Fast BWT in Small Space by Blockwise Suffix Sorting

Juha Kärkkäinen

Department of Computer Science, P.O.Box 68
FI-00014 University of Helsinki, Finland

Abstract

We present a new space and time efficient algorithm for computing the Burrow–Wheeler transform (BWT). For any choice of a parameter $v \in [3, n^{2/3}]$, the computation of BWT for a text of length n takes $\mathcal{O}(n \log n + vn)$ worst-case time and $\mathcal{O}(n \log n + \sqrt{v}n)$ average-case time using $\mathcal{O}(n \log n/\sqrt{v})$ bits of space in addition to the text and the BWT. For example, if $v = \log^2 n$, the time is $\mathcal{O}(n \log^2 n)$ in the worst case and $\mathcal{O}(n \log n)$ on average with the additional space requirement of $\mathcal{O}(n)$ bits. The algorithm is alphabet-independent: it uses only character comparisons, and the complexities do not depend on the alphabet size unless v does. A practical implementation is 2–3 times slower than one of the fastest and most space efficient previous algorithms while needing only one third of the main memory. The algorithm is based on suffix arrays, but unlike any other algorithm, it can construct the suffix array a small block at a time without storing the rest of the suffix array anywhere.

1 Introduction

The Burrows–Wheeler transform (BWT) [3] of a text is a reversible transformation of the text that has a central role in some of the best data compression methods. The transform does not compress the text but the transformed text is easier to compress using simple and fast methods [24]. Computing the BWT typically needs significantly more time and space than the other steps of the compression [28].

A second important application of BWT is the construction of compressed full text indexes, which support fast substring searching on the text while taking

Email address: `juha.karkkainen@cs.helsinki.fi` (Juha Kärkkäinen).

little more space than the compressed text [21]. Some compressed indexes are directly based on BWT (for example [6,7]) while others can be efficiently constructed from the BWT [12]. Computing the BWT is the computational bottleneck in compressed index construction, too.

Usually, the BWT is computed from the *suffix array* (SA), the lexicographically sorted array of all the suffixes of the text. Computing BWT from SA is simple and fast, and a lot of effort has been spent in developing fast and space efficient algorithms for constructing the suffix array, i.e., for sorting the set of all suffixes [27]. However, all such algorithms need to store the suffix array, which can be much larger than the text or the BWT. The suffix array needs $\Omega(n \log n)$ bits of space while the text and the BWT can be encoded using $\mathcal{O}(n \log \sigma)$ bits for an alphabet size σ. In practice, the size of the suffix array is usually at least $4n$ bytes while the text and the BWT typically need only n bytes each, and sometimes even less, for example $2n$ bits each for a DNA sequence.

The large space requirement of the suffix array is a problem because it effectively restricts the size of texts that one can process on a given computer. In particular, the construction of a compressed index can require much more space than the compressed index itself. For example, a computer with 1 GB of main memory may be able to handle compressed indexes for texts larger than 1 GB, but cannot construct one for texts of size 200 MB.

Contribution In this paper, we get rid of the need to store the full suffix array. The idea relies on an observation about the way BWT is computed from the suffix array SA: to compute BWT[i] we only need SA[i], not the full suffix array. Thus, if we can construct the suffix array a small piece or block at a time, we can compute the corresponding block of BWT and then discard the SA block. No space is needed for the full suffix array.

We present the first algorithm that can compute the suffix array efficiently in smaller pieces. The basic idea is similar to sample sorting: choose a random set of splitters, sort them, and then distribute all the elements into the buckets delimited by the splitters. In our case, the distribution is done separately for each bucket, which is then sorted, used for computing a part of the BWT, and discarded before processing the next bucket.

A trivial implementation of this approach would require at least quadratic time in the worst case due to potentially expensive suffix comparisons. We use a combination of techniques to address this and other issues. These include a new linear-time algorithm for finding all suffixes that are lexicographically smaller than a given pattern, novel uses of the difference cover sample technique developed in [2] (see also [17]), and a deterministic method for choosing the splitters.

The algorithm is space and time efficient both in theory and in practice, and allows adjusting the space–time tradeoff. For any $v \in [3, n^{2/3}]$, the computation of BWT for a text of length n takes $\mathcal{O}(n \log n + vn)$ time using $\mathcal{O}(n \log n/\sqrt{v})$ bits of space in addition to the text and the BWT. For random texts, the expected running time is $\mathcal{O}(n \log n + \sqrt{v}n)$.

The algorithm is alphabet-independent: it uses only character comparisons, and the complexities do not depend on the alphabet size unless v does. For example, if we choose $v = \log^2 n$, the time is $\mathcal{O}(n \log^2 n)$ in the worst case and $\mathcal{O}(n \log n)$ on average with the additional space requirement of $\mathcal{O}(n)$ bits. On the other hand, for an alphabet of size σ, we can set $v = \log_\sigma^2 n$. Then the space complexity is $\mathcal{O}(n \log \sigma)$ *when including the text and the BWT* while the time complexity is $\mathcal{O}(n(\log n + \log_\sigma^2 n))$ in the worst case and $\mathcal{O}(n \log n)$ on average.

The algorithm can be implemented space efficiently without any compressed data structures and using a full machine word for each integer or pointer (except possibly the characters in the text and the BWT). We have an implementation that computes the BWT of a text file of n bytes using less than $2n$ bytes of main memory. This includes the text itself but not the BWT, which is written directly to disk. In comparison with one of the fastest and most space-efficient algorithms for suffix array construction [25], it is 2–3 times slower but uses only one third of the memory. We have also implemented another version for DNA sequences that needs less than one byte of memory per character but is much slower.

Related Work There are numerous algorithms for constructing suffix arrays [27]. The theoretically best ones work in linear time [17–19]. There are several so-called lightweight algorithms that need little space in addition to the text and the suffix array [28,25,2,13,23]. Some of these are also among the fastest algorithms in practice [27]. All of them, though, need to have at least the text and the full suffix array in memory.

When there is not enough main memory, one alternative is external memory algorithms. For suffix array construction, there are several external memory algorithms, see for example [17,16,4,15,5]. BWT can be easily computed from a suffix array that is stored on disk without a lot of main memory. When the size of the text significantly exceeds the main memory, external memory algorithms are the only alternative, and the recent work in [5] has made them a practical alternative in terms of running time, too. Their carefully engineered implementation achieves a speed of about 300–400 MB/hour on a PC with multiple fast disks. The speed of our implementation is slightly over 1 GB/hour for typical files.

There are also space-efficient algorithms for directly constructing compressed text indexes [12,20,10,26]. The best theoretical results are $\mathcal{O}(n \log\log\sigma)$ time using $\mathcal{O}(n\log\sigma)$ bits of space [12], and $\mathcal{O}(n)$ time using $\mathcal{O}(n \log n \log_\sigma^\alpha n)$ bits of space [26], where σ is the size of the alphabet and $\alpha = \log_3 2$. The only implementation we are aware of is in [11] (based on the algorithm in [20]). It works only for DNA, and its space requirement is between our general and DNA specific implementations ($10n$ vs. $14.5n$ and $7n$ bits). Estimated from their results, its speed is similar to the DNA specific implementation and much slower than the general implementation.

Organization of the Paper We start with basic definitions in Section 2 and an outline of the algorithm in Section 3. The details of various parts of the algorithm are described in Sections 4–7, and all is drawn together in Section 8. Finally, there are experimental results in Section 9 and some concluding remarks in Section 10.

2 Preliminaries

Let the *text* $T[0,n) = t_0t_1\cdots t_{n-1}$ be a string of length n over a general alphabet, i.e., we assume that the characters can be compared and copied in constant time but make no other assumptions on the alphabet. For $i \in [0,n]$, let S_i denote the *suffix* $T[i,n) = t_it_{i+1}\cdots t_{n-1}$. We also extend the notation to sets: for $C \subseteq [0,n]$, $S_C = \{S_i \mid i \in C\}$. The *suffix array* $\mathrm{SA}[0,n]$ of T is a permutation of $[0,n]$ satisfying $S_{\mathrm{SA}[0]} < S_{SA[1]} < \cdots < S_{\mathrm{SA}[n]}$.

The *Burrows–Wheeler transform* of T is the string $\mathrm{BWT}[0,n]$:

$$\mathrm{BWT}[i] = \begin{cases} T[\mathrm{SA}[i]-1] & \text{if } \mathrm{SA}[i] \neq 0 \\ \$ & \text{if } \mathrm{SA}[i] = 0 \end{cases} \tag{1}$$

Here \$ is a special character that is distinct from (and usually considered to be smaller than) all other characters. This definition is equivalent to the common description of BWT as the last column in a matrix, whose rows are the rotations (cyclic shifts, conjugates) of $T[0,n)\$$ in lexicographical order.

Example 1 *Let* $T[0,6) =$ *BANANA*. *Then* $\mathrm{SA}[0,6] = (6,5,3,1,0,4,2)$ *and* $\mathrm{BWT}[0,6] =$ *ANNB\$AA*.

3 Algorithm Outline

The usual way to compute the BWT is to first construct the suffix array SA and then use Equation 1 to compute the BWT. Our algorithm uses Equation 1, too, but the difference is that the SA is computed in smaller blocks. That is, for some $0 = i_0 < i_1 < i_2 < \ldots < i_r = n+1$, the algorithm first computes $\mathrm{SA}[0, i_1)$ and uses it to compute $\mathrm{BWT}[0, i_1)$, then it computes $\mathrm{SA}[i_1, i_2)$ and $\mathrm{BWT}[i_1, i_2)$, and so on. The division of SA into blocks is determined by using a sample of suffixes as splitters.

Here is the algorithm in more detail:

(1) Choose a random sample C of size $r-1$ from $[0, n)$.
(2) Sort the set S_C of splitter suffixes. Let $j_1, j_2, \ldots, j_{r-1}$ be the elements of C ordered such that $S_{j_1} < S_{j_2} < \cdots < S_{j_{r-1}}$. For convenience, let $S_{j_0} = S_n$ be the empty suffix, i.e., the smallest of the suffixes, and let S_{j_r} denote a string that is larger than any suffix.
(3) For each $k \in [0, r)$:
  (a) For each $j \in [0, n]$, determine whether $S_{j_k} \le S_j < S_{j_{k+1}}$ is true or not. If it is, store j in B_k
  (b) Compute $\mathrm{SA}[i_k, i_{k+1})$ by sorting the suffixes S_{B_k}.
  (c) Compute $\mathrm{BWT}[i_k, i_{k+1})$ from $\mathrm{SA}[i_k, i_{k+1})$ using Equation 1.

To make the algorithm concrete, we need to specify further details, in particular, how to sort suffixes in Steps 2 and 3b (Section 4), and how to compute the blocks in Step 3a (Section 5).

The space-time tradeoff of the algorithm is controlled by two parameters, v and $b_{\max}$. For the moment, we assume that the following conditions hold:

No long repetitions. The text has no repetitions longer than v, i.e., any two suffixes can be compared in time $\mathcal{O}(v)$.
No large blocks. No block B_k is larger than $b_{\max}$.

We will later see how to deal with the cases when the conditions are not satisfied (Sections 6 and 7).

4 Sorting Suffixes

Under the no-long-repetitions assumption, we can sort a set of suffixes efficiently using a simple string sorting. The difficulty of sorting strings depends on the lengths of common prefixes among the strings. We formalize this as follows:

Definition 2 *Let M be a set of m strings. The* distinguishing prefix *of a string S in M is the shortest prefix of S that is not a prefix of any other string in M. Let $\mathrm{DP}_M(S)$ denote the length of the distinguishing prefix, and let $\mathrm{DP}(M)$ denote the sum of the lengths in M, i.e., $\mathrm{DP}(M) = \sum_{S \in M} \mathrm{DP}_M(S)$. The* v-restricted *distinguishing prefix measures are defined as $\mathrm{DP}^v_M(S) = \min(v, \mathrm{DP}_M(S))$ and $\mathrm{DP}^v(M) = \sum_{S \in M} \mathrm{DP}^v_M(S)$.*

When M is a set of text suffixes, $\mathrm{DP}(M) = \mathcal{O}(vm)$ under the no-long-repetitions assumption, and $\mathrm{DP}^v(M) = \mathcal{O}(vm)$ even without the assumption. For random texts, $\mathrm{DP}(M) = \mathrm{DP}^v(M) = \mathcal{O}(m \log m)$ on average (see [14]).

We will use the multikey quicksort algorithm of Bentley and Sedgewick [1] for string sorting:

Lemma 3 ([1]) *Using the multikey quicksort algorithm, a set M of m strings can be sorted in $\mathcal{O}(m \log m + \mathrm{DP}(M))$ time using $\mathcal{O}(\log m)$ extra space.* [1]

Step 2 involves sorting a set of $r - 1$ suffixes and Step 3b r sets containing a total of n suffixes. None of the sets is larger than $r + b_{\max}$. This gives the following result.

Lemma 4 *The total time complexity of the sorting steps (Steps 2 and 3b) is $\mathcal{O}(n \log n + \mathrm{DP}(S_{[0,n]}))$. The space complexity (excluding the text) is $\mathcal{O}(r + b_{\max})$ (under the no-large-blocks assumption).*

The time is $\mathcal{O}(n \log n + vn)$ under the no-long-repetitions assumption and $\mathcal{O}(n \log n)$ on average for random texts. Note that $r + b_{\max} = \Omega(\sqrt{n})$.

5 Building the Blocks

Next, we consider Step 3a of the algorithm: finding all suffixes S_j that are between the block boundaries S_{j_k} and $S_{j_{k+1}}$.

The trivial method is to compare each suffix to the boundaries. Under the no-long-repetitions assumption, this can be done in $\mathcal{O}(vn)$ time. However, the procedure is repeated $\mathcal{O}(r)$ times leading to the total time of $\mathcal{O}(rvn)$. We will next present a faster method.

The method is based on a linear-time algorithm for finding all suffixes of a text T that are lexicographically smaller than a query string P. A pseudocode for the algorithm is given in Figure 1. The idea is simple: compute the length of the

[1] Throughout the paper, the space requirement is reported in machine words (or $\mathcal{O}(\log n)$ bit integers) unless another unit (bits) is explicitly specified.

```
SmallerSuffixes (T[0,n), P[0,m))    // Report all suffixes of T that are
                                    // lexicographically smaller than P
  Precompute lcp(P, P[i,m)) for all i ∈ [1,m]
  i := 0; j := −1; k := −1
  while i ≤ n do
      // T[i,n) is the suffix being compared to P
      // T[j,k) is a previously found prefix of P with maximal k
      // Precondition: k − j = lcp(T[j,n), P)
      if i > k then k := i; ℓ := 0
      else ℓ := lcp(P, P[i − j, m))
      if i + ℓ = k then
          while ℓ < m and k < n and P[ℓ] = T[k] do
              k := k + 1; ℓ = ℓ + 1
          j := i
      else if i + ℓ > k then
          ℓ := k − i
          j := i
      // Postcondition: ℓ = lcp(T[i,n), P)
      if ℓ ≠ m and (i + ℓ = n or T[i + ℓ] < P[ℓ]) then
          report that T[i,n) < P
      i := i + 1
```

Fig. 1. An algorithm for computing suffixes smaller than a given string.

longest common prefix (lcp) between P and each suffix of T (see Postcondition in the pseudocode), and compare the mismatching characters to determine the order. The lcp-computation takes advantage of previously computed lcp-values between P and an earlier suffix of T (see Precondition), and between P and its own suffix (a precomputed table).

If done naively, the precomputation needs $\mathcal{O}(m^2)$ time but this can be reduced to $\mathcal{O}(m)$ in several ways. Probably the simplest way is to call a modified version of SmallerSuffixes with $T = P[1, m)$. Clearly, the lcp-values ℓ computed in each round (see Postcondition) are then exactly what is needed, and one of the modifications is to store them instead of using them in order comparisons. The only other modification is to remove the precomputation. When a precomputed lcp-value is needed, it has already been computed in an earlier round of the algorithm: When processing $T[i, n) = P[i + 1, m)$, the precomputed value that may be needed is $lcp(P, P[i - j, m)) = lcp(P, T[i - j - 1, n))$ for some $j \geq 0$.

Lemma 5 *The* SmallerSuffixes *algorithm reports all suffixes of text $T[0, n)$ that are lexicographically smaller than the string $P[0, m)$ in $\mathcal{O}(n + m)$ time and in $\mathcal{O}(m)$ additional space.*

PROOF. The correctness of the algorithm can be easily checked by verifying that the pre- and postcondition remain satisfied during the whole execution. The algorithm runs clearly in linear time except for the precomputation and the inner while-loop. Each round of the inner while-loop increments k, and since its value never decreases, the while-loop is executed at most n times. The precomputation with the modified algorithm needs $\mathcal{O}(m)$ time. □

Now we can build a block in $\mathcal{O}(n)$ time by running SMALLERSUFFIXES (T, S_{j_k}) and SMALLERSUFFIXES (T, $S_{j_{k+1}}$) in parallel [2]. A suffix belongs to the block if and only if it is reported by the latter but not by the former. Under the no-long-repetitions assumption, it is sufficient to store only $\mathcal{O}(v)$ precomputed lcp's.

Lemma 6 *The total time complexity of the block building steps (Step 3a) is $\mathcal{O}(rn)$. The space complexity (excluding the text) is $\mathcal{O}(b_{\max} + v)$ (under the no-long-repetitions and no-large-blocks assumptions).*

6 Handling Long Repetitions

So far we have assumed, that there are no repetitions longer than v in the text. If this assumption does not hold, there are two problems:

(1) The worst case sorting time increases to $\Theta(n^2)$, because $\mathrm{DP}(S_{B_k})$ can be $\Theta(n|B_k|)$. This happens, for example, if the text is periodic.
(2) The precomputed lcp-table in the SMALLERSUFFIXES algorithm may need to grow up to size $\Theta(n)$.

We address both problems using a difference cover sample (DCS) [2,17].

A *difference cover sample* $\mathrm{DCS}_v(T)$ of a text T is a data structure introduced in [2] that enables efficient comparison of suffixes. The following lemma summarizes the key features of $\mathrm{DCS}_v(T)$.

Lemma 7 ([2]) *The difference cover sample $\mathrm{DCS}_v(T)$ of text T with period $v \in [3, n]$ can be constructed in $\mathcal{O}(|D| \log |D| + \mathrm{DP}^v(D))$ time and in $\mathcal{O}(v + |D|)$ space (excluding the text), where D is a set of $\Theta(n/\sqrt{v})$ suffixes. Let S_i and S_j be two suffixes of T with a common prefix of length $v-1$, i.e., $T[i, i+v-1) = T[j, j+v-1)$. Given $\mathrm{DCS}_v(T)$ the lexicographical order of S_i and S_j can be determined in constant time.*

[2] The parallel running requires a special implementation but does not affect the complexity.

The BWT-algorithm constructs $\mathrm{DCS}_v(T)$ in the beginning and then uses it as follows:

(1) Sort a set of suffixes using the multikey quicksort but using only prefixes of length $v-1$. Any group of k suffixes that remains unsorted because of a common prefix, can be sorted in $\mathcal{O}(k \log k)$ time using $\mathrm{DCS}_v(T)$. Thus, a set M of m suffixes can be sorted in $\mathcal{O}(m \log m + \mathrm{DP}^v(M))$ time, which is $\mathcal{O}(m \log m + vm)$ in the worst case.
(2) Modify the SmallerSuffixes algorithm so that it computes lcp's only up to the length $v-1$ and uses $\mathrm{DCS}_v(T)$ to determine the ordering when necessary. Now only $\mathcal{O}(v)$ space is needed for the precomputed lcp's.

With these techniques we can improve our earlier results.

Lemma 8 *Using* $\mathrm{DCS}_v(T)$ *(but not counting the time and space complexity of building and storing it), the time complexity of sorting in Lemma 4 is reduced to* $\mathcal{O}(n \log n + \mathrm{DP}^v(S_{[0,n]}))$*, and the no-long-repetitions assumption can be removed from Lemma 6.*

The time and space complexity of $\mathrm{DCS}_v(T)$ itself must be accounted, too.

Lemma 9 $\mathrm{DCS}_v(T)$ *can be constructed in* $\mathcal{O}((n/\sqrt{v}) \log(n/\sqrt{v}) + \mathrm{DP}^v(S_{[0,n]}))$ *time and* $\mathcal{O}(v + n/\sqrt{v})$ *space (excluding the text).*

7 Choosing Splitters

Finally, let us take a closer look at the how the splitter suffixes are chosen. No block should be larger than $b_{\max}$, a parameter determined by the available memory. At the same time, we want to keep the number r of blocks small because processing each block takes at least $\Omega(n)$ time (Lemma 6). We will next describe a deterministic procedure that makes all block sizes between $b_{\max}/2$ and $b_{\max}$ achieving the asymptotically optimal number $r = \mathcal{O}(n/b_{\max})$ of blocks.

The idea is to modify the block building Step 3a. When processing a block larger than $b_{\max}$, the modified Step 3a starts as before by scanning the text and collecting suffixes belonging to the block. When $b_{\max}$ suffixes have been collected, the scan is suspended, and the suffixes are sorted. The median is chosen as a new splitter, and the second half of the collected suffixes is discarded. The collecting scan then resumes (but with the new splitter as the upper boundary). The procedure is repeated whenever the number of collected suffixes reaches $b_{\max}$. When the collecting scan is complete, between $b_{\max}/2$ and $b_{\max}$ of the smallest suffixes of the original oversized block have been collected and are processed as a block in Steps 3b and 3c.

The algorithm can start with no splitters at all, i.e., with one block containing the whole suffix array. Splitters are created during the modified Step 3a as described above. All the splitters are kept, not just the last one of each scan. Since each splitter is at least $b_{\max}/2$ elements away from other splitters, the resulting blocks are never smaller than $b_{\max}/2$. Thus the total number of blocks and splitters at the end is $\mathcal{O}(n/b_{\max})$.

Lemma 10 *The total time complexity of the modified Step 3a is* $\mathcal{O}(n^2/b_{\max} + n\log n + \mathrm{DP}^v(S_{[0,n]}))$. *The space complexity (excluding the text and* $DCS_v(T)$*) is* $\mathcal{O}(b_{\max} + v + n/b_{\max})$.

PROOF. The space complexity is the same as before (Lemma 6) except that now up to $\mathcal{O}(n/b_{\max})$ splitters can be created and stored during a single scan. As before, each call to Step 3a spends $\mathcal{O}(n)$ time in scanning the text for a total time of $\mathcal{O}(rn) = \mathcal{O}(n^2/b_{\max})$. Resuming the scan after a suspension may cause an additional delay of $\mathcal{O}(v)$, but this does not increase the total time. The time complexity of the splitter computation is dominated by the sorting. Since there is $\mathcal{O}(n/b_{\max})$ sortings of $\mathcal{O}(b_{\max})$ suffixes, the total time complexity is no more than for sorting the blocks in Step 3b (Lemma 4). □

The modification of Step 3a did not change the asymptotic time complexity of the whole algorithm. In practice, though, the extra time spent in computing the splitters matters. It is better to start with some set of splitters and create new ones during Step 3a only when needed. In fact, our implementation initially chooses more random splitters than necessary, and then combines adjacent small blocks whenever the combined size does not exceed $b_{\max}$. This leads to at most $2n/b_{\max} + 1$ blocks (since the size of two adjacent blocks exceeds $b_{\max}$), very few of which are larger than $b_{\max}$.

To be able to combine small blocks we need to know the sizes of the blocks. We compute the sizes by using the string binary searching technique that Manber and Myers [22] developed for binary searching on suffix arrays. A generalization of the technique for other search data structures is described in [9]. The key result is the following:

Lemma 11 *Let* M *be a sorted set of* m *strings. The set* M *can be preprocessed in* $\mathcal{O}(\mathrm{DP}(M))$ *time and in* $\mathcal{O}(m)$ *space so that a binary search on* M *using a query string* S *can be accomplished in* $\mathcal{O}(\log m + \mathrm{DP}_M(S))$ *time.*

When the strings are suffixes of T and the difference cover sample $\mathrm{DCS}_v(T)$ is available, the preprocessing time can be reduced to $\mathcal{O}(\mathrm{DP}^v(M))$ and the binary search time to $\mathcal{O}(\log m + \mathrm{DP}^v_M(S))$. Then the time for computing the block sizes is at most $\mathcal{O}(n \log m + \mathrm{DP}^v(S_{[0,n]}))$ for m initial splitters, which does not increase the total time complexity of the algorithm. Thus, the number of initial splitters can be as high as the space complexity $\mathcal{O}(m)$ allows.

8 The Final Algorithm

We are now ready to summarize the properties of the BWT algorithm. The algorithm is controlled by two parameters: v, the difference cover sample period, and $b_{\max}$, the maximum block size. Summing up the space complexities from Lemmas 4, 9, and 10 (with $r = \mathcal{O}(n/b_{\max})$) gives $\mathcal{O}(v + n/\sqrt{v} + b_{\max} + n/b_{\max})$. By setting $b_{\max} = n/\sqrt{v}$ and making the restriction $v \le n^{2/3}$, this is simplified to $\mathcal{O}(n/\sqrt{v})$. Summing the time complexities similarly leads to the following result.

Theorem 12 *The BWT of a text of length n can be computed in $\mathcal{O}(n \log n + \sqrt{v}n + \mathrm{DP}^v(S_{[0,n]}))$ time and $\mathcal{O}(n/\sqrt{v})$ space (in addition to the text and the BWT), for any $v \in [3, n^{2/3}]$.*

Remark 13

(1) $\mathrm{DP}^v(S_{[0,n]})$ *is $\mathcal{O}(vn)$ in the worst case but for large v it can be much smaller. In particular, for random texts the expected value is $\mathcal{O}(n \log n)$ (see [14]).*
(2) The algorithm is alphabet independent. The encoding of the text and the BWT is free as long as the characters can be accessed, compared and copied in constant time. The complexities do not depend on the alphabet size (unless v does).
(3) For an alphabet of size σ, the text and the BWT can be encoded using $\mathcal{O}(n \log \sigma)$ bits. Measured in bits, the additional space complexity is $\mathcal{O}(n \log n/\sqrt{v})$ since everything can be encoded with $\mathcal{O}(\log n)$-bit integers. If $v = \mathcal{O}(\log_\sigma^2 n)$, the total space complexity is $\mathcal{O}(n \log \sigma)$ bits.

Interesting choices of the parameter v include $v = \log^2 n$ and $v = \log_\sigma^2 n$ leading to the following results.

Corollary 14 *The BWT of a text of length n can be computed in $\mathcal{O}(n \log^2 n)$ worst case time and in $\mathcal{O}(n \log n)$ average case time using $\mathcal{O}(n)$ bits of space in addition to the text and the BWT.*

Corollary 15 *The BWT of a text of length n over an alphabet of size σ can be computed in $\mathcal{O}(n(\log n + \log_\sigma^2 n))$ worst case time, in $\mathcal{O}(n \log n)$ average case time, and in $\mathcal{O}(n \log \sigma)$ bits of space.*

9 Experiments

The algorithm has been implemented as a program `bwt` that reads the text from a file and writes the BWT to another file. BWT is never stored in memory but is written directly to disk. There is also a second program `dnabwt` for the four letter DNA alphabet that stores the text using just two bits per character.

The implementation allows a choice of the parameter v. The following table gives the memory consumption in bits (not including the text and some minor data structures):

v	16	32	64	128	256	512	1024	2048
bits	$20n$	$14n$	$9n$	$6.5n$	$5n$	$3.5n$	$2.5n$	$1.8n$

For `bwt` we chose $v = 128$, which makes the total space consumption less than $2n$ bytes. Similarly, for `dnabwt` we chose $v = 256$ to get under n bytes.

For comparison, we used two programs that are based on fast and space-efficient algorithms for constructing the full suffix array. The first one (MF) is the deep-shallow algorithm of Manzini and Ferragina [25]. The implementation is available at `http://www.mfn.unipmn.it/~manzini/lightweight/index.html`. This is typically one of the fastest and most space-efficient algorithms available [27] but it does slow down on highly repetitive texts.

The second one (BK) is the algorithm of Burkhardt and Kärkkäinen [2]. The implementation is available at `http://www.cs.helsinki.fi/u/tpkarkka/publications/CPM03.tar.gz`. BK uses essentially the same DCS-based sorting code for the full suffix array that `bwt` uses for blocks (but with $v = 32$).

The programs are written in C++ and were compiled with `g++ -O3` except MF is written in C and was compiled with `gcc -O3`. The gcc/g++ version was 4.0.2. The tests were run on a PC with 2.6 GHz Intel Pentium 4 processor and 4 GB of main memory running Linux. The running times were measured using the unix `time` command. We report the CPU time, i.e., the sum of `user` and `sys` time. The memory consumption is the total size of the process at its maximum as reported by the unix `top` command.

We used six kinds of texts:

english Concatenation of English text files from the Gutenberg project provided by the Pizza&Chili Corpus `http://pizzachili.di.unipi.it/texts.html`. The test files are prefixes of the original.

random-64 Random text where each character is drawn independently and uniformly from an alphabet of size 64.

Table 1
Runtime (in seconds) and memory footprint (in GBytes) of BWT construction algorithms.

	text size = 256 MB				text size = 1 GB	
text	bwt	dnabwt	MF	BK	bwt	dnabwt
english	546	–	287	573	2746	–
random-64	511	–	241	605	2566	–
repeat-64	2994	–	43751	1372	13082	–
DNA	585	1974	223	589	–	–
random-DNA	574	1876	237	582	2898	8250
repeat-DNA	2986	12619	70125	1323	12555	52668
memory	0.46	0.23	1.3	1.5	1.8	0.90

repeat-64 A string of length 1024 and of type random-64 repeated until the required length.

DNA Concatenation of DNA sequences provided by the Pizza&Chili Corpus `http://pizzachili.di.unipi.it/texts.html`. The small fraction of characters other than `A`, `C`, `G` or `T` was removed. The size of the file is about 400 MB so there is no 1 GB DNA file in the experiments.

random-DNA Random text where each character is drawn independently and uniformly from the alphabet $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$.

repeat-DNA A string of length 1024 and of type random-DNA repeated until the required length.

The results are reported in Table 1. The 1 GB files are too large for MF and BK. The results show that `bwt` is quite competitive in speed. It is 2–3 times slower than MF for most texts but much faster on repetitive data while taking barely over one third of the space. The times for `bwt` and BK are very similar, because both spend most of their time in string sorting. The larger slow-down of `bwt` for repetitive data is probably due to the larger value of the parameter v.

`dnabwt` is significantly slower than `bwt` but still fast enough for overnight computation of BWT for multi-gigabyte texts. The slowness is probably primarily due to slow processing of characters packed in two bits, and significant speedup may be possible using techniques such as those described in [8].

10 Concluding Remarks

We have presented an algorithm that can compute the Burrows–Wheeler transform of a text using very little space, both in theory and in practice, and is still quite fast, again both in theory and in practice.

The underlying technique of blockwise suffix sorting is a versatile tool with many possibilities including:

- It can be used for fast semi-external suffix array construction, i.e., computing the suffix array onto disk quickly while keeping only the text completely in main memory.
- The blocks can be computed in sequential order making it possible to pipeline the BWT to another process such as the compression stage of a BWT-based compressor while storing no more than a small piece of the BWT at any time.
- Each block can be computed independently, allowing a parallel or distributed computation of the BWT or the suffix array.

References

[1] J. L. Bentley, R. Sedgewick, Fast algorithms for sorting and searching strings, in: Proc. 8th Annual Symposium on Discrete Algorithms, ACM, 1997.

[2] S. Burkhardt, J. Kärkkäinen, Fast lightweight suffix array construction and checking, in: Proc. 14th Annual Symposium on Combinatorial Pattern Matching, vol. 2676 of LNCS, Springer, 2003.

[3] M. Burrows, D. J. Wheeler, A block-sorting lossless data compression algorithm, Technical Report 124, SRC (digital, Palo Alto) (May 1994).

[4] A. Crauser, P. Ferragina, Theoretical and experimental study on the construction of suffix arrays in external memory, Algorithmica 32 (1) (2002) 1–35.

[5] R. Dementiev, J. Mehnert, J. Kärkkäinen, P. Sanders, Better external memory suffix array construction, ACM Journal of Experimental Algorithmics. To appear.

[6] P. Ferragina, G. Manzini, Indexing compressed text, J. ACM 52 (4) (2005) 552–581.

[7] P. Ferragina, G. Manzini, V. Mäkinen, G. Navarro, Compressed representations of sequences and full-text indexes, ACM Transactions on Algorithms. To appear.

[8] K. Fredriksson, Faster string matching with super–alphabets, in: Proc. 9th International Symposium on String Processing and Information Retrieval (SPIRE '02), LNCS 2476, Springer–Verlag, 2002.

[9] R. Grossi, G. F. Italiano, Efficient techniques for maintaining multidimensional keys in linked data structures (extended abstract), in: Proc. 26th International Conference on Automata, Languages and Programming, vol. 1644 of LNCS, Springer, 1999.

[10] W.-K. Hon, T.-W. Lam, K. Sadakane, W.-K. Sung, Constructing compressed suffix arrays with large alphabets, in: Proc. 14th International Symposium on Algorithms and Computation, vol. 2906 of LNCS, Springer, 2003.

[11] W.-K. Hon, T.-W. Lam, W.-K. Sung, W.-L. Tse, C.-K. Wong, S.-M. Yiu, Practical aspects of compressed suffix arrays and FM-index in searching DNA sequences, in: Proc. 6th Workshop on Algorithm Engineering and Experiments (ALENEX '04), 2004.

[12] W.-K. Hon, K. Sadakane, W.-K. Sung, Breaking a time-and-space barrier in constructing full-text indices, in: Proc. 44th Annual Symposium on Foundations of Computer Science, IEEE, 2003.

[13] H. Itoh, H. Tanaka, An efficient method for in memory construction of suffix arrays, in: Proc. 6th Symposium on String Processing and Information Retrieval, IEEE, 1999.

[14] P. Jacquet, W. Szpankowski, Autocorrelation on words and its applications - analysis of suffix trees by string-ruler approach, J. Comb. Theory, Ser. A 66 (2) (1994) 237–269.

[15] J. Kärkkäinen, S. S. Rao, Full-text indexes in external memory, in: U. Meyer, P. Sanders, J. Sibeyn (eds.), Algorithms for Memory Hierarchies (Advanced Lectures), vol. 2625 of LNCS, chap. 7, Springer, 2003, pp. 149–170.

[16] J. Kärkkäinen, P. Sanders, Simple linear work suffix array construction, in: Proc. 30th International Conference on Automata, Languages and Programming, vol. 2719 of LNCS, Springer, 2003.

[17] J. Kärkkäinen, P. Sanders, S. Burkhardt, Linear work suffix array construction, J. ACM. In press.

[18] D. K. Kim, J. S. Sim, H. Park, K. Park, Constructing suffix arrays in linear time, J. Discrete Algorithms 3 (2–4) (2005) 126–142.

[19] P. Ko, S. Aluru, Space efficient linear time construction of suffix arrays, J. Discrete Algorithms 3 (2–4) (2005) 143–156.

[20] T.-W. Lam, K. Sadakane, W.-K. Sung, S.-M. Yiu, A space and time efficient algorithm for constructing compressed suffix arrays, in: Proc. 8th Annual International Conference on Computing and Combinatorics, vol. 2387 of LNCS, Springer, 2002.

[21] V. Mäkinen, G. Navarro, Compressed full text indexes, ACM Computing Surveys. To appear.

[22] U. Manber, G. Myers, Suffix arrays: A new method for on-line string searches, SIAM J. Comput. 22 (5) (1993) 935–948.

[23] M. A. Maniscalco, S. J. Puglisi, Faster lightweight suffix array construction, in: Proc. 17th Australasian Workshop on Combinatorial Algorithms, 2006.

[24] G. Manzini, An analysis of the Burrows–Wheeler transform, J. ACM 48 (3) (2001) 407–430.

[25] G. Manzini, P. Ferragina, Engineering a lightweight suffix array construction algorithm, Algorithmica 40 (1) (2004) 33–50.

[26] J. C. Na, Linear-time construction of compressed suffix arrays using $o(n \log n)$-bit working space for large alphabets, in: Proc. 16th Annual Symposium on Combinatorial Pattern Matching, vol. 3537 of LNCS, Springer, 2005.

[27] S. Puglisi, W. Smyth, A. Turpin, A taxonomy of suffix array construction algorithms, ACM Computing Surveys. To appear.

[28] J. Seward, On the performance of BWT sorting algorithms, in: Proc. Data Compression Conference, IEEE, 2000.