Fast BWT in Small Space by Blockwise Suffix Sorting

Juha Kärkkäinen

Department of Computer Science, University of Helsinki, Finland

juha.karkkainen@cs.helsinki.fi

The usual way to compute the Burrows–Wheeler transform (BWT) [3] of a text is by constructing the suffix array of the text. Even with space-efficient suffix array construction algorithms [12, 2], the space requirement of the suffix array itself is often the main factor limiting the size of the text that can be handled in one piece, which is crucial for constructing compressed text indexes [4, 5]. Typically, the suffix array needs $4n$ bytes while the text and the BWT need only n bytes each and sometimes even less, for example $2n$ bits each for a DNA sequence.

We reduce the space dramatically by constructing the suffix array in *blocks* of lexicographically consecutive suffixes. Given such a block, the corresponding block of the BWT is trivial to compute.

Theorem 1 *The BWT of a text of length n can be computed in $\mathcal{O}(n \log n + n\sqrt{v} + D_v)$ time (with high probability) and $\mathcal{O}(n/\sqrt{v} + v)$ space (in addition to the text and the BWT), for any $v \in [1, n]$. Here $D_v = \sum_{i \in [0,n)} \min(d_i, v) = \mathcal{O}(nv)$, where d_i is the length of the shortest unique substring starting at i.*

Proof (sketch). Assume first that the text has no repetitions longer than v, i.e., $d_i \leq v$ for all i. Choose a set of $\mathcal{O}(v)$ random suffixes that divide the suffix array into blocks. The sizes of the blocks are counted in $\mathcal{O}(n \log v + D_v)$ time using the string binary search technique from [11]. Blocks are then combined to obtain $\mathcal{O}(\sqrt{v})$ blocks of size $\mathcal{O}(n/\sqrt{v})$. The suffixes in a block are collected in $\mathcal{O}(n)$ time and $\mathcal{O}(v)$ extra space using a modified Knuth–Morris–Pratt algorithm with (the prefixes of) the bounding suffixes as patterns. A block B is sorted in-place in $\mathcal{O}(|B| \log |B| + D_v(B))$ time using the multikey quicksort [1], where $D_v(B)$ is as D_v but summed over the suffixes in B. Repetitions longer than v are handled in all stages with the difference cover sampling (DCS) data structure from [2] that supports constant time order comparison of any two suffixes that have a common prefix of length v. The DCS data structure can be constructed in $\mathcal{O}((n/\sqrt{v}) \log(n/\sqrt{v}) + D_v(C))$ time and $\mathcal{O}(n/\sqrt{v} + v)$ space, where C is a set of $\mathcal{O}(n/\sqrt{v})$ suffixes. □

With the choice of $v = \log^2 n$, we get an algorithm using $\mathcal{O}(n)$ bits of space and running in $\mathcal{O}(n \log n)$ time on average and in $\mathcal{O}(n \log^2 n)$ time in the worst case.

The algorithm is also fast and space-efficient in practice. The following table shows the space requirement of a practical implementation for some v (not including the text, the BWT and about $16v + \mathcal{O}(\log n)$ bytes).

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

For small v, the runtime is dominated by the sorting of blocks making the performance similar to the algorithm in [2], which is competitive with the best algorithms. For larger v, the time needed for the $\mathcal{O}(\sqrt{v})$ scans to collect suffixes for a block takes over. The D_v term is dominant only in pathological cases.

There are two other categories of algorithms for computing the BWT when there is not enough space for the suffix array: compressed suffix array construction [10, 6, 7] and external memory suffix array construction [8, 9]. Our guess is that the blockwise suffix sorting is the fastest alternative in practice when v is not too large, and we are in the process of verifying this experimentally.

References

[1] J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proc. 8th Annual Symposium on Discrete Algorithms*, pages 360–369. ACM, 1997.

[2] S. Burkhardt and J. Kärkkäinen. Fast lightweight suffix array construction and checking. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching*, volume 2676 of *LNCS*, pages 55–69. Springer, 2003.

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

[4] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. 41st Annual Symposium on Foundations of Computer Science*, pages 390–398. IEEE, 2000.

[5] P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Proc. 12th Annual Symposium on Discrete Algorithms*, pages 269–278. ACM–SIAM, 2001.

[6] W.-K. Hon, T.-W. Lam, K. Sadakane, and W.-K. Sung. Constructing compressed suffix arrays with large alphabets. In *Proc. 14th International Symposium on Algorithms and Computation*, volume 2906 of *LNCS*, pages 240–249. Springer, 2003.

[7] W.-K. Hon, K. Sadakane, and W.-K. Sung. Breaking a time-and-space barrier in constructing full-text indices. In *Proc. 44th Annual Symposium on Foundations of Computer Science*, pages 251–260. IEEE, 2003.

[8] J. Kärkkäinen and S. S. Rao. Full-text indexes in external memory. In U. Meyer, P. Sanders, and J. Sibeyn, editors, *Algorithms for Memory Hierarchies (Advanced Lectures)*, volume 2625 of *LNCS*, chapter 7, pages 149–170. Springer, 2003.

[9] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proc. 30th International Conference on Automata, Languages and Programming*, volume 2719 of *LNCS*, pages 943–955. Springer, 2003.

[10] T.-W. Lam, K. Sadakane, W.-K. Sung, and S.-M. Yiu. A space and time efficient algorithm for constructing compressed suffix arrays. In *Proc. 8th Annual International Conference on Computing and Combinatorics*, volume 2387 of *LNCS*, pages 401–410. Springer, 2002.

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

[12] G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. In *Proc. 10th Annual European Symposium on Algorithms*, volume 2461 of *LNCS*, pages 698–710. Springer, 2002.