Fundamenta Informaticae XXI (2001) 1001–1017
IOS Press

External Memory Algorithms for String Problems*

Kangho Roh
School of Computer Science and Engineering, Seoul National University, Seoul, Korea
khroh@theory.snu.ac.kr

Maxime Crochemore†
Institut Gaspard-Monge, Université de Marne-la-Vallée, Paris, France
Department of Computer Sciences, King's College London, London, United Kingdom
Maxime.Crochemore@univ-mlv.fr

Costas S. Iliopoulos
Department of Computer Sciences, King's College London, London, United Kingdom
csi@dcs.kcl.ac.uk

Kunsoo Park‡
School of Computer Science and Engineering, Seoul National University, Seoul, Korea
kpark@theory.snu.ac.kr

Abstract. In this paper we present external memory algorithms for some string problems. External memory algorithms have been developed in many research areas, as the speed gap between fast internal memory and slow external memory continues to grow. The goal of external memory algorithms is to minimize the number of input/output operations between internal memory and external memory.

These years the sizes of strings such as DNA sequences are rapidly increasing. However, external memory algorithms have been developed for only a few string problems. In this paper we consider five string problems and present external memory algorithms for them. They are the problems of finding the maximum suffix, string matching, period finding, Lyndon decomposition, and finding the minimum of a circular string. Every algorithm that we present here runs in a linear number of I/Os in the external memory model with one disk, and they run in an optimal number of disk I/Os in the external memory model with multiple disks.

Keywords: External memory algorithm, maximum suffix, string matching, period finding, Lyndon decomposition, minimum of a circular string

Address for correspondence: Kunsoo Park, School of Computer Science and Engineering, Seoul National University, Seoul 151-744, Korea

*A preliminary version of this paper appeared in the proceedings of 17th Australasian Workshop on Combinatorial Algorithms.
†Partially granted by CNRS.
‡Supported by FPR05A2-341 of 21C Frontier Functional Proteomics Project from Korean Ministry of Science & Technology.

1. Introduction

Data sets in many applications are often too big to fit into main memory. In such applications it is important that algorithms take into account the memory constraints of the system [33, 38]. In most modern systems the memory is organized into a hierarchy. At the top level, fast internal memory which is expensive and has low storage capacity is located. At the bottom level, slower external memory which is cheap and has high storage capacity is located. Therefore the input/output communication (or simply *I/O*) between two levels in memory hierarchy can be a bottleneck in massive data set applications. One approach to optimizing performance is to develop algorithms that minimize I/Os between internal memory and external memory. These algorithms are referred to as *external memory algorithms* or more simply *EM algorithms*.

Early work in external memory algorithms focused on fundamental problems such as sorting, matrix multiplication, and FFT [1, 30, 39, 3]. More recently, research on EM algorithms has moved towards solving graph and geometric problems [7]. Work on graph problems includes transitive closure computations, some graph traversal problems, and memory management for maintaining connectivity information and paths on graphs [36, 29, 19].

String algorithmics is an important subject in algorithm research, and its applications include text processing, sequence analysis, information retrieval, DNA sequencing, etc. These years the sizes of strings such as DNA sequences are rapidly increasing. For example, the human genome sequence is too long to be loaded into the main memory of most computers. Thus external memory algorithms are necessary for string problems. However, external memory algorithms have been developed for only a few string problems: string B-trees by Ferragina and Grossi [16, 17], string sorting by Arge-Ferragina-Grossi-Vitter [3], suffix trees by Farach-Ferragina-Muthukrishnan [15] and Clark and Munro [8], dictionary matching by Ferragina and Luccio [18].

In this paper we present external memory algorithms for the following five string problems: 1) finding the maximum suffix of a string [11, 12], 2) string matching [6, 12, 20, 21, 26, 34], 3) finding the period of a string [10, 11], 4) Lyndon decomposition of a string [2, 14, 25], 5) finding the minimum of a circular string [5, 24, 35]. These are well-studied and fundamental problems in string algorithmics.

We first give two external memory algorithms for the maximum suffix problem: one uses four memory blocks and requires $6\lceil \frac{N}{B} \rceil$ I/O operations, and the other uses six memory blocks and $4\lceil \frac{N}{B} \rceil$ I/O operations, where N is the size of the given string and B is the block transfer size. Hence there is a tradeoff between memory blocks and I/O operations. Our algorithms for the remaining four problems are based on the maximum suffix algorithms; they incorporate in their algorithms either the maximum suffix algorithms directly or variations of the maximum suffix algorithms. Every algorithm we present in this paper runs in a linear number of I/Os in the external memory model with one disk. We also consider the external memory model with multiple disks. Every algorithm that we present runs in an optimal number of disk I/Os in the external memory model with multiple disks.

These results with previous external memory algorithms for string problems are just the beginning of a new research area, i.e., that of developing external memory algorithms for string problems. Many interesting problems in string algorithmics [22, 23, 13] can be considered for external memory algorithms.

The remainder of the paper is organized as follows. Section 2 introduces the external memory model and the string problems that we consider in this paper. Section 3 presents the maximum suffix problem and two external memory algorithms that solve it. Section 4 and Section 5 give a string matching algorithm and a period finding algorithm based on the maximum suffix algorithm, respectively. We consider the Lyndon decomposition problem in Section 6 and the problem of finding the minimum of a circular string in Section 7. We conclude in Section 8.

2. Preliminaries

2.1. External memory model

We describe a simple but reasonably accurate model of the memory system [38]. In this model, there are 2 kinds of memory. One is internal memory and the other is external memory (disk). The internal memory and CPU are very fast, and disk access is slow. In order to amortize this access time for a larger amount of data, the disk reads or writes a large collection of contiguous data items at once. The collection of contiguous data items is called a *block* [38, 7]. In order to model the behavior of the I/O system, we can capture the main properties of the memory system as follows [38].

- N = problem size (in units of data items),
- M = internal memory size (in units of data items),
- B = block transfer size (in units of data items),
- D = number of disk drives.

First we assume that $D = 1$ and there is just one CPU. We define a single I/O to be the process of reading or writing of a block (B contiguous items). The I/O complexity of an algorithm is the number of disk I/Os that the algorithm performs. We refer to $O\left(\frac{N}{B}\right)$ I/Os as "linear number of I/Os" in the external memory model with one disk.

We also consider an external memory model with multiple disks. Vitter and Shriver introduced a practical parallel disk model which has D independent disk drives [39]. In an I/O step, each of the D disks can transfer a block of size B simultaneously. When $T(N)$ is the number of disk I/Os for an external memory model with one disk, $T\left(\frac{N}{D}\right)$ is the optimal number of disk I/Os in the model with multiple disks.

2.2. Problem definitions

In this paper we consider five string problems. All of these problems have linear time algorithms in the internal memory model. We will present an efficient external memory algorithm for every problem. We assume that there is an ordering on the alphabet Σ. The notation $a \prec b$ means that character a is lexicographically smaller than character b. The notation $a \preceq b$ means that $a = b$ or $a \prec b$. The notations '$\prec$' and '$\preceq$' can be extended to strings in a similar way.

1 Maximum Suffix.
The maximum suffix of a string is the lexicographically largest suffix of the string. The maximum suffix problem is to find the maximum suffix of a given string. There are several ways that this computation can be done in internal memory. One may use suffix tree construction [28], suffix array construction [27], or factor automata construction [9]. Crochemore and Perrin [12] gave a simple and elegant linear-time algorithm for the maximum suffix problem.

2 String Matching.
The string matching problem is to find all occurrences of a pattern string in a text string. There are many string matching algorithms, e.g., Knuth-Morris-Pratt (KMP) and Boyer-Moore (BM) algorithms [26, 6]. In this paper we consider string matching algorithms that require only constant additional memory space [12, 11, 20, 21, 34].

3 Period Finding.
Let $T[1..N]$ be a string. We call an integer p $(1 \leq p \leq N)$ a period of T if $T[i] = T[i+p]$ for all $1 \leq i \leq N - p$. The shortest period of T is called *the period* of T and denoted by $per(T)$. The period finding problem is to find $per(T)$. There are some algorithms that find the period of a string [10, 11].

4 Lyndon Decomposition
For a given string T, the Lyndon decomposition is a unique decomposition $T = w_1 w_2 \cdots w_k$ with the following two properties. One is that the strings $w_1, w_2, \cdots, w_k$ are non-increasing in lexicographic order. The other is that each w_i is strictly less than any of its proper circular shift. Being circular means that string A of length n has n equivalent representations, namely $A[i..n]A[1..i-1]$ for $1 \leq i \leq n$, where $A[1..0]$ is the empty string. A proper circular shift $A[i..n]A[1..i-1]$ is a circular shift with $i \neq 1$. Duval [14] gave an algorithm that finds the Lyndon decomposition of a string. Smyth and Iliopoulos [25] gave an alternative algorithm for Lyndon decomposition.

5 Minimum of Circular String.
Let $T[1..N]$ be a string. The minimum-of-circular-string problem is to find the lexicographically smallest string among the N circular shifts of T. Shiloach [35] gave a linear-time algorithm that finds the minimum of a circular string.

3. Maximum Suffix

3.1. MS-decomposition

A string $T[1..N]$ has N suffixes. The maximum suffix of T is the lexicographically largest suffix among the N suffixes and is denoted by $max(T)$.

Let $v = max(T)$ and u be the string such that $T = uv$. The string v can be written as $w^e w'$ where $|w| = per(v),\ e \geq 1$, and w' is a prefix of w. The sequence (u, w, e, w') is called the *MS-decomposition* of T, where MS stands for maximum suffix. An MS-decomposition of $T = uw^e w'$ can be expressed by four integers (i, j, k, p) such that

$$i = |u|, j = |uw^e|, k = |w'|, p = |w|. \tag{1}$$

The 4-tuple (i, j, k, p) is called the *MS-tuple* of T.

Example 3.1. Let a string A be *bbccbccbc*. The maximum suffix of A is *ccbccbc* and its period is 3. So the MS-decomposition of A is $(bb, ccb, 2, c)$, and the MS-tuple of A is $(2, 8, 1, 3)$

3.2. Internal memory algorithm

Algorithm 1 is an internal memory maximum suffix algorithm, which is a modified version of the one in [12] so that its invariants can be clearly stated. It finds $max(T)$ of a given string T and the period of $max(T)$. Theorem 3.1 gives the main idea of Algorithm 1. For any string T and a character x, $max(Tx)$ can be computed by Theorem 3.1.

Theorem 3.1. [12] Let A be a string and (i, j, k, p) be the MS-tuple of A. Let x be a character and a character x' be $A[i + k + 1]$. Then we have

$$max(Ax) = \begin{cases} max(A)x, & \text{if } x' \succeq x \\ max(A[j + 1..j + k]x), & \text{if } x' \prec x. \end{cases}$$

Example 3.2. We show an example for Theorem 3.1. See Figure 1. For a given string $A = bbccbccbc$, the MS-tuple of A is (2,8,1,3) as shown in Example 3.1. In (a), the new character $x = a$ is smaller than $x' = c$. Therefore the maximum suffix of Ax becomes $max(A)x = ccbccbca$. In (b), the new character $x = d$ is larger than x'. Hence, $max(Ax)$ is equal to $max(cd)$, which is d in this case.

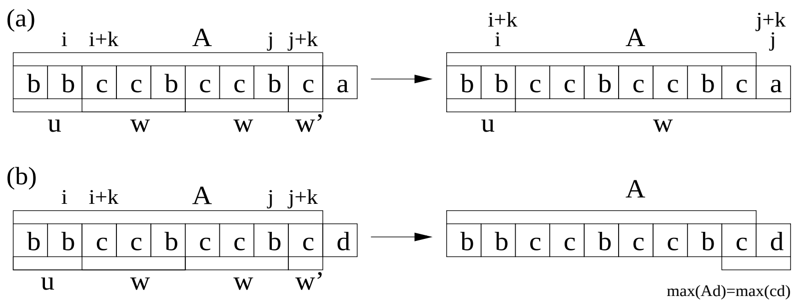

Figure 1. Example of Theorem 3.1

Algorithm 1 has four integers i, j, k, and p and has a series of iterations. After an iteration, Algorithm 1 maintains the following invariant.

- $max(T[i + 1..N]) = max(T)$
- $(0, j - i, k, p)$ is the MS-tuple of $T[i + 1..j + k]$ (i.e., $T[i + 1..j + k]$ is the maximum suffix of itself).

The initial values of i, j, k, and p are $0, 1, 0$, and 1, respectively. The suffix $T[i + 1..N]$ is the maximum suffix of T when $j + k$ becomes N.

Now we describe the details of an iteration. At the beginning of an iteration, we increase k by 1 and compare $T[i + k]$ and $T[j + k]$. According to the ordering between $T[i + k]$ and $T[j + k]$, three cases can occur. In each case, the variables i, j, k, and p are computed by Theorem 3.1.

In case 1, $T[i + k]$ and $T[j + k]$ are the same (i.e., the periodicity continues). If $k < p$, the variables i, j, k, and p are not changed. If $k = p$, the variables i, j, k, and p become $i, j + k, 0$, and p, respectively. Figure 2 shows case 1.

Algorithm 1 Maximum suffix algorithm

function MAXSUFFIX ($T[1..N]$)
$i = 0, j = 1, k = 0, p = 1$
for $j + k < N$ **do**
  $k = k + 1$
  if $T[i + k] = T[j + k]$ **then**
    if $k = p$ **then** $j = j + k, k = 0$ **fi**
  else if $T[i + k] \succ T[j + k]$ **then**
    $j = j + k, k = 0, p = j - i$
  else
    $i = j, j = i + 1, k = 0, p = 1$
  fi
od
($T[i + 1..N]$ is the maximum suffix of $T[1..N]$)

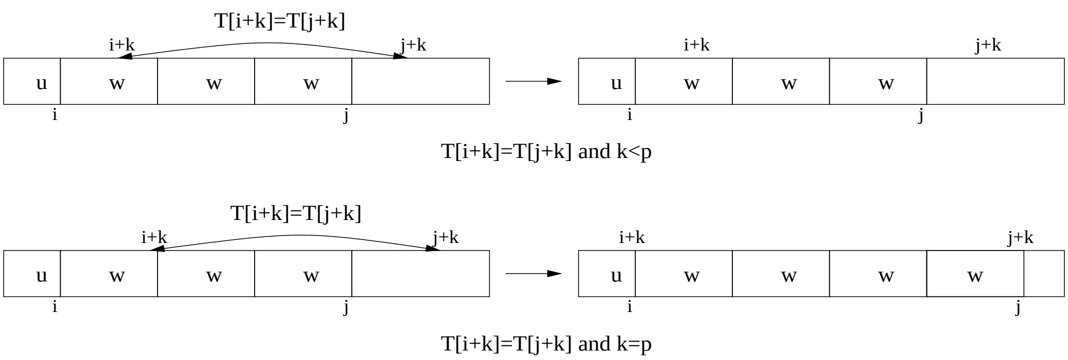

Figure 2. Case 1

In case 2, $T[i + k]$ is larger than $T[j + k]$. By Theorem 3.1, $T[i + 1..j + k]$ is the maximum suffix of $T[1..j + k]$ and $|T[i + 1..j + k]|$ becomes the period of itself. Therefore the variables i, j, k, and p become $i, j + k, 0$ and $j + k - i$, respectively. Figure 3 shows this situation.

In case 3, $T[i + k]$ is smaller than $T[j + k]$. In this case, $max(T)$ and $max(T[j + 1..N])$ are the same by Theorem 1. Thus, Algorithm 1 finds the maximum suffix of $T[j + 1..N]$ instead of that of T. The variables i, j, k, and p become $j, j + 1, 0$, and 1, respectively. See Figure 4. Note that in case 1 with $k = p$ and case 2, the value of $i + k$ decreases by k, and the value of $j + k$ decreases by $k - 1$ in case 3.

After an iteration, the variable $i + j + k$ increases by at least 1. Because $i + j + k$ cannot be larger than $2N$, the time complexity of Algorithm 1 is $O(N)$ in internal memory.

3.3. External memory algorithm

For a string T of length N, suppose that T is composed of continuous blocks in external memory, that is $T = b_1 b_2 \cdots b_{\lceil \frac{N}{B} \rceil}$. Algorithm 1 runs in linear time in internal memory, but it is not an efficient algorithm in external memory. It was shown in [32] that the number of disk I/Os of Algorithm 1 can be $O(N)$.

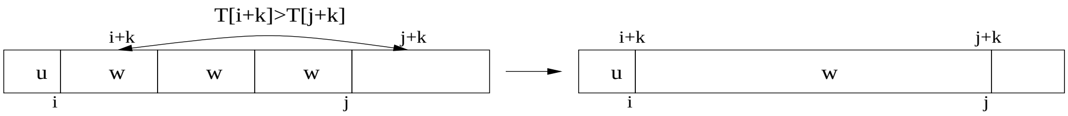

Figure 3. Case 2

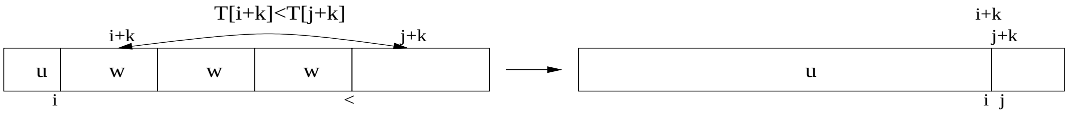

Figure 4. Case 3

We present an external memory algorithm which will be called EMMS (external memory maximum suffix) algorithm. Assume that the number of disks is one. We consider an external memory model with multiple disks later. EMMS maintains four memory blocks b_I, b_{I+K}, b_J, and b_{J+K}. The initial value of each of them is the first block of T. While EMMS runs, b_I and b_J always have the blocks which are accessed by i and j, respectively. b_{I+K} and b_{J+K} have the blocks which are accessed by $i+k$ and $j+k$, respectively, with one exception described below.

We now describe the details of an iteration of EMMS. At the beginning of an iteration, variable k increases by 1. If $i+k$ accesses the next block of b_{I+K} after the increase of k, EMMS reads the next block of b_{I+K} to b_{I+K}. Similarly, EMMS maintains b_{J+K}.

If case 1 with $k = p$ or case 2 occurs, the values $i+k$ and j become i and $j+k$ respectively. When $i+k$ decreases, there are three cases depending on the relation between b_I and b_{I+K}. Case X is that b_I and b_{I+K} are the same. Case Y is that b_{I+K} is the next block of b_I. Case Z is that b_{I+K} is at least one block apart from b_I. When case X or case Z occur, b_{I+K} gets the block which is accessed by $i+k$. However, if case Y occurs, b_{I+K} keeps the next block of b_I (i.e., the current block of b_{I+K}). This is the one exception mentioned above. In case 1 with $k < p$, EMMS does nothing.

In case 3, i and $i+k$ increase to j, and j and $j+k$ become $j+1$. Thus, b_I and b_{I+K} get the current block of b_I. For b_J and b_{J+K}, first b_J gets the block accessed by $j+1$ if it is not the current block of b_J. Now the change for b_{J+K} is the same as that of b_{I+K}, i.e., there are three cases depending on the relation between b_J and b_{J+K}.

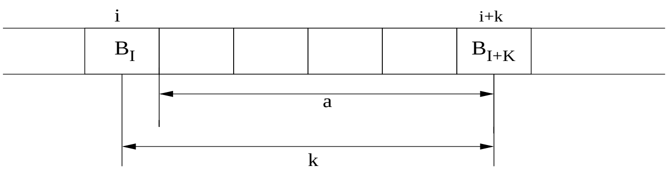

Figure 5. Case Z

Now we compute the number of disk I/Os. First we compute the maximum number of disk I/Os for i and $i+k$. Because b_I gets the block of b_J when i increases, i does not need disk I/Os. So we count disk I/Os for $i+k$ only. The variable $i+k$ can increase from 1 to $N-1$ and it can decrease on the way. If case X or case Y occurs, no disk I/O is needed until $i+k$ becomes equal to the current value of $i+k$ before it decreases. For case Z, see Figure 5. Let an integer a be $i+k$ minus the position of the last character of b_I. If case Z occurs, at most $\lceil \frac{a}{B} \rceil$ memory blocks can be reread until $i+k$ becomes the current value of itself. Hence, the maximum number of disk I/Os for $i+k$ is $\lceil \frac{N-1}{B} \rceil + \sum \lceil \frac{a}{B} \rceil$ for all occurrences of case Z. Let an integer α be the sum of all k's of case Z, and let an integer β be the number of case Z's. We get $\lceil \frac{N-1}{B} \rceil + \sum \lceil \frac{a}{B} \rceil \leq \lceil \frac{N-1}{B} \rceil + \sum(\lfloor \frac{a}{B} \rfloor + 1) \leq \lceil \frac{N-1}{B} \rceil + \lfloor \sum \frac{a}{B} \rfloor + \beta \leq \lceil \frac{N-1}{B} \rceil + \lfloor \sum \frac{k}{B} \rfloor + \beta \leq \lceil \frac{N-1}{B} \rceil + \lfloor \frac{\alpha}{B} \rfloor + \beta$. The value of β is at most $\lfloor \frac{\alpha}{B+1} \rfloor$ because k is larger than B when case Z occurs. Since α is at most $N-1$ because j increases by k when $i+k$ decreases by k and j is at most $N-1$, the maximum number of disk I/Os for $i+k$ becomes $2\lceil \frac{N-1}{B} \rceil + \lfloor \frac{N-1}{B+1} \rfloor \leq 3\lceil \frac{N}{B} \rceil$. Similarly, we can compute the maximum number of disk I/Os for j and $j+k$. It is also bounded by $3\lceil \frac{N}{B} \rceil$. Thus, the I/O complexity of EMMS is $6\lceil \frac{N}{B} \rceil$.

Now we present a modified version of EMMS which will be called mEMMS (modified EMMS). Its I/O complexity is $4\lceil \frac{N}{B} \rceil$. The mEMMS algorithm maintains six memory blocks $b_I, b_{I+K}, b_J, b_{J+K}, b_{I+B}$, and b_{J+B}. While mEMMS runs, b_I, b_{I+K}, b_J, and b_{J+K} always have the blocks which are accessed by $i, i+k, j$, and $j+k$, respectively. The block b_{I+B} may have the next block of b_I or is $null$. Similarly, b_{J+B} may have the next block of b_J or is $null$.

We now explain how mEMMS maintains b_{I+B}. After an iteration, b_{I+B} maintains the following invariant. If $i+k$ accesses a block which is not b_I (i.e., $i+k$ accesses a block to the right of b_I), b_{I+B} has the next block of b_I. Otherwise, b_{I+B} is $null$.

The initial value of b_{I+B} is $null$ and that of $i+k$ is 0. While i keeps the current value, $i+k$ can increase only by 1 when it increases. Thus, the first block which is accessed by $i+k$ and is not b_I, must be the next block of b_I. b_{I+B} gets the next block of b_I when $i+k$ accesses it, and keeps it until i increases and accesses a block which is not the current block of b_I. When i accesses a block which is not b_I, b_{I+B} becomes $null$. The invariant for b_{I+B} is satisfied. Similarly, mEMMS maintains b_{J+B}.

We compute the number of disk I/Os. When case Z occurs, b_{I+B} always has the next block of b_I. Therefore in case Z, at most $\lceil \frac{a}{B} \rceil - 1$ memory blocks are reread until $i+k$ becomes the current value. Thus, the maximum number of disk I/Os for $i+k$ is $\lceil \frac{N-1}{B} \rceil + \sum(\lceil \frac{a}{B} \rceil - 1)$ for all occurrences of case Z. Hence, we get $\lceil \frac{N-1}{B} \rceil + \sum(\lceil \frac{a}{B} \rceil - 1) \leq \lceil \frac{N-1}{B} \rceil + \sum \lfloor \frac{a}{B} \rfloor \leq \lceil \frac{N-1}{B} \rceil + \lfloor \sum \frac{a}{B} \rfloor \leq \lceil \frac{N-1}{B} \rceil + \lfloor \sum \frac{k}{B} \rfloor \leq \lceil \frac{N-1}{B} \rceil + \lfloor \frac{\alpha}{B} \rfloor \leq 2\lceil \frac{N}{B} \rceil$. Similarly, the maximum number of disk I/Os for j is also bounded by $2\lceil \frac{N}{B} \rceil$. Thus, the I/O complexity of mEMMS is $4\lceil \frac{N}{B} \rceil$.

Now we consider an external memory model with multiple disks. *Disk striping* is a practical paradigm with multiple disks [37, 38]. Figure 6 shows an example with $D = 5$ and $B = 2$. The input data items are striped across the disks, and I/Os are permitted only on entire stripes, one stripe at a time. For example, data items 10 to 19 can be accessed in one disk I/O time. Note that the effect of disk striping is that the multiple disks act as a single disk with a block of size DB.

The disk striping paradigm can be applied to mEMMS easily. Because mEMMS maintains six memory blocks of size DB and four integers, we assume that the internal memory size M is large enough to have them. Then, the I/O complexity becomes $4\lceil \frac{N}{DB} \rceil$. This is an optimal I/O complexity with the external disk model with multiple disks.

	D_0		D_1		D_2		D_3		D_4	
stripe 0	0	1	2	3	4	5	6	7	8	9
stripe 1	10	11	12	13	14	15	16	17	18	19
stripe 2	20	21	22	23	24	25	26	27	28	29
stripe 3	30	31	32	33	34	35	36	37	38	39

The data layout on the multiple disks with $D = 5$ and $B = 2$.

Figure 6. Disk striping

4. String Matching

Many string matching algorithms find the occurrences of a pattern P inside a text T by considering increasing positions in T. At each position met during the execution of an algorithm, a scan is done to decide whether the pattern occurs there or not, and a shift is made. Thus, these kinds of algorithms perform a series of scans and shifts, and so we refer to them as *scan-and-shift algorithms*. For instance, the Knuth-Morris-Pratt (KMP for short) is a scan-and-shift algorithm [26]. The scan of the pattern against the text at a given position can be realized in several ways. But a scan from left to right is certainly most natural.

KMP maintains an array (failure function) whose size is proportional to the length of P. Whenever KMP meets a mismatch while it performs a scan, it accesses the array. Thus, if the array is too big to be stored in internal memory and mismatches occur frequently, KMP cannot be efficient in external memory.

The scan-and-shift algorithm in Figure 7 scans T and P, and it stops when it meets a mismatch. Let a string A be $P[1..x]$ when it meets a mismatch after x matches and b be the text character at the position where a mismatch occurs. In this situation, the best length of a shift is $per(Ab)$. But the pre-computation of all periods leads to the same problem as KMP has. Hence, we will compute an approximation of $per(Ab)$ without pre-computation.

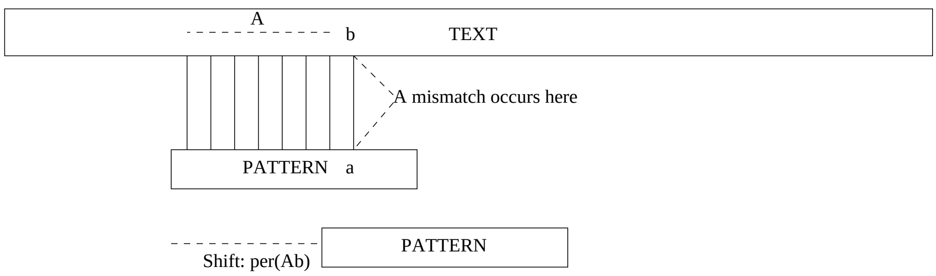

Figure 7. Scan-and-shift algorithm

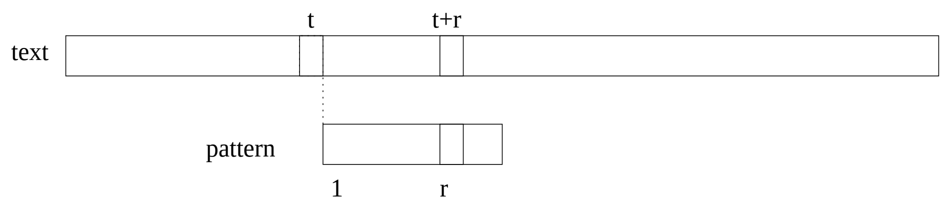

Figure 8. Variables t and r

4.1. String matching using maximum suffix

We present an external memory string matching algorithm, which will be called the *EMSM* (external memory string matching) algorithm. EMSM has a series of iterations which consist of a scan and a shift just like other scan-and-shift algorithms. A shift in an iteration consists of three steps. Consider a string Ab as in Figure 7. The first step is to compute the MS-decomposition of Ab. The second step is to compute the length of the shift. In the third step, the pattern is shifted to the right.

EMSM maintains the following variables and memory blocks. The variables t and r are the pointers on T and P as shown in Figure 8. Two pointers $t+r$ and r are used when EMSM performs a scan. The block b_T has the block which is accessed by t in T. Two blocks b_{T+R} and b_R basically have the block accessed by $t+r$ and r in T and P, respectively, with one exception described below. Moreover, a memory block b_{R-B} is used, which has the previous block of b_R or is $null$. The reason why EMSM maintains this block is to avoid the problem that a variable accesses two adjacent blocks repeatedly. The way that EMSM maintains b_{R-B} is described below. In the second step of the shift, EMSM uses two additional memory blocks.

Now we describe an iteration. EMSM performs a scan to the right and compares $T[t+r]$ and $P[r]$. During the scan, if r accesses the next block of b_R, b_{R-B} gets the current block of b_R and b_R gets the next block of b_R. The scan is stopped when $|P|$ matches are found or a mismatch occurs. If $|P|$ matches are found (i.e., the pattern P occurs at the position $t+1$ of T), let a string V be $T[t+1..t+|P|+1]$. If a mismatch between $T[t+r]$ and $P[r]$ (i.e., $r-1$ matches occur), let V be $T[t+1..t+r]$. The string V is the same as Ab in Figure 7.

After the scan, EMSM performs a shift. For the first step of the shift, EMSM computes the MS-decomposition of V using EMMS or mEMMS. Let uw^ew' and (i,j,k,p) be the MS-decomposition and the MS-tuple of V.

In the second step of the shift, EMSM computes the length of the shift which is an approximation of $per(V)$ [11]. If u is a suffix of w, the length of the shift is $per(X) = |w|$. Otherwise, it is $\max(|u|, \min(|w^ew'|, |uw^e|))$. The value of $\max(|u|, \min(|w^ew'|, |uw^e|))$ is not the exact value of $per(V)$ but it is equal to or larger than a half of the length of V [11]. EMSM performs a test whether u is a suffix of w or not when $|u| < |w|$ from the positions $t+i$ and $t+j$ to the left. During the test, two memory blocks, one for u and the other for w, are used.

In the third step, the pattern is shifted and t increases by the length of the shift. If u is a suffix of w, EMSM doesn't make comparisons between $T[t+1..t+|V|-|w|]$ and $P[1..|V|-|w|]$, which are of the form $uw^{e-1}w'$ after the shift [11]. The variables and the blocks are updated as follows. The variable r decreases to $|X|-|w|+1$. If u is not a suffix of w, the variable r becomes 1. After the updates,

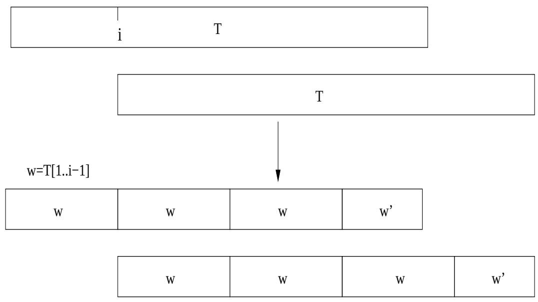

Figure 9. An Overhanging Occurrence and a Period

if r accesses b_{R-B} or b_R after the decrease, b_R keeps the current block. This is the exception for b_R mentioned above. Otherwise, b_R gets the block which is accessed by r and b_{R-B} becomes $null$. If $t+r$ accesses b_T and b_{T+R} is the next block of b_T, b_{T+R} keeps the current block. This is the exception for b_{T+R} mentioned above and is similar to that for b_{I+B} in EMMS.

Now we compute the number of disk I/Os. The variable t needs $\lceil \frac{N}{B} \rceil$ disk I/Os because it monotonically increases. Consider the variable $t+r$. When u is a suffix of w in an iteration, $t+r$ can decrease by at most $\lfloor \frac{|V|}{2} \rfloor$. The number of disk I/Os for $t+r$ is linear by a similar argument that the number of disk I/Os for $i+k$ is linear in EMMS. To reduce the number of disk I/Os, the technique for $i+k$ in mEMMS can be used for $t+r$ (i.e., EMSM keeps the next block of b_T using an additional block). Similarly, we can show that the number of disk I/Os for r is linear. During the second step of shift, the number of disk I/Os is at most $\lfloor \frac{2|u|}{B} \rfloor$. Because $|u|$ is smaller than the length of the shift and the sum of the lengths of shifts of all iterations is smaller than N, the number of disk I/Os for the second step is linear. We can also show that the number of disk I/Os for the computation of the MS-decomposition is also linear. Hence, the I/O complexity of EMSM is linear.

5. Period Finding

In this section we consider the problem of computing the period of a given string. When p is a period of string T of length N, the substring $T[1..N-p]$ is both a proper prefix and suffix of T. Therefore if EMSM meets $N-i+1$ consecutive matches when it performs a scan at the position i, $i-1$ is a period of T. Figure 9 illustrates.

Therefore we can modify the EMSM algorithm to find all periods of T. The I/O complexity for period finding is the same as that of EMSM.

6. Lyndon Decomposition

Any string T can be written uniquely $T = w_1 w_2 \cdots w_n$ with the following two properties.

1. the strings $w_1, w_2, \cdots, w_n$ are non-increasing in lexicographic order.
2. each w_i $(1 \le i \le n)$ is strictly less than any of its proper circular shifts.

We call this decomposition the Lyndon decomposition and w_i the Lyndon word. Example 6.1 shows some examples of Lyndon words and Lyndon decomposition.

Example 6.1. Consider two strings $aabbc$ and $abaab$. The first string is a Lyndon word because every proper circular shift is larger than $aabbc$. The second string is not a Lyndon word because $aabab$ is less than $abaab$. The Lyndon decomposition of $T = cbbcbbbaab$ is *c*, *bbc*, *b*, *b*, *b*, *aab*.

A Lyndon word has the following property.

Lemma 6.1. [14] T is a Lyndon word if and only if for each proper suffix w of T, one has $w \succ T$.

In this section we describe an external memory algorithm to find the Lyndon decomposition of a string which is called the *EMLD* (external memory Lyndon decomposition) algorithm. We slightly modify the EMMS algorithm to make the EMLD algorithm. We reverse the signs in case 2 and case 3 of EMMS. That is, we change '$\succ$' in case 2 and '$\prec$' in case 3 to '$\prec$' and '$\succ$', respectively. The EMLD algorithm outputs Lyndon words during the computation.

While EMLD runs, it maintains the Lyndon decomposition of $T[1..j]$. The Lyndon decomposition of $T[1..j]$ consists of two parts. One is the Lyndon decomposition of $T[1..i]$ and the other is that of $T[i+1..j]$. EMLD maintains the following invariants.

- Every Lyndon word of the Lyndon decomposition of $T[1..i]$ is also a Lyndon word of the Lyndon decomposition of T. That is, every Lyndon word of the Lyndon decomposition of $T[1..i]$ is not changed once it is determined.
- The Lyndon decomposition of $T[i+1..j]$ has the form of $T[i+1..i+p]^{\frac{j-i}{p}}$ if $T[i+1..i+p]$ is a Lyndon word.

The initial values of all variables are the same as those for EMMS. Case 1 is the same as that of EMMS. In case 2, $T[i+k]$ is smaller than $T[j+k]$. The string $T[i+1..j+k]$ becomes a Lyndon word of the Lyndon decomposition of $T[i+1..j+k]$.

In case 3, $T[i+k]$ is larger than $T[j+k]$. In this case, every prefix of $T[j+1..N]$ is smaller than $T[i+1..i+p]$. The Lyndon words $T[i+1..i+p]$, $T[i+p+1..i+2p]$, ..., $T[j-p+1..j]$ become the Lyndon words of the Lyndon decomposition of T [14]. EMLD adds them into the Lyndon decomposition of $T[1..i]$. The I/O complexity of EMLD is the same as that of mEMMS.

Now we describe the increasing patterns of i and j in EMLD that will be used in Section 7. The Lyndon decomposition of $T = w_1 w_2 \cdots w_n$ can be rewritten as $T = {v_1}^{e_1} {v_2}^{e_2} \cdots {v_m}^{e_m}$ with the following properties.

1. the strings $v_1, v_2, \cdots, v_m$ are decreasing in lexicographic order.
2. each v_i $(1 \le i \le m)$ is a Lyndon word.
3. $e_i \ge 1$ $(1 \le i \le m)$.
4. $\sum_{i=1}^{m} e_i = n$.

Example 6.2. The Lyndon decomposition of $T = cbbcbbbaab$ is c, bbc, b, b, b, aab. Thus, v_1, v_2, v_3 and v_4 are c, bbc, b, and aab, respectively, and e_1, e_2, e_3, and e_4 are $1, 1, 3$, and 1, respectively.

Suppose that $T = {v_1}^{e_1}{v_2}^{e_2} \cdots {v_m}^{e_m}$ is the Lyndon decomposition of T. While EMLD runs, i becomes 0, $|{v_1}^{u_1}|$, $|{v_1}^{u_1}{v_2}^{u_2}|$, $|{v_1}^{u_1}{v_2}^{u_2}{v_3}^{u_3}|$, and so on. When i is $|{v_1}^{u_1} \cdots {v_{i'}}^{u_{i'}}|$, the variable j becomes $|{v_1}^{u_1} \cdots {v_{i'}}^{u_{i'}}|+1$. Next, j increases and becomes $|{v_1}^{u_1} \cdots {v_{i'}}^{u_{i'}} v_{i'+1}|$. After that, j repeatedly increases by $|v_{i'+1}|$ until j becomes $|{v_1}^{u_1} \cdots {v_{i'}}^{u_{i'}} {v_{i'+1}}^{u_{i'+1}}|$, and then i increases.

7. Minimum of Circular String

In this section we deal with circular strings. For a string T of length N, let T_i denote a circular shift $T[i..N]T[1..i-1]$. T_{i_0} is minimum if $T_{i_0} \preceq T_i$ for all $1 \leq i \leq N$. We call i_0 a *minimum starting point* or more simply an *msp* of T. We present an external memory algorithm to find the *msp*s of a given string, which will be called the *EMMC* (external memory minimum of a circular string) algorithm.

T has q (≥ 1) *msp*s if and only if it consists of q equal substrings of length N/q. Let S be the shortest prefix of T such that $T = S^q$. Let $msp(T)$ denote the smallest *msp* of T. If $q > 1$, $msp(T)+|S|\times j$ $(0 \leq j \leq q-1)$ are the *msp*s of T. Also, $msp(S)$ and $msp(T)$ are the same and $S_{msp(S)}$ is a Lyndon word.

Example 7.1. Suppose that $T = abaaabaaabaa$. Then $S = abaa$ and $msp(T) = msp(S) = 3$. Hence, $3, 7$, and 11 are the *msp*s of T.

7.1. Algorithm

From Lemma 6.1 and the definition of an *msp*, we can get the following theorem. Let $T = S^q$ for $q \geq 1$, i.e., T^2 has $2q$ *msp*s.

Theorem 7.1. Every *msp* of T^2 points to the first character of a Lyndon word in the Lyndon decomposition of T^2.

Proof:
Suppose that an *msp* does not point to the first character of a Lyndon word. Let an *msp* of T^2 point to a character of a Lyndon word w that is not the first character of w. See Figure 10. By definition of *msp*, v is equal to or smaller than the prefix w' of w of length $|v|$. Then, v is smaller than w because v is a prefix of w when $v = w'$. But v must be larger than w by Lemma 6.1. This is a contradiction.

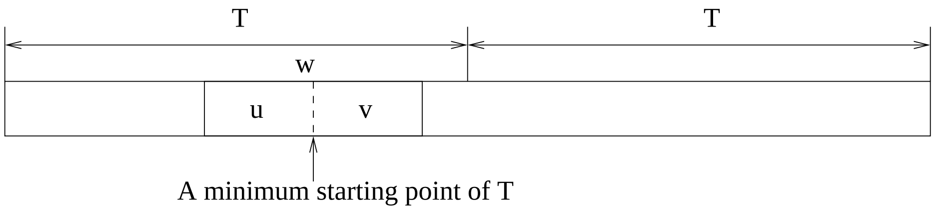

Figure 10. Minimum starting point and Lyndon decomposition

□

Now we show the form of the Lyndon decomposition of T^2 for a given string $T = S^q$.

Theorem 7.2. For $T = S^q$, the Lyndon decomposition of T^2 consists of those of the following three strings. And the Lyndon decomposition of (2) consists of $2q - 1$ equal Lyndon words of length $|S|$.

(1) $T^2[1..msp(T^2) - 1] = S[1..msp(S) - 1]$

(2) $T^2[msp(T^2)..|T^2| - |S| + msp(T^2) - 1] = (S_{msp(S)})^{2q-1}$

(3) $T^2[|T^2| - |S| + msp(T^2)..|T^2|] = S[msp(S)..|S|]$

Proof:

From Theorem 7.1, a Lyndon word that begins in (1) and ends in (2) cannot exist. Similarly, a Lyndon word that begins in (2) and ends in (3) does not exist. Thus, every Lyndon word in the Lyndon decompositions of (1), (2), and (3) is a Lyndon word of T^2. Therefore, the Lyndon decomposition of T^2 consists of those of (1), (2), and (3).

Now we show that the Lyndon decomposition of (2) consists of $2q-1$ equal Lyndon words of length $|S|$. Let a string L be $T^2[msp(T^2) + |S| * i..msp(T^2) + |S| * (i + 1) - 1]$ for any $0 \le i \le 2q - 2$. The string $T^2 = S^{2q}$ has $2q$ *msp*s and the distance between two consecutive *msp*s is $|S|$ [35]. Thus, L has one *msp* of T^2 because of $|S| = |L|$. By definition of L, the position of the first character of L is an *msp* of T^2. Therefore, L is equal to or smaller than every substring of T^2 of length $|L|$, Thus, L is the least circular shift of itself, and is a Lyndon word. Because (2) is a concatenation of $2q - 1$ equal strings which is L, the Lyndon decomposition of (2) consists of $2q - 1$ equal Lyndon words that are $|L|$.

□

Example 7.2. We show an example for Theorem 7.2. See Figure 11. The input string T is $abaaabaaabaa = (abaa)^3$. In this example, (1), (2), and (3) are ab, $aaabaaabaaabaaabaaab = (aaab)^5$, aa, respectively. And L is $aaab$. The Lyndon decomposition of (2) consists of $2q - 1 = 5$ equal Lyndon words, that is $aaab$ in this example.

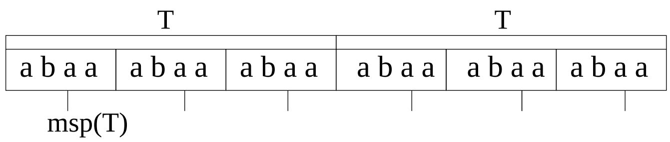

Figure 11. Example for Theorem 7.2

In Theorem 7.2, the length of (2) is at least $|T| = N$, because of $(2q - 1) \times |S| = 2N - |S| \ge N$. From the increasing pattern of variables i and j in Section 6, we know that when $EMLD$ computes the Lyndon words in (2), variable i increases by the length of (2). At that time the situation of $j - i \ge N$ occurs, and it can occur only once. Therefore the $EMLD$ algorithm can find (2) for a given string T^2. And it can compute the *msp*s of T.

8. Conclusion

In this paper, we have presented external memory algorithms for five string problems: finding maximum suffix, string matching, period finding, Lyndon decomposition, and finding the minimum of a circular string. These are basic problems in string algorithmics, and more involved problems can be considered for developing external memory algorithms.

References

[1] Aggarwal, A., Vitter, J. S.: The input/output complexity of sorting and related problems, *Communications of the ACM*, **31**(9), 1988, 1116–1127, ISSN 0001-0782.

[2] Apostolico, A., Crochemore, M.: Fast parallel Lyndon factorization with applications., *Mathematical Systems Theory*, **28**(2), 1995, 89–108.

[3] Arge, L., Ferragina, P., Grossi, R., Vitter, J. S.: On sorting strings in external memory (extended abstract), *STOC '97: Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, ACM Press, New York, NY, USA, 1997, ISBN 0-89791-888-6.

[4] Bender, M. A., Demaine, E. D., Farach-Colton, M.: Cache-oblivious B-trees, *FOCS '00: Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, IEEE Computer Society, Washington, DC, USA, 2000, ISBN 0-7695-0850-2.

[5] Booth, K. S.: Lexicographically least circular substrings., *Inf. Process. Lett.*, **10**(4/5), 1980, 240–242.

[6] Boyer, R. S., Moore, J. S.: A fast string searching algorithm, *Communications of the ACM*, **20**(10), 1977, 762–772, ISSN 0001-0782.

[7] Chiang, Y.-J., Goodrich, M. T., Grove, E. F., Tamassia, R., Vengroff, D., Vitter, J.: External-memory graph algorithms, *SODA '95: Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1995, ISBN 0-89871-349-8.

[8] Clark, D. R., Munro, J. I.: Efficient suffix trees on secondary storage, *SODA '96: Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1996, ISBN 0-89871-366-8.

[9] Crochemore, M.: Transducers and repetitions, *Theor. Comput. Sci.*, **45**(1), 1986, 63–86, ISSN 0304-3975.

[10] Crochemore, M.: String matching and periods, *Theor. Comput. Sci.*, **39**(1), 1989, 63–86, ISSN 0304-3975.

[11] Crochemore, M.: String-matching on ordered alphabets, *Theor. Comput. Sci.*, **92**(1), 1992, 33–47, ISSN 0304-3975.

[12] Crochemore, M., Perrin, D.: Two-way string-matching, *J. ACM*, **38**(3), 1991, 650–674, ISSN 0004-5411.

[13] Crochemore, M., Rytter, W.: *Jewels of Stringology*, World Scientific, 2002.

[14] Duval, J.-P.: Factoring words over an ordered alphabet, *Journal of Algorithms*, **4**(4), 1983, 363–381, ISSN 0304-3975.

[15] Farach, M., Ferragina, P., Muthukrishnan, S.: Overcoming the memory bottleneck in suffix tree construction, *FOCS '98: Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society, Washington, DC, USA, 1998, ISBN 0-8186-9172-7.

[16] Ferragina, P., Grossi, R.: Fast string searching in secondary storage: theoretical developments and experimental results, *SODA '96: Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1996, ISBN 0-89871-366-8.

[17] Ferragina, P., Grossi, R.: The string B-tree: a new data structure for string search in external memory and its applications, *J. ACM*, **46**(2), 1999, 236–280, ISSN 0004-5411.

[18] Ferragina, P., Luccio, F.: On the parallel dynamic dictionary matching problem: new results with applications, *ESA '96: Proceedings of the 4th Annual European Symposium on Algorithms*, Springer-Verlag, London, UK, 1996, ISBN 3-540-61680-2.

[19] Feuerstein, E., Marchetti-Spaccamela, A.: Memory paging for connectivity and path problems in graphs, *ISAAC '93: Proceedings of the 4th International Symposium on Algorithms and Computation*, Springer-Verlag, London, UK, 1993, ISBN 3-540-57568-5.

[20] Galil, Z., Seiferas, J.: Time-space-optimal string matching (preliminary report), *STOC '81: Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, ACM Press, New York, NY, USA, 1981.

[21] Gasieniec, L., Kolpakov, R. M.: Real-time string matching in sublinear space., *CPM*, 2004.

[22] Gonnet, G. H., Baeza-Yates, R.: *Handbook of Algorithms and Data Structures: in Pascal and C (2nd ed.)*, Addison-Wesley Longing Publishing Co., Inc., Boston, MA, USA, 1991, ISBN 0-201-41607-7.

[23] Gusfield, D.: *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press, New York, 1997.

[24] Iliopoulos, C. S., Smyth, W. F.: Optimal algorithms for computing the canonical form of a circular string, *Theor. Comput. Sci.*, **92**(1), 1992, 87–105, ISSN 0304-3975.

[25] Iliopoulos, C. S., Smyth, W. F.: A fast average case algorithm for Lyndon decomposition, *Internat. J. Computer Math*, **57**, 1995, 15–31, ISSN 0004-5411.

[26] Knuth, D. E., Morris, J. H., Pratt, V. R.: Fast pattern matching in strings, *SIAM J. Comp*, **6**(2), 1977, 323–350, ISSN 0004-5411.

[27] Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches, *SODA '90: Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1990, ISBN 0-89871-251-3.

[28] McCreight, E. M.: A space-economical suffix tree construction algorithm, *J. ACM*, **23**(2), 1976, 262–272, ISSN 0004-5411.

[29] Nodine, M. H., Goodrich, M. T., Vitter, J. S.: Blocking for external graph searching, *PODS '93: Proceedings of the 12th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, ACM Press, New York, NY, USA, 1993, ISBN 0-89791-593-3.

[30] Nodine, M. H., Vitter, J. S.: Paradigms for optimal sorting with multiple disks, *Proceedings of the 26th Hawaii Int. Conf. on Systems Sciences*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1993, ISBN 0-89871-251-3.

[31] Rajsbaum, S., Ed.: *LATIN 2002: Theoretical Informatics, 5th Latin American Symposium, Cancun, Mexico, April 3-6, 2002, Proceedings*, vol. 2286 of *Lecture Notes in Computer Science*, Springer, 2002, ISBN 3-540-43400-3.

[32] Roh, K., Crochemore, M., Iliopoulos, C. S., Park, K.: External memory algorithms for string problems., *Proceedings of the 17th Australasian Workshop on Combinatorial Algorithms*, 2006.

[33] Ruemmler, C., Wilkes, J.: An introduction to disk drive modeling, *Computer*, **27**(3), 1994, 17–28, ISSN 0018-9162.

[34] Rytter, W.: On maximal suffixes and constant-space linear-time versions of KMP algorithm, *LATIN 2002: Proceedings of the 5th Latin American Symposium on Theoretical Informatics*, Springer-Verlag, London, UK, 2002, ISBN 3-540-43400-3.

[35] Shiloach, Y.: Fast canonization of circular strings, *J. Algorithms*, **2**, 1981, 107–121, ISSN 0004-5411.

[36] Ullman, J. D., Yannakakis, M.: The input/output complexity of transitive closure, *SIGMOD '90: Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, ACM Press, New York, NY, USA, 1990, ISBN 0-89791-365-5.

[37] Vitter, J. S.: External memory algorithms, *PODS '98: Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, ACM Press, New York, NY, USA, 1998, ISBN 0-89791-996-3.

[38] Vitter, J. S.: External memory algorithms and data structures: dealing with massive data, *ACM Comput. Surv.*, **33**(2), 2001, 209–271, ISSN 0360-0300.

[39] Vitter, J. S., Shriver, E. A. M.: *Algorithms for parallel memory I: two-level memories*, Technical report, Providence, RI, USA, 1992.