Testing k-binomial equivalence*

Dominik D. Freydenberger[1], Pawel Gawrychowski[2], Juhani Karhumäki[3], Florin Manea[4], and Wojciech Rytter[2]

[1] Institute for Computer Science, University of Bayreuth, Germany
ddfy@ddfy.de
[2] Institute of Informatics, University of Warsaw, Poland,
{gawry,rytter}@mimuw.edu.pl
[3] Department of Mathematics and Statistics, University of Turku, Finland
karhumak@utu.fi
[4] Department of Computer Science, Kiel University, Germany
flm@informatik.uni-kiel.de

Abstract. Two words w_1 and w_2 are said to be k-binomial equivalent if every non-empty word x of length at most k over the alphabet of w_1 and w_2 appears as a scattered factor of w_1 exactly as many times as it appears as a scattered factor of w_2. We give two different polynomial-time algorithms testing the k-binomial equivalence of two words. The first one is deterministic (but the degree of the corresponding polynomial is too high) and the second one is randomised (it is more direct and more efficient). These are the first known algorithms for the problem which run in polynomial time.

1 Introduction

An *alphabet* is a finite and non-empty set of symbols (also called letters). Any finite sequence of symbols from an alphabet Σ is called a *word* over Σ. The set of all words over V is denoted by Σ^* and the *empty word* is denoted by ϵ; also Σ^+ is the set of non-empty words over Σ, Σ^k is the set of all words over Σ of length exactly k, while $\Sigma^{\leq k}$ is the set of all words over Σ of length at most k. Given a word w over an alphabet Σ, we denote by $|w|$ its length; for some $1 \leq i \leq |w|$ we denote the i-th letter of w by $w[i]$. We also denote the factor that starts with the i-th letter and ends with the j-th letter in w by $w[i..j]$. For $w, x \in \Sigma^*$ we denote by $|w|_x$ the number of distinct occurrences of x as a factor of w.

A *scattered factor* of $w \in \Sigma^*$ is a word $w[i_1] \cdots w[i_k]$ for some $k \geq 1$ such that $i_j < i_{j+1}$ for all $1 \leq j \leq k-1$. The *binomial coefficient* of u and v, denoted $\binom{u}{v}$, equals the number of occurrences of v as a scattered factor of u. Clearly, for $a \in \Sigma$ we have $\binom{u}{a} = |u|_a$, while for $x \in \Sigma^+$ with $|x| \geq 2$ it is not necessary that $|u|_x = \binom{u}{x}$.

* The results presented in this paper were partly obtained during the Dagstuhl seminar 14111, in March 2014. Dominik Freydenberger was supported by the DFG grant FR 3551/1-1. Juhani Karhumäki was supported by Academy of Finland under the grant 257857. Florin Manea was supported by the DFG grant 596676. Wojciech Rytter was supported by the grant NCN2014/13/B/ST6/00770 of the Polish Science Center.

Example 1. If $u = bbaa$ and $v = ba$ we have $\binom{u}{v} = \binom{bbaa}{ba} = 4$, as $u[1]u[3] = u[2]u[3] = u[1]u[4] = u[2]u[4] = ba$; clearly, $|u|_{ba} = 1$.

For more details regarding these binomial coefficients see Chapter 6, by Sakarovitch and Simon, from the handbook [8].

A well known equivalence relation between words is that of abelian equivalence. Two words $w_1, w_2 \in \Sigma^*$ are said to be *abelian equivalent* if for all $a \in \Sigma$ we have $|w_1|_a = |w_2|_a$; equivalently, w_1 and w_2 are abelian equivalent if they have the same Parikh vector, thus being permutations of each other. This relation was extended in [7] (see also [6]), where the *k-abelian equivalence* relation was defined. Two words $w_1, w_2 \in \Sigma^*$ are said to be *k-abelian equivalent* if for all $x \in \Sigma^{\leq k}$ we have $|w_1|_x = |w_2|_x$. Obviously, the 1-abelian equivalence relation is the same as the abelian equivalence.

As $|w_1|_a = \binom{w_1}{a}$, another way to generalise the abelian equivalence relation is to define the *k-binomial equivalence* (see the conference paper [12], as well as its journal version [11]). Two words $w_1, w_2 \in \Sigma^*$ are said to be *k-binomial equivalent* if for all $x \in \Sigma^{\leq k}$ we have $\binom{w_1}{x} = \binom{w_2}{x}$; if w_1 and w_2 are k-binomial equivalent, we write $w_1 \equiv_k w_2$. Again, it is easy to see that the 1-binomial equivalence is the same as the abelian equivalence. Combinatorial properties of the k-binomial equivalence relation are studied in [12, 11, 10].

Recently, in [3, 4] a series of algorithmic results regarding the k-abelian equivalence were shown. As a basic result, it was shown that one can test whether two words are k-abelian equivalent in linear time. Therefore, it seems natural to us to study a similar problem in the context of k-binomial equivalence. That is, we are interested in the following problem.

Problem 1. Given $w_1, w_2 \in \Sigma^*$, with $|w_1| = |w_2| = n$, and $k \leq n$, decide whether $w_1 \equiv_k w_2$.

Our main result shows that Problem 1 can be solved in polynomial time. The proof of this result uses a series of known results from the theory of finite automata, which does not exploit in any way the properties of k-binomial equivalence. Moreover, the degree of the polynomial characterising the time complexity of this algorithm is rather high, so we do not give it explicitly. Instead, we also show a simpler and much more direct Monte-Carlo algorithm solving the same problem. Our solutions assume a basic understanding of formal languages and automata theory; for more details, see [13] and [14].

Before moving to the main sections of this paper, we just point out that the complexity results we show here hold in the unit-cost RAM with logarithmic size memory word. In this model (which is generally used in the analysis of algorithms) we assume that, if the size of the input is n (e.g., we are given a word of length n), each memory cell can store $\mathcal{O}(\log n)$ bits, or, in other words, that *the machine word size* is $\mathcal{O}(\log n)$. The instructions are executed one after another, with no concurrent operations. The model contains common instructions: arithmetic (add, subtract, multiply, divide, remainder, shifts and bitwise operations, equality testing, etc.), data movement (indirect addressing, load the content of a memory cell, store a number in a memory cell, copy the content of a memory

cell to another), and control (conditional and unconditional branch, subroutine call and return). Each such instruction takes a constant amount of time. This model allows measurin g the number of instructions executed in an algorithm, making abstraction of the time spent to execute each of the basic instructions.

2 A polynomial deterministic algorithm

The first step we take towards solving Problem 1 is to construct, for a word w, a non-deterministic finite automaton A_w that accepts exactly the scattered factors of length at most k of w and, moreover, has exactly $\binom{w}{x}$ paths labelled with the scattered factor x of w.

Let us assume that $|w| = n$; then A_w has $nk + 2$ states; these states are

$$Q_w = \{(0,0)\} \cup \{(i,j) \mid 1 \leq i \leq n, 1 \leq j \leq k\} \cup \{(n+1, k+1)\}.$$

The initial state of the automaton is $(0,0)$, while every state (i,j) with $0 < j \leq k$ and $i \geq j$ is final. The state $(n+1, k+1)$ is an error state; this state and the initial state are the only states that are not final.

We define the transition function δ_w for all $(i,j) \in Q_w$ and all $a \in \Sigma$ by

$$\delta_w\left((i,j), a\right) = \begin{cases} \{(\ell, j+1) \in Q_w \mid \ell > i, w[\ell] = a\} & \text{if this set is non-empty,} \\ \{(n+1, k+1)\} & \text{otherwise.} \end{cases}$$

See Figure 1 for an illustration. An immediate consequence of this definition is that $\delta_w((n+1, k+1), a) = \{(n+1, k+1)\}$ holds for all $a \in \Sigma$.

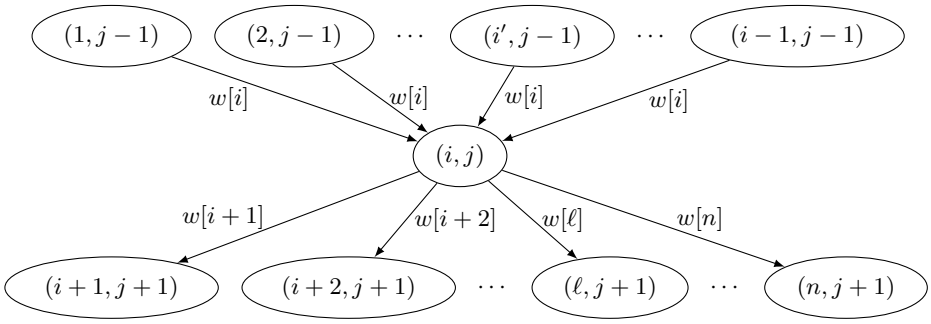

Fig. 1. The definition of the transition function: all the transitions leaving (i,j) and leading to a non-error state, as well as all transitions going to (i,j). We have $(\ell, j+1) \in \delta_w((i,j), w[\ell])$, with $i < \ell \leq n$ and $j < k$.

It is not hard to see that A_w accepts exactly the words $w[i_1] \cdots w[i_{k'}]$ with $k' \leq k$ and $i_1 < \ldots < i_{k'}$. Indeed, to accept such a word the automaton starts

in the state $(0, 0)$, and then goes through the states

$$(i_1, 1), (i_2, 2), \ldots, (i_j, j), \ldots, (i_{k'}, k');$$

as $1 \leq i_1 < \ldots < i_{k'}$ it is clear that $i_{k'} \geq k'$, so the state reached by the automaton is an accepting one. For the reverse implication, assume that the word x is accepted by A_w on the path formed by the states

$$(0, 0), (i_1, 1), (i_2, 2), \ldots, (i_j, j), \ldots, (i_{k'}, k').$$

By the definition of A_w we immediately get that $i_j < i_{j+1}$ for all $1 \leq j \leq k' - 1$; also, $i_1 > 0$. Thus, $i_j > j$ $1 \leq j \leq k'$. Moreover, each transition ending in (i_j, j) is labelled with $w[i_j]$, so $x = w[i_1] \cdots w[i_{k'}]$ is a scattered factor of w.

Finally, the argument above shows that there is a bijective correspondence between the sequences of indices defining the scattered factors of length at most k of w and the paths of A_w. In conclusion, A_w accepts the set of scattered factors of length at most k of w and, moreover, has exactly as many paths labelled with the scattered factor x of w as the total number of occurrences of x as a scattered factor of w (i.e., $\binom{w}{x}$).

Before coming back to the solution Problem 1, we recall that two non-deterministic finite automata are said to be *path-equivalent* if for each word x the number of distinct accepting paths labelled with x of A_1 equals the number of distinct accepting paths labelled with x of A_2, or both are infinite.

In our problem, we were given w_1 and w_2 and wanted to test whether $w_1 \equiv_k w_2$. By the above, it is enough to construct A_{w_1} and A_{w_2} and test whether A_{w_1} and A_{w_2} are path-equivalent. The latter property is decidable (see [15, 14] and the references within for a discussion on this problem and its complexity).

In the following, we show that this algorithm runs in polynomial time in our model of computation. The construction of the two automata A_{w_1} and A_{w_2} takes $\mathcal{O}(nk)$ time. Moreover, as none of A_{w_1} and A_{w_2} has transitions labelled with ϵ, it follows from [15] that there is an algorithm deciding the path-equivalence of A_{w_1} and A_{w_2} that runs in polynomial time with respect to the size of these automata (so, essentially, with respect to nk). Note that the algorithm presented in [15] is only shown to run in polynomial time in a computational model where it is assumed that the arithmetic operations between any (no matter how big) rational numbers can be done in constant time. To show that the algorithm still runs in polynomial time in our model of computation, we need to go further into details.

Basically, the algorithm of [15] applied to the two automata we constructed either decides that A_{w_1} and A_{w_2} identifies the lexicographically first word x such that A_{w_1} has a different number of accepting paths labelled with x than A_{w_2}. To do this, the algorithm explores the set of words from Σ^* in lexicographical order; it maintains a list of words V and for each $v \in V$ the array $P(v)$ storing the number of accepting paths in A_{w_1} and A_{w_2} (that is, an array storing for each final state of the two automata, how many paths labelled with v connect the initial state of the respective automata to that final state). If the list V contains at some moment the words $x_1, \ldots, x_\ell$ and the new considered word is x, the algorithm

checks if the array $P(x)$ is linearly independent from $P(x_1), \ldots, P(x_\ell)$. If yes, x is added to V and the algorithm further tries all words xa with $a \in \Sigma$. If no, the algorithm stops trying any other word that has x as a prefix. In [15] it is shown that only a polynomial number of words should be tried in this process, since V may contain up to $2nk$ words (as many words as the number of final states of the two automata). In our particular case, it is clear that all words that are longer than k are not accepted by any of our automata (i.e., the array $P(x)$ of some x longer than k contains only 0s); so, essentially, our algorithm will only try words of length at most $k+1$. Each such word x that is accepted by one of our automata is accepted on at most n^ℓ paths, where $\ell \leq k$ is the length of x, in total. So, its array $P(x)$ can be stored in at most $\mathcal{O}(nk^2)$ memory words (that is, k memory words for each final state, or, in other words, k memory words for each component of the array). At each step of the algorithm, we test whether the newly considered x produces an array $P(x)$ linearly independent from the arrays $P(y)$ with $y \in V$; since all these arrays contain only words that can be stored on k memory words, this test can be done in polynomial time. Indeed, if we use either a Gaussian elimination method or a modular method, such a test can be implemented in polynomial time (see, e.g., [1] and the references within, as well as [5]). Finally, the algorithm just checks whether there exists a word x in V which is accepted on a different number of paths in w_1 than in w_2. Again, this clearly takes polynomial time.

This concludes our analysis. We do not go into details and compute the exact complexity of the algorithm described above: we just state that it runs in polynomial time. While the preprocessing phase in which A_{w_1} and A_{w_2} are constructed is rather simple, computing the complexity of the algorithm from [15] requires really going into the implementation details of each step (for instance, testing the linear independency of the arrays), and this is not our purpose. We just note that the exponent of n in the complexity of this algorithm is at least 3 (in other words, the algorithm is at least cubic in n). The main result of our paper is, thus, the following theorem.

Theorem 1. *Problem 1 can be solved in polynomial time.*

Although based on a rather simple idea (the construction of the two automata), the algorithm presented in this section has a drawback: the main part of the computation is hidden in the algorithm checking the path equivalence of these two automata. Accordingly, in the following section we present a direct and more efficient randomised algorithm testing the k-binomial equivalence of two words.

3 A Monte-Carlo algorithm

We begin with a series of prerequisites. The first one is a folklore result; although it is really well known, we give a short sketch of the proof for completeness. In the following polylog means $\mathcal{O}((\log n)^c)$ for some constant c.

Lemma 1. *We can generate $p \in [N, 2N]$ in $\mathcal{O}(\text{polylog}(N))$ time, so that the probability that p is prime is at least $1 - \frac{1}{N}$.*

Proof. We recall that testing whether a number p is prime can be done in $\tilde{\mathcal{O}}((\log p)^{12.5})$ time, using the AKS algorithm [2].

Let $\pi(n)$ denote the number of primes smaller than n. It is well-known that $\lim_{n\to\infty} \frac{\pi(n)}{n/\ln n} = 1$. So, for n large enough we can safely assume that $\pi(n) \approx \frac{n}{\ln n}$. Now, the probability that a number chosen at random between N and $2N$ is prime can be approximated as $\frac{\frac{2N}{\ln 2N} - \frac{N}{\ln N}}{N} \approx \frac{1}{\ln N}$. It can be shown that if we try $\mathcal{O}(f(N) \ln n)$ numbers between N and $2N$, then the probability that none of them is prime is approximatively $e^{-f(N)}$; hence, for $f(N) = \ln(2N)$, we have that the error probability is approximatively $\frac{1}{2N}$ (so upper bounded by $\frac{1}{N}$).

Thus, the algorithm generating a prime between N and $2N$ works as follows. For $\mathcal{O}(\ln(2N))$ times, we pick at random a number between N and $2N$ (which takes about $\mathcal{O}(\log N)$ time, as we have to randomly pick $\log N$ bits) and test its primality using the AKS test. If we find a prime, we store it and return it at the end. The total time complexity of this approach is $\tilde{\mathcal{O}}((\log N)^{14.5})$ and the probability it returns a prime is at least $1 - \frac{1}{N}$.

In fact, the complexity of the algorithm can be decreased quite a lot, while preserving the error probability, by using the Rabin-Miller probabilistic primality test (see [9]) instead of the AKS algorithm. In this approach the algorithm can err also at every primality test, but trying enough numbers guarantees that the error probability is still less than $\frac{1}{N}$. □

The second auxiliary result is a particular case of the Schwartz-Zippel lemma. For a prime number p, let $\mathbb{F}_p$ denote the finite field (also known as Galois field) with p elements, consisting of the integers modulo p. It is well known that a non-zero polynomial $Q \in \mathbb{F}_p[X]$ of degree d has at most d distinct roots in $\mathbb{F}_p$. Thus, the following trivially holds:

Lemma 2. *Let Q be a non-zero polynomial of degree d over the field $\mathbb{F}_p$. Then, the probability that a randomly chosen $x \in \mathbb{F}_p$ is a root of Q is at most $\frac{d}{p}$:*

$$\Pr_{x \in \mathbb{F}_p}[Q(x) = 0] \leq \frac{d}{p}.$$

We now continue with the main part of this section.

For $v \in \{0,1\}^+$ let $bin(v)$ be the number whose binary representation is v. We define the crucial polynomial:

$$Q_{k,w}(x) = \sum_{|v| \leq k} \binom{w}{v} x^{bin(1v)}.$$

Example 2.

$$\begin{aligned} Q_{2,0010}(x) = &\binom{0010}{0} x^{bin(10)} + \binom{0010}{1} x^{bin(11)} + \binom{0010}{00} x^{bin(100)} + \\ &\binom{0010}{01} x^{bin(101)} + \binom{0010}{10} x^{bin(110)} + \binom{0010}{11} x^{bin(111)} \\ &= 3x^2 + x^3 + 3x^4 + x^5 + x^6 \end{aligned}$$

Clearly the powers of the variable x encode uniquely the scattered factors of the word w, consequently:

Observation 1 *$w_1 \equiv_k w_2$ if and only if $Q_{k,w_1} \equiv Q_{k,w_2}$.*

By definition, for any word w with $|w| \geq k$, the degree of $Q_{k,w}$ is $2^{k+1}-1$, so we cannot afford (time-wise) to construct it explicitly for any of the words w_1 and w_2, as enumerating the coefficients of such a polynomial would take exponential time. But, by Lemma 2, to check whether $Q_{k,w_1} \equiv Q_{k,w_2}$ (or, alternatively, $Q_{k,w_1} - Q_{k,w_2} \equiv 0$) holds with high probability, it is enough to check whether $Q_{k,w_1}(x) - Q_{k,w_2}(x) = 0$ for some randomly chosen $x \in \mathbb{F}_p$.

So, what we should see now is how to compute efficiently $Q_{k,w}(x)$; this is solved in the next lemma, where we show how this value is computed in $\mathcal{O}(k^2 n)$ for some word w of length n. Note that even though we choose p which is exponential in k (between n^k and $2n^k$), its binary representation needs $\mathcal{O}(k \log n)$ bits, that is $\mathcal{O}(k)$ memory words. Thus, operating on the potentially large numbers of $\mathbb{F}_p$ is not a bottleneck in achieving a polynomial time algorithm: each operation takes $\mathcal{O}(k)$ time in our model. The bottleneck is that we cannot construct $Q_{k,w}$ explicitly, so we need to go around this in order to compute the value we need, namely $Q_{k,w}(x)$.

Lemma 3. *For a word word w of length n, the value $Q_{k,w}(x)$ in the field $\mathbb{F}_p$ can be computed in $\mathcal{O}(k^2 n)$ time.*

Proof. We use an auxiliary polynomial. Let

$$Q'_{t,w}(x) = \sum_{|v|=t} \binom{w}{v} x^{bin(v)}.$$

We make the convention that $Q'_{0,w}(x) = 1$ and also note that, as no factor of length k occurs in the empty word, we have $Q'_{k,\epsilon}(x) = 0$

It is enough to compute the polynomials Q' since

$$\sum_{t=1}^{k} x^{2^t} Q'_{t,w}(x) = \sum_{t=1}^{k} \left(\sum_{|v|=t} \binom{w}{v} x^{bin(1v)} \right) = Q_{k,w}(x).$$

We use dynamic programming to compute all $Q'_{k',w'}(x)$, where $1 \leq k' \leq k$ and w' is a suffix of w. We denote:

$$T[k', i] = Q'_{k',w[i..n]}(x).$$

Every such $T[k', i]$ is computed just once and in a constant number of operations in $\mathbb{F}_p$, each taking $\mathcal{O}(k)$ time. Also, we compute (in $\mathbb{F}_p$) and store all the numbers $x^{2^{k'}}$ for $1 \leq k' \leq k$, which are then used to compute $Q_{k,w}(x)$, as shown below.

Indeed, once the array $T[\cdot, \cdot]$ is computed, we just have to return

$$Q_{k,w}(x) = \sum_{k'=1}^{k} x^{2^{k'}} T[k', 1].$$

This final computation can be done in $\mathcal{O}(k^3)$ time. Hence the claimed overall complexity follows.

First, we claim that the following recurrence holds:

$$T[k', i] = \begin{cases} 1 & \text{if } k' = 0 \\ 0 & \text{if } k' > 0 \text{ and } i = n+1 \\ T[k', i+1] + T[k'-1, i+1] & \text{if } k' > 0 \text{ and } i \leq n \text{ and } w[i] = 0 \\ T[k', i+1] + T[k'-1, i+1]x^{2^{k'}} & \text{if } k' > 0 \text{ and } i \leq n \text{ and } w[i] = 1 \end{cases}$$

Now, it is clear that the above recurrence gives us $T[0, i] = Q'_{0,w[i..n]}(x)$ and, as $w[n+1..n] = \epsilon$ by convention, $T[k, n+1] = Q'_{k,w[n+1..n]}(x)$.

We now show why our claim holds in the other cases by considering i from n down to 1; we proceed by induction on $n+1-i$. That is, we assume that $T[\cdot, \cdot]$ is defined by the above recurrence and also that $T[k', j] = Q'_{k',w[j..n]}(x)$ for all $k' > 0$ and j such that $n+1 \geq j > i$. We show that $T[k', i] = Q'_{k',w[i..n]}(x)$ for all $k' > 0$.

For our proof, we see each $T[k'][j]$, with $j \geq i$, as a polynomial in x. Then, we compute for each $\ell \leq 2^{k'} - 1$ the coefficient of x^ℓ in $Q'_{k',w[i..n]}(x)$ and show it is equal to the coefficient of x^ℓ in $T[k', i]$ (as obtained from the recurrence). In the following, let v_ℓ denote the k'-bits binary expansion of ℓ (i.e., $|v_\ell| = k$ and $bin(v_\ell) = \ell$).

In the first case we analyse, let $w[i] = 0$.

If v_ℓ has the most significant bit equal to 1, then the number of occurrences of v_ℓ in $w[i..n]$ (i.e., the coefficient of x^ℓ in $Q'_{k',w[i..n]}(x)$) equals the number of occurrences of v_ℓ in $w[i+1..n]$, which is the coefficient of x^ℓ in $Q'_{k',w[i+1..n]}(x)$. Moreover, we have $\ell \geq 2^{k'-1}$, so the coefficient of x^ℓ in $Q_{k'-1,w[j..n]}(x)$ is 0, for all $j > i$. Consequently, the coefficient of x^ℓ in $Q'_{k',w[i..n]}(x)$ equals the sum of the coefficient of x^ℓ in $Q'_{k',w[i+1..n]}(x)$ (which, by induction, equals the coefficient of x^ℓ in $T[k', i+1]$) and the coefficient of x^ℓ in $Q'_{k'-1,w[i+1..n]}(x)$ (which is 0 and, by induction, equals the coefficient of x^ℓ in $T[k'-1, i+1]$).

If v_ℓ has the most significant bit 0 then $bin(v_\ell) = bin(v_\ell[2..k']) = \ell < 2^{k'}$. Now, the number of occurrences of v_ℓ in $w[i..n]$ equals the number of occurrences of v_ℓ in $w[i+1..n]$ plus the number of occurrences of $v_\ell[2..k']$ in $w[i+1..n]$ (which is the coefficient of $x^{bin(v_\ell[2..k'])} = x^\ell$ in $Q'_{k'-1,w[i+1..n]}$). So, again, the

coefficient of x^ℓ in $Q'_{k',w[i..n]}(x)$ equals the coefficient of x^ℓ in $Q'_{k',w[i+1..n]}(x)$ (which, by induction, equals the coefficient of x in $T[k', i+1]$) plus the coefficient of x^ℓ in $Q'_{k'-1,w[i+1..n]}(x)$ (which, by induction, equals the coefficient of x^ℓ in $T[k'-1, i+1]$).

This shows that, indeed, if $0 < k', i \leq n$, and $w[i] = 0$ we have $Q'_{k',w[i..n]}(x) = T[k', i+1] + T[k'-1, i+1]$. In conclusion, $Q'_{k',w[i..n]}(x) = T[k', i]$.

In the second case, let $w[i] = 1$.

If v_ℓ has the most significant bit 0 (so $\ell < 2^{k'-1}$), then the number of occurrences of v_ℓ in $w[i..n]$ equals the number of occurrences of v_ℓ in $w[i+1..n]$, so the coefficient of x^ℓ in $Q'_{k',w[i..n]}(x)$ equals the coefficient of x^ℓ in $Q'_{k',w[i+i..n]}(x) = T[k', i+1]$; note that the coefficient of x^ℓ in $x^{2^{k'}} Q_{k'-1,w[i+1..n]}(x)$ is 0.

If v_ℓ has the most significant bit 1 (so $\ell \geq 2^{k'-1}$), then the number of occurrences of v_ℓ in $w[i..n]$ equals the number of occurrences of v_ℓ in $w[i+1..n]$ plus the number of occurrences of $v_\ell[2..k']$ in $w[i+1..n]$. As $bin(v_\ell[2..k']) = bin(v_\ell) - 2^{k'}$, we get that the coefficient of x^ℓ in $Q'_{k',w[i..n]}$ is the sum of the coefficient of x^ℓ in $Q'_{k',w[i+1..n]}(x)$ and the coefficient of $x^{\ell-2^{k'}}$ in $Q'_{k'-1,w[i+1..n]}(x)$ (which can be seen as the coefficient of x^ℓ in $x^{2^{k'}} Q'_{k'-1,w[i+1..n]}(x)$).

In conclusion, we get that if $w[i] = 1$ then $Q'_{k',w[i..n]}(x) = Q'_{k',w[i+i..n]}(x) + x^{2^{k'}} Q'_{k'-1,w[i+1..n]}(x) = T[k', i+1] + T[k'-1, i+1] x^{2^{k'}}$. So, $Q'_{k',w[i..n]}(x) = T[k', i]$.

So the recurrent definition of $Q_{k',w[i..n]}(x)$ holds and we can compute all these values in $\mathcal{O}(k^2 n)$ time. As an important consequence, we have $Q'_{k',w}(x) = T[k', 1]$ as claimed. Thus, the conclusion of the lemma now follows as explained. □

We conclude this section by putting together all the preliminary results we have shown, to obtain the final Monte-Carlo algorithm solving Problem 1.

Randomised Algorithm ;

let p be a prime from $[n^k, 2n^k]$

choose x at random from $\mathbb{F}_p$

compute $Q_{k,w_1}(x)$ and $Q_{k,w_2}(x)$

$Q_{w_1,w_2}(x) := Q_{k,w_1}(x) - Q_{k,w_2}(x)$

return YES if $Q_{w_1,w_2}(x) = 0$
and NO otherwise

The overall time complexity is clearly polynomial both in k and in n; as $k \leq n$, we conclude that this algorithm runs in polynomial time. Moreover, the running time of the algorithm is linear in n, as generating a prime number between n^k and n^{2k} takes $\mathcal{O}(\text{polylog}(2n^k)) \subseteq \mathcal{O}((k \log n)^c)$ time, where $c \geq 2$ is a constant that depends on the specific algorithm we use.

Now, if $w_1 \equiv_k w_2$ then $Q_{w_1,w_2}(x) = 0$ for all $x \in \mathbb{F}_p$, so the algorithm will always return YES. Otherwise, there are two ways it could err. First, we could

have generated a non-prime p. Second, we could have generated a prime p, but our choice of x was unfortunate. The chance for the former is at most $\frac{1}{n^k}$. The chance for the latter, by the Schwart-Zippel lemma, is at most $\frac{2^{k+1}-1}{n^k}$. By the union bound, for large enough n, the total error probability is at most $\frac{1}{n}$ as required.

Theorem 2. *Problem 1, for input words of length n, can be solved by a Monte-Carlo algorithm with running time*

$$\mathcal{O}(nk^2 + (k \log n)^c) \subseteq \mathcal{O}(nk^c),$$

where c is a constant depending on the algorithm used to generate a prime number between n^k and $2n^k$. Our algorithm solving Problem 1 always returns a positive answer when the input words w_1 and w_2 are k-binomial equivalent, and returns a negative answer when w_1 and w_2 are not k-binomial equivalent with probability at least $1 - \frac{1}{n}$.

4 Conclusion

In this paper we considered the problem of deciding whether two given words w_1 and w_2 are k-binomial equivalent. We gave two polynomial algorithms solving this problem. The first one was deterministic, and was heavily relying on a known result showing that deciding whether two non-deterministic finite automata are path-equivalent can be done in linear time. The second one was a direct algorithm, its running time was linear in the length of the input words, but it was no longer deterministic.

The main consequence of our result is that also finding all the factors of a long word which are k-binomial equivalent to a shorter one can be done in polynomial time; in other words, the problem of pattern matching under k-binomial equivalence can be solved in polynomial time. Indeed, one can check (using the algorithms presented in this paper) for all factors of the text whether they are k-binomial equivalent to the pattern and return those for which this property holds. The next theorem follows.

Theorem 3. *Given two words w and x and a number k, we can find all the factors of w that are k-binomial equivalent to x in polynomial time.*

The main open problems remaining from this work are to find simpler and more efficient algorithms solving Problem 1 as well as a pattern matching under k-binomial equivalence solution that does not use testing k-binomial equivalence as a subroutine.

Acknowledgements

We thank Manfred Kufleitner and Eric Rowland for participating in the discussions on this problem during the Dagstuhl seminar 14111, that finally lead to the work presented here.

References

1. Abbott, J., Bronstein, M., Mulders, T.: Fast deterministic computation of determinants of dense matrices. In: Proceedings of the 1999 International Symposium on Symbolic and Algebraic Computation. pp. 197–204. ISSAC '99 (1999)
2. Agrawal, M., Kayal, N., Saxena, N.: Primes is in p. Annals of Mathematics 160(2) (2004)
3. Ehlers, T., Manea, F., Mercaş, R., Nowotka, D.: k-abelian pattern matching. In: Proc. 17th International Conference on Developments in Language Theory. Lecture Notes in Computer Science, vol. 8633, pp. 178–190. Springer (2014)
4. Ehlers, T., Manea, F., Mercas, R., Nowotka, D.: k-abelian pattern matching. J. Discrete Algorithms 34, 37–48 (2015)
5. von zur Gathen, J., Gerhard, J.: Modern Computer Algebra. Cambridge University Press, New York, NY, USA (1999)
6. Huova, M., Karhumäki, J., Saarela, A., Saari, K.: Local squares, periodicity and finite automata. In: Rainbow of Computer Science, pp. 90–101. Springer (2011)
7. Karhumäki, J., Saarela, A., Zamboni, L.Q.: On a generalization of abelian equivalence and complexity of infinite words. J. Comb. Theory, Ser. A 120(8), 2189–2206 (2013)
8. Lothaire, M.: Combinatorics on Words. Cambridge University Press (1997)
9. Rabin, M.O.: Probabilistic algorithm for testing primality. Journal of Number Theory 12(1), 128 – 138 (1980)
10. Rao, M., Rigo, M., Salimov, P.: Avoiding 2-binomial squares and cubes. Theor. Comput. Sci. 572, 83–91 (2015)
11. Rigo, M., Salimov, P.: Another generalization of abelian equivalence: Binomial complexity of infinite words. Theoretical Computer Science pp. – (2015)
12. Rigo, M., Salimov, P.: Another generalization of abelian equivalence: Binomial complexity of infinite words. In: Proc. 9th International Conference WORDS 2013. Lecture Notes in Computer Science, vol. 8079, pp. 217–228. Springer (2013)
13. Rozenberg, G., Salomaa, A.: Handbook of Formal Languages, vol. 1. Springer, Berlin (1997)
14. Sakarovitch, J.: Elements of Automata Theory. Cambridge University Press (2009)
15. Tzeng, W.: A polynomial-time algorithm for the equivalence of probabilistic automata. SIAM J. Comput. 21(2), 216–227 (1992)