# Searching for Gapped Palindromes

Roman Kolpakov[1] and Gregory Kucherov[2]

[1] Moscow University, 119899 Moscow, Russia, foroman@mail.ru
[2] LIFL/CNRS/INRIA, Parc scientifique de la Haute Borne, 40, Avenue Halley 59650 Villeneuve d'Ascq, France, Gregory.Kucherov@lifl.fr

**Abstract.** 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 both classes, we propose algorithms that run in time $O(n + S)$, 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 [Dro95], or in general Sturmian words [DP99,DLDL05]. More generally, a so-called *palindrome complexity* of words has been studied [ABCD03].

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 [Sli73,Gal78,Sli81,BBD$^+$03]. Palindrome computation has also been an important problem for parallel models of computation [ABG94,BG95], as well as for distributed models such as systolic arrays [Col69,vdSSer].

Interestingly, a problem related to palindrome recognition was also considered in the seminal Knuth-Morris-Pratt paper presenting the well-known string matching algorithm [KMP77]. 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 [Coo71], it can be solved by a linear algorithm on the usual RAM model.

Manacher [Man75] 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 biggest 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 case of RNA). Moreover, biologically meaningful 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. [WGC$^+$04,LJDL07]). A linear-time algorithm for computing palindromes with *fixed* spacer length is presented in [Gus97]. A method for computing *approximate* biological palindromes has been proposed e.g. in [PB02].

*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 $MinLen \leq |v|$, where $MinGap, MaxGap, MinLen$ 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 on the alphabet size.) We note that because of the variable spacer length, the above-mentioned algorithm from [Gus97] 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.

## 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 word. An odd palindrome is a word $vav^T$, where $v$ is a word, and $a$ a letter of the alphabet. A *gapped palindrome* is a word of the form $vuv^T$ for some 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 $MinLen$, a gapped palindrome $vuv^T$ is called *length-constrained* if it verifies $MinGap \leq |u| \leq MaxGap$ and $MinLen \leq |v|$.

Consider a word $w = w[1] \dots w[n]$ that contains some gapped palindrome $vuv^T$. Assume $v = w[l'..l'']$, and $v^T = w[r'..r'']$. We use notation $w[l' : l'', r' : r'']$ for this palindrome. This palindrome is called *maximal* if its arms cannot be extended inward or outward. This means that *(i)* $w[l''+1] \neq w[r'-1]$, and *(ii)* $w[l'-1] \neq w[r''+1]$ provided that $l' > 1$ 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 [KK05,KK00a], 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 [KK05]). 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 $(f_1 f_2 \dots f_{i-1})^T$.

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 the Weiner's algorithm that processes the suffixes from shortest to longest (i.e. processes the input word from right to left) [CR94]. For $i = 1, 2 \dots m$, we construct the suffix tree $T_i$ of the word $(f_1 f_2 \dots f_i)^T$, 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 $w[l' : l'', r' : r'']$ that verify one of the conditions:

1. $r'' = t_{i-1}$ and $l' > s_{i-1}$, or

2. $t_{i-1} < r'' \leq t_i$ and $l' \leq s_i$.

Complementary, define $Q(i)$ to be the set of all maximal long-armed palindromes $w[l' : l'', r' : r'']$ that verify $l' > s_i$ and $r'' < t_i$.

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

### 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 $w[l' : l'', r' : r'']$ which satisfy one of the conditions:

1. $r'' = t_{i-1}$ and $l' > s_{i-1}$, or
2. $t_{i-1} < r'' \leq t_i$ and $r' \leq s_i$.

$P'(i)$ are maximal long-armed palindromes with the right arm crossing (or touching from the right) the border between $f_{i-1}$ and $f_i$.

$P''(i) \subseteq P(i)$ contains all palindromes $w[l' : l'', r' : r'']$ which verify both $l' \leq s_i$ and $l'' \geq t_{i-1}$. 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 $w[l' : l'', r' : r'']$ which satisfy the conditions $l'' < t_{i-1}$ and $r' > s_i$. 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 $w[l' : l'', r' : r'']$ be a palindrome from $P'(i)$, and let $q = r' - l'' - 1$ be the spacer length. Then the right arm $w[r'..r'']$ is a concatenation of a possibly empty prefix $u = w[r'..t_{i-1}]$ and a possibly empty suffix $v = w[s_i..r'']$. Then the left arm $w[l'..l'']$ is a concatenation of the prefix $v^T = w[l'..t_{i-1} - j]$ and suffix $u^T = w[s_i - j..l'']$ where $j = 2|u| + q$ (see Fig. 1 in the Appendix). Moreover, since the palindrome is maximal, $v$ has to be the longest common prefix of words $w[s_i..n]$ and $w[1..t_{i-1} - j]^T$, and $u$ has to be the longest common suffix of words $w[1..t_{i-1}]$ and $w[s_i - j..n]^T$. Since the spacer length $q$ is no more than the arm length $|u| + |v|$, we have $q \leq |u| + |v|$, i.e. $j \leq 3|u| + |v|$.

**Lemma 1.** $|u| < F_{i-1}$.

*Proof.* If $|v| = 0$, i.e. $r'' = t_{i-1}$, then the lemma follows from the condition $l' > s_{i-1}$. If $|v| > 0$, i.e. $r'' > t_{i-1}$, then from $|u| \geq F_{i-1}$ we obtain that the prefix $w[s_{i-1}..r'']$ of $w[s_{i-1}..n]$ occurs in $(f_1 f_2 \ldots f_{i-2})^T$ 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 $(f_1 f_2 \ldots f_{i-2})^T$. (If $f_{i-1}$ is a single letter that doesn't occur to the left, then we obviously have $|u| = 0$.)

From the condition $r'' \leq t_i$ we also have $|v| \leq F_i$ and then $j \leq 3|u| + |v| < 3F_{i-1} + F_i$. 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 $w[1..t_{i-1}-j]^T$ and the longest common suffix $LS(j)$ of words $w[s_{i-1}..t_{i-1}]$ and $w[s_i-j..n]^T$ (see Fig. 1). These computations can be done in time $O(F_{i-1}+F_i)$. Then each palindrome of $P'(i)$ corresponds to a value of $j$ which satisfies the following conditions:

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

Inversely, if $j$ satisfies the above conditions, then there exists a palindrome $w[l':l'',r':r'']$ for $l'=s_i-j-LP(j)$, $l''=t_{i-1}-j+LS(j)$, $r'=s_i-LS(j)$, and $r''=t_{i-1}+LP(j)$. Once conditions 1-3 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 $P''(i)$.** Let $w[l':l'',r':r'']$ be a maximal long-armed palindrome from $P''(i)$, and $q=r'-l''-1$ be the spacer length. Then the left copy $w[l'..l'']$ is a concatenation of a possibly empty prefix $u=w[l'..t_{i-1}]$ and a possibly empty suffix $v=w[s_i..l'']$. Then the right arm $w[r'..r'']$ is a concatenation of the prefix $v^T=w[r'..t_{i-1}+j]$ and suffix $u^T=w[s_i+j..r'']$, where $j=2|v|+q$. Moreover, $v$ has to be the longest common prefix of words $w[s_i..n]$ and $w[1..t_{i-1}+j]^T$, and $u$ has to be the longest common suffix of words $w[1..t_{i-1}]$ and $w[s_i+j..n]^T$ (see Fig. 2). Since the spacer length $q$ has to be no more than the arm length $|u|+|v|$, we have that $q\leq|u|+|v|$, i.e. $j\leq|u|+3|v|$.

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 $w[s_i..t_{i-1}+j]^T$ and the longest common suffix $LS(j)$ of words $w[1..t_{i-1}]$ and $w[s_i+j..s_{i+1}]^T$. 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:

1. $3LP(j)+LS(j)\geq j$,
2. $j+LS(j)\leq F_i$,
3. $LP(j)<j/2$.

If some $j$ satisfies the above conditions, the algorithm outputs the palindrome $w[l':l'',r':r'']$ where $l'=s_i-LS(j)$, $l''=t_{i-1}+LP(j)$, $r'=s_i+j-LP(j)$, 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 $w[l':l'',r':r'']$ from $P'''(i)$ such that $s_i+\lfloor\frac{F_i}{2^k}\rfloor\leq r''<s_i+\lfloor\frac{F_i}{2^{k-1}}\rfloor$.

**Lemma 2.** *For any palindrome $w[l':l'',r':r'']\in P'''_k(i)$, 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 then the spacer length is no more than $\lfloor\frac{F_i}{2^k}\rfloor$. Then, $l''\geq r'-1-\lfloor\frac{F_i}{2^k}\rfloor\geq s_i$ which contradicts the definition of $P'''(i)$.

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

Moreover, $u$ and $v$ satisfy the relations $|u| < \lfloor\frac{F_i}{2^k}\rfloor$ and $0 < |v| \leq \lfloor\frac{F_i}{2^{k-1}}\rfloor - \lfloor\frac{F_i}{2^k}\rfloor$. Thus, $q \leq |u| + |v| < \lfloor\frac{F_i}{2^{k-1}}\rfloor$, and then $j = 2|u| + q < 2\lfloor\frac{F_i}{2^k}\rfloor + \lfloor\frac{F_i}{2^{k-1}}\rfloor < 2\lfloor\frac{F_i}{2^{k-1}}\rfloor$. On the other hand, from the condition $l'' < t_{i-1}$ we have also $|u| < j - \lfloor\frac{F_i}{2^k}\rfloor$ 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 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 $w[1..t_{i-1} + \lfloor\frac{F_i}{2^k}\rfloor - j]^T$, and the longest common suffix $LS(j)$ of words $w[s_i..t_{i-1} + \lfloor\frac{F_i}{2^k}\rfloor]$ and $w[s_i + \lfloor\frac{F_i}{2^k}\rfloor - j : n]^T$ (Fig. 3). Each palindrome of $P_k'''(i)$ corresponds then to a value $j$ verifying the following conditions:

1. $LP(j) + 3LS(j) \geq j$,
2. $0 < LP(j) \leq \lfloor\frac{F_i}{2^{k-1}}\rfloor - \lfloor\frac{F_i}{2^k}\rfloor$,
3. $LS(j) < \min(\lfloor\frac{F_i}{2^k}\rfloor, j - \lfloor\frac{F_i}{2^k}\rfloor)$.

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

Thus the total time for computing of $P(i)$ is $O(F_{i-1} + F_i)$.

### 3.2 Computing $Q(i)$

Recall that $Q(i)$ contains all palindromes $w[l' : l'', r' : r'']$ which verify $s_i < l'$ and $r'' < t_i$, 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 [KK00b] (see also [KK05]). 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

Each of the sets $P'(i)$, $P''(i)$, $P'''(i)$ is computed in time $O(F_{i-1} + F_i)$, and so is $P(i)$. Summing over all $i$, all involved palindromes are computed in time $O(n)$. Time computed for 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 1.** *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.*

## 4 Length-constrained palindromes

Recall that a gapped palindrome $vuv^T$ is called *length-constrained* if $MinGap \leq |u| \leq MaxGap$ and $MinLen \leq |v|$ for some pre-defined constants $MinGap$, $MaxGap$ and $MinLen$. 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 $MinLen = 3$, $MinGap = 3$ and $MaxGap = 5$, then the palindrome $...a\boxed{gtt}aaca\boxed{ttg}g...$ verifies length constraints but is not maximal, while its extension $...a\boxed{gtta}ac\boxed{attg}g...$ is maximal but does not verify length constraints.

**First step.** Consider an input word $w = w[1..n]$. For a position $i$, we consider words $W(i^+) = w[i..i + MinLen - 1]$ and $W(i^-) = (w[i - MinLen..i - 1])^T$, 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 $w\#w^T\$$. 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 $MinLen$. 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 on the alphabet size [KS03].

**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 $(w[1..i - 1])^T$ and $w[j..n]$). This can be done in constant time using lowest common ancestor queries on the suffix tree for $w\#w^T\$$ [Gus97], but can be also done with the suffix array using the results of [KS03]. 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 $w[i_\ell] = w[i_{\ell+1}] = \ldots = w[i_{\ell+k_\ell}]$ 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 the first item of the current run to the first item of the next run, avoiding the traversal of the whole 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 (valid only for the first item of a 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 $\ell^+$ belongs to equivalence class $C$. We then start the traversal from this position looking for the positions $i$ falling into the interval $[j - MaxGap..j - MinGap]$. 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 run links 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 2.** *For any predefined constants $MinLen$,$MinGap$,$MaxGap$, all length-constrained palindromes can be found in time $O(n + S)$.*

*Proof.* The first step is done in time $O(n)$ using suffix array. At the second step, finding starting positions from intervals $[j - MaxGap, j - MinGap]$ in the list for 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 [KS03].

Algorithm 1 in the Appendix 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 $\ell^+$ 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.)
$NextFirstItem(C)$**:** pointer to the first item in the run following the run containing $PreviousStartItem(C)$.

## 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 computing 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 $w\#w^T\$$, 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 $w\#w^T\$$) in order to implement constant-time common subword queries.

## 6 Concluding remarks

The algorithm for computing long-armed palindromes from Section 3 can be generalized to palindromes $vuv^T$ verifying condition $|u| \leq c|v|$ for some constant $c \geq 1$. The resulting complexity is $O(cn + S)$.

An interesting open question is whether one can compute the reverse Lempel-Ziv factorization in time $O(n)$ independent on the alphabet size.

*Acknowledgments* Part of this work was done during the stay of R. Kolpakov at Inria Lille - Nord Europe, 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)

## References

[ABCD03] J.-P. Allouche, M. Baake, J. Cassaigne, and D. Damanik. Palindrome complexity. *Theor. Comput. Sci*, 292(1):9–31, 2003.

[ABG94] A. Apostolico, D. Breslauer, and Z. Galil. Parallel detection of all palindromes in a string. In *11th Annual Symposium on Theoretical Aspects of Computer Science*, volume 775 of *lncs*, pages 497–506, Caen, France, 24–26 February 1994. Springer.

[BBD$^+$03] T. Biedl, J. Buss, E. Demaine, M. Demaine, M. Hajiaghayi, and T. Vinar. Palindrome recognition using a multidimensional tape. *Theor. Comput. Sci*, 302(1-3):475–480, 2003.

[BG95] D. Breslauer and Z. Galil. Finding all periods and initial palindromes of a string in parallel. *Algorithmica*, 14:355–366, 1995.

[Col69] S. N. Cole. Real-time computation by $n$-dimensional iterative arrays of finite-state machines. *IEEE Transactions on Computers*, 18:349–365, April 1969.

[Coo71] S. Cook. Linear time simulation of deterministic two-way pushdown automata. In *Proceedings of the 5th World Computer Congress, Vol. 1, IFIP'71 (Ljubljana, Yugoslavia, August 23-28, 1971)*, volume 1, pages 75–80, 1971.

[CR94] M. Crochemore and W. Rytter. *Text algorithms.* Oxford University Press, 1994.

[DLDL05] A. De Luca and A. De Luca. Palindromes in Sturmian words. In Clelia de Felice and Antonio Restivo, editors, *Developments in Language Theory, 9th International Conference, DLT 2005, Palermo, Italy, July 4-8, 2005, Proceedings*, volume 3572 of *Lecture Notes in Computer Science*, pages 199–208. Springer, 2005.

[DP99] X. Droubay and G. Pirillo. Palindromes and Sturmian words. *Theoret. Comput. Sci.*, 223:73–85, 1999.

[Dro95] X. Droubay. Palindromes in the Fibonacci word. *Information Processing Letters*, 55(4):217–221, August 1995.

[Gal78] Z. Galil. Palindrome recognition in real time by a multitape turing machine. *Journal of Computer and System Sciences*, 16(2):140–157, April 1978.

[Gus97] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

[KK00a] R. Kolpakov and 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)*, pages 162–168. IEEE, September 2000.

[KK00b] R. Kolpakov and G. Kucherov. On maximal repetitions in words. *Journal of Discrete Algorithms*, 1(1):159–186, 2000.

[KK05] R. Kolpakov and G. Kucherov. Identification of periodic structures in words. In J. Berstel and D. Perrin, editors, *Applied combinatorics on words*, volume Encyclopedia of Mathematics and its Applications, vol. 104 of *Lothaire books*, chapter 8, pages 430–477. Cambridge University Press, 2005.

[KMP77] D. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6:323–350, 1977.

[KS03] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003. Proceedings*, volume 2719 of *Lecture Notes in Computer Science*, pages 943–955. Springer, 2003.

[LJDL07] L. Lu, H. Jia, P. Dröge, and J. Li. The human genome-wide distribution of DNA palindromes. *Functional and Integrative Genomics*, 7(3):221–227, July 2007.

[Man75] G. Manacher. A new linear-time "on-line" algorithm for finding the smallest initial palindrome of a string. *Journ. ACM*, 22(3):346–351, July 1975.

[PB02] A. H. L. Porto and V. C. Barbosa. Finding approximate palindromes in strings. *Pattern Recognition*, 35:2581–2591, 2002.

[Sli73] A. O. Slisenko. Recognition of palindromes by multihead turing machines. In V. P. Orverkov and N. A. Sonin, editors, *Problems in the Constructive Trend in Mathematics VI (Proceedings of the Steklov Institute of Mathematics 129)*, pages 30–202, 1973.

[Sli81] A. Slissenko. A simplified proof of real-time recognizability of palindromes on Turing machines. *J. of Soviet Mathematics*, 15(1):68–77, 1981. Russian original in: *Zapiski Nauchnykh Seminarov LOMI*, 68:123–139, 1977.

[vdSSer] J. van de Snepscheut and J. Swenker. On the design of some systolic algorithms. *J. ACM*, 36(4):826–840, 1989, October.

[WGC$^+$04] P.E. Warburton, J. Giordano, F. Cheung, Y. Gelfand, and 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*, 14:1861–1869, 2004.

## Appendix

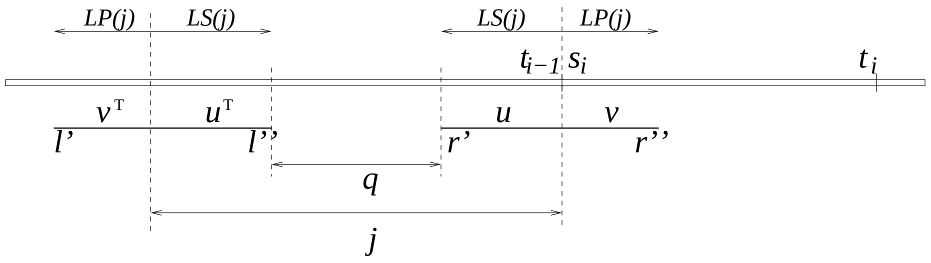

**Fig. 1.** Computing palindromes of $P'(i)$ (Section 3.1)

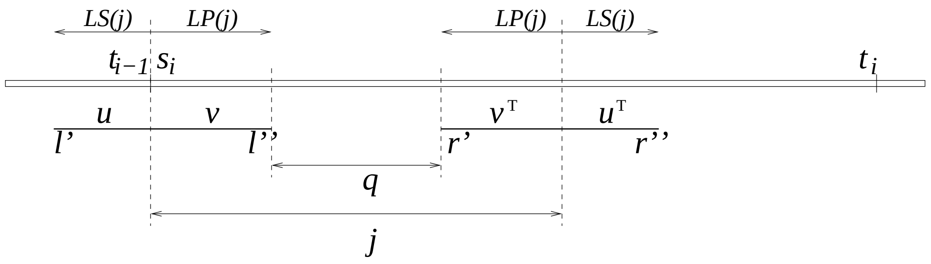

**Fig. 2.** Computing palindromes of $P''(i)$ (Section 3.1)

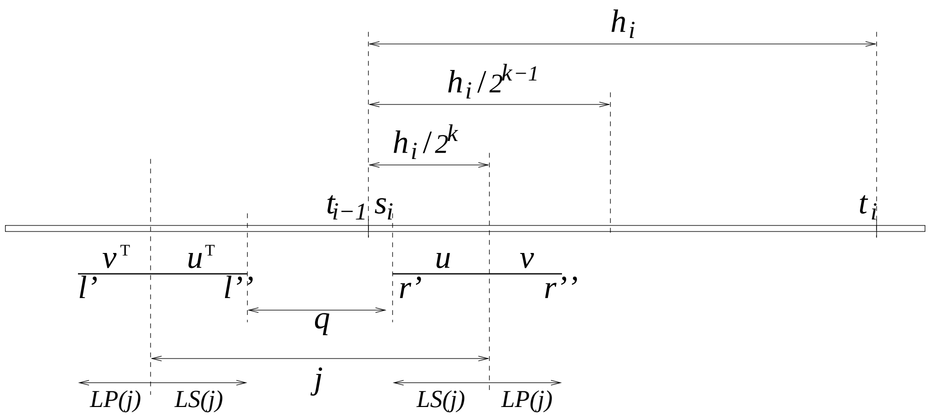

**Fig. 3.** Computing palindromes of $P'''(i)$ (Section 3.1)

```
for $j \leftarrow MinLen + 1$ to $n$ do
    /* insert position $j^-$ to the appropriate list */
    begin
        $C \longleftarrow LeftClass(j)$;
        create a new item $NewItem$ to the list of class $C$;
        $NewItem.position \longleftarrow j$;
        $LastItem(C).NextItem \longleftarrow NewItem$;
        if $w[j] \neq w[LastItem(C).position]$ then
            $LastRun(C).NextRun \longleftarrow NewItem$;
            $LastRun(C) \longleftarrow NewItem$;
        end
        $LastItem(C) \longleftarrow NewItem$;
    end
    /* find all maximal length-constrained palindromes with the right
       arm starting at position $j$ */
    begin
        $C \longleftarrow RightClass(j)$;
        /* find, in the list for class $c$, the first position greater
           than or equal to $(j - MaxGap)$ */
        $SearchItem \longleftarrow PreviousStartItem(C)$;
        while $SearchItem.position < j - MaxGap$ do
            $SearchItem \longleftarrow SearchItem.NextItem$;
            if $SearchItem = NextFirstItem(C)$ then
                $NextFirstItem(C) \longleftarrow SearchItem.NextRun$;
            end
        end
        $PreviousStartItem(C) \longleftarrow SearchItem$;
        /* for each position in the list for class $c$ between
           $(j - MaxGap)$ and $(j - MinGap)$, check if there exists a
           corresponding maximal palindrome */
        while $SearchItem.position \leq (j - MinGap)$ do
            if $w[SearchItem.position] \neq w[j-1]$ then
                $lp \longleftarrow$ length of the longest common prefix of words
                    $w[j + MinLen..n]$ and
                    $(w[1..SearchItem.position - MinLen - 1])^T$;
                output the palindrome $w[SearchItem.position - MinLen - lp :$
                     $SearchItem.position - 1, j : j + MinLen + lp - 1]$;
                $SearchItem \longleftarrow SearchItem.NextItem$;
            end
            else
                if $SearchItem$ is the first item of a run then
                $SearchItem \longleftarrow SearchItem.NextRun$;
                else $SearchItem \longleftarrow NextFirstItem(C)$;
            end
        end
    end
end
```

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