DOI: 10.3745/KIPSTA.2008.15-A.4.239

스트링의 최대 서픽스를 계산하는 효율적인 외부 메모리 알고리즘

김 성 권[†] · 김 수 철[††] · 조 정 식[††]

요 약

외부 메모리 계산 모델에서 스트링의 최대서픽스를 찾는 문제를 고려한다. 외부메모리 모델에서서는 디스크와 내부메모리 사이의 디스크 입출력 횟수를 줄이는 알고리즘을 설계하는 것이 중요 사항이다. 길이가 N인 스트링은 N개의 서픽스를 가지는데, 이중에서 사전 순서에 따라 가장 큰 것을 최대 서픽스라 부른다. 최대서픽스를 구하는 것은 여러 스트링 문제를 해결하는 데 중요한 역할을 한다.

본 논문에서는 길이가 N인 스트링의 최대 서픽스를 구하는 외부메모리 알고리즘을 제시한다. 이 알고리즘은 네 개의 내부 메모리 블록을 사용하고 최대 4(N/L)번의 디스크 입출력을 한다. 여기서 L은 블록의 크기이다.

키워드 : 스트링, 최대 서픽스, 외부메모리 알고리즘

Efficient External Memory Algorithm for Finding the Maximum Suffix of a String

Sung Kwon Kim[†] · Soo-Cheol Kim[††] · Jung-Sik Cho[††]

ABSTRACT

We study the problem of finding the maximum suffix of a string on the external memory model of computation with one disk. In this model, we are primarily interested in designing algorithms that reduce the number of I/Os between the disk and the internal memory. A string of length N has N suffixes and among these, the lexicographically largest one is called the maximum suffix of the string. Finding the maximum suffix of a string plays a crucial role in solving some string problems.

In this paper, we present an external memory algorithm for computing the maximum suffix of a string of length N. The algorithm uses four blocks in the internal memory and performs at most $4(N/L)$ disk I/Os, where L is the size of a block.

Keyword : External Memory Algorithms, Maximum Suffix, Strings

1. Introduction

Let Σ be an alphabet. Let s and t be two strings over Σ. If s is lexicographically less than t, we denote this by $s < t$. Let $t = t_1 \dots t_N$, where N is the length of t. For $1 \le a \le b \le N$, $t_a \dots t_b$ is a substring of t. The substrings with $a = 1$ are called prefixes and those with $b = N$ are called suffixes. t has N suffixes. Among these the lexicographically largest one is called the maximum suffix of t, denoted $ms(t)$.

The maximum suffix of a string can be found in linear time [2, 3]. Finding the maximum suffix of a string is a key operation in solving the following four string problems: string matching, period finding, computing the minimum of a circular string, and Lyndon decomposition [7]. The string matching is to find the occurrences of a pattern in a text, and the Knuth-Morris-Pratt algorithm [6] is one of the well-known algorithms for the problem. A string ω is said to be the period of another string t if $t = \omega^e \omega'$, where ω' is a prefix of ω and ω is as short as possible. In other words, repeating e copies of ω and appending ω' after it results in t. The period of a string can be computed in linear time [2]. String t has N circular shifts, namely $t_i \dots t_N t_1 \dots t_{i-1}$ for $1 \le i \le N$. The minimum of a circular string is the lexicographically smallest one among these circular shifts. Shiloach [8] gives a linear time algorithm for the problem. The Lyndon decomposition decomposes the string t into $t = w_1 w_2 \dots w_n$, where the strings $w_1, w_2, \dots w_n$ are lexicographically non-increasing and each w_i $(1 \le i \le n)$ is strictly less than any of its circular shifts except for w_i itself. The Lyndon decomposition of a string can be found by the algorithm due to Duval [5].

Roh et al. [7] presents external memory algorithms for the

※ 이 논문은 2007년도 중앙대학교 학술연구비 지원에 의한 것임.
† 종신회원 : 중앙대학교 컴퓨터공학과 교수
†† 정 회 원 : 중앙대학교 컴퓨터공학과 박사과정
논문접수 : 2008년 4월 8일
수 정 일 : 2008년 6월 13일
심사완료 : 2008년 6월 16일

maximum suffix problem, and solves the four problems either directly employing the maximum suffix algorithms or indirectly using variations of the algorithms. More efficient external memory algorithms for the maximum suffix problem also will improve the external memory algorithms for the four problems.

Two external memory algorithms for computing the maximum suffix of a string are presented in [7]. One of them maintains four blocks in the internal memory and uses at most 6(N/L) disk I/Os. The other uses six blocks and performs 4 (N/L) disk I/Os. Our algorithm will perform 4(N/L) disk I/Os with only four blocks in the internal memory. L is the block size.

In Section 2, we review the internal memory algorithms described in [4, 7]. Our external memory algorithm and its analysis will be given in Section 3. In Section 4, we give some concluding remarks.

2. Preliminaries

Consider a prefix $s = t_1 \ldots t_{d-1}$ of t, where $2 \le d-1 \le N-1$. Let $y = ms(s)$ and let $x$be the string such that $s = xy$. Since y is a string, it can be represented as $y = w^e w'$, where w is the period of y. Then $s = xw^e w'$. Let a, b and p be integers such that $|x| = a-1, |xw^e| = b-1$, and $|w| = p$. Then $x = t_1 \ldots t_{a-1}$,, $w^e = t_a \ldots t_{b-1}$ and $w' = t_b \ldots t_{d-1}$. Let c be another integer such that $|w'| = d-b = c-a$. Figure 1 depicts a decomposition of s and the relationship between the variables.

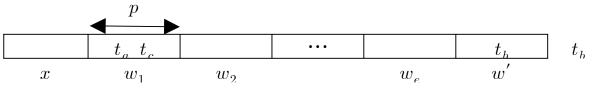
For example, if $\Sigma = \{f, g\}$ and $s = fffgfgfg$, then $y = gfgfg$, $x = fff$, $w = gf$, $e = 2$, and $w' = g$. And, $a = 4$, $b = 8$, $p = 2$, and $c = 5$.

We now review the internal memory algorithm in [4] and [7], which has been adapted and modified for our purpose. Knowing $ms(s)$ and the values of $x, w, e, w', a, b,$ c and p for s, let us try to compute the maximum suffix of $s' = st_d = t_1 \ldots t_d$. Since w' is a prefix of w, $w' = t_a \cdots t_{c-1} = t_b \cdots t_{d-1}$. Compare t_c and t_d. Based on the result of this comparison, there are four cases (1)-(4) to consider.

(1) $t_c = t_d$: In this case we have $ms(s') = ms(s)t_d$, and $w' \leftarrow w't_d$. If $|w'| < |w|$, nothing further is done. (2) If $|w'| = |w|$, then another w has been found, and thus $e \leftarrow e+1$ and $w' \leftarrow \epsilon$.

(3) $t_c > t_d$: We also have $ms(s') = ms(s)t_d$. However, $w't_d$ is not a prefix of w, and thus w is no longer a period. The new period is , and $e \leftarrow 1$ and $w' \leftarrow \epsilon$.

(4) $t_c < t_d$: Since $w't_d > w$, it will be the case that $ms(s') > ms(s)$. So, $x \leftarrow xw^e$ and we have to compute $ms(s') = ms(w't_d) = ms(t_b \ldots t_d)$.



(Fig. 1) Decomposition of s.

Based on the case analysis, Figure 2 shows the internal memory algorithm IMMS(t), which returns a at its completion. Then, $ms(t) = t_a \ldots t_N$.

3. External Memory Algorithm

We assume that there is only one disk. Let L be the size of a block, meaning that a block of L characters long is read from the disk into the internal memory at once. Assume that $L \ge 2$. Let $t = t_1 \ldots t_N$ be the input string. For convenience, assume that N/L is an integer. Partition t into blocks $K_1, \ldots, K_{N/L}$, where $K_1 = t_1 \ldots t_L, \ldots, K_i = t_{(i-1)L+1} \ldots t_{iL}, \ldots, K_{N/L} = t_{(N/L-1)L+1} \ldots t_N$. For $1 \le a \le N$, $bl(a)$, denotes the index of the block which t_a belongs to, i.e., $bl(a) = i$ if $(i-1)L+1 \le a \le iL$.

Two external memory algorithms for computing the maximum suffix of a string are presented in [7]. One of them maintains four blocks in the internal memory, A, C, B, and D, which have the blocks accessed by *a, c, b*, and *d*, respectively. This algorithm uses at most 6(*N/L*) disk I/Os. The other uses six blocks, A, C, B, D, A^*, and B^*, and performs 4*(N/L)* disk I/Os. A^* and B^* have the block next to A and B, respectively.

Our external memory algorithm, called EMS, is shown in (Figure 3). EMS exactly follows the internal memory algorithm in (Figure 2). EMS maintains four blocks in the internal memory, A, A^*, C and D. The blocks $A, C,$ and D always have the blocks that are accessed by the indices *a, c* and *d*, respectively. In other words, $A = K_{bl(a)}$, $C = K_{bl(c)}$, and $D = K_{bl(d)}$. A^* has the block next to A, i.e., $A^* = K_{bl(a)+1}$ if $bl(a) \le N/L - 1$ and $A^* = \phi$, otherwise. Another block B appears in comments. The block $B$is imaginary in the sense that it never resides in the internal memory and thus never appears in executable statements, but it assumed to always $B = K_{bl(b)}$. It is used only for the purpose of analysis of complexity. In EMS, $\leftarrow$ denotes assignments between internal memory locations, and $\Leftarrow$ (denotes assignments from the disk into a block in the internal memory. For a block X in the internal memory, *next*(X) denotes the block next to X, i.e., if $X = K_i$ then *next* $(X) = K_{i+1}$ for $1 \le i \le N/L - 1$ and *next*(X) = **undefined** for $i = N/L$.

Initially, EMS assigns $A \leftarrow C \leftarrow D \Leftarrow K_1$ and $A^* \Leftarrow K_2$.

In case (1), after executing $c \leftarrow c'+1$ EMS checks if t_c, which will be accessed at the next iteration of the while loop, is still in the internal memory. If $bl(c) = bl(c')$, nothing needs to be done because the block C already has t_c, Otherwise, C needs to be updated so that the new block C contains t_c. If $bl(a') = bl(c')$ (i.e., if $A = C$), then t_c is in A^*, and so $C \leftarrow A^*$. If $bl(a') \ne bl(c')$, EMS reads the next block of C from the disk, $C \Leftarrow$ *next*(C). After executing $d \leftarrow d'+1$, EMS does similar operations to d and D as it does to c and C above, to make sure that t_d is in D at the next iteration.

In case (2), EMS assigns $C \leftarrow A$ after executing $c \leftarrow a'$. After increasing $d \leftarrow d'+1$, EMS executes the same operations to d and D as in case (1).

```
       a←c←p←1;
       b←d←2;
       while(d ≤ N)
          if(t_c = t_d)
(1)          if(d − b + 1 < p)     // |w'| < |w|
                c←c + 1;
                d←d + 1;
(2)          else     // |w'| = |w|
                c←a;
                d←d + 1
                b←d;
(3)          else if(t_c > t_d)
                c←a;
                d←d + 1;
                b←d;
                p←d − a;
(4)          else   // t_c < t_d
                a←c←b;
                b←b + 1;
                d←b;
                p←1;
       return a;
```

(Fig. 2) Internal memory algorithm[4, 7]

Case (3) is similar to case (2).

In case (4), after executing $a \leftarrow c \leftarrow b'$, we need to make A and C have $A = C = K_{bl(b')}$. If $bl(c') \neq bl(b')$ (i.e., if $C \neq B$), then $K_{bl(b')}$ has to be read from the disk and assigned to A and C, $A \leftarrow C \Leftarrow K_{bl(b')}$. Since A has been changed, we have to update $A^* =$ *next* (A). If $bl(a') \neq bl(c') = bl(b')$ (i.e., if $A \neq C = B$), then $K_{bl(b')}$ is already in the internal memory and it is sufficient to do $A \leftarrow C$ and to update $A^* =$ *next*(A). Otherwise (i.e., if $A = C = B$), nothing is to be done.

EMS then increases b and assigns it to d, $b \leftarrow b' + 1$ and $d \leftarrow b$. At this point, EMS has $A = C = B$. We check if b and b' belong to different blocks by comparing $bl(b')$ and $bl\ (b)$. If they are different, then $D \leftarrow A^*$; otherwise $D \leftarrow A$.

It will shown that EMS performs at most $4N/L$ disk inputs by an amortized analysis [1]. Initially it is assumed that EMS assigns N/L tokens to each of four blocks A^*, C B and D. An internal memory block has to pay one token to the disk whenever it inputs a new block from the disk. In EMS, for each assignment $\Leftarrow$, a comment line states which block pays for it. In case (2) and (3), B gives $bl(d') - bl(b')$ tokens to C and in case (4), A^* gives $bl(b') - bl(a')$ tokens to D.

A^* pays one token for reading K_1 at the start of EMS. In case (4) A^*advances from $K_{bl(a')+1}$ to $K_{bl(b')+1}$, skipping $bl(b') - bl(a')$ blocks. See Figure 4. This number of toke ns are delivered to D. Since a never decreases during the execution of EMS, A^*never moves backwards, i.e., never goes from K_i to K_j for $i > j$. So, the total number of tokens given to D by A^* is

$$\sum_{case(4)} (bl(b') - bl(a')) \leq N/L - 1$$

Hence, A^*spends at most N/L tokens.

Since b never decreases, B also never moves backwards. In case (2) and (3) B advances from $K_{bl(b')}$ to $K_{bl(d')}$, skipping $bl(d') - bl(b')$ blocks, as shown in Figure 5. Note that after $b \leftarrow d$, it has to be $B = D$. This number of tokens are given to C. So, the total number of tokens given to C is bounded by

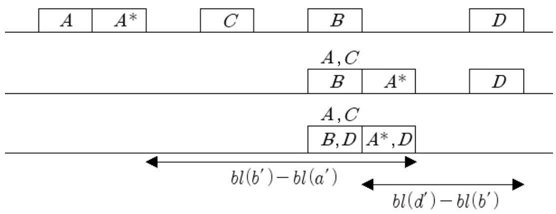
$$\sum_{case(2),(3)} (bl(d') - bl(b')) \leq N/L - 1$$

Hence, B spends at most N/L – 1 tokens.

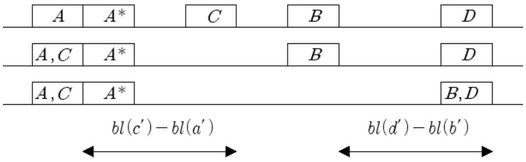
A gets a new block both at the start of the algorithm and every occurrence of in case (4). Whenever A gets a new block, A^*has to change its block by reading *next*(A) from the disk. C pays one token for each of these inputs into A^*. That is, the initial reading of $A^* \Leftarrow K_2$ and every reading of $A^* \Leftarrow$ *next* (A) in case (4) are paid for by C. Note that it holds that $A = C$

```
       a←c←p←1; b←d←2;
       A←C←D⇐K_1;   // A* pays
       A*⇐K_2   // C pays
       while(d ≤ N)
          a'←a; b'←b; c'←c; d'←d;
          if(t_c = t_d)
(1)          if(d − b + 1 < p)
                c←c' + 1;
                if(bl(c') ≠ bl(c))
                   if(bl(a') = bl(c')) C←A*;
                   else C⇐ next(C);  // C pays
                d←d' + 1
                if(bl(d') ≠ bl(d))
                   if(bl(a') ≠ bl(d')) D←A*;
                   else D⇐ next(D);  // D pays
(2)          else  // B gives bl(d') − bl(b') tokens to C
                c←a';
                C←A;
                d←d' + 1; b←d;
                if(bl(d') ≠ bl(d))
                   if(bl(a') ≠ bl(d')) D←A*;
                   else D⇐ next(D);  // D pays
(3)       else if(t_c > t_d)  // B gives bl(d') − bl(b) tokens to C
             c←a';
             C←A;
             d←d' + 1; b←d; p←d − a;
             if(bl(d') ≠ bl(d))
                if(bl(a') = bl(d')) D←A*;
                else D⇐ next(D);  // D pays
(4)       else  // A* gives bl(b') − bl(a') tokens to D
             a←c←b';
             if(bl(c') ≠ bl(b'))
                A←C⇐K_bl(b'); A*⇐ next(A); // C pays two tokens
             else if(bl(c') ≠ bl(a'))
                A←C; A*⇐ next(A); // C pays
             b←b' + 1; d←b; p←1;
             if(bl(b') ≠ bl(b)) D←A*;
             else D←A;
       return a;
```

(Fig. 3) EMS: External memory algorithm



(Fig. 4) Case(4): The first half of case(4) changes from the top to the middle; the second half changes from the middle to the bottom (either $D = A$ or $D = A^*$).



(Fig. 5) Case (2),(3): $C \Leftarrow A$ changes from the top to the middle; $b \leftarrow d$ changes from the middle to the bottom.

at the times of these readings. If $A = C$, then $next(C) = A^*$. In this case, $C \Leftarrow next(C)$ will be replaced by $C \leftarrow A^*$(EMS exactly does this way in case (1). This will save one token for C. This token saved by C is used to pay in advance for getting a new block to A^*.

C also pays one token for $C \Leftarrow K_{bl(b')}$ in case (4). This happens when $bl(c') \neq bl(b')$, i.e., when $C \neq B$. Assignment $C \Leftarrow K_{bl(b')}$ advances C at least one block from the current position. This "free" advancement of C is used for the payment.

Now we need to show that C has a sufficient number of tokens for paying for the readings into A^* as well as the readings into C. C has N/L tokens at the start. C receives $bl(d') \neq bl(b')$ tokens from B in case (2) and (3). By doing $C \Leftarrow A$ in case (2) and (3), C moves backwards from $K_{bl(c')}$ to $K_{bl(a')}$, retreating $bl(c') - bl(a')$ blocks. To get back to the original block $K_{bl(c')}$, C has to input $bl(c') - bl(a')$ blocks in the worst case. See Figure 5. One of these disk inputs can be saved due to A^*. So, C needs $bl(c') - bl(a') - 1$ more tokens. Since $c' - a' = d' - b'$, we have either $bl(c') - bl(a') = bl(d') - bl(b')$, $bl(c') - bl(a') = bl(d') - bl(b') + 1$, or $bl(c') - bl(a') = bl(d') - bl(b') - 1$. In any case, $bl(c') - bl(a') - 1 \leq bl(d') - bl(b')$. So, C receives from B a sufficient number of tokens.

In case (4), D moves backwards from $K_{bl(d')}$ either to $A = K_{bl(b')}$ retreating $bl(d') - bl(b')$ blocks, or to $A^* = K_{bl(b')+1}$ retreating $bl(d') - bl(b') - 1$ blocks. Returning to the original position $K_{bl(d')}$ from A requires at most $bl(d') - bl(b')$ block inputs from the disk. One of these block inputs can be saved due to A^*. So, returning to the original position $K_{bl(d')}$ from A^*requires at most $bl(d') - bl(b') - 1$ block inputs from the disk. See Figure 4. In either case, D needs at most $bl(d') - bl(b') - 1$ extra tokens. A^*gives $bl(b') - bl(a')$ tokens to D. Since $b' - a' > c' - a' = d' - b'$, it is easy to see that $bl(b') - bl(a') \geq bl(d') - bl(b') - 1$. So, D is given by A^* a number of tokens that is enough to go back to the original position.

[Theorem 1] *Given a string of length N, on the one-disk external memory model with block size L, the maximum suffix of the string can be found using at most $4 \lceil N/L \rceil$ disk I/Os.*

4. Conclusions

An external memory algorithm for computing the maximum suffix of a string has been presented. The algorithm uses four blocks in the internal memory and performs at most 4(*N/L*) disk I/Os. One of the future works is to decrease the number of disk I/Os while still using four blocks.

References

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, Introduction to Algorithms, Second Ed., MIT Press and McGraw-Hill, 2001.

[2] M. Crochemore, String matching and periods, Bulletin of the EATCS, Vol.39, pp.149-153, 1989.

[3] M. Crochemore, String matching on ordered alphabets, Theor. Comput. Sci., Vol.92, No.1, pp.33-47, 1992.

[4] M. Crochemore and D. Perrin, Two-way string matching, J. ACM, Vol.38, No.3, pp.651-675, 1991.

[5] J.P. Duval, Factoring words over an ordered alphabet, J. Algorithms, Vol.4, No.1, pp.363-381, 1983.

[6] D.E. Knuth, J.H. Morris, and V.R. Pratt, Fast pattern matchings in strings, SIAM J. Comp., Vol.6, No.2, pp.323-350, 1977.

[7] K. Roh, M. Crochemore, C.S. Iliopoulos, and K. Park, External Memory Algorithms for String Problems, Proceedings of the 17th Australasian Workshop on Combinatorial Algorithms, Central Australia, July 2006.

[8] Y. Shiloach, Fast canonization of circular strings, J. Algorithms, Vol.2, pp.107-121, 1981.

[9] J.S. Vitter, External memory algorithm and data structures: dealing with massive data, ACM Computing Surveys, Vol.33, No.2, pp.209-271, 2001.

김 성 권

e-mail : skkim@cau.ac.kr

1981년 서울대학교 계산통계학과(학사)

1983년 한국과학기술원 전산학과(공학석사)

1990년 8월 University of Washington 전산학과(공학박사)

1991년 3월~1996년 2월 경성대학교 전자 통계학과 조교수

1996년 3월~현 재 중앙대학교 컴퓨터공학과 교수

관심분야 : 계산기하학, 생물정보학, 암호응용 및 정보보호 등

김 수 철

e-mail : sckim@alg.cse.cau.ac.kr

2004년 중앙대학교 컴퓨터공학과(학사)

2007년 중앙대학교 컴퓨터공학과(공학석사)

2007년 3월~현 재 중앙대학교 컴퓨터공학과 박사과정

관심분야 : RFID 보안, Sensor network 보안, 암호응용 및 정보보호 등

조 정 식

e-mail : mfg@alg.cse.cau.ac.kr

2003년 강남대학교 전자계산학과(학사)

2005년 중앙대학교 컴퓨터공학과(공학석사)

2005년 3월~현 재 중앙대학교 컴퓨터공학과 박사과정

관심분야 : RFID 보안, Sensor network 보안, 암호응용 및 정보보호 등