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 $\Sigma$ be an alphabet. Let $s$ and $t$ be two strings over $\Sigma$. If $s$ is lexicographically less than $t$, we denote this by $s < t$. Let $t = t_1 \cdots t_N$, where $N$ is the length of $t$. For $1 \leq a \leq b \leq N$, $t_a \cdots t_b$ 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 $w$ is said to be the period of another string $\sigma$ if $\sigma = \phi \sigma'$, where $\phi$ is a prefix of $\phi'$ and $\phi$ is as short as possible. In other words, repeating $\epsilon$ copies of $\phi$ and appending $\phi'$ after it results in $\sigma$. The period of a string can be computed in linear time [2]. String $t$ has $N$ circular shifts, namely $t, t_1t_2 \cdots t_{N-1}, t_2 \cdots t_N t_1$ for $1 \leq i \leq N$. 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 $t = w_1w_2 \cdots w_n$, where the strings $w_1, w_2, \ldots, w_n$ are lexicographically non-increasing and each $w_i$ ($1 \leq i \leq n$) 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
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 \(x = t_1 \cdots t_i\), of \(t\), where \(2 \leq d - 1 \leq N - 1\).

Let \(y = m(s)\) and let \(x\) be the string such that \(x = s_y\). Since \(y\) is a string, it can be represented as \(y = w_1w_2\), where \(w\) is the period of \(y\). Then \(s = xw_1w_2\).

Let \(a, b, c, e\) and \(p\) be integers such that \(|w| = a - 1, |xw| = b - 1, |w| = p\). Then \(x = t_1 \cdots t_{i-1}, w_1 = t_{i+1} \cdots t_{i-1,}\) and \(w_2 = t_{i+2} \cdots t_{i-1,}\). Let \(c\) be another integer such that \(|w_1| = d - b = c - a\). Figure 1 depicts a decomposition of \(s\) and the relationship between the variables.

For example, if \(\Sigma = \{f, g\}\) and \(s = sfgfsgf\), then \(y = gsfSG\), \(x = sfg\), \(w = sfg\), \(c = 2\), and \(p = 2\). 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 \(m(s)\) and the values of \(x, w, c, e, p\) for \(s\), let us try to compute the maximum suffix of \(s' = s_{t_1} \cdots t_c\).

Since \(w'\) is a prefix of \(w', w' = t_{i+1} \cdots t_{i-1,}\) and \(w' = w\).

Consider \(t_i\) and \(t_e\). Based on the result of this comparison, there are four cases (1)-(4) to consider.

1. \(t_i = t_e\): In this case we have \(m(s') = m(s)\).
2. \(t_i < t_e\): We also have \(m(s') = m(s)\).
3. \(t_i > t_e\): We have \(w' = w\).

Based on the case analysis, Figure 2 shows the internal memory algorithm \(IMMS(t)\), which returns \(a\) at its completion. Then, \(m(s(t)) = t_{e+1} \cdots t_i\).

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 \(L \geq 2\). Let \(t = t_1 \cdots t_x\) be the input string. For convenience, assume that \(N/L\) is an integer. Partition \(t\) into blocks \(K_1, \ldots, K_{N/L}\), where \(K_1 = t_1 \cdots t_x, K_2 = t_{x+1} \cdots t_{x+L}, \ldots, K_{N/L} = t_{(N/L-1)L+1} \cdots t_N\). For \(1 \leq a \leq N, b(s)\), denotes the index of the block which \(t_a\) belongs to, i.e., \(b(s) = i\) if \((i-1)L + 1 \leq a \leq iL\).

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, \(A, B, C, D\), and \(B', C\), which have the blocks accessed by \(a, b, d\), 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, \(A', B', C', D\). The blocks \(A, C\), and \(D\) always have the blocks that are accessed by the indices \(a, c\), and \(d\), respectively. In other words, \(A = K_{b(s)}, C = K_{c(s)}\), and \(D = K_{d(s)}\). \(A'\) has the block next to \(A\), i.e., \(A' = K_{b(s)+1}\) if \(b(s) \leq N/L\) and \(A' = b(s), \) otherwise. Another block \(B\) appears in comments. The block \(b(s)\) is imaginary in the sense that it never resides in the internal memory and thus never appears in executable statements, but it is always assumed that \(B = b(s)\).

It is used only for the purpose of analysis of complexity. In EMS, \(\sim\) denotes assignments between internal memory locations, and \(\sim\) (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., \(X = K_{b(s)}\), then next\((X) = K_{b(s)+1}\) for \(1 \leq i \leq N/L\) and next\((X) = undefined\) for \(i = N/L\).

Initially, EMS assigns \(A = C = D = K_1\) and \(A' = K_2\).

In case (1), after executing \(\wedge c\), and \(\wedge c\) checks if \(t_e\), which will be accessed at the next iteration of the while loop, is still in the internal memory. If \(\wedge (c) = \wedge (c)\), nothing needs to be done because the block \(C\) already has \(t_e\). Otherwise, \(C\) needs to be updated so that the new block \(C\) contains \(t_e\). If \(\wedge (c) = \wedge (c)\) (i.e., if \(A = C\)), then \(t_e\) is in \(A'\), and so \(\wedge (c)\).

If \(b(s) \neq b(s)\), EMS reads the next block of \(c\) from the disk, \(c\), until next\((c)\). After executing \(\wedge d + 1\) EMS, does similar operations to \(d\) and \(p\) as it does to \(c\) and \(c\) above, to make sure that \(t_e\) is in \(D\) at the next iteration.

In case (2), EMS assigns \(A = C = D = \wedge (c)\). After increasing \(\wedge d + 1\), EMS executes the same operations to \(d\) and \(D\) as in case (1).
whenever it inputs a new block from the disk. In EMS, for

At this point, EMS has

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

\[
\sum_{i=0}^{\min(i, j)} (b(d'_i) - b(a'_i)) \leq N/L - 1
\]

Hence, \(A^+\) spends at most N/L tokens.

Since \(a\) never decreases, \(B\) also never moves backwards.

In case (2) and (3) \(B\) advances from \(K_{102}\) to \(K_{103}\) skipping \(b(d'_i) - b(a'_i)\) blocks, as shown in Figure 3. Note that after \(b \rightarrow d\), it has to be \(B \rightarrow D\). This number of tokens are given to \(C\).

So, the total number of tokens given to \(C\) is bounded by

\[
\sum_{i=0}^{\min(i, j)} (b(d'_i) - b(a'_i)) \leq N/L - 1
\]

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 \(A' = K_a\), and every reading of \(A' = next(A)\) in case (4) are paid for by \(C\). Note that it holds that \(A = C\)

\[
\begin{align*}
    a &\rightarrow c \leftarrow p \rightarrow 1; \\
    b &\rightarrow d \rightarrow 2; \\
    \text{while}(d \leq N) \\
    \text{if}(t_a = t_c) \\
    (1) &\quad \text{if}(d - b + 1 < p) /\!\!/ |w'| < |w| \\
    c &\rightarrow a; \\
    d &\rightarrow d + 1; \\
    b &\rightarrow d; \\
    \text{else} \\
    (2) &\quad // |w'| \geq |w| \\
    c &\rightarrow a; \\
    d &\rightarrow d + 1; \\
    b &\rightarrow d; \\
    \text{else if}(t_a > t_c) \\
    (3) &\quad c \leftarrow a; \\
    d &\rightarrow d + 1; \\
    b &\rightarrow d; \\
    p &\rightarrow d - a; \\
    \text{else if}(t_a < t_c) \\
    (4) &\quad c \leftarrow c - b; \\
    d &\rightarrow d + 1; \\
    b &\rightarrow d; \\
    p &\rightarrow d; \\
    \text{return} a;
\end{align*}
\]

(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 \(A = C = K_{103}\). If \(b(c') = b(b')\) (i.e., if \(C \neq B\)), then \(K_{103}\) has to be read from the disk and assigned to \(A\) and \(C\), \(A = C = K_{103}\). Since \(A\) has been changed, we have to update \(A' = next(A)\). If \(b(a') = b(c') = b(b')\) (i.e., if \(A = C = B\)), then \(K_{103}\) is already in the internal memory and it is sufficient to do \(A \leftrightarrow C\) and to update \(A' = next(A)\). Otherwise (i.e., if \(A = C \neq B\)), 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 \(b(b')\) and \(b(b)\). 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 \(\bowtie\), a comment line states which block pays for it. In case (2) and (3), \(B\) gives \(b(d') - b(a')\) tokens to \(C\) and in case (4), \(A\) gives \(b(b') - b(a')\) tokens to \(D\).

\(A^+\) pays one token for reading \(K_1\) at the start of EMS. In case (4) \(A^+\) advances from \(K_{03}\) to \(K_{02}\), skipping \(b(b') - b(a')\) blocks. See Figure 4. This number of tokens 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

\[
\sum_{i=0}^{\min(i, j)} (b(d'_i) - b(a'_i)) \leq N/L - 1
\]
at the times of these readings. If \( A = C \), then \( \text{next}(C) = A' \).

In this case, \( C = \text{next}(C) \) will be replaced by \( C' \). The way this does this 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 \( C = \text{K} \) in case (4). This happens when \( \text{block}(C) = \text{block}(B) \), i.e., when \( C = B \). Assignment \( C = \text{K} \) 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 \( \text{block}(C') = \text{block}(B') \) tokens from \( B \) in case (2) and (3). By doing \( C = A \) in case (2) and (3), \( C \) moves backwards from \( \text{K} \) to \( \text{K} \), replacing \( \text{block}(C') = \text{block}(B') \) blocks. To get back to the original block \( \text{K} \), \( C \) has to input \( \text{block}(C') = \text{block}(B') \) blocks in the worst case. See Figure 5. One of these disk inputs can be saved due to \( A' \). So, \( C \) needs \( \text{block}(C') - \text{block}(A') \) more tokens.

In case (4), \( D \) moves backwards from \( \text{K} \) either to \( \text{K} \) or \( \text{K} \) or \( \text{K} \) or \( \text{K} \), depending \( \text{block}(D') = \text{block}(B') \) blocks, or to \( \text{K} \) to \( \text{K} \) to \( \text{K} \) to \( \text{K} \). Returning to the original position \( \text{K} \) from \( A \) requires at most \( \text{block}(D') - \text{block}(B') \) block inputs from the disk. One of these block inputs can be saved due to \( A' \). So, \( A' \) requires at least \( \text{block}(D') - \text{block}(B') - 1 \) extra tokens. \( A' \) gives \( \text{block}(D') = \text{block}(B') \) tokens to \( D \). Since \( \text{block}(D') = \text{block}(B') \), it is easy to see that \( \text{block}(D') - \text{block}(B') \geq \text{block}(D') - \text{block}(B') - 1 \). 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 \( t \), the maximum suffix of the string can be found using at most \( 4(N/L) \) 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**


