FUNCTIONAL PEARLS

Inverting the Burrows-Wheeler Transform

RICHARD BIRD and SHIN-CHENG MU

Programming Research Group, Oxford University
Wolfson Building, Parks Road, Oxford, OX1 3QD, UK

Abstract

The Burrows-Wheeler Transform is a string-to-string transform which, when used as a preprocessing phase in compression, significantly enhances the compression rate. However, it often puzzles people how the inverse transform is carried out. In this pearl we to exploit simple equational reasoning to derive the inverse of the Burrows-Wheeler transform from its specification. We also outline how to derive the inverse of two more general versions of the transform, one proposed by Schindler and the other by Chapin and Tate.

1 Introduction

The Burrows-Wheeler Transform (Burrows & Wheeler, 1994) is a method for permuting a string with the aim of bringing repeated characters together. As a consequence, the permuted string can be compressed effectively using simple techniques such as move-to-front or run-length encoding. In (Nelson, 1996), the article that brought the BWT to the world's attention, it was shown that the resulting compression algorithm could outperform many commercial programs available at the time. The BWT has now been integrated into the high-performance utility `bzip2`, available from (Seward, 2000).

Clearly the best way of bringing repeated characters together is just to sort the string. But this idea has a fatal flaw as a preliminary to compression: there is no way to recover the original string unless the complete sorting permutation is produced as part of the output. Instead, the BWT achieves a more modest permutation, one that aims to bring some but not all repeated characters into adjacent positions. Moreover, the transform can be inverted using a single additional piece of information, namely an integer b in the range $0 \leq b < n$, where n is the length of the output (or input) string.

It often puzzles people, at least on a first encounter, as to why the BWT is invertible and how the inversion is actually carried out. We identify the fundamental reason why inversion is possible and use it to derive the inverse transform from its specification. As a bonus, we can further derive the inverse of two variations of the BWT transform, one proposed in (Schindler, 1997), another in (Chapin & Tate, 1998).

2 Defining the BWT

The BWT is specified by two functions: $bwp :: String \to String$, which permutes the string and $bwn :: String \to Int$, which computes the supplementary integer. The restriction to strings is not essential to the transform, and we can take bwp to have the Haskell type $Ord\ a \Rightarrow [a] \to [a]$, so lists of any type will do provided there is a total ordering relation on the elements. The function bwp is defined by

$$bwp \quad = \quad map\ last \cdot lexsort \cdot rots \tag{1}$$

The function $lexsort :: Ord\ a \Rightarrow [[a]] \to [[a]]$ sorts a list of lists into lexicographic order and is considered in greater detail in the following section. The function $rots$ returns the rotations of a list and is defined by

$$\begin{array}{lcl}
rots & :: & [a] \to [[a]] \\
rots\ xs & = & take\ (length\ xs)\ (iterate\ lrot\ xs) \\
lrot & :: & [a] \to [a] \\
lrot\ xs & = & tail\ xs +\!\!+ [head\ xs]
\end{array}$$

The function $lrot$ performs a single left rotation. In words, (1) reads: take the last column in the lexicographically sorted matrix of rotations of the input. The definition of bwp is constructive, but we won't go into details – at least, not in this pearl – as to how the program can be made more efficient.

The function bwn is specified by

$$lexsort\ (rots\ xs)\ !!\ bwn\ xs \quad = \quad xs \tag{2}$$

where $ys\ !!\ k$ returns the element of ys in position k. In words, $bwn\ xs$ returns some position at which xs occurs in the sorted list of rotations of xs. If xs is a repeated string, then $rots\ xs$ will contain duplicates, so $bwn\ xs$ is not defined uniquely by (2).

As an illustration, consider the string `yokohama`. The rotations and the lexicographically sorted rotations are as follows:

```
y o k o h a m a        0  a m a y o k o h
o k o h a m a y        1  a y o k o h a m
k o h a m a y o        2  h a m a y o k o
o h a m a y o k        3  k o h a m a y o
h a m a y o k o        4  m a y o k o h a
a m a y o k o h        5  o h a m a y o k
m a y o k o h a        6  o k o h a m a y
a y o k o h a m        7  y o k o h a m a
```

The output of bwp is the string `hmooakya`, the last column of the second matrix, and bwn `"yokohama"` $= 7$ because row number 7 in the sorted matrix of rotations is the input string.

The BWT helps compression because it brings together characters with a common context. To give a brief illustration, an English text may contain many occurrences of words such as "this", "the", "that" and some occurrences of "where", "when", "she", " he" (with a space), etc. Consequently, many of the rotations beginning with "h" will end with a "t", some with a "w", an "s" or a space. The chance is smaller that a rotation beginning with "h" would end in a "x", a "q", or an "u", etc. Thus the BWT brings together a subset of the letters, say, those "t"s, "w"s and "s"s. A move-to-front encoding phase is then able to convert the characters into a series of small-numbered indexes, which improves the effectiveness of entropy-based compression techniques such as Huffman or arithmetic coding. For a fuller understanding of the role of the BWT in data compression, consult (Burrows & Wheeler, 1994; Nelson, 1996).

The inverse transform $\mathit{unbwt} :: \mathit{Ord}\ a \Rightarrow \mathit{Int} \rightarrow [a] \rightarrow [a]$ is specified by

$$\mathit{unbwt}\,(\mathit{bwn}\,\mathit{xs})\,(\mathit{bwp}\,\mathit{xs}) \quad = \quad \mathit{xs} \tag{3}$$

To compute *unbwt* we have to show how the matrix of lexicographically sorted rotations, or at least row number t, where $t = \mathit{bwn}\,\mathit{xs}$, can be recreated solely from the knowledge of its last column. To do so we need to examine lexicographic sorting in more detail.

3 Lexicographic sorting

Let $(\leq) :: a \rightarrow a \rightarrow \mathit{Bool}$ be a linear ordering on a. Define $(\leq_k) :: [a] \rightarrow [a] \rightarrow \mathit{Bool}$ inductively by

$$\begin{array}{lcl} \mathit{xs} \leq_0 \mathit{ys} & = & \mathit{True} \\ (x : \mathit{xs}) \leq_{k+1} (y : \mathit{ys}) & = & x < y \ \vee \ (x = y \ \wedge \ \mathit{xs} \leq_k \mathit{ys}) \end{array}$$

The value $\mathit{xs} \leq_k \mathit{ys}$ is defined whenever the lengths of *xs* and *ys* are both no smaller than k.

Now, let $\mathit{sort}\,(\leq_k) :: [[a]] \rightarrow [[a]]$ be a stable sorting algorithm that sorts an $n \times n$ matrix, given as a list of lists, according to the ordering $\leq_k$. Thus $\mathit{sort}\,(\leq_k)$, which we henceforth abbreviate to $\mathit{sort}\,k$, sorts a matrix on its first k columns. Stability means that rows with the same first k elements appear in their original order in the output matrix. By definition, $\mathit{lexsort} = \mathit{sort}\,n$.

Define $\mathit{cols}\,j = \mathit{map}\,(\mathit{take}\,j)$, so $\mathit{cols}\,j$ returns the first j columns of a matrix. Our aim in this section is to establish the following fundamental relationship, which is the key property for establishing the existence of an algorithm for inverting the BWT. Provided $1 \leq j \leq k$, we have

$$\mathit{cols}\,j \cdot \mathit{sort}\,k \cdot \mathit{rots} \quad = \quad \mathit{sort}\,1 \cdot \mathit{cols}\,j \cdot \mathit{map}\,\mathit{rrot} \cdot \mathit{sort}\,k \cdot \mathit{rots} \tag{4}$$

The function *rrot* denotes a single right rotation, defined by:

$$\mathit{rrot}\,\mathit{xs} \quad = \quad \mathit{last}\,\mathit{xs} : \mathit{init}\,\mathit{xs}$$

Equation (4) looks daunting, but take $j = n$ (so $\mathit{cols}\,j$ is the identity) and $k = n$ (so $\mathit{sort}\,k$ is a complete lexicographic sorting). Then (4) states that the following transformation on the sorted rotations is the identity: move the last column to the front and re-sort the rows on the new first column. As we will see, this implies that the permutation that produces the first column from the last column is the same as that which produces the second from the first, and so on.

To prove (4) we will need some basic properties of rotations and sorting. For rotations, one identity suffices:

$$\mathit{map\ rrot} \cdot \mathit{rots} \quad = \quad \mathit{rrot} \cdot \mathit{rots} \tag{5}$$

More generally, applying a rotation to the columns of a matrix of rotations has the same effect as applying the same rotation to the rows.

For sorting we will need

$$\mathit{sort}\ k \cdot \mathit{map\ rrot}^k \quad = \quad (\mathit{sort}\ 1 \cdot \mathit{map\ rrot})^k \tag{6}$$

where f^k is the composition of f with itself k times. Equivalently, equation (6) can be read as $\mathit{sort}\ k = (\mathit{sort}\ 1 \cdot \mathit{map\ rrot})^k \cdot \mathit{map\ lrot}^k$. This identity formalises the fact that one can sort a matrix on its first k columns by first rotating the matrix to bring these columns into the last k positions, and then repeating k times the process of rotating the last column into first position and *stably* sorting according to the first column only. Since $\mathit{map\ rrot}^n = \mathit{id}$, the initial processing is omitted in the case $k = n$, and we have the standard definition of *radix sort*. In this context see (Gibbons, 1999) which deals with the derivation of radix sorting in a more general setting.

Substituting $k+1$ for k in (6) and expanding the right-hand side, we obtain

$$\mathit{sort}\ (k+1) \cdot \mathit{map\ rrot}^{k+1} \quad = \quad \mathit{sort}\ 1 \cdot \mathit{map\ rrot} \cdot \mathit{sort}\ k \cdot \mathit{map\ rrot}^k$$

Since $\mathit{rrot}^k \cdot \mathit{rrot}^{n-k} = \mathit{rrot}^n = \mathit{id}$ we can compose both sides with $\mathit{map\ rrot}^{n-k}$ to obtain

$$\mathit{sort}\ (k+1) \cdot \mathit{map\ rrot} \quad = \quad \mathit{sort}\ 1 \cdot \mathit{map\ rrot} \cdot \mathit{sort}\ k \tag{7}$$

Finally, we will need the following two properties of columns. Firstly, for arbitrary j and k:

$$\mathit{cols}\ j \cdot \mathit{sort}\ k \quad = \quad \mathit{cols}\ j \cdot \mathit{sort}\ (j\ \mathbf{min}\ k) \ = \ \mathit{sort}\ (j\ \mathbf{min}\ k) \cdot \mathit{cols}\ j \tag{8}$$

In particular, $\mathit{cols}\ j \cdot \mathit{sort}\ k = \mathit{cols}\ j \cdot \mathit{sort}\ j$ whenever $j \leq k$. Furthermore, since $\mathit{sort}\ k$ sorts the list of strings by the first k characters only, we have:

$$\mathit{cols}\ j \cdot \mathit{sort}\ k \cdot \mathit{perm} \quad = \quad \mathit{cols}\ j \cdot \mathit{sort}\ k \tag{9}$$

whenever $j \leq k$ and *perm* is any function that permutes its argument.

Having introduced the fundamental properties (5), (7), (8) and (9), we can now prove (4). With $1 \leq j \leq k$ we reason:

$$\begin{array}{ll}
 & \mathit{sort}\ 1 \cdot \mathit{cols}\ j \cdot \mathit{map\ rrot} \cdot \mathit{sort}\ k \cdot \mathit{rots} \\
= & \quad \{\text{by } (8)\} \\
 & \mathit{cols}\ j \cdot \mathit{sort}\ 1 \cdot \mathit{map\ rrot} \cdot \mathit{sort}\ k \cdot \mathit{rots} \\
= & \quad \{\text{by } (7)\} \\
 & \mathit{cols}\ j \cdot \mathit{sort}\ (k+1) \cdot \mathit{map\ rrot} \cdot \mathit{rots} \\
= & \quad \{\text{by } (8)\} \\
 & \mathit{cols}\ j \cdot \mathit{sort}\ k \cdot \mathit{map\ rrot} \cdot \mathit{rots}
\end{array}$$

$$
\begin{array}{ll}
= & \{\text{by } (5)\} \\
& \mathit{cols}\, j \cdot \mathit{sort}\, k \cdot \mathit{rrot} \cdot \mathit{rots} \\
= & \{\text{by } (9)\} \\
& \mathit{cols}\, j \cdot \mathit{sort}\, k \cdot \mathit{rots}
\end{array}
$$

Thus, (4) is established.

4 The derivation

Our aim is to develop a program that reconstructs the sorted matrix from its last column. In other words, we aim to construct $\mathit{sort}\, n \cdot \mathit{rots} \cdot \mathit{unbwt}\, t$. In fact, we will try to construct a more general expression $\mathit{cols}\, j \cdot \mathit{sort}\, k \cdot \mathit{rots} \cdot \mathit{unbwt}\, t$ (of which the former expression is the case $j = k = n$) because the more general expression is used in the two variants of the BWT described in Sections 5 and 6.

First observe that for $0 \leq j$,

$$
\mathit{cols}\,(j+1) \cdot \mathit{map}\; \mathit{rrot} \quad = \quad \mathit{join} \cdot \mathit{fork}\,(\mathit{map}\; \mathit{last}, \mathit{cols}\, j) \tag{10}
$$

where $\mathit{join}\,(\mathit{xs}, \mathit{xss}) = \mathit{zipWith}\,(:)\, \mathit{xs}\, \mathit{xss}$ is the matrix xss with xs adjoined as a new first column, and $\mathit{fork}\,(f, g)\, x = (f\, x, g\, x)$. Hence

$$
\begin{array}{ll}
& \mathit{cols}\,(j+1) \cdot \mathit{sort}\, k \cdot \mathit{rots} \cdot \mathit{unbwt}\, t \\
= & \{\text{by } (4)\} \\
& \mathit{sort}\, 1 \cdot \mathit{cols}\,(j+1) \cdot \mathit{map}\; \mathit{rrot} \cdot \mathit{sort}\, k \cdot \mathit{rots} \cdot \mathit{unbwt}\, t \\
= & \{\text{by } (10)\} \\
& \mathit{sort}\, 1 \cdot \mathit{join} \cdot \mathit{fork}\,(\mathit{map}\; \mathit{last}, \mathit{cols}\, j) \cdot \mathit{sort}\, k \cdot \mathit{rots} \cdot \mathit{unbwt}\, t \\
= & \{\text{since } \mathit{fork}\,(f, g) \cdot h = \mathit{fork}\,(f \cdot h, g \cdot h)\} \\
& \mathit{sort}\, 1 \cdot \mathit{join} \cdot \mathit{fork}\,(\mathit{map}\; \mathit{last} \cdot \mathit{sort}\, k \cdot \mathit{rots} \cdot \mathit{unbwt}\, t, \\
& \qquad\qquad\qquad \mathit{cols}\, j \cdot \mathit{sort}\, k \cdot \mathit{rots} \cdot \mathit{unbwt}\, t)
\end{array}
$$

In particular, consider $t = \mathit{bwn}\, \mathit{xs}$ for an input xs and $k = n$, the length of xs. Since $\mathit{bwp} = \mathit{map}\; \mathit{last} \cdot \mathit{sort}\, n \cdot \mathit{rots}$, and $\mathit{bwp}\,(\mathit{unbwt}\, t\, \mathit{xs}) = \mathit{xs}$, the equality shown above reduces to:

$$
\begin{array}{l}
(\mathit{cols}\,(j+1) \cdot \mathit{sort}\, n \cdot \mathit{rots} \cdot \mathit{unbwt}\, t)\, \mathit{xs} \\
\qquad = (\mathit{sort}\, 1 \cdot \mathit{join} \cdot \mathit{fork}\,(\mathit{id}, \mathit{cols}\, j \cdot \mathit{sort}\, n \cdot \mathit{rots} \cdot \mathit{unbwt}\, t))\, \mathit{xs}
\end{array}
$$

Setting $\mathit{recreate}\, j = \mathit{cols}\, j \cdot \mathit{sort}\, n \cdot \mathit{rots} \cdot \mathit{unbwt}\, t$, we have just constructed a recursive definition for *recreate*:

$$
\begin{array}{lcl}
\mathit{recreate}\, 0 & = & \mathit{map}\,(\mathit{const}\,[\,]) \\
\mathit{recreate}\,(j+1) & = & \mathit{sort}\, 1 \cdot \mathit{join} \cdot \mathit{fork}\,(\mathit{id}, \mathit{recreate}\, j)
\end{array}
$$

The Haskell code for *recreate* is given in Figure 1. The function $\mathit{sortby} :: (a \rightarrow a \rightarrow \mathit{Bool}) \rightarrow [a] \rightarrow [a]$ is a stable sorting algorithm. It is identical to the standard function *sortBy* except for a slightly different type.

Now we know that *recreate* reconstructs the entire matrix, we just need to pick

```
recreate :: Ord a => Int -> [a] -> [[a]]
recreate 0     = map (const [])
recreate (j+1) = sortby leq . join . fork (id, recreate j)
    where leq us vs = head us <= head vs
          join = uncurry (zipWith (:))
          fork (f,g) x = (f x, g x)
```

Fig. 1. Computation of *recreate*

a particular row. Taking $j = n$, we have $\mathit{unbwt}\,t = (!!\,t) \cdot \mathit{recreate}\,n$. The problem is that this implementation of *unbwt* involves computing $\mathit{sort}\,1$ a total of n times. To avoid repeated sorting, observe that $\mathit{recreate}\,1\,\mathit{ys} = \mathit{sort}\,\mathit{ys}$, where the function *sort* now sorts a list rather than a matrix of one column. Furthermore, for some suitable permutation *sp* we have

$$\mathit{sort}\,\mathit{ys} \quad = \quad \mathit{permby}\,\mathit{sp}\,\mathit{ys}$$

where $\mathit{permby} :: (\mathit{Int} \to \mathit{Int}) \to [a] \to [a]$ applies a permutation to a list:

$$\mathit{permby}\,p\,[x_0, \ldots, x_{n-1}] \quad = \quad [x_{p(0)}, \ldots, x_{p(n-1)}]$$

It follows that

$$\mathit{recreate}\,(j+1)\,\mathit{ys} \quad = \quad \mathit{join}\,(\mathit{permby}\,\mathit{sp}\,\mathit{ys}, \mathit{permby}\,\mathit{sp}\,(\mathit{recreate}\,j\,\mathit{ys}))$$

Equivalently,

$$\mathit{recreate}\,n\,\mathit{ys} \quad = \quad \mathit{transpose}\,(\mathit{take}\,n\,(\mathit{iterate}1\,(\mathit{permby}\,\mathit{sp})\,\mathit{ys})) \tag{11}$$

where $\mathit{transpose} :: [[a]] \to [[a]]$ is the standard Haskell function for transposing a matrix and $\mathit{iterate}1 = \mathit{tail} \cdot \mathit{iterate}$. The tth row of a matrix is the tth column of the transposed matrix, that is, $(!!\,t) \cdot \mathit{transpose} = \mathit{map}\,(!!\,t)$, so we can use the naturality of $\mathit{take}\,n$ to obtain

$$\mathit{unbwt}\,t\,\mathit{ys} \quad = \quad \mathit{take}\,n\,(\mathit{map}\,(!!\,t)\,(\mathit{iterate}1\,(\mathit{permby}\,\mathit{sp})\,\mathit{ys}))$$

Suppose we define $\mathit{spl} :: \mathit{Ord}\,a \Rightarrow [a] \to \mathit{Int} \to (a, \mathit{Int})$ by

$$\mathit{spl}\,\mathit{ys}\,i \quad = \quad \mathit{sort}\,(\mathit{zip}\,\mathit{ys}\,[1..])\,!!\,i$$

where *sort* now sorts a list of pairs. Then

$$\mathit{spl}\,\mathit{ys}\,j \quad = \quad (\mathit{ys}\,!!\,\mathit{sp}\,j, \mathit{sp}\,j)$$

Hence

$$\mathit{map}\,(!!\,k)\,(\mathit{iterate}1\,(\mathit{permby}\,\mathit{sp})\,\mathit{ys}) \quad = \quad \mathit{thread}\,(\mathit{spl}\,\mathit{ys}\,k)$$

where $\mathit{thread}\,(x, j) = x : \mathit{thread}\,(\mathit{spl}\,\mathit{ys}\,j)$.

The final algorithm, written as a Haskell program, is given in Figure 2. In a real implementation, the sorting in *spl* would be performed by counting the histogram of the input, which can be done in linear time using a mutable array. The "threading" part can be performed in linear time, assuming constant-time array look-up. In (Seward, 2001) it was observed that the main inefficiency with the algorithm lies in

```
unbwt :: Ord a => Int -> [a] -> [a]
unbwt t ys = take (length ys) (thread t)
  where spl i =  sort (zip ys [0..]) !! i
        thread i = x : thread j
           where (x,j) = spl i
```

Fig. 2. Computation of *unbwt*

the cache misses involved in the threading, arising as a result of accessing a large array in non-sequential order.

5 Schindler's variation

The main variation of BWT is to exploit the general form of (4) rather than the special case $k = n$. Suppose we define

$$bwpS\,k \quad = \quad map\,last \cdot sort\,k \cdot rots$$

This version, which sorts only on the first k columns of the rotations of a list, was considered in (Schindler, 1997). The derivation of the previous section shows how we can recreate the first k columns of the sorted rotations from $ys = bwp\,k\,xs$, namely by computing $recreate\,k\,ys$.

The remaining columns cannot be computed in the same way. However, we can reconstruct the tth row, where $t = bwn\,k\,xs$ and

$$sort\,k\,(rots\,xs)\,!!\,t \quad = \quad xs$$

The first k elements of xs are given by $create\,k\,ys\,!!\,t$, and the last element of xs is $ys\,!!\,t$. Certainly we know

$$take\,k\,(rrot\,xs) \quad = \quad [x_n, x_1, \ldots, x_{k-1}]$$

This list begins with the *last* row of the unsorted matrix, and consequently, since sorting is stable, will be the *last* occurrence of the list in $create\,k\,ys$. If this occurrence is at position p, then $ys\,!!\,p = x_{n-1}$. Having discovered x_{n-1}, we know $take\,k\,(rrot^2\,xs)$. This list begins the penultimate row of the unsorted matrix, and will be either the last occurrence of the list in the sorted matrix, or the penultimate one if it is equal to the previous list. We can continue this process to discover all of $[x_{k+1}, \ldots, x_n]$ in reverse order. Efficient implementation of this phase of the algorithm requires building an appropriate data structure for repeatedly looking up elements in reverse order in the list $zip\,(recreate\,k\,ys)\,ys$ and removing them when found. A simple implementation is given in Figure 3.

6 Chapin and Tate's variation

Primarily for the purpose of showing that the pattern of derivation in this paper can be adapted to other cases, we will consider another variation. Define the following alternative of BWT:

$$bwpCT\,k \quad = \quad map\,last \cdot twist^k \cdot lexsort \cdot rots$$

```
unbwt :: Ord a => Int -> Int -> [a] -> [a]
unbwt k p ys = us ++ reverse (take (length ys - k) vs)
  where us  = yss !! p
        yss = recreate k ys
        vs  = u:search k (reverse (zip yss ys)) (take k (u:us))
        u   = ys !! p

search :: Eq a => Int -> [([a],a)] -> [a] -> [a]
search k table xs = x:search k table' (take k (x:xs))
                    where (x,table') = dlookup table xs

dlookup :: Eq a => [(a,b)] -> a -> (b,[(a,b)])
dlookup ((a,b):abs) x = if a==x then (b,abs)
                        else (c,(a,b):cds)
                        where (c,cds) = dlookup abs x
```

Fig. 3. Computation of Schindler's variation

```
twist :: Eq a => Int -> [[a]] -> [[a]]
twist k = concat . mapEven (map reverse). groupby (take k)

mapEven, mapOdd :: (a->a) -> [a] -> [a]
mapEven f [] = []
mapEven f (x:xs) = f x : mapOdd f xs
mapOdd f [] = []
mapOdd f (x:xs) = x : mapEven f xs
```

Fig. 4. One possible choice of *twist*

where the function *twist* rearranges the rows of the matrix. One possible choice of *twist* is shown in Figure 4. As an example, consider the rotations of the string `aabab`:

```
aabab    ababa    abaab
abaab    abaab    ababa
ababa    aabab    aabab
baaba    baaba    babaa
babaa    babaa    baaba
```

Shown on the left is the sorted matrix of rotations. The matrix in the middle is the result of applying *twist*. The rows are first partitioned into groups by *groupby* according to their first characters. The even numbered groups (counting from zero) are then reversed. In the example, the group starting with `a` is reversed. Shown on the right is the result of applying $twist^2$ to the matrix in the middle. The rows are partitioned into three groups, starting with `ab`, `aa`, and `ba` respectively. The noughth and the second group are reversed.

The idea of twisting the matrix of sorted rotations was proposed in (Chapin & Tate, 1998), where a similar but slightly more complicated version of *twist* was considered based on the Gray code. Chapin and Tate's generalisation can marginally improve the compression ratio of the transformed text.

For $twist^k$ to be invertible, however, we need only the property that for $0 < j \leq k$,

$$cols\, j \cdot twist^k \quad = \quad cols\, j \cdot twist^{j-1} \tag{12}$$

In words, further twisting ($twist^k$ where $j \leq k$) does not change the first j columns after they have been set by $twist^{j-1}$. In the example above, for instance, the call to $twist^2$ does not change the first two columns of the matrix in the middle, nor do successive calls to $twist^k$ where $k \geq 2$. Any *twist* satisfying (12) suffices to make *bwtCT* invertible. This separation of concerns on compression rate and invertibility means that one can try many possible choices satisfying (12) and experiment with the effect on compression.

To derive an algorithm for the reverse transform we need the following analogue of (4):

$$\begin{aligned} & cols\, j \cdot twist^k \cdot lexsort \cdot rots \\ = \quad & twist^k \cdot sort\, 1 \cdot untwist^k \cdot cols\, j \cdot map\, rrot \cdot twist^k \cdot lexsort \cdot rots \end{aligned} \tag{13}$$

where *untwist* is inverse to *twist*. The proof of (13) follows a similar path to the derivation in Section 3. When $k = 0$ (so $twist^k = id$), equation (13) reduces to a special case of (4). In words, (13) means that the following operation is an identity on a matrix generated by $twist^k \cdot lexsort \cdot rots$: move the last column to the first, untwist it, sort it by the first character, and twist it again.

Based on (13) one can now derive an algorithm similar to that of Section 4. Defining

$$recreateCT\; j\; k \quad = \quad cols\, j \cdot twist^k \cdot lexsort \cdot rots \cdot unbwtCT\; t$$

we can construct a recursive definition for *recreateCT* which is similar to (11), but with the permutation *sp* simulating $twist^i \cdot sort\, 1 \cdot untwist^i$ for appropriate i, rather than just *sort* 1. The details are more complicated than for the corresponding definition of *recreate* (which builds one column in each step) because in *recreateCT* the permutation *sp* changes each time a new column is built. So the algorithm has to construct a new permutation as well as a new column at each step. The resulting algorithm will thus return a pair whose first component is the reconstructed matrix and the second component is a permutation representating *sp*. In the first step we build the first column and a permutation simulating $twist \cdot sort\, 1 \cdot untwist$; in the second step we build the second column and a permutation for $twist^2 \cdot sort\, 1 \cdot untwist^2$, and so on. Further details are omitted in this pearl.

7 Conclusions

We have shown how the inverse Burrows-Wheeler transform can be derived by equational reasoning. The derivation can be re-used to invert the more general versions proposed by Schindler and by Chapin and Tate.

Other aspects of the BWT also make interesting topics. In (Manber & Myers, 1993) it is shown how to sort the rotations of a given string in $O(n \log n)$ time using suffix arrays, where n is the length of the string. How efficiently it can be done in a functional setting remains unanswered, though we conjecture that $O(n(\log n)^2)$ is the best possible.

Acknowledgements

We would like to thank Ian Bayley, Jeremy Gibbons, Ralf Hinze, Geraint Jones, and Barney Stratford for constructive criticism of earlier drafts of this pearl, and to Julian Seward who made some useful comments regarding the practical aspects of the Burrows-Wheeler transform. Also we would like to thank two anonymous referees for their detailed and useful comments, and for identifying many typos and infelicities in an earlier draft of this paper.

References

Burrows, M. and Wheeler, D. J. (1994) *A block-sorting lossless data compression algorithm*. Tech. rept. Digital Systems Research Center. Research Report 124. Available online at `http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-124.html`.

Chapin, B. K. and Tate, S. (1998) Higher compression from the Burrows-Wheeler transform by modified sorting. *Data Compression Conference 1998* p. 532. IEEE Computer Society Press. (Poster Session).

Gibbons, J. (1999) A pointless derivation of radixsort. *Journal of Functional Programming* **9**(3):339–346.

Manber, U. and Myers, G. (1993) Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing* **22**(5):935–948.

Nelson, M. (1996) Data compression with the Burrows-Wheeler transform. *Dr. Dobb's Journal* September.

Schindler, M. (1997) A fast block-sorting algorithm for lossless data compression. *Data Compression Conference 1997* p. 469. IEEE Computer Society Press. (Poster Session).

Seward, J. (2000) `bzip2`. `http://sources.redhat.com/bzip2/`.

Seward, J. (2001) Space-time tradeoffs in the inverse B-W transform. *Data Compression Conference 2001* pp. 439–448.