When a Dollar Makes a BWT

Sara Giuliani, Zsuzsanna Lipták, and Romeo Rizzi

Dipartimento di Informatica, University of Verona
Strada le Grazie, 15, 37134 Verona, Italy
{zsuzsanna.liptak,romeo.rizzi}@univr.it,
sara.giuliani_01@studenti.univr.it

Abstract. The Burrows-Wheeler-Transform (BWT) is a reversible string transformation which plays a central role in text compression and is fundamental in many modern bioinformatics applications. The BWT is a permutation of the characters, which is in general better compressible and allows to answer several different query types more efficiently than the original string.
It is easy to see that not every string is a BWT image, and exact characterizations of BWT images are known. We investigate a related combinatorial question. In many applications, a sentinel character \$ is added to mark the end of the string, and thus the BWT of a string ending with \$ contains exactly one \$ character. We ask, given a string w, in which positions, if any, can the \$-character be inserted to turn w into the BWT image of a word ending with the sentinel character. We show that this depends only on the standard permutation of w and give a combinatorial characterization of such positions via this permutation. We then develop an $\mathcal{O}(n \log n)$-time algorithm for identifying all such positions, improving on the naive quadratic time algorithm.[1]

Keywords: combinatorics on words, Burrows-Wheeler-Transform, permutations, splay trees, efficient algorithms

1 Introduction

The Burrows-Wheeler-Transform (BWT), introduced by Burrows and Wheeler in 1994 [4], is a reversible string transformation which is fundamental in string compression and is at the core of many of the most frequently used bioinformatics tools [20, 21, 22]. The BWT, a permutation of the characters of the original string, is particularly well compressible if the original string has many repeated substrings, thus making it highly relevant for natural language texts and for biological sequence data. This is due to what is sometimes referred to as *clustering effect* [33]: repeated substrings cause equal characters to be grouped together, resulting in longer runs of the same character than in the original string, and as a result, in higher compressibility.

Given a word (or string) v over a finite ordered alphabet, the BWT is a permutation of the characters of v, such that position i contains the last character of the ith ranked rotation of v, with respect to lexicographic order, among all rotations of v. For example, the BWT of the word `banana` is `nnbaaa`, see Fig. 1 (left). A fundamental property of the BWT is that it is *reversible*, i.e. given w which is the BWT of some word, such a word v can be found in linear time in the length of w [4]. Moreover, v is unique up to rotation.

rotations of `banana`	BWT
`abanan`	`n`
`anaban`	`n`
`ananab`	`b`
`banana`	`a`
`nabana`	`a`
`nanaba`	`a`

rotations of `nanana`	BWT
`ananan`	`n`
`ananan`	`n`
`ananan`	`n`
`nanana`	`a`
`nanana`	`a`
`nanana`	`a`

rotations of `nanana$`	BWT
`$nanana`	`a`
`a$nanan`	`n`
`ana$nan`	`n`
`anana$n`	`n`
`na$nana`	`a`
`nana$na`	`a`
`nanana$`	$

Fig. 1: BWT of the strings `banana`, `nanana` and `nanana$`.

The BWT is defined for every word, even where not all rotations are distinct, as is the case with the word `nanana`, whose BWT is `nnnaaa`, see Fig. 1 (center). (Words for which all rotations are distinct are called *primitive*.) On the other hand, not every word is a BWT image, i.e. not every word is the BWT of some word. For example, `banana` is not the BWT of any word.

It can be decided algorithmically whether a given word w is a BWT image, by slightly modifying the above-mentioned reversing algorithm: if w is not a BWT image, then the algorithm terminates in $\mathcal{O}(n)$ time with an error message (where n is the length of w). Combinatorial characterizations of BWT images are also known [23, 28]: whether w is a BWT image, depends on the number and characteristics of the cycles of its *standard permutation* (see Sec. 2). In particular, w is the BWT image of a primitive word if and only if its standard permutation is cyclic. Moreover, a necessary condition is that the runlengths of w be co-prime [23].

In many situations, it is convenient to append a sentinel character \$ to mark the end of the word v; this sentinel character is defined to be lexicographically smaller than all characters from the given alphabet. For example, BWT(`nanana`\$) = `annnaa`\$, see Fig. 1 (right). Clearly, all rotations of v\$ are distinct, thus, the inverse of the BWT becomes unique, due to the condition that the sentinel character must be at the end of the word. In other words, given a word w with exactly one occurrence of \$, there exists at most one word v such that $w = \mathrm{BWT}(v\$)$.

In this paper, we ask the following combinatorial question: Given a word w over alphabet Σ, in which positions, if any, can we insert the \$-character such that the resulting word is the BWT image of some word v\$? We call such

positions *nice*. Returning to our earlier examples: there are two nice positions for the word `annnaa`, namely 3 and 7: `an$nnaa` and `annnaa$` are BWT images. However, there is none for the word `banana`: in no position can $ be inserted such that the resulting word becomes a BWT image.

We are interested both in characterizing nice positions for a given word w, and in computing them. Note that using the BWT reversing algorithm, these positions can be computed naively in $\mathcal{O}(n^2)$ time. Our results are the following:

- we show that the question which positions are nice depends only on the standard permutation of w;
- we give a full combinatorial characterization of nice positions, via certain subsets which form what we call *pseudo-cycles* of the standard permutation of w; and
- we present an $\mathcal{O}(n \log n)$ time algorithm to compute all nice positions of an n-length word w.

1.1 Related work

The BWT has been subject of intense research in the last two decades, from compression [11, 16, 17, 29], algorithmic [9, 24, 30], and combinatorial [10, 13, 32] points of view (mentioning just a tiny selection from the recent literature). It has also been extended in several ways. One of these, the extended BWT, generalizes the BWT to a multiset of strings [3,26,27], with successful applications to several bioinformatics problems [8,27,31]. A very recent development is the introduction of Wheeler graphs [12], a generalization of a fundamental underlying property of the BWT to data other than strings.

There has been much recent work on inferring strings from different data structures built on strings (sometimes called reverse engineering), and/or just deciding whether such a string exists, given the data structure itself. For instance, this question has been studied for directed acyclic word graphs (DAWGs) and suffix arrays [2], prefix tables [6], LCP-arrays [18], and suffix trees [5, 15, 36]. A number of papers study which permutations are suffix arrays of some string [14, 19, 34], giving a full characterization in terms of the standard permutation.

The analogous question for BWT images was answered fully in [28] for strings over binary alphabets, and in [23] for strings over general alphabets. In the latter publication, the authors also asked the question which strings can be "blown up" to become a BWT: Given the runs (blocks of equal characters) in w, when does a BWT image exist whose runs follow the same order, but each run can be of the same length or longer than the corresponding one in w? The authors fully characterize such strings, showing that the non-existence of a global ascent in w is a necessary and sufficient condition.

Another work treating a related question to ours is [25], where the authors ask and partially answer the question of which strings are fixpoints of the BWT.

Overview The paper is organized as follows. In Section 2, we provide the necessary formal background on the BWT and the standard permutation, and

in Section 3 give a complete characterization of nice positions of a string w. We present our algorithm for computing all nice positions in Section 4. We close with a discussion and outlook in Section 5. All proofs have been omitted due to lack of space, and will be included in the full version of the paper.

2 Basics

Words. Let Σ be a finite ordered alphabet. A *word* (or *string*) over Σ is a finite sequence of elements from Σ (also called *characters*). We write words as $w = w_1 \cdots w_n$, with w_i the ith character, and $|w| = n$ its *length*. Note that we index words from 1. The *empty string* is the only string of length 0 and is denoted ε. The set of all words over Σ is denoted Σ^*. The concatenation $w = uv$ of two words u, v is defined by $w = u_1 \cdots u_{|u|} v_1 \cdots v_{|v|}$. Let $w = uxv$, with u, x, v possibly empty. Then u is called a *prefix*, x a *factor* (or *substring*), and v a *suffix* of w. A factor (prefix, suffix) u of w is called *proper* if $u \neq w$. For a word u and an integer $k \geq 1$, $u^k = u \cdots u$ denotes the k-fold concatenation of u. A word w is called a *primitive* if $w = u^k$ implies $k = 1$.

Two words w, w' are called *conjugates* if there exist words u, v, possibly empty, such that $w = uv$ and $w' = vu$. Conjugacy is an equivalence relation, and the set of all words which are conjugates of w constitute w's *conjugacy class*. Given a word $w = w_1 \cdots w_n$, the *ith rotation* of w is $w_i \cdots w_n w_1 \cdots w_{i-1}$. Clearly, two words are conjugates if and only if one is a rotation of the other.

The set of all words over Σ is totally ordered by the *lexicographic order*: Let $v, w \in \Sigma^*$, then $v \leq_{\text{lex}} w$ if v is a prefix of w, or there exists an index j s.t. for all $i < j$, $v_i = w_i$, and $v_j < w_j$ according to the order on Σ.

In the context of string data structures, it is often necessary to mark the end of words in a special way. To this end, let $\$ \notin \Sigma$ be a new character, called *sentinel*, and set $\$ < a$ for all $a \in \Sigma$. Let $\Sigma_\$^*$ denote the set of all words over Σ with an additional $\$$ at the end. The mapping $w \mapsto w\$$ is a bijection from Σ^* to $\Sigma_\$^*$. Clearly, every word in $\Sigma_\$^*$ is primitive.

Permutations. Let n be a positive integer. A *permutation* is a bijection from $\{1, 2, \ldots, n\}$ to itself. Permutations are often written using the two-line notation $\left(\begin{smallmatrix} 1 & 2 & \ldots & n \\ \pi(1) & \pi(2) & \ldots & \pi(n) \end{smallmatrix}\right)$. A *cycle* in a permutation π is a minimal subset $C \subseteq \{1, \ldots, n\}$ with the property that $\pi(C) = C$. A cycle of length 1 is called a *fixpoint*, and one of length 2 a *transposition*. Every permutation can be decomposed uniquely into disjoint cycles, giving rise to the *cycle representation* of a permutation π, i.e. as a composition of the cycles in the cycle decomposition of π. For example, $\pi = \left(\begin{smallmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 4 & 2 & 5 & 6 & 3 & 1 \end{smallmatrix}\right) = (1\ 4\ 6)(2)(3\ 5)$. Permutations whose cycle decomposition consists of just one cycle are called *cyclic*.

Finally, given a word w, the *standard permutation* of w, denoted σ_w, is the permutation defined by: $\sigma_w(i) < \sigma_w(j)$ if and only if either $w_i < w_j$, or $w_i = w_j$ and $i < j$. For example, the standard permutation of `banana` is $\left(\begin{smallmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 4 & 1 & 5 & 2 & 6 & 3 \end{smallmatrix}\right)$.

Burrows-Wheeler-Transform. It is easiest to define the Burrows-Wheeler-Transform (BWT) via a construction: Let $v \in \Sigma^*$ with $|v| = n$, and let M be an $n \times n$-matrix containing as rows all n rotations of v (not necessarily distinct) in lexicographic order (see Fig. 1). Then $w = \mathrm{BWT}(v)$ is the last column w of M. If v is primitive, then this is equivalent to saying that $w = w_1 \cdots w_n$ such that w_i equals the last character of the jth rotation of v, where the jth rotation has rank i among all rotations of v w.r.t. lexicographic order.

Linear-time construction algorithms of the BWT are well-known [33], and the BWT is *reversible*: given a word w which is the BWT of some word v, v can be recovered from $w = \mathrm{BWT}(v)$ uniquely up to its conjugacy class, again in linear time.

The algorithm for computing v, given $\mathrm{BWT}(v) = w$ is based on the following insights about the matrix M: (1) the last character in each row is the one *preceding* the first in the same row, (2) since the rows are rotations of the same word, every character in the last column occurs also in the first column, (3) the first column lists the characters of v in lexicographical order, and (4) the ith occurrence of character c in the last column of M equals the ith occurrence of character c in the first column. This last property can be used to define a mapping from the last to the first column, called *LF-mapping* [4], which assigns to each position i the corresponding position j in the first column—this is, in fact, the standard permutation of w. Now, if w is the BWT of a word, then such a word v can be reconstructed, from last character to first, via iteratively applying the standard permutation σ_w, and noting that $w_{\sigma_w(i)}$ is the character preceding w_i in v. In other words, $w_1 = v_n$ and $w_{\sigma_w^i(1)} = v_{n-i}$ for $1 \leq i \leq n-1$.

Let $w \in \Sigma^*$, $w = w_1 \cdots w_n$ and $1 \leq i \leq n+1$. We denote by $dol(w, i)$ the $(n+1)$-length word $w_1 \cdots w_{i-1}\$w_i \cdots w_n$, i.e. the word which results from inserting \$ into w in position i. Given a word w' over $\Sigma \cup \{\$\}$ with exactly one occurrence of \$, denote by $undol(w')$ the word which results from deleting the character \$ from w'. We refer to a position i as *nice* if $dol(w, i) \in \mathrm{BWT}(\Sigma_\$^*)$, i.e. if there exists a word $v \in \Sigma^*$ such that $\mathrm{BWT}(v\$) = w$. We can now state our problem:

> **Dollar-BWT Problem:** Given a word $w \in \Sigma^*$, $|w| = n$, compute all nice positions of w, i.e. all $1 \leq i \leq n+1$ such that $dol(w, i) \in \mathrm{BWT}(\Sigma_\$^*)$.

In the next section, we give a combinatorial characterization of nice positions of a word w.

3 A characterization via pseudo-cycles

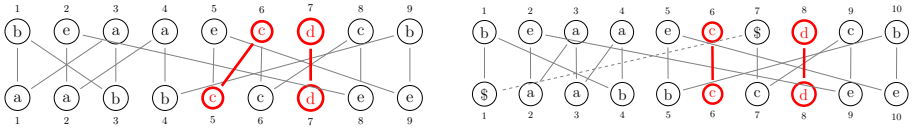
Given $w \in \Sigma^*$, $|w| = n$, and $1 \leq i \leq n+1$, denote by σ_i the standard permutation of $dol(w, i)$. We first note that whether i is nice depends only on σ_i.

Lemma 1. *Given $w \in \Sigma^*$, $|w| = n$, and $1 \leq i \leq n+1$. Then i is nice if and only if σ_i is cyclic.*

We use a bipartite graph G_w to visualize the standard permutation of w (see Fig. 2). The top row corresponds to w, and the bottom row to the characters of w in alphabetical order. When w is a BWT, then this implies that the top row corresponds to the last column of matrix M, and the bottom row to the first. (This graph is therefore sometimes called BWT-graph.) Let us refer to the nodes in the top row as $x_1, \ldots, x_n$ and to those in the bottom row as $y_1, \ldots, y_n$. Nodes x_i are labeled by character w_i, and nodes y_i are labeled by the characters of w in lexicographic order. We connect (x_i, y_j) if and only if $i = j$ or $j = \sigma_w(i)$. It is easy to see that the node set of any cycle S in G_w has the form $\{x_k, y_k \mid k \in \mathcal{I}\}$ for some $\mathcal{I} \subseteq \{1, \ldots, n\}$, and that S is a cycle in G_w if and only if $\mathcal{I}$ is a cycle in σ.

Fig. 2: Standard permutation for $w = \texttt{beaaecdcb}$ with $\sigma = \left(\begin{smallmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 3 & 8 & 1 & 2 & 9 & 5 & 7 & 6 & 4 \end{smallmatrix}\right)$ and $\sigma_7 = \left(\begin{smallmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 4 & 9 & 2 & 3 & 10 & 6 & 1 & 8 & 7 & 5 \end{smallmatrix}\right)$. Red edges cause fixpoints in σ_7.

(a) Standard permutation of w (b) Standard permutation of $dol(w, 7)$



Now observe what happens when we insert a dollar into w in position i. Let $\sigma = \sigma_w$, $w' = dol(w, i)$ and $\sigma_i = \sigma_{w'}$ the resulting permutation. It holds that

$$\sigma_i(j) = \begin{cases} \sigma(j) + 1 & \text{if } j < i, \\ 1 & \text{if } j = i, \text{ and} \\ \sigma(j-1) + 1 & \text{if } j > i. \end{cases} \tag{1}$$

In particular, if j is a fixpoint in σ and $i \leq j$, then $j+1$ will be a fixpoint in σ_i. Similarly, if $\sigma(j) = j-1$ and $j > i$, then j is a fixpoint in σ_i. These two cases are illustrated in Fig. 2, where 7 is a fixpoint in σ and $\sigma(6) = 5$, so the insertion of \$ in position $i = 7$ leads to the two fixpoints 6 and 8 in σ_7. In particular, position 7 is not nice.

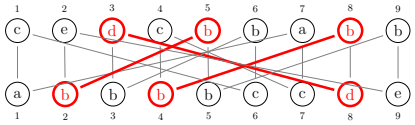
Indeed, this observation can be generalized: if S is a cycle in σ, then no position $i \leq \min S$ is nice. Similarly, if S is such that $\sigma(S) = S - 1 = \{j - 1 \mid j \in S\}$, then no position $i > \max S$ is nice. In both cases, insertion of \$ in such a position would turn S into a cycle. However, the situation can also be more complex, as is illustrated in Fig. 3.

In Theorem 3 we will give a necessary and sufficient condition for creating a proper cycle by inserting a \$ in some position. First we need a definition.

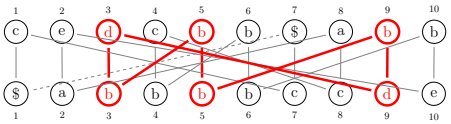
Definition 1. Given a permutation π of $\{1, \ldots, n\}$, a *pseudo-cycle* w.r.t. π is a non-empty subset $S \subseteq \{1, \ldots, n\}$ which can be partitioned into two subsets S_{left}

Fig. 3: Standard permutation of $w = \texttt{cedcbbabb}$ with $\sigma = \left(\begin{smallmatrix}1&2&3&4&5&6&7&8&9\\6&9&8&7&2&3&1&4&5\end{smallmatrix}\right)$ and of $dol(w, 7)$ with $\sigma_7 = \left(\begin{smallmatrix}1&2&3&4&5&6&7&8&9&10\\7&10&9&8&3&4&1&2&5&6\end{smallmatrix}\right)$.

(a) Pseudo-cycle in the standard permutation of w



(b) Cycle in the standard permutation of $dol(w, 7)$



and S_{right}, possibly empty, such that $S_{\text{left}} < S_{\text{right}}$, and $\pi(S) = (S_{\text{left}} - 1) \cup S_{\text{right}}$. Let $a = \max S_{\text{left}}$, and $a = 0$ if S_{left} is empty. Further, let $b = \min S_{\text{right}}$, and $b = n + 1$ if S_{right} is empty. The *critical interval* $R \subseteq \{1, 2, \ldots, n + 1\}$ of the pseudo-cycle S is defined as $R = [a + 1, b]$.

For example, in Fig. 3, $S = \{3, 5, 8\}$ is a pseudo-cycle, with $S_{\text{left}} = \{3, 5\}$, $S_{\text{right}} = \{8\}$, and $R = \{6, 7, 8\}$. Note that every cycle C of a permutation is a pseudo-cycle, with $C = C_{\text{right}}$. In Fig. 2, we highlighted two pseudo-cycles: $S_1 = \{6\}$, with critical interval $R_1 = \{7, 8, 9, 10\}$, and $S_2 = \{7\}$, with $R_2 = \{1, 2, 3, 4, 5, 6, 7\}$. The elements of the critical interval are exactly those positions i which, when the \$ is inserted in position i, turn S into one or more cycles, as we will see in Lemma 2. In particular, putting $S = \{1, \ldots, n\}$, we get that $i = 1$ is never nice. This is easy to see since (1) is a cycle in the standard permutation of $\$w$ for every word w.

Definition 2. Let $1 \leq i \leq n + 1$ and $S \subseteq \{1, 2, \ldots, n\}$. We define

$$\begin{aligned} \mathit{shift}(S, i) &= \{x \mid x \in S \text{ and } x < i\} \cup \{x + 1 \mid x \in S \text{ and } x \geq i\}, \text{ and} \\ \mathit{unshift}(S, i) &= \{x \mid x \in S \text{ and } x < i\} \cup \{x - 1 \mid x \in S \text{ and } x > i\}. \end{aligned}$$

The next lemma states the correspondence between cycles and pseudo-cycles.

Lemma 2. *Let $w \in \Sigma^*$ and $\sigma = \sigma_w$. Let $1 \leq i \leq n + 1$, and $U \subseteq \{1, 2, \ldots, n + 1\} \setminus \{i\}$. Then U is a cycle in the permutation σ_i if and only if $S = \mathit{unshift}(U, i)$ is a pseudo-cycle w.r.t. σ, and i belongs to the critical interval of S.*

Theorem 3. *Let w be a word of length n over Σ. A position i, $1 \leq i \leq n + 1$, is nice if and only if there is no pseudo-cycle S w.r.t. the standard permutation $\sigma = \sigma_w$ whose critical interval contains i.*

With Theorem 3, we can now prove the statements about our first example strings `banana` and `annnaa`. The word `banana` has the pseudo-cycles $S_1 = \{2\}$ with critical interval $R_1 = \{3, 4, 5, 6, 7\}$; and $S_2 = \{3, 5, 6\}$ with $R_2 = \{1, 2, 3\}$. Therefore, every position is contained in some critical interval. For the word `annnaa`, we have $S_1 = \{1\}$ with critical interval $R_1 = \{1\}$; $S_2 = \{2, 3, 4, 5, 6\}$

with $R_2 = \{1, 2\}$; $S_3 = \{3, 5\}$ with $R_3 = \{4, 5\}$; $S_4 = \{4, 6\}$ with $R_4 = \{5, 6\}$; and all other pseudo-cycles are unions of these. The two positions 3 and 7 are not contained in any critical interval, and are therefore nice. In fact, `an$nnaa` = BWT(`ananna$`), and `annnaa$` = BWT(`nanana$`).

4 Algorithm

It is easy to compute all nice positions, given a word w, by inserting \$ in each position i and running the BWT reverse algorithm, in a total of $\mathcal{O}(n^2)$ time. Here we present an $\mathcal{O}(n \log n)$ time algorithm for the problem.

The underlying idea is that, if we know σ_i, the standard permutation of $dol(w, i)$, then it is not too difficult to compute σ_{i+1}. We summarize the underlying facts in the following lemma.

Lemma 4. *Let $w \in \Sigma^*$, $|w| = n$, and for $1 \leq i \leq n+1$, let σ_i be the standard permutation of $dol(w, i)$. Then*

1. *$\sigma_1(1) = 1$ and for $i > 1$, $\sigma_1(i) = \sigma(i-1) + 1$,*
2. *for $1 \leq i \leq n$, σ_i and σ_{i+1} differ only in the points i and $i+1$: $\sigma_{i+1}(i) = \sigma_i(i+1)$ and $\sigma_{i+1}(i+1) = \sigma_i(i)$,*
3. *$\sigma_i(i) = 1$ for all i.*

Therefore, we have that $\sigma_{i+1} = (1, \sigma_i(i+1)) \cdot \sigma_i$, i.e. the standard permutation σ_{i+1} is the result of applying a transposition on σ_i. As we show next, applying a transposition on a permutation has either the effect of splitting a cycle, or that of merging two cycles.

Lemma 5. *Let $\pi = C_1 \cdots C_k$ be the cycle decomposition of the permutation π, $x \neq y$, and $\pi' = (\pi(x), \pi(y)) \cdot \pi$.*

1. *If x and y are in the same cycle C_i, then this cycle is split into two. In particular, let $C_i = (c_1, c_2, \ldots, c_j, \ldots, c_m)$, with $c_m = x$ and $c_j = y$. Then $\pi' = (c_1, c_2, \ldots, c_j)(c_{j+1} \ldots c_m) \prod_{\ell \neq i} C_\ell$.*
2. *If x and y are in different cycles C_i and C_j, then these two cycles are merged. In particular, let $C_i = (c_1, c_2, \ldots, c_m)$, with $c_m = x$, and $C_j = (c'_1, c'_2, \ldots, c'_r)$, with $c'_r = y$, then $\pi' = (c_1, c_2, \ldots, c_m, c'_1, c'_2, \ldots, c'_r) \prod_{\ell \neq i,j} C_\ell$.*

Example 1. $w =$ `acccbccbab`

$$\sigma = \left(\begin{smallmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 1 & 6 & 7 & 8 & 3 & 9 & 10 & 4 & 2 & 5 \end{smallmatrix}\right) = (1)(2, 6, 9)(3, 7, 10, 5)(4, 8)$$

$$\sigma_1 = \left(\begin{smallmatrix} \mathbf{1} & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ \mathbf{1} & 2 & 7 & 8 & 9 & 4 & 10 & 11 & 5 & 3 & 6 \end{smallmatrix}\right) = (\mathbf{1})(\mathbf{2})(3, 7, 10)(4, 8, 11, 6)(5, 9) \qquad \textit{merge}$$

$$\sigma_2 = \left(\begin{smallmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ \mathbf{2} & \mathbf{1} & 7 & 8 & 9 & 4 & 10 & 11 & 5 & 3 & 6 \end{smallmatrix}\right) = (1, \mathbf{2})(\mathbf{3}, 7, 10)(4, 8, 11, 6)(5, 9) \qquad \textit{merge}$$

$$\sigma_3 = \left(\begin{smallmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 2 & \mathbf{7} & \mathbf{1} & 8 & 9 & 4 & 10 & 11 & 5 & 3 & 6 \end{smallmatrix}\right) = (1, 2, 7, 10, \mathbf{3})(\mathbf{4}, 8, 11, 6)(5, 9) \qquad \textit{merge}$$

$$\sigma_4 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 2 & 7 & \mathbf{8} & \mathbf{1} & 9 & 4 & 10 & 11 & 5 & 3 & 6 \end{pmatrix} = (1, 2, 7, 10, 3, 8, 11, 6, \mathbf{4})(\mathbf{5}, 9) \qquad \textit{merge}$$

$$\boxed{\sigma_5 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 2 & 7 & 8 & \mathbf{9} & \mathbf{1} & 4 & 10 & 11 & 5 & 3 & 6 \end{pmatrix} = (1, 2, 7, 10, 3, 8, 11, \mathbf{6}, 4, 9, \mathbf{5})} \qquad \textit{split}$$

$$\sigma_6 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 2 & 7 & 8 & 9 & \mathbf{4} & \mathbf{1} & 10 & 11 & 5 & 3 & 6 \end{pmatrix} = (1, 2, \mathbf{7}, 10, 3, 8, 11, \mathbf{6})(4, 9, 5) \qquad \textit{split}$$

$$\sigma_7 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 2 & 7 & 8 & 9 & 4 & \mathbf{10} & \mathbf{1} & 11 & 5 & 3 & 6 \end{pmatrix} = (1, 2, \mathbf{7})(10, 3, \mathbf{8}, 11, 6)(4, 9, 5) \qquad \textit{merge}$$

$$\sigma_8 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 2 & 7 & 8 & 9 & 4 & 10 & \mathbf{11} & \mathbf{1} & 5 & 3 & 6 \end{pmatrix} = (1, 2, 7, 11, 6, 10, 3, \mathbf{8})(4, \mathbf{9}, 5) \qquad \textit{merge}$$

$$\boxed{\sigma_9 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 2 & 7 & 8 & 9 & 4 & 10 & 11 & \mathbf{5} & \mathbf{1} & 3 & 6 \end{pmatrix} = (1, 2, 7, 11, 6, \mathbf{10}, 3, 8, 5, 4, \mathbf{9})} \qquad \textit{split}$$

$$\sigma_{10} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 2 & 7 & 8 & 9 & 4 & 10 & 11 & 5 & \mathbf{3} & \mathbf{1} & 6 \end{pmatrix} = (1, 2, 7, \mathbf{11}, 6, \mathbf{10})(3, 8, 5, 4, 9) \qquad \textit{merge}$$

$$\sigma_{11} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 2 & 7 & 8 & 9 & 4 & 10 & 11 & 5 & 3 & \mathbf{6} & \mathbf{1} \end{pmatrix} = (1, 2, 7, 11)(6, 10)(3, 8, 5, 4, 9)$$

In the example, changes from one permutation to the next are marked in red. At iteration i, the result of the transposition of $\sigma_{i-1}(i-1)$ and $\sigma_{i-1}(i)$ is marked in red in the two-line notation, while in the cyclic notation, we mark in red the positions of i and $i+1$. Finally, the boxes highlight cyclic σ_i, namely i such that $dol(w, i) \in \mathrm{BWT}(\Sigma^*_\$)$.

4.1 High-level description of algorithm

The algorithm first computes the standard permutation σ of w and initializes a counter c with the number of cycles of σ. It then computes σ_1 according to Lemma 4, part 1. It increments counter c by 1 for $i = 1$, since σ_1 always has a fixpoint (1). Then the algorithm iteratively computes the new permutation σ_{i+1}, updating c in each iteration. By Lemma 5, c either increases or decreases by 1 in every iteration: it increases if $i+1$ is in the same cycle as i, and it decreases if it is in a different cycle. Whenever c equals 1, the algorithm reports the current value i. See Algorithm 1 for the pseudocode.

4.2 Implementation with splay trees

For the algorithm's implementation, we need an appropriate data structure for maintaining and updating the current permutation σ_i. Using an array to keep σ_i would allow us to update it in constant time in each step, but would not give us the possibility to efficiently decide whether $i+1 \in C$ in line 11. Thus we need a data structure to maintain the cycles of σ_i. The functionalities we seek are (a) decide whether two elements are in the same cycle, (b) split two cycles, (c) merge two cycles. The data structure we have chosen is a forest of splay trees [35]. This data structure supports the above operations in amortized $\mathcal{O}(\log n)$ time.

Algorithm 1: FINDNICEPOSITIONS(w)

Given a word w, return a set $\mathcal{I}$ of positions in which the \$-character can be inserted to turn w into a BWT image.

```
1  n ← |w|
2  σ ← standard permutation of w        // variant of counting sort
3  c ← number of cycles of σ
4  I ← ∅
5  for i ← n + 1 down to 2 do                    // compute σ_1 from σ
6      σ(i) ← σ(i − 1)
7  σ(1) ← 1
8  c ← c + 1                // σ_1 has one more cycle than σ
9  for  i ← 1 to n do
10     C ← cycle of σ which contains 1        // C also contains i
11     if i + 1 ∈ C then
12         c ← c + 1                        // case split
13     else
14         c ← c − 1                        // case merge
15     σ ← UPDATE(σ, i)                      // now σ = σ_{i+1}
16     if  c = 1 then
17         I ← I ∪ {i + 1}
18 return I
19 procedure UPDATE(σ, i):                                   // σ(i) = 1
20     σ(i) ← σ(i + 1)
21     σ(i + 1) ← 1
22     return σ
```

Splay trees are self-adjusting binary search trees. They are not necessarily balanced, but they have the property that at every access-operation, the element x accessed is moved to the root and the tree is adjusted in such a way as to move nodes on the path from the root to x closer to the root, thus reducing access-time to these nodes for future operations. The operation of self-adjusting, called *splaying*, consists of a series of the usual edge rotations in binary search trees. Which rotations are applied depends on the position of the node with respect to its parent and grandparent (the cases are referred to as *zig*, *zig-zig*, and *zig-zag*). Splay trees can implement the standard operations on binary search trees, such as *access, insert, delete, join, split* in amortized logarithmic time, in the total number of nodes involved. We refer the reader to the original article [35] for more details.

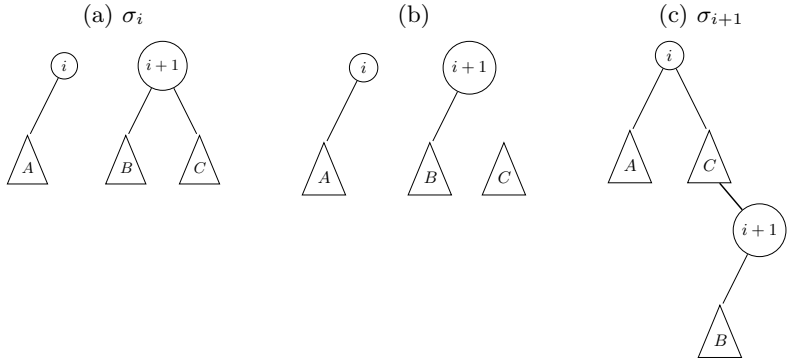
We represent the current permutation σ_i as a forest of splay trees, where each tree corresponds to a cycle of σ_i. Let $(c_1, c_2, \ldots, c_k)$ be an arbitrary rotation of a cycle in σ_i. We consider the cycle as a ranked list of the elements from c_1 to c_k and assign element c_j its position j as key. Doing this we can build the splay tree of the cycle keying the elements by their position in the cycle: for a node v of the tree, the elements of the list which come before v are contained in the left subtree of v, and the elements which come after in the right subtree of v.

Note that by construction, we will necessarily have 1 as left-most node and i as right-most node in the first cycle of σ_i.

We now explain how to update the data structure.

If i and $i+1$ are in distinct cycles of σ_i, then the transposition of $\sigma_i(i)$ and $\sigma_i(i+1)$ leads to the merge of their cycles (Lemma 5). We show the implementation of the merge-step using splay trees in Fig. 4. Let the two cycles have the following form $(1, A, i)$ resp. $(B, i+1, C)$, with A, B, C sequences of numbers. We show in (a) the corresponding splay trees rooted in i resp. in $i+1$, after the *access* operations on the two elements involved, which move i and $i+1$ to the roots of their respective trees. Now we have a *split* of the right subtree of node $i+1$, the result of which is shown in (b). Next, a *join* operation links the C subtree to the node i as its right child.

Fig. 4: The implementation of the merge-step with splay trees.



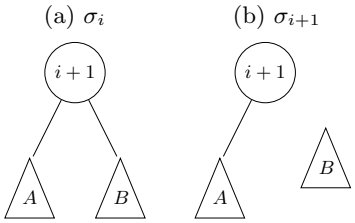
If i and $i+1$ are in the same cycle of σ_i, then the transposition of $\sigma_i(i)$ and $\sigma_i(i+1)$ leads to a split of the cycle (Lemma 5). We show the implementation of this operation with splay trees in Fig. 5. Let the cycle have the form $(1, A, i+1, B, i)$. The corresponding splay tree after *access* $i+1$ is shown in (a). The *split* operation cuts the right subtree of $i+1$ producing the two new trees in (b).

4.3 Analysis

We now show that Algorithm 1 takes $\mathcal{O}(n \log n)$ time in the worst case.

Computing the standard permutation of w takes $\mathcal{O}(n)$ time (using a variant of Counting Sort [7], and noting that the alphabet of the string has cardinality at most n). The computation of σ_1 (lines 5 to 7) takes $\mathcal{O}(n)$ time. All steps in one iteration of the for-loop (lines 9 to 17) take constant time, except deciding whether $i+1 \in C$ (line 11), and updating σ (line 15). For deciding whether $i+1 \in C$, we *access* $i+1$. If the answer is YES, we will have a split-step: this is a *split*-operation on the tree for C (Fig. 5). If the answer is NO, then we *access*

Fig. 5: The implementation of the split-step with splay trees.



i, and merge the two trees (Fig. 4); the implementation of this consists of one *split*- and two *join*-operations on the trees.

Therefore, in one iteration of the for-loop, we either have one *access* and one *split* operation (for a split-step), or two *access*-, one *split*-, and two *link*-operations (merge-step), thus in either case, at most five operations. There are n iterations of the for-loop, so at most $5n$ operations. Together with the initial insertion of the $n+1$ nodes, we get a total of $6n$ operations. We report the relevant theorem:

Theorem 6 (Balance Theorem with Updates, Thm. 6 in [35]). *A sequence of m arbitrary operations on a collection of initially empty splay trees takes $\mathcal{O}(m + \sum_{j=1}^{m} \log n_j)$ time, where n_j is the number of items in the tree or trees involved in operation j.*

For our algorithm, we have $m = \mathcal{O}(n)$ operations altogether, each involving no more than $n+1$ nodes, thus Theorem 6 guarantees that the total time spent on the splay trees is $\mathcal{O}(n + n \log n)$. Adding to this the computation of σ_1 and the initialization of the splay trees, each in $\mathcal{O}(n)$ time, and of the constant-time operations within the for-loop, we get altogether $\mathcal{O}(n \log n)$ time. Memory usage is $\mathcal{O}(n)$, since the forest of splay trees consists of $n + 1$ vertices in total. We summarize in the following theorem:

Theorem 7. *Algorithm 1 runs in $\mathcal{O}(n \log n)$ time and uses $\mathcal{O}(n)$ space, for an input string of length n.*

5 Conclusion

In this paper, we studied a combinatorial question on the Burrows-Wheeler transform, namely in which positions (called *nice* positions) the sentinel character can be inserted in order to turn a given word w into a BWT image. We developed a combinatorial characterization of nice positions and presented an efficient algorithm to compute all nice positions in the word.

Ongoing work includes developing conditions on nice positions which can be derived directly from the word and tested in linear time. For example, we can show that all nice positions have the same parity, and we are able to give lower bounds on nice positions. These results are based on properties of the standard permutation of the original word.

References

1. Aigner, M.: Discrete mathematics. Oxford University Press (2007)
2. Bannai, H., Inenaga, S., Shinohara, A., Takeda, M.: Inferring Strings from Graphs and Arrays. In: 28th International Symposium on Mathematical Foundations of Computer Science 2003, MFCS 2003. Lecture Notes in Computer Science, vol. 2747, pp. 208–217 (2003)
3. Bonomo, S., Mantaci, S., Restivo, A., Rosone, G., Sciortino, M.: Suffixes, conjugates and Lyndon words. In: International Conference on Developments in Language Theory. pp. 131–142 (2013)
4. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. Tech. rep., DIGITAL System Research Center (1994)
5. Cazaux, B., Rivals, E.: Reverse engineering of compact suffix trees and links: A novel algorithm. J. Discrete Algorithms **28**, 9–22 (2014)
6. Clément, J., Crochemore, M., Rindone, G.: Reverse engineering prefix tables. In: 26th International Symposium on Theoretical Aspects of Computer Science, STACS 2009. pp. 289–300 (2009)
7. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT Press (2009)
8. Cox, A.J., Jakobi, T., Rosone, G., Schulz-Trieglaff, O.: Comparing DNA sequence collections by direct comparison of compressed text indexes. In: 12th International Workshop on Algorithms in Bioinformatics, WABI 2012. Lecture Notes in Computer Science, vol. 7534, pp. 214–224 (2012)
9. Crochemore, M., Grossi, R., Kärkkäinen, J., Landau, G.M.: Computing the Burrows-Wheeler transform in place and in small space. J. Discrete Algorithms **32**, 44–52 (2015)
10. Daykin, J.W., Groult, R., Guesnet, Y., Lecroq, T., Lefebvre, A., Léonard, M., Prieur-Gaston, É.: A survey of string orderings and their application to the Burrows–Wheeler transform. Theor. Comput. Sci., **710**, 52–65 (2018)
11. Ferragina, P., Giancarlo, R., Manzini, G., Sciortino, M.: Boosting textual compression in optimal linear time. J. ACM **52**(4), 688–713 (2005)
12. Gagie, T., Manzini, G., Sirén, J.: Wheeler graphs: A framework for BWT-based data structures. Theor. Comput. Sci. **698**, 67–78 (2017)
13. Giancarlo, R., Restivo, A., Sciortino, M.: From first principles to the Burrows and Wheeler transform and beyond, via combinatorial optimization. Theor. Comput. Sci. **387**(3), 236–248 (2007)
14. He, M., Munro, J.I., Rao, S.S.: A categorization theorem on suffix arrays with applications to space efficient text indexes. In: 16th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005. pp. 23–32 (2005)
15. I, T., Inenaga, S., Bannai, H., Takeda, M.: Inferring strings from suffix trees and links on a binary alphabet. Discrete Applied Mathematics **163**, 316–325 (2014)
16. Kaplan, H., Landau, S., Verbin, E.: A simpler analysis of Burrows–Wheeler-based compression. Theor. Comput. Sci.
17. Kaplan, H., Verbin, E.: Most Burrows-Wheeler based compressors are not optimal. In: Annual Symposium on Combinatorial Pattern Matching, CPM 2007. pp. 107–118 (2007)
18. Kärkkäinen, J., Piatkowski, M., Puglisi, S.J.: String Inference from Longest-Common-Prefix Array. In: 44th International Colloquium on Automata, Languages, and Programming, ICALP 2017. pp. 62:1–62:14 (2017)

19. Kucherov, G., Tóthmérész, L., Vialette, S.: On the combinatorics of suffix arrays. Inf. Process. Lett. **113**(22-24), 915–920 (2013)
20. Lam, T.W., Li, R., Tam, A., Wong, S., Wu, E., Yiu, S.M.: High Throughput Short Read Alignment via Bi-directional BWT. In: 2009 IEEE International Conference on Bioinformatics and Biomedicine. pp. 31–36 (2009)
21. Langmead, B., Trapnell, C., Pop, M., Salzberg, S.L.: Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. Genome Biology **10**(3), R25 (2009)
22. Li, H., Durbin, R.: Fast and accurate long-read alignment with Burrows–Wheeler transform. Bioinformatics **26**(5), 589–595 (01 2010)
23. Likhomanov, K.M., Shur, A.M.: Two Combinatorial Criteria for BWT Images. In: 6th International Computer Science Symposium in Russia - Theory and Applications, CSR 2011. pp. 385–396 (2011)
24. da Louza, F.A., Gagie, T., Telles, G.P.: Burrows-Wheeler transform and LCP array construction in constant space. J. Discrete Algorithms **42**, 14–22 (2017)
25. Mantaci, S., Restivo, A., Rosone, G., Russo, F., Sciortino, M.: On Fixed Points of the Burrows-Wheeler Transform. Fundam. Inform. **154**(1-4), 277–288 (2017)
26. Mantaci, S., Restivo, A., Rosone, G., Sciortino, M.: An extension of the Burrows-Wheeler Transform. Theor. Comput. Sci. **387**(3), 298–312 (2007)
27. Mantaci, S., Restivo, A., Rosone, G., Sciortino, M.: A new combinatorial approach to sequence comparison. Theory of Computing Systems **42**(3), 411–429 (2008)
28. Mantaci, S., Restivo, A., Sciortino, M.: Burrows–Wheeler transform and Sturmian words. Inf. Proc. Letters **86**(5), 241–246 (2003)
29. Manzini, G.: An analysis of the Burrows-Wheeler transform. J. ACM **48**(3), 407–430 (2001)
30. Policriti, A., Prezza, N.: LZ77 computation based on the run-length encoded BWT. Algorithmica **80**(7), 1986–2011 (2018)
31. Prezza, N., Pisanti, N., Sciortino, M., Rosone, G.: SNPs detection by eBWT positional clustering. Algorithms for Molecular Biology **14**(1), 3:1–3:13 (2019)
32. Restivo, A., Rosone, G.: Balancing and clustering of words in the Burrows-Wheeler transform. Theor. Comput. Sci. **412**(27), 3019–3032 (2011)
33. Rosone, G., Sciortino, M.: The Burrows-Wheeler transform between data compression and combinatorics on words. In: Conference on Computability in Europe. pp. 353–364 (2013)
34. Schürmann, K., Stoye, J.: Counting suffix arrays and strings. Theor. Comput. Sci. **395**(2-3), 220–234 (2008)
35. Sleator, D.D., Tarjan, R.E.: A Data Structure for Dynamic Trees. In: 13th Annual ACM Symposium on Theory of Computing. pp. 114–122. STOC 1981, ACM (1981)
36. Starikovskaya, T.A., Vildhøj, H.W.: A suffix tree or not a suffix tree? J. Discrete Algorithms **32**, 14–23 (2015)