carries over to the case where the scd is a parameter and therefore it is just as unlikely to find an algorithm that is not exponential in the scd. This observation justifies the exponential dependency of our algorithms on the number of repeated variables and the scd.

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 [12] it is shown that MATCH is still \( \mathcal{NP} \)-complete if the alphabet size and the length of the words the variables are mapped to are bounded, whereas it is in \( \mathcal{P} \) 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.

\textbf{Theorem 1.} The following problem is \( \mathcal{NP} \)-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 \( \mathcal{NP} \)-complete even for the trivial patterns \( x_1x_2\cdots x_k \), which means that, under the assumption \( \mathcal{P} \neq \mathcal{NP} \), 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 [7], it is shown that it is \( \mathcal{NP} \)-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 \( \mathcal{NP} \)-completeness is preserved if the length bound is dropped and instead we have a lower bound on the number of factors.

\section{Algorithmic Results}

In this section we propose a series of algorithms for the matching problem for several classes of patterns. We begin by looking at classes where the number of repeated variables is bounded: we consider the basic classes where no variable or, more interestingly, only one variable is...
repeated, and then investigate the class of patterns in which \( k \geq 2 \) variables are repeated. Then, we look at the more involved case of patterns with bounded scope coincidence degree. The basic case is, in this setting, \( \text{Pat}_{nc} \), where the scope coincidence degree is upper bounded by 1; after presenting an algorithm solving the matching problem for \( nc \)-patterns, we analyse the general case when the upper bound of the scope coincidence degree is \( k \geq 2 \).

We assume for every input word \( w \) of length \( n \) that \( \text{alph}(w) \subseteq \{1, \ldots, n\} \) (i.e., the symbols are integers). This is a common assumption in algorithmics on words (see the discussion in [19]). 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 tree and suffix array structures, as well as data structures allowing us to retrieve in \( O(1) \) time the length of the longest common prefix of any suffixes \( w[i..n] \) and \( w[j..n] \) of \( w \), denoted \( \text{LCP}_w(i, j) \) (the subscript \( w \) is omitted when there is no danger of confusion). These are \( \text{LCP} \) data structures (see, e.g., [19, 16]). Symmetrically we can build structures allowing us to retrieve in \( O(1) \) time the length of the longest common suffix of any two prefixes \( w[1..i] \) and \( w[1..j] \) of \( w \), denoted \( \text{LCS}(i, j) \).

The first case we approach is that of regular patterns. It was already known from [29] that the matching problem for such patterns can be solved in linear time, when the alphabet of the input word \( w \) is constant. However, it is not hard to see that the same time bound can be achieved in our setting (when the input words are over integer alphabets): the matching problem for \( \text{Pat}_{reg} \) is solvable in \( O(|w| + |\alpha|) \) time, where \( w \) is the input word and \( \alpha \) the input pattern. More interesting is the case of patterns that contain one repeated variable.

\begin{lemma}
The matching problem for \( \text{Pat}_{\text{var} \leq 1} \) is solvable in \( O(|w|(|w| + |\alpha|)) \) time, where \( w \) is the input word and \( \alpha \) is the input pattern.
\end{lemma}

The main idea behind the algorithm used to obtain this result is to find an assignment for the repeating variable from the input pattern \( \alpha \), say \( x \), such that all constant factors are well placed within the word, and then fill up (using a linear pattern matching algorithm to correctly align the 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.

Finding the factors of the word which are images of the parts of the pattern that do not contain the repeated variable can be done in a quadratic time preprocessing.

To find a suitable assignment for \( x \) we first choose the length of its image \( \ell \leq n \) and, in linear time, partition the suffix array of the word in several clusters (i.e., blocks of consecutive positions that are not extendable to the left or right) such that the suffixes contained in one cluster share a common prefix of length \( \ell \), that will correspond to the image of \( x \). Note that if there exists a mapping of the pattern to the word, with the image of \( x \) of length \( \ell \), then it maps all factors starting with \( x \) to prefixes of elements in the same cluster. Essentially, to check if such a mapping exists we use a greedy processing of each cluster. Fixing the cluster, we also fix the image of \( x \), denoted \( w_x \) in the following, and the suffixes in that cluster provide all the occurrences of \( w_x \) in \( w \). Now, we can assume without loss of generality that \( \alpha \) starts and ends with variables different from \( x \). If this would not be the case, we just isolate the lengthwise maximal prefix \( \alpha_p \) and suffix \( \alpha_s \) of \( \alpha \) that contain only occurrences of \( x \) and terminals; the images of these two factors of \( \alpha \) are now known, as we know the image of \( x \). So we check if the image of \( \alpha_p \) is a prefix and the image of \( \alpha_s \) is a suffix of \( w \). If yes, we just have to check whether the rest of the pattern (\( \alpha \) without \( \alpha_p \) and \( \alpha_s \)) maps to the rest of the word (\( w \) without the images of \( \alpha_p \) and \( \alpha_s \)); this puts us in the aforementioned case. Further, we sort the elements in the cluster in the order of their occurrence in the word. Then the \( i^{th} \) occurrence of \( x \) in \( \alpha \) is mapped to the leftmost occurrence of \( w_x \) surrounded by the same terminal words as the variable \( x \) in \( \alpha \), such that the factor of \( \alpha \) occurring between the \( i^{th} \) and \( i-1^{th} \) occurrence of \( x \) matches the factor of \( w \) found between the respective
images of these variables. As the variables $x$ are considered from left to right and mapped,
respectively, to occurrences of $u_x$ that appear ordered in the cluster, we can implement the
above strategy in $O(k)$ time, where $k$ represents the size of the cluster; the test whether we
can correctly match the factors of $\alpha$ that do not contain $x$ to factors of $w$ is done using the
information gathered during the preprocessing. We process in this way all clusters having at
least as many elements as the number of repeated variables in the pattern. The total number
of elements in these clusters is at most $n - \ell$ so this procedure takes $O(n)$ time.

The time spent for each possible value for $\ell$ is $O(n)$. Hence, in total our algorithm needs
$O(n^2)$ time to decide whether there exists an assignment that maps the pattern to the word.

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 Lemma 2 on this pattern.

\begin{theorem} \label{thm:main}
The matching problem for $\text{PAT}^{\var\leq k}$ is solvable in $O\left(\frac{|w|2^{3k}}{(k-1)!}\right)$ time, where
\end{theorem}

$w$ is the input word.

We now consider the more involved case of patterns with a bounded scope coincidence
degree. The following combinatorial results are well known (see, e.g., [8]).

\begin{lemma} [8] \label{lem:primitive}
Let $u_1, u_2, u_3$ be primitive words, such that $|u_1| < |u_2| < |u_3|$ and $u^2_i$
are prefixes (suffixes) of a word $w$, for all $1 \leq i \leq 3$. Then $2|u_1| < |u_3|$. As a consequence,
we have $|(u|u \text{ primitive, } u^2 \text{ prefix (respectively, suffix) of } w)| \leq 2 \log |w|$.

Assume that $w \in \Sigma^*$ is of length $n$. For each $i \leq n$ we define the set
$P_i = \{u | u$ is a primitive word such that $u^2$ is a suffix of $w[1..i]\}.$
Lemma 4 shows that $|P_i| \leq 2 \log n$ for all $1 \leq i \leq n$. 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).

\begin{lemma} [8] \label{lem:prefix}
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 $w$, with $i \in \{1, 2, \ldots, n\}$.

Note that in [8] there are examples of words of length $n$ for which $\sum_{i \leq n} |P_i| \in \Theta(n \log n)$.

Next, we extend the results of Lemmas 4 and 5. Instead of primitively rooted squares,
we consider words of the form $uwv$ for some fixed word $v$, with $uv$ primitive (or, equivalently,
with $vu$ primitive). It is not hard to show the following lemma.

\begin{lemma} \label{lem:prefix}
For a fixed $v$, let $u_1 v u_1, u_2 v u_2, u_3 v u_3$ be prefixes (suffixes) of a word $w$ such that
$|u_1| < |u_2| < |u_3|$ and $u_i v$ are primitive for all $1 \leq i \leq 3$. Then $\frac{3|u_1|}{2} < |u_3|$. As a consequence,
we have $|(u|v u \text{ primitive, } v u \text{ prefix (respectively, suffix) of } w)| \in O(\log |w|)$.

Consider two words $w, v \in \Sigma^*$, with $|w| = n$. Following the case of the primitively rooted
squares, for each $i \leq n$ we define the set
$R_i^v = \{u |\\ w u v$ is a suffix of $w[1..i]$ with $w u v$ primitive}\}.

Again, $R_i^v$ can be stored efficiently by the lengths of the words it contains. Clearly,
$\sum_{i \leq n} |R_i^v| \in O(n \log n).$ Moreover, as $uwv$ with $uv$ primitive is just a primitively rooted
square when $v = e$, it follows that for certain values of $v$, we have that $\sum_{i \leq n} |R_i^v| \in \Theta(n \log n)$.

The following result extends in a non-trivial manner the result of Lemma 5.

\begin{lemma} \label{lem:prefix}
Given two words $w, v \in \Sigma^*$, with $|w| = n$, we can compute in $O(n \log n)$ time
all the sets $R_i^v$ associated to $w$, for $i \in \{1, 2, \ldots, n\}$.

We begin the high level description of the proof of this lemma with several preliminary
facts. Given the word $w$, its dictionary of basic factors [9] is a data structure that associates
Pattern Matching with Variables: Fast Algorithms and Hardness Results

labels to the factors of the form $w[i..i + 2^k - 1]$ (called basic factors), for $k \geq 0$ and $1 \leq i \leq n - 2^k + 1$, such that every two identical factors of the whole word 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. Looking deeper into the combinatorial structure of $w$ we note that a basic factor $w[i..i + 2^k - 1]$ occurs either at most twice in a factor $w[j..j + 2^{k+1} - 1]$ or the positions where $w[i..i + 2^k - 1]$ occurs in $w[j..j + 2^{k+1} - 1]$ form an arithmetic progression of ratio $per(w[i..i + 2^k - 1])$ (see [21]). Hence, the occurrences of $w[i..i + 2^k - 1]$ in $w[j..j + 2^{k+1} - 1]$ can be presented in a compact manner: either at most two positions, or the starting position of the progression and its ratio. Using this property and the dictionary of basic factors one can produce in $O(n \log n)$ a data structure that allows us to test the primitivity of each factor of $w$ in $O(1)$ time. Moreover, for a certain $v$, we can also produce in $O(n \log n)$ time a data structure answering the following type of queries in $O(1)$ time: “Given $i$ and $k$ return the compact representation of the occurrences of $w[i..i + 2^k - 1]$ in $w[i..i + 2^k - 1]$."

Returning to the proof of the lemma, after the above preprocessing we find all the occurrences of $v$ in $w$. Consider now one of these occurrences $w[i..j] = v$. We are searching all the prefixes $u$ of $w[j + 1..n]$ that are also suffixes of $w[1..i - 1]$ with $uv$ primitive. Checking naively each prefix of $w[j + 1..n]$ for these properties takes too long. Thus, we analyse simultaneously all the prefixes of $w[j + 1..n]$ that have the length between $2^k$ and $2^{k+1}$, for $0 \leq k \leq \log n$. All these factors share as common prefix the basic factors $x = w[j + 1..j + 2^k]$. Using the data structures we constructed, we retrieve in $O(1)$ time a compact representation of the occurrences of $x$ in $w[i..i + 2^k - 1]$. We know that every possible candidate for the factor $u$ we search for starts with one of these occurrences. Using a series of involved combinatorics on words insights, one can identify all the possible factors $u$ that start on such a position and fulfill also the requirement that $uv$ is primitive, in time proportional to their number. As in $w$ there are at most $O(n \log n)$ factors $uvu$ fulfilling our requirements with each uniquely identified by the corresponding occurrence of $v$, and, moreover, the time we spend in analysing each occurrence of $v$ is proportional to $\log n$ plus the number of valid factors $uvu$ centred around that occurrence of $v$, the result of Lemma 7 follows.

We are now ready to solve the matching problem for non-cross patterns.

\textbf{Theorem 8.} The matching problem for $\text{Pat}_{nc}$ is solvable in $O(|w|m \log |w|)$ time, where $w$ is the input word and $m$ is the number of one-variable blocks occurring in the pattern.
\(O(n \log n)\) time the positions \(j\) such that \(w_0\Pi_{k=1,i-1}(\alpha_kw_k)\alpha_i\) matches \(w[j, j']\), where \(p_i\) is the number of one-variable blocks of \(\alpha_i\). To do this, we have to analyse the structure of \(\alpha_i\).

The simplest case is when \(\alpha_i = x_i\). Then, \(w_0\Pi_{k=1,i-1}(\alpha_kw_k)\alpha_i\) matches \(w[1,j]\) if and only if there exists \(j' < j\) such that \(w_0\Pi_{k=1,i-1}(\alpha_kw_k)\alpha_i\) matches \(w[1,j']\); finding all such positions \(j\) takes \(O(n)\) time, so our claim holds in this case.

Consider next the case when \(\alpha_i = x_i^k\) with \(k \geq 2\). In a first phase, for each position \(j\) and each primitively rooted square suffix \(t^2\) of \(w[1,j]\) we check whether \(t^k\) is a suffix of \(w[1,j]\) and if \(w_0\Pi_{k=1,i-1}(\alpha_kw_k)\alpha_i\) matches \(w[1,j] - k|t|\); if both these checks are true, we conclude that \(w_0\Pi_{k=1,i-1}(\alpha_kw_k)\alpha_i\) matches \(w[1,j]\) when \(x_i\) is mapped to \(t\), and we store this information.

In this way, we found all the positions \(j\) such that \(w_0\Pi_{k=1,i-1}(\alpha_kw_k)\alpha_i\) matches \(w[1,j]\) when \(x_i\) is mapped to a primitive word; we just have to analyse the case when \(x_i\) is mapped to a non-primitive word. Now, for each position \(j\) (considered in increasing order) and each primitively rooted square suffix \(t^2\) of \(w[1,j]\), we check whether \(t^k\) is a suffix of \(w[1,j]\), and, differently from the previous case, if \(w_0\Pi_{k=1,i-1}(\alpha_kw_k)\alpha_i\) matches \(w[1,j] - k|t|\) such that \(x_i\) is mapped to a power of \(t\) in \(O(n\log n)\), so our claim holds in this case, as well.

Finally, we consider the more complicated case of \(\alpha_i\) containing at least one terminal. We let \(\alpha_i = x_i^0\Pi_{k=1,p_i}(\alpha_kw_k)x_i^{q_i}\) be the decomposition of \(\alpha_i\) in one-variable blocks. First, we assume that \(p_i = q_i = 1\), so \(\alpha_i\) ends with \(x_iw_{p_i}x_i\); it may be the case that \(x_i\) is mapped to a word \(u\) such that \(w_{p_i}u\) is primitive. Then, for each position \(j\), we consider all the suffixes \(w_{p_i}u\) of \(w[1,j]\) such that \(w_{p_i}u\) is primitive (these factors are in \(R^\alpha_{p_i, q_i}\) and all of them can be identified in \(O(n \log n)\) according to Lemma 7). For each such suffix, we determine the factor \(u\), the image of \(x_i\). Next, in \(O(p_i)\) time we check whether the image \(\gamma_i\) of \(\alpha_i\) under the substitution of \(x_i\) with \(u\) is a suffix of \(w[1,j]\). If so, then we check whether \(w_0\Pi_{k=1,i-1}(\alpha_kw_k)\alpha_i\) matches \(w[1,j] - |\gamma_i|\), and, if our check is again true, conclude that \(w_0\Pi_{k=1,i-1}(\alpha_kw_k)\alpha_i\) matches \(w[1,j]\) when \(x_i\) is mapped to the \(u\) determined above. Further, we look at the case when \(x_i\) is mapped to a word \(u\) such that \(w_{p_i}u\) is a repetition. For a position \(j\), we consider each primitively rooted square suffix \(t^2\) of \(w[1,j]\). We can determine in constant time the exponent \(r_0\) and the prefix \(t_0\) of \(t\) such that \(w_{p,i} = t^r_0t_0\); this means that \(u = t_1t^r_0\), where \(t_0t_1 = t\) but \(t_1\) is not known. Just like before, we can check easily the cases when \(r_1 \in \{0, 1\}\) (so, the value \(u\) to which \(x_i\) is mapped becomes fixed) in time \(O(p_i)\). In the following, let us assume that \(r_1 \geq 2\) and, for simplicity, take \(t_0 \neq x\); thus, \(t_1 \neq t\) and, as \(t_0\) is known, so is \(t_1\). If \(r_{p_{i-1}, i} \geq 2\), then the image \(u\) of \(x_i\) is uniquely determined: we just note that the word \(w_{p_{i-1}, i}u\) is \(|t|\)-periodic, and cannot be extended with \(|t|\) letters to the left without breaking the period, so we uniquely determine \(u\) by identifying the longest \(|t|\)-periodic suffix \(w'\) of \(w[1,j]\) and noting that \(w_{p_{i-1}, i}u\) is its longer suffix of the form \(t_1\{t\}^*\). Then, we can check again in \(O(p_i)\) whether \(\alpha_i\) matches the suffix of \(w[1,j]\) and continue just as we did before. Therefore, let us assume \(r_{p_{i-1}, i} = 1\); if \(w_{p_{i-1}, i} \notin \{t\}^*t_0\), then again we can determine the image of \(x_i\) by the same reasons, and we can continue similarly. This process continues in this manner, and we either get that the image of \(x_i\) is uniquely determined, or that \(\alpha_i = x_i\Pi_{k=1,p_i}(t^r_0t_0x_i)\), so the image of \(\alpha_i\) is \(|t|\)-periodic. Fortunately, the latter case can be solved in the same manner as the case when \(\alpha_i\) was just a repetition: we already know the positions \(j\) such
that $w_0 \Pi_{k=1,i-1}(\alpha_k w_k) \alpha_i$ matches $w[1..j]$ when $w_{p_i,i} x_i$ is mapped to a power of $t$ with lower exponent, so we just have to extend with powers of $t$ of exponent equal to the number of occurrences of $x_i$ in $\alpha_i$, as we did before. The case when $t_0 = \varepsilon$ is treated similarly: either the image of $x_i$ can be uniquely determined, or the image of both $x_i$ and all the factors $w_k$ are powers of $t$ and we can apply our previous dynamic programming approach. In the same manner, our last case, when $\ell_{p_i,i} \geq 2$ implies that either $x_i$ is mapped to a primitive word $t$ such that $t^2$ is a suffix of $w[1..j]$, or it is mapped to a power of such a word $t$. In both cases, an analysis similar to the above leads to the correct computation of all the positions $j$ such that $w_0 \Pi_{k=1,i-1}(\alpha_k w_k) \alpha_i$ matches $w[1..j]$. The total time needed for such an analysis is $O(n p_i \log n)$, as for each position $j$ and each $t \in P_j$ we need to do $O(p_i)$ steps, checking for each possible image of $x_i$ that $\alpha_i$ is mapped correctly to a suffix $w[j' + 1..j]$ of $w[1..j]$, where $w_0 \Pi_{k=1,i-1}(\alpha_k w_k)$ matched $w[1..j']$. Again, our claim holds.

It only remains to see that $\alpha$ matches to $w$ if there exists a position $j$ such that $w_0 \Pi_{k=1,i-1}(\alpha_k w_k) \alpha_i$ matches $w[1..j]$ and $w[j + 1..n] = w_r$. The total time is, clearly, $O(n \log n \sum_{i=1}^k p_i)$; summing up, the time complexity of our algorithm is $O(n m \log n)$.

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

\textbf{Theorem 9.} The matching problem for $\text{Pat}_{\text{scd} \leq k}$ is solvable in $O\left(\frac{|w|^{1/m}}{(k-1)^{1/2}}\right)$ time, where $w$ is the input word and $m$ is the number of one-variable blocks occurring in the pattern.

5 The Hardness of Factorising a Word into Distinct Factors

So far, we presented a series of upper bounds for the time needed to solve various matching problems. In this section, we prove the $NP$-completeness of the following problem.

**UnFact**

**Instance:** A word $w$ and an integer $k \geq 1$.

**Question:** Does there exist a unique factorisation of $w$ with size at least $k$?

As shall be explained later on, this has implications on the injective version of the matching problem, which can be solved in $O\left(\frac{|w|^{1/m}}{k^{1/2}}\right)$ ($k$ and $m$ are the numbers of variables and one-variable blocks, respectively), just as the general matching problem.

For the completeness result, we use the following as the base problem for our reduction.

**3D-Match**

**Instance:** An integer $\ell \in \mathbb{N}$ and a set $S \subseteq \{(p,q,r) \mid 1 \leq p < \ell + 1 \leq q < 2\ell + 1 \leq r \leq 3\ell\}$.

**Question:** Does there exist a subset $S'$ of $S$ with cardinality $\ell$ such that, for each two elements $(p,q,r), (p',q',r') \in S'$, $p \neq p'$, $q \neq q'$ and $r \neq r'$?

An instance of 3D-Match is a set $S$ of triples, the $3$ 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 $(S,\ell)$ is a selection of $\ell$ triples such that no two of them coincide in any component. Hence, for every $i \in \{1,2,3\}$, if we collect all the $i$th components of the $\ell$ 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 [15].

We define a mapping $g$ from 3D-Match to UnFact. Let $(S,\ell)$ be an instance of 3D-Match, where $\ell \in \mathbb{N}, S = \{s_1, s_2, \ldots, s_k\}$ with $s_i = (p_i, q_i, r_i), 1 \leq i \leq k$. Next, we construct a word $w$ over the alphabet $\Sigma = \{a, c_1, c_2, c_3, q_i, j, \#l, \#_0 \mid 1 \leq i \leq k, 1 \leq j \leq 4, 1 \leq l \leq 3\ell\}$. Let $v = v_1 v_2 \cdots v_k$, where, for every $1 \leq i \leq k$, $v_i = c_i p_i a b_{i,1} b_{i,2} q_i a b_{i,3} b_{i,4} r_i a S_i$. Furthermore, we define $\hat{u} = 1 \#_1 \cdots \#_{3\ell-2} (3\ell-1) \#_{3\ell-1} (3\ell) \#_{3\ell}$ and $\vec{u} = \vec{u}_1 \vec{u}_2 \cdots \vec{u}_k$, where,
for every $1 \leq i \leq k$, $\overline{u_i} = b_{i,1} \%_{i,1} b_{i,2} \%_{i,2} b_{i,3} \%_{i,3} b_{i,4} \%_{i,4}$. Finally, $u = a \#_0 \overline{u}$, $w = uv$, $\ell = 7\ell + 6(k - \ell) + |u|$ and $g(S, \ell) = (w, \ell)$. This concludes the definition of the mapping $g$.

In the following, let $(S, \ell)$ be a fixed instance of 3D-MATCH and $(w, \ell) = g(S, \ell)$.

We now explain the mapping $g$ in an intuitive way. Every triple $s_i = (p_i, q_i, r_i)$ of $S$ is represented by $v_i = \epsilon_i p_i a b_i, 1 b_i, 2 q_i a b_i, 3 b_i, 4 r_i a \#_i$, where the factors $p_i a$, $q_i a$ and $r_i a$ represent the single components. Each of the remaining symbols $\epsilon_i$, $\#_i$, $b_i, j, 1 \leq j \leq 4$, has exactly one occurrence in $w$; thus, every factor that contains one of these will necessarily be distinct. Hence, the factors $p_i a$, $q_i a$ and $r_i a$ of the unsafe factorisation may also occur in the unsafe factorisation of some $v_j$, $i \neq j$, 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 $v_i, 1 \leq i \leq k$. The factorisation $e_i p_i | a b_i, 1 | b_i, 2 q_i | a b_i, 3 | b_i, 4 r_i | a \#_i$ is called safe and the factorisation $e_i | p_i a | b_i, 1 b_i, 2 | q_i a | b_i, 3 b_i, 4 | r_i a | \#_i$ is called unsafe. The safe factorisation contains only distinct factors, whereas the factors $p_i a$, $q_i a$ and $r_i a$ of the unsafe factorisation may also occur in the unsafe factorisation of some $v_j$; 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 $\{s_{t_1}, s_{t_2}, \ldots, s_{t_s}\}$ is a solution of $(S, \ell)$, then we can factorise all $v_i$, $1 \leq i \leq \ell$, into the unsafe factorisation, all other $v_j$, $j \notin \{t_1, t_2, \ldots, t_s\}$, into the safe factorisation and the prefix $u$ into $|u|$ individual factors. This yields a factorisation of $w$ with $|u| + 7\ell + 6(k - \ell) = \ell$ factors and its uniqueness follows from the fact that $\{s_{t_1}, s_{t_2}, \ldots, s_{t_s}\}$ is a solution of $(S, \ell)$ and that the symbols from $u$ do not occur as single factors in $v$.

\textbf{Lemma 10.} If $(S, \ell)$ has a solution, then there is a unique factorisation of $w$ with size $\ell$.

Proving the converse of Lemma 10 is more difficult. The idea is to first show that if there exists a unique factorisation of $w$ of size $\ell$, then there also exists one with at least the same size and the following properties: (1) no factor overlaps the boundaries between $u$ and $v$ or between some $v_i$ and $v_{i+1}$, $1 \leq i \leq k - 1$. (2) $u$ is split into $|u|$ factors. Property (1) is easily 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 $w$.

\textbf{Lemma 11.} If $w$ has a unique factorisation $f$ with size $\ell$, then, for some $\ell' \geq \ell$, there exists a unique factorisation $f'$ of $w$ of size $\ell'$, such that no $f'$-factor overlaps positions $|u|$ and $|u| + 1$ or positions $|w_1v_2 \cdots v_i|$ and $|w_1v_2 \cdots v_i| + 1$, for some $i$, $1 \leq i \leq k - 1$.

Property (2) requires a more careful argument. If $u$ is not split into $|u|$ factors, then in $u$ there exists a factor $x\pi$, where $x$ is a single symbol and $\pi$ is some non-empty factor, and $x\pi$ is also a factor of the factorisation (with Lemma 11 we can assume that $x\pi$ lies inside of $u$). If $|\pi| \geq 2$, then we cut off $x$, which results in two factors $x$ and $\pi$. Since $|\pi| \geq 2$, the factor $\pi$ must contain a symbol with only one occurrence in $w$, which means that it is not repeated. If $x$ is repeated, then this can only happen in some $v_i$ and we can now show that the factor $x$ must have a neighbour in $v_i$ that starts with a symbol $y \in \{b_{i,1}, b_{i,2}, b_{i,3}, b_{i,4}, \epsilon_i, \#_i\}$. We now simply append $x$ to this neighbour. If $y \in \{\epsilon_i, \#_i\}$, then the factor is distinct since $\epsilon_i$ and $\#_i$ have only one occurrence in $w$. If, on the other hand, $y \in \{b_{i,1}, b_{i,2}, b_{i,3}, b_{i,4}\}$, then this new factor can only be repeated in $w$; but all factors in $u$ of size at least 2 contain a symbol that does not occur in $u$, thus, the newly constructed factor is distinct. If $|\pi| = 1$, then the situation is easier, since we can simply cut $x\pi$ into $x$ and $\pi$ and if one of these new factors is repeated, then we can append it to its other neighbour without producing a repeated factor.

\textbf{Lemma 12.} If $w$ has a unique factorisation $f$ of size $\ell$, then, for some $\ell' \geq \ell$, $w$ has a unique factorisation $f'$ of size $\ell'$, such that every single symbol of $u$ is an $f'$-factor.
Finally, we show the converse of Lemma 10. If there is a unique factorisation \( f \) of \( w \) of size \( \hat{\ell} \), then we can assume that it is of the form ensured by Lemmas 11 and 12. We can further conclude that if a single symbol of some \( v_i \) is a factor of \( f \), then it is \( e_i \) or \( $i$, since otherwise it would be repeated in \( u \). In particular, this means that no \( v_i \) can be split into more than 7 factors and if a \( v_i \) is split in exactly 7 factors, this then must be the safe factorisation defined above. Now if \( f \) splits \( T \) of the \( v_i \), \( 1 \leq i \leq k \), into 7 factors and the remaining \( k - T \) of the \( v_i \), \( 1 \leq i \leq k \), into 6 or less factors, then \( f \) splits \( w \) into at most \( |u| + TF + 6(k - T) \) factors. Since \( \ell = 7\ell + 6(\ell - \ell) + |u| \leq |u| + TF + 6(k - T) \) must be satisfied, we can conclude \( \ell \leq T \), which means that at least \( \ell \) factors \( v_i \) are factorised into the safe factorisation. This directly implies that the corresponding \( \ell \) triples from \( S \) constitute a solution for \((S, \ell)\).

Since the reduction is clearly polynomial, the main result, i.e., Theorem 14, follows.

**Theorem 14.** \( \text{UnFact} \) is \( \mathcal{NP} \)-complete.

We note that if a word has a unique factorisation of size \( k \), then it also has a unique factorisation of size \( k' \) for all \( 1 \leq k' \leq k \). 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. In particular, this means that \((w,k)\) is a positive \(\text{UnFact} \) instance if and only if \(x_1x_2\cdots x_k \) matches \( w \) in an injective way; thus, we can conclude that \( \text{inj-Match} \) is \( \mathcal{NP} \)-complete for many classes of patterns for which its non-injective version can be easily solved in polynomial time.

**Corollary 15.** \( \text{inj-Match} \) is \( \mathcal{NP} \)-complete for \( \text{Pat}_{\text{reg}}, \text{Pat}_{\text{nc}}, \text{Pat}_{\text{var} \leq k}, \text{Pat}_{\text{scd} \leq k}, k \geq 1 \).

We wish to point out that our proof of Theorem 14 requires an unbounded alphabet and it is open whether \(\text{UnFact} \) is \( \mathcal{NP} \)-complete for fixed alphabets.\(^3\) Consequently, it does not imply that the injective matching problem for the classes of patterns mentioned in Corollary 15 is still \(\mathcal{NP} \)-complete if the alphabet is fixed. However, for the injective matching problem with a fixed alphabet, we can show a similar, but slightly weaker, result:

**Theorem 16.** \( \text{inj-Match} \) is \( \mathcal{NP} \)-complete for \( \text{Pat}_{\text{nc}} \) and \( \text{Pat}_{\text{scd} \leq k}, k \geq 1, \) if the alphabet is constant.

Theorem 16 can also be proved by a reduction from 3D-Match. We shall give a definition of this reduction, but omit the proof of its correctness, and leave it to the reader.

Let \((S, \ell)\) be an instance of 3D-Match, where \( \ell \in \mathbb{N} \), \( S = \{s_1, s_2, \ldots, s_k\} \) with \( s_i = (p_i, q_i, r_i) \), \( 1 \leq i \leq k \). We define a word \( w \) over the alphabet \( \Sigma = \{a, b, \$, $, \#\} \) and a pattern \( \alpha \) which uses the variables \( x_{i,j}, 1 \leq i \leq \ell, 1 \leq j \leq 3, \) and \( y_1, z_1, 1 \leq i \leq \ell - 1, 1 \leq j \leq 2\ell + 2 \). We first define the factors \( \beta_i = x_{\ell}^{2\ell+2}z_1^{2\ell+2}, 1 \leq i \leq \ell, \)

\[
\alpha = (a^m b)^2 (a^m b)^2 (a^m b)^2, 1 \leq i \leq k, \quad \text{and} \quad \# = (\#^m \#^m \#^m \#^m \#^m), 1 \leq i \leq 2k + 2, \quad \text{where} \ m = \max \{2k + 2, 3\ell\} + 1.
\]

Then, in order to form \( \alpha \) and \( w \), these factors are combined as follows:

\[
\begin{align*}
\alpha &= \left( z_1^m y_1 z_2^m \beta_1 z_3^m y_2 z_4^m \beta_2 z_5^m y_3 z_6^m \beta_3 \cdots \beta_k z_{2\ell+1} y_{\ell+1} z_{2\ell+2}^m \right) \\
&= (\#^m \#^m \#^m \#^m \#^m) \left( \#^m \#^m \#^m \#^m \#^m \right) \cdots \left( \#^m \#^m \#^m \#^m \#^m \right) \left( \#^m \#^m \#^m \#^m \#^m \right).
\end{align*}
\]

A collection \( \{s_{i_1}, s_{i_2}, \ldots, s_{i_{\ell}}\} \) of \( \ell \) elements from \( S \) translates into a substitution \( h \) with \( h(\alpha) = w \) as follows. For every \( 1 \leq i \leq \ell \), the factor \( z_{\ell}^m \beta_i z_{\ell+1}^m \) is mapped to \( \#_{\ell+1} \beta_i \#_{2\ell+1} \), and the variables \( y_1, 1 \leq i \leq \ell - 1, \) with only one occurrence, are mapped to the remaining factors in between. Furthermore, it can be shown that if \( \{s_{i_1}, s_{i_2}, \ldots, s_{i_{\ell}}\} \) is a solution for \((S, \ell)\), then \( h \) is injective. Proving the other direction is more difficult and requires a lemma which states that any substitution \( h \) with \( h(\alpha) = w \) necessarily maps every \( \beta_i \) to some \( u_j \).

---

\(^3\) As shown in [7], the variant where we require the factorisation to have short factors instead of a large size is \(\mathcal{NP} \)-complete also for fixed alphabets.
References


