A Linear-Time Burrows-Wheeler Transform Using Induced Sorting

Daisuke Okanohara[1] and Kunihiko Sadakane[2]

[1] Department of Computer Science, University of Tokyo,
Hongo 7-3-1, Bunkyo-ku, Tokyo 113-0013, Japan
`hillbig@is.s.u-tokyo.ac.jp`

[2] National Institute of Informatics (NII),
Hitotsubashi 2-1-2, Chiyoda-ku, Tokyo 101-8430, Japan
`sada@nii.ac.jp`

Abstract. To compute Burrows-Wheeler Transform (BWT), one usually builds a suffix array (SA) first, and then obtains BWT using SA, which requires much redundant working space. In previous studies to compute BWT directly [5,12], one constructs BWT incrementally, which requires $O(n \log n)$ time where n is the length of the input text. We present an algorithm for computing BWT directly in linear time by modifying the suffix array construction algorithm based on induced sorting [15]. We show that the working space is $O(n \log \sigma \log \log_\sigma n)$ for any σ where σ is the alphabet size, which is the smallest among the known linear time algorithms.

1 Introduction

A Burrows-Wheeler Transform (BWT) [1] is a transformation from a text to a text, which is useful for many applications including data compression, compressed full-text indexing, pattern mining to name a few.

To compute BWT, one usually builds a suffix array (SA) first, and then obtains BWT from SA. Although both steps can be done in linear time in the length of the text [8,9,10], it requires large working space. Although the space for the result of BWT is $n \lg \sigma$ bits [1] where n is the length of the text, and σ is the alphabet size, that for SA is $n \lg n$ bits. For example, in the case of human genomes, $n = 3.0 \times 10^9$ and $\sigma = 4$, the size of BWT is about 750 MB, and that of SA is 12 GB. Therefore, the working space is about 16 times larger than that for BWT.

Previous studies [5,12] showed that one can compute BWT without SA by implicitly adding suffixes from the shortest ones to longest ones. However, these algorithms are slow due to the large constant factor, and their computational cost are $O(n \log n)$ time. There also exist other types of algorithms. Hon et al. [6] gave an algorithm using $O(n \log \sigma)$ space and $O(n \log \log \sigma)$ time. Na and Park [13] gave one using $O(n \log \sigma \log_\sigma^\alpha n)$ space and $O(n)$ time where $\alpha = \log_3 2$.

In this paper, we present an algorithm for computing BWT directly in linear time using $O(n \log \sigma \log \log_\sigma n)$-bit space. Our algorithm is based on the suffix array construction algorithm based on induced sorting [15]. In original SA algorithm, the whole

[1] $\lg x$ denotes $\lceil \log_2 x \rceil$.

Table 1. Time and space complexities. H_0 is the order-0 empirical entropy of the string and $H_0 \leq \log \sigma$. $\alpha = \log_3 2$.

Time	Space (bits)	References
$O(n)$	$O(n \log n)$	[2,8,9,10,15] (compute SA)
$O(n \log n)$	$O(nH_0)$	[5]
$O(n \log \log \sigma)$	$O(n \log \sigma)$	[6]
$O(n)$	$O(n \log \sigma)$	[6] ($\log \sigma = O((\log \log n)^{1-\epsilon})$)
$O(n)$	$O(n \log \sigma \log_\sigma^\alpha n)$	[13]
$O(n)$	$O(n \log \sigma \log \log_\sigma n)$	This paper

SA are induced from the carefully sampled SA. Our algorithm simulates this algorithm by using BWT only, and induces whole BWT from the sampled BWT. Our algorithm works in linear time and requires the working space close to that for input and output. Moreover our algorithm is simple and easy to implement. Table 1 gives a comparison with other algorithms. Our algorithm uses the smallest space among the linear time algorithms.

2 Preliminaries

Let $T[1, n]$ be an input text, n its length, and Σ its alphabet set, with $\sigma = |\Sigma|$. We denote the i-th character of T by $T[i]$, and the substring from i-th character to j-th character for $i \leq j$ by $T[i, j]$. We assume that T is followed by a special character $T[n] = \$$, which is lexicographically smaller than any other characters in T, and do not appear in T elsewhere. We also assume $\sigma \leq n$ because otherwise $n \log \sigma = \Omega(n \log n)$.

2.1 Suffix Arrays and Burrows-Wheeler Transform

A suffix of T is $T_i = T[i, n]$ ($i = 1, \ldots, n$). Then, a suffix array of T, $SA[1, n]$ is defined as an integer array $SA[1, n]$ of length n such that $T_{SA[i]} < T_{SA[i+1]}$ for all $i = 1, \ldots, n-1$ where $<$ between strings denotes the lexicographical order of them. SA requires $n \lg n$ bit of space.

A Burrows-Wheeler Transform (BWT) of a text T, $B[1, n]$ is defined as follows;

$$B[i] = \begin{cases} T[SA[i]-1] & (SA[i] > 1) \\ T[n] & (SA[i] = 1). \end{cases} \tag{1}$$

We will denote BWT not only as the transformation, but also the result of the transformation.

BWT has several characteristics; First, BWT is a reversible transformation. That is, the original text can be recovered from BWT without any additional information [1]. Second, BWT is often easy to compress. For example, by using the compression boosting technique [3], we can compress BWT in k-th order empirical entropy for any k by using simple compression algorithms, which does not consider the context information. Third, BWT can be used for constructing compressed full-text indexes. For example, we can build a compressed suffix array [4], and a FM-index from BWT in $O(n)$ time [5].

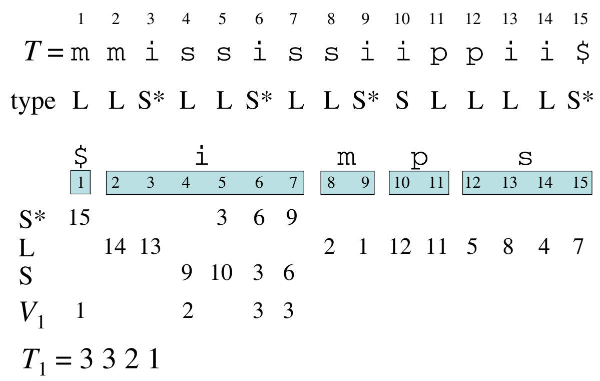

Fig. 1. An example of induced sorting for $T = mmississiippii\$$. S*substrings are located at positions $(15, 3, 6, 9)$ in T, and their positions are stored in SA. From them, L-type suffixes $(14, 13, 2, 1, 12, 11, 5, 8, 4, 7)$ are induced. Then S-type suffixes $(9, 10, 3, 6)$ are induced from L-type suffixes. We obtain names V_1 of the S*substrings, and finally obtain the shortened string $T_1 = 3321$.

Forth, BWT is also useful for many other applications, such as data compression [1], compressed full-text indexes [14], and pattern mining [11].

Since BWT is the result of the shuffled input text, the space of BWT is $n \lg \sigma$ bits. Therefore, the space of BWT is much smaller than that for the suffix array when $\sigma \ll n$, such as genome sequences.

To compute BWT, one usually constructs SA first, and then obtains BWT using (1). Although the total computational time is linear in the length of the input text [8,9,10], its working space is $n \lg n$ bits and is much larger than that for BWT. Previous studies [5,12] show how to compute BWT without SA. These algorithms incrementally build BWT by implicitly adding the suffixes from the shortest ones. Although these algorithms are remarkably simple, they require $O(n \log n)$ time and also slow in practice due to the large constant for keeping dynamic data structures. In another study [7], one divides input into the small blocks according to their first characters in suffixes so that the working space would be small. However it requires $O(n \log n)$ time and relies on the complex handling of long repetitions. Therefore, no previous work can compute BWT in linear time using small working space.

2.2 Storing Increasing Sequences

Let $s_1, s_2, \ldots, s_n$ be a strictly increasing sequence of integers such that $0 \leq s_1 < s_2 < \cdots < s_n < U$. A naive representation of the sequence uses $n \lg U$ bits of space. Instead we can represent it succinctly by using a clever encoding with the following properties, which is rephrased from [6].

Lemma 1. *A sequence* $s_1, s_2, \ldots, s_n$ *of* n *integers such that* $0 \leq s_1 < s_2 < \cdots < s_n < U$ *can be stored in a bit-stream of* $n(2 + \lg \frac{U}{n})$ *bits. The bit-stream can be* *constructed incrementally in $O(n)$ time in the sense that the integers can be given in arbitrary order, provided that both the value s_i and its index i are given. Furthermore, after $O(n)$ time preprocessing to the bit-stream to construct an auxiliary data structure of $O(n \log\log n / \log n)$ bits, the i-th smallest integer s_i ($1 \leq i \leq n$) is obtained in constant time.*

Proof. The bit-stream consists of two parts: upper stream and lower stream. Each integer s_i is originally encoded in $\lg U$ bits, and its lower $\lg \frac{U}{n}$ bits are stored in the lower stream as it is. The upper $\lg n$ bits are stored in the upper stream after converting its original binary encoding to the following one. The upper stream is represented by a $0, 1$ vector $B[1, 2n]$, and the i-th number s_i is encoded by setting $B[i + \lfloor \frac{s_i}{n} \rfloor] = 1$. It is easy to show that each bit of B corresponds to at most one number in the sequence, and the bit position for s_i does not depend on other numbers. Therefore the upper stream can be constructed for any input order of the numbers.

The element s_i is obtained from the bit-streams as follows. The upper $\lg n$ bits of the binary encoding of s_i are computed by $\mathbf{select}(B, i) - i$, where $\mathbf{select}(B, i)$ is the position of i-th 1 in B and it is computed in constant time using an auxiliary data structure of $O(n \log\log n / \log n)$ bits [16] which is constructed by $O(n)$ time preprocessing. The lower $\lg \frac{U}{n}$ bits of the binary encoding of s_i are obtained directly from the lower stream in constant time. By concatenating the upper and the lower parts, we obtain s_i. □

3 Constructing SA Based on Induced Sorting

Our novel algorithm for computing BWT is based on the liner-time suffix array construction algorithm using purely induced sorting [15]. We will explain their algorithm here again for the sake of clarity. We call this algorithm *SAIS* (Suffix Array construction algorithm based on Induced Sorting).

First, we classify suffixes into two types; S-type, and L-type as follows.

Definition 1. *A suffix T_i is called S-type if $T_i < T_{i+1}$, and called L-type if $T_i > T_{i+1}$. The last suffix is defined as S-type.*

We also classify a character $T[i]$ to be S- or L-type if T_i is S- or L-type, respectively. We can determine the type of each suffixes in O(1) time by scanning T once from right to left as follows. First, $T[n]$ is defined as S-type. Next, for i from $n-1$ to 1, we classify a suffix by using the following rule;

– T_i is S-type if ($T[i] < T[i+1]$) or ($T[i] = T[i+1]$ and T_{i+1} is S-type).
– T_i is L-type otherwise.

Obviously, in SA, the pointers for all the suffixes starting with a same character must span consecutively. Let's call a sub-array in SA for all the suffixes with a same character as a bucket. Specifically, we call c-bucket a bucket starting with a character c. Further, in the same bucket, all L-type suffixes precede to the S-type suffixes due to their definition. Therefore, each bucket can be split into two sub-bucket with respect to the types of suffixes inside: we call them L- and S-type buckets each.

We also introduce S*-type suffixes.

Definition 2. *A suffix T_i is called S*-type if T_i is S-type and T_{i-1} is L-type (called Left-Most-S type in [15]). A character $T[i]$ is called S*-type if T_i is S*-type.*

Then, given sorted S*-type suffixes, we can induce the order of L-type and S-type suffixes as follows. These steps can be done in linear time.

– Given sorted S*suffixes, put all of them into their corresponding S-type buckets in SA, with their relative orders unchanged.
– Scan SA from the head to the end. For each item $SA[i]$, if $c = T[SA[i] - 1]$ is L-type, then put $SA[i] - 1$ to the current head of the L-type c-bucket and forward the current head one item to the right.
– Scan SA from the end to the head. For each item $SA[i]$, if $c = T[SA[i] - 1]$ is S-type, put $SA[i] - 1$ to the current end of the S-type c-bucket and forward the current end one item to the left.

Next, we explain how to obtain sorted S*suffixes in linear time.

We introduce S*substring.

Definition 3. *An S*substring is (i) a substring $T[i, j]$ with both $T[i]$ and $T[j]$ being S*characters, and there is no other S*character in the substring, for $i \neq j$; or (ii) $T[n]$.*

Let we denote these S*substrings in T as $R_1, R_2, \ldots, R_{n'}$ where R_i is the i-th S* substring in T. Let σ_1 be the number of different S*substrings in T. Then we assign names $V_i \in [1, \sigma_1]$ to R_i, $(i = 1, \ldots, n')$ so that $V_i < V_j$ if $R_i < R_j$ and $V_i = V_j$ if $R_i = R_j$. Finally, we construct a new text $T_1 = V_1, V_2, \ldots, V_{n'}$ whose length is n' and the alphabet size is σ_1.

We recursively apply the linear-time suffix array construction algorithm to T_1 and obtain the order of S*suffixes. Since the relative order of any two S*suffixes in T is the same for corresponding suffixes in T_1 [15], we can determine the order of the S*suffixes by using the result of the recursive algorithm.

To compute the names of S*substrings, we again use the induced algorithm modified that input are *unsorted* S*suffixes; we place unsorted S*suffixes at the end of S-type buckets, and apply inducing procedure; induce the order of L-type suffixes from S*suffixes, and the order of S-type suffixes from L-type suffixes. As a result, we obtain the sorted S*substrings. Then, we can assign names to each S*substring in linear time by checking their suffixes from the beginning to the ending. An example is shown in Figure 1.

Finally, to obtain the total computational cost, we use the following lemma;

Lemma 2. *[15] The length of T_1 is at most half of that of T.*

The SAIS algorithm for an input of length n requires $O(n)$ time and the time required to solve the same problem of half the length. Therefore, the total time complexity is $O(n)$.

4 Direct Construction of BWT

We explain our algorithm to obtain BWT without SA by modifying SAIS. We use the same definitions of S-, L-, S*-type suffixes/characters, and S*-substrings as in the previous section. In addition, we call BWT character $B[i] = T[SA[i]-1]$ ($B[i] = T[n]$ if $SA[i] = 1$) S-, L-, S*-type if $T[SA[i]]$ is S-, L-, S*-type character. Our idea is that we can simulate SAIS by using only S*substrings, in that we can induce L-type BWTs from sorted S*-BWTs and induce S-type BWTs from sorted L-type BWTs. Similarly, we can determine the order of S*substrings by using the inducing algorithm.

In our algorithm, we do not store SA, but keep S*substrings directly. Specifically, we keep following arrays, each of which stores the list of substrings for each character $c \in \Sigma$.

- S^*_c : Store substrings whose last character is c and S*-type.
- L_c : Store substrings whose last character is c and L-type.
- LS_c : Store substrings whose last character is c and S-type, and the next to the last character in the original text is L-type.
- S_c : Store substrings whose last character is c and S-type and the next to the last character in the original text is S-type.

These arrays support the following operations.

- $A.push\_back(q)$: Add the substring q at the end of the array A.
- $A.pop\_front()$: Return the substring at the front of the array A, and remove it.
- $A.reverse()$: Reverse the order in the array A.

For example, after the operation $A.push\_back$(“abc”), $A.push\_back$(“bcd”), and $A.push\_back$(“cde”), $A = \{$“abc”, “bcd”, “cde”$\}$. The operation results are $A.pop\_front() =$ “abc” and $A.pop\_front() =$ “bcd”. We will discuss how to store these substrings in the section 5.

Note that, since our algorithm only uses these FIFO operations (and reverse operations.), we can implement this on external memory architecture easily.

In addition to these arrays, we keep following three arrays to store the result of BWT.

- $E[1, n']$: Store the end characters of S*substrings.
- BL_c : Store the result of L-type BWT for a c bucket.
- BS_c : Store the result of S-type BWT for a c bucket.

The overall algorithm is shown in the algorithm 1. All S*substrings are placed in S^*_c, and then moved to L_c, LS_c, and S_c in turn, and this is almost the same as in SAIS.

First, an input text $T[1, n]$ is decomposed into S*substrings $R_1, R_2, \ldots, R_{n'}$. Let σ_1 be the number of different S*substrings. Then we assign names V_i to R_i, $(i = 1, \ldots, n')$ from $1 \ldots \sigma_1$ so that $V_i < V_j$ if $R_i < R_j$ and $V_i = V_j$ if $R_i = R_j$. Then, we recursively call BWT-IS to determine the BWT of T_1. Let $B_1[1, n']$ be the result of BWT of T_1. Then, we induce the B from B_1. All steps except the assigning of names and induce are obviously done in linear time. We will see these steps in the following sections.

Algorithm 1. BWT-IS(T, n, k): The algorithm for computing BWT for T

```
Input: $T[1,n]$ : An input text
$n$: An input length
$k$: A number of alphabets
Scan $T$ once to classify all characters in $T$ as S- or L-type (Also S$^*$type).
Decompose $T$ into S$^*$substrings $R[1,\dots n']$ ($R[i] \in \{1,\dots,k\}^*$)
Name S$^*$substrings by using the result of Induce($R$), and get a new shortened string $T_1[1,n']$,
$T_1[i] \in \{1,\dots,k'\}$.
if Each character in $T_1$ is unique then
  Directly compute $B_1$ from $T_1$
else
  $B_1$ = BWT-IS($T_1$, $n'$, $k'$) // recursive call
end if
Decode S$^*$strings $B_1$ into $R'[1,\dots,n']$
$B$ = Induce($R'$)
Output: $B$
```

4.1 Induce BWT

We explain how to induce B, the BWT of original substring, from B_1, the BWT of S*substrings. The overall algorithm is shown in the algorithm 2.

First we lookup the original S*substrings, and keep the reversed ones. We reverse it because all operations are represented by pop_front and $push\_back$ only. We do not require the reverse operation if we replace pop_front and $push\_back$ with pop_back and $push\_front$. Each substring is appended the character $c = k-1$ which denotes the sign of the end of the string. We place these substring at S^*_c where c is the first character of the substring.

Second, for each character i from 1 to σ, we lookup L_i one by one. and check whether the first character c (Since S*strings are reversed, this corresponds to the last character in S*strings) is L-type or not. Particularly if $c \geq i$ then it is L-type and append it to the array L_c. If not, we place it at LS_i. After enumerating all the elements in L_i, we next lookup the substrings in S^*_i, and move it to L_c where c is the first character of each substring. Note that we can omit the check of L-type here because all the last characters in S^*_i should be L-type.

Third, for each character i from σ to 1, we lookup the substrings in the array S_i, and check whether it is empty or not. If so, we place the last character of E (we determine the position of S*substring) at the end of BS_i. After seeing all elements in S_i, we check LS_i similarly. In this case, we can omit the check whether it is empty because all substrings in this arrays should not be empty.

After obtaining the L-type and S-type BWTs, we just append these substrings in order and return it as the result of BWT.

4.2 Assigning Names to S*Strings

Let we explain how to compute the names of S*substrings. This is almost the same as in the induced algorithm in the previous section. As in SAIS algorithm, we apply

Algorithm 2. Induce(R): The algorithm for inducing BWT from the S*substrings

```
Input: $R[1, n']$ : A list of S$^*$substrings.
for $i = 1$ to $n'$ do
  $U := Reverse(R_i)$
  $U.push\_back(k-1)$ // Sentinel
  $c := U.pop\_front()$.
  $S^*_c.push\_back(U)$
end for
for $i = 1$ to $k$ do
  while $U := L_i.pop\_front()$ do
    $c := U.pop\_front()$
    $BL_i.push\_back(c)$
    if $c < i$ then
      $LS_i.push\_back(c + U)$
    else
      $L_c.push\_back(U)$
    end if
  end while
  $LS_i := Reverse(LS_i)$
  while $U := S^*_i.pop\_front()$ do
    $c := U.pop\_front()$
    $E.push\_back(c)$
    $L_c.push\_back(U)$
  end while
end for
$E := Reverse(E)$
for $i = k$ to $1$ do
  while $U := S_c.pop\_front()$ do
    $c := U.pop\_front()$
    if $c < i$ then
      $BS_i.push\_front(c)$
      $S_c.push\_back(U)$
    else
      $c2 := E_c.pop\_front()$ // Reach the sentinel
      $BS_i.push\_back(c2)$
    end if
  end while
  while $U := LS_i.pop\_front()$ do
    $c := U.pop\_front()$
    $BS_i.push\_back(c)$
    $S_c.push\_back(U)$
  end while
end for
for $i = 1$ to $k$ do
  $BS_i := Reverse(BS_i)$
  $B := B + BL_i + BS_i$
end for
Output: $B$
```

the algorithm to unsorted S*substrings, and as a result, we obtain sorted S*substrings. At this time, we do not place the BWT characters, and we keep S*substrings without removing. This is achieved by changing $pop\_front()$ operations for the substring U in the algorithm 2 by the following $cycle()$ operation.

– $A.cycle()$: Return the character at the front of the array A, and moves it to the end of A.

For example, for $A =$ "abcd", $A.cycle() =$ "a" and after this operation $A =$ "bcda". This cycle operation can be done in $O(1)$ time when the length of substring is $O(\log n)$ bits. Otherwise, we keep pointers and simulate the cycle operation, which is also done in $O(1)$ time.

Next, given sorted S*substrings, we calculate the names of them, which is trivial, and we place them in an original order to obtain the shortened string $T_1 = V_1, V_2, \ldots, V_{n'}$. However, since we don't have SA, we cannot find the original positions of each names directly.

First, we store positions $p_1, \ldots, p_{n'}$ of S*substrings in T using the data structure of Lemma 1. Namely, we define $q_i = cn + p_i$ for $i = 1, \ldots, n'$ where $c = T[p_i]$, and store all q_i by using Lemma 1 using $n(2 + \log\sigma) + o(n)$ bits. From q_i, c and p_i are obtained in constant time by $c = \lfloor q_i/n \rfloor$ and $p_i = q_i \bmod n$. We call this data structure Φ. To compute Φ, for each $c \in \Sigma$ we count the number of occurrences of c in T. This is done in $O(n)$ time for all c by using an integer array of $\sigma \log n \le n \log\sigma$ bits because $\sigma \le n$. Then we scan T from right to left to determine the positions $p_{n'}, p_{n'-1}, \ldots, p_1$ of S*substrings in this order. For each p_i we compute q_i and store it in Φ. We also store the S*substring located at p_i in a bucket. Namely, we obtain the head character $c = T[p_i]$, and store the substring into bucket for c. Each bucket stores the concatenation of S*substrings with the same head character. We store a pointer to indicate the position to append a new S*substring for each bucket. The space for storing all the pointers is $O(\sigma \log n) = O(n \log\sigma)$.

If the length of an S*substring is at most $\log_\sigma n$, it is encoded in at most $\log n$ bits, and therefore it takes constant time to append it to the end of a bucket. Otherwise, instead of storing the S*substring itself, we store the index p_i and the length of the S*substring. Because there exist at most $n \log\sigma / \log n$ S*substrings of length more than $\log_\sigma n$, we can store the indexes and lengths in $O(n \log\sigma)$ bits.

The S*substrings are sorted by this modified induced-sorting in $O(n)$ time and $O(n \log\sigma)$-bit working space. During the modified induced-sorting for sorting S*substrings, we compute the following function Ψ and store it by the data structure of Lemma 1. Assume that in the original algorithm a suffix $SA[j]$ is induced from $SA[i]$, that is, $T[SA[i] - 1] = c$ and $SA[j]$ belongs to the bucket for c. In our modified algorithm corresponding this, we define $\Psi[j] = cn + i$. Because this induce-sorting scans SA from left to right and buckets are sorted with c, Ψ is strictly increasing. Therefore we can store Ψ in $n(2 + \log\sigma) + o(n)$ bits by Lemma 1. We also store a bit-vector $P_{S^*}[i]$ indicating that the i-th suffix in the sorted order corresponds to an S*substring.

After the induced-sorting, we obtain $\Psi[i]$ for $i = 2, \ldots, n$ ($\Psi[1]$ is not defined because $SA[1]$ is the last suffix.). To compute names of S*substrings, we use another bit-vector $V[i]$ indicating that the S*substring corresponding to $V[i]$ is different from its left neighbor. To compute V, we scan P_{S^*} to enumerate S*substrings, and if $P_{S^*}[i] = 1$,

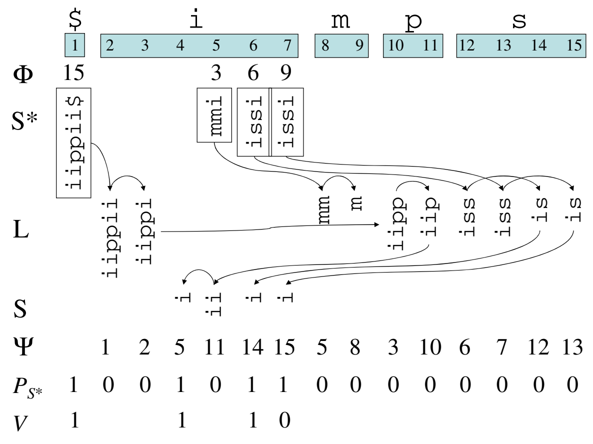

Fig. 2. An example of our modified induced-sorting corresponding to the original one in Figure 1. Positions of S*substrings are stored in Φ. We also store S*substrings in queues. Then L-type suffixes are induced from them. We actually move the substrings among queues, which are illustrated by arrows in the figure. The inverse of the movements are memorized as Ψ. The bit-vector P_{S^*} represents where are the S*substrings stored, and the bit-vector V encodes the names of them.

we compute $i := \Psi[i]$ repeatedly until we reach the position i that corresponds to the head of an S*substring, whose position in T is computed by $\Phi[i]$. We can determine if two adjacent S*substrings in sorted order are different or not in time proportional to their lengths. Therefore computing V takes $O(n)$ time because the total length of all the S*substrings is $O(n)$. If V is ready, we can compute the name of the S*substring corresponding to $P_{S^*}[i]$ by $\mathbf{rank}(P_{S^*}, i)$ which returns the number of 1's in $P_{S^*}[0, i]$. The $\mathbf{rank}$ function is computed in constant time using an $O(n \log\log n / \log n)$-bit auxiliary data structure [16].

5 Succinct Representation of Substring Information

We explain the data structure for storing the list of (prefix of) S*strings. As noted in the previous section, these arrays should support $push\_back(q)$, $pop\_front()$, and $cycle()$ operations. Note that in our algorithm, $A.pop\_front()$ is not called when A is empty. If the length of an S*substring is at most $\log_n \sigma$ bits, we directly store it and the operations $push\_back(q)$ and $pop\_front()$ are done in constant time. Otherwise, instead of storing the S*substring itself, we store the index p_i and the length of the S*substring. Because there exist at most $n \log \sigma / \log n$ S*substrings of length more than $\log_\sigma n$, we can store the indexes and lengths in $O(n \log \sigma)$ bits.

Next, we estimate the size for BWTs in the recursive steps. It seems that in the worst case $n' = n/2$, the naive encoding of names will cost $n' \log n' = \Omega(n \log \sigma)$ bits. To guarantee that the string $T_1 = V_1, V_2, \ldots, V_{n'}$ is encoded in $O(n \log \sigma)$ bits, we use the following encoding of the names, which is summarized as follows.

Lemma 3. *For a string T of length n with alphabet size σ, the shortened string $T_1 = V_1, V_2, \ldots, V_{n'}$ is encoded in $\mathrm{O}(n \log \sigma)$ bits, and given i, the name of V_i and consecutive $\mathrm{O}(\log n)$ bits at any position of the S*substring are computed in constant time. This encoding can be done in $\mathrm{O}(n)$ time during the modified induced-sorting.*

Proof. The encoding consists of two types of codes; one is for S*strings whose lengths are at most $\frac{1}{2}\log_\sigma n$, and the other is for the rest. We call the former *short S*strings* and the latter *long S*strings*. For short S*strings, the code is its binary encoding itself, and for long S*strings the code is their names. To distinghish the type, we use a bit-vector $F[1, n']$ such that $F[i] = 1$ indicates V_i is a long S*string.

For computing the name V_i of a short S*string R_i, we obtain $\frac{1}{2}\log_\sigma n$ bits of T whose position is the beginning of R_i. To compute the name from the $\frac{1}{2}\log_\sigma n$ bits, we construct a decoding table such that for all bit patterns of $\frac{1}{2}\log_\sigma n$ bits which begin with the code of R_i we store the name V_i. This table can be constructed in $\mathrm{O}(n)$ time in the modified induced-sorting. We scan S*strings in lexicographic order, and for each one we obtain its name and its position in T. From T we obtain the code of R_i, and fill a part of the table with the name. The size of the table is $\mathrm{O}(\sigma^{\frac{1}{2}\log_\sigma n}\log n) = \mathrm{O}(\sqrt{n}\log n)$ bits.

For long S*strings, we first construct the bit-vector F and the auxiliary data structure for $\mathbf{rank}$. Then during the modified induced-sorting, if there is a long S*substring R_i, we store its name in an array entry $W[\mathbf{rank}(F, i)]$. Because there exist at most $\frac{n}{\frac{1}{2}\log_\sigma n}$ long S*substrings, we can store their names in $\mathrm{O}(n \log \sigma)$ bits.

To obtain consecutive $\mathrm{O}(\log n)$ bits at any position of the S*substring, we use another bit-vector $G[1, n]$ such that $G[i] = 1$ stands for $T[i]$ is the head of an S*substring. We construct the auxiliary data structure for $\mathbf{select}$. By $\mathbf{select}(G, i)$ we obtain the position of V_i in T. Then it is obvious that any consecutive $\mathrm{O}(\log n)$ bits are obtained in constant time. □

6 Time and Space Analysis

Our algorithm for an input of length n requires $\mathrm{O}(n)$ time and the problem with the half length. Obviously, the time complexity is $\mathrm{O}(n)$.

Next we analyze the space complexity. At the recursive step, the input space is $\mathrm{O}(n \log \sigma)$ bits using the lemma 3. In addition, at each step, we keep the mapping information from the name to the original S*substring. We keep this by using an array list, which requires $n \lg \sigma$ bits of space in the worst case.

After $\lg \lg_\sigma n$ steps, the input length becomes $n' = n/2^{\lg \log_\sigma n} = n/\log_\sigma n$ and the size of suffix array for this input is $n' \lg n' = n/\log_\sigma n(\lg n - \lg \log_\sigma n) < n \lg \sigma$. Therefore we can use SAIS using the space less than $n \lg \sigma$. Therefore the total space is $\mathrm{O}(n \log \sigma \log \log_\sigma n)$ bits.

7 Conclusion

In this paper, we present an algorithm for BWT. Our algorithm directly computes BWT, and does not require suffix arrays. Our algorithm works in linear time, and requires

$O(n \log \sigma \log \log_\sigma n)$ bits of space for any alphabets where n is the length of an original input space, and σ is the alphabet size.

As a next step, we consider how to efficiently build the longest common prefix array, or compressed suffix trees from BWT only. And we are also interested in the problem, whether can we compute BWT in linear time using $2n \lg \sigma + o(n \log \sigma)$ bits only?

References

1. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. Technical report, Digital Equipment Corporation (1994)
2. Farach, M.: Optimal Suffix Tree Construction with Large Alphabets. In: Proc. of FOCS, pp. 137–143 (1997)
3. Ferragina, P., Giancarlo, R., Manzini, G., Sciortino, M.: Compression boosting in optimal linear time. Journal of the ACM 52(4), 688–713 (2005)
4. Grossi, R., Vitter, J.S.: Compressed suffix arryas and suffix trees with applications to text indexing and string matching. Computing 35(2), 378–407 (2005)
5. Hon, W.-K., Lam, T.-W., Sadakane, K., Sung, W.-K., Yiu, S.-M.: A space and time efficient algorithm for constructing compressed suffix arrays. Algorithmica 48(1), 23–36 (2007)
6. Hon, W.K., Sadakane, K., Sung, W.K.: Breaking a Time-and-Space Barrier in Constructing Full-Text Indices. SIAM Journal on Computing 38(6), 2162–2178 (2009)
7. Kärkkäinen, J.: Fast bwt in small space by blockwise suffix sorting. Theoretical Computer Science 387(3), 249–257 (2007)
8. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. ACM 53(6), 918–936 (2006)
9. Kim, D.-K., Sim, J.S., Park, H.-J., Park, K.: Linear-time construction of suffix arrays. In: Baeza-Yates, R., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, pp. 186–199. Springer, Heidelberg (2003)
10. Ko, P., Aluru, S.W.: Space-efficient linear time construction of suffix arrays. Discrete Algorithm 3, 143–156 (2005)
11. Lippert, R.: Space-efficient whole genome comparisons with burrows-wheeler transforms. j. of computational biology. Computational Biology 12(4) (2005)
12. Lippert, R., Mobarry, C., Walenz, B.: A space-efficient construction of the burrows wheeler transform for genomic data. In: Computational Biology (2005)
13. Chae Na, J., Park, K.: Alphabet-independent linear-time construction of compressed suffix arrays using o(nlogn)-bit working space. Theoretical Computer Science 385(1-3), 127–136 (2007)
14. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys 39(1), 2–61 (2007)
15. Nong, G., Zhang, S., Chan, W.H.: Linear suffix array construction by almost pure induced-sorting. In: Proc. of DCC (2009)
16. Raman, R., Raman, V., Rao, S.S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: Proc. of SODA, pp. 233–242 (2002)