Information Processing Letters 110 (2010) 908–912

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

Counting distinct palindromes in a word in linear time

Richard Groult [a], Élise Prieur [b,*], Gwénaël Richomme [c,d]

[a] Université de Picardie Jules Verne, Laboratoire MIS, 33, rue Saint Leu, F-80039 Amiens cedex 1, France
[b] Université de Rouen, LITIS EA-4108, 1, rue Thomas Becket, F-76821 Mont-St-Aignan cedex, France
[c] Université Paul-Valéry Montpellier 3, UFR 4, Département MIAp, Route de Mende, 34199 Montpellier cedex 5, France
[d] LIRMM (CNRS, Univ. Montpellier 2) – UMR 5506, 161, rue Ada, 34095 Montpellier cedex 5, France

ARTICLE INFO

Article history:
Received 12 October 2009
Received in revised form 16 July 2010
Accepted 16 July 2010
Available online 21 July 2010
Communicated by P.M.B. Vitányi

Keywords:
Palindrome
Palindromic richness
Palindromic fullness
Algorithms

ABSTRACT

We design an algorithm to count the number of distinct palindromes in a word w in time $O(|w|)$, by adapting an algorithm to detect all occurrences of maximal palindromes in a given word and using the longest previous factor array. As a direct consequence, this shows that the palindromic richness (or fullness) of a word can be checked in linear time.

1. Introduction

In recent years, palindromes were studied in many papers in combinatorics on finite and infinite words. For instance, constructions of infinite words using palindromic closure (see, e.g., [7,8]) and the study of palindromic complexity of infinite words (see, e.g., [1,5,8]) have been developed. In [8], Droubay et al. observed that any finite word w of length $|w|$ contains at most $|w|+1$ distinct palindromes (including the empty word). A word w with this maximal number $|w|+1$ of distinct palindromes is called *rich* or *full* by various authors. Subsequently, the notion of palindromic richness has been extended to infinite words and several connected results have been obtained in both the finite and infinite contexts (see, e.g., [11, Section 6.2.2] for a recent survey, [12] for a unified study of structural and combinatorial properties of rich words and [5,16]). S. Labbé indicated to us that a word w can be rich in palindromes (in the sense that it contains many distinct palindromes) without being full of palindromes (in the sense that it cannot contain more). Nevertheless, from now on, we use only the term "rich" since, as far as we know, it is used more often in previous papers than the term "full".

The aim of the present paper is to answer a recent question raised during JORCAD'08 (Journées Rouennaises de Combinatoire et Algorithmique en l'honneur de Jean-Pierre Duval): does there exist a linear-time algorithm to determine whether a word is rich? (Note that the time complexity of our algorithms is evaluated with respect to the length of the processed word.) We answer the previous question in the affirmative by providing a linear-time algorithm on a random access machine that computes the number of distinct palindromes in a word.

Since 1973 and the first algorithm due to Slisenko [18], many algorithms for searching for palindromes have been developed. Manacher proposed an algorithm to find the shortest palindromic prefix of a given finite word [14]. Galil was interested in "palindrome recognition in real-time by a multitape Turing machine" [10]. Different parallel algorithms to find palindromes were also designed [2,4]. Since then, palindromes and their applications have continued to be studied [15,17,19].

* Corresponding author.
E-mail addresses: richard.groult@u-picardie.fr (R. Groult), elise.prieur@univ-rouen.fr (É. Prieur), gwenael.richomme@univ-montp3.fr (G. Richomme).

0020-0190/$ – see front matter
doi:10.1016/j.ipl.2010.07.018

But as far as we know, the aim of all these algorithms is to detect all occurrences of palindromes in a word. Here we are concerned with a slightly different problem, which is to determine all distinct palindromes in a given word. For instance, the word *aa* contains 4 occurrences of palindromes (the empty word; *a*, which occurs twice; and *aa*) but only 3 distinct palindromes. Actually the two problems are closely related, and we next show how to adapt known algorithms to detect all palindromes for our purpose.

Section 2 explains the key element of our algorithm, a result of Droubay et al. [8] showing that the number of distinct palindromes occurring in a word is equal to the number of prefixes of w having a unioccurrent longest palindromic suffix (following [8] and other papers dealing with rich words, we use the term "unioccurrent" for "uniquely occurring"). Thus, we first compute an array LPS that stores, for each prefix p of a word w, the length of the longest palindromic suffix of p (Sections 3 and 4). Second, we determine the number of distinct palindromes in w using the key element (Section 5). Actually, the array LPS is computed in Section 3 from another array LMP that stores, for each prefix p, the length of the longest maximal palindrome that is a suffix of p, i.e., the length of the longest palindromic suffix of p that cannot be extended as a longer palindrome in w. Section 4 explains how a known algorithm for computing the number of occurrences of palindromes in a word can be used to compute LMP. Our resulting algorithm is summarized in Section 6 and an example is provided.

2. Distinct palindromes

We assume that the reader is familiar with basic notions on words (see our references for instance). Here we work over an arbitrary alphabet A and we let ε denote the *empty word*. Given a finite word $w = w_1 w_2 \cdots w_n$ over A with each w_i in A, the *length* of w, denoted by $|w|$, is n. The length of the empty word is 0. The word $w_n w_{n-1} \cdots w_2 w_1$, denoted by $\widetilde{w}$, is called the *reversal* of w. The word w is a *palindrome* if $w = \widetilde{w}$. Recall that a word v is a *factor* (resp. a *prefix*, a *suffix*) of w if $w = pvs$ (resp. $w = vs$, $w = pv$) for some words p and s.

The following observation is the key element of our algorithm.

Proposition 1. *(See [8].) The number of distinct palindromes in w is equal to the number of prefixes of w having a unioccurrent longest palindromic suffix.*

Let us explain this proposition. If two palindromes are suffixes of the same prefix of a word w, then the shorter one is also a prefix of the longer one, and so occurs previously in w. More precisely (see Fig. 1), if p_1 and p_2 are suffixes of the same prefix, the shorter one, say p_1, is a suffix of the longer one, p_2. Moreover, if p_2 is a palindrome, $\widetilde{p}_1$ is a prefix of p_2. When p_1 is a palindrome, this means that $\widetilde{p}_1 = p_1$ is a prefix of p_2. In other words, any first occurrence u of a palindrome in a word w is the longest palindromic suffix of a prefix p of w.

Moreover by the definition of unioccurrence, u is unioccurrent in p, and hence we get Proposition 1.

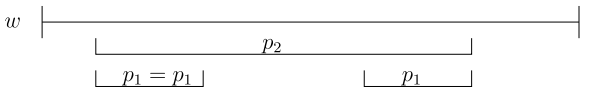

Fig. 1. Palindrome p_1 is a suffix of palindrome p_2, hence p_1 is also a prefix of p_2.

3. Longest palindromic suffixes

The total number of occurrences of all palindromes in a word can be quadratic in the length of the word, as is the case for the word a^n. Hence algorithms (such as Gusfield's [13]) that compute all palindromes in a word actually compute maximal palindromes; a notion we now define. Given a word $w = w_1 \cdots w_n$ with each w_i in A and integers i, j with $1 \leqslant i \leqslant j \leqslant n$, we let $w[i..j]$ denote the factor $w_i w_{i+1} \cdots w_{j-1} w_j$. (An occurrence of) a palindrome $p = w[i..j]$ is said to be *maximal in w* if p is a prefix or a suffix of w (that is $i = 1$ or $j = n$), or if $w[i-1..j+1]$ is not a palindrome.

One can observe that the longest palindromic suffix ending at a position i is not necessarily maximal in w. For instance, in the word $w = abba$, the longest palindromic suffix *bb* ending at position 3 is not maximal. Conversely the longest maximal palindrome ending at position i (if it exists) is not necessarily the longest palindromic suffix ending at position i. For instance, the longest maximal palindrome ending at position 7 in *bbbaabbb* is *bb* but the longest palindrome ending at this position is *bbaabb*. Despite the two previous facts, Proposition 4 below shows that if, for each position i of a word w, we know the longest maximal palindrome in w ending at position i then we can determine the longest palindromic suffix at each position.

Definition 2. Let w be a word. For all i such that $1 \leqslant i \leqslant |w|$, we let $\mathrm{LMP}[i]$ denote the length of the longest maximal palindrome in w ending at position i whenever it exists, and -1 otherwise:

$$\mathrm{LMP}[i] = \max\bigl(\{-1\} \cup \bigl\{\ell \bigm| w[i-\ell+1..i] \text{ is a maximal palindrome in } w\bigr\}\bigr).$$

Definition 3. Let w be a word. For all i such that $1 \leqslant i \leqslant |w|$, we let $\mathrm{LPS}[i]$ denote the length of the longest palindromic suffix ending at position i:

$$\mathrm{LPS}[i] = \max\bigl\{\ell \bigm| w[i-\ell+1..i] \text{ is a palindrome}\bigr\}.$$

Proposition 4. *Let w be a word of length n. Then* $\mathrm{LPS}[n] = \mathrm{LMP}[n]$, *and for all i such that $1 \leqslant i < n$, we have* $\mathrm{LPS}[i] = \max(\mathrm{LMP}[i], \mathrm{LPS}[i+1] - 2)$.

The array LPS *can be computed in linear time from the array* LMP.

Proof. By the definition of maximal palindromes, the longest palindromic suffix of w is maximal, and hence $\mathrm{LPS}[n] = \mathrm{LMP}[n]$.

Let i be such that $1 \leqslant i < n$. Assume that $\mathrm{LPS}[i+1] \leqslant 1$, that is $\mathrm{LPS}[i+1] = 1$. All palindromic suffixes of $w[1..i]$

are maximal, since otherwise the suffix of $w[1..i+1]$ of length $\mathrm{LPS}[i]+2>1$ would be a palindrome. Hence $\mathrm{LPS}[i]=\mathrm{LMP}[i]=\max(\mathrm{LMP}[i],\mathrm{LPS}[i+1]-2)$.

Assume that $\mathrm{LPS}[i+1]\geqslant 2$. This hypothesis implies that the suffix of $w[1..i]$ of length $\mathrm{LPS}[i+1]-2$ is a palindrome, and hence by the definition of LPS, $\mathrm{LPS}[i]\geqslant \mathrm{LPS}[i+1]-2$. Moreover, if $\mathrm{LPS}[i]\neq\mathrm{LMP}[i]$, that is the length of a palindromic suffix of $w[1..i]$ is greater than $\mathrm{LMP}[i]$, then $\mathrm{LPS}[i]+2\leqslant\mathrm{LPS}[i+1]$, and finally $\mathrm{LPS}[i]=\mathrm{LPS}[i+1]-2$. Once again $\mathrm{LPS}[i]=\max(\mathrm{LMP}[i],\mathrm{LPS}[i+1]-2)$.

The last statement in the proposition follows immediately from the given formula to compute $\mathrm{LPS}[i]$ (see lines 4–7 of the algorithm described in Section 6). □

4. Longest maximal palindromic suffixes

In order to prove that array LPS can be computed in linear time, we need to prove the following proposition. This section is devoted to its proof.

Proposition 5. *The array* LMP *can be computed in linear time.*

We first link the notion of longest maximal palindromes ending at a position and the notion of the longest common prefix of two words. Then we explain how to exploit an algorithm given in Gusfield's book [13].

Definition 6. Let w be a word and let i,j be integers with $1\leqslant i,j\leqslant|w|$. We let $\mathrm{lcp}_{w,\widetilde{w}}(i,j)$ denote the length of the *longest common prefix* of the suffix of w beginning at position i and the suffix of the reversal $\widetilde{w}$ of w beginning at position j, i.e., the maximal integer k such that $w[i..i+k-1]=\widetilde{w}[j..j+k-1]$.

Let us say that a palindrome is an *even* (resp. *odd*) palindrome if its length is even (resp. odd). For a palindrome $p=p_1\cdots p_m$ with $p_i\in A$, let us recall that the *centre* of p is the integer i such that $p_ip_{i+1}\cdots p_m=p_{i-1}p_{i-2}\cdots p_1$ if m is even, and $p_ip_{i+1}\cdots p_m=p_ip_{i-1}\cdots p_1$ otherwise.

Note that there is a basic link between maximal palindromes and longest common prefixes. Let w be a word of length n and let k be an integer such that $1\leqslant k\leqslant n$. The maximal odd palindrome centred at position k has length $2\ell-1$ with $\ell=\mathrm{lcp}_{w,\widetilde{w}}(k,n-k+1)$ and ends at position $k+\ell-1$. The maximal even palindrome centred at position k has length 2ℓ with $\ell=\mathrm{lcp}_{w,\widetilde{w}}(k,n-k+2)$ and ends at position $k+\ell-1$. In other words, for each i such that $1\leqslant i\leqslant n$ and for each maximal palindrome of length L ending at position i, there exists a position k such that $L=2\mathrm{lcp}_{w,\widetilde{w}}(k,n-k+1)-1$ and $i=k+\frac{L+1}{2}-1$ if L is odd, or $L=2\mathrm{lcp}_{w,\widetilde{w}}(k,n-k+2)$ and $i=k+\frac{L}{2}-1$ if L is even, and so we get the following result.

Proposition 7. *For any word w and any integer i with $1\leqslant i\leqslant|w|$,* $\mathrm{LMP}[i]$ *is the greatest element of the union of the following three sets*:

$$\begin{cases}\{-1\},\\ \{2\mathrm{lcp}_{w,\widetilde{w}}(k,|w|-k+1)-1\mid \forall k,\ i=k+\mathrm{lcp}_{w,\widetilde{w}}(k,|w|-k+1)-1\},\\ \{2\mathrm{lcp}_{w,\widetilde{w}}(k,|w|-k+2)\mid \forall k,\ i=k+\mathrm{lcp}_{w,\widetilde{w}}(k,|w|-k+2)-1\}.\end{cases}$$

Gusfield gave a linear-time algorithm that computes all maximal palindromes occurring in a word [13, p. 197]. Actually, his algorithm considers each position as the centre of odd and even palindromes and computes, for each position k, the lengths ℓ ($\mathrm{lcp}_{w,\widetilde{w}}(k,n-k+1)$) and ℓ' ($\mathrm{lcp}_{w,\widetilde{w}}(k,n-k+2)$) corresponding to odd and even palindromes of length $2\ell-1$ and $2\ell'$ centred at position k. Thus using Proposition 7, we can compute the array LMP in linear time. Proposition 5 is proved.

5. Unioccurrent longest palindromic suffixes

Using Propositions 4 and 5, the longest palindromic suffix ending at each position of a word can be computed in linear time. By Proposition 1, we need to determine, still in linear time, the prefixes of w having a unioccurrent longest palindromic suffix. We show how this can be done below.

For each position i in a word w ($1\leqslant i\leqslant|w|$), $\mathrm{LPF}[i]$ denotes the length of the longest previous factor at position i, which is the length of the longest factor of w starting at position i that previously occurs in w. More formally:

$$\mathrm{LPF}[i]=\max\{\ell\mid w[i..i+\ell-1]\text{ is a factor of }w[1..i+\ell-2]\cup\{0\}\}.$$

In [6], Crochemore and Ilie proved that the array LPF can be computed in linear time. Let us observe the following.

Fact 8. *For each i, $1\leqslant i\leqslant|w|$, the palindrome $w[i-\mathrm{LPS}[i]+1..i]$ is unioccurrent in $w[1..i]$ if and only if* $\mathrm{LPF}[i-\mathrm{LPS}[i]+1]<\mathrm{LPS}[i]$.

Indeed, for any prefix pv of w, the factor v is unioccurrent in pv if and only if the length of the longest previous factor at $|p|+1$ that previously occurs in w is less than $|v|$.

Corollary 9. *The number of non-empty distinct palindromes in w is*

$$\#\{i\mid 1\leqslant i\leqslant|w|\text{ and }\mathrm{LPF}[i-\mathrm{LPS}[i]+1]<\mathrm{LPS}[i]\}$$

and this quantity can be computed in linear time.

Proof. The first part is a direct consequence of Fact 8 and Proposition 1. Since LPF and LPS can be computed in linear time (resp. by Crochemore and Ilie's algorithm [6] and by Proposition 4), we obtain the desired result. □

6. Complete algorithm

We now summarize our algorithm CountDistinctPalindromes(w) that counts and returns the number of distinct palindromes in a word w.

Algorithm 1. CountDistinctPalindromes(w)

```
1.  n ← |w|
    ▷ Compute the longest previous factors
2.  LPF ← ComputeLPF(w)
    ▷ Construct the longest maximal palindromic suffixes of w
3.  LMP ← LongestMaxPalSuf(w)
    ▷ Compute the longest palindromic suffixes of w
4.  LPS[n] ← LMP[n]
5.  for i ← n − 1 downto 1 do
6.      LPS[i] ← max(LMP[i], LPS[i + 1] − 2)
7.  end for
    ▷ Compute the number of unioccurrent longest palindromic suffixes
    of w including the empty word
8.  NbPal ← 1
9.  for i ← 1 to n do
10.     if LPF[i − LPS[i] + 1] < LPS[i] then
11.         NbPal ← NbPal + 1
12.     end if
13. end for
14. return NbPal
```

Using Gusfield's book [13] together with Propositions 4 and 7 and Corollary 9, we get the following result.

Theorem 10. *The algorithm* CountDistinctPalindromes(w) *returns the number of distinct palindromes occurring in a word w in linear time.*

Notice that the algorithm can be adapted to report all distinct palindromes. Indeed, since the first occurrence of a palindrome is a unioccurrent longest palindromic suffix (see the discussion about Proposition 1), one can modify line 11 of the above algorithm to report the palindrome $w[i - \mathrm{LPS}[i] + 1..i]$, or just the pair $(i - \mathrm{LPS}[i] + 1, i)$ to store all distinct palindromes in linear space.

Example 11.

a) Let $w = abbabaababa$.

i	1	2	3	4	5	6	7	8	9	10	11
w	a	b	b	a	b	a	a	b	a	b	a
LMP	1	1	1	4	3	3	1	−1	3	8	5
LPS	1	1	2	4	3	3	2	4	6	8	5
LPF	0	0	1	2	2	1	3	4	3	2	1

b) Let $w' = abbabaabbba$.

i	1	2	3	4	5	6	7	8	9	10	11
w'	a	b	b	a	b	a	a	b	b	b	a
LMP	1	1	1	4	3	3	1	4	2	2	5
LPS	1	1	2	4	3	3	2	4	**2**	3	5
LPF	0	0	1	2	2	1	3	**2**	3	2	1

The number of distinct palindromes in w is 11, and in w' is 10 since for w', $\mathrm{LPF}[9 - \mathrm{LPS}[9] + 1] = 2$ which is not less than $\mathrm{LPS}[9] = 2$.

7. Discussion and conclusion

We have presented a linear-time method to count all distinct palindromes in a given word. Our algorithm can be used to fulfill our initial aim, which was to decide in linear time whether a word w is rich, i.e., if it contains $|w|$ non-empty distinct palindromes. It can also be used to compute the palindromic defect defined in [5] (the palindromic defect of a word w is the difference between $|w| + 1$ and the number of distinct palindromes in w) in linear time. A previous algorithm based on the same key element but without time optimization was recently provided by A. Blondin-Massé, S. Brlek, A. Garon and S. Labbé for computing this defect [3].

The algorithm we presented uses different algorithms based on suffix trees (see Gusfield [13] mentioned in the proof of Proposition 7) and suffix arrays (see Crochemore and Ilie [6] in the proof of Corollary 9). Notice also that the $\mathrm{lcp}_{w,\widetilde{w}}$ can also be computed using a suffix array (see, e.g., [9]) and the LPF with a suffix tree (as mentioned in [6]).

It seems to be an open problem to find an on-line algorithm to decide if a word is rich (or full), or to compute the palindromic closure of a word. Such an algorithm could exist since a word wa is rich if and only if w is rich and the longest palindromic suffix of wa is unioccurrent in wa.

Acknowledgements

The third author would like to thank Sébastien Labbé for interesting discussions about full and rich words.

The authors especially thank the anonymous referee for carefully reading the manuscript and for providing them with many helpful suggestions which have greatly improved our writing. They are also indebted to Amy Glen and Laurent Mouchard for their proofreading of one of the latest versions of this paper and for all their advice.

References

[1] J.-P. Allouche, M. Baake, J. Cassaigne, D. Damanik, Palindrome complexity, Theoret. Comput. Sci. 292 (2003) 9–31.
[2] A. Apostolico, D. Breslauer, Z. Galil, Parallel detection of all palindromes in a string, Theoret. Comput. Sci. 141 (1–2) (1995) 163–173.
[3] A. Blondin Massé, S. Brlek, A. Garon, S. Labbé, Combinatorial properties of f-palindromes in the Thue–Morse sequence, Pure Math. Appl. 19 (2–3) (2008) 39–52.
[4] D. Breslauer, Z. Galil, Finding all periods and initial palindromes of a string in parallel, Algorithmica 14 (4) (1995) 355–366.
[5] S. Brlek, S. Hamel, M. Nivat, C. Reutenauer, On the palindromic complexity of infinite words, Internat. J. Found. Comput. Sci. 15 (2004) 293–306.
[6] M. Crochemore, L. Ilie, Computing longest previous factor in linear time and applications, Inform. Process. Lett. 106 (2) (2008) 75–80.
[7] A. de Luca, Sturmian words: structure, combinatorics and their arithmetics, Theoret. Comput. Sci. 183 (1997) 45–82.
[8] X. Droubay, J. Justin, G. Pirillo, Episturmian words and some constructions of de Luca and Rauzy, Theoret. Comput. Sci. 255 (2001) 539–553.
[9] J. Fischer, V. Heun, Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE, in: M. Lewenstein, V. Valiente (Eds.), Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching, in: Lecture Notes in Comput. Sci., vol. 4009, 2006, pp. 36–48.
[10] Z. Galil, Palindrome recognition in real time by a multitape Turing machine, J. Comput. System Sci. 16 (2) (1978) 140–157.
[11] A. Glen, J. Justin, Episturmian words: A survey, RAIRO Inform. Theor. Appl. 43 (2009) 403–442.
[12] A. Glen, J. Justin, S. Widmer, L.Q. Zamboni, Palindromic richness, European J. Combin. 30 (2009) 510–531.
[13] D. Gusfield, Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology, Cambridge University Press, New York, NY, USA, 1997.

[14] G. Manacher, A new linear-time on-line algorithm finding the smallest initial palindrome of a string, J. ACM 22 (3) (1975) 346–351.
[15] S.J. Pan, R.C.T. Lee, Looking for all palindromes in a string, in: The 23rd Workshop of Combinatorial Mathematics and Computational Theory, 2006, pp. 166–169; see http://algo2006.csie.dyu.edu.tw/paper/2/A24.pdf.
[16] A. Restivo, G. Rosone, Burrows-wheeler transform and palindromic richness, Theoret. Comput. Sci. 410 (30–32) (2009) 3018–3026.
[17] M.-F. Sagot, A. Viari, Flexible identification of structural objects in nucleic acid sequences: palindromes, mirror repeats, pseudoknots and triple helices, in: A. Apostolico, J. Hein (Eds.), Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching, in: Lecture Notes in Comput. Sci., vol. 1264, Aarhus/Springer-Verlag, Berlin/Denmark, 1997, pp. 224–246.
[18] 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: Proc. Steklov Inst. Math., vol. 129, 1973, pp. 30–202.
[19] J.T. Udding, The maximum length of a palindrome in a sequence, in: W.H.J. Feijen, A.J.M. van Gasteren, D. Gries, J. Misra (Eds.), Beauty is Our Business, Springer-Verlag, Berlin, 1990, pp. 410–416 (Chapter 49).