Space-Efficient Construction of the Burrows-Wheeler Transform

Timo Beller[1], Maike Zwerger[1], Simon Gog[2], and Enno Ohlebusch[1]

[1] Institute of Theoretical Computer Science, University of Ulm, 89069 Ulm, Germany
{Timo.Beller,Maike.Zwerger,Enno.Ohlebusch}@uni-ulm.de
[2] Department of Computing and Information Systems, The University of Melbourne, VIC, 3010, Melbourne, Australia
Simon.Gog@unimelb.edu.au

Abstract. The Burrows-Wheeler transform (BWT), originally invented for data compression, is nowadays also the core of many self-indexes, which can be used to solve many problems in bioinformatics. However, the memory requirement during the construction of the BWT is often the bottleneck in applications in the bioinformatics domain.

In this paper, we present a linear-time semi-external algorithm whose memory requirement is only about one byte per input symbol. Our experiments show that this algorithm provides a new time-memory trade-off between external and in-memory construction algorithms.

1 Introduction

In 1994 Burrows and Wheeler [5] presented the Burrows-Wheeler transform (BWT). This reversible transformation produces a permutation of the input string, in which symbols tend to occur in clusters. Because of this clustering, in virtually all cases the BWT compresses much easier than the original string, and Burrows and Wheeler suggested their transformation as a preprocessing step in data compression. Data compression has become a major application for the Burrows-Wheeler transform, e.g. it is the basis of the bzip2 algorithm.

Interestingly, the BWT has become the core of self-indexes [7, 14] which have applications in bioinformatics and information retrieval. In the data compression scenario it is possible to split a large input and construct the BWT for small blocks, since decoding and encoding are done sequentially. However, this is not possible for self-indexes because the optimal search routine requires the BWT of the whole text. In this case, both the runtime and the memory requirement of the construction of the BWT are critical. In the past, there were impressive improvements in algorithms constructing the suffix array. Theoretical worst-case time complexity, practical runtime and memory footprint have been improved. As the BWT can easily (fast and space efficiently) be obtained from the suffix array, the construction of the BWT profited indirectly from these improvements. However, $n \log n$ bits seems to be a lower memory bound for fast suffix array construction. On the other hand, this memory bound seems not to be valid for BWT construction, as there are algorithms that directly construct the BWT

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-3-319-02432-5_33

O. Kurland, M. Lewenstein, and E. Porat (Eds.): SPIRE 2013, LNCS 8214, pp. 5–16, 2013.
© Springer-Verlag Berlin Heidelberg 2013

using less that $n \log n$ bits, but still depend on the input size. External algorithms take only a given amount of memory, which is independent of the input size and normally user defined. In the past, external algorithms were presented that compute the suffix array or the BWT, e.g. [4, 6, 8, 11]. While this approach finally solves the memory problem (the algorithm needs only as much memory as available), it is commonly known that external algorithms have a significant slow down.

Thus, external algorithms are only used when the input does not fit in RAM. Currently, this happens already for quite small files: In our experiments on a machine equipped with 8 GB RAM, the suffix array construction algorithm divsufsort[1] already suffered from swapping effects for inputs larger than 1.5 GB. A direct computation of the BWT may allow bigger inputs: An implementation of Sadakane[2] can construct the BWT for inputs up to 3 GB on that machine. But this implementation is limited to inputs of 4 GB (even if much more RAM would be available). We show in this paper that the space requirements can further be improved: We present a new semi-external algorithm to compute the BWT. Semi-external algorithms are in between internal algorithms and external algorithms. To be more precise, *semi-external algorithms* are—at least in this paper—algorithms that are allowed to use an input dependent amount of memory (like internal algorithms), but also use disk memory (like external algorithms). In practice, semi-external algorithms store all data on disk that is accessed sequentially, while data with random access pattern is kept in main memory. Our implementation has no limitation on the input size and can construct the BWT of a 6 GB file with only 8 GB of RAM. In contrast, internal suffix array construction algorithms would need over 54 GB of RAM (or 31 GB if bit compression would be used) to compute the suffix array of a 6 GB file, because they must keep at least the input and the output in memory.

2 Preliminaries

Let Σ be an ordered alphabet of size σ whose smallest element is the so-called sentinel character \$. In the following, S is a string of length n on Σ having the sentinel character at the end (and nowhere else). For $1 \leq i \leq n$, $S[i]$ denotes the *character at position* i in S. For $i \leq j$, $S[i..j]$ denotes the *substring* of S starting with the character at position i and ending with the character at position j. Furthermore, S_i denotes the i-th suffix $S[i..n]$ of S. The *suffix array* SA of the string S is an array of integers in the range 1 to n specifying the lexicographic ordering of the n suffixes of S, that is, it satisfies $S_{\mathsf{SA}[1]} < S_{\mathsf{SA}[2]} < \cdots < S_{\mathsf{SA}[n]}$.

The suffix array SA is often enhanced with the so-called LCP-array containing the lengths of longest common prefixes between consecutive suffixes in SA. Formally, the LCP-array is an array so that $\mathsf{LCP}[1] = -1 = \mathsf{LCP}[n+1]$ and $\mathsf{LCP}[i] = |\mathsf{lcp}(S_{\mathsf{SA}[i-1]}, S_{\mathsf{SA}[i]})|$ for $2 \leq i \leq n$, where $\mathsf{lcp}(u, v)$ denotes the longest common prefix between two strings u and v. The Burrows-Wheeler

[1] `http://code.google.com/p/libdivsufsort/`
[2] `http://researchmap.jp/muuw41s7s-1587/#_1587`

transform [5] converts a string S into the permuted string $\mathsf{BWT}[1..n]$ defined by $\mathsf{BWT}[i] = S[\mathsf{SA}[i] - 1]$ for all i with $\mathsf{SA}[i] \neq 1$ and $\mathsf{BWT}[i] = \$$ otherwise.

As in [15–18], we distinguish between S-type, L-type and LMS-type suffixes: S_i is called S-type if $i = n$ or $S_i < S_{i+1}$. Analogously, we call S_i an L-type suffix if $S_i > S_{i+1}$. An S-type suffix is (also) an LMS-type suffix provided that S_{i-1} is an L-type suffix. Note that S_1 is never an LMS-type suffix, but S_n is always an LMS-type suffix. We call $S[i..j]$ an LMS-substring if S_i and S_j are LMS-type suffixes and for every $k, i < k < j$, S_k is not of type LMS. Additionally, $\$S[1..k]$ is also an LMS-substring, where S_k is the first LMS-type suffix in S.

A rank query $rank_b(B, i)$ on a bit-vector B counts the number of occurrences of bit b in $B[1..i]$. Similarly a select query $select_b(B, i)$ on a bit-vector B returns the position of the i-th occurrence of bit b in B. By pre-processing B one can answer both queries in constant time [10].

3 Related Work

There are many suffix array construction algorithms with different time and space complexities. We refer to the overview article [19] for details. It is widely agreed that in practice Yuta Mori's divsufsort is one of the fastest algorithms to compute the SA. For $n < 2^{31}$, it uses $5n$ bytes and $9n$ bytes otherwise.

In contrast to suffix array construction algorithms, the direct computation of the BWT has received much less attention. In [13], it is shown how to compute the BWT for biological data in $\mathcal{O}(n \log n)$ time. In [18], a linear-time algorithm for computing the Burrows-Wheeler transform was presented. This algorithm uses $\mathcal{O}(n \log \sigma \log \log_\sigma n)$ working space.

External algorithms for computing the BWT are described in [8, 11]. They construct the BWT by splitting the input into blocks of fixed length and computing the BWTs of these blocks. Afterwards, one has to merge the BWTs of the blocks to obtain the BWT of the input. In contrast, [1] presented an external algorithm for computing the BWT of a collection of short strings. However, this task is conceptually easier and can not easily be adapted to the case of arbitrary strings.

Algorithms also exist for computing the suffix array in external memory, see e.g. [6]. Very recently, [4] presented an external algorithm, not only for suffix array construction, but also for the computation of the LCP array. This algorithm is also based on the induced sorting algorithm and it is reported to be faster than the previous external suffix array construction algorithms.

4 The Induced Sorting Algorithm

As our new algorithm is based on the induced sorting algorithm, we briefly revisit this elegant algorithm here. For more details and correctness, we refer to [15].

The suffix array can be divided into σ buckets, where all suffixes in a bucket start with the same character. Within a bucket, L-type suffixes are smaller than

S-type suffixes. So every bucket can further be divided in two ranges, an L-type range and an S-type range. In the following, assume that A is an array of size n, which is divided into buckets and ranges as described before. Fig. 1 illustrates the induced sorting algorithm by an example.

Step 1. Input S is scanned from right to left in order to detect all indexes j in S at which an LMS-type suffix starts. All these indexes are written consecutively to the rightmost free position in the S-type range of the corresponding $S[j]$ bucket in A.

Step 2. Array A is scanned from left to right. Assume we are at position i in A. If $A[i]$ is empty, we go to the next position $i+1$. Otherwise let $j = A[i]$; we check if $S[j-1] \geq S[j]$. If so, we delete $A[i]$ and write $j-1$ to the leftmost free position in the L-type range of the corresponding $S[j-1]$ bucket.

Step 3. After we finished the left-to-right scan, we scan A from right to left. Assume again that we are at position i of the array A. If $A[i]$ is empty, we go to the next position $i-1$. Otherwise for $j = A[i]$ we check if $S[j-1] \leq S[j]$. If so, we delete $A[i]$ and write $j-1$ to the rightmost free position in the S-type range of the corresponding $S[j-1]$ bucket.

Step 4. The two scans in steps 2 and 3 sort the LMS-substrings (but not the LMS-type suffixes). In this step, the induced sorting algorithm replaces each LMS-substring by its lexicographical name and concatenates them in text order. First, all LMS-type positions are moved to the second half of A. This is possible because there are at most $\frac{n}{2}$ LMS-substrings. Then the second half of A is scanned from left to right. Assume that we are at a non-empty position i in A and $j = A[i]$. We compare the LMS-substring starting at $S[j]$ with the LMS-substring starting at $S[A[i-1]]$. If the substrings are identical, j gets the same lexicographical name, otherwise, j gets the next larger lexicographical name. The name is moved to $A[\lfloor \frac{j}{2} \rfloor]$. Finally, all names are placed into the second half of A, overwriting the LMS-type positions there. We now interpret these values as a new string S'. Note that S' usually has a different alphabet size than S.

Step 5. The order of the LMS-type suffixes is now obtained from the suffix array of S'. If every symbol in S' is unique, then one can easily create the suffix array. Otherwise, the induced sorting algorithm recursively computes the suffix array of the string S'. In either case, the suffix array of S' is written to the first half of A.

Step 6. The inverse suffix array of S' is now calculated and stored in the second half of A (overwriting S'). Then, a right-to-left scan of S is executed to find all LMS-type positions (again). Each LMS-type position is written (with the help of the inverse suffix array of S') in the correct lexicographical order to the first half of A. Afterwards the induced sorting algorithm removes the inverse suffix array of S' and places the LMS-type positions stably into the S-type ranges of their corresponding buckets in A.

Step 7. Array A is scanned from left to right and the indexes are moved as described in step 2 . However, this time indexes placed into an L-type range are not erased.

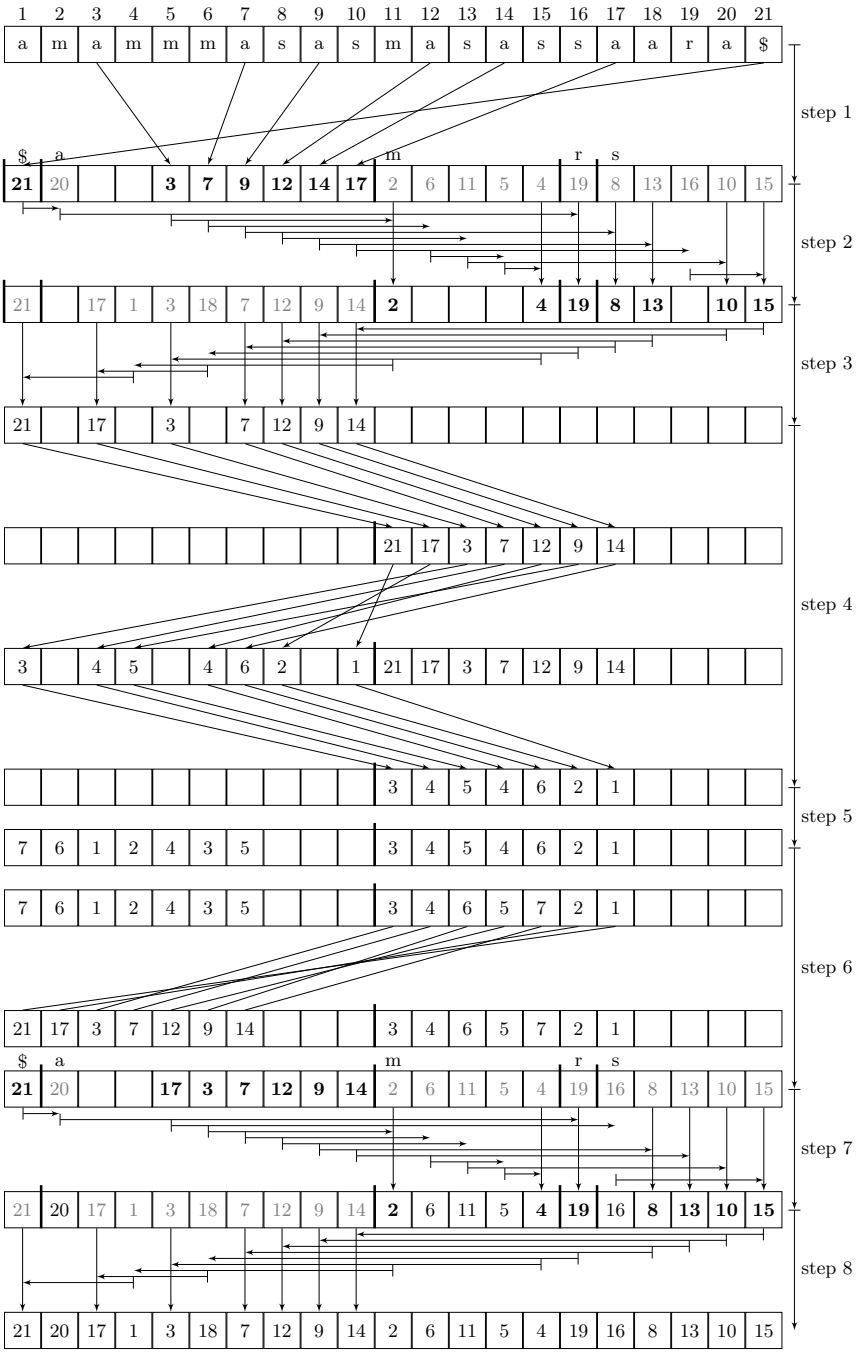

Fig. 1. Steps of the induced sorting algorithm: It computes the suffix array of the input string *amammmasasmasassaara*\$. The movements of the indexes are illustrated with arrows, temporary results are shown in gray.

Step 8. Array A is scanned from right to left and the indexes are moved as described in step 3, but again indexes placed into an S-type range are not erased. After this step, A contains the suffix array of S.

The induced sorting algorithm, as described in this section, uses the input string S, the array A and σ pointers to the rightmost (leftmost) free position of the S-type (L-type) buckets. The space requirement for the pointers are only relevant in the recursive calls of the induced sorting algorithm because in the recursive calls σ is no longer negligible small. Surprisingly, [17] showed that one can get rid of these pointers in the recursive levels. The resulting algorithm is optimal for an internal algorithm, as it keeps only input, output and a constant number of variables (for constant alphabet size) in main memory. In order to reduce the space further, one has to allow the use of disk. Unfortunately, random accesses on disk are very slow and most of the accesses done by the induced sorting algorithm are random accesses to both the input string S and the array A. We show in Section 5 how to modify the induced sorting algorithm to get rid of the A array, while using only sequential accesses to disk.

5 Semi-external Construction of the Burrows-Wheeler Transform

In this section n_{S}, n_{L}, and n_{LMS} denote the number of S-type, L-type and LMS-type suffixes of S, respectively. The following steps correspond to the steps of the induced sorting algorithm, but this time the BWT of S instead of the suffix array is calculated. Fig. 2 illustrates all steps of the new algorithm.

Step 1. In this step S can reside on disk, as it is read sequentially. Furthermore, only n_{LMS} indexes are written into A. We can save space by storing the indexes (without gaps) in an array $A_{\mathsf{LMS},left}$ of size n_{LMS}, which is written to disk and will be read sequentially in step 2. The next two steps require random access to the input string S, therefore S is loaded from disk.

Step 2. Only the L-type positions of A are accessed here. Thus, we use A_{L} of size n_{L} instead of A. However, at each end of a bucket, we must read (sequentially) from $A_{\mathsf{LMS},left}$ to place all LMS-type suffixes belonging to the next bucket. Then we continue by scanning the next bucket of A_{L}. Additionally, if an index would not be moved by the original induced sorting algorithm, we write it sequentially into the array $A_{\mathsf{LMS},right}$. So after performing step 2, $A_{\mathsf{LMS},right}$ contains all indexes, while $A_{\mathsf{LMS},left}$ and A_{L} are empty.

Step 3. Similar to step 2, only the S-type positions of A are accessed now. So instead of A, we now use the array A_{S} of size n_{S}. As before, between two buckets one must read (sequentially) from $A_{\mathsf{LMS},right}$. Again, if an index would not be moved by the original induced sorting algorithm, we write it sequentially into $A_{\mathsf{LMS},left}$. At the end of step 3, $A_{\mathsf{LMS},left}$ contains the LMS-type indexes sorted according to their corresponding LMS-substrings.

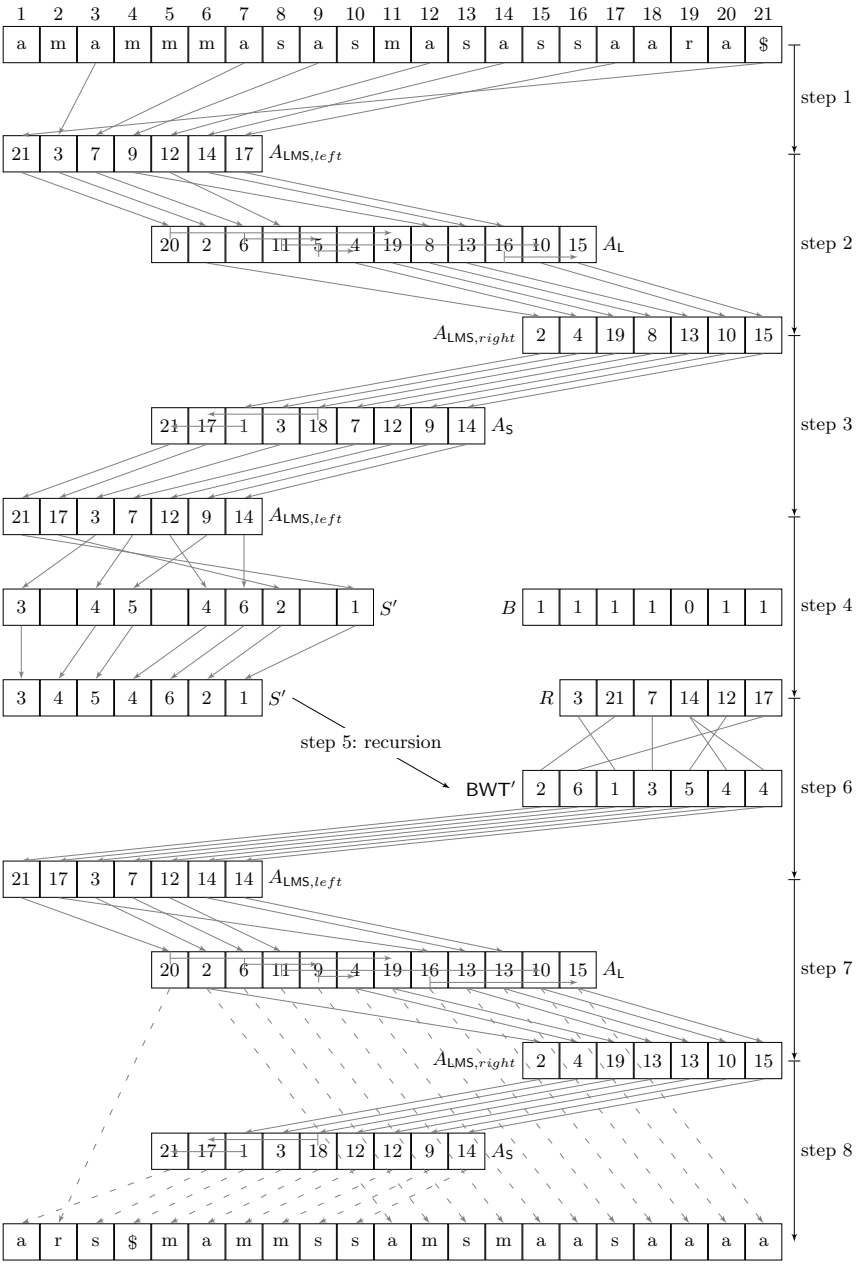

Fig. 2. Steps of the semi-external construction of the Burrows-Wheeler Transform for the input string *amammmasasmasassaara*$

Step 4. For the creation of the renamed string S' a bit-vector B of size n_{LMS} is computed, which indicates whether two consecutive entries in $A_{\mathsf{LMS},left}$ correspond to identical LMS-substrings or not. During this computation an array R is constructed, which contains the mapping of the lexicographical names for the LMS-substrings to their (end)positions in S. For identical LMS-substrings only one position has to be stored. Then $A_{\mathsf{LMS},left}$ is read sequentially again. The corresponding lexicographical name of $j = A_{\mathsf{LMS},left}[i]$ (determined with the help of B) is written to $S'[\lfloor \frac{j}{2} \rfloor]$. Afterwards, the gaps in S' are removed (preserving the order of the entries), and $A_{\mathsf{LMS},left}$ is deleted.

Step 5. The Burrows-Wheeler transform of S' (called BWT') is calculated directly if the characters of S' are pairwise distinct, and recursively otherwise.

Step 6. The array R contains the mapping of the lexicographical names to the corresponding positions in S. So $A_{\mathsf{LMS},left}$ can be filled sequentially based on the equation $A_{\mathsf{LMS},left}[i] = R[\mathsf{BWT}'[i]]$.

Step 7. Array A_{L} is created again and scanned as in step 2 from left to right. Between two buckets, one must read (sequentially) from $A_{\mathsf{LMS},left}$. Again, if an index would not be moved by the original induced sorting algorithm, we write it sequentially to the array $A_{\mathsf{LMS},right}$. Additionally, we begin to produce the BWT of S. To be precise, every time an index i is placed into A_{L}, the character $S[i-1]$ is written to the correct position of the BWT. So after performing this step, every entry in the BWT that corresponds to an L-type suffix is set correctly, while there are gaps corresponding to S-type suffixes. Furthermore, $A_{\mathsf{LMS},right}$ contains all indexes, while $A_{\mathsf{LMS},left}$ and A_{L} are empty.

Step 8. Array A_{S} is created again and scanned as described in step 3 from right to left (but we do not need $A_{\mathsf{LMS},left}$). During this computation, the gaps of the BWT of S are filled. To be precise, each time an index i is placed into A_{S}, the character $S[i-1]$ is written to the correct position of the BWT. After this step, the BWT of S is completely calculated.

The correctness and linear runtime of this algorithm follows directly from the correctness and runtime of the induced sorting algorithm.

Table 1. Access pattern to the data structures during the different steps of the algorithm. In step 4, random access is needed first to S and then to S'.

step	random access	sequential access
1	$A_{\mathsf{LMS},left}$	S
2	S, A_{L}	$A_{\mathsf{LMS},left}$, $A_{\mathsf{LMS},right}$
3	S, A_{S}	$A_{\mathsf{LMS},left}$, $A_{\mathsf{LMS},right}$
4	S/S', B	$A_{\mathsf{LMS},left}$, R
5	S'	
6	R	BWT'
7	S, A_{L}	BWT', $A_{\mathsf{LMS},right}$, BWT
8	S, A_{S}	$A_{\mathsf{LMS},right}$, BWT

Table 1 summarizes which data structures are needed in memory, and which can reside on disk because only sequential access is needed. The memory peak is now in steps 2, 3, 7, and 8 because in these steps the text and a relatively large array (A_{L} or A_{S}) is accessed randomly. However, one can reduce the space for A_{L} and A_{S} because of the special access pattern: These arrays are read sequentially, while the write access occurs only at positions that were not already read. We describe now how to replace A_{L} of size n_{L} with A'_{L} of size $k < n_{\mathsf{L}}$. The idea is to split A_{L} in $\lceil \frac{n_{\mathsf{L}}}{k} \rceil$ parts of size k. A'_{L} covers only one part of A_{L}, while for all other parts arrays P_i are created. Assume that we have to write value v to position p, where p does not belong to the part of A_{L} that corresponds to A'_{L}. In this case, both values v and p are written to the corresponding array P_i. When our reading position reaches the end of A'_{L}, we read the array P_i that covers the next part and write the values with an appropriate offset to A'_{L}. Because read and write accesses on P_i are sequentially, it can reside on disk. We deal analogously with A_{S}.

6 Practical Optimization for Very Small Alphabets

The BWT has important applications in bioinformatics. In this field, the alphabet size is very small, e.g. 4 or 5 in case of DNA data. Thus, it is worthwhile to optimize the algorithm for inputs with very small alphabet.

Let ℓ be a fixed natural number. We call an LMS-substring s *short* if $|s| \leq \ell$ and *long* otherwise. For a short LMS-substring s, we define its number as:

$$number(s) = \sum_{i=1}^{|s|} ord(s[i]) \cdot \sigma^{\ell-i} + \sum_{i=|s|+1}^{\ell} (\sigma - 1) \cdot \sigma^{\ell-i}$$

where $ord(a) = |\{a' \in \Sigma : a' < a\}|$ for every $a \in \Sigma$. For two short LMS-substrings s_1 and s_2, $number(s_1) < number(s_2)$ if and only if s_1 has a smaller lexicographical name than s_2. Now, we can obtain S' by another approach: We create a bit-vector B_{short} of size σ^ℓ to mark the numbers of all short LMS-substrings. By scanning S once from right to left, all LMS-substrings can be found. If the current LMS-substring s is short, we calculate its number $i = number(s)$ and set $B_{short}[i] = 1$. Otherwise, we store its starting position together with its position in S' (which is the number of LMS-type suffixes before the current one in S). Afterwards we (naively) sort the long LMS-substrings according to their lexicographical order. Then we create another bit-vector B_{LMS}, where $B_{\mathsf{LMS}}[i] = 0$ if the i-th smallest LMS-substring is longer than ℓ and $B_{\mathsf{LMS}}[i] = 1$ otherwise. B_{LMS} can be calculated by scanning V and B_{short} in parallel. During this scan, the lexicographical names of the long LMS-substrings can be written to S'. At last, the lexicographical names of the short LMS-substrings are inserted into S': S is scanned again from right to left. When we find a short LMS-substring s, we calculate the number of short LMS-substrings that are smaller than s by $r = rank_1(B_{short}, number(s))$, and obtain the lexicographical name with $select_1(B_{\mathsf{LMS}}, r)$.

Sorting the long LMS-substrings can be done in $\mathcal{O}(n \log n)$ using multikey quicksort [3], so this optimization does not have a linear runtime. However, it is in practice faster than the linear method described in Section 5 because we can exploit that LMS-substrings are usually very short and thus (for $\ell = 8$) there are not so many long LMS-substrings. Unfortunately, this optimization does not work in the recursive steps because in the recursive calls the alphabet size is not small enough.

7 Experimental Results

We implemented the algorithm using Simon Gog's [9] library sdsl (`http://github.com/simongog/sdsl`). In particular, we used bit-compressed integers, which causes a slow down but avoids problems with inputs larger than 2^{32}.

The experiments were conducted on a machine with a Intel(R) Core i5-3570 processor (3.40 GHz; L1 Cache=256 KB, L2 Cache=1 MB, and L3 Cache=6 MB) and 8 GB RAM. The operating system was Ubuntu 12.04.2 LTS. All programs were compiled with g++ (version 4.6.3) using the provided makefile.

As test files we used DNA data of different size because this is the main application. We concatenated the genomes[3] from Human (hg19), Mouse (mm10) and Gorilla (gorGor3) and deleted all characters other than A, C, G, T and N. Then we took prefixes of size 1 GB (genome1), 3 GB (genome2), and 6 GB (genome3).

For a comparison with internal memory algorithms, we used Yuta Mori's divsufsort. It needs $5n$ bytes for inputs smaller than 2^{31} and $9n$ bytes otherwise. Additionally, an implementation from Sadakane (called dbwt in the following) was used. This implementation is based on [18] (but has some simplifications compared to the algorithm described in [18]) and usually uses less than $2.5n$ bytes. Unfortunately, dbwt is limited to inputs smaller than 2^{32} bytes and it is unclear if it can be modified so that it can handle bigger inputs without increasing the memory footprint or runtime.

For a comparison with external memory algorithms, we took the following three implementations: bwtdisk 0.9.0 from Giovanni Manzini based on the algorithms described in [8]. This program can handle compressed inputs and can produce compressed outputs, but we did not make use of that option. LS from Kunihiko Sadakane. It is an external memory variant of the Larsson-Sadakane algorithm presented in [12]. This implementation can use multiple processors and we tested it with all 4 available processors. eSAIS 0.5.2 [4] does not compute the BWT but the suffix array and (optional) the LCP array. We turned the LCP construction off to construct only the suffix array.

For a fair comparison with our new algorithm, we allowed each external implementation to take n bytes of RAM. However, LS can only take a power of 2, so we allowed it the usage of 2^{32} byte for the 3 GB input and 2^{33} byte for the 6 GB input.

[3] Downloaded from `http://genome.ucsc.edu`

Table 2. Each column shows the runtime in seconds and in parentheses the maximum memory usage in byte per input character. The files genome2 and genome3 were too large for divsufsort on the machine equipped with 8 GB of RAM. Because dbwt is limited to files smaller than 4 GB, genome3 (6 GB) could not be calculated with dbwt.

algorithm	genome1	genome2	genome3
divsufsort	204 (5.00)	-	-
dbwt	229 (1.95)	705 (2.00)	-
this paper	412 (1.00)	1 475 (1.00)	3 387 (1.00)
bwtdisk	1 751 (1.05)	5 693 (1.05)	12 342 (1.05)
eSAIS	4 042 (1.08)	14 225 (1.02)	28 324 (1.06)
LS	9 382 (0.82)	34 200 (1.07)	94 728 (1.07)

Table 2 shows the experimental results. On the small genome1 file, divsufsort is the fastest algorithm, followed by dbwt. Compared to dbwt our algorithm is about 2 times slower, but uses only about half of the space. The same is true for the genome2 file. The 6 GB file (genome3) was far too big for the internal memory algorithms divsufsort and dbwt on the machine with 8 GB of RAM. The suffix array construction algorithm divsufsort would require about 54 GB of RAM and dbwt is limited to inputs of at most 4 GB. That is why our algorithm is important. Of course, one can always resort to an external algorithm if internal memory algorithms need too much RAM. But as our experiments show, our algorithm is the faster alternative (provided that there is enough RAM for it): The implementation described in this paper is over 3 times faster than the fastest external algorithm bwtdisk. Compared to eSAIS it is nearly one order of magnitude faster. However, one should keep in mind that the comparison with eSAIS is not fair because eSAIS constructs the suffix array and not the BWT.

8 Conclusion and Future Work

In this paper we presented a new method to construct the BWT space efficiently. It is a semi-external algorithm, which is based on the induced sorting algorithm. The implementation is not limited to inputs smaller than 4 GB and experiments show that it needs only about n bytes to compute the BWT of a length n DNA sequence. Thus, it needs about half of the space dbwt uses and over 5 times less space than suffix array construction algorithms. Furthermore, it is faster than external algorithms when they are allowed to use n bytes of memory. So only in cases when the input does not fit in RAM, external algorithms must be used. In all other cases, one can construct the BWT with a non-external algorithm. Note that n bytes are enough to compute the LCP-array from the BWT as shown in [2] and also to construct the suffix array (semi-externally) from the BWT. So it is now possible to construct SA, BWT and LCP with about n bytes without using an external algorithm. These arrays are components of several full-text indexes.

In the full paper, we will show how the presented algorithm can be modified so that it directly computes the suffix array.

References

1. Bauer, M.J., Cox, A.J., Rosone, G.: Lightweight algorithms for constructing and inverting the BWT of string collections. Theoretical Computer Science 483, 134–148 (2013)
2. Beller, T., Gog, S., Ohlebusch, E., Schnattinger, T.: Computing the longest common prefix array based on the Burrows-Wheeler transform. Journal of Discrete Algorithms 18, 22–31 (2013)
3. Bentley, J.L., Sedgewick, R.: Fast algorithms for sorting and searching strings. In: Proc. 8th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 360–369 (1997)
4. Bingmann, T., Fischer, J., Osipov, V.: Inducing suffix and lcp arrays in external memory. In: Proc. Wkshp. Algorithm Engineering and Experiments (2013)
5. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. Research Report 124, Digital Systems Research Center (1994)
6. Dementiev, R., Kärkkäinen, J., Mehnert, J., Sanders, P.: Better external memory suffix array construction. Journal of Experimental Algorithmics 12, Article No. 3.4 (2008)
7. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Proc. IEEE Symposium on Foundations of Computer Science, pp. 390–398 (2000)
8. Ferragina, P., Gagie, T., Manzini, G.: Lightweight data indexing and compression in external memory. Algorithmica 63(3), 707–730 (2012)
9. Gog, S.: Compressed Suffix Trees: Design, Construction, and Applications. PhD thesis, University of Ulm, Germany (2011)
10. Jacobson, G.: Space-efficient static trees and graphs. In: Proc. 30th Annual Symposium on Foundations of Computer Science, pp. 549–554. IEEE (1989)
11. Kärkkäinen, J.: Fast BWT in small space by blockwise suffix sorting. Theoretical Computer Science 387(3), 249–257 (2007)
12. Larsson, J., Sadakane, K.: Faster suffix sorting. Theoretical Computer Science 387(3), 258–272 (2007)
13. Lippert, R.A., Mobarry, C.M., Walenz, B.P.: A space-efficient construction of the Burrows-Wheeler transform for genomic data. Journal of Computational Biology 12(7), 943–951 (2005)
14. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys 39(1), Article No. 2 (2007)
15. Nong, G., Zhang, S., Chan, W.: Linear suffix array construction by almost pure induced-sorting. In: Proc. Data Compression Conference, pp. 193–202 (2009)
16. Nong, G., Zhang, S., Chan, W.: Two efficient algorithms for linear time suffix array construction. IEEE Transactions on Computers 60(10), 1471–1484 (2011)
17. G. Nong Practical Linear-Time O(1)-Workspace Suffix Sorting for Constant Alphabets. ACM Transactions on Information Systems (to appear, July 2013)
18. Okanohara, D., Sadakane, K.: A linear-time Burrows-Wheeler transform using induced sorting. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 90–101. Springer, Heidelberg (2009)
19. Puglisi, S.J., Smyth, W.F., Turpin, A.: A taxonomy of suffix array construction algorithms. ACM Computing Surveys 39(2), Article No. 4 (2007)