Finding Gapped Palindromes Online

Yuta Fujishige[1(✉)], Michitaro Nakamura[2], Shunsuke Inenaga[1], Hideo Bannai[1], and Masayuki Takeda[1]

[1] Department of Informatics, Kyushu University, Fukuoka, Japan
{yuta.fujishige,inenaga,bannai,takeda}@inf.kyushu-u.ac.jp
[2] Department of Physics, Kyushu University, Fukuoka, Japan

Abstract. A string s is said to be a *gapped palindrome* iff $s = xyx^R$ for some strings x, y such that $|x| \geq 1$, $|y| \geq 2$, and x^R denotes the reverse image of x. In this paper we consider two kinds of gapped palindromes, and present efficient *online* algorithms to compute these gapped palindromes occurring in a string. First, we show an online algorithm to find all maximal *g-gapped palindromes* with fixed gap length $g \geq 2$ in a string of length n in $O(n \log \sigma)$ time and $O(n)$ space, where σ is the alphabet size. Second, we show an online algorithm to find all maximal *length-constrained gapped palindromes* with arm length at least $A \geq 1$ and gap length in range $[g_{\min}, g_{\max}]$ in $O(n(\frac{g_{\max} - g_{\min}}{A} + \log \sigma))$ time and $O(n)$ space. We also show that if A is a constant, then there exists a string of length n which contains $\Omega(n(g_{\max} - g_{\min}))$ maximal LCGPs, which implies we cannot hope for a significant speed-up in the worst case.

1 Introduction

A *palindrome* is a string of form xax^R, where x is a string called the left arm, a is either the empty string or a single character, and x^R is the reversed string of x called the right arm. Finding palindromic substrings in a given string w is a classical problem on string processing. The earliest work on this problem dates back to at least 1970's when Manacher [10] proposed an online algorithm to find all prefix palindromes in w in $O(n)$ time, where n is the length of w. Later, Apostolico et al. [1] pointed out that Manacher's algorithm can be used to find all maximal palindromes in w in $O(n)$ time, where a maximal palindrome is a substring palindrome $w[i..j] = w[i..j]^R$ of w whose arms cannot be further extended based on the same center $\frac{i+j}{2}$.

A natural generalisation of palindromes is *gapped palindromes* of form xyx^R, where y is a string of length at least 2 called a *gap*[1]. Finding gapped palindromes has applications in bioinformatics, e.g.; RNA secondary structures called hairpins can be regarded as a kind of gapped palindrome $xy\overline{x}^R$, where $\overline{x}$ represents the complement of x ($\overline{x}$ is obtained by exchanging A with U and exchanging C with G in x). The most basic type of gapped palindromes is *g-gapped palindromes*, where $g \geq 2$ is a pre-defined fixed length of the gaps. For three parameters $g_{\min}$,

[1] If y is a single character, then xyx^R is a palindrome of odd length. Thus we here assume y is of length at least 2.

© Springer International Publishing Switzerland 2016
V. Mäkinen et al. (Eds.): IWOCA 2016, LNCS 9843, pp. 191–202, 2016.
DOI: 10.1007/978-3-319-44543-4_15

$g_{\max}$, and A such that $2 \leq g_{\min} \leq g_{\max}$ and $A \geq 1$, Kolpakov and Kucherov [8] introduced *length-constrained gapped palindromes* (*LCGPs*) which has arms of length at least A and gaps of length in range $[g_{\min}, g_{\max}]$. This is a natural generalisation of g-gapped palindromes with $g_{\min} = g_{\max} = g$ and $A = 1$.

In this paper, we consider the problems of finding these gapped palindromes in a string in an *online manner*. Namely, our input is a growing string to which new characters can be appended, and each character of the string arrives one by one, from left to right. Let n be the length of the final string w. We propose:

(1) An online algorithm to compute all maximal g-gapped palindromes in w in $O(n \log \sigma)$ time and $O(n)$ space, where σ is the alphabet size. This algorithm can be modified to output only *distinct* maximal g-gapped palindromes in an online manner, in the same complexity.
(2) An online algorithm to compute all maximal LCGPs in w in $O(n(m+\log \sigma))$ time and $O(n)$ space, where $m = \max\{\frac{g_{\max}-g_{\min}}{A}, 1\}$.

Formal definitions of the maximality of these gapped palindromes will be given in Sects. 3 and 4, respectively.

We remark that using a slightly modified version of Solution (1), it is trivial to obtain an $O(n(g_{\max} - g_{\min} + \log \sigma))$-time solution for finding all maximal LCGPs, by simply testing gap lengths $g_{\min}, g_{\min}+1, \ldots, g_{\max}$ separately. Hence, in the case where A is not a constant and $\log \sigma$ is not a dominating term, then Solution (2) speeds up this trivial method by a factor of A. On the other hand, in the case where A is a constant, then we show that there exists a string of length n which contains $\Omega(nm)$ maximal LCGPs, meaning that we cannot hope significant speed-up in the worst case.

Solution (2) is based on Solution (1) and is quite different from the offline solution by Kolpakov and Kucherov [8]. To our knowledge, these are the first efficient online algorithms that compute *any kind* of gapped palindromes.

Related work. A number of efficient *offline* algorithms for computing various kinds of gapped palindromes have been proposed in the literature.

Let w be an input string w of length n over the integer alphabet. There exists a folklore $O(n)$-time algorithm (see e.g. [6]) which finds all maximal g-gapped palindromes for a given fixed gap length g; the suffix tree [4,12] of string $w^R\#w\$$ and a constant-time LCA data structure [2] over the suffix tree are constructed during preprocessing, and then computing each maximal g-gapped palindrome reduces to an outward longest common extension (LCE) query, which can be answered by an LCA query on the tree. Our algorithm for computing all maximal g-gapped palindromes can be regarded as an online version of this algorithm.

Kolpakov and Kucherov [8] proposed an $O(n + L)$-time offline algorithm to find all maximal LCGPs, where L is the number of outputs. Their algorithm consists of the following two steps: In the first step, it computes all (not necessarily outward maximal) LCGPs whose arms are of length exactly A. Let (i, j) be the pair of the ending position i and the beginning position j of the left and right arms of each of the above LCGPs, respectively. In the second step, for each LCGP computed above, the algorithm performs an outward LCE query from

i and j, using the same suffix-tree based data structure as for the maximal g-gapped palindromes above. However, each time a new character is appended to the growing string, the LCE value from the same pair of positions may increase, and it is impossible to know beforehand when the growth of the LCE value for each pair of positions stops. Thus, it seems difficult to apply Kolpakov and Kucherov's solution to our online setting.

There exist efficient offline solutions for finding other kinds of gapped palindromes. Kolpakov and Kucherov [8] also proposed an $O(n)$-time[2] offline algorithm to compute all maximal *long-armed palindromes* (those whose arms are longer than their gap) in a given string w of length n. Kolpakov and Kucherov's algorithm uses a variant of Lempel-Ziv factorisation called the reversed LZ factorisation of strings. Let $f_1, \ldots, f_k$ be the reversed LZ factorisation of w. Then, for each pair f_i of adjacent factors, their algorithm focuses on positions $\frac{|f_i|}{2^k}$ for every $1 \leq k \leq \lceil \frac{|f_i|}{2} \rceil$ in f_i. This implies that the length of each f_i needs to be precomputed. However, in the online setting, the length of the last factor that is a suffix of the current string can extend each time a new character is appended. It is therefore unclear whether we can extend their solution to the online scenario.

Very recently, Gawrychowski et al. [5] considered a generalisation of long-armed palindromes called α-*gapped palindromes*; For a parameter $\alpha > 1$, a gapped palindrome xyx^R is said to be an α-gapped palindrome iff $|xy| \leq \alpha|y|$. Gawrychowski et al. [5] proposed an $O(\alpha n)$-time offline algorithm which computes all maximal α-gapped palindromes in an input string w of length n. This algorithm requires a preprocessing of the input w for integer $c \geq 2$ such that the occurrences of a substring of length 2^k (called a basic factor therein) in another substring of length $c2^k$ can be computed efficiently. Thus, it seems difficult to apply their result to the online setting.

2 Preliminaries

2.1 Strings

Let Σ be an ordered alphabet of size σ. An element of Σ^* is called a *string*. The length of string w is denoted by $|w|$. The empty string is denoted by ε. For any non-empty string w, $w[i]$ denotes the character at position i of w for $1 \leq i \leq |w|$, and $w[i..j]$ denotes the substring of w that begins at position i and ends at position j in w for $1 \leq i \leq j \leq |w|$. For convenience, let $w[i..j] = \varepsilon$ for $i > j$. For $0 \leq i \leq |w| + 1$, $w[1..i]$ and $w[i..|w|]$ are called a *prefix* and a *suffix* of w, respectively. Let w^R denotes the *reversed image* of w, namely, $w^R = x[|x|] \cdots x[1]$. For instance, if $w =$ `desserts`, then $w^R =$ `stressed`. For any strings x and y, let $lcp(x, y)$ denote the length of the longest common prefix of x and y.

[2] Originally, Kolpakov and Kucherov [8] stated their algorithm works in $O(n+S)$ time, where S is the number of outputs. It follows from a recent work by Gawrychowski et al. [5] that $S = O(n)$.

2.2 Gapped Palindromes

A string p is said to be a *gapped palindrome* iff $p = xyx^R$ for some non-empty strings x, y with $|y| > 1$. The intervals $[1, |x|]$, $[|y|+1, |xy|]$, and $[|xy|+1, |xyx|]$ in p are called the *left arm*, *gap*, and *right arm* of gapped palindrome $p = xyx^R$. Note that in general the choice of arms and gap are not unique for the same string p. For instance, if $p = \mathtt{abccbba}$, then we can take $x = \mathtt{ab}$ and $y = \mathtt{ccb}$, or $x = \mathtt{a}$ and $y = \mathtt{bccbb}$.

A gapped palindrome xyx^R is said to be a *length-constrained palindrome* (*LCGP*) iff $|x| \geq A$ and $g_{\min} \leq |y| \leq g_{\max}$ for some fixed integer parameters $A \geq 1$ and $1 < g_{\min} \leq g_{\max}$. A gapped palindrome xyx^R is said to be a *g-gapped* palindrome iff $|y| = g$ for some fixed integer $g > 1$. Note that any g-gapped palindrome is a special case of a length-constrained palindrome with $g_{\min} = g_{\max} = g$ and $A = 1$.

An occurrence of a gapped palindrome $p = xyx^R$ in a string w is identified by a triple (i, j, a) such that a denotes the length of each arm, and i, j denote the ending and beginning positions of the left and right arms of p, respectively. Namely, $w[i-a+1..i] = x$, $w[i+1..j-1] = y$, and $w[j..j+a-1] = x^R$. The *center* of an occurrence (i, j, a) of a gapped palindrome in w is $\frac{i+j}{2}$.

2.3 Suffix Trees and LCE Queries

The suffix tree of a string w, denoted $STree(w)$, is a path-compressed trie which represents all suffixes of w. More formally, $STree(w)$ is an edge-labelled rooted tree such that (1) Every internal node is branching; (2) The out-going edges of every internal node begin with mutually distinct characters; (3) Each edge is labelled by a non-empty substring of w; (4) For each suffix s of w, there is a unique path from the root which spells out s (the path possibly ends on an edge). It follows from the definition of $STree(w)$ that if $n = |w|$ then the number of nodes and edges in $STree(w)$ is $O(n)$. By representing every edge label x by a pair (i, j) of integers such that $x = w[i..j]$, $STree(w)$ can be represented with $O(n)$ space.

For any node v of $STree(w)$, let $str(v)$ denotes the substring of w that is obtained by concatenating the edge labels in the path from the root to v. Each node v stores the length $|str(v)|$ of the string it represents. For each non-root node v, let $slink(v) = (v, u)$ be a reversed edge called the *suffix link* of v, such that $str(u) = str(v)[2..|str(v)|]$. It is well-known that $STree(w)$ with the suffix links of all nodes can be constructed online in $O(n \log \sigma)$ time and $O(n)$ space [11].

The *locus* of a substring x of w in $STree(w)$ is the ending point of the path P_x that spells out x from the root. If the ending point of P_x lies on an edge label, then the locus is represented by triple $\langle u, s, t \rangle$ such that u is the deepest node in the path P_x and s, t are positions of w with $str(u)w[s..t] = x$.

Given an ordered pair (i, j) of positions in a string w of length n, a *reversed longest common extension query* $rlce_w(i, j)$ returns $lcp((w[1..i])^R, w[j..n])$. Computing $rlce_w(i, j)$ reduces to the lowest common ancestor (LCA) problem on $STree(w')$, where $w' = w^R\#w$ and $\#$ is a special delimiter which does not

occur in w. Let $v_{i,j}$ be the LCA of the two leaves which represent the suffixes $w'[n-i+1..2n+1]$ and $w'[n+j+1..2n+1]$. Then, we have that $|str(v_{i,j})| = rlce_w(i,j)$. Using an LCA data structure (e.g. [2]), we can answer $rlce_w(i,j)$ query for any pair (i,j) of positions in $O(1)$ time after an $O(n)$-time preprocessing on $STree(w')$.

3 Online Algorithms to Compute All Maximal *g*-gapped Palindromes

An occurrence (i,j,a) of a g-gapped palindrome xyx^R in a string w is said to be *maximal*, if the arms x, x^R cannot be extended outward, i.e., if $w[b-1] \neq w[e+1]$, $b = 1$, or $e = n$, where $b = i - a + 1$ and $e = j + a - 1$[3].

Example 1. Consider string aabaacabbcaabb and let $g = 3$. This string has 3-gapped maximal palindromes $(1,5,1) = \mathtt{a} \cdot \mathtt{aba} \cdot \mathtt{a}$, $(6,10,4) = \mathtt{baac} \cdot \mathtt{ab}$ $\mathtt{b} \cdot \mathtt{caab}$, $(7,11,1) = \mathtt{a} \cdot \mathtt{bbc} \cdot \mathtt{a}$, and $(9,13,2) = \mathtt{bb} \cdot \mathtt{caa} \cdot \mathtt{bb}$.

3.1 Computing all Maximal *g*-gapped Palindromes Online

In this subsection, we propose online algorithms to compute all maximal g-gapped palindromes in a string w of length n, where $g > 1$ is a given fixed integer parameter (since $g = 1$ gives odd palindromes, we set $g > 1$).

As was mentioned in Sect. 1, there exists an *offline* algorithm which computes all g-gapped maximal palindromes in $O(n)$ time and space for an input string w of length n over an integer alphabet. However, in our scenario the input string w is given online, and we wish to process each character from left to right. In the sequel, we will show our online algorithm which can deal with this setting.

For each $k = 1, \ldots, n$, our algorithm maintains the *longest* g-gapped suffix palindrome of $w[1..k]$ (if it exists). For each g-gapped palindrome to compute, we maintain two variables i, j $(i < j < k)$ that represent the ending position of the left arm and the beginning position of the right arm of g-gapped palindrome, respectively. Assume $(i, j, a_{i,j})$ is the longest g-gapped suffix palindrome of $w[1..k]$, where the gap of length g is $w[i+1..j-1]$, $j = i + g + 1$ and $j + a_{i,j} - 1 = k$. In case there are no g-gapped suffix palindromes of $w[1..k]$, then let $a_{i,j} = 0$, $i = k - g$ and $j = k + 1$. Depending on the next character $w[k+1]$, we have two cases:

1. If $w[i - a_{i,j}] = w[k+1]$, then there exists a longer g-gapped palindrome centered at $\frac{i+j}{2}$. We then naïvely extend the arm length by $a_{i,j} \leftarrow a_{i,j} + 1$, and proceed to the forthcoming character by updating $k \leftarrow k + 1$.

[3] Since the gap length is fixed to g and since it simplifies the description of the algorithm, here we do not consider inward maximality of the arms. However, it is easy to modify our algorithm so that it outputs all g-gapped palindromes that are both outward and inward maximal with the same efficiency.

2. If $w[i-a_{i,j}] \neq w[k+1]$, then it appears that $(i, j, a_{i,j})$ is the longest g-gapped maximal palindrome ending at position k, and hence we output it. We then shift the gap to the right by updating $i \leftarrow i+1$ and $j \leftarrow j+1$. There are two-sub cases.
   (a) If $j > k+1$, then it appears that there is no g-gapped suffix palindrome of $w[1..k+1]$. We therefore update $k \leftarrow k+1$ and proceed to the forthcoming character, with the current values of i and j.
   (b) If $j \leq k+1$, then we compute $a_{i,j}$ (we will later describe how to efficiently compute it for updated i and j). There are two sub-cases:
      i. If $j + a_{i,j} - 1 = k + 1$, then $(i, j, a_{i,j})$ is the longest g-gapped suffix palindrome of $w[1..k+1]$. We proceed to the forthcoming character by updating $k \leftarrow k+1$.
      ii. If $j + a_{i,j} - 1 < k + 1$, then $(i, j, a_{i,j})$ is the maximal g-gapped palindrome with the gap beginning at position $i+1$, and hence we output it. We then shift the gap to the right by updating $i \leftarrow i+1$ and $j \leftarrow j+1$, and go to either Case 2a or Case 2b depending on the value of j.

In order to efficiently compute $a_{i,j}$ of Case 2 above in our online scenario, we utilize the following results:

Theorem 1 ([7]). *There exists an $O(n \log \sigma)$-time $O(n)$-space algorithm to maintain the suffix tree with suffix links for a bidirectionally growing string to which new characters can be prepended and appended, where n is the length of the final string.*

Theorem 2 ([3]). *There exists a linear-space algorithm for a rooted tree that supports the following operations and query in $O(1)$ worst-case time: which supports the following operations and query in $O(1)$ worst-case time: (1) Insert a new node; (2) Delete an existing node; (3) LCA query for any pair of nodes in the current tree.*

We are ready to show the main result of this section:

Theorem 3. *For a growing string to which new characters are appended, we can compute all maximal g-gapped palindromes in an online manner, in $O(n \log \sigma)$ time and $O(n)$ space, where n is the length of the final string.*

Proof. The correctness immediately follows from the above arguments.

The time complexity is shown as follows. In the sequel, we consider the amortised time cost for each $k = 1, \ldots, n$. For each k that falls into Case 1, it clearly takes $O(1)$ time. For each k that falls into Case 2b, we output several maximal g-gapped palindromes. It takes $O(1)$ time to output the longest maximal g-gapped palindrome. The key is how to compute the arm lengths $a_{i,j}$ of shorter maximal g-gapped palindromes. For this sake we maintain $STree(w'_k)$ where $w'_k = (w[1..k])^R \# w[1..k]$, where # is a special delimiter which does not appear elsewhere in w'_k (see also Fig. 1 for an example).

Note that computing $a_{i,j}$ is equivalent to computing $rlce_{w[1..k]}(i, j)$, and thus

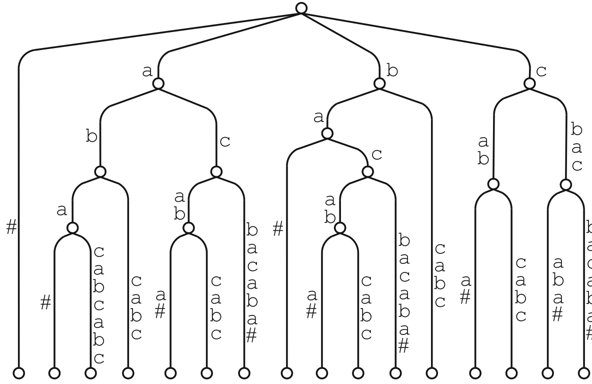

Fig. 1. $STree(w'_k)$ with $w[1..k] =$ `abacabcabc` and $w'_k =$ `cbacbacaba#abacabcabc`. The label strings after # are omitted for simplicity.

is equivalent to computing $|str(v_{i,j})|$, where $v_{i,j}$ is the LCA of the nodes of $STree(w'_k)$ which represent the suffixes $w'_k[k-i+1..2k+1]$ and $w'_k[k+j+1..2k+1]$ of w'_k. Since # is unique in w'_k, the suffix $w'_k[k-i+1..2k+1]$ is always represented by a leaf of $STree(w'_k)$ and hence can easily be accessed in $O(1)$ time. However, notice that the other suffix $w'_k[k+j+1..2k+1]$ is not represented by a node when the path that spells out $w'_k[k+j+1..2k+1]$ from the root ends on an edge (this can happen when $w'_k[k+j+1..2k+1] = w[j..k]$ is a prefix of another suffix of $w[1..k]$). Consider such a case, and let $\langle u_j, s_j, t_j \rangle$ be the locus for the suffix $w'_k[k+j+1..2k+1]$. Since u_j is the nearest ancestor to the locus, we can use u_j for the LCA query instead of the locus for $w'_k[k+j+1..2k+1]$.

What remains is how to quickly find the loci for increasing j. For this we can use a similar technique to Ukkonen's online suffix tree construction algorithm [11]: Assume that the locus $\langle u_j, s_j, t_j \rangle$ for the suffix $w'_k[k+j+1..2k+1] = w[j..k]$ in $STree(w'_k)$ is given. To find the locus for $\langle u_{j+1}, s_{j+1}, t_{j+1} \rangle$ for the next suffix $w'_k[k+j+2..2k+1] = w[j+1..k]$, we first follow the suffix link of u_j and arrive at $z = slink(u_j)$. We then traverse the path from z which spells out $w'_k[s_{j+1}..t_{j+1}]$. The last piece of this path gives the locus $\langle u_{j+1}, s_{j+1}, t_{j+1} \rangle$ (see also Fig. 2).

Using a similar analysis to [11], the cost to find this locus is amortised to $O(\log \sigma)$. Since the total number of outputs (maximal g-gapped palindromes) is linear in n, the amortised cost per output is $O(\log \sigma)$. The cost to update $STree(w'_k)$ to $STree(w'_{k+1})$ is amortised to $O(\log \sigma)$ by Theorem 1. Each LCA query can be answered in $O(1)$ time by Theorem 2. Hence, the total time complexity is $O(n \log \sigma)$. The total space requirement is clearly $O(n)$. This completes the proof. □

3.2 Computing all Distinct Maximal g-gapped Palindromes Online

Consider a g-gapped palindrome $p = xyx^R$ which has at least two maximal occurrences in a string w. When considering "distinctness" of two maximal occurrences (i, j, a) and (i', j', a) of p, we take into account the left and right neighbouring characters for a technical reason. Namely, two maximal occurrences (i, j, a) and (i', j', a) of a g-gapped palindromes are said to be *distinct* iff (1) $w[b-1] \neq w[b'-1]$ or (2) $w[e+1] \neq w[e'+1]$, where $b = i-a+1$, $e = j+a-1$, $b' = i'-a+1$, and $e' = j' + a - 1$.

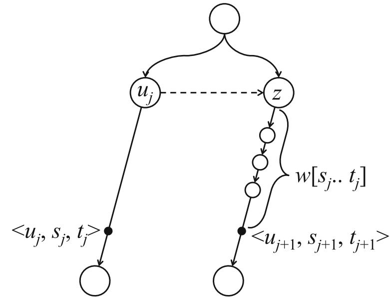

Fig. 2. Illustration of how to find the locus $\langle u_{j+1}, s_{j+1}, t_{j+1}\rangle$ of the next suffix $w'_k[k + j + 2..2k + 1] = w[j+1..k]$ using the suffix link of u_j, where $\langle u_j, s_j, t_j\rangle$ is the locus of the previous suffix $w'_k[k + j + 1..2k + 1] = w[j..k]$. The cost for walking down from node z to the locus for $\langle u_{j+1}, s_{j+1}, t_{j+1}\rangle$ is $O(\log \sigma)$ amortised.

Our online algorithm of Sect. 3.1 can be modified to output all distinct maximal g-gapped palindromes in an online manner.

For any string w, let *lusuf*(w) denote the longest suffix of w which appears at least twice in w (we assume that the empty string ε appears $|w|+1$ times in w so *lusuf*(w) always exists). We make use of the following simple observation:

Observation 1. *Let (i, j, a) be an occurrence of a maximal g-gapped palindrome xyx^R in a string w, and let $c_\ell = w[i-a]$ and $c_r = w[j+a]$. Then, it is the first (i.e. left-most) maximal occurrence of xyx^R in w iff $|c_\ell xyx^R c_r| = j-i+2a+1 > |lusuf(w[1..j+a-1])|$.*

Theorem 4. *For a growing string to which new characters are appended, we can compute all distinct maximal g-gapped palindromes in an online manner, in $O(n \log \sigma)$ time and $O(n)$ space, where n is the length of the final string.*

Proof. On top of *STree*(w'_k) used in Theorem 3, we build another suffix tree *STree*$(w[1..k])$ for increasing $k = 1, \ldots, n$ using Ukkonen's online algorithm [11]. For each k, Ukkonen's algorithm maintains an invariant called the *active point* which indicates the locus of *lusuf*$(w[1..k])$. When we process the kth character $w[k]$, we store $|lusuf(w[1..h])|$ for all $1 \leq h \leq k$. Let $(i, j, a_{i,j})$ be an occurrence of a maximal g-maximal found at the k-th stage of the algorithm where we have processed $w[1..k]$. Then, we can determine in $O(1)$ time whether or not it is the first maximal occurrence of the g-gapped palindrome using Observation 1 (recall that the right mismatched position $j + a_{i,j}$ never exceeds k and hence we know $|lusuf(w[1..j+a_{i,j}])|$). Since Ukkonen's online algorithm works in $O(n \log \sigma)$ time and $O(n)$ space, the theorem holds. ☐

We note that a similar technique was used by Kosolobov et al. [9] in their online algorithm to find all distinct palindromes (without gaps) in a given string.

4 Online Algorithms to Compute all Maximal LCGPs

An occurrence (i, j, a) of an LCGP in a string w of length n is said to be *outward-maximal* iff $w[i-a] \neq w[j+a]$, $i-a+1=1$, or $j+a-1=n$, and it is said to be *inward-maximal* iff $w[i+1] \neq w[j-1]$. It is said to be *maximal* iff it is both outward-maximal and inward-maximal[4].

Example 2. Consider string aabaacabbcaabb and let $g_{\min} = 1$, $g_{\max} = 4$, and $A = 2$. All the maximal LCGPs in this string are $(2, 4, 2) =$ aa · b · aa, $(4, 7, 2) =$ ba · ac · ab, $(6, 10, 4) =$ baac · abb · caab, and $(9, 13, 2) =$ bb · caa · bb.

4.1 Computing all Maximal LCGPs Online

In this section, we present an online algorithm to compute all maximal LCGPs of a given string w. This algorithm works in $O(n(m + \log \sigma))$ time and $O(n)$ space, where $n = |w|$ and $m = \max\{\frac{g_{\max}-g_{\min}}{A}, 1\}$.

Let $d = \frac{g_{\max}-g_{\min}}{2}$. For ease of explanation, we assume that $d \bmod A = 0$ and we will describe our algorithm for this case. However, the algorithm can easily be extended to a general case with $d \bmod A \neq 0$, retaining the same efficiency.

For each $k = 1, \ldots, n$ in increasing order, we maintain a pair (i, j) of positions such that $j - i = g_{\min} + 1$ and the longest inward-maximal suffix LCGP of $w[1..k]$ is centered at $\frac{i+j}{2}$ (if it exists). If it does not exist, then let $i = k - g_{\max}$ and $j = k - g_{\max} + g_{\min} + 1$. For $1 \leq l \leq \frac{d}{A}$, we consider the positions $i - l \cdot A$ and $j + l \cdot A$ in $w[1..k]$, called *sampled* positions. The following simple lemma suggests how we can use these sampled positions for efficient computation of LCGPs.

Lemma 1. *Let (i', j', a') be any maximal LCGP whose center is $\frac{i+j}{2}$ (i.e., $\frac{i'+j'}{2} = \frac{i+j}{2}$). Then, there exists l $(1 \leq l \leq \frac{d}{A})$ such that $j + l \cdot A \in [j', j' + a' - 1]$ and $i - l \cdot A \in [i' - a' + 1, i']$. Moreover, for each such l, (i', j', a') is the unique maximal LCGP satisfying the above conditions.*

Proof. The existence of l is clear from the fact that the arms of LCGPs must be at least A long (see also Fig. 3). By definition, the arms of two different maximal LCGPs with the same center cannot overlap. Thus, for each l, there exists at most one LCGP whose left and right arms contain sampled positions $i - l \cdot A$ and $j + l \cdot A$, respectively. This completes the proof. □

Let l $(1 \leq l \leq \frac{d}{A})$ be the smallest integer such that $i - l \cdot A$ (resp. $j + l \cdot A$) is contained in the left arm (resp. the right arm) of the longest suffix inward-maximal LCGP of $w[1..k]$ that is centered at $\frac{i+j}{2}$, and let a_l be the length of the arm of this LCGP. Also, let i_l, j_l be the ending position of the left arm and the beginning position of the right arm of this LCGP, respectively. Note $\frac{i_l+j_l}{2} = \frac{i+j}{2}$ and $j_l + a_l - 1 = k$. Depending on the next character $w[k+1]$, we have two cases:

[4] Since the gap length varies in range $[g_{\min}, g_{\max}]$, we here consider both outward and inward maximality of the arms.

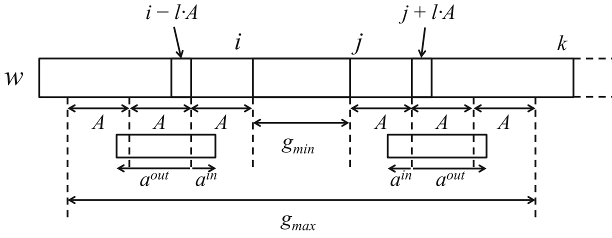

Fig. 3. Illustration for Lemma 1. Since any LCGP centered at $\frac{i+j}{2}$ with gap length in range $[g_{\min}, g_{\max}]$ contains a pair $(i - l \cdot A, j + l \cdot A)$ of sampled positions for some l, we can compute it by two LCEs from the sampled positions.

1. If $w[i_l - a_l] = w[k+1]$, then $(i_l, j_l, a_l + 1)$ is the longest suffix inward-maximal LCGP of $w[1..k+1]$ centered at $\frac{i+j}{2}$. Thus, we naïvely extend the arm length outward by $a_l \leftarrow a_l + 1$, and proceed to the forthcoming character by updating $k \leftarrow k + 1$.
2. If $w[i_l - a_l] \neq w[k + 1]$, then it appears that (i_l, j_l, a_l) is a maximal LCGP centered at $\frac{i+j}{2}$ and ending at position k, and hence we output it. To compute other maximal LCGPs centered at $\frac{i+j}{2}$, we do the following: We update $l \leftarrow l + 1$, and consider a pair $(i - l \cdot A, j + l \cdot A)$ of the sampled positions and compute the outward LCE $a_l^{out} = rlce_{w[1..k+1]}(i - l \cdot A, j + l \cdot A)$ and the inward LCE $a_l^{in} = rlce_{w[1..k+1]}(j + l \cdot A - 1, i - l \cdot A + 1)$ from these sampled positions (see also Fig. 3). There are three sub-cases depending on the LCE values:
   (a) If $a_l^{out} + a_l^{in} < A$ or $a_l^{in} > l \cdot A$, then there is no maximal LCGP with gap length in range $[g_{\min}, g_{\max}]$ that is centered at $\frac{i+j}{2}$ and contains the sampled positions $i - l \cdot A$ and $j + l \cdot A$. We update $l \leftarrow l + 1$, and go to one of the following sub-cases.
      i. If $l \leq \frac{d}{A}$, then we compute the outward and inward LCEs from the pair of sampled positions with l.
      ii. If $l > \frac{d}{A}$, then there is no suffix gapped palindrome of $w[1..k]$ that is centered at $\frac{i+j}{2}$ and has a gap length in range $[g_{\min}, g_{\max}]$. We therefore update $i \leftarrow i + 1$, $j \leftarrow j + 1$, $l \leftarrow 1$, $k \leftarrow k + 1$ and proceed to the forthcoming character.
   (b) If $a_l^{out} + a_l^{in} \geq A$, $a_l^{in} \leq l \cdot A$, and $j + l \cdot A + a_l^{out} \leq k$, then $(i_l, j_{l'}, a_l)$ is a maximal LCGP centered at $\frac{i+j}{2}$ where $i_l = i - l \cdot A + a_l^{in}$, $j + l \cdot A + a_l^{out}$, and $a_l = a_l^{out} + a_l^{in}$. We output it and update $l \leftarrow l + 1 + \lfloor \frac{a_l^{out}}{A} \rfloor$ (this is to skip the subsequent sampled positions which are also contained in the same LCGP due to Lemma 1).
      i. If $l \leq \frac{d}{A}$, then we compute the outward and inward LCEs from the pair of sampled positions with l.
      ii. If $l > \frac{d}{A}$, then there is no inward-maximal suffix gapped palindrome of $w[1..k]$ that is centered at $\frac{i+j}{2}$ and has a gap length in range $[g_{\min}, g_{\max}]$. We therefore update $i \leftarrow i+1$, $j \leftarrow j+1$, $l \leftarrow 1$, $k \leftarrow k+1$ and proceed to the forthcoming character.

(c) If $a_l^{out}+a_l^{in} \geq A$, $a_l^{in} \leq l \cdot A$, and $j+l \cdot A+a_l^{out} = k+1$, then (i_l, j_l, a_l) is an inward-maximal gapped suffix palindrome of $w[1..k+1]$ with gap length in range $[g_{\min}, g_{\max}]$. Moreover, since we have processed l in increasing order, it is guaranteed that (i_l, j_l, a_l) is the longest such one. Hence, we proceed to the next character by updating $k \leftarrow k+1$.

Theorem 5. *For a growing string to which new characters are appended, we can compute all LCGPs in an online manner, in $O(n(m+\log\sigma))$ time and $O(n)$ space, where n is the length of the final string and $m = \max\{\frac{g_{\max}-g_{\min}}{A}, 1\}$.*

Proof. The correctness should be clear from the above arguments.

For each $k = 1, \ldots, n$, we consider a fixed center $\frac{i+j}{2}$ and compute all LCGPs with this center. We perform at most $\frac{2d}{A}$ LCE queries for each k, as there are $\frac{d}{A}$ sampled positions for each k. Since each LCE query can be answered in $O(1)$ time as in the proof of Theorem 3, the total time cost of the LCE queries for all $k = 1, \ldots, n$ is $O(\frac{d}{A}n) = O(mn)$. We use additional $O(n \log \sigma)$ time to maintain the suffix tree augmented with the dynamic LCA data structure for bidirectionally growing string $w'_k = (w[1..k])^R\#w[1..k]$. Thus the total time complexity is $O(n(m + \log\sigma))$.

The total space requirement is dominated by the suffix tree and the dynamic LCA data structure, and hence is $O(n)$. □

4.2 Optimality of our Algorithm

The following corollary is immediate from Theorem 5.

Corollary 1. *For constant parameters $g_{\min}$, $g_{\max}$, A and a constant-size alphabet, we can compute all maximal LCGPs in a string of length n in an online manner, in optimal $O(n)$ time and space.*

We can show that even for non-constant gap constraints $g_{\min}$ and $g_{\max}$, the running-time of our algorithm is optimal in the worst case. For any string w, let L_w denote the number of all maximal LCGPs in w w.r.t. given parameters $g_{\min}$, $g_{\max}$, and A. It immediately follows from Lemma 1 that L_w is upper-bounded by the total number of sampled positions in w. Hence $L_w = O(mn)$, where $n = |w|$ and $m = \max\{\frac{g_{\max}-g_{\min}}{A}, 1\}$. It is also true that there is an instance w for which $L_w = \Omega(mn)$ if A is a constant: For example, consider string $z = (\mathtt{abc})^{\frac{n}{3}}$. This string z contains maximal gapped palindromes of form $\mathtt{a}(\mathtt{bc}(\mathtt{abc})^p)\mathtt{a}$ with arm $\mathtt{a}$, $\mathtt{b}(\mathtt{c}(\mathtt{abc})^p\mathtt{a})\mathtt{b}$ with arm $\mathtt{b}$, and $\mathtt{c}((\mathtt{abc})^p\mathtt{ab})\mathtt{c}$ with arm $\mathtt{c}$ for all $0 \leq p \leq \frac{n}{3} - 2$. Thus, for $A = 1$ and for any $2 \leq g_{\min} \leq g_{\max}$, the string z contains $L_z = \Theta((g_{\max} - g_{\min})n) = \Theta(\frac{g_{\max}-g_{\min}}{A}n) = \Theta(mn)$ maximal LCGPs. Hence the running time $O(m(n + \log\sigma))$ of our algorithm is optimal in the worst case, for a constant-size alphabet.

5 Conclusions

In this paper, we presented an online algorithm which finds all maximal g-gapped palindromes occurring in a string w of length n in $O(n \log\sigma)$ time, where σ is the

alphabet size. We also showed that the above online algorithm can be extended to find more general length-constrained gapped palindromes (LCGPs) occurring in w in $O(n(\frac{g_{\min}-g_{\max}}{A} + \log\sigma))$ time, for given parameters $2 \leq g_{\min} \leq g_{\max}$ and $A \geq 1$. We also showed that if A is a constant, then there exists a string which contains $\Omega((g_{\min}-g_{\max})n)$ maximal LCGPs. This implies that for a constant-size alphabet the running time of our algorithm is optimal in the worst case.

To our knowledge, the proposed methods are the first online algorithms to find any kind of gapped palindromes in strings. Therefore, there remain many open problems. In particular, we are interested in the following:

- Is there a string of length n which contains $\Omega(\frac{g_{\min}-g_{\max}}{A}n)$ maximal LCGPs for *non-constant* A?
- Can we reduce the $n\frac{g_{\min}-g_{\max}}{A}$ factor to L_w in the $O(n(\frac{g_{\min}-g_{\max}}{A} + \log\sigma))$-time algorithm for finding all maximal LCGPs, thereby obtaining an optimal algorithm?
- Can the *maximal α-gapped palindromes* [5] of a given string be computed online efficiently?

References

1. Apostolico, A., Breslauer, D., Galil, Z.: Parallel detection of all palindromes in a string. Theor. Comput. Sci. **141**(1&2), 163–173 (1995)
2. Bender, M.A., Farach-Colton, M.: The LCA problem revisited. In: Gonnet, G.H., Viola, A. (eds.) LATIN 2000. LNCS, vol. 1776, pp. 88–94. Springer, Heidelberg (2000)
3. Cole, R., Hariharan, R.: Dynamic LCA queries on trees. SIAM J. Comput. **34**(4), 894–923 (2005)
4. Farach-Colton, M., Ferragina, P., Muthukrishnan., S.: On the sorting-complexity of suffix tree construction. J. ACM **47**(6), 987–1011 (2000)
5. Gawrychowski, P., Tomohiro, I., Inenaga, S., Köppl, D., Manea, F.: Efficiently finding all maximal α-gapped repeats. In: STACS 2016 (to appear, 2016). http://arxiv.org/abs/1509.09237
6. Gusfield, D.: Algorithms on Strings, Trees, and Sequences. Cambridge University Press, New York (1997)
7. Inenaga, S.: Bidirectional construction of suffix trees. Nord. J. Comput. **10**(1), 52 (2003)
8. Kolpakov, R., Kucherov, G.: Searching for gapped palindromes. Theor. Comput. Sci. **410**(51), 5365–5373 (2009)
9. Kosolobov, D., Rubinchik, M., Shur, A.M.: Finding distinct subpalindromes online. In: PSC 2013, pp. 63–69 (2013)
10. Manacher, G.K.: A new linear-time on-line algorithm for finding the smallest initial palindrome of a string. J. ACM **22**(3), 346–351 (1975)
11. Ukkonen, E.: On-line construction of suffix trees. Algorithmica **14**(3), 249–260 (1995)
12. Weiner, P.: Linear pattern matching algorithms. In: 14th Annual Symposium on Switching and Automata Theory, pp. 1–11 (1973)