Maximal Closed Substrings*

Golnaz Badkobeh[1][0000−0001−5550−7149], Alessandro De Luca[2][0000−0003−1704−773X], Gabriele Fici[3][0000−0002−3536−327X], and Simon J. Puglisi[4][0000−0001−7668−7636]

[1] Department of Computing, Goldsmiths University of London, United Kingdom
g.badkobeh@gold.ac.uk
[2] DIETI, Università di Napoli Federico II, Italy
alessandro.deluca@unina.it
[3] Dipartimento di Matematica e Informatica, Università di Palermo, Italy
gabriele.fici@unipa.it
[4] Department of Computer Science, University of Helisnki, Finland
simon.puglisi@helsinki.fi

Abstract. A string is closed if it has length 1 or has a nonempty border without internal occurrences. In this paper we introduce the definition of a *maximal closed substring* (MCS), which is an occurrence of a closed substring that cannot be extended to the left nor to the right into a longer closed substring. MCSs with exponent at least 2 are commonly called *runs*; those with exponent smaller than 2, instead, are particular cases of *maximal gapped repeats*. We show that a string of length n contains $\mathcal{O}(n^{1.5})$ MCSs. We also provide an output-sensitive algorithm that, given a string of length n over a constant-size alphabet, locates all m MCSs the string contains in $\mathcal{O}(n \log n + m)$ time.

Keywords: Closed Word · Maximal Closed Substring · Run.

1 Introduction

The distinction between open and closed strings was introduced by the third author in [8] in the context of Sturmian words.

A string is *closed* (or *periodic-like* [6]) if it has length 1 or it has a border that does not have internal occurrences (i.e., it occurs only as a prefix and as a suffix). Otherwise the string is *open*. For example, the strings a, $abaab$ and $ababa$ are closed, while ab and $ababaab$ are open. In particular, every string whose exponent — the ratio between the length and the minimal period — is at least 2, is closed [1].

In this paper, we consider occurrences of closed substrings in a string with the property that the substring cannot be extended to the left nor to the right into another closed substring. These are called the *maximal closed substrings* (MCS) of the string. For example, if $S = abaabab$, then the set of pairs of starting and

* Supported by organization x.

ending positions of the MCSs of S is

$$\{(1,1),(1,3),(1,6),(2,2),(3,4),(4,8),(5,5),(6,6),(7,7),(8,8)\}$$

This notion encompasses that of a *run* (maximal repetition) which is a MCS with exponent 2 or larger. It has been conjectured by Kolpakov and Kucherov [12] and then finally proved, after a long series of papers, by Bannai et al. [2], that a string of length n contains less than n runs.

On the other hand, maximal closed substrings with exponent smaller than 2 are particular cases of *maximal gapped repeats* [11]. An α-gapped repeat ($\alpha \geq 1$) in a string S is a substring uvu of S such that $|uv| \leq \alpha|u|$. It is maximal if the two occurrences of u in it cannot be extended simultaneously with the same letter to the right nor to the left. Gawrychowski et al. [10] proved that there are words that have $\Theta(\alpha n)$ maximal α-gapped repeats.

In this paper, we address the following problems:

1. How many MCSs can a string of length n contain?
2. What is the running time of an algorithm that, given a string S of length n, returns all the occurrences of MCSs in S?

We show that:

1. A string of length n contains $\mathcal{O}(n^{1.5})$ MCSs.
2. There is an algorithm that, given a string of length n over a constant-size alphabet, locates all m MCSs the string contains in $\mathcal{O}(n \log n + m)$ time.

2 Preliminaries

Let $S = S[1..n] = S[1]S[2]\cdots S[n]$ be a string of n letters drawn from an alphabet Σ of constant size. The length n of a string S is denoted $|S|$. The *empty string* has length 0. A *prefix* (resp. a *suffix*) of S is any string of the form $S[1..i]$ (resp. $S[i..n]$) for some $1 \leq i \leq n$. A *substring* of S is any string of the form $S[i..j]$ for some $1 \leq i \leq j \leq n$. It is also commonly assumed that the empty string is a prefix, a suffix and a substring of any string.

An integer $p \geq 1$ is a *period* of S if $S[i] = S[j]$ whenever $i \equiv j \pmod p$. For example, the periods of $S = aabaaba$ are 3, 6 and every $n \geq 7 = |S|$.

We recall the following classical result:

Lemma 1 (Periodicity Lemma (weak version) [9]). *If a string S has periods p and q such that $p + q \leq |S|$, then $\gcd(p,q)$ is also a period of S.*

Given a string S, we say that a string $\beta \neq S$ is a *border* of S if β is both a prefix and a suffix of S (we exclude the case $\beta = S$ but we do consider the case $|\beta| = 0$). Note that if β is a border of S, then $|S| - |\beta|$ is a period of S; conversely, if $p \leq |S|$ is a period of S, then S has a border of length $|S| - p$.

The following well-known property of borders holds:

Property 1. If a string has two borders β and β', with $|\beta| < |\beta'|$, then β is a border of β'.

The *border array* $\mathsf{B}_S[1..n]$ of string $S = S[1..n]$ is the integer array where $\mathsf{B}_S[i]$ is the length of the longest border of $S[1..i]$. When the string S is clear from the context, we will simply write B instead of B_S.

For any $1 \le i \le n$, let $\mathsf{B}^1[i] = \mathsf{B}[i]$ and $\mathsf{B}^j[i] = \mathsf{B}[\mathsf{B}^{j-1}[i]]$ for $j \ge 2$. We set

$$\mathsf{B}^+[i] = \{|\beta| \mid \beta \text{ is a border of } S[1..i]\}.$$

By Property 1, we have $\mathsf{B}^+[i] = \{\mathsf{B}^j[i] \mid j \ge 1\}$.

For example, in the string $S = aabaaaabaaba$, we have $\mathsf{B}^+[6] = \{0, 1, 2\}$. Indeed, $\mathsf{B}[6] = 2$, and $\mathsf{B}^2[6] = \mathsf{B}[2] = 1$, while $\mathsf{B}^j[6] = 0$ for $j > 2$.

The *OC array* [5] $\mathsf{OC}_S[1..n]$ of string S is a binary array where $\mathsf{OC}_S[i] = 1$ if $S[1,i]$ is closed and $\mathsf{OC}_S[i] = 0$ otherwise. We also define the array P_S where $\mathsf{P}_S[i]$ is the length of the longest repeated prefix of $S[1..i]$, that is, the longest prefix of $S[1..i]$ that has at least two occurrences in $S[1..i]$. Again, if S is clear from the context, we omit the subscripts.

Let S be a string of length n. Since for every $1 \le i \le n$, the longest repeated prefix v_i of $S[1..i]$ is the longest border of $S[1..j]$, where $j \le i$ is the ending position of the second occurrence of v_i, we have that

$$\mathsf{P}[i] = \max_{1 \le j \le i} \mathsf{B}[j]. \tag{1}$$

Lemma 2 ([7]). *Let S be a string of length n. For every $1 \le i \le n$, one has*

$$\mathsf{P}[i] = \sum_{j=1}^{i} \mathsf{OC}[j] - 1, \tag{2}$$

that is, $\mathsf{P}[i]$ *is the* rank *of* 1*'s in* $\mathsf{OC}[1..i]$ *minus one.*

Proof. For every repeated prefix v of S, the second occurrence of v in S determines a closed prefix of S; conversely, every closed prefix of S of length greater than 1 ends where the second occurrence of a repeated prefix of S ends. Indeed, the length of the longest repeated prefix increases precisely in those positions in which we have a closed prefix. That is, $\mathsf{P}[i] = \mathsf{P}[i-1] + \mathsf{OC}[i]$, for any $1 < i \le n$, which, together with $\mathsf{P}[1] = 0 = \mathsf{OC}[1] - 1$, yields (2).

As a consequence of (1) and (2), if two strings have the same border array, then they have the same OC array, but the converse is not true in general (take for example *aaba* and *aabb*).

The OC array of a string can be obtained from its P array by taking the differences of consecutive values, putting 1 in the first position (cf. [8]). Since the border array can be easily computed in linear time [13], it is possible to compute the OC array in linear time.

Example 1. The OC, B, and P arrays for $S = aabaaaabaaba$ are shown in the following table:

i	1	2	3	4	5	6	7	8	9	10	11	12
S	a	a	b	a	a	a	a	b	a	a	b	a
OC	1	1	0	0	1	0	0	1	1	1	0	0
B	0	1	0	1	2	2	2	3	4	5	3	4
P	0	1	1	1	2	2	2	3	4	5	5	5

3 A bound on the number of MCS

The goal of this section is to prove our bound $\mathcal{O}(n^{1.5})$ in the number of MCSs in a string of length n. This will be derived from a bound on the number of runs in the OC array.

In the next lemmas, we gather some structural results on the OC array.

Lemma 3 ([7, Remark 8]). *If* $\mathsf{OC}[i] = 1$*, then* $\mathsf{B}[i] = \mathsf{P}[i]$*, and* $\mathsf{B}[i-1] = \mathsf{P}[i-1]$ *(provided* $i > 1$*).*

Lemma 4. *For all* i *and* k *such that* $\mathsf{OC}[i+1..i+k+1] = 0^k1$*, if* $\mathsf{P}[i] \geq k$ *then* $\mathsf{P}[i] - k \in \mathsf{B}^+[i]$.

Proof. By Lemma 2 and Lemma 3, $\mathsf{P}[i+k+1] = \mathsf{P}[i] + 1$ is the length of the longest border of S at position $i+k+1$. The assertion is then a consequence of the following simple observation: Let u, v and x be strings; if ux is a border of vx, then u is a border of v. In fact, letting $v = S[1..i]$, and $x = S[i+1..i+k+1]$, as $\mathsf{B}^+[i+k+1] > k$, the longest border of vx can be written as ux for some u of length $\mathsf{P}[i] + 1 - k - 1 = \mathsf{P}[i] - k$.

Lemma 5. *For all* i *and* k *such that* $\mathsf{OC}[i..i+k+1] = 10^k1$*, if* $\mathsf{P}[i] \geq k$ *then* $\mathsf{P}[i] - k \in \mathsf{B}^+[\mathsf{P}[i]]$.

Proof. Immediate by Lemmas 3 and 4, as $\mathsf{B}[i] = \mathsf{P}[i]$ and $\mathsf{P}[i] - k \in \mathsf{B}^+[i]$.

Lemma 6. *If* $\mathsf{OC}[i..i+k_1+k_2+t+1] = 10^{k_1}1^t0^{k_2}1$ *and* $k_1, k_2 > 0$*, then* $\mathsf{P}[i] < k_1 + k_2$.

Proof. By contradiction. Assume $\mathsf{P}[i] \geq k_1 + k_2$. Then by Lemma 5 we have $\mathsf{P}[i] - k_1 \in \mathsf{B}^+[\mathsf{P}[i]]$, which implies that k_1 is a period of $S[1..\mathsf{P}[i]]$. Similarly, k_2 is a period of $S[1..\mathsf{P}[i]+t]$ and then of $S[1..\mathsf{P}[i]+1]$ and $S[1..\mathsf{P}[i]]$, since $\mathsf{P}[i] \geq k_2$. By the Periodicity Lemma 1 we know that $K = \gcd(k_1, k_2)$ is also a period of $S[1..\mathsf{P}[i]]$. Note that $k_1 - k_2$ is divisible by K.

Furthermore, $S[i+1] \neq S[i+1+k_1]$ because $\mathsf{OC}[i+1]$ is not 1. By Lemma 4, we have $\mathsf{P}[i] + 1 - k_1 \in \mathsf{B}^+[i+1]$, which implies $S[i+1] = S[\mathsf{P}[i] + 1 - k_1]$.

However, $S[i+1+k_1] = S[\mathsf{P}[i]+1] = S[\mathsf{P}[i]+1-k_2] = S[\mathsf{P}[i]+1-k_2-(k_1-k_2)] = S[\mathsf{P}[i]+1-k_1] = S[i+1]$, which is a contradiction.

Theorem 1. *Let* S *be a string of length* n*. Then the number of runs in its* OC *array is* $\mathcal{O}(\sqrt{n})$.

Proof. Let $\mathsf{OC}_S = 1^{t_1}0^{k_1}\cdots 1^{t_m}0^{k_m}$, where $k_m \geq 0$ and all other exponents are positive. By Lemma 6, we have for $1 < i < m$,

$$k_{i-1} + k_i \geq \sum_{r=1}^{i-1} t_r \geq i - 1\,.$$

This implies

$$n = \sum_{i=1}^{m}(t_i + k_i) \geq m + \sum_{j=1}^{\lfloor\frac{m-1}{2}\rfloor}(k_{2j-1} + k_{2j}) \geq m + \sum_{j=1}^{\lfloor\frac{m-1}{2}\rfloor}(2j-1) = m + \left\lfloor\frac{m-1}{2}\right\rfloor^2$$

so that $n = \Omega(m^2)$ and then $m = \mathcal{O}(\sqrt{n})$.

The bound in the previous proposition is tight. Indeed, there exists a binary string whose OC array is $\prod_{k>0} 10^k$. Actually, the string is uniquely determined by its OC array and can be defined by $u = a\prod_{k>0}\overline{u[k]}u[1..k] = abaaabbabababaa\cdots$.

The following proposition is a direct consequence of the definition of MCS. Essentially, it says that we can check if $S[i..j]$ is a MCS by looking at the OC array of the suffixes starting at position i and $i-1$.

Proposition 1. *Let S be a string of length n. If $S[i..j]$ is a MCS, then $\mathsf{OC}_{S[i..n]}[j-i+1] = 1$ and either $j-i+1 = n$ or $\mathsf{OC}_{S[i..n]}[j-i+2] = 0$. Moreover, either $i = 1$ or $\mathsf{OC}_{S[i-1..n]}[j-i+2] = 0$.*

Example 2. Let $S = aabaaaabaaba$. The OC arrays of the first few suffixes of S are displayed below.

S	a	a	b	a	a	a	a	b	a	a	b	a
$\mathsf{OC}_{S[1..n]}$	1	1	0	0	1	0	0	1	1	1	0	0
$\mathsf{OC}_{S[2..n]}$		1	0	1	0	0	0	1	1	1	0	0
$\mathsf{OC}_{S[3..n]}$			1	0	0	0	0	1	1	1	0	0
$\mathsf{OC}_{S[4..n]}$				1	1	1	1	0	0	0	0	0
$\mathsf{OC}_{S[5..n]}$					1	1	1	0	0	0	0	0
$\mathsf{OC}_{S[6..n]}$						1	1	0	0	1	1	1

One can check for instance that $S[4..7]$ is a MCS, because the $4 = (7-4+1)$th entry of $\mathsf{OC}_{S[4..n]}$ is a 1 which does not have another 1 on its right nor on top of it (i.e., in the OC array of the previous suffix). Similarly, $S[6..12]$ is a MCS because the last entry of $\mathsf{OC}_{S[6..n]}$ is 1 with a 0 on top.

As a consequence of the previous proposition, the number of MCSs in S is bounded from above by the total number of runs of 1s in all the OC arrays of the suffixes of S.

From Theorem 1, we therefore have a bound of $\mathcal{O}(n\sqrt{n})$ on the number of MCSs in a string of length n.

4 An algorithm for locating all MCS

In the previous section, we saw that one can locate all MCSs of S by looking at the OC arrays of all suffixes of S. However, since the OC array of a string of length n requires $\Omega(n)$ time to be constructed, this yields an algorithm that needs $\Omega(n^2)$ time to locate all MCSs.

We now describe an algorithm for computing all the maximal closed substrings in a string S of length n. For simplicity of exposition we assume that S is on a binary alphabet $\{a, b\}$, however the algorithm is easily adapted for strings on any constant-sized alphabet. The running time is asymptotically bounded by $n \log n$ plus the total number of MCSs in S.

The inspiration for our approach is an algorithms for finding maximal pairs under gap constraints due to Brodal, Lyngsø, Pedersen, and Stoye [3]. The central data structure is the suffix tree of the input string, which we now define.

Definition 1 (Suffix tree). *The suffix tree $T(S)$ of the string S is the compressed trie of all suffixes of S. Each leaf in $T(S)$ represents a suffix $S[i..n]$ of S and is annotated with the index i. We refer to the set of indices stored at the leaves in the subtree rooted at node v as the leaf-list of v and denote it $LL(v)$. Each edge in $T(S)$ is labelled with a nonempty substring of S such that the path from the root to the leaf annotated with index i spells the suffix $S[i..n]$. We refer to the substring of S spelled by the path from the root to node v as the path-label of v and denote it $L(v)$.*

At a high level, our algorithm for finding MCSs processes the suffix tree (which is a binary tree, for binary strings) in a bottom-up traversal. At each node the leaf lists of the (two, for a binary string) children are intersected. For each element in the leaf list of the smaller child, the successor in the leaf list of the larger child is found. Note that because the element from the smaller child and its successor in the larger child come from different subtrees, they represent a pair occurrences of substring $L(v)$ that are right-maximal. To ensure left maximality, we must take care to only output pairs that have different preceding characters. We explain how to achieve this below.

Essential to our algorithm are properties of AVL trees that allow their efficient merging, and the so-called "smaller-half trick" applicable to binary trees. These proprieties are captured in the following lemmas.

Lemma 7 (Brown and Tarjan [4]). *Two AVL trees of size at most n and m can be merged in time $\mathcal{O}(\log \binom{n+m}{n})$.*

Lemma 8 (Brodal et al. [3], Lemma 3.3). *Let T be an arbitrary binary tree with n leaves. The sum over all internal nodes v in T of terms that are $\mathcal{O}(\log \binom{n_1+n_2}{n_1})$, where n_1 and n_2 are the n_1 numbers of leaves in the subtrees rooted at the two children of v, is $\mathcal{O}(n \log n)$.*

As stated above, our algorithm traverses the suffix tree bottom up. At a generic step in the traversal, we are at an internal node v of the suffix tree. Let the two children of node v be v_ℓ and v_r (recall the tree is a binary suffix tree, so every internal node has two children). The leaf lists of each child of v are maintained in two AVL trees — note, there are *two AVL trees for each of the two children*, two for v_ℓ and two for v_r. For a given child, say v_r, one of the two AVL trees contains positions where $L(v_r)$ is proceeded by an a symbol, and the other AVL tree contains positions where $L(v_r)$ is proceeded by a b symbol in S. Call these the a-tree and b-tree, respectively.

Without loss of generality, let v_r be the smaller of $v's$ children. We want to search for the successor and predecessor of each of the elements of v_r's a-tree amongst the elements v_ℓ's b-tree, and, similarly the elements of v_r's b-tree with the elements from v_ℓ's a-tree. Observe that the resulting pairs of elements represent a pair of occurrences of $L(v)$ that are both right and left maximal: they have different preceding characters and so will be left maximal, and they are siblings in the suffix tree and so will be right maximal. These are candidate MCSs. What remains is to discard pairs that are not consecutive occurrences of $L(v)$, to arrive at the MCSs. Discarding is easy if we process the elements of $LL(v_r)$ in order (which is in turn easy, because they are stored in two AVL trees). To see this, consider two consecutive candidates that have the same right border position (a successor found in $LL(v_\ell)$; discarding for left borders is similar). The first of these candidates can clearly be discarded because there is an occurrence of $L(v)$ (from $LL(v_r)$) in between the two borders, preventing the pair of occurrences from forming an MCS. Because we only compute a successor/predecessor for each of the elements of the smaller of v's children, by Lemma 8 the total time for all successor/predecessor searches will be $O(n \log n)$ (discarding also takes time proportinal to the smaller subtree, and so does not increase this complexity). After this, the a-tree and b-tree of the smaller child are merged with their counterparts from the larger child.

Thus, by Lemmas 7 and 8, the overall processing is bounded by $\mathcal{O}(n \log n)$ in addition to the number of MCSs that are found.

The above approach is easily generalized from strings on binary alphabets to those on any alphabet of constant size by replacing nodes of the suffix tree having degree $d > 2$ with binary trees of height $\log d$. This does not increase the height of the suffix tree asymptotically and so preserves the runtime stated above. It would be interesting to design algorithms for general alphabets, and we leave this as an open problem.

References

1. G. Badkobeh, G. Fici, and Zs. Lipták. On the Number of Closed Factors in a Word. In *LATA 2015, 9th International Conference on Language and Automata Theory and Applications*, volume 8977 of *Lecture Notes in Computer Science*, pages 381–390. Springer International Publishing, 2015.
2. H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, and K. Tsuruta. The "runs" theorem. *SIAM J. Comput.*, 46(5):1501–1514, 2017.
3. G. S. Brodal, R. B. Lyngsø, C. N. S. Pedersen, and J. Stoye. Finding maximal pairs with bounded gap. In M. Crochemore and M. Paterson, editors, *Proc. 10th*

Annual Symposium Combinatorial Pattern Matching (CPM), LNCS 1645, pages 134–149. Springer, 1999.
4. M. R. Brown and R. E. Tarjan. A fast merging algorithm. *J. ACM*, 26(2):211–226, 1979.
5. M. Bucci, A. D. Luca, and G. Fici. Enumeration and structure of trapezoidal words. *Theor. Comput. Sci.*, 468:12–22, 2013.
6. A. Carpi and A. de Luca. Periodic-like words, periodicity, and boxes. *Acta Informatica*, 37(8):597–618, 2001.
7. A. De Luca, G. Fici, and L. Q. Zamboni. The sequence of open and closed prefixes of a Sturmian word. *Adv. Appl. Math.*, 90:27–45, 2017.
8. G. Fici. Open and Closed Words. *Bull. Eur. Assoc. Theor. Comput. Sci. EATCS*, 123:140–149, 2017.
9. N. J. Fine and H. S. Wilf. Uniqueness theorems for periodic functions. *P. Am. Math. Soc.*, 16(1):109–114, 1965.
10. P. Gawrychowski, T. I, S. Inenaga, D. Köppl, and F. Manea. Tighter bounds and optimal algorithms for all maximal α-gapped repeats and palindromes - finding all maximal α-gapped repeats and palindromes in optimal worst case time on integer alphabets. *Theory Comput. Syst.*, 62(1):162–191, 2018.
11. R. Kolpakov, M. Podolskiy, M. Posypkin, and N. Khrapov. Searching of gapped repeats and subrepetitions in a word. *J. Discrete Algorithms*, 46-47:1–15, 2017.
12. R. M. Kolpakov and G. Kucherov. Finding maximal repetitions in a word in linear time. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 596–604. IEEE Computer Society, 1999.
13. J. H. Morris and V. R. Pratt. A linear pattern-matching algorithm. Technical Report 40, University of California, Berkeley, 1970.