Theoretical Computer Science 411 (2010) 3209–3223

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Growth rates of complexity of power-free languages

Arseny M. Shur *

Ural State University, 620083 Ekaterinburg, Russia

ARTICLE INFO

Article history:
Received 14 October 2008
Accepted 19 May 2010
Communicated by D. Perrin

Keywords:
Growth rate
Regular language
Power-free language
Finite antidictionary

ABSTRACT

We present a new fast algorithm for calculating the growth rate of complexity for regular languages. Using this algorithm we develop a space and time efficient method to approximate growth rates of complexity of arbitrary power-free languages over finite alphabets. Through extensive computer-assisted studies we sufficiently improve all known upper bounds for growth rates of such languages, obtain a lot of new bounds and discover some general regularities.

1. Introduction

The study of words and languages avoiding repetitions has been one of the central topics in combinatorics of words since the pioneering work of Thue [24]. For a survey, see [1] and the references therein. A repetition is called *avoidable* in the given alphabet, if there exist infinitely many words over this alphabet having no repetition of this type. Thue proved that squares are avoidable in the ternary alphabet, while cubes and overlaps are avoidable already over two letters. Integral powers, which are certainly the simplest repetitions, can be generalized in several ways. Among such generalizations we mention patterns, abelian powers, relational powers, and, of course, *fractional powers*, or *exponents*, which are studied in this paper. An exponent of a word is the ratio among its length and its minimal period. If $\beta > 1$ is a rational number, then a word is called β-free (β^+-free) if all its factors have exponents less than β (respectively, at most β).

When a repetition is known to be avoidable it is natural to calculate some quantitative measure of its avoidability. If ρ denotes an avoidable repetition (or, more generally, an avoidable property of words) over the alphabet Σ, then the "size" of the language $L(\rho) \subseteq \Sigma^*$ of all words avoiding ρ is an appropriate measure of avoidability of ρ. Such a size is given by the function $C_{L(\rho)}(n) = |L(\rho) \cap \Sigma^n|$, which is called *combinatorial complexity* of $L(\rho)$. Brandenburg [2] showed that the combinatorial complexities of the language of all binary cube-free words and the language of all ternary square-free words both have exponential growth. Restivo and Salemi [17] proved that the combinatorial complexity of the language of all binary overlap-free words grows as a polynomial. Since then, many papers have appeared in this area, see, e.g., [7,11,13,14,16,18], where some bounds of the order of growth for particular power-free languages were obtained. But up to now there has been no universal method for obtaining such bounds. Furthermore, the existing algorithms for computer-assisted studies are highly inefficient and hardly work on alphabets with more than 3 symbols.

In this paper we suggest a new universal approach for estimating the order of growth of an arbitrary power-free language L. More precisely, we estimate the *growth rate* of L which is defined by $\alpha(L) = \limsup_{n\to\infty}(C_L(n))^{1/n}$.

* Tel.: +7 9222072837.
E-mail addresses: aimsure@mail.ru, Arseny.Shur@usu.ru.

0304-3975/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2010.05.017

We begin with the study of regular languages (Section 3). If a regular language L is given by a deterministic automaton $\mathcal{A}$ satisfying some natural reachability conditions, then the growth rate of L is known to be equal to the spectral radius (or the *Frobenius root*) of the adjacency matrix of $\mathcal{A}$ (see, e.g., [12]). However, finding the Frobenius root straightforwardly, starting from an automaton through explicit computation with its adjacency matrix can be tedious. We show how to avoid these tedious computations using all the matrices involved implicitly. The resulting algorithm (Algorithm 1) calculates the growth rate of a regular language from its recognizing automaton and uses only quasilinear time and space in the size of this automaton. High efficiency of Algorithm 1 allows us to cope with very big automata that are well beyond capabilities of other methods.

This method cannot be directly used in the case of power-free languages, as infinite power-free languages are not regular. Instead, we approximate the growth rate of any such language L from above, using a decreasing sequence of regular languages called *k-approximations* of L. Such approximations have been used before, e.g. in [2,12,16]. The intersection of all k-approximations equals L, so k indicates the "quality" of approximation. Unfortunately, the automata recognizing k-approximations are quite big: their size can be roughly described as an exponential function of k multiplied by the factorial of the size of the alphabet. In Section 4 we make a crucial improvement to this approach: from a k-approximation we build a relatively small automaton with the same spectral radius as the recognizing automaton for the k-approximation (Algorithms 3 and 4). The size of this "smaller" automaton, as well as the time needed for its construction, still depends exponentially on k, but only *linearly* on the size of the alphabet. Processing these "smaller" automata by Algorithm 1 allows one to operate with bigger alphabets and to obtain very tight upper bounds.

Using implementations of Algorithms 1, 3 and 4, we conducted an extensive study of the growth rates of power-free languages for alphabets of 2 to 10 letters. Selected results of those studies are presented in Section 5.

2. Preliminaries

We recall necessary notation and definitions. For more background, see [5,9,15].

2.1. Words, languages, and automata

An *alphabet* Σ is a nonempty finite set, the elements of which are called *letters*. *Words* are finite sequences of letters. As usual, we write Σ^n for the set of all n-letter words and Σ^* for the set of all words over Σ, including the *empty word* λ. A word u is a *factor* (respectively *prefix*, *suffix*) of the word w if w can be represented as $\bar{v}u\hat{v}$ (respectively $u\hat{v}$, $\bar{v}u$) for some (possibly empty) words $\bar{v}$ and $\hat{v}$. A *factor* (respectively *prefix*, *suffix*) of w is called *proper*, if it does not coincide with w. The words u and w are *conjugates*, if there are words $\bar{v}$ and $\hat{v}$ such that $u = \bar{v}\hat{v}$, $w = \hat{v}\bar{v}$. The subsets of Σ^* are called *languages* (over Σ). A language is *factorial* if it is closed under taking factors of its words.

A word w is *forbidden* for a language L if it is a factor of no word from L. The set of all minimal (with respect to the factorization order) forbidden words for a language is called the *antidictionary* of this language. If a factorial language L over the alphabet Σ has the antidictionary M, then the following equalities holds:

$$L = \Sigma^* - \Sigma^* M \Sigma^*, \qquad M = \Sigma L \cap L\Sigma \cap (\Sigma^* - L).$$

We see that any antidictionary determines a unique factorial language, which is regular if the antidictionary is also regular (in particular, finite).

A word $w \in \Sigma^*$ can be viewed as a function $\{1, \ldots, n\} \to \Sigma$. Then a *period* of w is any period of this function. The *exponent* of w is the ratio among its length and its minimal period; if this ratio is greater, than 1, then w is a *fractional power*. The prefix of w whose length is the minimal period of w, is called the *root* of w. If $\beta > 1$ is a rational number, then w is called β-free (β^+-free) if all its factors have exponents less than β (respectively, at most β). By β-free (β^+-free) languages we mean the languages of *all* β-free (respectively β^+-free) words over a given alphabet. These languages are obviously factorial and are also called *power-free* languages. Following [2], we use only the term β-free, assuming that β belongs to the set of "extended rationals". This set consists of all rational numbers and all such numbers with a plus; the number x^+ covers x in the usual $\leq$ order. For the arithmetic operations it will be enough to view x^+ as the number $(x + \frac{1}{n})$, where n is extremely big. For example, the condition $r \leq k/2^+$ means just that $r < k/2$.

We consider *deterministic finite automata* (dfa's) with *partial* transition function. The language recognized by a dfa $\mathcal{A}$ is denoted by $L(\mathcal{A})$. We view a dfa as a digraph, sometimes even omitting the labels of edges. Thus, we write a transition as a triple (u, a, v) or as a pair (u, v) depending on whether the letter a is essential or not. A *trie* is a dfa which is a tree such that the initial vertex is its root and the set of terminal vertices is the set of all its leaves. Only *directed* walks in digraphs are considered. For a dfa, the number of words of length n in the language it recognizes obviously equals the number of *accepting walks* of length n in the automaton. So, to calculate combinatorial complexity we count walks rather than words. For a fixed automaton, we denote the number of (u, v)-walks of length n by $P_{uv}(n)$ and the number of walks of length n ending at the vertex v by $P_{*v}(n)$. A dfa is *consistent* if each of its vertices is contained in some accepting walk.

2.2. Growth rates

For an arbitrary language L, we are interested in the asymptotic behaviour of its combinatorial complexity $C_L(n)$, more precisely, in the growth rate $\alpha(L) = \limsup_{n\to\infty}(C_L(n))^{1/n}$. For factorial languages, in particular for power-free ones, the following theorem holds.

Theorem 2.1 ([12]). *For an arbitrary factorial language L one has $\alpha(L) = \lim_{n\to\infty}(C_L(n))^{1/n} = \inf\{(C_L(n))^{1/n}\}$. Moreover, $\alpha(L) = 0$ if and only if L is finite, $\alpha(L) > 1$ if and only if L is infinite and has the exponential complexity, and $\alpha(L) = 1$ otherwise.*

In the study of the growth rates of power-free languages the main focus is on the minimal such languages (for different alphabets). These minimal languages are the binary 2^+-free (overlap-free), ternary $(7/4)^+$-free, quaternary $(7/5)^+$-free languages, and, according to famous Dejean's conjecture [6], the $(m/m-1)^+$-free languages over m-letter alphabets with $m \geq 5$. This conjecture has been recently proved in all cases, see, e.g. [3,25,26].

2.3. Digraphs and linear algebra

A *strongly connected component* (scc) of a digraph G is a maximal with respect to inclusion subgraph G' such that there exists a walk from any vertex of G' to any other vertex of G'. A digraph is *strongly connected*, if it consists of a unique scc. The *imprimitivity number* of an scc (or of a strongly connected digraph) is the greatest common divisor of lengths of all cycles in it. If this number equals 1, then the scc or the digraph is called *primitive*.

The *index* of a digraph is the maximum absolute value of the eigenvalues of its adjacency matrix. According to the classical Perron–Frobenius theorem, the index is itself an eigenvalue of this matrix (the *Frobenius root*). The Frobenius root admits a nonnegative real eigenvector. For the case of a strongly connected digraph, the Frobenius root is an eigenvalue of multiplicity 1.

Growth rates of regular languages and indices of digraphs are closely connected. Namely, if $\mathcal{A}$ is a consistent dfa, $L = L(\mathcal{A})$, then $\alpha(L)$ equals the index of $\mathcal{A}$. A short proof of this fact can be found in [21]. In what follows, we denote the index of $\mathcal{A}$ by $\alpha(\mathcal{A})$.

The jth component of a vector $\mathbf{x}$ is denoted by $[\mathbf{x}]_j$.

3. An algorithm for growth rates of regular languages

3.1. Algorithm

As was mentioned in the preliminaries, the growth rate of a regular language is an eigenvalue of the adjacency matrix of an automaton representing the language. Hence, to obtain the exact growth rate one should find an exact solution to a polynomial equation, which is not possible in the general case. On the other hand, a real root of a polynomial equation can be approximated with any precision with rather simple methods. So it is natural to say that an algorithm *calculates* the growth rate of a language if this algorithm returns an approximation to this growth rate within any prescribed range of the absolute error.

Theorem 3.1. *There is an algorithm which, given a consistent dfa $\mathcal{A}$ with N vertices and a number δ, $0 < \delta < 1$, calculates the growth rate of the language $L(\mathcal{A}) \subseteq \Sigma^*$ with the absolute error at most δ in time $\Theta(-\log\delta\cdot|\Sigma|\cdot N)$ using $\Theta(-\log\delta\cdot N)$ additional space.*

Below we provide the required algorithm. Its idea is based on the following lemma from [22]:

Lemma 3.2. *Let G be a non-singleton strongly connected digraph, α be its index, and r be its imprimitivity number. Then for arbitrary vertices u, v of G there exist numbers $i \in \{0,\dots,r-1\}$, $\mu > 0$, and γ, $0 < \gamma \leq \alpha$, such that*

$$P_{uv}(n) = \begin{cases} \mu\alpha^n + O((\alpha-\gamma)^n), & \text{if } n \bmod r = i, \\ 0 & \text{otherwise}. \end{cases}$$

So, for $r = 1$ the sequence $\{P_{uv}(n)/P_{uv}(n-1)\}$ converges to the limit α at an exponential rate. Thus, for any δ, $0 < \delta < 1$ all members of this sequence, starting from some $\bar{n} = O(-\log\delta)$, belong to the δ-neighborhood of α. Obviously, the same is true for the sequence $\{P_{*v}(n)/P_{*v}(n-1)\}$. The latter sequence will be used in our algorithm.

There is a simple operation which transforms any strongly connected digraph to a primitive strongly connected digraph: the addition of a single loop to each vertex. From the matrix point of view, this operation just adds the identity matrix to the adjacency matrix of the digraph. Hence, this operation retains all eigenvectors of the original matrix and adds 1 to each of its eigenvalues (in particular, to the Frobenius root).

Recall that the index of an arbitrary digraph equals the maximum of the indices of its scc's. Now we introduce our algorithm.

Algorithm 1.
Input: consistent dfa $\mathcal{A}$, number δ, $0 < \delta < 1$.
Output: number $\bar{\alpha}$ such that $|\bar{\alpha} - \alpha(\mathcal{A})| < \delta$.

```
01. find all scc's of A
02. if all scc's are singletons or cycles
03.     if there are cycles return ᾱ = 1 else return ᾱ = 0
04. else
05.     for each nontrivial scc A_i
06.         for each vertex u
07.             define counters u.old and u.new
08.             u.old ← 1;  u.new ← 0
09.         while true do
10.             for each vertex u
11.                 for each edge (u, v)
12.                     v.new ← v.new + u.old
13.                 u.new ← u.new + u.old
14.             minrate ← min_u(u.new/u.old)
15.             maxrate ← max_u(u.new/u.old)
16.             if  maxrate − minrate < 2δ
17.                 α_i ← (maxrate + minrate)/2
18.                 break while
19.             for each vertex u
20.                 u.old ← u.new;  u.new ← 0
21.     return ᾱ = max_i{α_i} − 1.
```

Proof of Theorem 3.1. The procedure in line 1 can be performed in $O(|\Sigma|{\cdot}N)$ time and $O(N)$ space using well-known Tarjan's algorithm. The assignment in line 3 is made according to Theorem 2.1. So, we turn to the main part of the algorithm.

Let $\mathcal{A}_i'$ be the digraph obtained from $\mathcal{A}_i$ by adding a single loop to each vertex. In the nth iteration of the `while` cycle after the execution of the cycle in lines 10–13 one has $v.new = P_{*v}(n)$, $v.old = P_{*v}(n-1)$ for each vertex v, where the number of walks is calculated in $\mathcal{A}_i'$. This fact can be easily shown by induction using the equality $P_{*v}(n) = \sum P_{*u}(n-1) + P_{*v}(n-1)$, where the sum is taken over all u's such that (u, v) is an edge in $\mathcal{A}_i$, and the additional summand corresponds to the new loop added to v.

As we already mentioned, each sequence $\{P_{*v}(n)/P_{*v}(n-1)\}$ converges to the limit $\alpha(\mathcal{A}_i') = \alpha(\mathcal{A}_i) + 1$. Hence, to prove the correctness of the algorithm it remains to show that $minrate \le \alpha(\mathcal{A}_i') \le maxrate$ in any iteration of the `while` cycle. This inequality follows from one result of matrix theory (see [9]):

– *if α is the Frobenius root of an irreducible primitive matrix A, $\mathbf{x}$ is a vector with positive components, then*

$$\min_j \frac{[\mathbf{x}A]_j}{[\mathbf{x}]_j} \le \alpha \le \max_j \frac{[\mathbf{x}A]_j}{[\mathbf{x}]_j}.$$

With respect to the adjacency matrices, "irreducible" just means that the digraph is strongly connected. So, if **Old** is the vector whose components are the values of all counters *.old* taken in some order, **New** is the vector whose components are the values of counters *.new* taken in the same order, and A_i' is the adjacency matrix of $\mathcal{A}_i'$, then $\mathbf{New} = \mathbf{Old}{\cdot}A_i'$. The required inequality now follows from the rules of computation of *minrate* and *maxrate* (lines 14–15 of Algorithm 1).

Thus, Algorithm 1 works correctly and it remains to check the time and space complexity. Each iteration of the `while` cycle requires $\Theta(|\Sigma|{\cdot}N)$ operations. The total number of iterations is $O(-\log\delta)$, because after this number of iterations the ratio $u.new/u.old$ for any vertex u falls into the δ-neighborhood of the number $\alpha(\mathcal{A}_i')$ (see the comment after Lemma 3.2). The space complexity is dominated by the amount needed to allocate the counters $u.old$ and $u.new$. The total number of counters is $2N$ and the length of each counter linearly depends on the number of iterations, which is $O(-\log\delta)$. Thus, the required conditions are met and the theorem is proved. □

Remark 3.3. When building the vector **New** from the vector **Old** on each iteration of the `while` cycle we actually perform a well-known iterative algorithm, which calculates the Frobenius root of an irreducible primitive matrix by means of its eigenvector [8]. For a matrix A, this algorithm takes an arbitrary initial vector $\mathbf{x}$ and apply A to it iteratively to obtain the approximation of the mentioned eigenvector. Let a matrix A have a Jordan basis $\mathbf{e}_1, \ldots, \mathbf{e}_r$ (where $\mathbf{e}_1$ corresponds to the Frobenius root α and is real-valued, while other vectors can be complex-valued). This algorithm works correctly for any initial vector $\mathbf{x} = x_1\mathbf{e}_1 + \cdots + x_r\mathbf{e}_r$ such that $x_1 \ne 0$. Indeed, the multiplication by A increases the first term of this sum exactly α times, while the length of the remaining linear combination can be increased at the rate at most γ, where $\gamma < \alpha$

is the second biggest absolute value of the eigenvalues of A. Therefore, the vector $\mathbf{x}A^n$ deviates from the eigenvector of α by a relatively small additive noise.

It is known that this algorithm is robust to rounding and scaling errors; such a good property allows us to pay little attention to the problems of storing big numbers and dividing them. Also note that our initial vector $(1, \ldots, 1)$ is good for the matrix A'_i in view of Lemma 3.2.

3.2. Two straightforward generalizations

The proof of Theorem 3.1 remains valid for any directed multigraph, since we did not use any specifics of automata. Hence, we may reformulate this theorem in purely graph terms.

Theorem 3.4. *There is an algorithm which, given a directed multigraph G with n vertices and m different edges and a number δ, $0 < \delta < 1$, calculates the index of G with the absolute error at most δ in time $\Theta(-\log \delta \cdot m)$ using $\Theta(-\log \delta \cdot n)$ additional space.*

In general, Algorithm 1 can not calculate the growth rate of a language given by a nondeterministic finite automaton (nfa). Indeed, counting walks and counting words are not the same thing for the case of nfa, and the growth rate of a language can differ from the index of recognizing nfa. But there exists an important class of nfa's for which Algorithm 1 works perfectly in view of the following lemma. Recall that a nfa is called *unambiguous*, if for any given word w and vertices u and v there exists at most one (u, v)-walk with the label w. The definition of a consistent nfa is the same as for dfa's.

Lemma 3.5. *Let $\mathcal{A}$ be an unambiguous consistent nfa, $L = L(\mathcal{A})$. Then $\alpha(L) = \alpha(\mathcal{A})$.*

Proof. By the definition of an unambiguous nfa, each function $P_{uv}(n)$ returns exactly the number of words of length n which can be read from u to v. Suppose that I and T are the sets of initial and terminal vertices of $\mathcal{A}$ respectively, A is the adjacency matrix of $\mathcal{A}$. Then for any n

$$\max_{I\times T} P_{uv}(n) \le C_L(n) \le \sum_{I\times T} P_{uv}(n) \le |A^n|,$$

where $|A^n|$ is the sum of all elements of A^n. Furthermore, it is easy to see that

$$\max_{I\times T} \limsup_{n\to\infty} (P_{uv}(n))^{1/n} \le \limsup_{n\to\infty} (C_L(n))^{1/n} \le \lim_{n\to\infty} |A^n|^{1/n}.$$

It is well-known in matrix theory that the limit in the right-hand side is equal to $\alpha(A)$ [9]. Since the middle expression of the last inequality equals $\alpha(L)$ by definition, it remains to show that the left-hand side also equals $\alpha(A)$. For the uniformity, we denote $\alpha(P_{uv}) = \limsup_{n\to\infty} (P_{uv}(n))^{1/n}$.

If (v', v) is an edge in $\mathcal{A}$, then $P_{uv}(n+1) \ge P_{uv'}(n)$, implying $\alpha(P_{uv}) \ge \alpha(P_{uv'})$. Similarly, if (u, u') is an edge, then $\alpha(P_{uv}) \ge \alpha(P_{u'v})$. Since any vertex belongs to a path from some initial vertex to some terminal vertex, the overall maximum of the numbers $\alpha(P_{uv})$ is achieved for some $u \in I$, $v \in T$. Since the function $|A^n|$ is a finite sum of functions $P_{uv}(n)$, we have $\max_{I\times T} \alpha(P_{uv}) = \alpha(A)$, as required. □

Algorithm 1 calculates the index of the processed automaton, so we have

Theorem 3.6. *There is an algorithm which, given an unambiguous consistent nfa $\mathcal{A}$ with N vertices and a number δ, $0 < \delta < 1$, calculates the growth rate of the language $L(\mathcal{A}) \subseteq \Sigma^*$ with the absolute error at most δ in time $\Theta(-\log \delta \cdot |\Sigma| \cdot N)$ using $\Theta(-\log \delta \cdot N)$ additional space.*

3.3. A faster "quasialgorithm"

Algorithm 1 has proved to be efficient enough, it allows one to process the automata with millions of vertices, and works with any consistent dfa. Nevertheless, for extensive computer-assisted studies it would be convenient to have a modification that works faster in a "typical" case and can detect an occurrence of a "special" case.

In our studies of power-free languages we made use of a "quasialgorithm" which is below referred to as Algorithm 1Q. It processes a dfa similarly to Algorithm 1 but calculates some other sequence of numbers. This sequence converges to the index of the processed automaton exponentially fast in the typical case, and can be calculated somewhat faster, than the sequences of *minrate*'s and *maxrate*'s in Algorithm 1. The two advantages of Algorithm 1Q over Algorithm 1 are the elimination of the preliminary step (no splitting into scc's) and the avoidance of divisions (one division per iteration instead of $\Theta(N)$ in Algorithm 1). The lack of Algorithm 1Q is the absence of a procedure which can assure that the current value falls into the δ-neighborhood of the index of the automaton. Instead, we use empirical rules which appeared to work perfectly for the automata arising from the studies of power-free languages. Namely, no divergence was found between the results

obtained by applying Algorithm 1Q and Algorithm 1 to the same automata. So, we made a lot of auxiliary computations with Algorithm 1Q. Its description is given below.

The starting point for Algorithm 1Q is the following theorem, which comprises some results of [22,23].

Theorem 3.7 ([22,23]). *Let $\mathcal{A}$ be a consistent dfa, r be the least common multiple of the imprimitivity numbers of all scc's of $\mathcal{A}$, $L = L(\mathcal{A})$. Then each of the sequences $\left\{\frac{C_L(nr+i+1)}{C_L(nr+i)}\right\}_1^\infty$, where $i = 0, \ldots, r-1$, converges to a nonnegative (possibly infinite) limit β_i. In the case of factorial language L all β_i's are finite and*

(1) *$\alpha(L) = (\beta_0 \cdot \cdots \cdot \beta_{r-1})^{1/r}$; in particular, $r = 1$ implies $\frac{C_L(n+1)}{C_L(n)} \longrightarrow \alpha(L)$;*

(2) *if $\mathcal{A}$ does not contain two scc's of index $\alpha(L)$ such that one of them is reachable from the other, then $\left|\frac{C_L(nr+i+1)}{C_L(nr+i)} - \beta_i\right| = O(\gamma^n)$ for any $i = 0, \ldots, r-1$ and some $\gamma < 1$.*

Recall that power-free languages are factorial, while the procedure of adding loops to all vertices of the dfa $\mathcal{A}$ guarantees that $r = 1$ for the resulting dfa $\mathcal{A}'$ (to obtain a dfa, we can assume that all loops are labeled by a new letter). Let $L' = L(\mathcal{A}')$. Then the sequence $\left\{\frac{C_{L'}(n+1)}{C_{L'}(n)}\right\}$ converges to the number $\alpha(L') = \alpha(L) + 1$. Moreover, the convergence has an exponential rate in the case which can be reasonably considered as typical (as we mention in Section 5 below, all obtained automata for approximations of power-free languages fall into this case).

It is not hard to see that the counters *.old* and *.new* attached to the vertices of $\mathcal{A}$ can be used to calculate the values of combinatorial complexity $C_{L'}(n)$. Namely, we take the initial vector **Old** $= (1, 0, \ldots, 0)$, where the only 1 corresponds to the initial vertex of $\mathcal{A}$, and work with the counters as in Algorithm 1. Then in nth iteration each counter $u.new$ gets the number of walks of length n from the initial vertex to u. Summing up the values of *.new* counters over the set of terminal vertices, one obtains exactly $C_{L'}(n)$. Note that all vertices of a consistent dfa recognizing a factorial language are terminal.

Algorithm 1Q.
Input: consistent dfa $\mathcal{A}$, number δ, $0 < \delta < 1$, number $\bar{n}$.
Output: number $\bar{\alpha}$ for which the procedure ORACLE suggests $|\bar{\alpha} - \alpha(\mathcal{A})| < \delta$, or the number $\bar{\delta}$ which estimates the approximation error after $\bar{n}$ iterations.
Initialization: Put $s.old = 1$ for the initial vertex s and set all other *.old* and *.new* counters to 0. Put $comp[0] = 1$.

```
01. for n = 1 to n̄
02.     for each vertex u
03.         for each edge (u, v)
04.             v.new ← v.new + u.old
05.         u.new ← u.new + u.old
06.     for each vertex u
07.         comp[n] ← comp[n] + u.new
08.         u.old ← u.new;  u.new ← 0
09.     rate[n] ← comp[n]/comp[n − 1]
10.     if ORACLE(rate, δ) = ᾱ
11.         return ᾱ; break for
12. return δ̄ = |rate[n̄] − rate[n̄ − 1]|.
```

How does the empirical procedure ORACLE work? First, it checks whether $|rate[n] - rate[n-1]| < \delta$, because this difference tends to 0 at exactly the same exponential rate as the approximation error $|rate[n] - \alpha(L')|$. If this inequality fails, ORACLE returns *false*, otherwise ORACLE studies whether the sequence *rate* is oscillating or monotonous. For the oscillating sequence, the calculation of *rate* is continued up to the moment when the difference between the last local maximum and local minimum drops below δ; ORACLE returns $rate[n] - 1$ during the corresponding iteration. If the sequence *rate* is monotonous, it is extrapolated as a geometric series to get the limit α; here, ORACLE immediately returns the number $\alpha - 1$.

Those empirical rules result from the study of the connections between the behaviour of the combinatorial complexity of a regular language and the second biggest in the absolute value eigenvalues of the adjacency matrix of the recognizing dfa. These connections are as follows (for the sake of brevity, we omit the details). Let γ be the second biggest the absolute value of the eigenvalues of the adjacency matrix A. If A has only positive real eigenvalue of absolute value γ, then the sequence *rate* is monotonous. If A has *no* positive real eigenvalue of absolute value γ, then *rate* is oscillating. Finally, if the positive real eigenvalue is accompanied with some different eigenvalues of absolute value γ, then the sequence *rate* can be oscillating, monotonous, or have a more complicated behaviour. Up to now, we have no examples of such "complicated" behaviour of the sequence *rate*.

If Algorithm 1Q outputs $\bar{\delta}$ (for $\bar{n}$ big enough), a slow convergence is suspected. Restarting the algorithm with different values of $\bar{n}$, we can check that the rate of the convergence is less than exponential and detect the special case which is described in Theorem 3.7(2).

4. Approximation of power-free languages

4.1. Simple method

The idea to use languages with finite antidictionaries for finding upper bounds to the growth rates of factorial languages goes back to Brandenburg [2] and has been used by many others. In this subsection we describe a simple method, which enables one to get such upper bounds using this idea, a textbook pattern matching algorithm, and Algorithm 1 (or 1Q).

Recall how to use the languages with finite antidictionaries to estimate the complexity of a given factorial language $L \subseteq \Sigma^*$. Let M be the antidictionary of L. Consider a family $\{M_k\}$ of finite subsets of M such that

$$M_1 \subseteq M_2 \subseteq \cdots \subseteq M_k \subseteq \cdots \subseteq M, \qquad M_1 \cup M_2 \cup \cdots \cup M_k \cup \cdots = M.$$

For the rest of the paper, we put $M_k = M \cap (\Sigma^1 \cup \cdots \cup \Sigma^k)$. Denote by L_k the factorial language over Σ having the antidictionary M_k. One has

$$L \subseteq \cdots \subseteq L_k \subseteq \cdots \subseteq L_1, \qquad L_1 \cap L_2 \cap \cdots \cap L_k \cap \cdots = L,$$

and for any n, there is k such that $L \cap \Sigma^n = L_k \cap \Sigma^n$. Then

$$C_L(n) = \cdots = C_{L_k}(n) \leq \cdots \leq C_{L_1}(n).$$

Hence the sequence $\{C_{L_k}(n)\}$ converges to $C_L(n)$ from above, implying that $\{(C_{L_k}(n))^{1/n}\}$ converges to $(C_L(n))^{1/n}$ from above. By Theorem 2.1, $\alpha(L) = \lim_{n\to\infty}(C_L(n))^{1/n}$ and $\alpha(L_k) \leq (C_{L_k}(n))^{1/n}$. Thus, $\{\alpha(L_k)\}$ converges to $\alpha(L)$. Since L_k is regular, we can build a recognizing consistent dfa for it and apply Algorithm 1 to find $\{\alpha(L_k)\}$ with any precision. In what follows, we refer to M_k as to *k-antidictionary* of L and to L_k as to *k-approximation* of L.

Luckily enough, there is a fast algorithm (Algorithm 2) building a consistent dfa for the language given by a finite antidictionary. This algorithm, based on the Aho–Corasick algorithm for pattern matching, is described in [4]. It works in linear time in the total size of the antidictionary. Thus, if one can construct the k-antidictionaries efficiently, then the growth rate of L can be approximated from above with really good precision. Here Algorithm 2 is presented in the most understandable form. The necessary details of realization are discussed below.

Algorithm 2.
Input: finite antidictionary M.
Output: dfa $\mathcal{A}$ recognizing the factorial language L with the antidictionary M.
Step 1. Construct a trie $\mathcal{T}$, recognizing M. ($\mathcal{T}$ is actually the graph of the prefix order on the set of all prefixes of M.)
Step 2. Associate each vertex in $\mathcal{T}$ with the word labeling the accepting walk ending at this vertex. (Now the set of vertices is the set of all prefixes of words from M.)
Step 3. Add all possible edges to $\mathcal{T}$, following the rule:
the edge (u, c, v) should be added if
u is not terminal, and
u has no outgoing edge labeled by c, and
v is the longest suffix of uc which is a vertex of $\mathcal{T}$.
(These edges are called *backward* while the edges of the trie are called *forward*.)
Step 4. Remove all terminal vertices and mark all remaining vertices as terminal to get $\mathcal{A}$.

In what follows, we refer to the automaton obtained by Algorithm 2 as to *FAD-automaton* (from Finite AntiDictionary).

Remark 4.1. (1) The set of vertices of $\mathcal{A}$ coincides with the set of all proper prefixes of the words from M.
(2) The triple (u, c, v) is a transition in $\mathcal{A}$ if and only if v is the longest suffix of the word uc which is a vertex of $\mathcal{A}$.

We must say a few words about the linear-time implementation of this algorithm (see the proofs in [4]), because it is not obvious. Actually, no explicit assignment of words to the vertices (step 2) is made. Instead, the *failure function* f is used to calculate the backward edges. This function is defined by $f(\lambda) = \lambda$, and for any other vertex u of $\mathcal{A}$ we find its longest *proper* suffix v which is also a vertex, and set $f(u) = v$. The calculation of f and of backward edges can be done in parallel. Namely, all vertices are processed in width-first order starting from the root of the trie. In this way, at the moment when we reach the vertex u the value $f(u)$ is already known and any vertex v with $|v| < |u|$ has a complete set of outgoing edges. For any letter c, we find the vertex w reached from $f(u)$ by the edge labeled by c. If u has an outgoing edge labeled by c, then we set $f(uc) = w$; if u has no such edge, then we add the edge (u, c, w) to the automaton.

Now describe a rather straightforward procedure calculating the k-antidictionary for a β-free language $L \in \Sigma^*$. Put $r = \lfloor k/\beta \rfloor$. Then r is the maximal length of the root of a word from M_k. To obtain M_k, we examine all β-free words of length $\leq r$ in the width-first order. For every such word u we build the shortest word $v = u^{\bar{\beta}}$ with $\bar{\beta} \geq \beta$ and check whether v contains a proper forbidden factor. If v has no such factors, then it belongs to M_k. The words of M_k are stored in a trie. Thus, examining the word u, we use the current trie to check the factors of v and to check which words of the form uc, where $c \in \Sigma$, are β-free.

Thus, we obtain a simple way to get an upper bound of the growth rate of a power-free language: choose the number k, calculate the k-antidictionary of the target language, build a FAD-automaton by Algorithm 2, and apply Algorithm 1 (or 1Q) to this automaton.

4.2. Using symmetry: equitable partitions and factorgraphs

A useful property of all power-free languages is the *permutation stability*, or *symmetry*: if a word w belongs to a power-free language $L \subseteq \Sigma^*$ and σ is an arbitrary permutation of Σ, then $\sigma(w)$ also belongs to L. It is easy to see that the antidictionary M of L has the same property, as well as all its k-antidictionaries and k-approximations. The symmetry of M_k implies symmetry in the corresponding FAD-automaton. The symmetric structure of the automaton allows us to reduce its size almost $|\Sigma|!$ times retaining the combinatorial complexity of the recognized language. Such a reduction sufficiently improves both the time and space complexity of the method described in the previous subsection, and allows us to study languages over bigger alphabets.

The reduction is based on the notion of *equitable partition* of a graph. Let $G = (V, E)$ be a graph. The partition $\pi = \{C_i\}_{i=1}^r$ of V is said to be equitable if for any i, j all vertices of C_i have the same number of adjacent vertices in C_j (for a digraph, one should count either "forward adjacent" or "backward adjacent" vertices). An equitable partition defines a nonnegative $r \times r$ matrix, the entries of which are those "adjacency numbers". This matrix can be viewed as an adjacency matrix of a directed multigraph which is denoted by G/π and called the *factorgraph* of G by the partition π. The vertices of G/π are the classes of π. For undirected graphs, the properties of equitable partitions are studied in detail in [10]. The most important for us property compares the number of walks in G and G/π. This property holds for digraphs as well:

Lemma 4.2. *Let π be an equitable partition of a digraph G. Then the number of length n walks in G starting at a fixed vertex u equals the number of length n walks in G/π starting at the corresponding vertex C_u. In particular, $\alpha(G) = \alpha(G/\pi)$.*

We omit the proof of Lemma 4.2 here, because the argument almost literally repeats the proof of [10] for undirected graphs. Now we describe an equitable partition of a symmetric FAD-automaton.

Proposition 4.3. *Let $\mathcal{A}$ be the FAD-automaton for a symmetric finite antidictionary $M \subseteq \Sigma^*$, and let $\pi = \{C_i\}_{i=1}^r$ be the partition such that the class C_u of the word u consists of all words which are images of u under the permutations of Σ. Then π is equitable.*

Proof. If u is a vertex of $\mathcal{A}$, then u is a proper prefix of some word $w \in M$ by Remark 4.1. Hence, for any permutation σ of Σ, one has $\sigma(w) \in M$ by symmetry of M, yielding that $\sigma(u)$ is a vertex of $\mathcal{A}$. Thus, the partition π is defined correctly. Further, if u has a forward outgoing edge labeled by a, then ua is a vertex, $\sigma(ua)$ is also a vertex, and $\sigma(u)$ has a forward outgoing edge labeled by $\sigma(a)$. Finally, consider a backward edge from u to v labeled by b. Then v is a suffix of ub and a vertex, while all longer suffixes of ub are not vertices. Hence, $\sigma(v)$ is a suffix of $\sigma(ub)$ and a vertex, while all longer suffixes of $\sigma(ub)$ are not vertices. By step 3 of Algorithm 2, there is a backward edge from $\sigma(u)$ to $\sigma(v)$ labeled by $\sigma(b)$.

We proved that there is a π-preserving bijection between the sets of forward adjacent vertices of the vertices u and $\sigma(u)$. Therefore, the partition π is equitable by definition. □

Recall that Algorithm 1 (and 1Q, as well) works with a digraph, using no "automaton" structure. So, by Lemma 4.2, we may apply any of these two algorithms to $\mathcal{A}/\pi$ in order to get the growth rate of the language recognized by $\mathcal{A}$.

4.3. Enhanced method: building factorAD and factorgraph

An obvious way to improve the simple method of Section 4.1 is to apply Algorithm 1 (or 1Q) to the factorgraph $\mathcal{A}/\pi$ instead of a symmetric FAD-automaton $\mathcal{A}$. In this way we obtain a substantial gain in time which probably compensates the time needed to perform the factorization of $\mathcal{A}$. On the other hand, we still need to built the automaton $\mathcal{A}$ and allocate it in memory. In view of Theorem 3.1, we can expect that Algorithms 1 and 1Q can process in a reasonable amount of time any automaton which is built and allocated. Thus, this improvement can neither extend the range of power-free languages for which the upper bounds can be obtained, nor obtain tighter bounds for the languages that can be processed. So, the usability of this single improvement is questionable.

Things change drastically if we can build the factorgraph $\mathcal{A}/\pi$ for the given k-approximation directly. The space needed to built and allocate the automaton is reduced almost $|\Sigma|!$ times and, as we will see below, the time needed to perform these operations shrinks by at least the same factor. Hence, a much bigger range of power-free languages can be processed, and better bounds can be obtained. The value of such an improvement to the simple method is indicated by the following fact: for all studied power-free languages the space, but not the time, was the critical resource for obtaining tight upper bounds.

In this section we present the direct algorithm to build the factorgraph $\mathcal{A}/\pi$. It is based on Algorithm 2, with some other trie built on step 1 and a more complicated rule for backward edges used on step 3.

Since the vertices of the FAD-automaton $\mathcal{A}$ are proper prefixes of the words from the antidictionary M (Remark 4.1), the vertices of $\mathcal{A}/\pi$ are classes of such prefixes. It is convenient to replace these classes with their canonical representatives. We choose the lexicographically minimal (*lexmin*) word in each class to be the representative. Note that the chosen lexmin words are exactly the proper prefixes of lexmin words from M. In what follows, we use the notation $Lexmin(w)$ for the lexmin word in the π-class of w.

Now examine the edges of $\mathcal{A}/\pi$. Suppose that an edge from u to v exists. Then for every word $\bar{u}$ with the property $Lexmin(\bar{u}) = u$ there exists a word $\bar{v}$ such that $Lexmin(\bar{v}) = v$ and $(\bar{u}, c, \bar{v})$ is an edge of $\mathcal{A}$ for some letter c. Let the vertex u have exactly t outgoing edges in $\mathcal{A}$ (with the different labels $c_1, \ldots, c_t$) to the vertices of the π-class of v. Then by definition of the equitable partition there are exactly t edges from u to v in $\mathcal{A}/\pi$. We will add the labels $c_1, \ldots, c_t$ to these edges. In this way, the factorgraph $\mathcal{A}/\pi$ can be viewed as a dfa and constructed using the ideas of Algorithm 2.

From the previous paragraph it follows that the edges of $\mathcal{A}/\pi$ can be constructed using only the edges of $\mathcal{A}$ starting in lexmin words. More precisely,

(*) $(u, c, Lexmin(v))$ is an edge of $\mathcal{A}/\pi$ if and only if (u, c, v) is an edge of $\mathcal{A}$.

We call the edge of $\mathcal{A}/\pi$ *forward*, if it results from a forward edge of $\mathcal{A}$ and *backward* otherwise.

Now, the main idea of constructing the automaton $\mathcal{A}/\pi$ is straightforward. First, we build the trie $\mathcal{FT}$ recognizing the *factorAD FM* which is the set of all lexmin words from the antidictionary M. Then we state step 3 of Algorithm 2 in the following form:

Step 3. Add all possible edges to $\mathcal{FT}$, following the rule:
the edge $(u, c, Lexmin(v))$ should be added if
u is not terminal, and
u has no outgoing edge labeled by c, and
v is the longest suffix of uc such that $Lexmin(v)$ is a vertex of $\mathcal{FT}$,

and apply Algorithm 2 to the trie $\mathcal{FT}$.

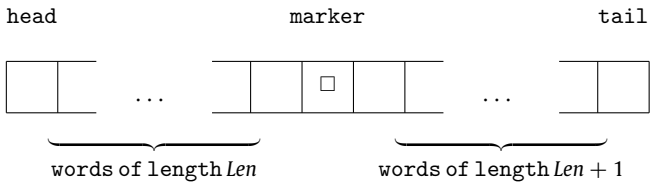
Nevertheless, an efficient implementation of this main idea is far from trivial. Below we give such an implementation for the particular case we are interested in: when the considered symmetric language with a finite antidictionary is a k-approximation of some power-free language. First we present Algorithm 3 which builds the trie recognizing the factorAD of the k-antidictionary of a given power-free language. For convenience we will identify all terminal vertices of this trie and adjust a unique label, say, '0', to the obtained vertex. Algorithm 3 uses two data structures: apart from the trie, we need an auxiliary queue $\mathcal{Q}$ to store permitted lexmin words which are potential roots of the minimal forbidden words. We insert the marker $\square$ to the queue in the points where the length of words increases. Also, it is convenient to store with each word in the queue the number of different letters in it. We suppose that $\Sigma = \{1, \ldots, m\}$. The function $Forbidden(w)$ returns 1 if the reading of the word w by the current trie $\mathcal{FT}$ stops in the terminal vertex (it does not matter, whether $\mathcal{FT}$ reads the whole word w or not), and 0 otherwise. Arithmetic operations with exponents were discussed in the preliminaries.

Algorithm 3.
Input: size $m > 1$ of the alphabet, exponent $\beta > 1^+$, length $k > 1$.
Output: trie $\mathcal{FT}$ for the factorAD FM_k, where M_k is the k-antidictionary for the β-free language over the m-letter alphabet.
Initialization: trie $\mathcal{FT}$ consists of a single initial (and non-terminal) vertex, queue $\mathcal{Q}$ is empty.

```
01. push □ to Q
02. push ('1'; 1) to Q
03. Len ← 0                    % current length of the root
04. pop W from Q               % start of the main cycle
05. if W = □ goto 16
06.                            % assume that W = (w; t)
07. u ← w^⌈Len·β⌉/|w|
08. if Forbidden(Lexmin(v)) = 0 for any suffix v of u
09.    add u to FT
10. if Len < ⌊k/β⌋
11.    for any letter c such that c ≤ t and Forbidden(wc) = 0
12.        push (wc; t) to Q
13.    if t < m
14.        push (w(t + 1); t + 1) to Q
15. goto 4
16. if Len < ⌊k/β⌋
17.    Len ← Len + 1
18.    push □ to Q
19.    goto 4
20. return FT
```

Lemma 4.4. *Given integers $m, k > 1$ and an exponent $\beta > 1^+$, Algorithm 3 outputs the trie recognizing the factorAD of the k-antidictionary for the β-free language over the m-letter alphabet.*

Proof. First we note that at any moment during the work of Algorithm 3 the queue consists of words of length *Len* and *Len* + 1 separated by a marker:



Now let us analyze the content of the trie and the queue at the moment when the algorithm pops the marker from $\mathcal{Q}$. We show that $\mathcal{FT}$ recognizes exactly the minimal forbidden lexmin words whose roots have the length less than or equal to *Len*, while $\mathcal{Q}$ consists of all permitted lexmin words of length *Len* + 1. These conditions obviously hold when $Len = 0$ (the first iteration of the main cycle). So, we have the inductive base and turn to the inductive step. To prove it, we assume that the algorithm processed the marker, increased *Len* to *Len* + 1 and the queue consists of all permitted lexmin words of length *Len* + 1, followed by the marker. Suppose that now the word w is being processed.

The word u constructed in line 7 is lexmin since w is, and is either minimal forbidden or contains a proper minimal forbidden factor with the root of length less than *Len* + 1. Such a factor, if it exists, is a prefix of some suffix v of u. Using the inductive assumption for $\mathcal{FT}$, we see that $\mathcal{FT}$ will accept some prefix of the word $Lexmin(v)$ giving then $Forbidden(Lexmin(v)) = 1$. Hence, u is added to $\mathcal{FT}$ if and only if it is a minimal forbidden lexmin word with the root w of length *Len* + 1. Since all potential roots of this length are contained in the queue, we get the inductive step for the trie.

Let c be a letter. Note that the word wc is lexmin if and only if w is lexmin and either w contains c, or w consists of the letters $0, \ldots, c-1$, but not c. Now an easy check of the lines 11–14 finishes the inductive step for the queue. □

Remark 4.5. Processing a word u, Algorithm 3 performs $O(|u|^2)$ operations (this is the complexity of checking the condition in line 8). The total number of processed words depends on k exponentially at the base approximately $\alpha^{1/\beta}$, where α is the growth rate of the given k-approximation.

Finally we present Algorithm 4 which builds the automaton $\mathcal{A}/\pi$ from the trie $\mathcal{FT}$. We adopt the linear-time implementation of Algorithm 2 discussed in Section 4.1. To do this, we define the *lexmin failure function* on the vertices of $\mathcal{A}/\pi$ by the rule $lf(u) = Lexmin(f(u))$. Further, let σ_u denote the partial permutation of the alphabet such that $\sigma_u(f(u)) = lf(u)$ and the domain of σ_u consists of the letters of $f(u)$ only. This permutation allows us to encode the transitions from $f(u)$ with the ones from $lf(u)$. In this way, the transitions from $lf(u)$ can be used to calculate the transitions from u. The value $\sigma_u[c] = 0$ indicates that the letter c is not in the domain of σ_u. The array of flags $Back_u$ serves to distinguish forward and backward edges from u (as we shall see later, our algorithm adds edges of both types to $\mathcal{FT}$). For a vertex u, the function $Last(u)$ returns the maximal letter occurring in u. Since u is a lexmin word, $Last(u)$ equals the number of different letters in u.

Algorithm 4.
Input: trie $\mathcal{FT}$ recognizing the factorAD FM_k.
Output: the factor-automaton $\mathcal{A}/\pi$ of the FAD-automaton $\mathcal{A}$.
Initialization: $\mathcal{A}/\pi$ is initialized by $\mathcal{FT}$; for any nonterminal vertex u of $\mathcal{A}/\pi$, $lf(u)$ is undefined, $Back_u = (0, \ldots, 0)$, $\sigma_u = (0, \ldots, 0)$.

```
01. add the edges (λ, 2, '1'), ..., (λ, m, '1')
02. for each nonempty nonterminal vertex u of A/π
        in width-first order starting with u = '1'
03.     for t = 1, ..., Min{m, Last(u) + 1}
04.         if no edge of the form (u, t, v) exists
05.             d ← σ_u[t]
06.             if d = 0
07.                 d ← Last(lf(u)) + 1
08.             add the edge (u, t, w) such that (lf(u), d, w) is an edge
09.             Back_u[t] ← 1
10.         else                    % (u, t, v) is a forward edge
11.             if v is nonterminal
12.                 σ_ut ← σ_u
13.                 d ← σ_u[t]
14.                 if d = 0
15.                     d ← Last(lf(u)) + 1
16.                     σ_ut[t] ← d
17.                 lf(ut) ← w for w such that (lf(u), d, w) is an edge
```

```
18.                 û ← f(u)
19.                 while Back_û[σ_ut[t]] = 1
20.                       σ̂ ← σ_û
21.                       if σ̂[σ_ut[t]] = 0
22.                           σ̂[σ_ut[t]] ← Last(lf(û)) + 1
23.                       σ_ut ← σ_ut ∘ σ̂
24.                       û ← lf(û)
25.     for t = Last(u) + 2, ..., m
26.         add the edge (u, t, w) such that (u, Last(u) + 1, w) is an edge
27.         Back_u[t] ← Back_u[Last(u) + 1]
28. delete the terminal vertex
29. return A/π
```

Lemma 4.6. *Given the trie $\mathcal{FT}$ recognizing the factorAD to a k-approximation for a power-free language, Algorithm 4 outputs the factor-automaton $\mathcal{A}/\pi$ of the FAD-automaton $\mathcal{A}$ recognizing the given k-approximation.*

Proof. It is easy to see that at the moment when we start the iteration of the outer cycle for the vertex u, we already know $lf(u)$ and σ_u, as well as $Back_v$ and all transitions from v for any vertex v with $|v| < |u|$. So, let us consider this particular iteration and prove that all transitions from u are calculated correctly.

First consider the "additional" cycle in lines 25–27. For any value of t in the given range the word ut is not lexmin and is π-equivalent to the lexmin word uc, where $c = Last(u) + 1$. Hence, any suffix of ut is π-equivalent to the suffix of uc of the same length. Note that the words uc and ut are not forbidden. Indeed, u is not forbidden, and the letter c (respectively, t) occurs in uc (respectively, ut) only once and then it cannot be the last letter of a fractional power. Hence, there are transitions (u, c, v_1) and (u, t, v_2) in $\mathcal{A}$. By Remark 4.1 and the definition of π, the words v_1 and v_2 are π-equivalent. By (*), $\mathcal{A}/\pi$ contains the transitions (u, c, w) and (u, t, w), where $w = Lexmin(v_1) = Lexmin(v_2)$. So, the edge in line 26 is added correctly. It remains to note that the edges (u, c, w) and (u, t, w) are either both forward, or both backward, depending on the length of w.

The above argument also shows the correctness of line 1. Here all added edges are obviously forward, so we do not change the array $Back_\lambda$.

The main cycle for u (lines 3–24) contains two nontrivial parts: the calculation of backward edges from u (lines 5–9) and the calculation of auxiliary functions for the descendants of u in the trie (lines 12–24). Consider the calculation of edges. If no edge from u with the label t is in the trie, then the automaton $\mathcal{A}$ contains a backward edge (u, t, v_1) such that $(f(u), t, v_1)$ is an edge also. Since the words $f(u)$ and $lf(u)$ are π-equivalent, there exist a letter d and a word v_2 in the π-class of v_1 such that $(lf(u), d, v_2)$ is an edge in $\mathcal{A}$. Let $w = Lexmin(v_1) = Lexmin(v_2)$. By (*), (u, t, w) and $(lf(u), d, w)$ are the edges in $\mathcal{A}/\pi$.

How to determine the letter d? The words $f(u)t$ and $lf(u)d$ are π-equivalent. Since $\sigma_u(f(u)) = lf(u)$, either we have $d = \sigma_u[t]$, or t does not occur in $f(u)$, implying that d does not occur in $lf(u)$. In the latter case, $d > Last(lf(u))$, because $lf(u)$ is a lexmin word. But all edges from $lf(u)$ with the labels $Last(lf(u)) + 1, \ldots, m$ have the same destination (the cycle in lines 25–27 was discussed above). So, if t does not occur in $f(u)$, we can choose $d = Last(lf(u)) + 1$. Hence, the backward edges are added correctly.

Now let (u, t, ut) be an edge of the trie $\mathcal{FT}$. We should calculate the vertex $lf(ut)$ and the permutation σ_{ut}. Recall that $f(ut) = v$, where $(f(u), t, v)$ is an edge in $\mathcal{A}$. As in the argument above, we choose the letter d such that $(lf(u), d, w)$ is an edge in $\mathcal{A}/\pi$, where $w = Lexmin(v)$. Then, $lf(ut) = w$ by definition. So, the calculation of $lf(ut)$ works correctly.

The calculation of σ_{ut} is more tricky. No problem arises if the edge $(lf(u), d, w)$ of $\mathcal{A}/\pi$ is forward. Since the word $lf(u)$ is lexmin, the word $lf(ut) = w = lf(u)d$ is lexmin also (the letter d was chosen such that $d \le Last(lf(u)) + 1$). Then $(lf(u), d, w)$ is a forward edge in $\mathcal{A}$. Hence, the edge $(f(u), t, v)$ in $\mathcal{A}$ is forward as well, that is, $f(ut) = v = f(u)t$. We know that $\sigma_u(f(u)) = lf(u)$ and the words $f(ut)$ and $lf(ut)$ are π-equivalent. So, to obtain the required equality $\sigma_{ut}(f(ut)) = lf(ut)$ it suffices to let $\sigma_{ut} = \sigma_u$ and additionally set $\sigma_{ut}[t] = Last(lf(u) + 1)$ if $\sigma_u[t]$ is undefined. This is exactly what we get if the `while` cycle in lines 19–24 is skipped.

Now let the edge $(lf(u), d, w)$ of $\mathcal{A}/\pi$ be backward. It is not easy to reconstruct such a situation; for instance, in the ternary $(7/4)^+$-free language those backward edges first appear in the factor-automaton for the 41-approximation (see Example 4.9 below). If $(lf(u), d, w)$ is backward, then $\mathcal{A}$ contains the backward edges $(lf(u), d, w_1)$ and $(f(u), t, v_1)$ for some π-equivalent words w_1 and v_1. Choose a positive integer i such that $\mathcal{A}$ contains the backward edges $(f(u), t, v_1), \ldots, (f^i(u), t, v_i)$ and the forward edge $(f^{i+1}(u), t, v_{i+1})$. Then we obtain $f(ut) = v_{i+1} = f^{i+1}(u)t$. By induction, we immediately get $lf^j(u) = Lexmin(f^j(u))$ for any $j = 1, \ldots, i + 1$. Hence, the automaton $\mathcal{A}/\pi$ contains the backward edges $(lf(u), d_1 = d, w_1), \ldots, (lf^i(u), d_i, w_i)$ and the forward edge $(lf^{i+1}(u), d_{i+1}, w_{i+1})$ such that $w_j = Lexmin(v_j)$ for any $j = 1, \ldots, i + 1$. So we have $lf(ut) = w_{i+1} = lf^{i+1}(u)d_{i+1}$ by definition.

We are ready to calculate the permutation σ_{ut}. It will be equal to the superposition of the "extended" versions of the permutations $\sigma_u, \sigma_{lf(u)}, \ldots, \sigma_{lf^i(u)}$. Since the function f preserves π-equivalence, the equality $\sigma_u(f(u)) = lf(u)$ implies $\sigma_u(f^2(u)) = f(lf(u)), \ldots, \sigma_u(f^{i+1}(u)) = f^i(lf(u))$. So if we extend σ_u to the new permutation $\sigma_{u,t}$ by setting $\sigma_{u,t}[t] = Last(lf(u) + 1)$ if $\sigma_u[t]$ is undefined, we get $\sigma_{u,t}(f^{i+1}(u)t) = f^i(lf(u)d_1)$. Then we continue to extend other permutations

from the list above and apply them consecutively to turn f's to lf's:

$$\sigma_{lf(u),d_1}(\sigma_{u,t}(f^{i+1}(u)t)) = \sigma_{lf(u),d_1}(f^i(lf(u)d_1)) = f^{i-1}(lf^2(u)d_2),$$
$$\cdots$$
$$\sigma_{lf^i(u),d_i}(\cdots(\sigma_{lf(u),d_1}(\sigma_{u,t}(f^{i+1}(u)t)))) = lf^{i+1}(u)d_{i+1}.$$

Therefore, the permutation $(\sigma_{u,t} \circ \sigma_{lf(u),d_1} \circ \cdots \circ \sigma_{lf^i(u),d_i})$ transforms $f(ut)$ to $lf(ut)$, whence it is equal to σ_{ut}. It remains to note that exactly this permutation is calculated in the `while` cycle of the algorithm. □

Remark 4.7. Processing a vertex, Algorithm 4 performs $O(1)$ operations except for the case when the `while` cycle is entered. In this special case the number of operations is bounded by the depth of the trie $\mathcal{FT}$ which is logarithmic to the size of the factor-automaton $\mathcal{A}/\pi$.

Summarizing the results of this section, we can formulate the enhanced method to get an upper bound of the growth rate of a power-free language: choose the number k, calculate the trie recognizing the factorAD of the k-antidictionary of the target language (Algorithm 3), build the factor-automaton $\mathcal{A}/\pi$ from this trie (Algorithm 4), and apply Algorithm 1 (or 1Q) to this factor-automaton. The time and space spent on each step of the enhanced method (building the trie, building the automaton, calculating the growth rate) are reduced by the factor of approximately $|\Sigma|!$ with respect to the corresponding steps of the simple method.

4.4. Two examples and a property of conjugates

Here we provide an example of the work of Algorithms 3 and 4, an example for the special case of Algorithm 4, and a statement concerning this special case.

Example 4.8. We illustrate the work of Algorithms 3 and 4 with a simple example. Let us take the 4-approximation of the ternary square-free language. The Algorithm 3 works as follows:

Queue	Operations
□; ($'1'$, 1)	$Len \leftarrow 1$
($'1'$, 1); □	$u = '11'$ is added to $\mathcal{FT}$; the condition $Len < \lfloor k/\beta \rfloor = 2$ passed
	($'11'$, 1) is not added to $\mathcal{Q}$ since $Forbidden('11') = 1$
	($'12'$, 2) is added to $\mathcal{Q}$
□; ($'12'$, 2)	$Len \leftarrow 2$
($'12'$, 2); □	$u = '1212'$ is added to $\mathcal{FT}$; the condition $Len < \lfloor k/\beta \rfloor$ failed
□	Finished

As a result, we get the trie shown in Fig. 1(left). Now consider the work of Algorithm 4 (the resulting automaton is given in Fig. 1, right):

Vertex u	t	Operations
λ (line 1)		The edges (λ, 2, $'1'$) and (λ, 3, $'1'$) are added
$'1'$	1	No (an existing edge to the terminal vertex)
	2	$\sigma_{12} \leftarrow (0, 0, 0)$, $d \leftarrow Last(\lambda) + 1 = 1$, then $\sigma_{12} \leftarrow (0, 1, 0)$
		$lf(12) \leftarrow '1'$ from the edge $(\lambda = lf(1), 1, '1')$
		`while` cycle is skipped
	3	The edge ($'1'$, 3, $'12'$) is added (line 26)
$'12'$	1	$\sigma_{121} \leftarrow (0, 1, 0)$, $d \leftarrow Last('1') + 1 = 2$, then $\sigma_{121} \leftarrow (2, 1, 0)$
		$lf(121) \leftarrow '12'$ from the edge $('1' = lf(2), 2, '12')$
		`while` cycle is skipped
	2	$d \leftarrow 1$
		the edge ($'12'$, 2, $'0'$) is added from $('1' = lf(2), 1, '0')$
		$Back_{12}[2] \leftarrow 1$
	3	$d \leftarrow 2$
		the edge ($'12'$, 3, $'12'$) is added from $('1' = lf(2), 2, '12')$
		$Back_{12}[3] \leftarrow 1$
$'121'$		The edges ($'121'$, 1, $'0'$) and ($'121'$, 3, $'12'$) are added
		by the same rules as above

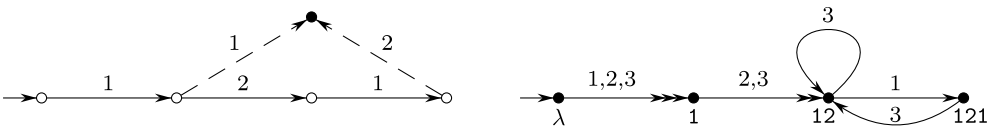


Fig. 1. The trie $\mathcal{FT}$ (left) and the automaton $\mathcal{A}/\pi$ (right) for the 4-approximation of the ternary 2-free language.

Example 4.9. Here is the situation in which the `while` cycle of Algorithm 4 is used. The 41-antidictionary of the ternary $(7/4)^+$-free language contains the word

$$u = xyx = 121321232131213212\ 31323\ 121321232131213212.$$

Consider the root xy of u. An interesting detail can be observed: the right cyclic shift of xy (that is, the word obtained from xy by moving its first letter to the end) is not a root of a minimal forbidden word. Indeed, the word

$$v = 213212321312132123\ 132\mathbf{31\ 213212321312132123}$$

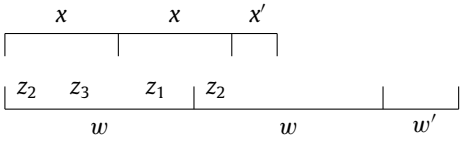
has a forbidden suffix of length 20 with the period 11, written in boldface. Note that the longest proper prefix of v is the longest proper suffix of u, that is, the first candidate for $f(u)$. But in fact, the longest prefix of v, which is a prefix of a word from the 41-antidictionary considered, is of length 18. If we take the 19-letter prefix of u as u' and the 18-letter prefix of v as v', we obtain $f(u') = v'$. At the same time, the edge from u' with the label 1 is forward, while the edge from v' with the same label is backward. So, the edge from $lf(u') = Lexmin(v')$ with the label $2 = \sigma_{u'}[1]$ is also backward, and the condition of the `while` cycle for u' is satisfied.

The effect considered in Example 4.9 can take place only if $\beta < 2$. To show this, we prove the following property of conjugate words.

Proposition 4.10. *Suppose that $\beta \geq 2$, L is the β-free language over the alphabet Σ, and w is the root of a minimal forbidden word for L. Then any conjugate of w is also the root of a minimal forbidden word for L.*

Proof. Let $w = x1$ and $v = 1x$. It is enough to prove that v is the root of a minimal forbidden word $u = vvv'$. (Recall that u has the period $|v|$ and the exponent greater than or equal to 2. It does not matter if the word v' is longer or shorter than v, or even empty.) The longest proper suffix of u is a permitted word, because it is a longest proper prefix of the minimal forbidden word www'. So, all forbidden factors of u are prefixes of u. Then u is either a minimal forbidden word, or contains some proper prefix xxx', which is a minimal forbidden word. Below we assume that u has such prefix xxx' and obtain contradictions in all possible cases.

Let $\beta = 2 + \varepsilon$, $\varepsilon \geq 0$. If $|xxx'| \leq (1 + \varepsilon)|w|$, then the suffix vv' of u contains a forbidden factor xxx', a contradiction. So, $|xxx'| > (1 + \varepsilon)|w|$. On the other hand, $|x'| < |w'|$, because all proper suffixes of xxx' are permitted words and $|x| < |w|$. Thus, $|xx| > |w|$, and the mutual location of the parts of u can be represented as follows (it does not matter whether x' and w' intersect or not):



Let $w = xz_1$ and $x = z_1z_2$. Since w begins with z_2 and $|x| > |z_2|$, we can write $x = z_2z_3$. By a well-known property of words (see [15, Proposition 1.3.4]), the equalities $x = z_1z_2$ and $x = z_2z_3$ imply $z_1 = pq, z_2 = (pq)^np, z_3 = qp$ for some words $p \neq \lambda$ and q, and some $n \geq 0$. Hence, $x = (pq)^{n+1}p$, $w = (pq)^{n+1}ppq$. The second w begins with $z_2x' = (pq)^npx'$, yielding that one of the words qp, x' is a prefix of the other one. On the other hand, x' is a prefix of x, and then one of the words pq, x' is a prefix of the other one.

If $|pq| \leq |x'|$ then the words pq and qp are both prefixes of x'. Hence, $pq = qp$, yielding that p and q are powers of some word r. Then w is also a power of r, and u has the exponent much bigger, than β, a contradiction. Now let $|x'| < |pq|$. Then $|x'| < |x|$ and $\varepsilon < 1$. Note that ww contains the factor $ppqp$. Since x' is a prefix of qp, this factor begins with ppx'. But x' is also a prefix of pq. If p is a prefix of x', then u contains p^3, which is impossible since $\varepsilon < 1$. Otherwise, x' is a prefix of p, and ppx' is a fractional power, the exponent of which is greater than the exponent of xxx'. This contradiction finishes the proof. □

Corollary 4.11. *Suppose that $\beta \geq 2$ and $\mathcal{A}$ is the FAD-automaton for the k-approximation of the β-free language over some alphabet. Then for any nonempty vertex u of $\mathcal{A}$ the failure function f returns the longest proper suffix of u. In particular, if u has a forward outgoing edge, labeled by c, then $f(u)$ also has a forward outgoing edge with this label. Hence, the `while` cycle of Algorithm 4 is never entered.*

Proof. Let c be the first letter of u. Since u is a vertex, it is a prefix of some minimal forbidden word with the root cw. Then by Proposition 4.10 the k-antidictionary contains a minimal forbidden word with the root wc. The longest proper suffix of u is a prefix of this forbidden word, and hence, a vertex of $\mathcal{A}$. So, this suffix equals $f(u)$ by definition. □

Hence, for exponents $\beta \geq 2$ one can simplify Algorithm 4, discarding the `while` cycle, which is never used.

5. Computer-assisted results

In this section we present the results of extensive computer assisted studies of growth rates of power-free languages over alphabets having from 2 to 10 letters. These studies allowed us to significantly improve all previously known upper bounds of growth rates, obtain a number of new bounds and discover some interesting facts and patterns.

During the studies, more than a thousand factor-automata were constructed for different values of m, β, and k, so that the structural properties of these automata should be quite common among all automata of this type. We start with two such properties.

Fact 5.1. *All automata we have obtained contain only one non-singleton scc.*

Thus, the condition of Theorem 3.7(2) is always satisfied, yielding fast convergence for Algorithm 1Q. The next property shows that the procedure of adding loops can be skipped for both Algorithms 1 and 1Q in order to increase the rate of convergence (this rate depends on the ratio α/γ for the adjacency matrix, see Remark 3.3).

Fact 5.2. *In almost all cases the non-singleton scc of recognizing automata has the imprimitivity number* 1. *The binary β-free languages with $2^+ \le \beta \le 7/3$ provide the only exceptions.*

Those "exceptional" languages are quite specific. The study of them by Algorithms 1 and 1Q is interesting only as a test for these algorithms, because the growth rates of all exceptional languages are equal to 1 (moreover, they have polynomial complexity, see [11]). We also point out that probably no other power-free languages of any subexponential complexity exist.

We note that the discovered exceptions were predictable. In [20], the author gave a description of all k-approximations of the famous Thue–Morse language, which forms a proper subset of the 2^+-free binary language. A precise form of the non-singleton scc in the automaton for any such k-approximation was found. This scc has the imprimitivity number of the order $\Theta(k)$. On the other hand, if a $(7/3)$-free binary word w can be extended to both sides to an arbitrarily long $(7/3)$-free binary word, then w must belong to the Thue–Morse language [19]. Hence, the k-approximations of the binary languages from 2^+-free to $(7/3)$-free are expected to behave in a similar way as the k-approximations of the Thue–Morse language. We checked the first few approximations to discover that the non-singleton scc's are the same for the Thue–Morse, 2^+-free, and $(7/3)$-free languages. By a careful analysis, this result probably can be extended to all k-approximations of those languages.

Finally we present Table 1 with a representative selection of our numerical results on upper bounds for growth rates of power-free languages. Up to now, only some partial cases were intensively studied. Thus, the bound 1.4576 for the binary 3-free language was obtained by Edlin [7], and the bound 1.3017886 for the ternary 2-free language belongs to Ochem and Reix [16]. In [11] the bound 1.2299 is given for the binary $(7/3)^+$-language. Most of our bounds are quite close to the growth rates of target languages. This conclusion is justified by the fast convergence of growth rates of approximating languages (the measure of convergence rate is described below) and is confirmed by some independent studies. For example, Richard and Grimm [18] used the method of differential approximants to suggest 1.301762 as the growth rate for ternary 2-free language with the admissible error $\pm 2 \cdot 10^{-6}$, and our last approximation is 1.30176188.

To estimate the rate of convergence we compare the growth rates obtained for different k-approximations of the same power-free language. We assume that the approximation error $|\alpha(L_k) - \alpha(L)|$ is multiplicatively dependent on the size of the factor-automaton $\mathcal{A}_k$, that is, when this size increases by some factor, this error decreases by some (other) factor. This assumption agrees with the obtained experimental results. For each k-approximation of a given language we calculate the value $\Delta(k) = \alpha(L_{k'}) - \alpha(L_k)$, where the factor-automaton $\mathcal{A}_{k'}$ is approximately half as large as $\mathcal{A}_k$. The absolute values and the behaviour of the function $\Delta(k)$ give a good idea about the tightness of the obtained upper bounds. Those absolute values for the best obtained approximation are given in the penultimate column of Table 1. The slowest convergence was observed for the $(7/5)^+$-free language over the 4-letter alphabet and the $(5/4)^+$-free language over the 5-letter alphabet. Extrapolating the behaviour of $\Delta(k)$ we conjecture the growth rate of the target power-free languages. These estimated growth rates are given in the last column of Table 1 (the digits given in parenthesis are not certain).

All computations were made using a PC with a 3,0 GHz CPU and 2 Gb of memory. For all studied power-free languages the time was not a critical resource: the only language the bound for which needed a few hours of calculation, was the quaternary $(7/5)^+$-free language. The space complexity in most cases was dominated by the allocation of the factor-automaton $\mathcal{A}/\pi$ with counters. Only for the minimal power-free languages over 8 or more letters the allocation of the queue $\mathcal{Q}$ in Algorithm 3 was space-critical.

From Table 1, it is easy to see that the growth rate of power-free languages, considered as a function of β over a fixed alphabet, has big jumps when β changes from r/s to $(r/s)^+$ for small s. These jumps are due to the "allowance" of short factors with the exponent r/s (like the factor a^3 when we move from 3-free to 3^+-free binary language). The obtained results allow us to formulate the following nice conjecture:

Conjecture 5.3. *The growth rate of power-free languages over a t-letter alphabet, $t \ge 3$, jumps by more than a unit at the points $(t-1)/(t-2), \ldots, 3/2, 2$.*

This conjecture is justified by our results for $t = 3, \ldots, 10$ (due to space constraints, only the results for 3, 4, and 5 letters are presented in Table 1).

Table 1
Upper bounds for the growth rates of power-free languages. The number $\bar{n}$ of iterations is counted for $\delta = 10^{-10}$.

$\lvert\Sigma\rvert$	Exp	k	$\lvert FM_k\rvert$	$\lvert\mathcal{A}_k/\pi\rvert$	$\bar{n}$	$\alpha(L_k)$	$\Delta(k)$	$\alpha(L)$ estim.
2	$(7/3)^+$	152	1 185 427	98 132 776	387	1.2206448166	$2\cdot10^{-9}$	1.220644814
2	5/2	155	977 954	85 891 045	339	1.2295017090	$7\cdot10^{-10}$	1.229501708
2	$(5/2)^+$	111	1 718 620	107 645 199	232	1.3663011099	$2\cdot10^{-10}$	1.366301109(7)
2	8/3	112	1 223 996	80 356 405	238	1.3762703910	$1\cdot10^{-10}$	1.376270391
2	$(8/3)^+$	99	1 538 273	89 115 544	189	1.4508611195	$3\cdot10^{-11}$	1.4508611195
2	3	108	1 241 239	83 006 914	190	1.457577286924	$1\cdot10^{-12}$	1.457577286923
2	3^+	73	1 414 760	64 323 422	133	1.795126408668	$2\cdot10^{-13}$	1.795126408668
3	$(7/4)^+$	116	1 646 916	85 722 260	310	1.24560931	$8\cdot10^{-7}$	1.245608
3	2	110	1 463 287	80 248 827	252	1.301761876	$4\cdot10^{-8}$	1.30176183
3	2^+	35	3 189 542	54 730 037	81	2.605879081	$3\cdot10^{-8}$	2.60587906
4	$(7/5)^+$	288	639 774	71 754 449	3 106	1.0695085	$1\cdot10^{-5}$	1.0694
4	(3/2)	231	767 047	73 577 456	1 132	1.0968025	$4\cdot10^{-6}$	1.06979
4	$(3/2)^+$	32	2 685 393	30 253 554	86	2.280572	$3\cdot10^{-5}$	2.28052
4	2	36	2 306 512	4 058 1402	70	2.62150802	$2\cdot10^{-8}$	2.6215080
4	2^+	27	1 532 902	20 583 499	56	3.72849442	$1\cdot10^{-8}$	3.72849442
5	$(5/4)^+$	141	1 200 217	58 923 876	440	1.158004	$5\cdot10^{-5}$	1.1577
5	(4/3)	146	1 056 309	58 902 152	423	1.164605	$1\cdot10^{-5}$	1.1645(6)
5	$(4/3)^+$	33	5 550 581	54 951 854	90	2.248963	$8\cdot10^{-5}$	2.2485
5	(3/2)	33	3 588 828	42 681 448	79	2.402445	$1\cdot10^{-5}$	2.40242
5	$(3/2)^+$	25	5 903 234	52 408 053	59	3.492828	$8\cdot10^{-6}$	3.49280
5	2	28	1 160 529	15 889 694	52	3.73253857	$1\cdot10^{-8}$	3.73253857
5	2^+	25	1 549 785	19 465 201	49	4.789850738	$2\cdot10^{-9}$	4.789850738
6	$(6/5)^+$	112	1 343 898	53 566 562	353	1.224727	$2\cdot10^{-5}$	1.2246
7	$(7/6)^+$	114	1 008 479	44 765 842	292	1.236909	$7\cdot10^{-6}$	1.2368(7)
8	$(8/7)^+$	119	537 208	27 405 270	275	1.234845	$2\cdot10^{-6}$	1.23483
9	$(9/8)^+$	116	163 583	8880 112	347	1.246678	$5\cdot10^{-7}$	1.24666
10	$(10/9)^+$	116	22 835	1 385 715	344	1.2393076	$2\cdot10^{-7}$	1.239307

Acknowledgements

The author is grateful to A. Bulatov for a number of useful remarks. Our special thanks to I. Gorbunova and A. Samsonov for the assistance in computational studies.

References

[1] J. Berstel, J. Karhumäki, Combinatorics on words: a tutorial, Bull. Eur. Assoc. Theoret. Comput. Sci. 79 (2003) 178–228.
[2] F.-J. Brandenburg, Uniformly growing k-th power free homomorphisms, Theoret. Comput. Sci. 23 (1983) 69–82.
[3] A. Carpi, On Dejean's conjecture over large alphabets, Theoret. Comput. Sci. 385 (2007) 137–151.
[4] M. Crochemore, F. Mignosi, A. Restivo, Automata and forbidden words, Inform. Process. Lett. 67 (3) (1998) 111–117.
[5] D.M. Cvetković, M. Doob, H. Sachs, Spectra of Graphs. Theory and Applications, 3rd ed., Johann Ambrosius Barth, Heidelberg, 1995.
[6] F. Dejean, Sur un Theoreme de Thue, J. Combin. Theory Ser. A 13 (1) (1972) 90–99.
[7] A. Edlin, The number of binary cube-free words of length up to 47 and their numerical analysis, J. Difference Equ. Appl. 5 (1999) 153–154.
[8] J.N. Franklin, Matrix Theory, Prentice-Hall Inc., Englewood Cliffs, NJ, 1968.
[9] F.R. Gantmacher, Application of the Theory of Matrices, Interscience, New York, 1959.
[10] C.D. Godsil, Algebraic Combinatorics, Chapman and Hall, New York, 1993.
[11] J. Karhumäki, J. Shallit, Polynomial versus exponential growth in repetition-free binary words, J. Combin. Theory. Ser. A 105 (2004) 335–347.
[12] Y. Kobayashi, Repetition-free words, Theoret. Comput. Sci. 44 (1986) 175–197.
[13] Y. Kobayashi, Enumeration of irreducible binary words, Discrete. Appl. Math. 20 (1988) 221–232.
[14] R. Kolpakov, On the number of repetition-free words, in: Electronic Proceedings of Workshop on Words and Automata, WOWA'06, S.-Petersburg, 2006, #6.
[15] M. Lothaire, Combinatorics on Words, Addison-Wesley, 1983.
[16] P. Ochem, T. Reix, Upper bound on the number of ternary square-free words, in: Electronic Proceedings of Workshop on Words and Automata, WOWA'06, S.-Petersburg, 2006, #8.
[17] A. Restivo, S. Salemi, Overlap-free Words on Two Symbols, in: Lect. Notes Comp. Sci., vol. 192, 1984, pp. 196–206.
[18] C. Richard, U. Grimm, On the entropy and letter frequencies of ternary square-free words, Electron. J. Combin. 11 (1) (2004) # R14.
[19] A.M. Shur, The structure of the set of cube-free Z-words in a two-letter alphabet, Izv. Ross. Akad. Nauk Ser. Mat. 64 (2000) 201–224 (Russian); English translation in Izv. Math. 64 (2000), 847–871.
[20] A.M. Shur, Combinatorial complexity of rational languages, Discr. Anal. Oper. Research, Ser. 1 12 (2) (2005) 78–99 (Russian).
[21] A.M. Shur, Comparing complexity functions of a language and its extendable part, RAIRO Theor. Inf. Appl. 42 (2008) 647–655.
[22] A.M. Shur, Calculating parameters and behavior types of combinatorial complexity for regular languages, Proc. Inst. Math. Mech. UB RAS 16 (2) (2010) 270–287.
[23] A.M. Shur, Combinatorial Complexity of Regular Languages, in: Lect. Notes Comp. Sci., vol. 5010, 2008, pp. 289–301.
[24] A. Thue, Über unendliche Zeichenreihen, Kra. Vidensk. Selsk. Skrifter. I. Mat.-Nat. Kl., Christiana 7 (1906) 1–22.
[25] J.D. Currie, N. Rampersad, A proof of Dejean's conjecture, http://arxiv.org/PScache/arxiv/pdf/0905/0905.1129v3.pdf.
[26] M. Rao, Last Cases of Dejean's conjecture, in: Proceedings of the 7th International Conference on Words, Salerno, Italy, –2009. #115.