Computing Regularities in Strings

W. F. Smyth
Department of Computing and Software
McMaster University
Hamilton, Ontario, Canada
smyth@mcmaster.ca
Digital Ecosystems and Business Intelligence Institute
Curtin University
Perth, WA, Australia

Munina Yusufu
Department of Computing and Software
McMaster University
Hamilton, Ontario, Canada
yusufum@mcmaster.ca
Digital Ecosystems and Business Intelligence Institute
Curtin University
Perth, WA, Australia

Abstract

Regularities in strings model many phenomena and thus form the subject of extensive mathematical studies [23]. Perhaps the most conspicuous regularities in strings are those that manifest themselves in the form of repeated subpatterns. In this paper, we study several forms of regularities of strings, that is, repeats, multirepeats, repetitions and runs. We present their similarities and differences by discussing their forms and properties and we explore the existing computation algorithms. We also discuss several data structures useful for computing regularities.

1. Introduction

Various forms of regularity are central to the recognition of important patterns in performing retrieval from massive data sets. In this paper, we investigate mathematical and algorithmical aspects of regularities in strings.

The study of strings began a little over 100 years ago with a mathematical study of periodicity [37], the simplest form of regularity. Today algorithms for computing regularities have myriad applications:

Data Compression: Regularities in the form of repeating substrings were the basis of `gzip`, one of the earliest and still widely-used compression algorithms [41], and remain central in more recent approaches [7].

Computational Biology: Repeats and repetitions of lengthy substrings in DNA and protein sequences are important markers in biological research [5].

Information Security: Spam, the electronic equivalent of junk mail, affects over 600 million users worldwide. Some methods for detecting spam are mainly based on similarity calculations on strings [39].

Data Mining: Various forms of regularity are central to the recognition of important patterns in retrieval from massive data sets [17].

Analysis of Musical Texts: The identification of melodies and rhythms in huge musical databases depends heavily on algorithms for computing string regularities, approximate and exact [12].

Software Engineering: Identifying approximate clones of methods or classes in large software systems is important to software maintenance [4].

Apart from expected benefits in application areas discussed above, there should also be spin-off benefits of this research within the general scientific/technological area of combinatorial algorithms.

2. Forms of Various Regularities

In this section we discuss the various regularities in strings, giving definitions, properties and relationship between them, while citing the existing computation algorithms.

2.1. Repeats

Intuitively, a ***repeat*** is a collection of repeating substrings, not necessarily adjacent. More formally, a ***repeat*** in $\boldsymbol{x}$ is a tuple

$$\boldsymbol{M_{x,u}} = (p; i_1, i_2, ..., i_e),$$

where $e \geq 2, 1 \leq i_1 < i_2 < ... < i_e \leq n$, and

$$\boldsymbol{u} = \boldsymbol{x}[i_1..i_1 + p - 1] = \boldsymbol{x}[i_2..i_2 + p - 1] = ... = \boldsymbol{x}[i_e..i_e + p - 1].$$

Note that it may happen, for some $j \in 1..e-1$, that $i_{j+1} - i_j = p$ or that $i_{j+1} - i_j < p -$ that is, the substrings of a repeat may be adjacent or even overlap. We call $\boldsymbol{u}$ the ***generator***, $\boldsymbol{p}$ the ***period***, and $\boldsymbol{e}$ the ***exponent*** of $\boldsymbol{M_{x,u}}$; also, $\boldsymbol{M_{x,u}}$ is called a ***square*** if $e = 2$. We say that $\boldsymbol{M_{x,u}}$ is ***complete*** if for every $i \in 1..n$ and $i \notin \{i_1, i_2, ..., i_e\}$, we are assured that $\boldsymbol{x}[i..i + p - 1] \neq \boldsymbol{u}$. We say that $\boldsymbol{M_{x,u}}$ is ***left-extendible*** (LE) if

$$(p; i_1 - 1, i_2 - 1, ..., i_e - 1)$$

is a repeat; similarly, $\boldsymbol{M_{x,u}}$ is ***right-extendible*** (RE) if

978-1-4244-4520-2/09/$25.00 ©2009 IEEE

$$(p; i_1 + 1, i_2 + 1, ..., i_e + 1)$$

is a repeat. If $\boldsymbol{M_{x,u}}$ is neither LE nor RE, we say that it is ***nonextendible*** (NE). In $\boldsymbol{x} = abaababa$, the NE repeats are

$$\boldsymbol{M_{x,a}} = (1; 1, 3, 4, 6, 8) \text{ and } \boldsymbol{M_{x,aba}} = (3; 1, 4, 6).$$

Of particular interest are repeating substrings $\boldsymbol{u}$ such that $\boldsymbol{M_{x,v_1uv_2}}$ is a repeat if and only if $\boldsymbol{v_1} = \boldsymbol{v_2} = \boldsymbol{\epsilon}$, the empty string – in other words, $\boldsymbol{u}$ is not a proper substring of any other repeating substring. We call such repeats ***supernonextendible*** (SNE). In the above example, (3; 1, 4, 6) is the unique SNE repeat.

In [16, p. 147] an algorithm is described that, given the suffix tree $\mathrm{ST}_{\boldsymbol{x}}$ of $\boldsymbol{x}$, computes all the NE (called "maximal") *pairs of repeats* in $\boldsymbol{x}$ in time $O(\alpha n+q)$, where q is the number of pairs output. [6] uses similar methods to compute all NE pairs $(p; i_1, i_2)$ such that $i_2 - i_1 \geq g_{min}$ (or $\leq g_{max}$) for user-defined ***gaps*** g_{min}, g_{max}. [1] shows how to use the suffix array $\mathrm{SA}_{\boldsymbol{x}}$ of $\boldsymbol{x}$ to compute the NE pairs in time $O(\alpha n+q)$. Since it may be that $\alpha \in O(n)$, all of these algorithms require $O(n^2)$ time in the worst case. [14] uses the suffix arrays of both $\boldsymbol{x}$ and its reversed string $\overline{\boldsymbol{x}} = \boldsymbol{x}[n]\boldsymbol{x}[n-1]\cdots\boldsymbol{x}[1]$ to compute all the complete NE repeats in $\boldsymbol{x}$ in $\Theta(n)$ time. More recently, [31] describes suffix array-based $\Theta(n)$-time algorithms to compute all ***substring equivalence classes*** — essentially the complete NE repeats — in $\boldsymbol{x}$.

In [34], we first describe a new algorithm PSY1 that, based on suffix array construction, computes all the complete NE repeats in $\boldsymbol{x}$ of length $p \geq p_{min}$. PSY1 executes in $\Theta(n)$ time independent of alphabet size and is an order of magnitude faster than the two other algorithms previously proposed for this problem. Second, we describe a new fast algorithm PSY2 for computing all complete SNE repeats in $\boldsymbol{x}$ that also executes in $\Theta(n)$ time independent of alphabet size, thus asymptotically faster than methods previously proposed. Both algorithms require $6n$ bytes of storage, including preprocessing.

2.2. Multirepeats

In this section we consider the multirepeats problem with various constraints.

A repeat $M_{x,u}$ of ***multiplicity*** m is the occurrence of generator u in string $\boldsymbol{x}$ m times. We define the ***quorum*** q to be the minimum number of strings in a set of string such that a maximal multirepeat must occur, in order to be considered valid.

A ***multirepeat*** is a repeat of minimum length p_{min} that occurs at least m_{min} times ($m_{min} \geq 2$) in each of at least $q \geq 1$ strings in a given set of strings. Consider, for example, the three strings given in Figure 1.

Given $p_{min} = 3$, $m_{min} = 2$ and $q = 2$, we see that ACG satisfies all the constraints including minimum period, minimum multiplicity, and quorum. Therefore ACG is a multirepeat in $\boldsymbol{s_1}$, $\boldsymbol{s_2}$ and $\boldsymbol{s_3}$.

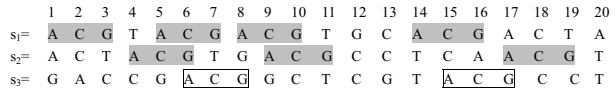

Figure 1. Three Strings $\boldsymbol{s_1}$, $\boldsymbol{s_2}$ and $\boldsymbol{s_3}$

Assuming that u occurs twice in a string $\boldsymbol{x}$ at positions i_1 and i_2, then the number of symbols between them is called a ***gap*** and it is equal to $g_1 = |i_2 - i_1| - p$. In the case that $g_1 = 0$, then $M_{x,u}$ is called a ***tandem repeat***; if $g_1 < 0$, then it is called ***overlapping***.

If restrictions are posed on the gaps between occurrences of u, then the gap g_i between the ith and $(i+1)$th occurrence of u is bounded as follows: $d_{min_i} \leq g_i \leq d_{max_i}$, where d_{min_i} and d_{max_i} are lower and upper bounds on the gap size, respectively. Thus, in this case a repeat $M_{x,u}$ is represented by the pair (u, d), where d is a tuple $((d_{min_1},d_{max_1}), (d_{min_2},d_{max_2}),...,(d_{min_{m-1}},d_{max_{m-1}}))$.

If we add the gap restriction to the above example, choosing $d_{min_i} = 1$ and $d_{max_i} = 4$ for all i, so that $1 \leq g_i \leq 4$, then ACG is a multirepeat only in $\boldsymbol{s_1}$ and $\boldsymbol{s_2}$ (shaded occurrences).

There exists only one algorithm [3] to compute multirepeats. This algorithm is not space-efficient since it uses suffix trees, one for each string in the set plus a "generalized" suffix tree for all of them. Thus it is not easy to implement. In addition, it has high time complexity.

In [18], we describe a family of efficient algorithms based on suffix arrays to compute maximal multirepeats under various constraints. Our algorithms are faster, more flexible and much more space-efficient than the algorithms in [3].

2.3. Repetitions

A ***repetition*** is a sequence of adjacent repeating substrings. More precisely, a repetition in a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is a substring $\boldsymbol{x}[i..i + pe - 1] = \boldsymbol{u}^e$, where $|\boldsymbol{u}| = p$ and $e \geq 2$. If moreover $\boldsymbol{u}$ itself is not a repetition, then $\boldsymbol{u^e}$ is said to be ***irreducible***; and if neither $\boldsymbol{x}[i - p..i - 1]$ nor $\boldsymbol{x}[i + pe..i + p(e + 1) - 1]$ equals $\boldsymbol{u}$, then $\boldsymbol{u^e}$ is said to be ***maximal***.

Analogous to a repeat, we call $\boldsymbol{u}$ the ***generator***, $\boldsymbol{p}$ the ***period***, and $\boldsymbol{e}$ the ***exponent*** of the repetition $\boldsymbol{u^e}$. Note that a repetition is completely specified by the triple (i, p, e). In the string

$$\begin{array}{rcccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \boldsymbol{x} = & a & b & a & a & b & a & b & a \end{array}$$

the repetitions are $(1, 3, 2) = (aba)^2, (3, 1, 2) = a^2, (4, 2, 2) = (ab)^2$, and $(5, 2, 2) = (ba)^2$. Since in each case $e = 2$, all of these are ***squares***.

About a quarter-century ago, three algorithms were discovered [8], [2], [26] that employed widely different approaches to computing all the repetitions in a given string $\boldsymbol{x}[1..n]$ in $O(n \log n)$ time; of these algorithms, two were based on a form of suffix tree calculation ([2] explicitly, [8] implicitly), while the third used a divide-and-conquer technique.

A lot of work has been done for identifying the repetitions in a string in recent years. In the area of computational biology, algorithms for finding repetitions are presented in [5]. [24] considers the problem of finding occurrences of contiguous repeats of substrings that are within some Hamming- or edit-distance of each other.

2.4. Runs

Intuitively, a ***run*** is a maximal sequence of overlapping repetitions of the same period.

The maximum number of repetitions in a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is $\Theta(n \log n)$. But this is a count of repetitions that are both maximal and irreducible. If instead we were asked to output the distinct squares $\boldsymbol{u^2}$ without these restrictions, we would find that $\boldsymbol{x} = a^n$, for example, would require $\lfloor n^2/4 \rfloor$ – that is, $\Theta(n^2)$ – outputs to specify squares

$$\boldsymbol{x}[1..2], \boldsymbol{x}[2..3], ..., \boldsymbol{x}[n-1..n], \boldsymbol{x}[1..4], \boldsymbol{x}[2..5], ..., \boldsymbol{x}[n-3..n],$$

and so on. Thus in restricting the output to maximal irreducible repetitions, we ***encode*** the output, by tacit agreement with the user, so as to reduce its quantity, hence the asymptotic complexity of the algorithm. For $\boldsymbol{x} = a^n$, this encoding dramatically reduces the output to a single repetition $(1, 1, n)$.

We now describe another encoding of repetitions that further reduces the quantity of output required to $\Theta(n)$. We say that a repetition $(i, p, e) = \boldsymbol{u}^e$ is ***left-extendible*** (LE) if there exists a repetition at position $i-1$ of $\boldsymbol{x}$ that is also of period p. If no such repetition exists, we say that (i, p, e) is NLE. Given an NLE repetition (i, p, e), denote by t the greatest integer such that, for every $j \in 0..t$, $(i+j, p, e)$ is a repetition. Note that since (i, p, e) is maximal, therefore $t \in 0..p-1$. We call t the ***tail*** of (i, p, e). Then a ***run (maximal periodicity)*** is a 4-tuple (i, p, e, t), where (i, p, e) is an NLE repetition of tail t.

The maximum number $\rho(n)$ of runs in a string of length n has been known to be $\theta(n)$ [22], and the exact bound is a subject of intense current research. It is known that $\rho(n) \geq 0.944565n$ [27] and $\rho(n) \leq 1.029n$ [11].

Optimal $\Theta(n)$-time algorithms for computing all runs exist based on suffix trees [13], [22] or suffix arrays [21], [19], [1] together with Lempel-Ziv factorization [41]. Faster and more space-efficient algorithms have recently been proposed [9].

3. Data Structures used for computing regularities

In this section, we discuss several data structures which have been widely used to compute regularities in strings, including their basic forms, properties, and construction time and space complexities.

3.1. Suffix Tree (ST)

The suffix tree is one of the most important data structures in string processing. The suffix tree for $\boldsymbol{x}$ of length n is defined as a rooted tree. Figure 2 shows the suffix tree of $\boldsymbol{x} = abcaabcabaccabaacb\$$. For $i \in 1..18$, the leaf nodes

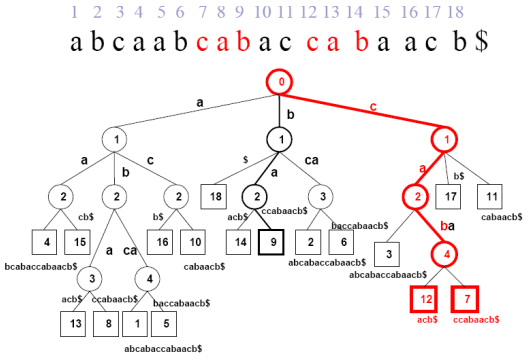

Figure 2. The Suffix Tree for String $\boldsymbol{x} = abcaabcabaccabaacb\$$

labeled i, denoting suffixes $\boldsymbol{x}[i..18]$, occur in lexicographical order from left to right. The internal nodes are labeled with the longest common prefix of the leaf nodes below. ST can be computed in $O(n \log \alpha)$ time [40], [25], where $\alpha \in O(n)$, and online [38] with the same time complexity; on an integer alphabet, $\Theta(n)$-time efficiency is possible [13], but the algorithm is not practical for long strings.

3.2. Suffix Array (SA)

Consider a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ defined on an ordered alphabet A of size α (where if there is no explicit bound on alphabet size, we suppose $\alpha \leq n$). We refer to the suffix $\boldsymbol{x}[i..n]$, $i \in 1..n$, simply as ***suffix*** i. Then the ***suffix array*** SA is an array $[1..n]$ in which $\mathrm{SA}[j] = i$ iff suffix i is the j^{th} in lexicographical order among all the suffixes of $\boldsymbol{x}$.

The SA array of the string

	1	2	3	4	5	6	7	8	9	10	11
$\boldsymbol{x} =$	a	b	a	b	a	b	b	b	a	a	a

is shown in the third column of Figure 3.

SA can be computed in $\Theta(n)$ worst-case time [19], [21], though various supralinear methods [30], [28] are certainly

i	x[SA[i]..n]	SA[i]	LCP[i]	LPF[i]	BWT[i]
1	a	11	0	0	a
2	aa	10	1	0	a
3	aaa	9	2	4	b
4	ababababbaaa	1	1	3	$
5	abababbaaa	3	4	2	b
6	ababbaaa	5	2	1	b
7	baaa	8	0	2	b
8	bababbbaaa	2	2	2	a
9	babbbaaa	4	3	1	a
10	bbaaa	7	1	2	b
11	bbbaaa	6	2	1	a

Figure 3. SA, LCP, LPF and BWT arrays of $\boldsymbol{x} = ababababbaaa$

much faster, as well as more space-efficient, in practice [33], in some cases requiring space only for $\boldsymbol{x}$ and SA itself, which requires only $4n$ bytes (4 bytes per input character) in its basic form, compared to 20–40n bytes for the corresponding suffix tree. In [1] an enhanced suffix array (ESA) is introduced, consisting of the suffix array together with an "lcp-interval tree".

3.3. Longest Common Prefix (LCP) Array

Another important data structure that is often used with the suffix array is the Longest Common Prefix (LCP) array. Let us denote the length of the longest common prefix of suffixes i and j by **lcp(i,j)**. Then, the LCP array contains the lengths of the longest common prefixes between successive suffixes of SA. That is, for $1 < i \leq n$,

$$\text{LCP}[\boldsymbol{i}] = lcp(\text{SA}[i], \text{SA}[i-1]).$$

Given $\boldsymbol{x}$ and SA, LCP can also be computed in $\Theta(n)$ time [20], [29], [35]: the first algorithm described in [29] requires $9n$ bytes of storage and is almost as fast in practice as that of [20], which requires $13n$ bytes. However the algorithm recently proposed in [35] is generally faster and requires about $6n$ bytes of storage for its execution, since it overwrites the suffix array. The fourth column of Figure 3 gives the LCP array of the string $ababababbaaa$.

3.4. Longest Previous Factor (LPF) Array

The Longest Previous Factor (LPF) array was introduced in [10], but also appears as the **prefix array** $\boldsymbol{\pi}$ in [15]. For any position i in a string $\boldsymbol{x}$, **LPF[i]** is defined to be the length of the longest factor of $\boldsymbol{x}$ starting at position i that occurs previously in $\boldsymbol{x}$. Formally, [10] defines LPF[i] as follows:

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

LPF array can be conveniently used for computing LZ factorizations, and also runs. For an example of LPF, see column 5 of Figure 3. It is shown in [10] that LPF is a permutation of LCP.

3.5. Burrows-Wheeler Transform (BWT) Array

We define the Burrows-Wheeler Transform BWT of $\boldsymbol{x}$ [7]: for SA[j] > 1, BWT[j] $= \boldsymbol{x}\big[\text{SA}[j]-1\big]$, while for j such that SA[j] $= 1$, BWT[j] $=$ \$, a sentinel letter not equal to any other in $\boldsymbol{x}$. BWT can clearly be computed in linear time from SA; some of our algorithms [34] and LZ algorithms [32] use the BWT array since it occupies only n rather than $4n$ bytes. BWT is illustrated in column 6 of Figure 3.

4. Conclusion

As recently as 10 years ago, it was not possible to claim, even difficult to imagine, that regularities in strings could be computed in time linear in string length. Today all the regularities mentioned here (repeats, multirepeats, repetitions and runs) and their data structures (suffix/LCP/LPF/BWT arrays) can be computed in linear time and space. These are impressive advances in an area of combinatorial algorithms that has numerous applications. We expect that the future will be as exciting as the recent past.

References

[1] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch, **Replacing suffix trees with enhanced suffix arrays**, *J. Discrete Algs. 2* (2004) 53–86.

[2] Alberto Apostolico and Franco P. Preparata, **Optimal off-line detection of repetitions in a string**, *Theoret. Comput. Sci. 22* (1983) 297–315.

[3] A. Bakalis, C.S. Iliopoulos, C. Makris, S. Sioutas, E. Theodoridis, A. Tsakalidis, and K. Tsichlas, **Locating maximal multirepeats in multiple strings under various constraints**, *The Computer Journal 50–2* (2007) 178–185.

[4] Hamid Abdul Basit, Simon J. Puglisi, W. F. Smyth, Andrew Turpin, and Stan Jarzabek, **Efficient token based clone detection with flexible tokenization**, *Proc. 6th Joint Meeting: European Software Engineering Conference & ACM SIGSOFT Symposium on Software Engineering* (2007) 513–516.

[5] G. Benson, **Tandem repeats finder: A program to analyze DNA sequences**, *Nucleic Acids Research 27-2* (1999) 573–580.

[6] Gerth S. Brodal, Rune B. Lyngso, Christian N. S. Pederesen, and Jens Stoye, **Finding maximal pairs with bounded gap**, *J. Discrete Algs. 1* (2000) 77–103.

[7] Michael Burrows and David J. Wheeler, *A block-sorting lossless data compression algorithm*, Technical Report 124, Digital Equipment Corporation (1994).

[8] Maxime Crochemore, **An optimal algorithm for computing the repetitions in a word**, *Inform. Process. Lett. 12–5* (1981) 244–250.

[9] G. Chen, S.J. Puglisi, and W.F. Smyth, **Fast and practical algorithms for computing all runs in a string**, *Proc. 18th Annual Symposium on Combinatorial Pattern Matching* (2007), 307–315.

[10] M. Crochemore and L. Ilie, **Computing longest previous factor in linear time and applications**, *Inform. Proc. Lett. 106* (2008) 75–80.

[11] Maxime Crochemore, Lucian Ilie, and Liviu Tinta, **The "runs" conjecture**, submitted for publication.

[12] Manolis Christodoulakis, C. S. Iliopoulos, M. Sohel Rahman, and W. F. Smyth, **Identifying rhythms in musical texts**, *Internat. J. Foundations of Computer Sci. 19-1* (2008) 37–52.

[13] Martin Farach, **Optimal suffix tree construction with large alphabets**, *Proc. 38th IEEE Symp. Found. Computer Science* (1997) 137–143.

[14] Frantisek Franek, W. F. Smyth, and Yudong Tang, **Computing all repeats using suffix arrays**, *J. Automata, Languages & Combinatorics 8–4* (2003) 579–591.

[15] Frantisek Franek, Jan Holub, W. F. Smyth, and Xiangdong Xiao, **Computing quasi suffix arrays**, *J. Automata, Languages & Combinatorics 8–4* (2003) 593–606.

[16] Dan Gusfield, *Algorithms on Strings, Trees and Sequences*, Cambridge University Press (1997) 534 pp.

[17] Jiawei Han and Micheline Kamber, *Data Mining: Concepts and Techniques*, 2nd edition, Morgan Kaufmann (2006).

[18] C. S. Iliopoulos, W. F. Smyth, and Munina Yusufu, **Faster algorithms for computing maximal multirepeats in multiple sequences**, submitted for publication (2008).

[19] Juha Kärkkäinen and Peter Sanders, **Simple linear work suffix array construction**, *Proc. 30th Internat. Colloq. Automata, Languages & Programming* (2003) 943–955.

[20] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park, **Linear-time longest-common-prefix computation in suffix arrays and its applications**, *Proc. 12th Annual Symp. Combinatorial Pattern Matching* (2001) 181–192.

[21] Pang Ko and Srinivas Aluru, **Space efficient linear time construction of suffix arrays**, *Proc. 14th Annual Symp. Combinatorial Pattern Matching* (2003) 200–210.

[22] R. Kolpakov and G. Kucherov, **Finding maximal repetitions in a word in linear time**, *Proc. 40th Annual IEEE Symp. Found. Computer Science* (1999) 596–604.

[23] M. Lothaire, *Combinatorics on Words*, Addison-Wesley, Read-ing, Mass. (1983)

[24] G. M. Landau and J. P. Schmidt, **An Algorithm for Approximate Tandem Repeats**, *Proc. 4th Annual Symposium on Combinatorial Pattern Matching* (1993) 120–133.

[25] Edward M. McCreight, **A space-economical suffix tree construction algorithm**, *J. Assoc. Comput. Mach. 32–2* (1976) 262–272.

[26] Michael G. Main and Richard J. Lorentz, **An $O(n \log n)$ algorithm for finding all repetitions in a string**, *J. Algs. 5* (1984) 422–432.

[27] Wataru Matsubara, Kazuhiko Kusano, Akira Ishino, Hideo Bannai, and Ayumi Shinohara1, **New lower bounds for the maximum number of runs in a string** , *Proc. Prague Stringology Conference 2008* (2008) 140–144.

[28] Michael Maniscalco and Simon J. Puglisi, **Faster lightweight suffix array construction**, *Proc. 17th Australasian Workshop on Combinatorial Algs.* (2006) 16–29.

[29] Giovanni Manzini, **Two space-saving tricks for linear time LCP computation**, *Proc. 9th Scandinavian Workshop on Alg. Theory* (2004) 372–383.

[30] Giovanni Manzini and Paolo Ferragina, **Engineering a lightweight suffix array construction algorithm**, *Algorithmica 40* (2004) 33–50.

[31] Kaziyuki Narisawa, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda, **Efficient computation of substring equivalence classes with suffix arrays**, *Proc. 18th Annual Symp. Combinatorial Pattern Matching* (2007) 340–351.

[32] Daisuke Okanohara & Kunihiko Sadakane, **An online algorithm for finding the longest previous factors**, *Proc. 16th Annual European Symp. on Algs.* (2008) 696–707.

[33] Simon J. Puglisi, W. F. Smyth, and Andrew Turpin, **A taxonomy of suffix array construction algorithms**, *ACM Computing Surveys 39–2* (2007) 1–31.

[34] Simon J. Puglisi, W. F. Smyth, and Munina Yusufu, **Fast optimal algorithms for computing all the repeats in a string**, *Proc. Prague Stringology Conference* (2008) 161–169.

[35] Simon J. Puglisi and Andrew Turpin, **Space-time tradeoffs for longest-common-prefix array computation**, *Proc. 19th Internat. Symposium on Algorithms and Computation (ISAAC'08)* (2008) 124–135.

[36] Bill Smyth, *Computing Patterns in Strings*, Pearson Addison-Wesley (2003) 423 pp.

[37] Axel Thue, **Über unendliche zeichenreihen**, *Norske Vid. Selsk. Skr. I. Mat. Nat. Kl. Christiana 7* (1906) 1–22.

[38] Esko Ukkonen, **On-line construction of suffix trees**, *Algorithmica 14* (1995) 249–260.

[39] Tanguy Urvoy, Thomas Lavergne, and Pascal Filoche, **Tracking Web spam with hidden style similarity**, *AIRWEB 2006* (2006) 25–31.

[40] Peter Weiner, **Linear pattern matching algorithms**, *Proc. 14th Annual IEEE Symp. Switching and Automata Theory* (1973) 1–11.

[41] Jacob Ziv and Abraham Lempel, **A universal algorithm for sequential data compression**, *IEEE Trans. Information Theory 23* (1977) 337–343.