Medium-Space Algorithms for Inverse BWT[*]

Juha Kärkkäinen[1] and Simon J. Puglisi[2]

[1] Department of Computer Science, University of Helsinki, Finland
juha.karkkainen@cs.helsinki.fi
[2] School of Computer Science and Information Technology,
Royal Melbourne Institute of Technology, Australia
simon.puglisi@rmit.edu.au

Abstract. The Burrows–Wheeler transform is a powerful tool for data compression and has been the focus of intense research in the last decade. Little attention, however, has been paid to the inverse transform, even though it is a bottleneck in decompression. We introduce three new inversion algorithms with improved performance in a wide range of the space-time spectrum, as confirmed by both theoretical analysis and experimental comparison.

1 Introduction

The Burrows–Wheeler transform (BWT) [2,1] is an invertible transformation of a text that has a central role in some of the best data compression methods. The transform itself performs no compression — the result is just a permutation of the text — but the transformed text is easier to compress using simple and fast methods [15]. Much effort has gone into developing efficient algorithms for the forward transform, largely owing to its close relation to constructing the suffix array [17] and compressed text indexes [16]. The less studied problem of inverting the transform is the subject of this paper.

The inverse transform is a bottleneck in decompression and thus needs to be fast, particularly in applications requiring frequent decompression such as on-the-fly disk compression. The space requirement is also an issue: a typical, fast implementation requires 5 times the space of the text. As already proposed in the original paper [2], the text can be broken into smaller blocks, each of which is compressed separately. However, a large block size is preferable because it allows better compression (see e.g. [4]). Furthermore, the block size is determined during the forward transform, possibly on a machine with more memory or using a space efficient algorithm (e.g. [11]), and the inverse transform is impossible unless a sufficiently space-efficient algorithm is available.

The key operation in the inversion is a rank query:

$$\mathrm{rank}(j) \equiv \left|\{i \mid i < j \text{ and } L[i] = L[j]\}\right| \ ,$$

[*] This work is supported by the Academy of Finland grant 118653 (ALGODAN) and by the Australian Research Council.

M. de Berg and U. Meyer (Eds.): ESA 2010, Part I, LNCS 6346, pp. 451–462, 2010.

where L is the transformed text. A single scan of L is sufficient to answer all distinct rank queries in the sequential order, but during the inversion they are needed in a different order. By tabulating the answers, we obtain a simple, linear time inversion algorithm, already described in [2]. There are a few variations but all of them need at least $n \log n$ bits of space for the tabulated answers or something equivalent. We call these *large-space algorithms.*

Space-efficient data structures for rank queries are widely used with compressed text indexes [16], but the query needed there is of a more general form:

$$\mathrm{rank}(c, j) \equiv \left|\{i \mid i < j \text{ and } L[i] = c\}\right| .$$

We call this the *general rank query* as opposed to the *special rank query* needed in inverse BWT algorithms. Obviously, any data structure for general rank queries can be used for special rank queries, so using the techniques from compressed text indexes, we obtain space-efficient algorithms for inverse BWT. Many of these algorithms need at most $n \log \sigma + o(n \log \sigma)$ bits of space, where σ is the size of the alphabet. This is only slightly more than needed for the text itself — and sometimes even less by way of compression — but this comes at a significant cost in query time, especially in practice. We call these *small-space algorithms.*

The focus of this paper is on *medium-space algorithms* that are between large-space and small-space algorithms with respect to both time and space complexity. The key characteristic of medium-space algorithms is the tabulation of *partial* answers to all special rank queries.

Related work. Seward [19] describes several large-space BWT inversion algorithms, among them the original algorithm from [2], and compares them experimentally. One of the algorithms, `mergedTL`, is the fastest known algorithm in practice. Seward also has two algorithms in the small-space category, but they are not competitive either in theory or in practice.

The only previous medium-space algorithm we are aware of is by Lauther and Lukovszki [13]. They also propose two small-space algorithms and provide experimental results for two of their algorithms. They identify the central role of the special rank query but not its relation to the general rank query.

Ferragina, Gagie and Manzini [3] have recently described an external memory algorithm for the inversion. It is, however, rather complicated and unlikely to be competitive except when external memory algorithms are the only option.

There is a large body of research on space-efficient data structures for (general) rank queries in the area of compressed text indexes and succinct data structures. The theoretically best results on BWT inversion achievable using those techniques are reported at the bottom of Table 1.

Our contribution. We introduce three new medium-space algorithms for BWT inversion, offering improved space–time tradeoffs, including the most space-efficient linear-time algorithm for large alphabets. The time and space complexities are shown in Table 1. The table also shows our improved analysis of the only previous medium-space algorithm in [13]. Experimental results show that the favorable tradeoff properties extend to practice too.

Table 1. Time and space complexities of BWT inversion algorithms. The sections correspond to large-, medium- and small-space algorithms. The space complexities exclude the space for input, output and $\mathcal{O}(\sigma \log n)$-bit data structures.

space (bits/symbol)	time per symbol	comment
$\lceil \log n \rceil + \lceil \log \sigma \rceil$	$\mathcal{O}(1)$	`mergeTL` in [19]
$\lceil \log n \rceil$	$\mathcal{O}(\log \sigma)$	`indexF` in [19]
$\log \lceil \log n \rceil + \log \sigma + \lceil \log \sigma \rceil$	$\mathcal{O}(1)$	[13]
$1 + \log b + \lceil \log \sigma \rceil$	$\mathcal{O}(\log(n/b)$ $-H_0 + b\sigma/n)$	this paper $\lceil \log n \rceil \leq b \leq n/\sigma$
$2 + \log\big(\lceil \log n \rceil + 3\lceil \log \sigma \rceil\big)$ $+ \log \sigma + \frac{2 \log \log n}{\log n}$	$\mathcal{O}(1)$	this paper
$1 + \log b + \log \sigma$	$\mathcal{O}(\log(n/b) - \log \sigma)$	this paper $2(1 + \lceil \log n \rceil) \leq b \leq n/\sigma$
$\lceil \log \sigma \rceil + (\sigma \log n)/b$	$\mathcal{O}(b)$	[13]
$\lceil \log \sigma \rceil + \frac{\sigma}{b_1}\left(\log b_1 + \frac{\log n}{b_2}\right)$	$\mathcal{O}(b_1 + b_2)$	[13]
$H_k + \mathcal{O}\left(\frac{\log \sigma \log \log n}{\log n} + \frac{\sigma^{k+1} \log n}{n}\right)$	$\mathcal{O}(1 + \frac{\log \sigma}{\log \log n})$	[5,14]
$\log \sigma + \mathcal{O}(\frac{\log \sigma}{\log \log \sigma})$	$\mathcal{O}(\log \log \sigma)$	[6]

Perhaps of independent interest is the identification of the special rank query as an operation of interest, separate from the general rank query. The separate nature is illustrated by the fact that extending the techniques used by the large- and medium-space algorithms to general rank queries would blow up the space by factor σ, which is usually too much. We note that, besides inverse BWT, the locate and display procedures over BWT-based compressed indexes (see [16]) perform repeated special rank queries.

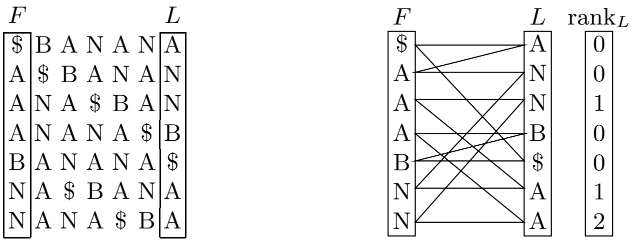
2 Preliminaries

Let $S = S[0..n] = S[0]S[1]\ldots S[n]$ be a string of $n+1$ symbols or characters. The first n characters of S are drawn from an ordered alphabet Σ, and the final character $S[n]$ is a special "end of string" symbol, \$, distinct from and lexicographically smaller than all the other symbols. We assume that the symbols in Σ are encoded with the integers $\{0, 1, .., \sigma - 1\}$ in an order preserving way.

For any $i \in 0..n$, the string $S[i..n]S[0..i-1]$ is a *rotation* of S. Let $\mathcal{M}$ be the $(n+1) \times (n+1)$ matrix whose rows are all the rotations of S in lexicographic order. Let F be the first and L the last column of $\mathcal{M}$, both taken to be strings of length $n+1$. The string L is the ***Burrows–Wheeler transform*** of S. An example is given in Fig. 1. Note that F and L are permutations of S.

For a string X, integers $j, r \in \{0, \ldots, |X| - 1\}$ and a symbol c, define the following functions:

$$\begin{aligned} \text{access}_X(j) &\equiv X[j] \\ \text{rank}_X(j) &\equiv \big|\{i \mid i < j \text{ and } X[i] = X[j]\}\big| \end{aligned}$$



F						L
\$	B	A	N	A	N	A
A	\$	B	A	N	A	N
A	N	A	\$	B	A	N
A	N	A	N	A	\$	B
B	A	N	A	N	A	\$
N	A	\$	B	A	N	A
N	A	N	A	\$	B	A

F	L	rank$_L$
\$	A	0
A	N	0
A	N	1
A	B	0
B	\$	0
N	A	1
N	A	2

Fig. 1. BWT matrix $\mathcal{M}$ and inverse BWT permutation for text $S = \text{BANANA\$}$

Inverse BWT in forward order

```
construct $F$ by sorting $L$
$j \leftarrow \text{select}_L(\$, 0)$
for $i \leftarrow 0$ to $n$ do
    $S[i] \leftarrow c \leftarrow \text{access}_F(j)$
    $r \leftarrow \text{rank}_F(j)$
    $j \leftarrow \text{select}_L(c, r)$
```

Inverse BWT in reverse order

```
construct $F$ by sorting $L$
$j \leftarrow \text{select}_L(\$, 0)$
for $i \leftarrow n$ downto $0$ do
    $S[i] \leftarrow c \leftarrow \text{access}_L(j)$
    $r \leftarrow \text{rank}_L(j)$
    $j \leftarrow \text{select}_F(c, r)$
```

Fig. 2. Two abstract algorithms for the inverse Burrows–Wheeler transform

$$\text{select}_X(c, r) \equiv \begin{cases} j & \text{if } X[j] = c \text{ and } \text{rank}_X(j) = r \\ \text{undefined} & \text{if there is no such } j \end{cases}$$

The notation $\text{access}_X(j)$ is used instead of $X[j]$ when X might be stored in a form that does not support trivial character access.

Two abstract inversion algorithms are given in Fig. 2. The first (left-hand side) algorithm constructs S from the beginning to the end and the second in the reverse order. Both algorithms follow the same unicyclic permutation but in different directions. An example of the permutation is shown in Fig. 1. To obtain concrete algorithms, we need to define the implementation of the operations access, rank, and select.

3 Basic Large-Space Algorithms

Of the two abstract algorithms in Fig. 2, we will focus on the reverse order algorithm, as its operations are easier to implement and faster in practice. All the algorithms mentioned in this paper are based on it.

Another feature shared by all the algorithms is the implementation of select_F based on the special nature of F. The string F contains the characters of S in sorted order and all copies of the same symbol are grouped together. For any symbol c, let $C[c]$ be the position of the first occurrence of c in F. We can implement select_F as

$$\text{select}_F(c, r) = C[c] + r \ .$$

The array C can be easily computed by scanning L.

The difference between various algorithms is the implementation of access_L and rank_L. We will describe next the algorithm from the seminal paper by Burrows and Wheeler [2]. They store L explicitly, making access_L trivial. The values $\mathrm{rank}_L(j)$ are stored in a table $R[0..n]$, which can be computed by scanning L while keeping account of the number of occurrences of each symbol.

The algorithm runs in linear time and needs $n(\lceil \log n \rceil + \lceil \log \sigma \rceil) + (\sigma + \mathcal{O}(1))\lceil \log n \rceil$ bits of space. It would be very fast in practice, but for cache misses. In the main loop, the sequence of accesses to L and R is essentially random with a high likelihood of a cache miss for each access. Seward [19] describes an optimized version that replaces the arrays L and R with a single array LR that stores both values. This can reduce the number of cache misses to a half, leading to a significant improvement in speed. This algorithm, which we call Algorithm `LR` (`mergeTL` in [19]), is the fastest known algorithm for BWT inversion. It is also the starting point for our medium-space algorithms.

4 Basic Medium-Space Algorithms

In this section we describe two simple medium-space algorithms. One of them is by Lauther and Lukovszki [13] and one is new.

Both algorithms modify Algorithm `LR` by storing only partial information about ranks in R (i.e., in the R-fields of the array LR). Every position $j \in \{0, \ldots, n\}$ is associated with a nearby *reference point* $\mathrm{ref}(j) \in \{0, \ldots, n\}$, and

$$R[j] = \mathrm{rank}_L(j) - \mathrm{rank}_L(L[j], \mathrm{ref}(j)) \ .$$

Now we can compute a rank query as $\mathrm{rank}_L(j) = \mathrm{rank}_L(L[j], \mathrm{ref}(j)) + R[j]$. The difference between the two algorithms is the choice of reference points and the computation of $\mathrm{rank}_L(L[j], \mathrm{ref}(j))$.

4.1 Algorithm `LR-B`

The first algorithm is by Lauther and Lukovszki [13]. We provide an improved analysis.

Divide R into $\lceil (n+1)/b \rceil$ blocks of size b. Every position in a block is associated with the same reference point, which is the center of the block. In other words, the reference points are the positions $b/2, b+b/2, 2b+b/2, \ldots$. As a small twist to the basic scheme, if j is in the first half of a block, i.e., if $j < \mathrm{ref}(j)$, we set

$$R[j] = \mathrm{rank}_L(L[j], \mathrm{ref}(j)) - \mathrm{rank}_L(j) - 1 \ .$$

Otherwise, i.e., if $j \geq \mathrm{ref}(j)$, we use the basic scheme and set

$$R[j] = \mathrm{rank}_L(j) - \mathrm{rank}_L(L[j], \mathrm{ref}(j)) \ .$$

Now all the values in R are in the range $[0, b/2 - 1]$ and can be stored using $\lceil \log b \rceil - 1$ bits. The ranks at the reference points are stored in a two-dimensional array R_{ref}, i.e., for all $c \in \Sigma$ and $j \in \{0, \ldots, \lceil (n+1)/b \rceil - 1\}$,

$$R_{\mathrm{ref}}[c, j] = \mathrm{rank}_L(c, b/2 + jb) \ .$$

We need at most $\sigma(n/b+1)\lceil \log n \rceil$ bits for the array R_{ref}.

The following theorem summarizes the properties of the algorithm. All proofs are omitted here due to lack of space and are provided in the full paper.

Theorem 1. *Setting $b = 2^k$ for $k = \lfloor \log(\sigma \lceil \log n \rceil) \rfloor$, Algorithm* `LR-B` *computes the inverse Burrows-Wheeler transform in $\mathcal{O}(n)$ time using at most*

$$n(\log \lceil \log n \rceil + \log \sigma + \lceil \log \sigma \rceil) + \mathcal{O}(\sigma \lceil \log n \rceil)$$

bits of space.

4.2 Algorithm `LR-I`

In our new algorithm, reference points are separate for each symbol of the alphabet. For a symbol c, the reference points are at every bth occurrence of c, i.e., at positions $\text{select}_L(c,0), \text{select}_L(c,b), \text{select}_L(c,2b), \ldots$. A position j is assigned to the closest preceding reference point for the symbol $L[j]$, i.e.,

$$\text{ref}(j) = \text{select}_L(L[j], b\lfloor \text{rank}_L(j)/b \rfloor) \ .$$

The array R is as in the basic scheme, i.e., $R[j] = \text{rank}_L(j) - \text{rank}_L(L[j], \text{ref}(j))$, and we need $\lceil \log b \rceil$ bits for each entry. The reference points for a symbol c are stored in an array I_c, i.e., $I_c[i] = \text{select}_L(c, ib)$. The arrays I_c, $c \in \Sigma$, can be seen as sparse inverted lists for the symbols. The total space for them is $n\lceil \log n \rceil / b + \mathcal{O}(\sigma \log n)$ bits. To compute $\text{rank}_L(j)$, we binary search $I_{L[j]}$ to find i such that $I_{L[j]}[i] \le j < I_{L[j]}[i+1]$, and then $\text{rank}_L(j) = ib + R[j]$.

Unlike `LR-B`, Algorithm `LR-I` offers a space-time tradeoff as shown by the following result.

Theorem 2. *Let $b = 2^k$ for an integral k. If $k = \lfloor \log \lceil \log n \rceil \rfloor$, the space requirement of Algorithm* `LR-I` *is at most*

$$n(1 + \log \lceil \log n \rceil + \lceil \log \sigma \rceil) + \mathcal{O}(\sigma \log n) \ .$$

For $\lfloor \log \lceil \log n \rceil \rfloor < k \le \log(n/\sigma)$, the space requirement is at most

$$n(1 + k + \lceil \log \sigma \rceil) + \mathcal{O}(\sigma \log n) \ .$$

The time complexity is $\mathcal{O}(n(\log(n/b) - H_0) + b\sigma)$, where $H_0 \le \log \sigma$ is the zeroth order empirical entropy of S (see Section 5).

5 Variable-Length Encoding

In this section, we show how to improve the algorithms of the previous section using variable-length encoding.

For a string X, let Σ_X be the set of symbols occurring in X, let $|X_c|$ be the number of occurrences of a symbol c in X, and let $f_X(c) = |X_c|/|X|$ be the frequency of c. The *zeroth order empirical entropy* of X is

$$H_0(X) = \sum_{c \in \Sigma_X} f_X(c) \log(1/f_X(c)) \ .$$

A *canonical prefix code* [18] for X is characterized by a non-decreasing sequence $\ell = (\ell_1, \ldots, \ell_{|\Sigma_X|})$ of positive, integral code lengths satisfying Kraft's inequality: $\sum_{i=1}^{|\Sigma_X|} 2^{-\ell_i} \leq 1$. The code lengths are assigned to symbols in decreasing order of symbol frequency; let $\ell(c)$ denote the code length of a symbol c. There exists an assignment of binary code words code(c) of $\ell(c)$ bits to each symbol c so that, for every $c, c' \in \Sigma_X$ with $f_X(c) < f_X(c')$,

- code(c) is not a prefix of code(c') (the code is *prefix-free*), and
- code(c) is lexicographically smaller than code(c') (the code is *canonical*).

Let $\ell(X)$ be the encoded length of X for a code ℓ:

$$\ell(X) = \sum_{i=0}^{m-1} \ell(X[i]) = m \sum_{c \in \Sigma_X} f_X(c)\ell(c) \ .$$

For any prefix code, $\ell(X) \geq mH_0(X)$. The equality is achieved with the *fractional* lengths $\ell(c) = \log(1/f_X(c))$. The Huffman code [9] is known to be the optimal code with integral lengths. However, for our purposes, we need a code where the code length of every symbol is close to the fractional optimum, which the Huffman code does not guarantee [12]. Furthermore, with one of our algorithms (`VRL-I`, Section 5.2), we have a strict upper limit h on the code lengths. We will be using the *length-limited rounded code* $\hat{\ell}_X^h$ with

$$\hat{\ell}_X^h(c) = \lceil h - \log(f_X(c)(2^h - \sigma_X) + 1) \rceil \ ,$$

for any integer $h \geq \lceil \log \sigma_X \rceil$. When there is no upper limit, the code is $\hat{\ell}_X = \hat{\ell}_X^\infty$ with $\hat{\ell}_X(c) = \lceil \log(1/f_X(c)) \rceil$. The properties of the code are established in the following lemma.

Lemma 1. *The code lengths $\hat{\ell}_X^h$ define a valid prefix code for X with $\hat{\ell}(c) \leq h$ for all $c \in \Sigma$. Furthermore, for all $c \in \Sigma$,*

$$\hat{\ell}_X^h(c) < \log(1/f_X(c)) + \log(2^h/(2^h - \sigma_X)) + 1 \ ,$$

and if $\log(1/f_X(c)) \geq h$*, then* $\hat{\ell}_X^h(c) = h$.

5.1 Algorithm `VLR-B`

As with `LR-B`, we divide the array LR into blocks of size b. The reference point for all positions in a block is now the beginning of the block (instead of the center).

Let B be a block. We encode the L-fields in B with the unlimited rounded code $\hat{\ell}_B$, and the R-fields using $\lceil \log |B_c| \rceil$ bits for a symbol c. The combined length of the two fields for a symbol c is

$$\lceil \log(1/f_B(c)) \rceil + \lceil \log |B_c| \rceil = \lceil \log(b/|B_c|) \rceil + \lceil \log |B_c| \rceil \leq \lceil \log b \rceil + 1 \ .$$

Thus we need $n(\lceil \log b \rceil + 1)$ bits for the whole LR array.

For each block B, we have a table V with an entry for each symbol c in Σ_B containing three fields

- The e-field has $\lceil \log b \rceil + 1$ bits with code(c) in the beginning and the rest of the field filled with zeros.
- The s-field contains the original code for c using $\lceil \log \sigma \rceil$ bits.
- The r-field is the rank of the symbol c at the reference point, i.e., at the beginning of the block in $\lceil \log n \rceil$ bits.

The table V is ordered by the e-field. Given $LR[j]$, we find the entry in $V[i]$ such that $V[i].e \leq LR[j] < V[i+1].e$. We obtain the symbol $L[j]$ from $V[i].s$ and its rank at the reference point from $V[i].r$. The rank relative to the reference point is $LR[j] - V[i].e$. Thus $\mathrm{rank}_L(j) = V[i].r + (LR[j] - V[i].e)$.

To speed up the search in V, there is another table $U[0..2^q - 1]$, $0 \leq q \leq \lceil \log b \rceil + 1$. The entries of U represent the bitstrings of length q. The entry for a bitstring Q contains a pointer to the first position in V with a code beginning with Q. If a code is shorter than q, say $\hat{\ell}_B(c) < q$, all bitstrings beginning with code(c) point to $V[c]$. We need $\lceil \log \sigma \rceil$ bits for each pointer.

Using U we can short-cut to a good starting point for the search in V. The search itself can be done linearly, and we still obtain a linear-time algorithm as shown by the following theorem.

Theorem 3. *Setting $q = \lfloor \log \sigma \rfloor$ and $b = 2^k$ for $k = \lfloor \log(\sigma(\lceil \log n \rceil + 3\lceil \log \sigma \rceil)) \rfloor$, Algorithm `VLR-B` computes the inverse Burrows-Wheeler transform in $\mathcal{O}(n)$ time using at most*

$$n\left(2 + \log(\sigma) + \log\left(\lceil \log n \rceil + 3\lceil \log \sigma \rceil\right) + \frac{2\log\log n}{\log n}\right) + \mathcal{O}(\sigma \log n)$$

bits of space.

5.2 Algorithm `VLR-I`

Our final algorithm is a modification of Algorithm `LR-I` to use variable-length fields in the LR array. Each entry in the LR array is $h > \log \sigma$ bits. The L-fields use the length-limited rounded code $\hat{\ell}_L^h$, leaving $h - \hat{\ell}_L^h(c)$ bits for the R field. Thus the reference points are placed at every $b(c)$th occurrence for $b(c) = 2^{h - \hat{\ell}_L^h(c)}$. The decoding of the LR entries is done as in Algorithm `VLR-B`. Otherwise the algorithm works as `LR-I`. The properties are summarized in the following theorem.

Theorem 4. *Let* $h_{\min} = \lfloor 1 + \log \lceil 1 + \log n \rceil + \log \sigma \rfloor$*. When* $h = h_{\min}$*, the space requirement of Algorithm* `VLR-I` *is at most*

$$n(2 + \log \lceil 1 + \log n \rceil + \log \sigma) + \mathcal{O}(\sigma \log n)$$

bits. For $h > h_{\min}$*, the space requirement is at most*

$$n(h+1) + \mathcal{O}(\sigma \log n)$$

bits. The time complexity is $\mathcal{O}(n(\max(1, \log n - h))$*.*

6 Experimental Results

For testing we used the files listed in Table 2[1]. All tests were conducted on a 3.0 GHz Intel Xeon CPU with 4Gb main memory and 1024K L2 Cache. The machine had no other significant CPU tasks running. The operating system was Fedora Linux running kernel 2.6.9. The compiler was g++ (gcc version 4.1.1) executed with the -O3 option. The times given are the minima of three runs and were recorded with the standard C `getrusage` function. The memory requirements are sums of the sizes of all data structures as reported by the `sizeof` function.

Table 2. Data sets used for empirical tests. For each type of data (DNA, XML, ENGLISH, PROTEIN) a 100Mb file was used.

Data set name	σ	H_0	mean LCP
XML	97	5.23	44
DNA	16	1.98	31
ENGLISH	239	4.53	2,221
PROTEIN	27	4.20	166

Table 3. Algorithms and their parameter settings. Underlined parameter values indicate that the implementation is optimized for the byte or word alignment provided by those parameters values.

Alg.	Description
`LR`	Algorithm `LR` (Sect. 3) = `mergedTL` in [19] with 32-bit integers
`IF`	`indexF` in [19] with 32-bit integers
`LR-B`	$k \in 5 + \lfloor \log \sigma \rfloor, \ldots, \underline{17}, \underline{25}$
`VLR-B`	$k \in 10, \ldots, 24$
`LR-I`	$k + \lceil \log \sigma \rceil \in 14, \underline{16}, \underline{24}, \underline{32}$
`VLR-I`	$h \in 12, 14, \underline{16}, \underline{24}, \underline{32}$
`LL`	The simple small-space algorithm in [13] Blocksizes are powers of two in $[\max(32, \sigma) \ldots \min(2048, 40\sigma)]$
`WT`	A simple algorithm using wavelet tree for rank queries (see text)

[1] Available from `http://pizzachili.dcc.uchile.cl/`

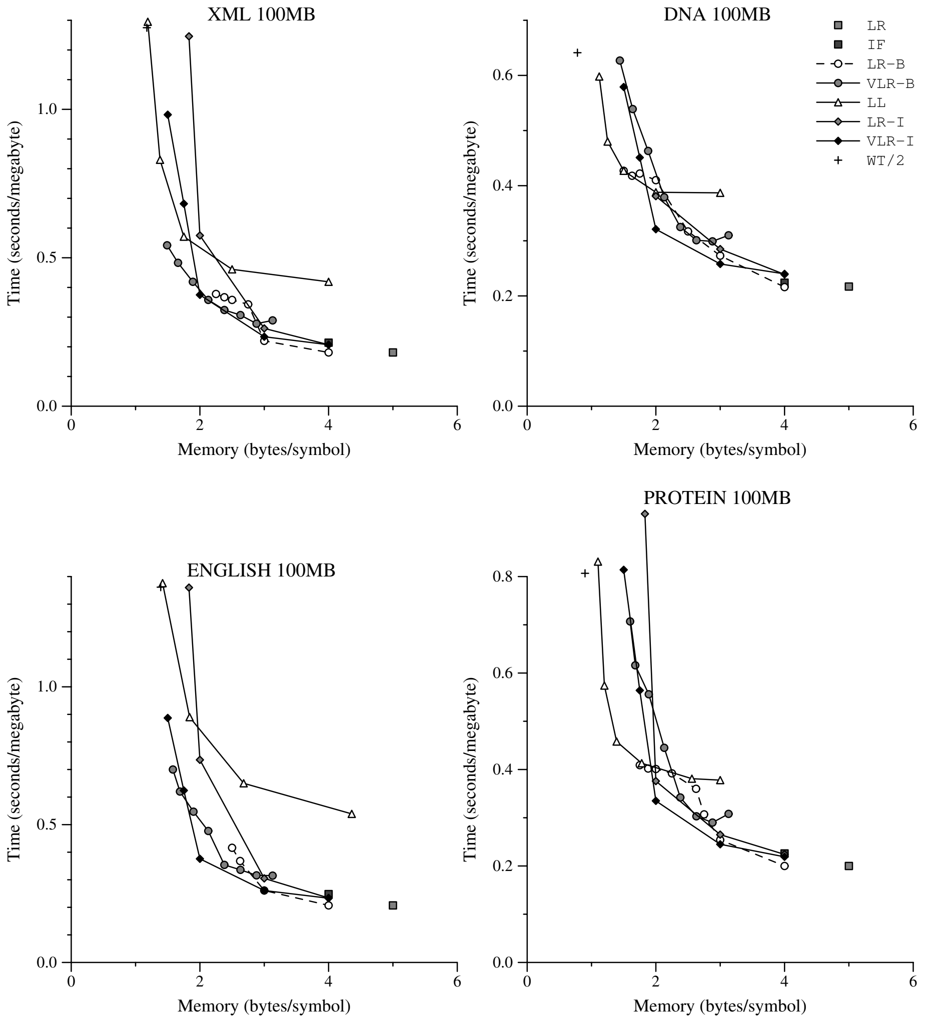


Fig. 3. Time-memory tradeoff for various inversion algorithms. For clarity, *the time shown for* `WT` *is half of the actual time*, which would be far outside the graph.

The focus of the experiments is on the four algorithms described in Sections 4 and 5, but for comparison we also implemented two large-space algorithms by Seward [19] and two simple small-space algorithms, one by Lauther and Lukovszki [13] and one based on the wavelet tree [7], which is a commonly used rank data structure with compressed text indexes [16]. We optimized the wavelet tree implementation for special rank queries and used the method of Vigna [21] (the fastest we know) for bitvector rank queries. For canonical prefix coding, we use the techniques of Turpin and Moffat [20][2] instead of the technique

[2] Originally downloaded from `http://ww2.cs.mu.oz.au/~alistair/mr_coder/`

of Sect. 5.1. In all medium- and small-space algorithms, we use $\sigma = |\Sigma_S|$ (see Table 2), which affects arrays of size σ, the height $\lceil \log \sigma \rceil$ of the wavelet tree, and the size $\lceil \log \sigma \rceil$ of the L-field in the LR array for Algorithms `LR-B` and `LR-I`. The algorithms and their parameter settings are summarized in Table 3.

The time and space requirements during BWT inversion are shown in Fig. 3. The times do not include reading the input or writing the output. The input and output are held in memory during the computation but are excluded from space requirements when the algorithm accesses them only sequentially.

All the medium-space algorithms display a fairly smooth space-time tradeoff curve, even the constant-time algorithms with no theoretical tradeoff. This is explained by cache effects. As the LR array (which always dominates the space) gets bigger, the other data structures get smaller and start to fit in the cache.

At the fast end of the space-time tradeoff, `LR-B` matches the speed of the fastest known algorithm, `LR`, in less memory. Note that this parameter setting ($k = 25$) was not implemented or even suggested by Lauther and Lukovszki [13]. The middle area is dominated by the algorithms `VLR-B` and `VLR-I` using variable-length encoding. They reduce the space by a factor of 2–3 compared with `LR` without slowing down by more than a factor of two. The results for the small end are mixed, and anyway should be considered incomplete, since there are many possibilities for improving the small-space algorithms.

7 Concluding Remarks

We have introduced three new algorithms for the BWT inversion and demonstrated, theoretically and experimentally, that they improve the state of the art, particularly in the middle area of the space-time tradeoff spectrum. We are continuing our research by focusing on the extremes of the spectrum.

At the small end, the two-level version of the small-space algorithm by Lauther and Lukovszki [13], and advanced techniques from compressed text indexes such as Huffman-shaped wavelet trees [8] and implicit compression boosting [14] appear promising approaches.

At the large end, we are experimenting with an algorithm that reduces cache misses by taking advantage of repetitions in the text. Another interesting avenue for future work is exploiting properties of modern processors such as parallelism and out-of-order execution [10].

References

1. Adjeroh, D., Bell, T., Mukherjee, A.: The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching. Springer, Heidelberg (2008)
2. Burrows, M., Wheeler, D.J.: A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California (1994)
3. Ferragina, P., Gagie, T., Manzini, G.: Lightweight data indexing and compression in external memory. In: López-Ortiz, A. (ed.) LATIN 2010. LNCS, vol. 6034, pp. 697–710. Springer, Heidelberg (2010)

4. Ferragina, P., Manzini, G.: On compressing the textual web. In: Proc. 3rd ACM International Conference on Web Search and Data Mining, pp. 391–400. ACM, New York (2010)
5. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Trans. Algorithms 3, Article 20 (2007)
6. Golynski, A., Munro, J.I., Rao, S.S.: Rank/select operations on large alphabets: a tool for text indexing. In: Proc. 17th ACM-SIAM Symposium on Discrete Algorithms, pp. 368–373. ACM, New York (2006)
7. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proc. 14th ACM-SIAM Symposium on Discrete Algorithms, pp. 841–850. SIAM, Philadelphia (2003)
8. Grossi, R., Gupta, A., Vitter, J.S.: When indexing equals compression: experiments with compressing suffix arrays and applications. In: Proc. 15th ACM-SIAM Symposium on Discrete Algorithms, pp. 636–645. SIAM, Philadelphia (2004)
9. Huffman, D.A.: A method for the construction of minimum-redundancy codes. Proceedings of the I.R.E. 40, 1098–1101 (1952)
10. Kärkkäinen, J., Rantala, T.: Engineering radix sort for strings. In: Amir, A., Turpin, A., Moffat, A. (eds.) SPIRE 2008. LNCS, vol. 5280, pp. 3–14. Springer, Heidelberg (2008)
11. Kärkkäinen, J.: Fast BWT in small space by blockwise suffix sorting. Theoretical Computer Science 387, 249–257 (2007)
12. Katona, G.O.H., Nemetz, T.O.H.: Huffman codes and self-information. IEEE Transactions on Information Theory IT-22, 337–340 (1976)
13. Lauther, U., Lukovszki, T.: Space efficient algorithms for the Burrows-Wheeler backtransformation. In: Brodal, G.S., Leonardi, S. (eds.) ESA 2005. LNCS, vol. 3669, pp. 293–304. Springer, Heidelberg (2005)
14. Mäkinen, V., Navarro, G.: Implicit compression boosting with applications to self-indexing. In: Ziviani, N., Baeza-Yates, R. (eds.) SPIRE 2007. LNCS, vol. 4726, pp. 229–241. Springer, Heidelberg (2007)
15. Manzini, G.: An analysis of the Burrows-Wheeler transform. Journal of the ACM 48, 407–430 (2001)
16. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys 39, Article 2 (2007)
17. Puglisi, S.J., Smyth, W.F., Turpin, A.: A taxonomy of suffix array construction algorithms. ACM Computing Surveys 39, 1–31 (2007)
18. Schwartz, E.S., Kallick, B.: Generating a canonical prefix encoding. Communications of the ACM 7, 166–169 (1964)
19. Seward, J.: Space-time tradeoffs in the inverse B-W transform. In: Storer, J., Cohn, M. (eds.) Proc. IEEE Data Compression Conference, pp. 439–448. IEEE Computer Society, Los Alamitos (2001)
20. Turpin, A., Moffat, A.: Housekeeping for prefix coding. IEEE Transactions on Communications 48, 622–628 (2000)
21. Vigna, S.: Broadword implementation of rank/select queries. In: McGeoch, C.C. (ed.) WEA 2008. LNCS, vol. 5038, pp. 154–168. Springer, Heidelberg (2008)