Backwards Search in Context Bound Text Transformations

Matthias Petri
RMIT University
School of CS&IT
matthias.petri@rmit.edu.au

Gonzalo Navarro
University of Chile
Dept. of Computer Science
gnavarro@dcc.uchile.cl

J. Shane Culpepper
RMIT University
School of CS&IT
shane.culpepper@rmit.edu.au

Simon J. Puglisi
RMIT University
School of CS&IT
simon.puglisi@rmit.edu.au

***Abstract*—The Burrows-Wheeler Transform (BWT) is the basis for many of the most effective compression and self-indexing methods used today. A key to the versatility of the BWT is the ability to search for patterns directly in the transformed text. A backwards search for a pattern P can be performed on a transformed text by iteratively determining the range of suffixes that match P. The search can be further enhanced by constructing a wavelet tree over the output of the BWT in order to emulate a suffix array. In this paper, we investigate new algorithms for search derived from a variation of the BWT whereby rotations are only sorted to a depth k, commonly referred to as a context bound transform. Interestingly, this BWT variant can be used to mimic a k-gram index, which are used in a variety of applications that need to efficiently return occurrences in text position order. In this paper, we present the first backwards search algorithms on the k-BWT, and show how to construct a self-index containing many of the attractive properties of a k-gram index.**

***Keywords*-backwards search; text indexing; BWT; context bound transform, k-gram index;**

I. Introduction

The *pattern matching problem* is defined as finding all occurrences of a pattern $P = p_1 p_2 \dots p_m$ in a text $T = t_1 t_2 \dots t_n$ where both T and P are sequences over a finite alphabet Σ of size σ. The traditional problem of pattern matching has undergone a revival in recent years, due mainly to significant advances in a new class of algorithms capable of indexing large text collections [1], [2], [3]. *Self-indexes* are now capable of searching for any pattern P of length m in time linear to m, and can reproduce any segment of the original text using only the index. Self-indexes are a synthesis of several fundamental text compression and search algorithms.

Many self-indexing methods are derived from a *suffix array* (SA) [4]. An SA supports search over a text T by sorting all suffixes in lexicographical order. The BWT permutes a text T into a more compressible representation. Remarkably, the output of the BWT can be used in conjunction with a *wavelet tree* [5] to emulate the search capabilities of a suffix array while using space proportional to the compressed representation of T. In order to emulate an SA search, a wavelet tree is constructed over the BWT output so that jumps between lexicographically ordered suffixes in T is possible. To perform the context jumps, it is necessary to determine the symbol at position $j-1$ in T for a starting position j. This process is used to perform backwards search for any pattern P. To do this efficiently, self-indexes exploit the duality between an SA and a BWT over T.

Prior to the discovery of the searching capabilities of the BWT, a folklore method for reducing space called a k-gram index was commonly used for substring searches [6], [7], [8], [9]. A k-gram index records the occurrences of each distinct substring of length k in an attempt to mimic the efficiency of inverted indexes. The k-gram index is still commonly used today to support substring search in text that is not amenable to term segmentation and extraction. While suffix arrays (and the BWT) are often more convenient when searching for general patterns, there are various applications where using a k-gram index is still advantageous. Since text suffixes need to be sorted only up to the first k symbols, a k-gram index can be built using less space and time, and are more I/O-friendly than full suffix arrays. Also, searches for patterns of a fixed length k can be performed very efficiently are returned with all occurrences by increasing text position order. However, a suffix array, returns positions lexicographically ordered by the suffix that follows the occurrences, which often requires additional processing.

An example where a k-gram index may be more convenient than a suffix array is for indexed approximate searching [10]. Backtracking in a suffix array is possible, but the time is exponential in the error threshold allowed. A more efficient solution is to split the pattern and search for each subpattern using a lower threshold (or even exactly). Using this approach, the full search can be completed by backtracking or by generating a "neighborhood" of all possible k-grams that match the subpattern within the error threshold. The approximate occurrences of the subpattern must be merged, and the occurrence lists for the distinct pieces are (essentially) intersected to find areas where an occurrence may be present. False matches must be filtered out from each possible occurrence using an on-line pattern matching algorithm such as the Boyer-Moore-Horspool algorithm [11]. This process requires subpatterns of a fixed length k, and having potential matches returned in text position order is vital to efficient intersection and union operations. A suffix array needs additional query time and space to sort the possible occurrences in text order. In fact,

one of the most prolific genomic search systems, BLAST, is reliant on a k-gram index and not SA based algorithms for this reason [12]. The k of interest in BLAST is around 11–12 for DNA, and 3–4 for protein sequences. However, the space cost to explicitly store all possible k-grams grows exponentially with k, limiting the substring segment sizes used in practice.

While FM-indexes utilize BWT to achieve suffix array compression and self-indexing [1], [2], [3], k-gram indexes have not received much attention beyond applying the classical techniques for compressing inverted lists [13], despite being widely used in practice.

In this paper, we explore a variant of self-indexing derived from k-grams. We build on a common alternative to the BWT, known as a context bound block sorting transformation (k-BWT). In a BWT, the symbols in T are fully sorted in lexicographical order of contexts. A context bound block sorting transformation only sorts the suffixes in T up to a certain length k. We explore the potential of the k-BWT as a self-index representation of text that offers k-gram index search capabilities. We present the first backwards search algorithm for k-BWT permuted text, showing that it is possible to search for patterns in the same way as in a k-gram index, retrieving the occurrences in text position order. We also show how to recover any substring of the original text. Somewhat surprisingly, operations within k-BWT appear to be significantly more challenging than on a full BWT, and so our solution trades some compression effectiveness when k is small. Our space performance improves for larger k values, whereas compression using a classical k-gram approach worsens as k increases. We present the first steps in this new direction and regard our work as opening an interesting and challenging area, with practical potential applications.

II. Notation

Throughout this paper we consider a text $T = T[0, n-1]$ of n symbols. The symbols in T are drawn from an alphabet Σ with $\sigma = |\Sigma|$. The last symbol in T is \$, an end of string marker defined to be lexicographically smaller than any $s \in \Sigma$. We define the string $T_i = T[i, n-1]T[0, i-1]$ as the i^{th} ***rotation*** of T. The substring $T[i, n-1]$ is referred to as the i^{th} ***suffix*** of T, or "suffix i". Because the unique \$ we can unambiguously refer to a rotation by the suffix of T by which it is prefixed: Rotation i is prefixed with suffix i. The BWT is formed with the characters preceding the lexicographically sorted rotations. Let $T_{i_0}, T_{i_1}, \ldots, T_{i_{n-1}}$ be the rotations in lexicographic order, then $T^{\text{BWT}} = T[i_0 - 1 \bmod n]T[i_1 - 1 \bmod n] \ldots T[i_{n-1} - 1 \bmod n]$.

The k-BWT depends upon a partial sort of the rotations of T, partial because it is based on an ordering of the prefixes of these rotations that are of length $k \geq 1$. We refer to this partial ordering as a ***k-ordering*** of rotations into ***k-order***, and to the process itself as a ***k-sort***. If two or more rotations are equal under k-order, we say that they fall into the same ***k-group***; they are accordingly said to be ***k-equal***. Throughout this paper we always assume a k-sort is ***stable***, so that within each k-group, the ordering of the rotations is the order of their starting positions in T. We refer to the output of the k-BWT as $T^{k\text{-BWT}}$.

III. Algorithmic Framework

Many succinct data structures depend on two basic operations over a bitvector $B[0, n-1]$:

$\text{RANK}_{0/1}(B, i)$: return the number of 0's/1's in $B[0, i]$

$\text{SELECT}_{0/1}(B, i)$: return the position of the i^{th} of 0's/1's in B

Both operations can be performed in constant time. A simple constant time $\text{RANK}_{0/1}$ solution uses $o(n)$ space in addition to storing B [14]. Efficient RANK_c and SELECT_c over an alphabet of size $\sigma > 2$ can be performed using a wavelet tree [5]. A wavelet tree decomposes the RANK_c and SELECT_c operations over $[0, \sigma - 1]$ into $\text{RANK}_{0/1}$ and $\text{SELECT}_{0/1}$ operations on a binary alphabet using a binary tree. The root of the tree represents the whole alphabet. Its children represent each half of the alphabet of the parent node. Each leaf node in the tree represents one symbol in $[0, \sigma - 1]$. When answering the RANK_c query for a specific symbol c, we perform $\text{RANK}_{0/1}$ operations at each level in the tree until we arrive at the leaf node representing c. The overall $\text{RANK}_\text{c}(T, i)$ can be computed by combining the $\text{RANK}_{0/1}$ results at each tree level in $\mathcal{O}(\log \sigma)$ time. Any symbol $T[i]$ is also computed in time $\mathcal{O}(\log \sigma)$ with a similar algorithm; we call this operation $\text{ACCESS}(T, i)$. Using a succinct representation of $\text{RANK}_{0/1}$ and $\text{SELECT}_{0/1}$ [15], a wavelet tree requires $nH_0 + o(n \log \sigma)$ bits of space, where $H_0 \leq \log \sigma$ is the zero-order entropy of T.[1] A wavelet tree built on T^{BWT} uses $nH_k + o(n \log \sigma)$ bits [16] for any $k \leq \alpha \log_\sigma(n) - 1$ and constant $\alpha < 1$. Here $H_k \leq H_{k-1} \leq \ldots \leq H_0 \leq \log \sigma$ is the k-th order entropy of T [17], a lower bound to the performance of any compressor using k-th order statistical modeling on T. The same space bound holds on $T^{k\text{-BWT}}$, where we only k-sort the rotations. A suffix array $\text{SA}[0, n-1]$ over T stores the offsets to all suffixes in T in lexicographical order. Any pattern P of length m occurring in T is a prefix of one or more suffixes in SA. These suffixes, due to the lexicographical order within SA, are grouped together in the range $\text{SA}[sp, ep]$. To determine $\text{SA}[sp, ep]$ we perform two binary searches over SA and T. Each binary search comparison requires up to m symbol comparisons in T, for a total of $\mathcal{O}(m \log n)$ time. Using additional auxiliary data structures this cost can be reduced to $\mathcal{O}(m + \log n)$ [18]. Suffix array construction is a well studied problem, and many efficient solutions with

[1] We assume logarithms are in base 2.

various tradeoffs are readily available [19], [20]. However, searching for a pattern P in T using only a suffix array requires $\mathcal{O}(n \log n)$ bits to store both T and SA, which in practice is at least 5 times the text size. With the BWT, the key operations of a basic SA can be emulated within much less space, close to the size of T in compressed form.

IV. The Burrows-Wheeler Transform

The Burrows-Wheeler Transform [21] (BWT) – also known as the "block-sorting transform" – produces a permutation of a string T, denoted T^{BWT}, by sorting the n cyclic rotations of T into full lexicographical order, and taking the last column of the resulting $n \times n$ matrix $\mathcal{M}$ to be T^{BWT}. The resulting string T^{BWT} tends to be more compressible as symbols are grouped together based on their context in T, which makes the BWT an important part in many state of the art compression systems [17]. To produce T^{BWT} for a given text T, it is not necessary to construct $\mathcal{M}$ as there is a duality between T^{BWT} and the SA over a text T: $T^{\text{BWT}}[i] = T[\text{SA}[i] - 1 \bmod n]$.

Remarkably, the original text T can be recovered from T^{BWT} in linear time without the need for any additional information. To recover T from only T^{BWT} we first recover the first column, F, in $\mathcal{M}$ by sorting the last column ($L = T^{\text{BWT}}$), in lexicographical order. By mapping the symbols in L to their respective positions in F so $L[i] = F[j]$ (usually referred to as the LF mapping, $j = \text{LF}(i)$) we can recover T backwards as $T[n-1] = T^{\text{BWT}}[0] = \$$ and $T[j-1] = T^{\text{BWT}}[\text{LF}(i)]$ if and only if $T[j] = T^{\text{BWT}}[i]$. The LF mapping is computed using the equation

$$\text{LF}(i) = \text{LF}(i, c) = C[c] + \text{Rank}_c(T^{\text{BWT}}, i) \qquad (1)$$

where c is the symbol $T^{\text{BWT}}[i]$, and $C[c]$ stores the number of symbols in T^{BWT} smaller than c. Figure 1 shows an example of the BWT permutation matrix $\mathcal{M}$ for the string $T =$ `acacacracaca$` including the LF mapping.

Using the LF mapping we can recover T starting with position of symbol \$ in T^{BWT}: $T^{\text{BWT}}[4] = \$$, thus $T[n-1] = \$$. $T[n-2] = T^{\text{BWT}}[\text{LF}(4) = 0] = a$. Consequently we recover $T[n-3] = T^{\text{BWT}}[\text{LF}(0) = 1] = c$ and $T =$ `acacacracaca$` in reverse order.

V. Context Bound Block Transforms

Independently, Schindler [22] and Yokoo [23] described a variation of the BWT which partially sorts the n rotations based on k length prefixes. This transform is widely known as the k-BWT.

Unlike the full BWT, the k-BWT only sorts the permutation matrix $\mathcal{M}$ up to a depth k ($\mathcal{M}_k$). Figure 2 shows the $\mathcal{M}_3$ rotations of the k-BWT for the string $T =$ `acacacracaca$` and $k = 3$, producing $T^{3\text{-BWT}}$. Due to the fixed sorting depth, multiple suffixes can be treated as k-equal during the sorting stage. These suffixes are grouped in *context groups*, where suffixes are stored in ascending order according to their position in T for each context group. The k-group boundaries can be can be marked in a bitvector, D_3, or just D for short. For our example shown in Figure 2, D_k is 1110001110011. We formally define D_k as follows:

i	LF	SA	F	$L(T^{\text{BWT}})$
0	1	12	\$ a c a c a c r a c a c	a
1	7	11	a \$ a c a c a c r a c a	c
2	8	9	a c a \$ a c a c a c r a	c
3	12	7	a c a c a \$ a c a c a c	r
4	0	0	a c a c a c r a c a c a	\$
5	9	2	a c a c r a c a c a \$ a	c
6	10	4	a c r a c a c a \$ a c a	c
7	2	10	c a \$ a c a c a c r a c	a
8	3	8	c a c a \$ a c a c a c r	a
9	4	1	c a c a c r a c a c a \$	a
10	5	3	c a c r a c a c a \$ a c	a
11	6	5	c r a c a c a \$ a c a c	a
12	11	6	r a c a c a \$ a c a c a	c

Figure 1. Example BWT permutation matrix $\mathcal{M}$ of the string $T =$ `acacacracaca$` with the output being the last column in $\mathcal{M}$: $T^{\text{BWT}} =$ `accr$ccaaaaac`.

i	D_3	LF_3	**F 1 2**	$L(T^{3\text{-BWT}})$
0	1	1	\$ a c a c a c r a c a c	a
1	1	7	a \$ a c a c a c r a c a	c
2	1	0	a c a c a c r a c a c a	\$
3	0	8	a c a c r a c a c a \$ a	c
4	0	12	a c a c a \$ a c a c a c	r
5	0	9	a c a \$ a c a c a c r a	c
6	1	10	a c r a c a c a \$ a c a	c
7	1	2	c a \$ a c a c a c r a c	a
8	1	3	c a c a c r a c a c a \$	a
9	0	4	c a c r a c a c a \$ a c	a
10	0	5	c a c a \$ a c a c a c r	a
11	1	6	c r a c a c a \$ a c a c	a
12	1	11	r a c a c a \$ a c a c a	c

Figure 2. The k-BWT permutation matrix for the string $T =$ `acacacracaca$` with $k = 3$.

Definition 1. *For any $0 \leq k < n$, let $D_k[0, n-1]$ be a bitvector, such that $D_k[0] = 1$ and, for $1 \leq i < n$,*

$$D_k[i] = \begin{cases} 0 & \text{if } \mathcal{M}_k[i][0, k-1] = \mathcal{M}_k[i-1][0, k-1] \\ 1 & \text{if } \mathcal{M}_k[i][0, k-1] \neq \mathcal{M}_k[i-1][0, k-1] \end{cases}$$

Recovering T from $T^{k\text{-BWT}}$ using LF_k requires additional work since the mapping only allows us to determine the context preceding the current symbol as a result of the

incomplete sorting of $\mathcal{M}_k$. For example, consider the following context jump in Figure 2, where our initial starting position is $T^{3\text{-BWT}}[10] = a$ and $\text{LF}_k(10) = 5$. The symbol preceding "a" should be $T^{3\text{-BWT}}[5] = c$, but due to the incomplete sorting of $\mathcal{M}_k$, the correct row – in the same k-group – is actually 4, which results in $T^{3\text{-BWT}}[4] = r$. When recovering T from $T^{3\text{-BWT}}$, LF_k only guarantees to jump to the correct preceding k-group. Individual context groups need to be processed in reverse sequential order. After performing LF_k, instead of using the row similar to the full BWT, we jump to the last *unprocessed* row within a given k-group. To consistently determine the correct context bounds, a bitvector D_k is required in order to reconstruct T from $T^{k\text{-BWT}}$. D_k can be reconstructed in $\mathcal{O}(n)$ time from $T^{k\text{-BWT}}$ [24] or it can be stored with $T^{k\text{-BWT}}$ at a cost of n bits. These and other trade-offs are explored in [25].

Interestingly, the k-BWT requires less processing in the rotation sorting phase of the transform than the BWT. Similarly to the BWT, we can build the k-BWT efficiently by constructing a suffix array (SA) over T, but must only compare each suffix to a depth of k. For texts containing long repetitions, a k-BWT permutation can be constructed much more efficiently. Within each k-group, the suffixes are ordered in ascending text order. This unique property of the k-BWT allows external memory construction. A simple external memory construction algorithm can determine the starting positions of each k-context in T by making one pass over T in $\mathcal{O}(n)$ time and $\mathcal{O}(\sigma^k \log n)$ space. In a second pass over the suffix array positions, the respective symbol in $T^{k\text{-BWT}}$ is written based on the context.

VI. Backwards Search

Performing a search in BWT permuted text is straightforward. Recall that all rows are sorted in lexicographical order in $\mathcal{M}$. Therefore, for a pattern P, all occurrences of P in T must have a corresponding row in $\mathcal{M}$ within a range $\langle sp, ep\rangle$, where $T_{sp}[0, m-1] = T_{ep}[0, m-1] = P$. To determine the range within $\mathcal{M}$, we first determine the range $\langle sp_m, ep_m\rangle$ within $\mathcal{M}$ that corresponds to p_m using $C[\ ]$. Then, for each symbol $j = m-1 \ldots 0$ in P, we iteratively find $\langle sp_j, ep_j\rangle$ by calculating the number of rows within $\langle sp_{j+1}, ep_{j+1}\rangle$ that are preceded by the symbol p_j in T. For a given row j, the LF mapping can be used to determine the row in $\mathcal{M}$ representing the symbol preceding j in T. The preceding row is determined by counting the number of occurrences of $c = T^{\text{BWT}}[j]$ *before* the current row and ranking these occurrences within $C[c]$. Assume we have located $\langle sp_{j+1}, ep_{j+1}\rangle$, which corresponds to the rows prefixed by $P[j+1, m]$. Then

$$sp_j \ = \ \text{LF}(sp_{j+1} - 1, p_j) + 1 \qquad (2)$$

will calculate the position in F of the first occurrence of p_j within $\langle sp_{j+1}, ep_{j+1}\rangle$, and thus compute the start of our range of rows within $\mathcal{M}$ that correspond to $P[j, m]$. Similarly, we compute

$$ep_j \ = \ \text{LF}(ep_{j+1}, p_j). \qquad (3)$$

For example, in Figure 1, assume we have previously determined the range of the rows corresponding to `ca` as $\langle 7, 10\rangle$. We can now determine the rows in $\mathcal{M}$ that match `aca` by performing $sp = \text{LF}(6, a) + 1 = 2$, as the number of `a`'s in T^{BWT} before row 7 is 1 and the rows prefixed by `a` in $\mathcal{M}$ lie within $\langle 1, 6\rangle$. Similarly, we determine the $ep = \text{LF}(10, a) = 5$. Thus the rows prefixed by `aca` are bounded within $\langle 2, 5\rangle$.

Once the area $\langle sp, ep\rangle$ is determined, self-indexes offer a way to find any occurrence position $\text{SA}[j]$, for $sp \leq j \leq ep$. This is accomplished by sampling T at regular intervals, and marking positions of SA that point to sampled text positions in a bitmap $E[0, n-1]$. Sampled suffix array positions are stored in an array $G[\text{RANK}_1(E, j)] = \text{SA}[j]$ if $E[j] = 1$. Given a desired value $\text{SA}[j]$, successive values $i = 0, 1, \ldots$ can be evaluated until $E[\text{LF}^i(j)] = 1$, producing the desired answer of $\text{SA}[j] = \text{SA}[\text{LF}^i(j)] + i$. If every s th text position is sampled, we guarantee i can be found for every $0 \leq i < s$, and sampling requires $\mathcal{O}((n/s) \log n)$ extra bits for G (and for E in compressed form [15]), and computes any entry of SA within s applications of LF.

Similarly, in order to recover any text substring $T[l, r-1]$ (including the whole T), we can use the same sampling of text position multiples of s, and store $H[i] = \text{SA}^{-1}[i \cdot s]$. Thus, we extend the range to $T[l, r'-1]$, for $r' = s \cdot \lceil r/s \rceil$ and display from the suffix array position $j = \text{SA}^{-1}[r']$. Then, we can display the area backwards as $T^{\text{BWT}}[j], T^{\text{BWT}}[\text{LF}(j)], T^{\text{BWT}}[\text{LF}^2(j)], \ldots$. Each such step requires one RANK_c and one ACCESS operation, which has the same cost as LF. Therefore, we display $T[l, r-1]$ within $\mathcal{O}(r - l + s)$ applications of LF.

VII. Backwards search in k-BWT permuted text

As explained, performing backwards search on a k-BWT permuted text is not so straightforward. The following lemma shows that one can, however, run the counting as usual on patterns of length $m \leq k$.

Lemma 1. *The* $\text{LF}(i, c)$ *mapping formula of Eq. (1) works correctly on the* k-BWT *if* i *is the last row of a context group.*

Proof. The formula counts the number of occurrences of c in $T^{\text{BWT}}[0, i]$. Since i is the last row of a k-context, the set of rows in $\mathcal{M}[0, i]$ is the same set of rows in $\mathcal{M}_k[0, i]$, and therefore the number of occurrences of any c in $T^{\text{BWT}}[0, i]$ is the same as in $T^{k\text{-BWT}}[0, i]$. ■

Therefore, we can seamlessly search for patterns up to length k.

Lemma 2. *If the length of a search pattern is $m \leq k$, then the algorithm used to compute the range* $\text{SA}[sp, ep]$ *for* BWT *is sufficient for* k-BWT.

Proof. As long as $m \leq k$, all ranges $\langle sp_i, ep_i \rangle$ will be composed of whole k-contexts. Therefore, the formulas in Eqs. (2) and (3) compute LF on rows that are at the end of k-contexts. By Lemma 1, all the $\langle sp_i, ep_i \rangle$ are thus computed correctly. ■

Note in particular that when $m = k$, we are able to collect the occurrences in text position order, since $\text{SA}[sp, ep]$ will correspond precisely to a k-context of $T^{k\text{-BWT}}$. However, we cannot compute correct ranges $\text{SA}[sp, ep]$ for patterns longer than k because the contexts are only k-sorted, and the occurrences of longer patterns are not contiguous within k-BWT.

For example, consider $\mathcal{M}_3$ in Figure 3. The rows prefixed by the pattern `caca` are 8 and 10. However, the row 9 within the range $\langle 8, 10 \rangle$ is not prefixed by `caca` but `cacr`.

A way to handle longer patterns $P[0, m-1]$ is to search for $P[m-k, m-1]$ as usual, and then track each candidate to determine whether it is an occurrence of the full P. But, we must be able to compute $\text{LF}(j)$ for any j. Computing arbitrary $\text{LF}(j)$ values is also necessary for locating occurrences and for displaying arbitrary substrings of T, by using the sampling mechanisms described at the end of Section VI. The remainder of the section is devoted to showing how to compute LF.

Theorem 1. *The function* LF *on matrix $\mathcal{M}_k$ can be computed in the time required to compute* RANK *and* ACCESS*, using $nH_k + 2nH_{k-1} + o(n \log \sigma)$ bits of space, for any $k \leq \alpha \log_\sigma(n) - 1$ and constant $\alpha < 1$.*

Proof. In order to compute $i = \text{LF}(j)$, we use bitmap D_k to find $p = \text{SELECT}_1(\text{RANK}_1(D_k, j))$, which corresponds to the beginning of the group row of $\mathcal{M}_k[j]$. Let $\mathcal{M}_k[j] = xayb$, where $|x| = k-1$ and $|a| = |b| = 1$, such that row j belongs to group C_{xa}. Then, $\mathcal{M}_k[i] = bxay$ belongs to the group C_{bx}. Moreover, since occurrences of xa are sorted in text position order inside C_{xa}, row j is the $(j - p + 1)$-th occurrence of xa in T in text position order.

In order to find the starting position p' of group C_{bx}, we use bitmap D_{k-1}. First, we find the starting position p^* of group C_x in $T^{k\text{-}1\text{-BWT}}$ using $p^* = \text{SELECT}_1(D_{k-1}, \text{RANK}_1(D_{k-1}, j))$. Now, $p' = C[b] + \text{RANK}_\text{b}(T^{k\text{-BWT}}, p^* - 1) + 1$ gives the desired starting position of group C_{bx} in $T^{k\text{-BWT}}$. We can confirm this is true since $\text{RANK}_\text{b}(T^{k\text{-BWT}}, p^* - 1)$ counts the number of text substrings of the form bz, with $z < x$ in lexicographic order.

Now, within group C_{bx}, the rows are sorted in text position order, thus row i corresponds to the $(i - p' + 1)$-th occurrence of bx in T in text order. Furthermore, we know that the row $\mathcal{M}_k[i]$ points to an occurrence of bxa in T, whereas $\mathcal{M}_k[j]$ points to the next position, xa preceded by b.

So, a way to connect j and i is as follows. Store the wavelet tree of $T^{k\text{-}1\text{-BWT}}$, where all the characters preceding x are in text position order for the area corresponding to group C_x. Similarly, store the wavelet tree of S, which is similar to $T^{k\text{-}1\text{-BWT}}$ but the characters following (not preceding) x are recorded for each context C_x (note the areas coincide for $T^{k\text{-}1\text{-BWT}}$ and S). Therefore, $r = \text{SELECT}_\text{a}(S, \text{RANK}_\text{a}(S, p^* - 1) + j - p + 1)$ finds the rank of row $\mathcal{M}_k[j]$ (i.e., its occurrence of xa), in text position order, among the occurrences of x in T, and $q = \text{RANK}_\text{b}(T^{k\text{-}1\text{-BWT}}, r) - \text{RANK}_\text{b}(T^{k\text{-}1\text{-BWT}}, p^* - 1)$ is

$\mathcal{M}_2$

i	D_2	**F**	**1**	**S**									$T^{2\text{-BWT}}$
0	1	$	a	c	a	c	a	c	r	a	c	a	c
1	1	a	$	a	c	a	c	a	c	r	a	c	a
2	1	a	c	a	c	a	c	r	a	c	a	c	a
3	0	**a**	**c**	a	c	r	a	c	a	c	a	$	a
4	0	**a**	**c**	*r*	a	c	a	c	a	$	a	c	a
5	0	a	c	a	c	a	$	a	c	a	c	a	c
6	0	**a**	**c**	a	$	a	c	a	c	a	c	r	a
7	1	c	a	c	a	c	r	a	c	a	c	a	$
8	0	c	a	c	r	a	c	a	c	a	$	a	c
9	0	c	a	c	a	$	a	c	a	c	a	c	r
10	0	c	a	$	a	c	a	c	a	c	r	a	c
11	1	c	r	a	c	a	c	a	$	a	c	a	c
12	1	r	a	c	a	c	a	$	a	c	a	c	a

$\mathcal{M}_3$

i	D_3	LF_3	**F**	**1**	**2**								$T^{3\text{-BWT}}$
0	1	1	$	a	c	a	c	a	c	r	a	c	a
1	1	7	a	$	a	c	a	c	a	c	r	a	c
2	1	0	a	c	a	c	a	c	r	a	c	a	$
3	0	8	a	c	a	c	r	a	c	a	c	$	a
4	0	12	a	c	a	c	a	$	a	c	a	c	r
5	0	9	a	c	a	$	a	c	a	c	r	a	c
6	1	**10**	a	c	r	a	c	a	c	a	$	a	c
7	1	2	c	a	$	a	c	a	c	a	c	r	a
8	1	3	**c**	**a**	**c**	a	c	r	a	c	a	c	$
9	0	4	**c**	**a**	**c**	r	a	c	a	c	a	$	a
10	0	5	**c**	**a**	**c**	a	$	a	c	a	c	a	r
11	1	6	c	r	a	c	a	c	a	$	a	c	a
12	1	11	r	a	c	a	c	a	$	a	c	a	c

Figure 3. The k-BWT permutation matrix $\mathcal{M}_k$ used to search for pattern $P =$ `cacr` in text $T =$ `acacacracaca$` where $k = 2$ (top) and $k = 3$ (bottom).

i	0	1	2	3	4	5	6	7	8	9	10	11	12
LF_3	1	7	0	8	12	9	10	2	3	4	5	6	11
LF	1	7	0	8	12	10	9	5	2	3	4	6	11
δ						+1	-1	+3	-1	-1	-1		

Figure 4. The difference between LF and $\text{LF}_3()$

the number of occurrences of bx up to that position. Hence, the answer is $i = \text{LF}(j) = p' + q - 1$.

Note this method requires finding out a. This can be stored in an array A of at most σ^k entries containing $\mathcal{M}_k[j][k]$ for all rows j belonging to each context. Therefore, $a = A[\text{RANK}_1(D_k, j)]$.

The total time required is a constant number of RANK and ACCESS operations on the wavelet trees. As for space, we have the wavelet trees of $T^{k\text{-BWT}}$ and those of $T^{k\text{-}1\text{-BWT}}$ and S. If we use Raman et al.'s method [15] to represent the bitmaps of the wavelet trees, $T^{k\text{-BWT}}$ requires $nH_k(T) + \mathcal{O}(\sigma^{k+1} \log n)$ bits for any $k \geq 0$. The existing proof of this space bound for T^{BWT} [16] makes use of the fact that the suffixes are k-sorted, and therefore it also applies to $T^{k\text{-BWT}}$. Similarly, $T^{k\text{-}1\text{-BWT}}$ requires $nH_{k-1} + \mathcal{O}(\sigma^k \log n)$ bits. Finally, S requires $nH_{k-1} + \mathcal{O}(\sigma^k \log n)$ bits because the sets of characters within each $(k-1)$-context is the same as for $(T^{rev})^{k\text{-}1\text{-BWT}}$, where T^{rev} is T read backwards, and the $(k-1)$-th order empirical entropy of S and $(T^{rev})^{k\text{-}1\text{-BWT}}$ are equal. Furthermore, the $(k-1)$-th order entropy between T and T^{rev} differs only by $\mathcal{O}(\log n)$ bits for any k [26]. The bitmaps D_k and D_{k-1} have only $\mathcal{O}(\sigma^k)$ bits set out of n, and can be represented within $\mathcal{O}(\sigma^k \log n) + o(n)$ bits while supporting constant-time $\text{RANK}_{0/1}$ and $\text{SELECT}_{0/1}$ [15]. Array A requires an additional $\mathcal{O}(\sigma^k \log k)$ bits.

To obtain the final space bound of the theorem we note that $\mathcal{O}(\sigma^{k+1} \log n) \subset o(n \log \sigma)$ if $k \leq \alpha \log_\sigma(n) - 1$ for any constant $\alpha < 1$. ■

VIII. Practical Evaluation and Alternatives

In our initial approach, we require three wavelet trees over different permutations of T to fully support LF in a k-BWT permuted text. Figure 4 shows the difference between LF_3 and the LF mapping to jump to the correct preceding row.

Instead of storing additional wavelet trees, we could also explicitly store the correction information δ for each row in $\mathcal{M}_k$ that does not jump to the correct preceding row using only LF_k. For each context C of size d, we can store the correction information required in at most $d \log d$ bits. The context length decreases as we increase the sorting depth k. Therefore, we need to store less correction information as k increases. The correction information can be calculated as follows in linear time:

1) Perform the reverse k-BWT to recover T from $T^{k\text{-BWT}}$.
2) During the reverse transform, each context group C is processed in reverse sequential order.
3) For each context group, keep track of the current position p.
4) The value δ_j is the correction information required to jump from row j to row v and can be calculated as $\delta_j = p_v - \text{LF}_k(j)$.

Note that we only store correction information for the n' non-trivial k-groups (that is, those of size more than 1). Those k-groups are stored according to their order within D_k at a cost of $n' \log n$ bits, in addition to the cost of storing the correction information for each k-group in $d \log d$ bits. To access the correct k-group for a given row j, we compute the number of non-trivial k-groups, o, preceding j, where $o = \text{RANK}_{10}(D_k, j)$ (this is an extension of binary rank for fixed substrings, which is easily handled in constant time within $o(n)$ extra space).

We can further apply the same concept to our wavelet tree approach: Remove the information in $\mathcal{M}_{k-1}$ associated with trivial k-groups. The k-groups in $\mathcal{M}_{k-1}$ of size 1 will never be used to calculate LF as Eq. (1) is already correct on the last row of each group, hence it is correct for all groups of size 1. To map a row j in $\mathcal{M}_k$ to its corresponding row j' in $\mathcal{M}_{k-1}$, we subtract the number of trivial groups before j', as these are not stored explicitly. A trivial k-group corresponds to two consecutive 1s in D_{k-1}. Therefore, the correct row $j'' = j' - \text{RANK}_{11}(D_{k-1}, j')$. By combining all of these techniques, the additional information required to perform LF *decreases* as we increase the sorting depth k.

We now compare our backward search approach storing three wavelet trees, to storing the correction information δ_j explicitly. In our experiment, we compare multiple 50 MB data sets from the Pizza & Chili Corpus and the TREC collection [27]. The number of k-groups and their mean average size for each data set is shown in Figure 5.

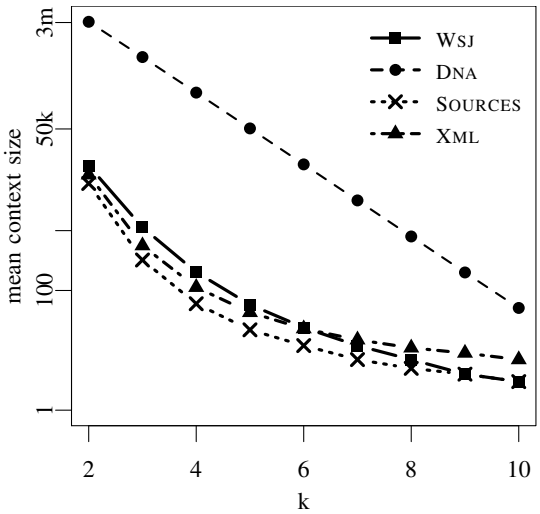

Figure 5. Mean average k-group size for each data set as k increases. Note the logarithmic scale of the y-axis.

The mean average context size decreases logarithmically

for DNA. For the data sets WSJ, SOURCES and XML, and a sorting depth of 5, the mean average k-group size is less than 100. For large k, this suggests that storing the correction information is a viable alternative to storing three wavelet trees (although these would also decrease their sizes due to the increase of trivial groups). To validate this assumption, the total space requirements for the wavelet tree approach and the explicit storage of correction information for each data set was measured. We use Huffman shaped wavelet trees in conjunction with succinct $\text{RANK}_{0/1}$ operations [15] to store and access the wavelet trees over $T^{k\text{-}1\text{-BWT}}$ and S. Figure 6 shows the size of the wavelet trees *as a percentage* of the space required to store the correction information explicitly using $d \log d$ bits for a k-group of size d.

As expected, the ratio increases with k. While both approaches benefit from the trivial groups that appear, the correction information benefits from smaller groups due to its $\log d$ space factor, whereas the wavelet tree space usage is independent of the group sizes. Note also that storing the correction information explicitly is more efficient for English text (WSJ) and source code (SOURCES) for $k > 6$ or 7. However, DNA and XML can always be stored more efficiently using the wavelet tree approach. The reason for this discrepancy is two-fold. Firstly, DNA has a large number of contexts, even for $k = 10$, relative to the other test collections. Secondly, the XML file shows similar k-group sizes to WSJ and SOURCES, but it can be stored more efficiently using the wavelet tree approach as it is more compressible.

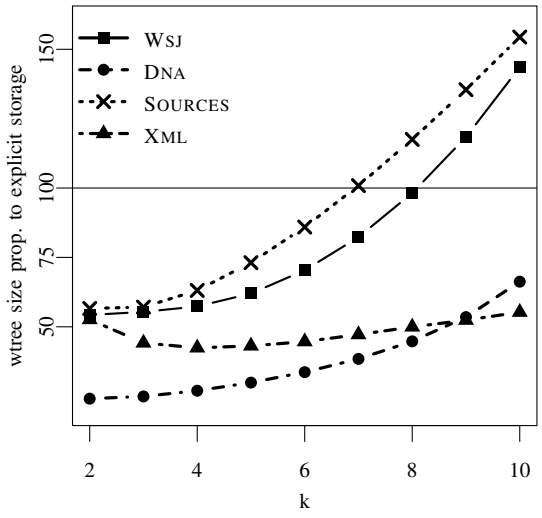

Figure 6. Storage requirements of the wavelet tree approach as a percentage of storing the correction information explicitly for variable k.

In addition to comparing our two LF mapping approaches, we now compare a simple k-gram inverted index to our k-BWT wavelet tree approach. We implemented a simple k-gram inverted index using d-gap and gamma encoded inverted lists for all match positions in the text. Figure 7 reports total size of each index for increasing values of k. The size of the k-gram index is reported as the size of all compressed posting lists, plus the space required to store all necessary k-grams (v) in a dictionary, namely $v \log n + vk \log \sigma$. To provide a fair estimate of the lower cost bound to optimally store the gamma encoded, d-gapped posting lists, the measured size of the q-gram index was halved. We compare this conservative estimate against the actual size to store $T^{k\text{-}1\text{-BWT}}$, S, D_k and D_{k-1} in succinct form as described above. Note we are not considering the cost of storing the compressed text in the k-gram index, or the cost of storing T^{BWT}, which should be similar.

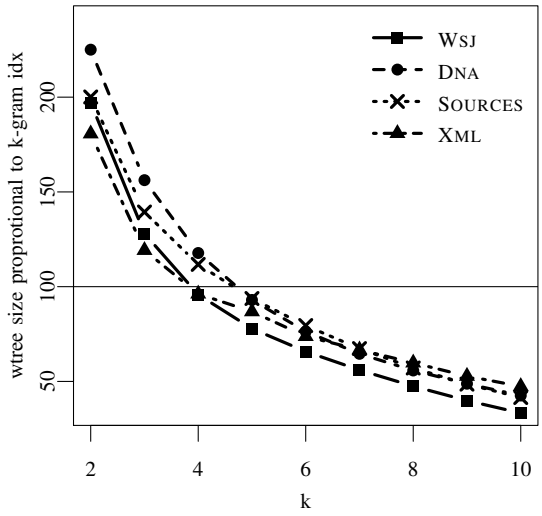

Figure 7. Storage requirements of the wavelet tree approach as a percentage of the k-gram index space usage.

For small k, the k-gram index is more space efficient than our k-BWT based wavelet tree approach. As we increase the sorting length, the k-gram index space usage grows rapidly, whereas our approach consistently uses *less* space. For $k = 4$ or 5, we already require less space than the k-gram index. For $k = 10$, the wavelet trees require 30% to 50% of the k-gram index size (recall this is a lower bound as we are halving the gamma-codes space). The size of the k-gram index increases as the posting listing size decreases for larger k. Smaller posting lists can be compressed less effectively using d-gap and gamma encoding. The dictionary size also increases dramatically as more unique k-grams are stored. The wavelet trees become more compressible as we increase the sorting length and the amount of extra information required to perform LF decreases as the number of trivial k-groups increases. To conclude the practical evaluation, we compare the absolute space usage of our approach to the k-gram index, a normal wavelet tree over T^{BWT} as used by a FM-Index, and a wavelet tree over $T^{k\text{-BWT}}$. Figure 8 shows absolute space usage in MB for the WSJ data set. The wavelet tree over T^{BWT} uses the least amount of space. For $k = 5$, the space usage of $T^{k\text{-BWT}}$ comes close to that of T^{BWT}. The k-

gram index is most efficient when k is small. But, the k-gram index space usage can grow exponentially with respect to sorting depth. The $T^{k\text{-BWT}}$ plus auxiliary information required to perform LF requires around 3 times the space of $T^{k\text{-BWT}}$. The amount of auxiliary information required decreases as the sorting depth increases.

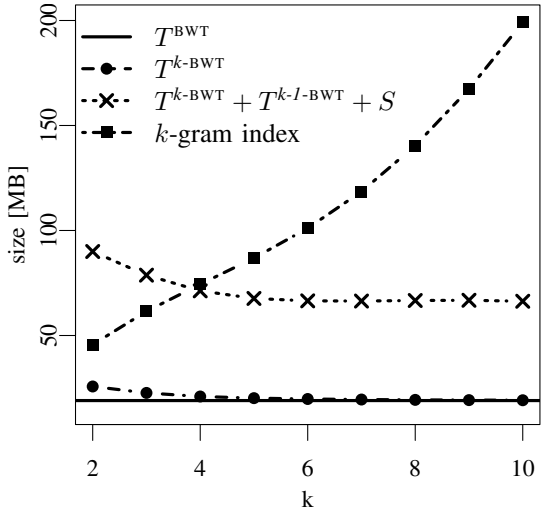

Figure 8. Absolute storage requirements in MB of the wavelet tree approach, the k-gram index, the wavelet tree over $T^{k\text{-BWT}}$, and over T^{BWT} for the WSJ data set.

IX. Application: A k-gram Self-Index

As explained in Section I, a context bound transformation can be regarded as a self-index representation of a k-gram index [6], [7], [8], [9]. A k-gram index is a popular alternative for constructing inverted indexes on languages that are not amenable to term tokenization and stemming, and a core component in the highly successful BLAST application for searching in genomic data [12].

In essence, our method can represent the sequence T in compressed form, and replaces the need to explicitly store the position offsets for each k-gram. We have shown in Section VII how to carry out searches for patterns of length k, and deliver all occurrence positions in text order. Thus, we obtain a listing that is explicitly stored in a classical k-gram index on the fly. We are also able to display any text substring from the self-indexed representation. Furthermore, our experiments in Section VIII show an interesting phenomenon: the space requirements for our index decrease as k grows, whereas in a classical k-gram index, the space grows exponentially faster with k. This makes our representation attractive, for example, in applications where using larger k values is desirable, but not possible a with classical k-gram index.

In Section VII we derived simple techniques to handle searches for patterns of length greater than k, for applications where such a search is necessary. In general, our index can mimic any of the well-known algorithms on k-gram indexes. For example, we can split the pattern P into k size chunks P_1,P_2, $\ldots P_r$. For each chunk P_i, we determine $\langle sp_i, ep_i \rangle$ and the locations of each match to P_i. We then intersect these results to to locate the larger pattern P. Since the ranges $\langle sp_i, ep_i \rangle$ for each chunk in produced ascending order, we can perform an r-way merge using a min-heap over the smallest element in each occurrence range to get the final occurrence listing.

Another example occurs in BLAST-like applications, where approximate searches for P are reduced to a set of searches for k-grams of P. Then, in the most general formulation [10], one looks for text areas where at least h distinct k-grams of P appear in nearby text positions. Retrieving the k-gram occurrences in text order is essential for the effectiveness of these methods.

Using our approach, we are able to leverage standard inverted indexing techniques to process queries, without explicitly storing the inverted lists for each k-gram occurrence. In fact, k-gram based inverted files often do not explicitly store the position of each k-gram, but rather the document occurrence. This saves space, but means false match filtering must be performed on each possible document occurrence [13]. Our method is able to return the exact positions without the need to perform false match filtering. As a final comment, an area where suffix arrays perform poorly is on *position-restricted searches* [28], where the goal is to return occurrences of P within an area of T. As a k-gram index delivers positions in text order, we can binary search the range of the relevant occurrences. This is much more complex for suffix arrays, which deliver the occurrences in any order, and need additional index structures to handle these queries efficiently.

X. Future Work

In this paper, we have outlined the initial algorithms required to create self-indexes capable of emulating all k-gram index functionality. Our first aim is to complete the implementation of these self-indexes and measure their time performance for a variety of queries and data collections. In particular, the explicit storage of the differences is likely to be much faster than storing the two extra wavelet trees, and finding other pragmatic solutions for succinctly storing the auxiliary information could further increase the performance. A robust implementation will also expose any additional time cost overhead incurred, for example in comparison to the time that a BLAST-based index must spend in finding candidate text areas and process them with a (slow) sequential approximate pattern matching algorithm. We would like to empirically establish if extracting all the occurrences in suffix array order is more cumbersome and space-demanding than our algorithm, or is significantly slower.

We also believe there may be better solutions to implement the LF mapping than we have presented here. It is indeed

somewhat surprising that an index traditionally regarded as simpler than a suffix array is harder to handle when regarded as a partial Burrows-Wheeler transform. In general, we feel we are opening an area full of new challenges with tangible applications, more than closing it. We hope more advances on this research topic will follow.

ACKNOWLEDGEMENTS

The second author was partially funded by Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile. The third and forth authors were supported by the Australian Research Council.

REFERENCES

[1] P. Ferragina and G. Manzini, "Opportunistic data structures with applications." in *Proceedings of the 41st IEEE Annual Symposium on Foundatations of Computer Science (FOCS 2000)*. IEEE Computer Society Press, November 2000, pp. 390–398.

[2] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro, "Compressed representations of sequences and full-text indexes," *ACM Transactions on Algorithms (TALG)*, vol. 3, no. 2, p. article 20, 2007.

[3] G. Navarro and V. Mäkinen, "Compressed full-text indexes." *ACM Computing Surveys*, vol. 39, no. 1, pp. 2–1 – 2–61, 2007.

[4] U. Manber and G. Myers, "Suffix arrays: A new method for on-line search." *SIAM Journal on Computing*, vol. 22, no. 5, pp. 935–948, 1993.

[5] R. Grossi, A. Gupta, and J. S. Vitter, "Higher-order entropy-compressed text indexes." in *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2003)*, January 2003, pp. 841–850.

[6] J. Ullman, "A binary n-gram technique for automatic correction of substitution, deletion, insertion and reversal errors in words." *The Computer Journal*, vol. 10, pp. 141–147, 1977.

[7] P. Jokinen and E. Ukkonen, "Two algorithms for approximate string matching in static texts." in *Proceedings of the 16th Annual Symposium on Mathematical Foundations of Computer Science (MFCS 1991)*, ser. LNCS, A. Tarlecki, Ed., vol. 520. Springer, September 1991, pp. 240–248.

[8] E. Sutinen and J. Tarhio, "Filtration with q-samples in approximate string matching." in *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching (CPM 1996)*, ser. LNCS, D. Hirschberg and G. Myers, Eds., vol. 1075. Springer, June 1996, pp. 50–63.

[9] O. Lehtinen, E. Sutinen, and J. Tarhio, "Experiments on block indexing." in *Proceedings of the 3rd South American Workshop on String Processing (WSP 1996)*, ser. Carleton University Press Informatics Series, N. Ziviani, R. Baeza-Yates, and K. Guimarães, Eds., no. 4, August 1996, pp. 183–193.

[10] G. Navarro, R. Baeza-Yates, E. Sutinen, and J. Tarhio, "Indexing methods for approximate string matching," *IEEE Data Engineering Bulletin*, vol. 24, no. 4, pp. 19–27, 2001.

[11] R. N. Horspool, "Practical fast searching in strings," *Software Practice and Experience*, vol. 10, no. 6, pp. 501–506, 1980.

[12] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool." *Journal of Molecular Biology*, vol. 215, pp. 403–410, 1990.

[13] S. J. Puglisi, W. F. Smyth, and A. Turpin, "Inverted files versus suffix arrays for locating patterns in primary memory." in *Proceedings of the 13th International Symposium on String Processing and Information Retrieval (SPIRE 2006)*, ser. LNCS, F. Crestani, P. Ferragina, and M. Sanderson, Eds., vol. 4209. Springer, October 2006, pp. 122–133.

[14] J. I. Munro, "Tables," in *Proceedings of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 1996)*, ser. LNCS, V. Chandru and V. Vinay, Eds., vol. 1180. Springer, December 1996, pp. 37–42.

[15] R. Raman, V. Raman, and S. S. Rao, "Succinct indexable dictionaries with applications to encoding k-ary trees and multisets," in *Proceedings of the 13th ACM-SIAM symposium on Discrete algorithms (SODA 2002)*. ACM/SIAM, January 2002, pp. 233–242.

[16] V. Mäkinen and G. Navarro, "Implicit compression boosting with applications to self-indexing," in *Proceedings of the 14th International Conference on String Processing and Information Retrieval (SPIRE 2007)*, ser. LNCS, N. Ziviani and R. A. Baeza-Yates, Eds., vol. 4726. Springer, October 2007, pp. 229–241.

[17] G. Manzini, "An analysis of the Burrows-Wheeler transform." *Journal of the ACM*, vol. 48, no. 3, pp. 407–430, May 2001.

[18] U. Manber and E. W. Myers, "Suffix arrays: A new method for on-line string searches," *SIAM J. Comput.*, vol. 22, no. 5, pp. 935–948, 1993.

[19] S. J. Puglisi, W. F. Smyth, and A. H. Turpin, "A taxonomy of suffix array construction algorithms." *ACM Computing Surveys*, vol. 39, no. 2, pp. 4–1 – 4–31, 2007.

[20] M. A. Maniscalco and S. J. Puglisi, "An efficient, versatile approach to suffix sorting," *J. Exp. Algorithmics*, vol. 12, pp. 1.2:1–1.2:23, June 2008.

[21] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm." Digital Equipment Corporation, Palo Alto, California, Tech. Rep. 124, May 1994.

[22] M. Schindler, "A fast block-sorting algorithm for lossless data compression." in *Proceedings of the 7th IEEE Data Compression Conference (DCC 1997)*, J. A. Storer and M. Cohn, Eds. Los Alamitos, California: IEEE Computer Society Press, March 1997, p. 469.

[23] H. Yokoo, "Notes on block-sorting data compression." *Electronics and Communications in Japan, Part 3*, vol. 82, no. 6, pp. 18–25, 1999.

[24] G. Nong, S. Zhang, and W. H. Chan, "Computing inverse ST in linear complexity." in *Proceedings of the 19th Annual Symposium on Combinatorial Pattern Matching (CPM 2008)*, ser. LNCS, P. Ferragina and G. M. Landau, Eds., vol. 5029. Springer, July 2008, pp. 178–190.

[25] J. S. Culpepper, M. Petri, and S. J. Puglisi, "Revisiting bounded context block-sorting transformations." RMIT University, Melbourne, Victoria, Australia, Tech. Rep. 1, Apr 2011.

[26] P. Ferragina and G. Manzini, "Indexing compressed text." *Journal of the ACM*, vol. 52, no. 4, pp. 552–581, 2005, a preliminary version appeared in FOCS 2000.

[27] P. Ferragina and G. Navarro, "Pizza & Chili corpus – Compressed indexes and their testbeds." September 2005. [Online]. Available: http://pizzachili.dcc.uchile.cl

[28] V. Mäkinen and G. Navarro, "Position-restricted substring searching," in *Proc. 7th Latin American Symposium on Theoretical Informatics (LATIN)*, ser. LNCS 3887, 2006, pp. 703–714.