On Combinatorial Generation of Prefix Normal Words

Péter Burcsi[1], Gabriele Fici[2], Zsuzsanna Lipták[3], Frank Ruskey[4], and Joe Sawada[5]

[1] Eötvös Loránd University, Budapest, Hungary, bupe@compalg.inf.elte.hu
[2] University of Palermo, Italy, gabriele.fici@math.unipa.it
[3] University of Verona, Italy, zsuzsanna.liptak@univr.it
[4] University of Victoria, Canada, ruskey@cs.uvic.ca
[5] University of Guelph, Canada, jsawada@uoguelph.ca

Abstract. A prefix normal word is a binary word with the property that no substring has more 1s than the prefix of the same length. This class of words is important in the context of binary jumbled pattern matching. In this paper we present an efficient algorithm for exhaustively listing the prefix normal words with a fixed length. The algorithm is based on the fact that the language of prefix normal words is a bubble language, a class of binary languages with the property that, for any word w in the language, exchanging the first occurrence of 01 by 10 in w results in another word in the language. We prove that each prefix normal word is produced in $O(n)$ amortized time, and conjecture, based on experimental evidence, that the true amortized running time is $O(\log(n))$.

1 Introduction

A binary word of length n is *prefix normal* if for all $1 \leq k \leq n$, no substring of length k has more 1s than the prefix of length k. For example, 1001010 is not prefix normal because the substring 101 has more 1s than the prefix 100. These words were introduced in [8], where it was shown that each binary word w has a canonical *prefix normal form* w' of the same length.

The study of prefix normal words and prefix normal forms is motivated by the string problem known as *binary jumbled pattern matching* (binary JPM). In that problem, we are given a text of length n over a binary alphabet, and two numbers x and y, and ask whether the text has a substring with exactly x 1s and y 0s. While the online version can be solved with a simple sliding window algorithm in $O(n)$ time, the offline version, where many queries are expected, has recently attracted much interest: here an index of size $O(n)$ can be generated which then allows answering queries in constant time [5]. However, the best construction algorithms up to date have running time $\mathrm{O}(n^2/\log n)$ [2, 13]. Several recent papers have yielded better results under specific assumptions, such as word level parallelism or highly compressible strings [1, 6, 10, 14], or for constructing an approximate index [7]; but the general case has not been improved. It was demonstrated in [1, 8] that prefix normal forms of the text can be used to construct this index. JPM over an arbitrary alphabet has also been studied [4,5,11]. Moreover, several variants of the original problem have recently been introduced: approximate JPM [3], JPM in the streaming model [12], JPM on trees and graphs [6,9].

We note that the connection of the present paper to binary JPM is that of supplying a new approach: We do not present an improvement of the JPM problem, but we strongly believe that a better understanding of these words will eventually lead to better solutions for JPM.

Bubble languages are an interesting new class of binary languages defined by the following property: $\mathcal{L}$ is a bubble language if, for every word $w \in \mathcal{L}$, replacing the first occurrence of 01 (if any) by 10 results in another word in $\mathcal{L}$ [15,16,17]. A generic generation algorithm for bubble languages was given in [17], leading to Gray codes for each of these languages. The algorithm's efficiency depends only on a language-dependent subroutine, which in the best case leads to CAT (constant amortized time) generation algorithms. Many important languages are bubble languages, including binary necklaces and Lyndon words, and k-ary Dyck words.

In this paper, we show that prefix normal words form a bubble language and present an efficient generation algorithm which runs in $O(n)$ amortized time per word, and which yields a Gray code for prefix normal words. Generating these words naively takes $O(2^n \cdot n^2)$ time. Based on experimental evidence, we conjecture that the running time of our algorithm is in fact $\Theta(\log(n))$ amortized. We also give a new characterization of bubble languages in terms of a closure property in the computation tree of a certain generation algorithm for all binary words (Prop. 1). We prove new properties of prefix normal words and present a linear time testing algorithm for words which have been obtained from prefix normal words via a simple operation. We present several open problems in the last section.

Most proofs have been omitted for lack of space and will be contained in the full version of the paper.

2 Basics

A *binary word* (or *string*) $w = w_1 \cdots w_n$ over $\Sigma = \{0,1\}$ is a finite sequence of elements from Σ. Its length n is denoted by $|w|$. We denote by Σ^n the words over Σ of length n, and by $\Sigma^* = \cup_{n\geq 0}\Sigma^n$ the set of finite words over Σ. The empty word is denoted by ε. Let $w \in \Sigma^*$. If $w = uv$ for some $u, v \in \Sigma^*$, we say that u is a *prefix* of w and v is a *suffix* of w. A *substring* of w is a prefix of a suffix of w. A *binary language* is any subset $\mathcal{L}$ of Σ^*.

In the following, we will often write binary words $w \neq 1^n$ in a canonical form $w = 1^s 0^t \gamma$, where $\gamma \in 1\{0,1\}^* \cup \{\varepsilon\}$ and $s \geq 0, t \geq 1$. In other words, s is the length of the first, possibly empty, 1-run of w, t the length of the first 0-run, and γ the remaining, possibly empty, suffix. Note that this representation is unique. We call $1^s 0^t$ the *critical prefix* of w and $cr(w) = s + t$ the *critical prefix length* of w. We denote by $|w|_c$ the number of occurrences in w of character $c \in \{0,1\}$, and by $\mathcal{B}_d^n$ the set of all binary strings w of length n such that $|w|_1 = d$ (the *density* of w is d). We denote by $swap(w, i, j)$ the string obtained from w by exchanging the characters in positions i and j.

2.1 Prefix Normal Words

Let $w \in \Sigma^*$. For $i = 0, \ldots, n$, we set

- $P(w, i) = |w_1 \cdots w_i|_1$, the number of 1s in the i-length prefix of w.
- $F(w, i) = \max\{|u|_1 : u$ is a substring of w and $|u| = i\}$, the maximum number of 1s over all substrings of length i.

Definition 1. *A binary word w is* prefix normal *if, for all $1 \leq i \leq |w|$, $F(w, i) = P(w, i)$. In other words, a word is prefix normal if no substring contains more* 1*s than the prefix of the same length.*

We denote by $\mathcal{L}_{\text{PN}}$ the language of prefix normal words. In [8] it was shown that for every word w there exists a unique word w', called its *prefix normal form*, or PNF(w), such that for all $1 \leq i \leq |w|$, $F(w, i) = F(w', i)$, and w' is prefix normal. Therefore, a prefix normal word is a word coinciding with its prefix normal form. In the following table we list all prefix normal words of length 5 followed by the set of binary words w such that PNF$(w) = w'$ (i.e., its equivalence class):

$11111 \Rightarrow \{11111\}$	$11000 \Rightarrow \{11000, 011000, 00110, 00011\}$
$11110 \Rightarrow \{11110,\ 01111\}$	$10101 \Rightarrow \{10101\}$
$11101 \Rightarrow \{11101,\ 10111\}$	$10100 \Rightarrow \{10100, 01010, 00101\}$
$11100 \Rightarrow \{11100,\ 01110,\ 00111\}$	$10010 \Rightarrow \{10010, 01001\}$
$11011 \Rightarrow \{11011\}$	$10001 \Rightarrow \{10001\}$
$11010 \Rightarrow \{11010, 10110, 01101, 01011\}$	$10000 \Rightarrow \{10000, 01000, 00100, 00010, 00001\}$
$11001 \Rightarrow \{11001, 10011\}$	$00000 \Rightarrow \{00000\}$.

Several methods were presented in [8] for testing whether a word is prefix normal; however, all ran in quadratic time in the length of the word. One open problem given there was that of enumerating prefix normal words (counting). The number of prefix normal words of length n can be computed by checking for each binary word whether it is prefix normal, i.e. altogether in $O(2^n \cdot n^2)$ time. In this paper, we present an algorithm that is far superior in that it generates only prefix normal words, rather than testing every binary word; it runs in $O(n)$ time per word; and it generates prefix normal words in cool-lex order, constituting a Gray code (subsequent words differ by a constant number of swaps or flips).

2.2 Bubble Languages and Combinatorial Generation

Here we give a brief introduction to bubble languages, mostly summarising results from [15, 17]. We also give a new characterization of bubble languages in terms of the computation tree of a generation algorithm (Prop. 1).

```
Algorithm Generate(s, t, γ)
(* current string resides in array w *)
1.  if $s > 0$ and $t > 0$
2.    then for $i = 1, 2, \ldots, t$
3.            do $w \leftarrow swap(w, s, s+i)$
4.               $Generate(s-1, i, 10^{t-i}\gamma)$
5.               $w \leftarrow swap(w, s, s+i)$
6.  $Visit()$
```

Fig. 1. The Recursive Swap Generation Algorithm

Definition 2. *A language $\mathcal{L} \subseteq \{0,1\}^*$ is called* a bubble language *if, for every word $w \in \mathcal{L}$, exchanging the first occurrence of* 01 *(if any) by* 10 *results in another word in $\mathcal{L}$.*

For example, the languages of binary Lyndon words and necklaces are bubble languages. A language $\mathcal{L} \subseteq \{0,1\}^n$ is a bubble language if and only if each of its fixed-density subsets $\mathcal{L} \cap \mathcal{B}_d^n$ is a bubble language [15]. This implies that for generating a bubble language, it suffices to generate its fixed-density subsets.

Next we consider combinatorial generation of binary strings. Let w be a binary string of length n. Let d be the number of 1s in w, and let $i_1 < i_2 < \ldots < i_d$ denote the positions of the 1s in w. Clearly, we can obtain w from the word 1^d0^{n-d} with the following algorithm: first swap the last 1 with the 0 in position i_d, then swap the $(d-1)$st 1 with the 0 in position i_{d-1} etc. Note that every 1 is moved at most once, and in particular, once the k'th 1 is moved into the position i_k, the suffix $w_{i_k} \cdots w_n$ remains fixed for the rest of the algorithm.

These observations lead us to the following generation algorithm (Fig. 1), which we will refer to as *Recursive Swap Generation Algorithm* (like Alg. 1 from [17], but without the language-specific subroutine). It generates recursively all binary strings from $\mathcal{B}_d^n$ with fixed suffix γ, where $\gamma \in 1\{0,1\}^* \cup \{\varepsilon\}$, starting from the string $1^s0^t\gamma$. The call $Generate(d, n-d, \varepsilon)$ generates all binary strings of length n with density d.

The algorithm swaps the last 1 of the first 1-run with each of the 0s of the first 0-run, thus generating a new string each, for which it makes a recursive call. During the execution of the algorithm, the current string resides in a global array w. In the subroutine *Visit()* we can print the contents of this array, or increment a counter, or check some property of the current string. The main point of *Visit()* is that it touches every object once.

Let T_d^n denote the recursive computation tree of $Generate(d, n-d, \varepsilon)$. As an example, Fig. 2 illustrates the computation tree T_4^7 (ignore for now the highlighted words, see Sec. 3). The depth of the tree equals d, the number of 1s, while the maximum degree is $n-d$, the number of 0s. In general, for the subtree rooted at $v = 1^s0^t\gamma$, we have depth s and maximum degree t; the number of children of v is t, and v's ith child is $1^{s-1}0^i10^{t-i}\gamma$. Note that suffix γ remains unchanged in the entire subtree, that the computation tree is isomorphic to the computation tree of 1^s0^t, and that the critical prefix length strictly decreases along any downward path in the tree.

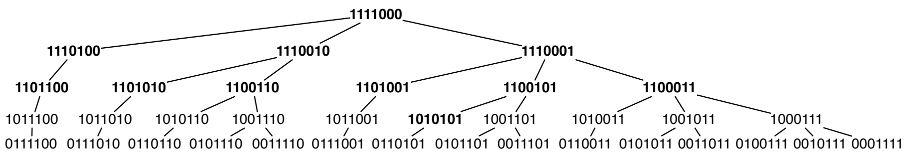

Fig. 2. The computation tree T_d^n for $n = 7, d = 4$. Prefix normal words in bold.

The algorithm performs a post-order traversal of the tree, yielding an enumeration of the strings of $\mathcal{B}_d^n$ in what is referred to as *cool-lex order* [15,17,19]. A pre-order traversal of the same tree, which implies moving line 4 of the algorithm before line 1, would yield an enumeration in *co-lex order*. A crucial property of cool-lex order is that any two subsequent strings differ by at most two swaps (transpositions), thus yielding a *Gray code* [15]. This can be seen in the computation tree T_d^n as follows. Note that in a post-order traversal of T_d^n, we have:

$$next(u) = \begin{cases} parent(u) & \text{if } u \text{ is rightmost child} \\ \text{leftmost descendant of } u\text{'s right sibling} & \text{otherwise.} \end{cases}$$

Let u, u' both be children of v. This means that for some $s, t, i, j \in \mathbb{N}$ and $\gamma \in 1\{0,1\}^* \cup \{\varepsilon\}$, we have $v = 1^s0^t\gamma$, $u = 1^{s-1}0^i10^{t-i}\gamma$, and $u' = 1^{s-1}0^j10^{t-j}\gamma$. Let v' be a descendant of v along the leftmost path, i.e. $v' = 1^k01^{s-k}0^{t-1}\gamma$ for some k. Then $v = swap(u, s, s+i)$ (parent), $u' = swap(u, s+i, s+j)$ (sibling), and $v' = swap(v, k, s+1)$ (descendant along leftmost path).

The following proposition states a crucial property of bubble languages with respect to the Recursive Swap Generation Algorithm. The proof follows immediately from the definition of bubble languages:

Proposition 1. *A language $\mathcal{L}$ is a bubble language if and only if, for every $d = 0, \ldots, n$, its fixed-density subset $\mathcal{L} \cap \mathcal{B}_d^n$ is closed w.r.t. parents and left siblings in the computation tree T_d^n of the Recursive Swap Generation Algorithm. In particular, if $\mathcal{L} \cap \mathcal{B}_d^n \neq \emptyset$, then it forms a subtree rooted in 1^d0^{n-d}.*

Using this property, the Recursive Swap Generation Algorithm can be used to generate *any* fixed-density bubble language $\mathcal{L}$, as long as we have a way of deciding, for a node $w = 1^s0^t\gamma$, already known to be in $\mathcal{L}$, which is its rightmost child (if any) that is still in $\mathcal{L}$. If such a child exists, and it is the kth child $u = 1^{s-1}0^k10^{t-k}\gamma$, then the bubble property ensures that all children to its left are also in $\mathcal{L}$. Thus, line 2. in the algorithm can simply be replaced by "for $i = 1, \ldots, k$". Moreover, this algorithm, which visits the words in the language in cool-lex order, will yield a Gray code, since because of this closure property, $next(u)$ will again either be the parent, or a node on the leftmost path of the right sibling, both of which are reachable within two swaps.

In [17], a generic generation algorithm was given which moves the job of finding this rightmost child k into a subroutine *Oracle(s, t, γ)*. If *Oracle(s, t, γ)*

runs in time $O(k)$, then we have a CAT algorithm. In general, this may not be possible, and a generic Oracle tests for each child from left to right whether it is in the language. Because of the bubble property, after the first negative test, we know that no more children will be in the language, and the running time of the algorithm is amortized that of the membership tester. The crucial trick is that we do not need a *general* membership tester, since all we want to know is which of the children of a node *already known to be in* $\mathcal{L}$ are in $\mathcal{L}$; moreover, the membership tester is allowed to use other information, which it can build up iteratively while examining earlier nodes.

3 Combinatorial Generation of Prefix Normal Words

In this section we prove that the set of prefix normal words $\mathcal{L}_{\text{PN}}$ is a bubble language. Then, by providing some properties regarding membership testing, we can apply the cool-lex framework to generate all prefix normal words of a given length n and density d in $O(f(n))$-amortized time, where $f(n)$ is the average critical prefix length of a prefix normal word of length n. (Proofs omitted.)

Lemma 1. *The language $\mathcal{L}_{PN}$ is a bubble language.*

The computation tree in Fig. 2 highlights the subtree corresponding to $\mathcal{L}_{\text{PN}} \cap \mathcal{B}_4^7$. Since $\mathcal{L}_{\text{PN}}$ is a bubble language, by Prop. 1 it is closed w.r.t. left siblings and parents. However, we still have to find a way of identifying the rightmost child of a node that is still in $\mathcal{L}_{\text{PN}}$.

The following lemma states that, given a prefix normal word w, in order to decide whether one of its children in the computation tree is prefix normal, it suffices to check the PN-property for one particular length only: the critical prefix length of the child node. Moreover, this check can be done w.r.t. γ only.

Lemma 2. *Let $w \in \mathcal{L}_{PN}$, with $w = 1^s0^t\gamma$, with $\gamma \in 1\{0,1\}^* \cup \{\varepsilon\}$. Let $\overline{\gamma} = \gamma 0^{s+t}$, i.e. γ padded with 0s to length n. Let $w' = swap(w, s, s+i)$. Then $w' \in \mathcal{L}_{PN}$, unless one of the following holds:*

1. *$\overline{\gamma}$ has a substring of length $s+i-1$ with at least s 1s, or*
2. *the string $w'_{s+i} \cdots w'_{2(s+i-1)}$ has at least s 1s.*

Moreover, the latter is the case if and only if $P(\overline{\gamma}, s+2(i-1)-t) \geq s-1$ (where by convention, we regard a prefix of negative length as the empty word).

Corollary 1. *Given $w = 1^s0^t\gamma \in \mathcal{L}_{PN}$. If we know $F(\gamma, j)$ and $P(\gamma, j)$ for all $j \leq s+t$, then it can be decided in constant time whether $w' = swap(w, s, s+i)$, for $i \leq t$, is prefix normal.*

Lemma 3. *Let $\gamma' = 10^r\gamma$, with $\gamma \in 1\{0,1\}^* \cup \{\varepsilon\}$. Then for all $i = 0, 1, \ldots, |\gamma'|$,*

$$F(\gamma', i) = \begin{cases} \max(P(\gamma', i), F(\gamma, i)) & \text{for } i \leq |\gamma| \\ \max(P(\gamma', i), F(\gamma, |\gamma|)) & \text{for } i > |\gamma|. \end{cases}$$

Corollary 2. *The F-function of γ for node $w = 1^s0^t\gamma$, up to entry $s+t$, can be computed in time $O(s+t)$ based on the F-function of w's parent node.*

By applying these results, the algorithm *GeneratePN($d, n-d, \varepsilon$)* will generate $\mathcal{L}_{\text{PN}} \cap \mathcal{B}_d^n$ in cool-lex Gray code order, see Fig. 3. Starting from the left child and proceeding right (with respect to the computation tree T_n^d), the algorithm will make a recursive call finding a child which is not prefix normal. The membership test is done in the subroutine *isPN*, which uses the conditions of Lemma 2. The algorithm maintains an array F which contains the maximum number of 1s in i-length substrings of γ (the F-function of γ), and a variable z. Before testing the first child, in *update($F, s+t$)*, it computes the current γ's F-function based on the parent's (Corollary 2). Note that it is not necessary to compute all of the F-function, since all nodes in the subtree have critical prefix length smaller than $s+t$, thus this update is done only up to length $s+t$. After the recursive calls to the children, the array is restored to its previous state in *restore($F, s+t$)*. The variable z contains the number of 1s in the prefix of γ which is spanned by the substring of case 2. of Lemma 2, for the first child. It is updated in constant time after each successful call to *isPN*, to include the number of 1s in the two following positions in γ.

```
Algorithm GeneratePN($s, t, \gamma$)
(* $w = 1^s0^t\gamma$ must be prefix normal *)
1.  if $s > 0$ and $t > 0$
2.    then update($F, s+t$)
3.         $z \leftarrow P(\gamma, s-t)$
4.         $i \leftarrow 1$
5.         while $i \leq t$ and isPN(swap($w, s, s+i$))
6.            do $w \leftarrow$ swap($w, s, s+i$)
7.               GeneratePN($s-1, i, 10^{t-i}\gamma$)
8.               update($z$)
9.               $i \leftarrow i+1$
10.              $w \leftarrow$ swap($w, s, s+i$)
11.        restore($F, s+t$)
12. Visit()
```

Fig. 3. Algorithm generating all prefix normal words in the subtree rooted in $1^s0^t\gamma$.

By concatenating the lists of prefix normal words with densities $0, 1, \ldots, n$, we obtain an exhaustive listing of $\mathcal{L}_{\text{PN}} \cap \Sigma^n$, see Fig. 4. As an example, *GeneratePN(5)* produces the following list of prefix normal words: 00000, 10000, 10100, 10010, 10001, 11000, 11010, 10101, 11001, 11100, 11011, 11101, 11110, 11111. Since the fixed-density listings are a cyclic Gray code (Theorem 3.1 from [15]), it follows that this complete listing is also a Gray code. In fact, if the fixed-density listings are listed by the odd densities (increasing), followed by the even densities (decreasing), the resulting listing would be a cyclic Gray code.

```
**Algorithm** *GeneratePN(n)*
(* generates all prefix normal words of length $n$ *)
1.  for $d = 0, 1, \dots, n$
2.      do initialize $F$ of length $n$ with all 0s
3.          GeneratePN($d, n-d, \varepsilon$)
```

Fig. 4. Algorithm generating all prefix normal words of length n.

Theorem 1. *Algorithm* GeneratePN(n) *generates all prefix normal words of length n in amortized $O(f(n))$ time per word, where $f(n)$ is the average critical prefix length of prefix normal words of length n. In particular, $f(n) = O(n)$.*

Proof. Since $1^d 0^{n-d}$ is prefix normal for every d, we only need to show that the correct subtrees of T_d^n are generated by the algorithm. By Lemma 2, only those children will be generated that are prefix normal; on the other hand, by the bubble property (Prop. 1), as soon as a child tests negative, no further children will be prefix normal. The running time of the recursive call on $w \in \mathcal{L}_{\mathrm{PN}}$ consists of (a) updating and restoring F (lines 2 and 9): the number of steps equals the critical prefix length $cr(w)$ of w; (b) computing z (line 3): again $cr(w)$ many steps; and (c) work within the while-loop (lines 5 to 8), which, for a word with k prefix normal children, consists of k positive and 1 negative membership tests, of k updates of z, and the recursive calls on the positive children. The membership tests take constant time by Corollary 1, so does the update of z. Since w has k prefix normal children, we charge the positive membership tests and the z-updates to the children, and the negative test to the current word. So for one word $w \in \mathcal{L}_{\mathrm{PN}}$, we get $3 \cdot cr(w) + O(1) + 2 \cdot O(1) = O(cr(w))$ work. □

Next we present experimental evidence for the following conjecture:

Conjecture 1. $f(n) = \Theta(\log n)$.

4 Experimental results

In this section we present some theoretical and numerical results about the number of prefix normal words and their structure. These have become available thanks to the algorithm presented, which allowed us to generate $\mathcal{L}_{\mathrm{PN}}$ up to length 50 on a home computer. Let $pnw(n) := |\mathcal{L}_{\mathrm{PN}} \cap \Sigma^n|$. The following lemma follows from the observation that $1^{\lceil n/2 \rceil} w$ is a prefix normal word of length n for all words w of length $\lfloor n/2 \rfloor$.

Lemma 4. *The number of prefix normal words grows exponentially in n. We have that $pnw(n) \geq 2^{\lfloor n/2 \rfloor}$.*

The first members of the sequence $pnw(n)$ are listed in [18], and these values suggest that the lower bound above is not sharp. We turn our attention to the growth rate of $pnw(n)$ as n increases. Note that $1 \leq pnw(n)/pnw(n-1) \leq 2$. The lower bound follows from the fact that all prefix normal words can be extended by adding a 0 to the end, and the upper bound is implied by the prefix-closed

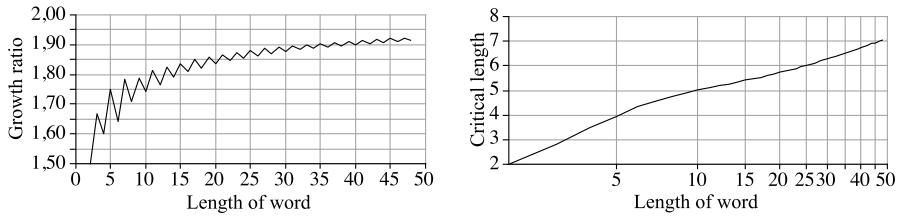

Fig. 5. The value of $pnw(n)/pnw(n-1)$ (left), and of $f(n) = E(cr(w))$ for prefix normal words w of length n, for $n \leq 50$ (right, loglinear scale).

property of $\mathcal{L}_{\text{PN}}$. Fig. 5 (left) shows the growth ratio for small values of n. The figure shows two interesting phenomena: the values seem to approach 2 slowly, i.e., the number of prefix normal words almost doubles as we increase the length by 1. Second, the values show on oscillation pattern between even and odd values. We have so far been unable to establish these observations theoretically.

The structure of prefix normal words is relevant for the generation algorithm, since the amortized running time of the algorithm is bounded from above by the average value $f(n)$ of the critical prefix length $cr(w)$, taken over all prefix normal words w. Note that this differs from the expected critical prefix length of the prefix normal form *of a uniformly random word*, for which we have the following:

Lemma 5. *Given a random word w of length n, let $w' = \text{PNF}(w)$. Let Z' be the r.v. denoting the critical prefix length of w'. Then for the expected value of Z' we have $E(Z') = \Theta(\log n)$.*

In contrast, the average value $f(n)$ of critical prefix length for prefix normal words is shown in Fig. 5 (right) for $n \leq 50$. The linear alignment of the data points together with Lemma 5 supports the conjecture that also $f(n) = \Theta(\log n)$.

5 Conclusion and Open Problems

Based on the observation that prefix normal words form a bubble language, we presented a Gray code for all prefix normal words of length n in $O(n)$ amortized time per word. Moreover, we conjecture that our algorithm runs in time $\Theta(\log(n))$ per word. The number of words that are *not* prefix normal grows exponentially and greatly dominates prefix normal words (e.g., $pnw(30)/2^{30} < 0.05$), so the gain of any algorithm whose running time is proportional to the output size, over brute-force testing of all binary words, is considerable.

We gave a linear time testing algorithm for words which are derived from a word w already known to be prefix normal. We pose as an open problem to find a strongly subquadratic time testing algorithm *for arbitrary words*. Another open problem is the fast computation of prefix normal forms, which would lead immediately to an improvement for indexed binary jumbled pattern matching.

Acknowledgements: We thank two anonymous referees whose suggestions significantly improved the presentation of the paper.

References

1. G. Badkobeh, G. Fici, S. Kroon, and Zs. Lipták. Binary jumbled string matching for highly run-length compressible texts. *Inf. Process. Lett.*, 113(17):604–608, 2013.
2. P. Burcsi, F. Cicalese, G. Fici, and Zs. Lipták. On Table Arrangements, Scrabble Freaks, and Jumbled Pattern Matching. In *Proc. of the 5th Intern. Conference on Fun with Algorithms (FUN 2010)*, volume 6099 of *LNCS*, pages 89–101, 2010.
3. P. Burcsi, F. Cicalese, G. Fici, and Zs. Lipták. On approximate jumbled pattern matching in strings. *Theory Comput. Syst.*, 50(1):35–51, 2012.
4. A. Butman, R. Eres, and G. M. Landau. Scaled and permuted string matching. *Inf. Process. Lett.*, 92(6):293–297, 2004.
5. F. Cicalese, G. Fici, and Zs. Lipták. Searching for jumbled patterns in strings. In *Proc. of the Prague Stringology Conference 2009 (PSC 2009)*, pages 105–117. Czech Technical University in Prague, 2009.
6. F. Cicalese, T. Gagie, E. Giaquinta, E. S. Laber, Zs. Lipták, R. Rizzi, and A. I. Tomescu. Indexes for jumbled pattern matching in strings, trees and graphs. In *Proc. of the 20th String Processing and Information Retrieval Symposium (SPIRE 2013)*, volume 8214 of *LNCS*, pages 56–63, 2013.
7. F. Cicalese, E. S. Laber, O. Weimann, and R. Yuster. Near linear time construction of an approximate index for all maximum consecutive sub-sums of a sequence. In *Proc. 23rd Annual Symposium on Combinatorial Pattern Matching (CPM 2012)*, volume 7354 of *LNCS*, pages 149–158, 2012.
8. G. Fici and Zs. Lipták. On prefix normal words. In *Proc. of the 15th Intern. Conf. on Developments in Language Theory (DLT 2011)*, volume 6795 of *LNCS*, pages 228–238. Springer, 2011.
9. T. Gagie, D. Hermelin, G. M. Landau, and O. Weimann. Binary jumbled pattern matching on trees and tree-like structures. In *Proc. of the 21st Annual European Symposium on Algorithm (ESA 2013)*, pages 517–528, 2013.
10. E. Giaquinta and Sz. Grabowski. New algorithms for binary jumbled pattern matching. *Inf. Process. Lett.*, 113(14-16):538–542, 2013.
11. T. Kociumaka, J. Radoszewski, and W. Rytter. Efficient indexes for jumbled pattern matching with constant-sized alphabet. In *Proc. of the 21st Annual European Symposium on Algorithm (ESA 2013)*, pages 625–636, 2013.
12. L.-K. Lee, M. Lewenstein, and Q. Zhang. Parikh matching in the streaming model. In *Proc. of 19th International Symposium on String Processing and Information Retrieval, SPIRE 2012*, volume 7608 of *Lecture Notes in Computer Science*, pages 336–341. Springer, 2012.
13. T. M. Moosa and M. S. Rahman. Indexing permutations for binary strings. *Inf. Process. Lett.*, 110:795–798, 2010.
14. T. M. Moosa and M. S. Rahman. Sub-quadratic time and linear space data structures for permutation matching in binary strings. *J. Discrete Alg.*, 10:5–9, 2012.
15. F. Ruskey, J. Sawada, and A. Williams. Binary bubble languages and cool-lex order. *J. Comb. Theory, Ser. A*, 119(1):155–169, 2012.
16. F. Ruskey, J. Sawada, and A. Williams. De Bruijn sequences for fixed-weight binary strings. *SIAM Journal of Discrete Mathematics*, 26(2):605–517, 2012.
17. J. Sawada and A. Williams. Efficient oracles for generating binary bubble languages. *Electr. J. Comb.*, 19(1):P42, 2012.
18. N. J. A. Sloane. The On-Line Encyclopedia of Integer Sequences. Available electronically at `http://oeis.org`. Sequence A194850.
19. A. M. Williams. *Shift Gray Codes.* PhD thesis, Univ. of Victoria, Canada, 2009.