Computing Longest Single-arm-gapped Palindromes in a String

Shintaro Narisada[1(✉)], Diptarama[1], Kazuyuki Narisawa[1], Shunsuke Inenaga[2], and Ayumi Shinohara[1]

[1] Graduate School of Information Sciences, Tohoku University, Sendai, Japan
{shintaro_narisada,diptarama}@shino.ecei.tohoku.ac.jp,
{narisawa,ayumi}@ecei.tohoku.ac.jp
[2] Department of Informatics, Kyushu University, Fukuoka, Japan
inenaga@inf.kyushu-u.ac.jp

Abstract. We introduce new types of approximate palindromes called *single-arm-gapped palindromes* (*SAGPs*). A SAGP contains a gap in either its left or right arm, which is in the form of either $wgucu^Rw^R$ or $wucu^Rgw^R$, where w and u are non-empty strings, w^R and u^R are their reversed strings respectively, g is a gap, and c is either a single character or the empty string. We classify SAGPs into two groups: those which have ucu^R as a maximal palindrome (type-1), and the others (type-2). We propose several algorithms to compute all type-1 SAGPs with longest arms occurring in a given string using suffix arrays, and them a linear-time algorithm based on suffix trees. We also show how to compute type-2 SAGPs with longest arms in linear time. We perform some preliminary experiments to evaluate practical performances of the proposed methods.

1 Introduction

A palindrome is a string that reads the same forward and backward. Discovering palindromic structures in strings is a classical, yet important task in combinatorics on words and string algorithmics (e.g., see [1,3,8,14]). A natural extension to palindromes is to allow for a *gap* between the left and right arms of palindromes. Namely, a string x is called a gapped palindrome if $x = wgw^R$ for some strings w, g with $|w| \geq 1$ and $|g| \geq 0$. Finding gapped palindromes has applications in bioinformatics, such as finding secondary structures of RNA sequences called *hairpins* [9]. If we further allow for another gap *inside either arm*, then such a palindrome can be written as $wg_2ug_1u^Rw^R$ or $wug_1u^Rg_2w^R$ for some strings w, g_1, g_2, u with $|u| \geq 1$, $|g_1| \geq 0$, $|g_2| \geq 0$, and $|w| \geq 1$. These types of palindromes characterize *hairpins with bulges* in RNA sequences, known to occur frequently in the secondary structures of RNA sequences [16]. Notice that the special case where $|g_1| \leq 1$ and $|g_2| = 0$ corresponds to usual palindromes, and the special case where $|g_1| \geq 2$ and $|g_2| = 0$ corresponds to gapped palindromes.

In this paper, we consider a new class of generalized palindromes where $|g_1| \leq 1$ and $|g_2| \geq 1$, i.e., palindromes with gaps *inside* one of its arms. We call such palindromes as *single-arm-gapped palindromes* (*SAGPs*). For instance,

B. Steffen et al. (Eds.): SOFSEM 2017, LNCS 10139, pp. 375–386, 2017.
DOI: 10.1007/978-3-319-51963-0_29

string abb|ca|cb|bc|bba is an SAGP of this kind, taking $w =$ abb, $g_1 = \varepsilon$ (the empty string), $g_2 =$ ca, and $u =$ cb.

We are interested in occurrences of SAGPs as substrings of a given string T. For simplicity, we will concentrate on SAGPs with $|g_1| = 0$ containing a gap in their left arms. However, slight modification of all the results proposed in this paper can easily be applied to all the other cases. For any occurrence of an SAGP $wguu^Rw^R$ beginning at position b in T, the position $b + |wgu| - 1$ is called the *pivot* of the occurrence of this SAGP. This paper proposes various algorithms to solve the problem of computing longest SAGPs for every pivot in a given string T of length n. We classify longest SAGPs into two groups: those which have uu^R as a maximal palindrome (*type-1*), and the others (*type-2*). Firstly, we show a naïve $O(n^2)$-time algorithm for computing type-1 longest SAGPs. Secondly, we present a simple but practical $O(n^2)$-time algorithm for computing type-1 longest SAGPs based on simple scans over the suffix array [15]. We also show that the running time of this algorithm can be improved by using a dynamic predecessor/successor data structure. If we employ the van Emde Boas tree [4], we achieve $O((n + occ_1) \log \log n)$-time solution, where occ_1 is the number of type-1 longest SAGPs to output. Finally, we present an $O(n + occ_1)$-time solution based on the suffix tree [17]. For type-2 longest SAGPs, we show an $O(n + occ_2)$-time algorithm, where occ_2 is the number of type-2 longest SAGPs to output. Combining the last two results, we obtain an optimal $O(n + occ)$-time algorithm for computing all longest SAGPs, where occ is the number of outputs. We performed preliminary experiments to compare practical performances of our algorithms for finding type-1 longest SAGPs. All proofs are omitted due to lack of space. A full version of this paper is available at arXiv:1609.03000.

Related work. For a *fixed* gap length d, one can find all gapped palindromes wgw^R with $|g| = d$ in the input string T of length n in $O(n)$ time [9]. Kolpakov and Kucherov [13] showed an $O(n + L)$-time algorithm to compute *long-armed palindromes* in T, which are gapped palindromes wgw^R such that $|w| \geq |g|$. Here, L denotes the number of outputs. They also showed how to compute, in $O(n + L)$ time, *length-constrained palindromes* which are gapped palindromes wgw^R such that the gap length $|g|$ is in a predefined range. Recently, Fujishige et al. [6] proposed online algorithms to compute long-armed palindromes and length-constrained palindromes from a given string. A gapped palindrome wgw^R is an *α-gapped palindrome*, if $|wg| \leq \alpha|w|$ for $\alpha \geq 1$. Gawrychowski et al. [7] showed that the maximum number of α-gapped palindromes occurring in a string of length n is at most $28\alpha n + 7n$. Since long-armed palindromes are 2-gapped palindromes for $\alpha = 2$, $L = O(n)$ and thus Kolpakov and Kucherov's algorithm runs in $O(n)$ time. Gawrychowski et al. [7] also proposed an $O(\alpha n)$-time algorithm to compute all α-gapped palindromes in a given string for any predefined $\alpha \geq 1$. Hsu et al. [10] showed an $O(kn)$ time algorithm that finds all maximal approximate palindromes uvw in a given string such that $|v| = q$ and the Levenshtein distance between u and w^R is at most k, for any $q \geq 0$ and $k > 0$. We emphasize that none of the above algorithms can directly be applied to computing SAGPs.

2 Preliminaries

Let $\Sigma = \{1, \ldots, \sigma\}$ be an integer alphabet of size σ. An element of Σ^* is called a *string*. For any string w, $|w|$ denotes the length of w. The empty string is denoted by ε. Let $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. For any $1 \leq i \leq |w|$, $w[i]$ denotes the i-th symbol of w. For a string $w = xyz$, strings x, y, and z are called a *prefix*, *substring*, and *suffix* of w, respectively. The substring of w that begins at position i and ends at position j is denoted by $w[i..j]$ for $1 \leq i \leq j \leq |w|$, i.e., $w[i..j] = w[i] \cdots w[j]$. For $j > i$, let $w[i..j] = \varepsilon$ for convenience. For two strings X and Y, let $lcp(X, Y)$ denote the length of the longest common prefix of X and Y.

For any string x, let x^R denote the reversed string of x, i.e. $x^R = x[|x|] \cdots x[1]$. A string p is called a *palindrome* if $p = p^R$. Let T be any string of length n. Let $p = T[b..e]$ be a palindromic substring of T. The position $i = \lfloor \frac{b+e}{2} \rfloor$ is called the *center* of this palindromic substring p. The palindromic substring p is said to be the *maximal palindrome* centered at i iff there are no longer palindromes than p centered at i, namely, $T[b-1] \neq T[e+1]$, $b = 1$, or $e = n$.

A string x is called a *single-arm-gapped palindrome* (SAGP) if x is in the form of either $wgucu^Rw^R$ or $wucu^Rgw^R$, with some non-empty strings $w, g, u \in \Sigma^+$ and $c \in \Sigma \cup \{\varepsilon\}$. For simplicity and ease of explanations, in what follows we consider only SAGPs whose left arms contain gaps and $c = \varepsilon$, namely, those of form $wguu^Rw^R$. But our algorithms to follow can easily be modified to compute other forms of SAGPs occurring in a string as well.

Let b be the beginning position of an occurrence of a SAGP $wguu^Rw^R$ in T, namely $T[b..b + 2|wu| + |g| - 1] = wguu^Rw^R$. The position $i = b + |wgu| - 1$ is called the *pivot* of this occurrence of the SAGP. This position i is also the center of the palindrome uu^R. An SAGP $wguu^Rw^R$ for pivot i in string T is represented by a quadruple $(i, |w|, |g|, |u|)$ of integers. In what follows, we will identify the quadruple $(i, |w|, |g|, |u|)$ with the corresponding SAGP $wguu^Rw^R$ for pivot i.

For any SAGP $x = wguu^Rw^R$, let $armlen(x)$ denote the length of the arm of x, namely, $armlen(x) = |wu|$. A substring SAGP $y = wguu^Rw^R$ for pivot i in a string T is said to be a *longest* SAGP for pivot i, if for any SAGP y' for pivot i in T, $armlen(y) \geq armlen(y')$.

Notice that there can be different choices of u and w for the longest SAGPs at the same pivot. For instance, consider string `ccabcabbace`. Then, $(7, 1, 3, 2)$ = `c`|`abc`|`ab`|`ba`|`c` and $(7, 2, 3, 1)$ = `ca`|`bca`|`b`|`b`|`ac` are both longest SAGPs (with arm length $|wu| = 3$) for the same pivot 7, where the underlines represent the gaps. Of all longest SAGPs for each pivot i, we regard those that have longest palindromes uu^R centered at i as *canonical* longest SAGPs for pivot i. In the above example, $(7, 1, 3, 2)$ = `c`|`abc`|`ab`|`ba`|`c` is a canonical longest SAGP for pivot 7, while $(7, 2, 3, 1)$ = `ca`|`bca`|`b`|`b`|`ac` is not. Let $SAGP(T)$ be the set of canonical longest SAGPs for all pivots in T. In this paper, we present several algorithms to compute $SAGP(T)$.

For an input string T of length n over an integer alphabet of size $\sigma = n^{O(1)}$, we perform standard preprocessing which replaces all characters in T with integers from range $[1, n]$. Namely, we radix sort the original characters in T, and

replace each original character by its rank in the sorted order. Since the original integer alphabet is of size $n^{O(1)}$, the radix sort can be implemented with $O(1)$ number of bucket sorts, taking $O(n)$ total time. Thus, whenever we speak of a string T over an integer alphabet of size $n^{O(1)}$, one can regard T as a string over an integer alphabet of size n.

Tools: Suppose a string T ends with a unique character that does not appear elsewhere in T. The *suffix tree* [17] of a string T, denoted by $STree(T)$, is a path-compressed trie which represents all suffixes of T. Then, $STree(T)$ can be defined as 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 T; (4) For each suffix s of T, there is a unique leaf such that the path from the root to the leaf spells out s. It follows from the above definition of $STree(T)$ that if $n = |T|$ then the number of nodes and edges in $STree(T)$ is $O(n)$. By representing every edge label X by a pair (i, j) of integers such that $X = T[i..j]$, $STree(T)$ can be represented with $O(n)$ space. For a given string T of length n over an integer alphabet of size $\sigma = n^{O(1)}$, $STree(T)$ can be constructed in $O(n)$ time [5]. For each node v in $STree(T)$, let $str(v)$ denote the string spelled out from the root to v. According to Property (4), we sometimes identify each position i in string T with the leaf which represents the corresponding suffix $T[i..n]$.

Suppose the unique character at the end of string T is the lexicographically smallest in Σ. The *suffix array* [15] of string T of length n, denoted SA_T, is an array of size n such that $SA_T[i] = j$ iff $T[j..n]$ is the ith lexicographically smallest suffix of T for $1 \leq i \leq n$. The *reversed suffix array* of T, denoted SA_T^{-1}, is an array of size n such that $SA_T^{-1}[SA_T[i]] = i$ for $1 \leq i \leq n$. The *longest common prefix array* of T, denoted LCP_T, is an array of size n such that $LCP_T[1] = -1$ and $LCP_T[i] = lcp(T[SA_T[i-1]..n], T[SA_T[i]..n])$ for $2 \leq i \leq n$. The arrays SA_T, SA_T^{-1}, and LCP_T for a given string T of length n over an integer alphabet of size $\sigma = n^{O(1)}$ can be constructed in $O(n)$ time [11,12].

For a rooted tree $\mathcal{T}$, the lowest common ancestor $LCA_{\mathcal{T}}(u, v)$ of two nodes u and v in $\mathcal{T}$ is the deepest node in $\mathcal{T}$ which has u and v as its descendants. It is known that after a linear-time preprocessing on the input tree, querying $LCA_{\mathcal{T}}(u, v)$ for any two nodes u, v can be answered in constant time [2].

Consider a rooted tree $\mathcal{T}$ where each node is either marked or unmarked. For any node v in $\mathcal{T}$, let $NMA_{\mathcal{T}}(v)$ denote the deepest marked ancestor of v. There exists a linear-space algorithm which marks any unmarked node and returns $NMA_{\mathcal{T}}(v)$ for any node v in amortized $O(1)$ time [18].

Let A be an integer array of size n. A range minimum query $RMQ_A(i, j)$ of a given pair (i, j) of indices $(1 \leq i \leq j \leq n)$ asks an index k in range $[i, j]$ which stores the minimum value in $A[i..j]$. After $O(n)$-time preprocessing on A, $RMQ_A(i, j)$ can be answered in $O(1)$ time for any given pair (i, j) of indices [2].

Let S be a set of m integers from universe $[1, n]$, where n fits in a single machine word. A predecessor (resp. successor) query for a given integer x to S asks the largest (resp. smallest) value in S that is smaller (resp. larger) than x. Let $u(m, n)$, $q(m, n)$ and $s(m, n)$ denote the time for updates (insertion/deletion)

of elements, the time for predecessor/successor queries, and the space of a dynamic predecessor/successor data structure. Using a standard balanced binary search tree, we have $u(m,n) = q(m,n) = O(\log m)$ time and $s(n,m) = O(m)$ space. The Y-fast trie [19] achieves $u(m,n) = q(m,n) = O(\log\log n)$ *expected* time and $s(n,m) = O(m)$ space, while the van Emde Boas tree [4] does $u(m,n) = q(m,n) = O(\log\log n)$ *worst-case* time and $s(n,m) = O(n)$ space.

3 Algorithms for Computing Canonical Longest SAGPs

In this section, we present several algorithms to compute the set $SAGP(T)$ of canonical longest SAGPs for all pivots in a given string T.

A position i in string T is said to be of *type-1* if there exists a SAGP $wguu^Rw^R$ such that uu^R is the maximal palindrome centered at position i, and is said to be of *type-2* otherwise. For example, consider T = `baaabaabaacbaabaabac` of length 20. Position 13 of T is of type-1, since there are canonical longest SAGPs $(13,4,4,2)$ = `abaa`|`baac`|`ba`|`ab`|`aaba` and $(13,4,1,2)$ = `abaa`|`c`|`ba`|`ab`|`aaba` for pivot 13, where `ba`|`ab` is the maximal palindrome centered at position 13. On the other hand, Position 6 of T is of type-2; the maximal palindrome centered at position 6 is `aaba`|`abaa` but there are no SAGPs in the form of w`g`w^R for pivot 6. The canonical longest SAGPs for pivot 6 is $(6,1,1,3)$ = `a`|`a`|`aba`|`aba`|`a`.

Let $Pos_1(T)$ and $Pos_2(T)$ be the sets of type-1 and type-2 positions in T, respectively. Let $SAGP(T,i)$ be the subset of $SAGP(T)$ whose elements are canonical longest SAGPs for pivot i. Let $SAGP_1(T) = \bigcup_{i\in Pos_1(T)} SAGP(T,i)$ and $SAGP_2(T) = \bigcup_{i\in Pos_2(T)} SAGP(T,i)$. Clearly $SAGP_1(T) \cup SAGP_2(T) = SAGP(T)$ and $SAGP_1(T) \cap SAGP_2(T) = \emptyset$. The following lemma gives an useful property to characterize the type-1 positions of T.

Lemma 1. *Let i be any type-1 position of a string T of length n. Then, a SAGP $wguu^Rw^R$ is a canonical longest SAGP for pivot i iff uu^R is the maximal palindrome centered at i and w^R is the longest non-empty prefix of $T[i+|u^R|+1..n]$ such that w occurs at least once in $T[1..i-|u|-1]$.*

We define two arrays *Pals* and *LMost* as follows:

$$Pals[i] = \{r \mid T[i-r+1..i+r] \text{ is a maximal palindrome in } T \text{ for pivot } i\}.$$
$$LMost[c] = \min\{i \mid T[i] = c\} \text{ for } c \in \Sigma.$$

By Lemma 1, a position i in T is of type-1 iff $LMost[i+Pals[i]+1] < i - Pals[i]$.

Lemma 2. *Given a string T of length n over an integer alphabet of size $n^{O(1)}$, we can determine whether each position i of T is of type-1 or type-2 in a total of $O(n)$ time and space.*

By Lemmas 1 and 2, we can consider an algorithm to compute $SAGP(T)$ by computing $SAGP_1(T)$ and $SAGP_2(T)$ separately, as shown in Algorithm 1.

Algorithm 1. computing $SAGP(T)$

```
   Input: string T of length n
   Output: SAGP(T)
 1 compute Pals;
 2 for i = n downto 1 do
 3 |  c = T[i]; NextPos[i] = LMost[c]; LMost[c] = i;
 4 for i = 1 to n do
 5 |  if LMost[i + Pals[i] + 1] < i − Pals[i] then
 6 |  |  Pos_1(T) = Pos_1(T) ∪ {i};      /* position i is of type-1 */
 7 |  else
 8 |  |  Pos_2(T) = Pos_2(T) ∪ {i};      /* position i is of type-2 */
 9 compute SAGP_1(T);            /* Sect. 3.1 */
10 compute SAGP_2(T);            /* Sect. 3.2 */
```

In this algorithm, we also construct an auxiliary array *NextPos* defined by $NextPos[i] = \min\{j \mid i < j,\ T[i] = T[j]\}$ for each $1 \leq i \leq n$, which will be used in Sect. 3.2.

Lemma 3. *Algorithm 1 correctly computes* $SAGP(T)$.

In the following subsections, we present algorithms to compute $SAGP_1(T)$ and $SAGP_2(T)$ respectively, assuming that the arrays *Pals*, *LMost* and *NextPos* have already been computed.

3.1 Computing $SAGP_1(T)$ for Type-1 Positions

In what follows, we present several algorithms corresponding to the line 9 in Algorithm 1. Lemma 1 allows us greedy strategies to compute the longest prefix w^R of $T[i + Pals[i] + 1..n]$ such that w occurs in $T[1..i - Pals[i] - 1]$.

Naïve Quadratic-Time Algorithm with RMQs. Let $T' = T\$T^R\#$. We construct the suffix array $SA_{T'}$, the reversed suffix array $SA^{-1}_{T'}$, the LCP array $LCP_{T'}$ for T', and the array $RMQ_{LCP_{T'}}$ to support RMQ on $LCP_{T'}$.

For each $Pals[i]$ in T, for each gap size $G = 1, \ldots, i - Pals[i] - 1$, we compute $W = lcp(T[1..i - Pals[i] - G]^R, T[i + Pals[i] + 1..n])$ in $O(1)$ time by $RMQ_{LCP_{T'}}$. Then, the gap sizes G with largest values of W give all longest SAGPs for pivot i. Since we test $O(n)$ gap sizes for every pivot i, it takes a total of $O(n^2)$ time to compute $SAGP_1(T)$. The working space of this method is $O(n)$.

Simple Quadratic-Time Algorithm Based on Suffix Array. Given a string T, we construct $SA_{T'}$, $SA^{-1}_{T'}$, and $LCP_{T'}$ for string $T' = T\$T^R\#$ as in the previous subsection. Further, for each position $n + 2 \leq j \leq 2n + 1$ in the reversed part T^R of $T' = T\$T^R\#$, let $op(j)$ denote its "original" position in the

string T, namely, let $op(j) = 2n - j + 2$. Let e be any entry of $SA_{T'}$ such that $SA_{T'}[e] \geq n + 2$. We associate each such entry of $SA_{T'}$ with $op(SA_{T'}[e])$.

Let $SA_{T'}[k] = i + Pals[i] + 1$, namely, the kth entry of $SA_{T'}$ corresponds to the suffix $T[i + Pals[i] + 1..n]$ of T. Now, the task is to find the longest prefix w^R of $T[i + Pals[i] + 1..n]$ such that w occurs completely inside $T[1..i - Pals[i] - 1]$. Let $b = i - Pals[i] + 1$, namely, b is the beginning position of the maximal palindrome uu^R centered at i. We can find w for any maximal SAGP $wguu^Rw^R$ for pivot i by traversing $SA_{T'}$ from the kth entry forward and backward, until we encounter the nearest entries $p < k$ and $q > k$ on $SA_{T'}$ such that $op(SA_{T'}[p]) < b - 1$ and $op(SA_{T'}[q]) < b - 1$, if they exist. The size W of w is equal to

$$\max\{\min\{LCP_{T'}[p+1], \ldots, LCP_{T'}[k]\}, \min\{LCP_{T'}[k+1], \ldots, LCP_{T'}[q]\}\}. \quad (1)$$

Assume w.l.o.g. that p gives a larger lcp value with k, i.e. $W = \min\{LCP_{T'}[p + 1], \ldots, LCP_{T'}[k]\}$. Let s be the largest entry of $SA_{T'}$ such that $s < p$ and $LCP_{T'}[s + 1] < W$. Then, any entry t of $SA_{T'}$ such that $s < t \leq p + 1$ and $op(SA_{T'}[t]) < b - 1$ corresponds to an occurrence of a longest SAGP for pivot i, with gap size $b - op(SA_{T'}[t]) - 1$. We output longest SAGP $(i, W, b - op(SA_{T'}[t]) - 1, |u|)$ for each such t. The case where q gives a larger lcp value with k, or p and q give the same lcp values with k can be treated similarly.

We find p and s by simply traversing $SA_{T'}$ from k. Since the distance from k to s is $O(n)$, the above algorithm takes $O(n^2)$ time. The working space is $O(n)$.

Algorithm Based on Suffix Array and Predecessor/Successor Queries. Let $occ_1 = |SAGP_1(T)|$. For any position r in T, we say that the entry j of $SA_{T'}$ is *active* w.r.t. r iff $op(SA_{T'}[j]) < r - 1$. Let $Active(r)$ denote the set of active entries of $SA_{T'}$ for position r, namely, $Active(r) = \{j \mid op(SA_{T'}[j]) < r - 1\}$.

Let $t_1 = p$, and let $t_2, \ldots, t_h$ be the decreasing sequence of entries of $SA_{T'}$ which correspond to the occurrences of longest SAGPs for pivot i. Notice that for all $1 \leq \ell \leq h$ we have $op(SA_{T'}[t_\ell]) < b - 1$ and hence $t_\ell \in Active(b)$, where $b = i - |u| + 1$. Then, finding t_1 reduces to a predecessor query for k in $Active(b)$. Also, finding t_ℓ for $2 \leq \ell \leq h$ reduces to a predecessor query for $t_{\ell-1}$ in $Active(b)$.

To effectively use the above observation, we compute an array U of size n from $Pals$ such that $U[b]$ stores a list of all maximal palindromes in T which begin at position b if they exist, and $U[b]$ is nil otherwise. U can be computed in $O(n)$ time e.g., by bucket sort. After computing U, we process $b = 1, \ldots, n$ in increasing order. Assume that when we process a certain value of b, we have maintained a dynamic predecessor/successor query data structure for $Active(b)$. The key is that the same set $Active(b)$ can be used to compute the longest SAGPs for every element in $U[b]$, and hence we can use the same predecessor/successor data structure for all of them. After processing all elements in $U[b]$, we insert all elements of $Active(b+1) \setminus Active(b)$ to the predecessor/successor data structure. Each element to insert can be easily found in constant time.

Since we perform $O(n + occ_1)$ predecessor/successor queries and $O(n)$ insertion operations in total, we obtain the following theorem.

Theorem 1. *Given a string T of size n over an integer alphabet of size $\sigma = n^{O(1)}$, we can compute $SAGP_1(T)$ in $O(n(u(n,n)+q(n,n))+occ_1 \cdot q(n,n))$ time with $O(n + s(n,n))$ space by using the suffix array and a predecessor/successor data structure, where $occ_1 = |SAGP_1(T)|$.*

Since every element of $Active(b)$ for any b is in range $[1, 2n + 2]$, we can employ the van Emde Boas tree [4] as the dynamic predecessor/successor data structure using $O(n)$ total space. Thus we obtain the following theorem.

Theorem 2. *Given a string T of size n over an integer alphabet of size $\sigma = n^{O(1)}$, we can compute $SAGP_1(T)$ in $O((n+occ_1)\log\log n)$ time and $O(n)$ space by using the suffix array and the van Emde Boas tree, where $occ_1 = |SAGP_1(T)|$.*

Optimal-Time Algorithm Based on Suffix Tree. In this subsection, we show that the problem can be solved in *optimal* time and space, using the following three suffix trees regarding the input string T. Let $\mathcal{T}_1 = STree(T\$T^R\#)$ for string $T\$T^R\#$ of length $2n + 2$, and $\mathcal{T}_2 = STree(T^R\#)$ of length $n + 1$. These suffix trees $\mathcal{T}_1$ and $\mathcal{T}_2$ are static, and thus can be constructed offline, in $O(n)$ time for an integer alphabet. We also maintain a growing suffix tree $\mathcal{T}_2' = STree(T^R[k..n])\#)$ for decreasing $k = n, \ldots, 1$.

Lemma 4. *Given $\mathcal{T}_2 = STree(T^R\#)$, we can maintain $\mathcal{T}_2' = STree(T^R[k..n]\#)$ for decreasing $k = n, \ldots, 1$ incrementally, in $O(n)$ total time for an integer alphabet of size $n^{O(1)}$.*

Theorem 3. *Given a string T of length n over an integer alphabet of size $\sigma = n^{O(1)}$, we can compute $SAGP_1(T)$ in optimal $O(n + occ_1)$ time and $O(n)$ space by using suffix trees, where $occ_1 = |SAGP_1(T)|$.*

Proof. We first compute the array U. Consider an arbitrary fixed b, and let uu^R be a maximal palindrome stored in $U[b]$ whose center is $i = b + |u| - 1$. Assume that we have a growing suffix tree $\mathcal{T}_2'$ for string $T^R[n-b+1..n]\#$ which corresponds to the prefix $T[1..b]$ of T of size b. We use a similar strategy as the suffix array based algorithms. For each position $2n - b + 2 \leq j \leq 2n + 1$ in string $T' = T\$T^R\#$, $1 \leq op(j) \leq b - 2$. We maintain the NMA data structure over the suffix tree $\mathcal{T}_1$ for string T' so that all the ancestors of the leaves whose corresponding suffixes start at positions $2n - b + 2 \leq j \leq 2n + 1$ are marked, and any other nodes in $\mathcal{T}_1$ remain unmarked at this step.

As in the suffix-array based algorithms, the task is to find the longest prefix w^R of $T[i + |u^R| + 1..n]$ such that w occurs completely inside $T[1..b - 2] = T[1..i-|u|-1]$. In so doing, we perform an NMA query from the leaf $i+|u^R|+1$ of $\mathcal{T}_1$, and let v be the answer to the NMA query. By the way how we have maintained the NMA data structure, it follows that $str(v) = w^R$.

To obtain the occurrences of w in $T[1..b-2]$, we switch to $\mathcal{T}_2'$, and traverse the subtree rooted at v. Then, for any leaf ℓ in the subtree, $(i, |str(v)|, b - op(\ell), |u|)$ is a canonical longest SAGP for pivot i.

After processing all the maximal palindromes in $U[b]$, we mark all unmarked ancestors of the leaf $2n - b$ of $\mathcal{T}_1$ in a bottom-up manner, until we encounter the lowest ancestor that is already marked. This operation is a preprocessing for the maximal palindromes in $U[b+1]$, as we will be interested in the positions between 1 and $op(2n-b) = b-1$ in T. In this preprocessing, each unmarked node is marked at most once, and each marked node will remain marked. In addition, we update the growing suffix tree $\mathcal{T}_2'$ by inserting the new leaf for $T^R[n-b..n]\#$.

We analyze the time complexity of this algorithm. Since all maximal palindromes in $U[b]$ begin at position b in T, we can use the same set of marked nodes on $\mathcal{T}_1$ for all of those in $U[b]$. Thus, the total cost to update the NMA data structure for all b's is linear in the number of unmarked nodes that later become marked, which is $O(n)$ overall. The cost for traversing the subtree of $\mathcal{T}_2'$ to find the occurrences of w can be charged to the number of canonical longest SAGPs to output for each pivot, thus it takes $O(occ_1)$ time for all pivots. Updating the growing suffix tree $\mathcal{T}_2'$ takes overall $O(n)$ time by Lemma 4. What remains is how to efficiently link the new internal node introduced in the growing suffix tree $\mathcal{T}_2'$, to its corresponding node in the static suffix tree $\mathcal{T}_1$ for string T'. This can be done in $O(1)$ time using a similar technique based on LCA queries on $\mathcal{T}_1$, as in the proof of Lemma 4. Summing up all the above costs, we obtain $O(n + occ_1)$ optimal running time and $O(n)$ working space. □

3.2 Computing $SAGP_2(T)$ for Type-2 Positions

In this subsection, we present an algorithm to compute $SAGP_2(T)$ in a given string T, corresponding to the line 10 in Algorithm 1.

Lemma 5. *For any type-2 position i in string T, every (not necessarily longest) SAGP for i must end at one of the positions between $i+2$ and $i + Pals[i]$.*

Lemma 6. *For any type-2 position i in string T, if $wguu^Rw^R$ is a canonical longest SAGP for pivot i, then $|w| = 1$.*

For every type-2 position i in T, let $u = T[i..i + Pals[i]]$. By Lemma 6, any canonical longest SAGP is of the form $cguu^Rc$ for $c \in \Sigma$. For each $2 \le k \le Pals[i]$, let $c_k = u^R[k]$, and let u_k^R be the proper prefix of u^R of length $k-1$. Now, observe that the largest value of k for which $LMost[c_k] \le i - |u_k| - 1$ corresponds to a canonical longest SAGP for pivot i, namely, $c_k g_k u_k u_k^R c_k$ is a canonical longest SAGP for pivot i, where $g_k = T[LMost[c_k] + 1..i - |u_k|]$. In order to efficiently find the largest value of such, we consider a function $findR(t, i)$ defined by

$$findR(t,i) = \min\{r \mid t \le r < i \text{ and } T[l] = T[r] \text{ for some } 1 \le l < r\} \cup \{+\infty\}.$$

Lemma 7. *For any type-2 position i in T, quadruple $(i, 1, r - LMost[T[r]], i - r)$ represents a canonical longest SAGP for pivot i, where $r = findR(i - Pals[i] + 1, i) \neq \infty$. Moreover, its gap is the longest among all the canonical longest SAGPs for pivot i.*

Algorithm 2. constructing the array *FindR*

Input: string T of length n
Output: array *FindR* of size n
Let Occ_1 and Occ_2 be arrays of size Σ_T initialized by $+\infty$;
Let *FindR* be an arrays of size n, and let *Stack* be an empty stack;
$min_{in} = +\infty$;
for $i = n$ **downto** 1 **do**
  $c = T[i]$; $Occ_2[c] = Occ_1[c]$; $Occ_1[c] = i$;
  $min_{in} = \min\{min_{in}, Occ_2[c]\}$;
  $Stack.push(i)$;
  while *Stack is not empty* ***and*** $LMost[T[Stack.top]] \geq i$ **do** $Stack.pop()$;
  $min_{out} = Stack.top$ **if** *Stack* is not empty **else** $+\infty$;
  $FindR[i] = \min\{min_{in}, min_{out}\}$

By Lemma 7, we can compute a canonical longest SAGP for any type-2 pivot i in $O(1)$ time, assuming that the function $findR(t, i)$ returns a value in $O(1)$ time. We define an array *FindR* of size n by

$$FindR[t] = \min\{r \mid t \leq r \text{ and } T[l] = T[r] \text{ for some } 1 \leq l < r\} \cup \{+\infty\}, \quad (2)$$

for $1 \leq t \leq n$. If the array *FindR* has already been computed, then $findR(t, i)$ can be obtained in $O(1)$ time by $findR(t, i) = FindR[t]$ if $FindR[t] < i$, and $+\infty$ otherwise. Algorithm 2 shows a pseudo-code to compute *FindR*.

Lemma 8. *Algorithm 2 correctly computes FindR in $O(n)$ time and space.*

By Lemma 8, we can compute $SAGP_2(T)$ for type-2 positions as follows.

Theorem 4. *Given a string T of length n over an integer alphabet of size $n^{O(1)}$, we can compute $SAGP_2(T)$ in $O(n + occ_2)$ time and $O(n)$ space, where $occ_2 = |SAGP_2(T)|$.*

Proof. For a given T, we first compute the array *FindR* by Algorithm 2. By Lemma 7, we can get a canonical longest SAGP $x_1 = (i, 1, |g_1|, Pals[i])$ if any, in $O(1)$ time by referring to *LMost* and *FindR*. Note that x_1 is the one whose gap $|g_1|$ is the longest. Let $b_1 = i - Pals[i] - |g_1|$ be the beginning position of x_1 in T. Then the next shorter canonical longest SAGP for the same pivot i begins at position $b_2 = NextPos[b_1]$. By repeating this process $b_{j+1} = NextPos[b_j]$ while the gap size $|g_j| = i - Pals[i] - |b_j|$ is positive, we obtain all the canonical longest SAGPs for pivot i. Overall, we can compute all canonical longest SAGPs for all pivots in T in $O(n + occ_2)$ time. The space requirement is clearly $O(n)$. □

We now have the main theorem from Theorems 3, 4 and Lemmas 2, 3 as follows.

Theorem 5. *Given a string T of length n over an integer alphabet of size $n^{O(1)}$, Algorithm 1 can compute $SAGP(T)$ in optimal $O(n + occ)$ time and $O(n)$ space, where $occ = |SAGP(T)|$.*

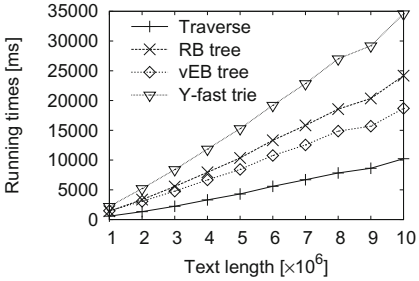

Fig. 1. Running times on random strings of length from 10^6 to 10^7 with $|\Sigma| = 10$.

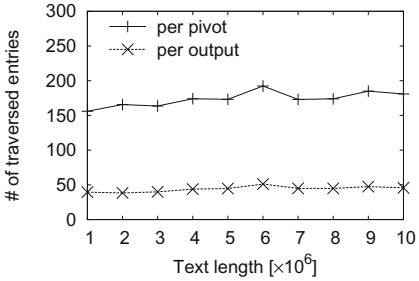

Fig. 2. Average numbers of traversed entries of suffix array per pivot/output.

4 Experiments

In this section, we show some experimental results which compare performance of our algorithms for computing $SAGP_1(T)$. We implemented the naïve quadratic-time algorithm (Naïve), the simple quadratic-time algorithm which traverses suffix arrays (Traverse), and three versions of the algorithm based on suffix array and predecessor/successor data structure, each employing red-black trees (RB tree), Y-fast tries (Y-fast trie), and van Emde Boas trees[1] (vEB tree), as the predecessor/successor data structure. We implemented all these algorithms with Visual C++ 12.0 (2013), and performed all experiments on a PC (Xeon W3565, 12 GB of memory) running on Windows 7.

We tested these programs on randomly generated strings of lengths from 10^6 to 10^7 with $|\Sigma| = 10$. Figure 1 shows the average running times of 10 executions, where Naïve is exculded because it was too slow. As one can see, Traverse was the fastest for all lengths. We also conducted the same experiments varying alphabet sizes as 2, 4, and 20, and obtained similar results as the case of alphabet size 10.

To verify why Traverse runs fastest, we measured the average numbers of suffix array entries which are traversed, per pivot and output (i.e., canonical longest SAGP). Figure 2 shows the result. We can observe that although in theory $O(n)$ entries can be traversed per pivot and output for a string of length n, in both cases the actual number is far less than $O(n)$ and grows very slowly as n increases. This seems to be the main reason why Traverse is faster than RB tree, vEB tree, and Y-fast trie which use sophisticated but also complicated predecessor/successor data structures.

We also tested Traverse, RB tree, vEB tree, and Y-fast trie on a genome of *M.tuberculosis* H37Rv (size: 4411529 bp, GenBank accession: NC_000962). The running times were Traverse: 4304.8, RB tree: 12126.7, vEB tree: 9729.8, Y-fast trie: 17862.6, all in milli-seconds. Here again, Traverse was the fastest.

[1] We modified a van Emde Boas tree implementation from https://code.google.com/archive/p/libveb/ so that it works with Visual C++.

Acknowledgements. This work was funded by ImPACT Program of Council for Science, Technology and Innovation (Cabinet Office, Government of Japan), Tohoku University Division for Interdisciplinary Advance Research and Education, and JSPS KAKENHI Grant Numbers JP15H05706, JP24106010, JP26280003.

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). doi:10.1007/10719839_9
3. Droubay, X., Pirillo, G.: Palindromes and sturmian words. Theor. Comput. Sci. **223**(1–2), 73–85 (1999)
4. van Emde Boas, P.: Preserving order in a forest in less than logarithmic time. In: FOCS, pp. 75–84 (1975)
5. Farach-Colton, M., Ferragina, P., Muthukrishnan, S.: On the sorting-complexity of suffix tree construction. J. ACM **47**(6), 987–1011 (2000)
6. Fujishige, Y., Nakamura, M., Inenaga, S., Bannai, H., Takeda, M.: Finding gapped palindromes online. In: Mäkinen, V., Puglisi, S.J., Salmela, L. (eds.) IWOCA 2016. LNCS, vol. 9843, pp. 191–202. Springer, Heidelberg (2016). doi:10.1007/978-3-319-44543-4_15
7. Gawrychowski, P., Tomohiro, I., Inenaga, S., Köppl, D., Manea, F.: Efficiently finding all maximal α-gapped repeats. In: STACS 2016, pp. 39:1–39:14 (2016)
8. Glen, A., Justin, J., Widmer, S., Zamboni, L.Q.: Palindromic richness. Eur. J. Comb. **30**(2), 510–531 (2009)
9. Gusfield, D.: Algorithms on Strings, Trees, and Sequences. Cambridge University Press, New York (1997)
10. Hsu, P.H., Chen, K.Y., Chao, K.M.: Finding all approximate gapped palindromes. Int. J. Found. Comput. Sci. **21**(6), 925–939 (2010)
11. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. J. ACM **53**(6), 918–936 (2006)
12. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Amir, A. (ed.) CPM 2001. LNCS, vol. 2089, pp. 181–192. Springer, Heidelberg (2001). doi:10.1007/3-540-48194-X_17
13. Kolpakov, R., Kucherov, G.: Searching for gapped palindromes. Theor. Comput. Sci. **410**(51), 5365–5373 (2009)
14. 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)
15. Manber, U., Myers, E.W.: Suffix arrays: a new method for on-line string searches. SIAM J. Comput. **22**(5), 935–948 (1993)
16. Shi, Y.Z., Wang, F.H., Wu, Y.Y., Tan, Z.J.: A coarse-grained model with implicit salt for RNAs: predicting 3D structure, stability and salt effect. J. Chem. Phys. **141**(10), 105102 (2014)
17. Weiner, P.: Linear pattern matching algorithms. In: 14th Annual Symposium on Switching and Automata Theory, pp. 1–11 (1973)
18. Westbrook, J.: Fast incremental planarity testing. In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 342–353. Springer, Heidelberg (1992). doi:10.1007/3-540-55719-9_86
19. Willard, D.E.: Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. Information Processing Letters **17**, 81–84 (1983)