Theoretical Computer Science 410 (2009) 5365–5373

Contents lists available at ScienceDirect

Theoretical Computer Science

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

Searching for gapped palindromes

Roman Kolpakov [a,*], Gregory Kucherov [b,c]

[a] *Moscow University, Russia*
[b] *LIFL and INRIA Lille - Nord Europe, Lille, France*
[c] *J.-V.Poncelet Lab., Moscow, Russia*

A B S T R A C T

We study the problem of finding, in a given word, all *maximal* gapped palindromes verifying two types of constraints, that we call *long-armed* and *length-constrained* palindromes. For each of the two classes, we propose an algorithm that runs in time $O(n + S)$ for a constant-size alphabet, where S is the number of output palindromes. Both algorithms can be extended to compute biological gapped palindromes within the same time bound.

1. Introduction

A palindrome is a word that reads the same backward and forward. Palindromes have long drawn attention of computer science researchers. In word combinatorics, for example, studies have been made on palindromes occurring in Fibonacci words [1], or in general Sturmian words [2,3]. More generally, a so-called *palindrome complexity* of words has been studied [4].

From an algorithmic perspective, identifying palindromic structures turned out to be an important test case for different algorithmic problems. For example, a number of works have been done on recognition of palindromic words on different types of Turing machines [5–8]. Palindrome computation has also been an important problem for parallel models of computation [9,10], as well as for distributed models such as systolic arrays [11,12].

Interestingly, a problem related to palindrome recognition was also considered in the seminal Knuth–Morris–Pratt paper presenting the well-known string matching algorithm [13]. The relation between classical string matching and palindrome detection is not purely coincidental. Both the detection of a pattern occurrence and the detection of an even prefix palindrome (even palindrome occurring at the beginning of the input string) can be solved on the 2-way deterministic push-down automaton (2-DPDA), and therefore by Cook's theorem [14], it can be solved by a linear algorithm on the usual RAM model.

Manacher [15] proposed a beautiful linear-time algorithm that computes the shortest prefix palindrome in the on-line fashion, i.e. in time proportional to its length. Actually, the algorithm is able to compute much more, namely to compute for each position of the word, the length of the longest palindrome centered at this position. This gives the exhaustive representation of all palindromes present in the word.

Words with palindromic structure are important in DNA and RNA sequences, as they reflect the capacity of molecules to fold, i.e. to form double-stranded *stems*, which insures a stable state of those molecules with low free energy. However, in those applications, the reversal of palindromes should be combined with the *complementarity* relation on nucleotides, where c is complementary to g and a is complementary to t (or to u, in the case of RNA). Moreover, biologically meaningful

* Corresponding author.
E-mail addresses: foroman@mail.ru (R. Kolpakov), Gregory.Kucherov@lifl.fr (G. Kucherov).

0304-3975/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2009.09.013

palindromes are *gapped*, i.e. contain a *spacer* between left and right copies. Those palindromes correspond, in particular, to *hairpin* structures of RNA molecules, but are also significant in DNA (see e.g. [16,17]). A linear-time algorithm for computing palindromes with *fixed* spacer length is presented in [18]. A method for computing *approximate* biological palindromes has been proposed e.g. in [19].

Results. In this paper, we are concerned with gapped palindromes, i.e. subwords of the form vuv^{T} for some u, v, where v^{T} is v spelled in the reverse order. Occurrences of v and v^{T} are called respectively *left* and *right* arm of the palindrome. We propose algorithms for computing two natural classes of gapped palindromes. The first class, that we call *long-armed palindromes*, verifies the condition $|u| \leq |v|$, i.e. requires that the length of the palindrome arm is no less than the length of the spacer. The second class is called *length-constrained palindromes* and is specified by lower and upper length bounds on the spacer length $MinGap \leq |u| \leq MaxGap$, and a lower bound on the arm length $MinArm \leq |v|$, where *MinGap*, *MaxGap*, *MinArm* are constants. Moreover, for both definitions, palindromes are additionally required to be *maximal*, i.e. their arms cannot be extended outward or inward preserving the palindromic structure. For both classes, our algorithms run in worst-case time $O(n + S)$, where n is the length of the input word and S is the number of output palindromes, for an alphabet of constant size. (For length-constrained palindromes, our algorithm is actually independent of the alphabet size.) We note that because of the variable spacer length, the above-mentioned algorithm from [18] cannot be efficiently applied to our problems. Both algorithms can be modified to find biological long-armed and length-constrained palindromes within the same running time. We also extend our algorithm for long-armed palindromes to *generalized long-armed palindromes* vuv^{T} verifying $|u| \leq c|v|$ for a constant $c \geq 1$. In this case, our algorithm runs in time $O(c^2 n + S)$ for a constant-size alphabet.

2. Basic definitions

Let w^{T} denote the reversal of w. An *even palindrome* is a word of the form vv^{T}, where v is some nonempty word. An *odd palindrome* is a word vav^{T}, where v is a nonempty word, and a a letter of the alphabet. A *gapped palindrome* is a word of the form vuv^{T} for some nonempty words u, v such that $|u| \geq 2$. Occurrences of v and v^{T} are called respectively *left arm* and *right arm* of the palindrome.

In this paper, we will be interested in two classes of palindromes. A gapped palindrome vuv^{T} is *long-armed* if $|u| \leq |v|$. For pre-defined constants *MinGap*, *MaxGap* (*MinGap* $\leq$ *MaxGap*) and *MinArm*, a gapped palindrome vuv^{T} is called *length-constrained* if it verifies $MinGap \leq |u| \leq MaxGap$ and $MinArm \leq |v|$.

Consider a word $w = w[1] \dots w[n]$ that contains some gapped palindrome vuv^{T}. Assume $v = w[\ell'..\ell'']$, and $v^{\mathrm{T}} = w[r'..r'']$. We use notation $w[\ell'..\ell'', r'..r'']$ for this palindrome. This palindrome is called *maximal* if its arms cannot be extended inward or outward. This means that *(i)* $w[\ell''+1] \neq w[r'-1]$, and *(ii)* $w[\ell'-1] \neq w[r''+1]$ provided that $\ell' > 1$ and $r'' < n$.

3. Long-armed palindromes

Let $w = w[1] \dots w[n]$ be an input word. For technical reasons, we require that the last letter $w[n]$ does not occur elsewhere in the word. In this section, we describe a linear-time algorithm for computing all gapped palindromes occurring in w which are both maximal and long-armed.

The algorithm is based on techniques used for computing different types of periodicities in words [20,21], namely on (an extension of) the Lempel–Ziv factorization of the input word and on longest extension functions. The variant of longest extension functions used here is defined as follows. Assume we are given two words $u[1..n]$ and $v[1..m]$ and we want to compute, for each position $j \in [1..n]$ in u, the length $LP(j)$ of the longest common prefix of $u[j..n]$ and v. Assume $m \leq n$ (otherwise we truncate v to $v[1..n]$). Then this computation can be done in time $O(n)$ (see [20]). If we have to compute $LP(j)$ for a subset of positions $j \in [1..N]$ for some $N \leq n$, then the time bound becomes $O(N + m)$. Similar bounds apply if we want to compute the lengths of longest common suffixes of $u[1..j]$ and v.

We now describe the algorithm. First, we compute the *reversed Lempel–Ziv factorization* of $w = f_1 f_2 \dots f_m$ defined recursively as follows:

- if a letter a immediately following $f_1 f_2 \dots f_{i-1}$ does not occur in $f_1 f_2 \dots f_{i-1}$ then $f_i = a$,
- otherwise, f_i is the longest subword of w following $f_1 f_2 \dots f_{i-1}$ which occurs in $(f_1 f_2 \dots f_{i-1})^{\mathrm{T}}$.

This factorization can be computed in time $O(n \log |A|)$, where A is the alphabet of w, by building the suffix tree for w^{T} with Weiner's algorithm that processes the suffixes from shortest to longest (i.e. processes the input word from right to left) [22]. For $i = 1, 2 \dots m$, we construct the suffix tree T_i of the word $(f_1 f_2 \dots f_i)^{\mathrm{T}}$, and compute f_{i+1} as the longest word that occurs immediately after $f_1 f_2 \dots f_i$ in w and is present in T_i. If no such word exists, f_{i+1} is defined to be the letter immediately following $f_1 f_2 \dots f_i$ in w. For each $i = 1, 2, \dots, m$, denote $f_i = w[s_i..t_i]$ $(s_i = t_{i-1} + 1)$ and $F_i = |f_i| = t_i - s_i + 1$.

After computing the reversed Lempel–Ziv factorization, we split all maximal long-armed palindromes into two categories that we compute separately: those which cross (or touch) a border between two factors and those which occur entirely within one factor. Formally, for each $i = 1, 2 \dots m$, we define the set $P(i)$ of all maximal long-armed palindromes

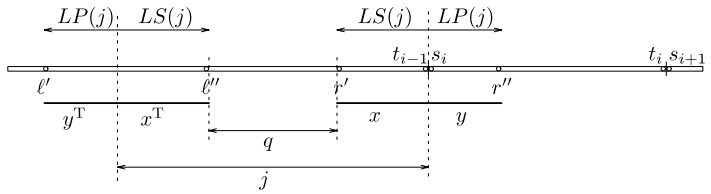

Fig. 1. Computing palindromes of $P'(i)$.

$w[\ell'..\ell'', r'..r'']$ that fall into one of the following two cases:

$$\ell' > s_{i-1} \quad \text{and} \quad r'' = t_{i-1}, \text{ or} \tag{1}$$

$$\ell' \le s_i \quad \text{and} \quad t_{i-1} < r'' \le t_i. \tag{2}$$

In words, $P(i)$ contains palindromes that either end at the border between f_{i-1} and f_i and start inside f_{i-1} (condition (1)), or end inside f_i and cross the border between f_{i-1} and f_i (condition (2)).

Complementarily, define $Q(i)$ to be the set of all maximal long-armed palindromes $w[\ell'..\ell'', r'..r'']$ that verify

$$\ell' > s_i \quad \text{and} \quad r'' < t_i, \tag{3}$$

i.e. occur completely inside f_i.

Observe that all sets $P(i)$ and $Q(i)$ are pairwise disjoint and the set $\cup_{i=1}^{m} P(i) \cup \cup_{i=1}^{m} Q(i)$ contains all maximal long-armed palindromes in w.

3.1. Computing $P(i)$

Each set $P(i)$ is further split into three disjoint sets $P'(i) \cup P''(i) \cup P'''(i)$. $P'(i) \subseteq P(i)$ is the set of all palindromes $w[\ell'..\ell'', r'..r'']$ which satisfy one of the conditions:

$$\ell' > s_{i-1} \quad \text{and} \quad r'' = t_{i-1}, \quad \text{or} \tag{4}$$

$$r' \le s_i \quad \text{and} \quad t_{i-1} < r'' \le t_i. \tag{5}$$

$P'(i)$ are maximal long-armed palindromes of $P(i)$ that either satisfy (1) (condition (4)), or have their right arm crossing (or touching from the right) the border between f_{i-1} and f_i (condition (5)).

$P''(i) \subseteq P(i)$ contains all palindromes $w[\ell'..\ell'', r'..r'']$ which verify

$$\ell' \le s_i \quad \text{and} \quad \ell'' \ge t_{i-1}. \tag{6}$$

Palindromes of $P''(i)$ have their left arm crossing (or touching) the border between f_{i-1} and f_i.

Finally, $P'''(i) \subseteq P(i)$ contains all palindromes $w[\ell'..\ell'', r'..r'']$ which satisfy the conditions

$$\ell'' < t_{i-1} \quad \text{and} \quad r' > s_i. \tag{7}$$

Palindromes of $P'''(i)$ are those for which the border between f_{i-1} and f_i falls inside the spacer.

Computing $P'(i)$.

Let $w[\ell'..\ell'', r'..r'']$ be a palindrome from $P'(i)$, and let $q = r' - \ell'' - 1$ be the spacer length. Then the right arm $w[r'..r'']$ is a concatenation of a possibly empty prefix $x = w[r'..t_{i-1}]$ and a possibly empty suffix $y = w[s_i..r']$. Then the left arm $w[\ell'..\ell'']$ is a concatenation of the prefix $y^{\mathrm{T}} = w[\ell'..t_{i-1} - j]$ and suffix $x^{\mathrm{T}} = w[s_i - j..\ell'']$ where $j = 2|x| + q$ (see Fig. 1). Moreover, since the palindrome is maximal, y has to be the longest common prefix of words $w[s_i..n]$ and $w[1..t_{i-1} - j]^{\mathrm{T}}$, and x has to be the longest common suffix of words $w[1..t_{i-1}]$ and $w[s_i - j..n]^{\mathrm{T}}$. Since the spacer length q is no more than the arm length $|x| + |y|$, we have $q \le |x| + |y|$, i.e. $j \le 3|x| + |y|$.

Lemma 1. *For any palindrome of $P'(i)$, we have $|x| < F_{i-1}$, where F_i is the length of the ith factor in the reversed Lempel–Ziv factorization.*

Proof. If $|y| = 0$, i.e. $r'' = t_{i-1}$ (condition (4)), then $|x| < F_{i-1}$ immediately follows from $\ell' > s_{i-1}$ (condition (4)). If $|y| > 0$, i.e. $r'' > t_{i-1}$ (condition (5)), then from $|x| \ge F_{i-1}$ we obtain that the prefix $w[s_{i-1}..r'']$ of $w[s_{i-1}..n]$ occurs in $(f_1 f_2 \dots f_{i-2})^{\mathrm{T}}$ as a subword of the left arm of the palindrome, which contradicts the definition of $f_{i-1} = w[s_{i-1}..r'']$ as the longest prefix of $w[s_{i-1}..n]$ that occurs in $(f_1 f_2 \dots f_{i-2})^{\mathrm{T}}$. (If f_{i-1} is a single letter that does not occur to the left, then we obviously have $|x| = 0$.) □

By the definition of long-armed palindromes, we have $j \le 3|x| + |y| < 3F_{i-1} + F_i$. For all $j < 3F_{i-1} + F_i$, we compute the longest common prefix $LP(j)$ of words $w[s_i..s_{i+1}]$ and $w[1..t_{i-1} - j]^{\mathrm{T}}$ and the longest common suffix $LS(j)$ of words $w[s_{i-1}..t_{i-1}]$ and $w[s_i - j..n]^{\mathrm{T}}$ (see Fig. 1). These computations can be done in time $O(F_{i-1} + F_i)$ using the technique of longest extension functions mentioned in the beginning of Section 3.

Then each palindrome of $P'(i)$ corresponds to a value of j which satisfies the following conditions:

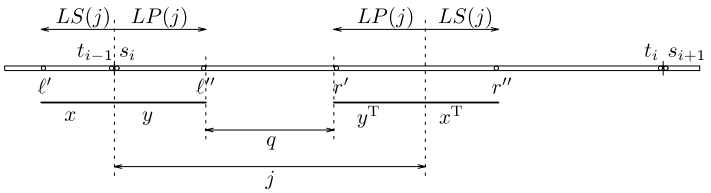

Fig. 2. Computing palindromes of $P''(i)$.

(i) $LP(j) + 3LS(j) \geq j$,
(ii) if $LP(j) = 0$ then $j < F_{i-1}$,
(iii) $LS(j) < j/2$.

Condition (i) is the main condition that ensures the palindrome to be long-armed. Condition (ii) excludes palindromes that end at the border between f_{i-1} and f_i and start before (or at) the border between f_{i-2} and f_{i-1}, as those palindromes do not belong to $P'(i)$ (cf condition (4)). Finally, condition (iii) ensures that the palindrome has a positive gap. Observe that we also need to ensure the condition $LP(j) \leq F_i$ according to the definition of $P'(i)$ ($r'' \leq t_i$ in condition (5)). However, this condition always holds as $LP(j) > F_i$ would contradict the definition of factorization, by an argument similar to the proof of Lemma 1.

Conversely, if j satisfies conditions (i)–(iii) above, then there exists a palindrome $w[\ell'..\ell'', r'..r'']$ for $\ell' = s_i - j - LP(j)$, $\ell'' = t_{i-1} - j + LS(j)$, $r' = s_i - LS(j)$, and $r'' = t_{i-1} + LP(j)$. Once conditions (i)–(iii) are verified for some j, the corresponding palindrome is output by the algorithm. The whole computation takes time $O(F_{i-1} + F_i)$.

Computing $P''(i)$.

We now focus on set $P''(i)$ which consists of those palindromes of $P(i)$ which have their left arm crossing (or touching) the border between f_{i-1} and f_i. Let $w[\ell'..\ell'', r'..r'']$ be a maximal long-armed palindrome from $P''(i)$, and $q = r' - \ell'' - 1$ be the spacer length. Then the left arm $w[\ell'..\ell'']$ is a concatenation of a possibly empty prefix $x = w[\ell'..t_{i-1}]$ and a possibly empty suffix $y = w[s_i..\ell'']$. Then the right arm $w[r'..r'']$ is a concatenation of prefix $y^{\mathrm{T}} = w[r'..t_{i-1}+j]$ and suffix $x^{\mathrm{T}} = w[s_i+j..r'']$, where $j = 2|y| + q$. Moreover, y has to be the longest common prefix of words $w[s_i..n]$ and $w[1..t_{i-1} + j]^{\mathrm{T}}$, and x has to be the longest common suffix of words $w[1..t_{i-1}]$ and $w[s_i + j..n]^{\mathrm{T}}$ (see Fig. 2). Since the spacer length q has to be no more than the arm length $|x| + |y|$, we have $q \leq |x| + |y|$, i.e. $j \leq |x| + 3|y|$.

Similarly to the case of $P'(i)$, we compute, for each $j = 1, 2, \ldots, F_i$, the longest common prefix $LP(j)$ of words $w[s_i..t_i]$ and $w[s_i..t_{i-1} + j]^{\mathrm{T}}$ and the longest common suffix $LS(j)$ of words $w[1..t_{i-1}]$ and $w[s_i + j..s_{i+1}]^{\mathrm{T}}$. Tables LP and LS are computed in time $O(F_i)$.

Each palindrome of $P''(i)$ corresponds to a value of j verifying the following conditions:

(i) $3LP(j) + LS(j) \geq j$,
(ii) $j + LS(j) \leq F_i$,
(iii) $LP(j) < j/2$.

Similar to the case of $P'(i)$, condition (i) ensures the palindrome to be long-armed, condition (ii) keeps only those that end inside f_i (condition (6)), and condition (iii) ensures that the gap size is positive.

If for some $j \leq F_i$, conditions (i)–(iii) above are satisfied, then the algorithm outputs the palindrome $w[\ell'..\ell'', r'..r'']$ where $\ell' = s_i - LS(j)$, $\ell'' = t_{i-1} + LP(j)$, $r' = s_i + j - LP(j)$, and $r'' = t_{i-1} + j + LS(j)$. The computation of $P''(i)$ is done in time $O(F_i)$.

Computing $P'''(i)$.

To compute $P'''(i)$, we partition it into disjoint subsets $P'''_k(i)$ for $k = 1, 2, \ldots, \lfloor \log_2 F_i \rfloor$, where $P'''_k(i)$ is the set of all palindromes $w[\ell'..\ell'', r'..r'']$ from $P'''(i)$ such that

$$s_i + \left\lfloor \frac{F_i}{2^k} \right\rfloor \leq r'' < s_i + \left\lfloor \frac{F_i}{2^{k-1}} \right\rfloor. \tag{8}$$

Lemma 2. *For any palindrome $w[\ell'..\ell'', r'..r''] \in P'''_k(i)$, we have $r' \leq s_i + \lfloor \frac{F_i}{2^k} \rfloor$.*

Proof. If $r' > s_i + \lfloor \frac{F_i}{2^k} \rfloor$, the arm length of the palindrome is no more than $\lfloor \frac{F_i}{2^k} \rfloor$, and because of the long-armed condition the spacer length should then be no more than $\lfloor \frac{F_i}{2^k} \rfloor$. Then, $\ell'' \geq r' - 1 - \lfloor \frac{F_i}{2^k} \rfloor \geq s_i$ which contradicts condition (7) defining $P'''(i)$. □

By the lemma, the right arm of the palindrome is a concatenation of a possibly empty prefix $x = w[r'..t_{i-1} + \lfloor \frac{F_i}{2^k} \rfloor]$ and a nonempty suffix $y = w[s_i + \lfloor \frac{F_i}{2^k} \rfloor..r'']$. Similar to the previous cases, x has to be the longest common suffix of the words $w[s_i..t_{i-1} + \lfloor \frac{F_i}{2^k} \rfloor]$ and $w[s_i + \lfloor \frac{F_i}{2^k} \rfloor - j..t_{i-1}]^{\mathrm{T}}$, and y has to be the longest common prefix of the words $w[s_i + \lfloor \frac{F_i}{2^k} \rfloor..n]$ and $w[1..t_{i-1} + \lfloor \frac{F_i}{2^k} \rfloor - j]^{\mathrm{T}}$, where $j = 2|x| + q$ and q is the spacer length of the palindrome (see Fig. 3).

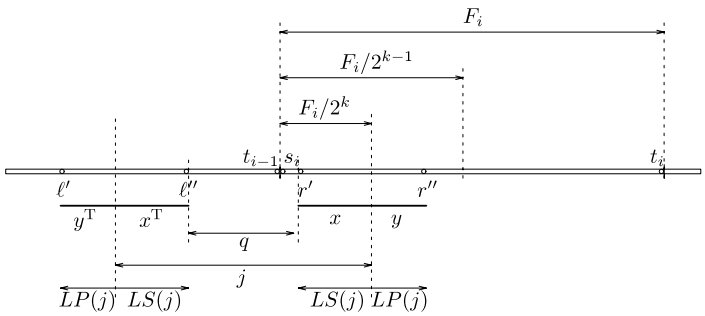

Fig. 3. Computing palindromes of $P'''(i)$.

Moreover, x and y satisfy the relations $|x| < \lfloor \frac{F_i}{2^k} \rfloor$ and $0 < |y| \le \lfloor \frac{F_i}{2^{k-1}} \rfloor - \lfloor \frac{F_i}{2^k} \rfloor$. Thus, $q \le |x| + |y| < \lfloor \frac{F_i}{2^{k-1}} \rfloor$, and then $j = 2|x| + q < 2\lfloor \frac{F_i}{2^k} \rfloor + \lfloor \frac{F_i}{2^{k-1}} \rfloor \le 2\lfloor \frac{F_i}{2^{k-1}} \rfloor$. On the other hand, from the condition $l'' < t_{i-1}$ we also have $|x| < j - \lfloor \frac{F_i}{2^k} \rfloor$ which implies $j > \lfloor \frac{F_i}{2^k} \rfloor$.

Now, to compute all palindromes from $P'''_k(i)$ we apply again the same procedure: for all j such that $\lfloor \frac{F_i}{2^k} \rfloor < j < 2\lfloor \frac{F_i}{2^{k-1}} \rfloor$, we compute the longest common suffix $LS(j)$ of words $w[s_i..t_{i-1} + \lfloor \frac{F_i}{2^k} \rfloor]$ and $w[s_i + \lfloor \frac{F_i}{2^k} \rfloor - j..n]^{\mathrm{T}}$, and the longest common prefix $LP(j)$ of words $w[s_i + \lfloor \frac{F_i}{2^k} \rfloor..s_i + \lfloor \frac{F_i}{2^{k-1}} \rfloor]$ and $w[1..t_{i-1} + \lfloor \frac{F_i}{2^k} \rfloor - j]^{\mathrm{T}}$ (Fig. 3). Each palindrome of $P'''_k(i)$ corresponds then to a value j verifying the following conditions:

(i) $LP(j) + 3LS(j) \ge j$,
(ii) $0 < LP(j) \le \lfloor \frac{F_i}{2^{k-1}} \rfloor - \lfloor \frac{F_i}{2^k} \rfloor$,
(iii) $LS(j) < \min(\lfloor \frac{F_i}{2^k} \rfloor, j - \lfloor \frac{F_i}{2^k} \rfloor)$.

Here condition (i) ensures the palindrome to be long-armed, condition (ii) ensures condition (8), and condition (iii) ensures condition (7).

If some j satisfies the above conditions (i)–(iii), we output the palindrome $w[\ell'..\ell'', r'..r'']$, where $\ell' = s_i + \lfloor \frac{F_i}{2^k} \rfloor - j - LP(j)$, $\ell'' = t_{i-1} + \lfloor \frac{F_i}{2^k} \rfloor - j + LS(j)$, $r' = s_i + \lfloor \frac{F_i}{2^k} \rfloor - LS(j)$, and $r'' = t_{i-1} + \lfloor \frac{F_i}{2^k} \rfloor + LP(j)$.

The required functions $LP(j)$ and $LS(j)$ can be computed in time $O(\frac{F_i}{2^k})$, and then $P'''_k(i)$ can be computed in time $O(\frac{F_i}{2^k})$. Summing up over $k = 1, 2, \ldots, \lfloor \log_2 F_i \rfloor$, $P'''(i)$ can be computed in time $O(F_i)$.

3.2. Computing $Q(i)$

Recall that $Q(i)$ contains all palindromes $w[\ell'..\ell'', r'..r'']$ which verify (3), i.e. occur as a proper subword of factor f_i. Since f_i has a reversed copy in $f_1 f_2 \ldots f_{i-1}$, a reverse of each palindrome of $Q(i)$ also occurs in that copy. Therefore, it can be "copied over" from that location. Technically, this is done exactly in the same way as in the algorithm for computing maximal repetitions presented in [23] (see also [20]). Recovering each palindrome of $Q(i)$ is done in constant time. We refer the reader to those papers for details of this procedure.

3.3. Putting all together

As explained in the beginning of Section 3, the reversed Lempel-Ziv factorization can be computed in time $O(n)$ for a constant-size alphabet. From the analysis of Section 3.1, sets $P'(i)$, $P''(i)$, $P'''(i)$ are computed in time $O(F_{i-1} + F_i)$, $O(F_i)$ and $O(F_{i-1} + F_i)$ respectively. Therefore, $P(i)$ is computed in time $O(F_{i-1} + F_i)$. Summing over all i, all involved palindromes are computed in time $O(n)$. The overall time spent to compute all $Q(i)$ is $O(n+T)$, where T is the number of output palindromes. Since all sets $P(i)$, $Q(i)$ are pairwise disjoint, we obtain the final result:

Theorem 3. *All maximal long-armed palindromes can be computed in time $O(n+S)$, where n is the length of the input word and S the number of output palindromes.*

Note that the whole algorithm is independent of the alphabet size except for the $O(n \log |A|)$ computation of the reversed Lempel–Ziv factorization described in the beginning of Section 3. Therefore, Theorem 3 is stated under the assumption of constant-size alphabet, otherwise the time bounds becomes $O(n \log |A| + S)$. However, when this paper was prepared for publication, we got known about the forthcoming paper [24] showing how to compute this factorization in an alphabet-independent linear time. This makes the bound of Theorem 3 independent of the alphabet size.

3.4. Generalized long-armed palindromes

The above algorithm can be extended to long-armed palindromes under a more general definition. Consider gapped palindromes vuv^{T} verifying condition $|u| \leq c|v|$ for some constant $c \geq 1$. We now show that using the above algorithm, such palindromes can be computed in time $O(c^2n + S)$, where S is the output size.

Consider the computation of $P'(i)$, $P''(i)$ and $P'''(i)$ by the above algorithm, applied to the extended palindrome definition.

The case of $P'(i)$ is generalized straightforwardly: the only change is in the range of j (see Fig. 1) that is now $j < (c+2)F_{i-1} + cF_i$. This implies that computing $P'(i)$ is done in time $O(c(F_{i-1} + F_i))$.

Computing $P''(i)$ does not require any modification at all and is still done in time $O(F_i)$.

Computing $P'''(i)$ requires most important modifications. Here, instead of performing binary division of f_i, we now divide it by τ, where $\tau = \frac{c+1}{c}$. We then have disjoint subsets $P'''_k(i), k = 1, 2, \ldots, \lfloor \log_\tau F_i \rfloor$, of palindromes $w[\ell'..\ell'', r'..r'']$ verifying $s_i + \lfloor \frac{F_i}{\tau^k} \rfloor \leq r'' < s_i + \lfloor \frac{F_i}{\tau^{k-1}} \rfloor$. It is straightforward to check that the generalization of Lemma 2 holds: any palindrome of $P'''_k(i)$ is such that $r' \leq s_i + \lfloor \frac{F_i}{\tau^k} \rfloor$. We then have (see Fig. 3) $q < c\lfloor \frac{F_i}{\tau^k} \rfloor$ and then $j < 2\lfloor \frac{F_i}{\tau^k} \rfloor + c\lfloor \frac{F_i}{\tau^{k-1}} \rfloor \leq (2 + c\tau)\frac{F_i}{\tau^k} \leq 2(1+c)\frac{F_i}{\tau^k}$ (as $\tau \leq 2$). Therefore, all palindromes of $P'''_k(i)$ can be computed in time $O(\frac{cF_i}{\tau^k})$. Summing up over k, all palindromes from $P'''(i)$ can be computed in time $O(\frac{cF_i}{\tau - 1}) = O(c^2F_i)$.

We conclude that all palindromes of $P(i)$ can be computed in time $O(cF_{i-1} + c^2F_i)$, and the total time for computing of all sets $P(i)$ is $O(c^2n)$. The computation of sets $Q(i)$ is not changed. We then obtain

Theorem 4. *All maximal palindromes vuv^{T} such that $|u| \leq c|v|$ for some constant $c \geq 1$ can be computed in time $O(c^2n + S)$, where n is the length of the input word and S the number of output palindromes.*

4. Length-constrained palindromes

Recall that a gapped palindrome vuv^{T} is called *length-constrained* if $MinGap \leq |u| \leq MaxGap$ and $MinArm \leq |v|$ for some pre-defined constants *MinGap*, *MaxGap* and *MinArm*. In this section, we are interested to compute, in a given word, all palindromes that are both length-constrained and maximal.

Note that we do not want to output palindromes that verify length constraints but are not maximal. The inward/outward extension of such a palindrome may lead to a palindrome that no longer verifies length constraints. For example, if $MinArm = 3$, $MinGap = 3$ and $MaxGap = 5$, then the palindrome $\ldots a\boxed{gtt}aaca\boxed{ttg}g\ldots$ verifies length constraints but is not maximal, while its extension $\ldots a\boxed{gtta}ac\boxed{attg}g\ldots$ is maximal but does not verify length constraints.

We now describe an algorithm that computes all length-constrained palindromes. It consists of two main steps that perform respectively a preparatory pre-processing and the main computation.

First step. Consider an input word $w = w[1..n]$. For a position i, we consider words $W(i^+) = w[i..i + MinArm - 1]$ and $W(i^-) = w[i - MinArm..i - 1]^{\mathrm{T}}$, where i^+, i^- are interpreted as start positions in forward and backward direction respectively. Consider the set $\mathcal{P} = \{i^+, i^- | i = 1..n\}$. For two positions $k_1, k_2 \in \mathcal{P}$, define the equivalence relation $k_1 \equiv k_2$ iff $W(k_1) = W(k_2)$. At the first step, we assign to each position i^-, i^+ the identifier (number) of its equivalence class under the above equivalence relation. This assignment can be done in time $O(n)$ using, e.g., the suffix array for the word $w\#w^{\mathrm{T}}\$$. A simple traversal of this suffix array allows the desired assignment: two successive alphabetically-ordered suffixes belong to the same equivalence class iff the length of their common prefix is at least *MinArm*. Deciding whether position i^+ or i^- should be assigned is naturally done depending on whether the suffix starts in w or in w^{T}. Further details are left out. Note that the suffix array can be constructed in time $O(n)$ independent of the alphabet size [25].

Second step. After the first preparatory step, the second step does the main job. Our goal is to find pairs of positions $i < j$ such that (i) $W(i^-) = W(j^+)$ (arm length constraint), (ii) $MinGap \leq j - i \leq MaxGap$ (gap length constraint), and (iii) $w[i] \neq w[j-1]$ (maximality condition). Each such pair of positions corresponds to a desired palindrome. The arm length of this palindrome can then be computed by computing the longest common subword starting at positions i^- and j^+ (i.e. the longest common prefix of $(w[1..i-1])^{\mathrm{T}}$ and $w[j..n]$). This can be done in constant time using lowest common ancestor queries on the suffix tree for $w\#w^{\mathrm{T}}\$$ [18], but can be also done with the suffix array using the results of [25]. The latter solution is independent on the alphabet size.

We are now left with describing how pairs i, j are found. This is done in an on-line fashion during the traversal of w from left to right. For each equivalence class, we maintain the list of all "minus-positions" $(i_1)^-, (i_2)^-, \ldots, (i_k)^-$ ($i_1 < i_2 < \ldots < i_k$) scanned so far and belonging to this equivalence class. Moreover, this list is partitioned into *runs* of consecutive list items $(i_\ell)^-, (i_{\ell+1})^-, \ldots, (i_{\ell+k_\ell})^-$ such that $w[i_\ell] = w[i_{\ell+1}] = \cdots = w[i_{\ell+k_\ell}]$ and $w[i_{\ell-1}] \neq w[i_\ell]$ and $w[i_{\ell+k_\ell}] \neq w[i_{\ell+k_\ell+1}]$ (provided that $w[i_{\ell-1}]$, $w[i_{\ell+k_\ell+1}]$ exist in the list).

Furthermore, we maintain a pointer from each run to the next run, so that we are able to "jump", in a constant time, from any item of the current run to the first item of the next run, avoiding the traversal of the whole run. Note that since new items will be added by the algorithm incrementally, we need to maintain these pointers in a dynamic manner. This, however, can be easily implemented by e.g. setting a pointer to the next run only from the first item of the previous run, and pointing from each item to the first item of its run.

The list items can then be implemented by a structure with the following fields:

position: position i such that i^- belongs to the corresponding equivalence class,
NextItem: pointer to the next item in the list,
NextRun: pointer to the first item of the next run.

Assume now we are processing a position j of w. First, we insert j to the list of the equivalence class of j^- and update links *NextItem* and *NextRun* accordingly. Then we have to find all positions i from the interval $[j - MaxGap..j - MinGap]$ such that i^- belongs to the equivalence class of j^+.

Let C be the identifier of the equivalence class of j^+. We need to check, in the list for C, those positions which belong to the interval $[j - MaxGap..j - MinGap]$. To efficiently access the corresponding fragment in the list, we remember the smallest position of the list belonging to the interval $[\ell - MaxGap..\ell - MinGap]$ for the last processed position $\ell < j$ such that ℓ^+ belongs to equivalence class C. We then start the traversal from this position looking for the positions i falling into the interval $[j - MaxGap..j - MinGap]$. Since during this process each item of the list is visited at most once, this trick allows us to bound the total time for finding the starting position of segments $[j - MaxGap..j - MinGap]$ by the total size of all the lists, i.e. by $O(n)$.

For each retrieved position i, we verify if $w[i] \neq w[j-1]$ (maximality condition). If this inequality does not hold, we jump to the first position of the next run of the list, using the *NextRun* link defined above, thus avoiding consecutive negative tests and insuring that the number of those tests is proportional to the number of output palindromes. The following theorem puts together the two steps of the algorithm.

Theorem 5. *For any pre-defined constants MinArm, MinGap, MaxGap, all length-constrained palindromes can be found in time* $O(n + S)$.

Proof. The first step is done in time $O(n)$ using a suffix array. At the second step, finding starting positions from intervals $[j - MaxGap, j - MinGap]$ in the list for the class of j^+ takes time $O(n)$ overall. Testing the maximality condition and outputting the resulting palindromes takes time $O(S)$, where S is the number of output palindromes. Finally, implementing the constant-time computation of longest common subwords starting at given positions is done in time $O(n)$ independent of the alphabet size using results of [25]. □

Algorithm 1 presents a pseudo-code of the algorithm. Besides variables *position*, *NextItem* and *NextRun* defined previously, the algorithm uses the following variables.

LeftClass(j): equivalence class of j^-,
RightClass(i): equivalence class of i^+,
LastItem(C): pointer to the last item in the list for class C,
LastRun(C): pointer to the first item of the current last run in the list for class C,
PreviousStartItem(C): pointer to the start item in the search interval for the last processed position ℓ^+ of class C, i.e. to the smallest position in the list for C belonging to the interval $[\ell - MaxGap..\ell - MinGap]$. (To avoid irrelevant algorithmic details, we assume that such a position always exists.)

5. Biological palindromes

Both algorithms presented in Sections 3 and 4 can be extended to biological palindromes, where the word reversal is defined in conjunction with the complementarity of nucleotide letters: $c \leftrightarrow g$ and $a \leftrightarrow t$ (or $a \leftrightarrow u$, in case of RNA). For example, $\ldots c\boxed{acat}aca\boxed{atgt}c\ldots$ is a maximal biological gapped palindrome.

The main part of either algorithm is extended in a straightforward way: each time the algorithm compares two letters, this comparison is replaced by testing their complementarity.

Some parts of the algorithms deserve a special attention. For the algorithm of Section 3 for long-armed palindromes, the computation of the reversed Lempel–Ziv factorization extends in a straightforward way too: when computing the next factor f_{i+1}, one has to use the complementarity relation. Similarly, the computation of extension functions *LP* and *LS* are also extended straightforwardly.

The algorithm of Section 4 for length-constrained palindromes requires a straightforward modification of the first step: we now need to compute the suffix array for $w\#w^{\mathrm{T}}\$$, where w^{T} stands for the "biological inversion" (i.e. reversal together with complement). At the second step, the algorithm uses the same suffix array (or alternatively, the suffix tree for $w\#w^{\mathrm{T}}\$$) in order to implement constant-time common subword queries.

Acknowledgments

Part of this work was done during the stay of R. Kolpakov at Inria Lille - Nord Europe in September 2007, supported by INRIA. R. Kolpakov acknowledges the support of the Russian Foundation for Fundamental Research (Grant 08-01-00863) and of the program for supporting Russian scientific schools (Grant NSh–5400.2006.1).

```
for j ← MinArm + 1 to n do
   /* insert position j⁻ to the appropriate list                                   */
   begin
      C ⟵ LeftClass(j);
      add a new item NewItem to the list of class C;
      NewItem.position ⟵ j;
      LastItem(C).NextItem ⟵ NewItem;
      if w[j] ≠ w[LastItem(C).position] then
         set pointer NextRun for the current run to point to NewItem;
         start a new run with NewItem;
      end
      LastItem(C) ⟵ NewItem;
   end
   /* find all maximal length-constrained palindromes with the right arm starting at
      position j                                                                   */
   begin
      C ⟵ RightClass(j);
      /* find, in the list for class C, the first position greater than or equal to
         (j − MaxGap)                                                             */
      SearchItem ⟵ PreviousStartItem(C);
      while SearchItem.position < j − MaxGap do
         SearchItem ⟵ SearchItem.NextItem;
      end
      PreviousStartItem(C) ⟵ SearchItem;
      /* for each position in the list for class C between (j − MaxGap) and (j − MinGap),
         check if there exists a corresponding maximal palindrome                 */
      while SearchItem.position ≤ (j − MinGap) do
         if w[SearchItem.position] ≠ w[j − 1] then
            lp ⟵ length of the longest common prefix of words  w[j + MinArm .. n] and
                 (w[1 .. SearchItem.position − MinArm − 1])^T;
            output the palindrome
            w[SearchItem.position − MinArm − lp .. SearchItem.position − 1, j .. j + MinArm + lp − 1];
            SearchItem ⟵ SearchItem.NextItem;
         end
         else
            SearchItem ⟵ SearchItem.NextRun;
         end
      end
   end
end
```

Algorithm 1: Step 2 of the algorithm for computing length-constrained palindromes.

References

[1] X. Droubay, Palindromes in the Fibonacci word, Information Processing Letters 55 (4) (1995) 217–221.
[2] X. Droubay, G. Pirillo, Palindromes and Sturmian words, Theoretical Computer Science 223 (1999) 73–85.
[3] A. De Luca, A. De Luca, Palindromes in Sturmian words, in: C. de Felice, A. Restivo (Eds.), Developments in Language Theory, 9th International Conference, DLT 2005, Palermo, Italy, July 4–8, 2005, Proceedings, in: Lecture Notes in Computer Science, vol. 3572, Springer, 2005, pp. 199–208.
[4] J.-P. Allouche, M. Baake, J. Cassaigne, D. Damanik, Palindrome complexity, Theoretical Computer Science 292 (1) (2003) 9–31.
[5] A.O. Slisenko, Recognition of palindromes by multihead turing machines, in: V.P. Orverkov, N.A. Sonin (Eds.), Problems in the Constructive Trend in Mathematics VI, in: Proceedings of the Steklov Institute of Mathematics, vol. 129, 1973, pp. 30–202.
[6] Z. Galil, Palindrome recognition in real time by a multitape turing machine, Journal of Computer and System Sciences 16 (2) (1978) 140–157.
[7] A. Slissenko, A simplified proof of real-time recognizability of palindromes on Turing machines, Journal of Soviet Mathematics 15 (1) (1981) 68–77; Zapiski Nauchnykh Seminarov LOMI 68 (1977) 123–139. Russian original.
[8] T. Biedl, J. Buss, E. Demaine, M. Demaine, M. Hajiaghayi, T. Vinar, Palindrome recognition using a multidimensional tape, Theoretical Computer Science 302 (1–3) (2003) 475–480.
[9] A. Apostolico, D. Breslauer, Z. Galil, Parallel detection of all palindromes in a string, in: 11th Annual Symposium on Theoretical Aspects of Computer Science, in: LNCS, vol. 775, Springer, Caen, France, 1994, pp. 497–506.
[10] D. Breslauer, Z. Galil, Finding all periods and initial palindromes of a string in parallel, Algorithmica 14 (1995) 355–366.
[11] S.N. Cole, Real-time computation by n-dimensional iterative arrays of finite-state machines, IEEE Transactions on Computers 18 (1969) 349–365.
[12] J. van de Snepscheut, J. Swenker, On the design of some systolic algorithms, Journal of the ACM 36 (4) (1989) 826–840.
[13] D. Knuth, J. Morris, V. Pratt, Fast pattern matching in strings, SIAM Journal of Computation 6 (1977) 323–350.

[14] S. Cook, Linear time simulation of deterministic two-way pushdown automata, in: Proceedings of the 5th World Computer Congress, (Ljubljana, Yugoslavia, August 23–28, 1971), in: IFIP'71, vol. 1, 1971, pp. 75–80.
[15] G. Manacher, A new linear-time on-line algorithm for finding the smallest initial palindrome of a string, Journal of the ACM 22 (3) (1975) 346–351.
[16] P. Warburton, J. Giordano, F. Cheung, Y. Gelfand, G. Benson, Inverted repeat structure of the human genome: The X-chromosome contains a preponderance of large, highly homologous inverted repeats that contain testes genes, Genome Research 14 (2004) 1861–1869.
[17] L. Lu, H. Jia, P. Dröge, J. Li, The human genome-wide distribution of DNA palindromes, Functional and Integrative Genomics 7 (3) (2007) 221–227.
[18] D. Gusfield, Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology, Cambridge University Press, 1997.
[19] A.H.L. Porto, V.C. Barbosa, Finding approximate palindromes in strings, Pattern Recognition 35 (2002) 2581–2591.
[20] R. Kolpakov, G. Kucherov, Identification of periodic structures in words, in: J. Berstel, D. Perrin (Eds.), Applied Combinatorics on Words, Vol. Encyclopedia of Mathematics and its Applications, in: Lothaire Books, vol. 104, Cambridge University Press, 2005, pp. 430–477. chapter 8.
[21] R. Kolpakov, G. Kucherov, Finding repeats with fixed gap, in: Proceedings of the 7th International Symposium on String Processing and Information Retrieval, (SPIRE), A Coruña, Spain (27–29 Septembre, 2000), IEEE, 2000, pp. 162–168.
[22] M. Crochemore, W. Rytter, Text Algorithms, Oxford University Press, 1994.
[23] R. Kolpakov, G. Kucherov, On maximal repetitions in words, Journal of Discrete Algorithms 1 (1) (2000) 159–186.
[24] M. Crochemore, C. Iliopoulos, M. Kubica, W. Rytter, T. Waleń, Efficient algorithms for two extensions of LPF table: the power of suffix arrays, in: Proc. 36th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM), Lecture Notes in Computer Science, Springer Verlag, 2010 (in press).
[25] J. Kärkkäinen, P. Sanders, S. Burkhardt, Linear work suffix array construction, Journal of the ACM 53 (6) (2006) 918–936.