Search and Modification in Compressed Texts

Stefan Böttcher, Alexander Bültmann, Rita Hartel
University of Paderborn, EIM - Electrical Engineering, Computer Science and Mathematics
Fürstenallee 11, 33102 Paderborn, Germany, {stb, alexb, rst}@uni-paderborn.de

Abstract. Text compression techniques like bzip2 lack the possibility to search or to update substrings at given positions of texts that have been compressed without prior decompression of the compressed text. We have developed Indexed Reversible Transformation (IRT), a modified version of the Burrows-Wheeler-Transformation (BWT) that in combination with run length encoding (RLE) and wavelet trees (WT) allows for position-based searching and updating substrings of compressed texts without prior decompression of the compressed text. As a result, IRT may be useful for a huge class of applications that due to space limitations prefer to search or to modify compressed texts instead of uncompressed texts.

1. Introduction

1.1 Motivation

There is a wide variety of applications that benefit in terms of data access time or space consumption from using compressed texts, ranging from data transfer to data storage on disk to text processing in limited main memory. When each text $S$ has to be stored in compressed format $c(S)$ only, a modification of $S$ is often performed by decompressing $c(S)$ to $S$, modifying $S$ to $S'$ and eventually compressing $S'$ to $c(S')$ and storing $c(S')$. Whenever time and space limitations are the bottleneck for such an application, and the application requires operations like searching, modifying, inserting, or updating words at a given text position, say, the $n$th word of a huge text $S$, it can be a significant advantage to perform these operations directly on the compressed text $c(S)$, in comparison to decompressing $c(S)$ to $S$, doing the search, insert, update or delete operation on $S$, and eventually compress a modified version of $S$. Direct modification of compressed texts is one of the contributions that distinguishes our approach from previous contributions to text compression.

1.2 Related work

Previous contributions to the field of texts compression can be classified into entropy encoders, e.g. [2], [3], [4], arithmetic coders, e.g. [5], [6], dictionary coders, e.g. [7], [8], [9], statistical compressors, e.g. [10], [11], grammar-based compressors, e.g. [14], and block-sorting compressors, e.g. Burrows-Wheeler-Transformation (BWT) combined with Move-To-Front and Huffman Encoding [1]. Common compressors like bzip2 are based on these algorithms. They do not allow for direct access to arbitrary parts of compressed data. Therefore, searching, modifying or partial decompression are not possible with previous compressors, but are unique features of IRT.

Other contributions based on BWT, e.g. [16] and [17], propose efficient pattern matching in text libraries or pattern collections and provide a possibility to add or remove new texts or patterns. However, patterns cannot be associated with a position, e.g. containing the $n$th word. Therefore, text collections as described in [16], [17] cannot be used for position-based insertion, deletion or modification of the $n$th word or other substrings in the compressed text without prior decompression. Furthermore, position-based word search in compressed texts, if possible at all, requires usually an external index, when using previous text compressors.

In comparison to these approaches, we present a technique for position-based search, insert, update, and delete, which allows e.g. to modify the $n$th word of a text, without full decompression of the text.

Our approach is based on using a modified alphabet for Burrows-Wheeler-Transformation (BWT) [1], combined with run-length FM-index (RLFMI) [12] and Wavelet-Trees (WT) [13] for further compres-
sion. In contrast to all other approaches, we modify the sorting of BWT input in such a way that our approach supports position-based search and modification in compressed text without prior decompression and requires only little extra space.

1.3 Contributions

This paper proposes an approach to block-sorting compression, called Indexed Reversible Transformation (IRT) \([1]\). IRT combines transformation of a text \(S\) to reach better compressibility of the transformed text \(\text{IRT}(S)\) with a self-index to words, such that in contrast to BWT\([1]\) and the approach taken in \([16]\) and \([17]\), IRT supports the following additional operations on \(\text{IRT}(S)\) without prior retransformation of \(\text{IRT}(S)\) into \(S\):

- searching for positions of words in \(S\), given the word's content and \(\text{IRT}(S)\)
- searching in \(\text{IRT}(S)\) the \(n\)th word of \(S\)
- retransforming the \(n\)th word of \(S\) in \(\text{IRT}(S)\)
- deleting the \(n\)th word of \(S\) from \(\text{IRT}(S)\)
- inserting a word as the \(n\)th word of \(S\) into \(\text{IRT}(S)\)
- merging another transformed text \(\text{IRT}(S')\) into \(\text{IRT}(S)\) simulating an insertion of \(S'\) behind the \(n\)th word of \(S\).

In order to compress a text \(S\) being transformed by IRT into \(\text{IRT}(S)\), we extend WT and RLFMI in such a way, that the proposed operations are supported. As a consequence, our extension of RLFMI and WT can be used for implementing the above mentioned operations on \(\text{IRT}(S)\).

2. The concepts behind our indexed reversible transformation, IRT

2.1 Used terms and notations

Let \(S\) be the concatenation of words, such that each word starts with a word delimiter (e.g. ‘$’) and continues with a possibly empty sequence of characters not containing the word delimiter. Furthermore, let \(|S|\) denote the length of \(S\), \(|S|_w\) denote the number of words in \(S\), \(S[n]\) denote the \(n\)th character of \(S\), \(S[n..n+i]\) denote the sequence of characters ranging from \(S[n]\) to \(S[n+i]\) for \(i \geq 0\). Let \(A\) denote the lexicographical order of the alphabet of \(S\), and let \(\text{BWT}(S, A)\) denote the Burrows-Wheeler-Transformation \([1]\) applied to \(S\) according to the ordering relation \(A\). For the \(n\)th word of \(S\), we call \(N\) the word index of the word in \(S\).

2.2 Construction of \(\text{IRT}(S)\) in comparison to \(\text{BWT}(S, A)\)

We construct \(\text{IRT}(S)\) in such a way, that in contrast to \(\text{BWT}(S, A)\), for all \(N \in \{1, \ldots, |S|_w\}\), \(\text{IRT}(S)\) stores the first characters of the \(n\)th word of \(S\) at the position \(N\) of \(\text{IRT}(S)\). This provides us an index to the first character of each word of \(S\), which is an advantage if we are searching e.g. for the word index of words which are alphabetical smaller than a given text constant.

The construction of \(\text{IRT}(S)\) is conceptually done by computing \(\text{BWT}(S_{\text{r}}, A_{\text{r}})\) where \(S_{\text{r}}\) denotes the reversed input text \(S\) and \(A_{\text{r}}\) denotes \(A\) after changing the lexicographical order of the word delimiters ‘$’ of \(S\) in the following way. The word delimiters ‘$’ of \(S\) get the smallest lexicographical order in \(A_{\text{r}}\). Furthermore, the lexicographical order of the word delimiters among themselves is determined by their occurrence in \(S\) from left to right. That is, the \(n\)th word delimiter ‘$’ appearing in \(S\) gets a smaller lexicographical order in \(A_{\text{r}}\) than the \(n+1\)th word delimiter ‘$’. To compute \(\text{BWT}(S_{\text{r}}, A_{\text{r}})\), the construction of the suffix array \(\text{SA}(S_{\text{r}})\) of \(S\) is sufficient \([1]\), since \(S_{\text{r}}\) ends with the first word delimiter ‘$’ of \(S\), i.e. \(S_{\text{r}}\) ends with the character with the smallest lexicographical order according to \(A_{\text{r}}\). \(\text{BWT}(S_{\text{r}}, A_{\text{r}})\) sorts all characters at position \(p\) in \(S_{\text{r}}\) according to the lexicographical order of the suffix of \(p\) in \(S_{\text{r}}\) starting at position \((p+1) \mod |S_{\text{r}}|\) in \(S_{\text{r}}\). Therefore, for all \(N \in \{1, \ldots, |S|_w\}\), the first character of the \(n\)th word of \(S\) is stored at the position \(N\) of \(\text{BWT}(S_{\text{r}}, A_{\text{r}})\).

In practice, however, instead of reversing \(S\) and computing \(\text{BWT}(S_{\text{r}}, A_{\text{r}})\), we compute \(\text{BWT}_{\text{r}}(S, A_{\text{r}})\), where \(\text{BWT}_{\text{r}}\) is based on the construction of the prefix array \(\text{PA}(S)\) of \(S\) instead of the suffix array \(\text{SA}(S)\), where \(\text{PA}(S)[i] = \text{SA}(S)[|S|-i+1]\) for \(1 \leq i \leq |S|\). Then, \(\text{IRT}(S) := \text{BWT}_{\text{r}}(S, A_{\text{r}})\) is computed by
sorting all characters at position $p$ in $S$ according to the lexicographical order of the prefix of $p$ in $S$ starting at position $((|S|+p-1) \mod |S|)$ in $S$.

For example, Figure 1 shows $IRT(S)$ in comparison to $BWT_{\$}(S, A)$ for a given input text $S = "$ab$raca$dabra"$. Note that $IRT(S)$ and $BWT_{\$}(S, A)$ differ e.g. at position 1, because $IRT(S)$ denotes $BWT_{\$}(S, A)$ that sorts the word delimiters '$$' according to their occurrence in $S$.

Searching in $IRT(S)$ requires computing successors or predecessors of a current position in $IRT(S)$. Successors on $IRT(S)$ are defined in terms of circular successorship of represented characters in $S$, and predecessors on $IRT(S)$ are defined in terms of circular predecessorship of represented characters in $S$. For $IRT(S)$, the position $p_n$ is called the successor of the position $p$, in $IRT(S)$, if there are positions $p$, $p_n$ such that $IRT(S)[p_n]$ represents $S[p]$ and $IRT(S)[p]$ represents $S[p_n]$ and $p_n$ is successor of $p$ in $S$, i.e. $p_n = (p + 1) \mod |S|$. The position $p_n$ is called predecessor of the position $p$ in $IRT(S)$, if there are positions $p, p_n$ such that $IRT(S)[p]$ represents $S[p]$ and $IRT(S)[p_n]$ represents $S[p_n]$ and $p_n$ is predecessor of $p$ in $S$, i.e. $p_n = (p + |S| - 1) \mod |S|$.

In the example of Figure 1, $IRT(S)[6]$ represents $S[12]$, position 13 in $S$ is the circular successor of position 12 in $S$, and $IRT(S)[10]$ represents $S[13]$, i.e., position 10 is the successor of position 6 in $IRT(S)$. Similarly, position 11 in $S$ is the circular predecessor of position 12 in $S$, and $IRT(S)[12]$ represents $S[11]$, i.e., position 12 is the predecessor of position 6 in $IRT(S)$.

To calculate the successor of a position $p$ directly on $IRT(S)$, we use two operations on character sequences:

- Rank$(S, p)$, which returns the number of occurrences of a character $c$ in $S[1..p]$ and
- Select$(S, k)$, which returns the position of the $k^{th}$ $c$ in $S$, i.e., Rank$(S, Select(S, k)) = k$.

And we consider Sort$(IRT(S)) = Sort(S)$, which is the characters of $S$ and of $IRT(S)$ respectively sorted according to $A_{\$}$. Then, the successor $p_n$ of a position $p$ in $IRT(S)$ with character $c = IRT(S)[p]$, $c \neq '$$'$, can be calculated by $p_n = Select(Select(IRTS), Rank(IRT(S), p))$. The predecessor $p_n$ of a position $p$ in $IRT(S)$ with character $c = Sort(IRTS)(p)$, $c \neq '$$'$, can be calculated by $p_n = Select(Select(IRTS), Rank(IRT(S), p))$.

### 2.3 Extending the example to search and modification of $IRT(S)$

We continue looking at the example input text $S = "$ab$raca$dabra"$, which is a concatenation of 3 words each of which starts with a unique word delimiter '$$'. $S$ is transformed into $IRT(S)$, such that every word of $S$ can be accessed directly in $IRT(S)$ without prior retransformation of $IRT(S)$ as described below. As a consequence, the search of words in $S$, the deletion of words from $S$, and the insertion of words into $S$ can be simulated on $IRT(S)$ without full retransformation of $IRT(S)$ into $S$.

Within $IRT(S)$, the first characters ‘a’, ‘r’ and ‘d’ of the 3 words “$ab$$”, “$raca$$”, and “$dabra$$” of $S$ can be found easily by looking at $IRT(S)[1]$, $IRT(S)[2]$, and $IRT(S)[3]$ (c.f. Figure 1). Therefore, we can start retransformation of the $n^{th}$ word of $S$ from $IRT(S)$ at the beginning of the $n^{th}$ word, i.e. at $IRT(S)[n]$, and then go to the successor of each position until another word delimiter is reached. This allows partial retransformation of the $n^{th}$ word of $S$ from $IRT(S)$, i.e. avoids full decompression of $IRT(S)$ (c.f. Section 2.4).

Furthermore, an arbitrary word $W$ of $S$ can be deleted from $IRT(S)$ by retransforming $W$ and marking only those positions $p\ldots p$ of $IRT(S)$ that represent positions of the characters of $W$ and by deleting the characters at the positions $p\ldots p$ from $IRT(S)$. For example, the second word of $S$ “$raca$$” can be deleted from $IRT(S)$, such that $IRT(S)$ is modified to $IRT(S')$, where $S' = "$ab$dabra$$”, without considering those characters of $IRT(S)$ that represent the characters of the other words “$ab$$” and “$dabra$$” of $S$. 
The other way round, a new word $W$ can be inserted as the $n^{th}$ word of $S'$ into IRT($S'$) without retransformation of IRT($S'$). If e.g. $W =$ “$sraca$” shall be inserted as the second word into $S'$ and thus as the second word into IRT($S'$), the first character of $W$ not being a word delimiter, ‘$r$’, is inserted at IRT($S'$)[2]. All following characters of $W$, ‘$a$’, ‘$c$’, ‘$a$’, are inserted into IRT($S'$) in such a way, that $W$ can be retransformed and the possibility to retransform the original content of IRT($S'$) is maintained. Thus, IRT($S'$) is extended to IRT($S$), where $S =$ “$SabSracaSdabra$”, without considering the characters in IRT($S$) that represent characters of the others words “$Sab$” and “$Sdabra$” of $S$.

2.4 Retransformation

In contrast to BWT, every word of an input text $S$ can be accessed directly in IRT($S$) and therefore can be retransformed separately without the requirement to retransform other words of IRT($S$). Retransformation of words from IRT($S$) can be done forwards or backwards.

Forward retransformation of the $n^{th}$ word $W_n$ of $S$ starts at position $p_w=n\cdot|S|$, since IRT($S$)[n] contains the first character occurring after the word delimiter of the $n^{th}$ word in $S$ and in case of empty words, the first character is a word delimiter ‘$'$’ terminating the current word. While IRT($S$)[p_w] ≠ ‘$'$’, forward retransformation repeatedly sets $p_w$ to the successor of $p_w$ on IRT($S$). If IRT($S$)[p_w] = ‘$'$’, the word delimiter indicates the end of the $n^{th}$ word. Consequently, when forward retransformation reaches a word delimiter in IRT($S$), and retransformation of the next word is required, the retransformation has to be continued at the beginning of the $n+1^{st}$ word, i.e. at position $n+1$ in IRT($S$).

Backward retransformation starts at any position $p_w=p_{\text{start}}$, for which IRT($S$)[p_{\text{start}}] = ‘$'$’. While IRT($S$)[p_w] ≠ ‘$'$’, where $p_w$ is the predecessor of $p_w$ on IRT($S$), backward retransformation repeatedly sets $p_w$ to the predecessor $p_w$ on IRT($S$). If IRT($S$)[p_w] = ‘$'$’, the successor position $p_w$ of $p_w$ indicates the order of the retransformed word. Note that the word order of $W$ cannot be determined considering the word delimiter ‘$'$’ at $p_{\text{start}}$ in IRT($S$), since the word delimiters in IRT($S$) do not have to be sorted according to their occurrences in $S$.

Forward retransformation and backward retransformation can not only be applied to full words, but can start at an arbitrary position of IRT($S$) to retransform an arbitrary part of IRT($S$).

Using forward and backward retransformation, words of $S$ satisfying specific search-criteria (e.g. containing a given substring $H$) can be searched without full retransformation of IRT($S$). This can be done by retransforming each word of IRT($S$) until a search criterion is fulfilled or can be excluded and returning the number of the word in case of success.

3. Modification of transformed text

3.1 Deletion and insertion of words

A set of words can be deleted from IRT($S$) as follows. First, the words to be deleted are retransformed consecutively and the positions of IRT($S$) visited are marked by using a bit vector of length $|S|$. In a second step, the characters at the marked positions are deleted from IRT($S$), and Sort(IRT($S$)) is updated, such that the frequency of each characters $c$ is equal in IRT($S$) and in Sort(IRT($S$)) after deleting all marked characters from IRT($S$) and from Sort(IRT($S$)).

To insert a word $W$ into IRT($S$), $W$ can be inserted character by character. Each character of $W$ has to be inserted at that position $p$, where the character would be looked up if the word would be retransformed. Furthermore, for each insertion of a character $c$ at position $p$ into IRT($S$), IRT($S$)[p+c] is inserted at position $p$ into Sort(IRT($S$)), where $p+c$ is the predecessor of $p$ in IRT($S$).

To insert $W$ as the $n^{th}$ word into IRT($S$), the first character $c_{\text{first}}$ of $W$ has to be inserted into IRT($S$) at position $i$ in IRT($S$), where $i = n$. Additionally, a word delimiter is inserted at Sort(IRT($S$)) at position $i$, since the predecessor of $c_{\text{first}}$ has to be a word delimiter.

The next character $c_{\text{next}}$ of $W$ is inserted into IRT($S$) at position $p_{\text{ins}}$, where $p_{\text{ins}}$ is the successor of $i$ in IRT($S$). Additionally IRT($S$)[i] has to be inserted at Sort(IRT($S$)) at position $p_{\text{ins}}$. The insertions into IRT($S$) and Sort(IRT($S$)) are repeated for all characters of $W$. Finally, after the last character $c_{\text{last}}$ of $W$ has been inserted, a word delimiter is inserted into IRT($S$) at position $p_{\text{ins}}$, as the successor of $c_{\text{last}}$, to indicate the end of $W$ in IRT($S$). Furthermore, $c_{\text{last}}$ has to be inserted into Sort(IRT($S$)) at position $p_{\text{ins}}$ as predecessor of the word delimiter, to complete the insertion of $W$. 
3.2 Merging two IRTs

Let $S_1$ and $S_2$ be two input texts and let the merge $M(S_1, S_2, n)$ be the result of inserting the words of $S_2$ into $S_1$ behind the $n$th word of $S_1$ at position $p_{\text{INS}}$ where $n = 0$ indicates the insertion of the words of $S_2$ in front of the first word of $S_1$, i.e., $p_{\text{INS}} = 0$. Then, the merge $M(S_1, S_2, n)$ is a text of length $|S_1|+|S_2|$. The merge operation, i.e., the insertion of $S_2$ into $S_1$ behind the $n$th word of $S_1$, can be simulated on transformed texts, i.e., $\text{IRT}(M(S_1, S_2, n))$ can be computed by only considering $n$, $\text{IRT}(S_1)$, and $\text{IRT}(S_2)$, as described in the remainder of this section.

Let $M_{\text{IRT}} = \text{IRT}(M(S_1, S_2, n))$. To compute $M_{\text{IRT}}$, we take advantage of the fact that the characters of $\text{IRT}(S)$ occur in the same order in $\text{IRT}(M(S_1, S_2, n))$. That is, the projection of $\text{IRT}(M(S_1, S_2, n))$ on all characters of $\text{IRT}(S)$ is equal to $\text{IRT}(S_1)$, because if we delete the words of $S_2$ from $\text{IRT}(M(S_1, S_2, n))$, we get exactly $\text{IRT}(S_1)$. Similarly, the characters of $\text{IRT}(S_2)$ occur in the same order in $\text{IRT}(M(S_1, S_2, n))$, i.e., the projection of $\text{IRT}(M(S_1, S_2, n))$ on all characters of $\text{IRT}(S_2)$ is equal to $\text{IRT}(S)$.

To compute $M_{\text{IRT}}$, first, $\text{Sort}(M_{\text{IRT}})$ is constructed by combining $\text{Sort}(\text{IRT}(S_1))$ and $\text{Sort}(\text{IRT}(S_2))$. $M_{\text{IRT}}$ is initialized as an empty array with length $|\text{IRT}(S_1)|+|\text{IRT}(S_2)|$. Then, each word of $\text{IRT}(S_2)$ is re-transformed character by character and simultaneously copied to $M_{\text{IRT}}$ as follows.

If the $n$th word $W$ of $\text{IRT}(S_2)$ is inserted as the $(n+n)$th word into $M_{\text{IRT}}$, the first character $c_1$ of $W$ is copied to position $p_{\text{MIRTS}} = n+n$ in $M_{\text{IRT}}$. As the position $p$ of $c_1$ in $\text{IRT}(S_2)$ is equal to $n$, the first $p$ characters of $\text{IRT}(S_2)$ and the first $p_{\text{MIRTS}}$ characters of $\text{IRT}(S)$ occur up to position $p_{\text{MIRTS}}$ in $M_{\text{IRT}}$, such that $\text{Rank}_{k}(M_{\text{IRT}}, p_{\text{MIRTS}}) = \text{Rank}_{k}(\text{IRT}(S_2), p) + \text{Rank}_{k}(\text{IRT}(S), p_{\text{MIRTS}} - p)$. Thus, the successor $p_{\text{MIRTS}}$ of position $p_{\text{MIRTS}}$ in $M_{\text{IRT}}$ can be calculated using $\text{Sort}(M_{\text{IRT}})$ as described in Section 2.2.

For each successor $p'$ of a given position $p$ in $\text{IRT}(S_2)$ with $c_1 = \text{IRT}(S_2)[p']$ until $c_1=^*S'$, $c_1$ is copied to position $p_{\text{MIRTS}}$ in $M_{\text{IRT}}$. Furthermore, as $\text{Rank}_{c_1}(M_{\text{IRT}}, p_{\text{MIRTS}}) = \text{Rank}_{c_1}(\text{IRT}(S_2), p') + \text{Rank}_{c_1}(\text{IRT}(S_1), p_{\text{MIRTS}} - p')$ where $p'$ is equal to the number of characters of $\text{IRT}(S_2)$ up to $p_{\text{MIRTS}}$ can be calculated in the same way, the successor of $p_{\text{MIRTS}}$ in $M_{\text{IRT}}$ can be calculated too. As all positions $p'$ for all characters $c_1$ of $W$ are visited while retransforming $W$ in $\text{IRT}(S_2)$, all positions $p'$ of the characters $c_1$ of $W$ in $M_{\text{IRT}}$ can be calculated in the same way without retransforming $\text{IRT}(S_1)$.

After all words $W$ of $\text{IRT}(S_2)$ have been copied to $M_{\text{IRT}}$ as described, all characters of $\text{IRT}(S_1)$ can be copied to the remaining empty positions of $M_{\text{IRT}}$ in the same order as they occur in $\text{IRT}(S_1)$.

Computing $\text{IRT}(W)$ and merging it behind the $n$th word into $\text{IRT}(S)$ can be used as a fast alternative for inserting a word $W$ as the $n$th word into $\text{IRT}(S)$ as described in Section 3.1.

4. Implementation

4.1 Compressed representation of $\text{IRT}(S)$ and of $\text{Sort}(\text{IRT}(S))$

Since $\text{IRT}$ only transforms the input text $S$, no compression is provided by $\text{IRT}$. However, $\text{IRT}$ offers the same benefit as BWT w.r.t. improving compressibility as only the order of the word delimiters ‘$’ in $S$ is changed by $\text{IRT}$ in comparison to BWT, and we do not expect the order of the word delimiters ‘$’ to be relevant to the compression result. However, $\text{IRT}$ allows position-based search and modification in $S$ being simulated on $\text{IRT}(S)$. In order to support these operations also on a compressed data structure $\text{CIRT}(S)$ generated from $\text{IRT}(S)$, we are using run-length FM-Index (RLFMI) [12] and a Wavelet-Tree (WT) [13] for compressing $\text{IRT}(S)$ to $\text{CIRT}(S)$.

Given the transformed text $\text{IRT}(S)$, RLFMI searches for sequences of equal characters, denoted as runs. Characters not belonging to a sequence of equal characters form a run of length 1. Then, RLFMI generates $R(\text{IRT}(S))$ by shortening each run to one character only, and computes two bit vectors $B(\text{IRT}(S))$ and $B'(\text{IRT}(S))$ with length $|\text{IRT}(S)|$ as follows.

$B(\text{IRT}(S))$ indicates for each character of $\text{IRT}(S)$ whether or not it occurs at the beginning of a run in $\text{IRT}(S)$ (cf. Figure 2). Thus, $B(\text{IRT}(S))[p]$ is set to 1 if a run starts at $\text{IRT}(S)[p]$, otherwise $B(\text{IRT}(S))[p]$ is set to 0. $B'(\text{IRT}(S))$ is computed by reordering the bits in $B(\text{IRT}(S))$, according to the lexicographical order of the characters that form the runs. Thereby, the relative order of runs of equal characters in $\text{IRT}(S)$ is maintained in $B'(\text{IRT}(S))$ (Figure 2).
Rank, Select, and the possibility to access every character are supported by RLFMI using \( R(\text{IRT}(S)) \), \( B(\text{IRT}(S)) \), and \( B'(\text{IRT}(S)) \) if Rank and Select operations are also supported on \( R(\text{IRT}(S)) \) (c.f. [12]).

In order to achieve these goals and to achieve better compression ratios, \( R(\text{IRT}(S)) \) can be compressed to \( \text{WT}(R(\text{IRT}(S))) \) using a binary wavelet-tree \( \text{WT} \). Each node of \( \text{WT} \) represents a part of the alphabet \( \Sigma_s \) of the input text \( S \). The root of \( \text{WT} \) represents the whole alphabet \( \Sigma_s \), and every leaf represents a distinct character of \( \Sigma_s \), such that each character of \( \Sigma_s \) is associated with a unique leaf of \( \text{WT} \). If a non-leaf node \( n \) of \( \text{WT} \) represents the characters in range \( \Sigma_n = [i,j] \), its left child in \( \text{WT} \) represents a left sub-range of \( \Sigma_n \), and its right child in \( \text{WT} \) represents the remainder of \( \Sigma_n \). Each node \( n \) of \( \text{WT} \) is associated with the subsequence \( R(\text{IRT}(S))_n \) of \( R(\text{IRT}(S)) \) formed by only the characters occurring in \( \Sigma_n \) and consists of a bit vector \( b_n \) indicating whether a character in \( R(\text{IRT}(S))_n \) belongs to the left child or to the right child of \( n \) (c.f. Figure 3). The fragmentation of the alphabet can be done in such a way, that \( \text{WT} \) becomes Huffman-shaped according to the expected occurrence of characters in texts to improve the compression ratio. As a result, \( \text{WT}(R(\text{IRT}(S))) \) maintains the structure of \( R(\text{IRT}(S)) \) and supports Rank and Select in addition to character-wise access (c.f. [13]).

In addition to the compressed representation of \( \text{IRT}(S) \), a space efficient representation of \( \text{Sort}(\text{IRT}(S)) \) is required, which supports Rank and Select. To achieve this goal, an array \( C \) is being used, such that for each character \( c \), \( C[c] \) contains the frequency of all characters occurring in \( S \) (and \( \text{IRT}(S) \) respectively) and being lexicographically smaller than \( c \). Thus, \( C[c]+1 \) is equal to the position of the first occurrence of \( c \) in \( \text{Sort}(\text{IRT}(S)) \), such that \( C \) can be used to calculate Rank and Select on \( \text{Sort}(\text{IRT}(S)) \) as follows.

\[
\text{Select}_{\text{c}}(\text{Sort}(\text{IRT}(S)), k) = C[c]+k.
\]

To calculate \( \text{Rank}_{\text{c}}(\text{Sort}(\text{IRT}(S)), p) \) using \( C \), three cases have to be distinguished. Let \( c_n \) denote the lexicographically next larger character of \( c \).

Case 1, \( p \leq C[c] \): position \( p \) is smaller than the position of the first occurrence of \( c \) in \( \text{Sort}(\text{IRT}(S)) \), i.e. \( \text{Rank}_{\text{c}}(\text{Sort}(\text{IRT}(S)), p) = 0 \).

Case 2, \( p > C[c] \): position \( p \) greater than the position of the last occurrence of \( c \) in \( \text{Sort}(\text{IRT}(S)) \), i.e. 
\[
\text{Rank}_{\text{c}}(\text{Sort}(\text{IRT}(S)), p) = C[c]-C[p].
\]

Case 3, \( p > C[c] \) and \( p \leq C[c_n] \): position \( p \) is located inside the occurrences of \( c \) in \( \text{Sort}(\text{IRT}(S)) \), i.e. 
\[
\text{Rank}_{\text{c}}(\text{Sort}(\text{IRT}(S)), p) = C[c_n]-p.
\]
4.2 Search, insertion and deletion of substrings in IRT(S) implemented on RLE and WT

Since RLE and WT support Rank, Select, and character-wise access (c.f. Section 4.1), decoding and search can be executed directly on the compressed data structure, i.e. on B(IRT(S)), B'(IRT(S)), and WT(R(IRT(S))), without requiring full reconstruction of R(IRT(S)) or IRT(S) or S. Therefore, the bit vectors of the compressed data structure have to support Rank and Select efficiently, as for example proposed in [15], where \( \text{Rank}_b(BV, p) \) returns the count of occurrences of \( b \)-bits, \( b \in \{0, 1\} \), up to position \( p \) in \( BV \) and \( \text{Select}(BV, p) \) returns the position of the \( p \)-th \( b \)-bit in \( BV \).

In addition to supporting Rank, Select and character-wise access (c.f. Section 4.1), RLFMI and WT can be extended to support insertion and deletion of single characters as follows, which can be generalized to the insertion or deletion of words as described in sections 3.1 and 3.2. The insertion of a character \( c \) into IRT(S) at position \( p \), can be simulated on RLFMI and WT, i.e. on B(IRT(S)), B'(IRT(S)), and WT(R(IRT(S))), as follows.

For each character \( c \) inserted at position \( p \) into IRT(S), two cases have to be distinguished. Case 1: If \( c \) starts no new run in IRT(S), i.e. \( p \geq 1 \) and the character at position \( p-1 \) in IRT(S) is equal to \( c \), a 0-bit is inserted at position \( p \) in B(IRT(S)). Case 2: If \( c \) starts a new run at position \( p \) in IRT(S), i.e. the character at position \( p-1 \) in IRT(S) differs from \( c \) or \( p = 1 \), a 1-bit is inserted at position \( p \) in B(IRT(S)). If \( c \) is not the last character of IRT(S), the same two cases have to be checked for the right neighbor \( c_n \) of character \( c \) at position \( p+1 \) in IRT(S). As the left neighbor of \( c_n \) in IRT(S) has changed, the bit at position \( p+1 \) in IRT(S) might have to be changed too. If the frequency of 1-bits in IRT(S) increases because of the insertion of a character \( c \), new characters have to be inserted into WT(R(IRT(S))). A character \( c' \) can be inserted into W(R(IRT(S))) at position \( p' \) as follows: Each bit of the Huffman code \( h(c') \) of \( c' \) is inserted into one node on the path from the root of WT(R(IRT(S))) to the leaf node representing \( c' \). Depending on whether the current bit of \( h(c') \) in the current node \( k \) is 0 or 1, further bits of \( h(c') \) are inserted into the root node of the left or the right sub-tree at position \( p' = \text{Rank}_0(k, p') + 1 \) or at position \( p' = \text{Rank}_1(k, p') + 1 \). Thereby, the bits are inserted into the nodes of \( k \) in such a way, that \( c' \) could be decoded correctly starting from the root of W(R(IRT(S))).

The deletion of a character \( c \) in IRT(S) at position \( p \) can be simulated on RLFMI and WT, i.e. on B(IRT(S)), B'(IRT(S)), and WT(R(IRT(S))). It can be done inverse to the insertion of a character \( c \) by removing all bits representing \( c \) in B(IRT(S)), B'(IRT(S)) and WT(R(IRT(S))) and, if needed, modifying the right neighbor of the deleted bit in B(IRT(S)).

Since for all cases of deletion and insertion, B(IRT(S)) has to be changed, B'(IRT(S)) has to be revised, such that it contains the runs of B(IRT(S)) in sorted order according to the characters that form the run.

To summarize, insertion and deletion can be executed directly on the compressed data structure, i.e. on B(IRT(S)), B'(IRT(S)) and WT(R(IRT(S))), without any decompression to IRT(S) or to S.

5. Evaluation

We have implemented IRT and have evaluated it on the datasets shown in Table 1.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bible</td>
<td>The Bible</td>
<td>4.047.392 bytes</td>
</tr>
<tr>
<td>rc</td>
<td>Robinson Crusoe written by Daniel Dafoe</td>
<td>680.528 bytes</td>
</tr>
<tr>
<td>b98</td>
<td>Records of statistical data of the East Division National Baseball League 1998</td>
<td>68.507 bytes</td>
</tr>
</tbody>
</table>

Table 1: Test data

In the following tests, \( S \) denotes the input text consisting of one of the test documents, and CIRT(S) denotes IRT(S) being compressed with RLFMI and WT.

In a first series of measurements, we compared the compression ratios on different datasets (c.f. Figure 4). For CIRT(S), the sum of the sizes of the bit vectors of WT(IRT(S)) and B(IRT(S)) are measured, since B'(IRT(S)) and C can be computed efficiently from the data formats B(IRT(S)) and WT(R(IRT(S))) when loading these data formats from disk.
The compression ratio of CIRT, i.e. applying IRT followed by RLFMI and WT to an input text, is between 33-45% weaker than the compression ratio of bzip2. When compressing natural language, CIRT compresses documents to 40% of their original size on average. For data containing fewer repetitions, the compression ratio increases by the same factor as for bzip2.

In a second series of measurements, we compared the compression times and decompression times on different datasets (c.f. Figure 5). Compressing larger documents (bible, rc) requires 25%-33% more time for CIRT in comparison to bzip2. When compressing small documents, the time needed by CIRT and by bzip2 is equal. Decompression using CIRT is up to 5 times slower than when using bzip2.

In a third series of measurements, we evaluated times needed for insertion and deletion of data into parts of the document ‘bible’ at specified positions, having compressed it with bzip2 and CIRT. To modify ‘bible’ being compressed with bzip2, we decompress it, modify the uncompressed text, and compress it afterwards. We have determined a boundary, up to which inserting at specified positions into ‘bible’ or deleting at specified positions from ‘bible’ being compressed with CIRT is faster than decompressing a text with bzip2, performing the same insert or delete operation on the uncompressed text and compressing the resulting text with bzip2. For insertions at specified positions into compressed documents, using CIRT is faster than using bzip2 up to a boundary of 18% of the original document size being inserted (c.f. Figure 6). Times for inserting data into a 1185kB sized document are illustrated in Figure 7. For deletions from compressed documents, using CIRT is faster than using bzip2 up to a boundary of 30% of the document size.
Figure 6: Insert and delete boundary for ‘bible’ comparing CIRT and bzip2

Figure 7: Insertion of data into document (having a size of 1185kB) comparing CIRT and bzip2

Figure 8: Time for queries on ‘bible’ compressed using CIRT compared with decompression time needed for bzip2

Figure 8 shows that searching for words fulfilling given criteria, i.e. being smaller than ‘Moses’ (1), being equal to ‘Moses’ (2), being smaller than ‘Moses’ but not being equal to ‘God’ (3), and having substring ‘ve’ (4), takes less time than only decompressing ‘bible’ if compressed with bzip2 (5). Note that if using bzip2, the search would have to be executed after decompression and would require some additional time (which depends on the technique to search the uncompressed data), i.e. the advantage of CIRT over bzip2 is even greater than shown in Figure 8.

On average, CIRT performs better than bzip2 for queries that can be executed directly on the compressed data. This applies to search queries, insertions (up to a size of 18% of the original data size) and deletions (up to a size of 30% of the original data size).

As CIRT combines compression of data with an additional index to words, the compression ratio is between 33%-50% weaker than that of bzip2. The times needed by CIRT for data compression or total decompression of compressed data is on average larger than the times needed for bzip2. Altogether, CIRT appears to be superior bzip2 when inserting or deleting small parts of the compressed texts or when searching in compressed texts is the bottleneck of processing compressed texts.

6. Summary and Conclusions

Position-based search and modification in compressed texts is not possible for common text compressors without prior decompression. We have developed an approach to block-sorting compression, called IRT, that combines text transformation, to reach better compressibility of the transformed text, with a self-index to words. As a consequence, IRT supports position-based searching, deleting, and inserting of arbitrary substrings without prior retransformation. Furthermore, we have described how to
compress \(IRT(S)\) to \(CIRT(S)\) by using RLFMI and WT, such that \(CIRT(S)\) supports position-based searching, deleting, and inserting of arbitrary substrings without prior decompression of \(CIRT(S)\) to \(IRT(S)\) or to \(S\).

Our experimental results have shown that \(CIRT\) is comparable in compression ratio and speed to bzip2. Furthermore, for a given text \(S\), queries can be executed faster on \(CIRT(S)\) than it takes to only decompress text compressed by bzip2 back to \(S\). Insertions into \(CIRT(S)\) up to a size of 18% of the size of \(S\) are faster than decompressing \(S\) by bzip2 and decompressing back to \(S\). Deletions from \(CIRT(S)\) up to a size of even 30% of the size of \(S\) are faster than decompressing to \(S\) followed by compressing \(S\) when using bzip2. To summarize, we consider \(CIRT\) as a useful text compression technique if queries or small modifications need to be executed frequently on the compressed text.

We expect that our approach can be extended to other operations on \(IRT(S)\) and \(CIRT(S)\), e.g., that sorting the words of \(S\) can be done without full decompression of \(CIRT(S)\) into \(S\).

References