Faster Average Case Low Memory Semi-External Construction of the Burrows-Wheeler Transform

German Tischler*

The Wellcome Trust Sanger Institute, Wellcome Trust Genome Campus
Hinxton, Cambridge, CB10 1SA, United Kingdom
german.tischler@sanger.ac.uk

Abstract

The Burrows Wheeler transform has applications in data compression as well as full text indexing. Despite its important applications and various existing algorithmic approaches the construction of the transform for large data sets is still challenging. In this paper we present a new semi external memory algorithm capable of constructing the transform in time $O(n \log^2 \log n)$ on average if sufficient internal memory is available to hold a fixed fraction of the input text. In the worst case the run-time is $O(n \log n \log \log n)$.

1 Introduction

The Burrows-Wheeler transform (BWT) was introduced to facilitate the lossless compression of data (cf. [3]). It has an intrinsic connection to some data structures used for full text indexing like the suffix array (cf. [11]) and is at the heart of some compressed full text self indexes like the FM index (see [8]). The FM index requires no more space than the k'th order entropy compressed input text plus some asymptotically negligible supporting data structures. Many construction algorithms for the BWT are based on its relation to the suffix array, which can be computed from the input text in time linear in the length of that text (see e.g. [10, 13]). While these algorithms run in linear time and are thus theoretically optimal they require $O(n \log n)$[1] bits of space for the uncompressed suffix array given an input text of

In: Costas S. Iliopoulos, Alessio Langiu (eds.): Proceedings of the 2nd International Conference on Algorithms for Big Data, Palermo, Italy, 7-9 April 2014, published at http://ceur-ws.org/

* Supported by the Wellcome Trust
[1] by log we mean $\log_2$ in this paper

length n while the text itself can be stored in a space of $n\lceil\log\sigma\rceil$ bits for an alphabet of size σ where we often have $\sigma \ll n$ and in most applications σ is constant. Algorithms for computing the suffix array in external memory have been proposed (see e.g. [2, 5]) but these algorithms require large amounts of space and input/output in external memory. An asymptotically optimal internal memory solution concerning time and space has been proposed [9]. However the space usage of this algorithm is $O(n)$ bits for constant alphabets, where an inspection of the algorithm suggests that the actual practical memory usage of the algorithm is several times the size of the text in bits. The practical space usage of the algorithm subsequently presented in [14] is lower (i.e. the involved constants are smaller) while theoretically not linear. It however still requires multiple times as much space as the input text. A sample implementation given by the authors only works for input sizes of up to 2^{32} (see [1]) and only handles a single level of the recursive algorithm. Given the implementation complexity of the algorithm it remains unclear if it would scale well. Crochemore et al present an algorithm computing the BWT in quadratic time with constant additional space (see [4]). In [1] Beller et al propose a semi external algorithm for the construction of the BWT based on induced sorting. An algorithm is called semi external if it uses non negligible amounts of internal as well as external memory. According to the authors the algorithm scales to arbitrary input sizes and uses about one byte (i.e. 8 bits) per input symbol in internal memory. An algorithm constructing the BWT of a text by block-wise merging using a finite amount of internal memory is presented in [7]. The algorithm partitions the text into a set of fixed size blocks. The run-time is $O(n^2/b)$ for a block size of b and a text length of n. It requires an amount of internal memory which is roughly sufficient to compute the suffix array of a single of these block. In particular the amount of internal memory used can be smaller than the space required for the text. In this paper we modify this algorithm to run in time $O(n \log n \log\log n)$ in the worst case and $O(n \log^2 \log n)$ on average for the case where we are able to keep a fixed fraction of the text in memory. Assuming the practically common case of a finite alphabet the algorithm in [7] uses blocks of size $O(n/\log n)$ blocks when provided with $O(n)$ bits of space in internal memory so its total run-time for this setting is $O(n \log n)$. In consequence our algorithm is faster on average and slower by $O(\log\log n)$ for a very unlikely worst case. Compared to the algorithm presented in [1] our algorithm can work with less internal memory. For DNA for instance the complete text can be held in memory using about 2 bits per symbol which is significantly less than a full byte (8 bits) per character.

2 Definitions

For a string $s = s_0 s_1 s_2 \ldots s_{m-1}$ of length $|s| = m$ we define $s[i] = s_i$ for $0 \leq i < m$ and for $s = s_0 s_1 \ldots$ we define $s[i] = s_i$ for $0 \leq i$. For a finite word u and a finite or infinite word v we write their concatenation as uv. For any finite words u, x and finite or infinite words w, v such that $w = uxv$ we call u a prefix, v a suffix and x a factor of w. The empty word consisting of no symbols is denoted by ϵ. For a string s and indices $0 \leq i \leq j < |s|$ we denote the factor $s[i]s[i+1]\ldots s[j]$ by $s[i,j]$. For any i, j such that $i > j$ the term $s[i,j]$ denotes the empty word. A finite word w

has period p iff $w[i] = w[i+p]$ for $i = 0, \ldots, |w| - p - 1$ and an infinite word w has period p iff $w[i] = w[i+p]$ for $i = 0, 1, \ldots$. For a finite word u and $k \in \mathbb{N}$ the k'th power u^k of u is defined by $u^0 = \epsilon$ and $u^{i+1} = u^i u$ for $i \in \mathbb{N}$. A word w is primitive if it is not a power of a word u such that $|u| < |w|$. A word u is a root of w if $w = u^k$ for some $k \in \mathbb{N}$. A word w is a square if there is a word u such that $w = u^2$. Throughout this paper let $\Sigma = \{0, 1, \ldots, \sigma - 1\}$ denote a finite alphabet for some $\sigma > 0$ and let $t = t_0 t_1 \ldots t_{n-1} \in \Sigma^n$ denote a finite string of length $n > 0$. We define the semi infinite string $\tilde{t}$ by $\tilde{t}[i] = t[i - \lfloor i/n \rfloor n]$ for $i \geq 0$. We define the suffix $\tilde{t}_i$ of $\tilde{t}$ as $\tilde{t}_i = \tilde{t}[i]\tilde{t}[i+1]\ldots$ and $\tilde{t}_i < \tilde{t}_j$ for $i, j \in \mathbb{N}, i \neq j$ iff either $\tilde{t}_i = \tilde{t}_j$ and $i < j$ or for the smallest $\ell \geq 0$ such that $\tilde{t}[i+\ell] \neq \tilde{t}[j+\ell]$ we have $\tilde{t}[i+\ell] < \tilde{t}[j+\ell]$. The suffix array A of t is defined as the permutation of the numbers $0, 1, \ldots, n-1$ such that $\tilde{t}_{A[i]} < \tilde{t}_{A[i+1]}$ for $i = 0, 1, \ldots, n-2$ and the Burrows-Wheeler transform (BWT) $B = b_0 b_1 \ldots b_{n-1}$ of t is given by $b_i = \tilde{t}[A[i] + n - 1]$ for $i = 0, 1, \ldots, n-1$.

3 BWT construction by block-wise merging

We give a short high level description of the algorithm by Ferragina et al. in [7] as we will be modifying it. Unlike our algorithm it assumes the input string to have a unique minimal terminator symbol. Given a block size b the input string t is partitioned into $c = \lceil n/b \rceil$ blocks $T_0, T_1, \ldots, T_{c-1}$ of roughly equal size. The algorithm starts by suffix sorting the last block, computing its BWT B_{c-1} and the bit array gt_{c-1} which denotes for each suffix in T_{c-1} but the first whether it is smaller or larger than the first. The BWT of $T_i \ldots T_{c-1}$ for $i < c-1$ is computed by first computing the suffix array for the suffixes starting in T_i by using the text of T_i and T_{i+1} in memory and handling the comparison of suffixes starting in T_i but equal until both have entered T_{i+1} by using the bit vector gt_{i+1} which explicitly stores the result of this comparison. The BWTs of T_i and $T_{i+1}T_{i+2}\ldots T_{c-1}$ are merged by computing the ranks of the suffixes starting in $T_{i+1}T_{i+2}\ldots T_{c-1}$ in the sorted set of suffixes of T_i and computing a gap array G_i which denotes how many suffixes from $T_{i+1}T_{i+2}\ldots T_{c-1}$ are to be placed before the suffixes in T_i, between two adjacent suffixes in T_i and after all suffixes in T_i. This process follows a backward search of $T_{i+1}T_{i+2}\ldots T_{c-1}$ in T_i. Using the array G_i it is simple to merge the two BWTs together. For computing the rank of a suffix from $T_{i+1}\ldots T_{c-1}$ it is necessary to know whether it is smaller or larger than the one at the start of $T_{i+1}T_{i+2}\ldots T_{c-1}$ as B_i is not a conventional BWT. For further details about the algorithm the reader is referred to [7].

4 Sorting single blocks

The algorithm by Ferragina et al processes each single block relying on knowledge about the priorly fully processed following block, in case of the last block the terminator. For our algorithm we need to be able to sort a single block without knowing the complete sorted order of the next block. For this purpose we need to be able to handle repetitions, one of the major challenges along the way, efficiently. For our block sorting only repetitions with a period not exceeding the maximum block size are relevant. Consider a block of b suffixes $\tilde{t}_{i+j}$ for some $i \in \mathbb{N}^+, 0 \leq j < b$. We say it generates a repetition with period $p, 1 \leq p \leq b$ iff $\tilde{t}[b-p, b-1] = \tilde{t}[b, b+p-1]$

and propagates a repetition with period $p, 1 \leq p \leq b$ iff $\tilde{t}_i[0, b+2p-1]$ has period p. If it propagates repetitions of any periods, then there is a unique minimal period dividing all other propagated periods. This unique minimal period can then be computed in time $O(b)$ and space $O(b \log \sigma)$ bits using minor modifications of standard string algorithms. As there is a unique minimal period propagated by a block if any and we are only interested in generated periods which are propagated by the next block we can compute the relevant generation properties of a block in the same time and space bounds. Given a target block size b' we partition the given text into a set of blocks of size either $b = \lceil \frac{n}{\lceil (n/b') \rceil} \rceil \leq b'$ or $b-1$ where the first $n \bmod b$ blocks have length b and the rest length $b-1$. For the propagation of repetitions we extend the blocks of length $b-1$ to size b by adding the (circularly) next character to the right. Using this information about short period repetitions in the input string, we are able to handle the sorting of a single block of suffixes extending beyond the end of the block efficiently by reducing long repetitions.

Lemma 4.1 *A block of b circular suffixes of $\tilde{t}$ can be sorted in lexicographical order using time $O(b)$ and space $O(b \log b)$ bits using precomputed repetition propagation data.*

For forward searching using the suffix array it is useful to in addition have the longest common prefix (LCP) array. For two strings u, v let $\mathrm{LCP}(u, v) = \mathrm{argmax}_{l=0}^{\min\{|u|,|v|\}} u[0, \ell-1] = v[0, \ell-1]$. For a block $\tilde{t}[i, i+b-1]$ for $i, b \in \mathbb{N}, b > 0$ let $\mathcal{A}$ denote the permutation of $i, i+1, \ldots, i+b-1$ such that $\tilde{t}_{\mathcal{A}[j]} < \tilde{t}_{\mathcal{A}[j+1]}$ for $j = 0, 1, \ldots, b-2$. Then the LCP array of the block is defined by $\mathrm{LCP}[0] = 0$ and $\mathrm{LCP}[i] = \mathrm{LCP}(\tilde{t}_{\mathcal{A}[i-1]}, \tilde{t}_{\mathcal{A}[i]})$ for $i = 1, 2, \ldots, b-1$. Using a repetition reduction method similar to the suffix sorting case we obtain the following result.

Lemma 4.2 *The LCP array for a block of b circular suffixes on $\tilde{t}$ can be computed in time $O(b)$ and space $O(b \log b)$ bits using precomputed repetition propagation data.*

5 Merging Pairs of Adjacent Blocks

In our modified algorithm we replace the completely skewed binary merge tree used in [7] by a balanced binary merge tree. Consequently we will need to be able to merge blocks with a block size in $\Omega(n)$. For merging two adjacent blocks we need the following components:

1. The BWT of the left and right block. These can be compressed and in external memory as they will be scanned sequentially.
2. An internal memory index of the left block suitable for backward search in $O(1)$ time per step. An FM type index using space $b_l H_k + o(n \log \sigma)$ bits can be used where b_l is the length of the left block and H_k denotes the k'th order entropy of the left block (see [12]).
3. The *gt* bit vectors for the left and right block. Scanned sequentially and thus can be read streaming from external memory.
4. The number of circular suffixes in the left block smaller than the rightmost suffix of the right block. Used as the start point for the backward search.

5. The gap array G.

The first three are equivalent to those used in [7]. The rank of the rightmost suffix in the right block relative to the suffixes of the left block can be obtained by employing forward search on one or more text blocks. If the left block is a single block which was produced by explicit suffix sorting using the method of Section 4, then the rank can be obtained using classical forward search in the suffix array while using the adjoined LCP array. This takes time $O(n + \log b)$ in the worst case (on average this can be expected to be $O(\log n + \log b)$, see [15]). If the left block was already obtained by merging c blocks together, then the desired rank can be obtained as the sum of the ranks of the suffix relative to all single blocks composing the left block in time $O(c(n + \log b))$. Assuming the blocks are merged together in a balanced binary merge tree the total time used for forward searches is $O(\frac{n}{b} \log \frac{n}{b} n)$ in the worst case and $O(\frac{n}{b} \log \frac{n}{b} \log n)$ on average. If we choose $b \in O(\frac{n}{\log n})$ then this becomes $O(n \log n \log \log n)$. The memory required for the index of the left block in internal memory will be $b_l \log \sigma + o(b_l \log \sigma)$ for a left block size of b_l assuming that the entropy compression is ineffective. This leaves us with the space required for the gap array. In the original algorithm this is a conventional array in internal memory taking space $O(b \log n)$ bits for a left block size of b. As we want to be able to merge blocks with size in $\Omega(n)$ this space requirement is too high. Using Elias γ code (cf. [6]) we can store the gap array for merging a left and right block of length b_l and b_r respectively in $O(b_l + b_r)$ bits of space. γ code however is not suitable for efficient updating as we would need it for computing the gap array. We solve this by producing partial sparse gap arrays and merging these together as needed. These sparse gap arrays are encoded using two γ coded number sequences where one encodes the indices of non-zero values in differential coding and the other encodes the non-zero values. The array G is produced by backward searching the suffixes of the right block in a suitable index of the left block. After each step exactly one element of G is incremented. The sum over the elements of G increases by exactly one for each step. For computing a complete gap array one option is to start by producing sparse arrays consisting of a single element of value 1. Whenever we have produced two partial arrays of sum s we immediately merge them together to a partial array of sum $2s$ in time $O(s)$. This method guarantees that the set of sparse gap arrays present at any one time is bounded in space by $O(b_l + b_r)$ bits. The total merging of partial gap arrays to obtain the final gap array then takes time $O(b_r \log b_r)$. If we accumulate $\frac{b_r}{\log^2 b_r}$ indices for incrementing before writing a partial gap array then we can reduce the merging time to $O(b_r \log \log b_r)$ without increasing the space used by the algorithm.

The *gt* array for the merged block can be composed by concatenating the *gt* array for the left block and an array storing the respective information for the right block computed while performing the backward search for filling the gap array. For this purpose we need to know the rank of the leftmost suffix in the left block. This can either be computed using forward search on the suffix arrays of the basic blocks or extracted from a sampled inverse suffix array which can be computed along the way. The sampled inverse suffix arrays of two blocks can just like the BWTs of the two blocks be merged using the gap array. This is also an operation based on stream accesses, so it can be done in external memory in time $O(b)$.

6 BWT Computation by Balanced Tree Block Merging

Using the building blocks described above we can now describe the complete algorithm for computing the BWT of t by merging basic blocks according to a balanced binary tree.

1. Choose a target block size $b' \in O(\frac{n}{\log n})$ and deduce a block size $b = \lceil \frac{n}{\lceil \frac{n}{b'} \rceil} \rceil$ such that the number of blocks c satisfies $c = \lceil \frac{n}{b} \rceil = \lceil \frac{n}{b'} \rceil$ and n can be split into blocks of size b and $b-1$ only. Split t such that the blocks of size b appear before those of size b'. This step takes constant space and time.
2. Compute which blocks in t propagate repetitions of period at most b and for each block which is followed by a block propagating a repetition whether it is generating this repetition. This takes time $O(n)$ in total and space $O(b \log \sigma) = O(\frac{n \log \sigma}{\log n}) \subseteq O(n)$ bits. The result data can be stored in external memory.
3. Compute a balanced merge tree for the blocks. Start with a root representing all blocks. If a node containing a single block is considered produce a leaf and stop. Otherwise for an inner node representing $k > 1$ blocks produce a left subtree from the $\lceil \frac{k}{2} \rceil$ leftmost blocks and a right subtree from $\lfloor \frac{k}{2} \rfloor$ rightmost blocks in t. The tree has $O(\log n)$ nodes. Each node stores at most two (start and end) block indices taking $O(\log \log n)$ bits and two node pointers also taking space $O(\log \log n)$ bits. So the total tree takes space $O(\log n \log \log n)$ bits. It can be computed in time $O(\log n)$.
4. Sort the blocks and store the resulting BWT, gt and sampled inverse suffix arrays in external memory. Using the suffix and LCP arrays of the basic blocks also compute the start ranks necessary for the backward searches when merging the blocks together. This takes time $O(n \log n \log \log n)$ in the worst case and $O(n)$ on average and space $O(b \log b) = O(\frac{n}{\log n} \log \frac{n}{\log n}) = O(n)$ bits of internal memory.
5. Process the merge tree. Mark all leafs as finished and all inner nodes as unfinished. While there are unfinished nodes choose any unfinished node with only finished children, merge the respective blocks and mark the node as finished. There are $O(\log n)$ leafs and the tree is balanced, so it has $O(\log \log n)$ levels. Each single level can be processed in time $O(n \log \log n)$. So the total run time for the tree merging phase is $O(n \log^2 \log n)$. The maximum internal memory space usage appears when performing the merge operation at the root of the tree. Here we need space $b_l H_k + o(b_l \log \sigma)$ bits where b_l denotes the sum of the length of the blocks in the left subtree which is $O(n)$ and H_k denotes the k'th order entropy of the text comprising those text blocks.

Summing over all steps the run-time of the algorithm is $O(n \log n \log \log n)$ in the worst case and $O(n \log^2 \log n)$ on average. In practice this means we can compute the BWT of a text as long as we are able to hold the text (more precisely the text for the left subtree of the merge tree) in internal memory. If we can hold a fixed fraction of the text in main memory, then we can still compute the BWT of the text in the same run-time by resorting to the original iterative merging scheme from [7]. We decompose the text into blocks of size b' such that $b' \leq \frac{n \log \sigma}{c \log n}$ where $\frac{1}{c}$ is the fixed fraction of the text we can hold in internal memory and compute a partial BWT for each of these blocks where the suffixes sorted are considered as coming from the whole text, i.e. suffix comparisons are still over $\tilde{t}$ and not limited to a single of the blocks. Then we merge these blocks along a totally skewed merge tree such that the left block always has size about b'. The size of the set of partial sparse gap arrays required at any time remains bounded by $O(n)$ bits. As the number of blocks is fixed, the total asymptotical run-time of the algorithm remains $O(n \log n \log \log n)$ in the worst case and $O(n \log^2 \log n)$ on average.

7 Conclusion

We have presented a new semi external algorithm for computing the Burrows-Wheeler transform. On average our new algorithm is faster then the algorithm of Ferragina et al published in [7]. In comparison with the algorithm by Beller et in [1] our algorithm can be applied for the case when less than 8 bits per symbol of internal memory are available. Due to space constraints proofs, parallelisation of our algorithm and the discussion of an implementation study are postponed to another paper. Sample code implementing parts of the ideas in this paper is available from the author on request.

References

[1] T. Beller, M. Zwerger, S. Gog, and E. Ohlebusch. Space-Efficient Construction of the Burrows-Wheeler Transform. In O. Kurland, M. Lewenstein, and E. Porat, editors, *SPIRE*, volume 8214 of *Lecture Notes in Computer Science*, pages 5–16. Springer, 2013.

[2] T. Bingmann, J. Fischer, and V. Osipov. Inducing Suffix and LCP Arrays in External Memory. In P. Sanders and N. Zeh, editors, *ALENEX*, pages 88–102. SIAM, 2013.

[3] M. Burrows and D. Wheeler. A Block-Sorting Lossless Data Compression Algorithm. Digital Systems Research Center. *RR-124*, 1994.

[4] M. Crochemore, R. Grossi, J. Kärkkäinen, and G. M. Landau. A Constant-Space Comparison-Based Algorithm for Computing the Burrows-Wheeler Transform. In J. Fischer and P. Sanders, editors, *CPM*, volume 7922 of *Lecture Notes in Computer Science*, pages 74–82. Springer, 2013.

[5] R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders. Better external memory suffix array construction. *ACM Journal of Experimental Algorithmics*, 12, 2008.

[6] P. Elias. Universal codeword sets and representations of the integers. *Information Theory, IEEE Transactions on*, 21(2):194–203, 1975.

[7] P. Ferragina, T. Gagie, and G. Manzini. Lightweight Data Indexing and Compression in External Memory. *Algorithmica*, 63(3):707–730, 2012.

[8] P. Ferragina and G. Manzini. Opportunistic Data Structures with Applications. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 390–398. IEEE, 2000.

[9] W.-K. Hon, K. Sadakane, and W.-K. Sung. Breaking a Time-and-Space Barrier in Constructing Full-Text Indices. In *FOCS*, pages 251–260. IEEE Computer Society, 2003.

[10] J. Kärkkäinen and P. Sanders. Simple Linear Work Suffix Array Construction. In *Automata, Languages and Programming*, pages 943–955. Springer, 2003.

[11] U. Manber and G. Myers. Suffix Arrays: a New Method for On-line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[12] G. Navarro and V. Mäkinen. Compressed Full-Text Indexes. *ACM Computing Surveys (CSUR)*, 39(1):2, 2007.

[13] G. Nong, S. Zhang, and W. H. Chan. Two Efficient Algorithms for Linear Time Suffix Array Construction. *Computers, IEEE Transactions on*, 60(10):1471–1484, 2011.

[14] D. Okanohara and K. Sadakane. A Linear-Time Burrows-Wheeler Transform Using Induced Sorting. In J. Karlgren, J. Tarhio, and H. Hyyrö, editors, *SPIRE*, volume 5721 of *Lecture Notes in Computer Science*, pages 90–101. Springer, 2009.

[15] W. Szpankowski. On the Height of Digital Trees and Related Problems. *Algorithmica*, 6(1-6):256–277, 1991.