1 SPACE EFFICIENT LINEAR TIME COMPUTATION OF THE BURROWS AND WHEELER-TRANSFORMATION

Stefan Kurtz

Technische Fakultät, Univ. Bielefeld, Postfach 100 131, 33501 Bielefeld, Germany*

kurtz@techfak.uni-bielefeld.de

Bernhard Balkenhol

Fakultät für Mathematik, Univ. Bielefeld, Postfach 100 131, 33501 Bielefeld, Germany

bernhard@mathematik.uni-bielefeld.de

1.1 INTRODUCTION

In [Burrows and Wheeler, 1994] a universal data compression algorithm (BW-algorithm, for short) is described which achieves compression rates that are close to the best known rates. Due to its simplicity, the algorithm can be implemented with relatively low complexity. Recently [Balkenhol et al., 1999] modified the BW-algorithm to improve the compression rate even further. For a thorough discussion on the information theoretic background of the BW-algorithm and more references, see [Balkenhol and Kurtz, 1998]. The most time and space consuming part of the BW-algorithm is the Burrows-Wheeler Transformation (BWT, for short), which permutes the input string in such a

*partially supported by DFG-grant Ku 1257/1-1

way that characters with a similar context are grouped together. In [Burrows and Wheeler, 1994], it was observed that for an input string of length n, this transformation can be computed in $O(n)$ time and space using suffix trees. However, suffix trees have a reputation of being very greedy for space, and therefore most researchers resorted to alternative non-linear methods for computing the BWT: The algorithm of [Manber and Myers, 1993] runs in $O(n \log n)$ worst case time and it requires $8n$ bytes of space. The algorithm of [Bentley and Sedgewick, 1997] is based on *Quicksort*. It is fast on average, but the worst case running time is $O(n^2)$. The Benson-Sedgewick algorithm requires $4n$ bytes. Its running time can be improved in practice, for the cost of $4n$ extra bytes. Recently, [Sadakane, 1998] showed how to combine the Manber-Myers Algorithm with the Bentley-Sedgewick Algorithm, to achieve a method running in $O(n \log n)$ worst case time and using $9n$ bytes.

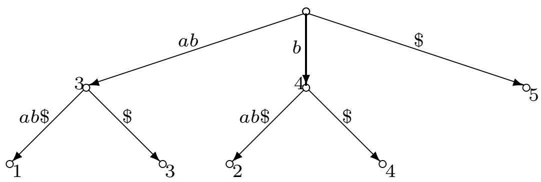
With the recently developed implementation technique of [Kurtz, 1998], suffix trees can be represented more space efficiently, so that the space advantage of the non-linear methods is considerably reduced. In this paper, we further improve on [Kurtz, 1998], and show that a suffix tree based method requires on average about the same amount of space as the non-linear methods mentioned above. The improvement is achieved by exploiting the fact, that in practice, the BW-algorithm processes long input strings in blocks of a limited size (for this reason some researchers use the notion of "Block-Sorting"-algorithm). Assuming a maximal block size of $2^{21} - 1 = 2{,}097{,}151$, we show that the suffix tree can be implemented in $8.83n$ bytes on average for the files of the Calgary Corpus. This is $0.6n$ and $9.77n$ bytes less than the implementation technique of [Kurtz, 1998] and of [McCreight, 1976], respectively. The worst case space requirement of our implementation technique is $16n$ bytes, compared to $20n$ bytes for [Kurtz, 1998] and $28n$ bytes for [McCreight, 1976]. The reduction of the space requirement due to an upper bound on n seems trivial. However, we will see that it involves a considerable amount of engineering work to achieve the improvement, while retaining the linear worst case running time for constructing the BWT.

This paper is organized as follows: In Section 1.2 we introduce some basic notions. Section 1.3 describes how to implement suffix trees space efficiently. In Section 1.4, we show how to read the BWT from the suffix tree. Section 1.5 reports on experimental results.

1.2 PRELIMINARIES

Let Σ be a finite ordered set, the *alphabet*. k denotes the size of Σ. We assume that x is a string over Σ of length $n \geq 1$ and that $\$ \in \Sigma$ is a character such that for any $i \in [1, n]$ we have $x_i < \$$. For any $i \in [1, n+1]$, let $S_i = x_i \dots x_n\$$ denote the ith non-empty suffix of $x\$$. Let $S_{j_1}, S_{j_2}, \dots, S_{j_{n+1}}$ be the sequence of all non-empty suffixes of $x\$$ in lexicographic order. This gives a bijective mapping $\varphi : [1, n+1] \to [1, n+1]$ defined by $\varphi(i) = j_i$. φ is the *suffix order on* $x\$$. Note that $\varphi(n+1) = n+1$, since $S_{n+1} = \$$. The *Burrows and Wheeler*

Figure 1.1 The suffix tree for $x = abab$. Leaves are annotated with leaf numbers and branching nodes with head positions.



Transformation of x is the string $\widetilde{x}$ of length $n+1$ such that for any $i \in [1, n+1]$ we have $\widetilde{x}_i = \$$ if $\varphi(i) = 1$, and $\widetilde{x}_i = x_{\varphi(i)-1}$ otherwise.

A Σ^+*-tree* T is a finite rooted tree with edge labels from Σ^+. For each $a \in \Sigma$, a node u in T has at most one a-edge $u \xrightarrow{av} w$ for some string v and some node w. Let u be a node in T. We denote u by $\overline{w}$ if and only if w is the concatenation of the edge labels on the path from the *root* to u. The node $\overline{\varepsilon}$ is the *root*. $depth(\overline{w}) := |w|$ is the *depth* of $\overline{w}$. A string s *occurs* in T if T contains a node $\overline{sv}$, for some string v.

1.3 SUFFIX TREES AND THEIR IMPLEMENTATION

The *suffix tree* for x, denoted by ST, is the Σ^+-tree T with the following properties: (i) each node is either a leaf, a branching node, or the *root*, and (ii) a string w occurs in T if and only if w is a substring of $x\$$.

ST can be constructed and represented in linear time and space using one of the algorithms described in [Weiner, 1973, McCreight, 1976, Ukkonen, 1995, Farach, 1997]. See also [Giegerich and Kurtz, 1997] which reviews [Weiner, 1973, McCreight, 1976, Ukkonen, 1995] and reveals relationships between these algorithms much closer than one would think. The *suffix link* for a node $\overline{aw}$ in ST is an unlabeled directed edge from $\overline{aw}$ to the node $\overline{w}$. Note that the latter exists in ST, whenever $\overline{aw}$ exists. We consider suffix links to be a part of the suffix tree, since they are required for most of the linear time suffix tree constructions (see [Weiner, 1973, McCreight, 1976, Ukkonen, 1995]). For any branching node $\overline{aw}$ in ST, *suffixlink*$(\overline{aw})$ refers to node $\overline{w}$.

The *raison d'etre* of a branching node $\overline{w}$ in ST is the first branching occurrence of w in t, i.e., the first occurrence of wa, for some $a \in \Sigma$, such that w occurs to the left, but not wa. We therefore introduce the notions *head* and *head position*: Let $head_1 = \varepsilon$ and for $i \in [2, n+1]$ let $head_i$ be the longest prefix of S_i which is also a prefix of S_j for some $j \in [1, i-1]$. For each branching node $\overline{w}$ in ST, let $headposition(\overline{w})$ denote the smallest integer $i \in [1, n+1]$ such that $w = head_i$. If $headposition(\overline{w}) = i$, then we say that the *head position* of $\overline{w}$ is i. Since there is a one-to-one correspondence between the *heads* and the branching nodes in ST (see [Kurtz, 1998]), the notion of head positions is well defined. Figure 1.1 shows the suffix tree for $x = abab$.

The head position j of some branching node $\overline{wu}$ tells us that the leaf $\overline{S_j}$ occurs in the subtree below node $\overline{wu}$. Hence wu is the prefix of S_j of length $depth(\overline{wu})$, i.e., the equality $wu = x_j \dots x_{j+depth(\overline{wu})-1}$ holds. As a consequence, the label of the incoming edge to node $\overline{wu}$ can be obtained by dropping the first $depth(\overline{w})$ characters of wu, where $\overline{w}$ is the predecessor of $\overline{wu}$: If $\overline{w} \xrightarrow{u} \overline{wu}$ is an edge in ST and $\overline{wu}$ is a branching node, then we have $u = x_i \dots x_{i+l-1}$ where $i = headposition(\overline{wu}) + depth(\overline{w})$ and $l = depth(\overline{wu}) - depth(\overline{w})$. Similarly, the label of the incoming edge to a leaf is determined from the leaf number and the depth of the predecessor: If $\overline{w} \xrightarrow{u} \overline{wu}$ is an edge in ST and $\overline{wu} = \overline{S_j}$ for some $j \in [1, n+1]$, then $u = x_i \dots x_n\$$ where $i = j + depth(\overline{w})$.

It is straightforward to show that for any branching node $\overline{aw}$ in ST either $headposition(\overline{aw})+1 = headposition(\overline{w})$ or $headposition(\overline{aw}) > headposition(\overline{w})$ holds, see [Kurtz, 1998]. As a consequence, we can discriminate all non-root nodes accordingly: $\overline{aw}$ is a *small* node if and only if $headposition(\overline{aw}) + 1 = headposition(\overline{w})$. $\overline{aw}$ is a *large* node if and only if $headposition(\overline{aw}) > headposition(\overline{w})$. The *root* is neither small nor large.

Let $b_1, b_2, \dots, b_q$ be the sequence of branching nodes ordered by their head position, i.e., $headposition(b_i) < headposition(b_{i+1})$ for any $i \in [1, q-1]$. Obviously, b_1 is the *root*. One can show that a small node in this sequence is always immediately followed by another branching node, and that b_q is a large node, see [Kurtz, 1998]. We can thus partition the sequence $b_2, \dots, b_q$ of branching nodes into *chains* of zero or more consecutive small nodes followed by a single large node. More precisely, a *chain* is a contiguous subsequence $b_l, \dots, b_r$, $r \geq l$, of $b_2, \dots, b_q$ such that *(i)* b_{l-1} is not a small node, *(ii)* $b_l, \dots, b_{r-1}$ are small nodes, and *(iii)* b_r is a large node.

One easily observes that any non-root branching node in ST is a member of exactly one chain. The following lemma, which is proved in [Kurtz, 1998], shows an interesting relationship between the small nodes and the large node of a chain:

Lemma 1 Let $b_l, \dots, b_r$ be a chain. The following properties hold for any $i \in [l, r-1]$:

(1) $suffixlink(b_i) = b_{i+1}$

(2) $depth(b_i) = depth(b_r) + (r - i)$

(3) $headposition(b_i) = headposition(b_r) - (r - i)$

According to this observation, it is not necessary to store $suffixlink(b_i)$, $depth(b_i)$, and $headposition(b_i)$ for any small node b_i. $suffixlink(b_i)$ refers to the next node in the chain, and if the distance $r - i$ of b_i to the large node b_r (denoted by $distance(b_i)$) is known, then $depth(b_i)$ and $headposition(b_i)$ can be obtained in constant time. This observation allows the following implementation technique: ST is represented by two tables T_{leaf} and T_{branch} which store the following values: For each *leaf number* $j \in [1, n+1]$, $T_{leaf}[j]$ stores a reference to the right brother of leaf $\overline{S_j}$. If there is no such brother, then $T_{leaf}[j]$ is a nil reference. Leaf $\overline{S_j}$ is referenced by leaf number j. Table T_{branch} stores the information for the small and the large nodes: For each small node $\overline{w}$, there is a *small record* which stores *distance*$(\overline{w})$, *firstchild*$(\overline{w})$, and *rightbrother*$(\overline{w})$. The latter two are references to the first child of $\overline{w}$ and to the right brother of $\overline{w}$, respectively. If there is no such brother of $\overline{w}$, then *rightbrother*$(\overline{w})$ is a nil reference. For any large node $\overline{w}$ there is a *large record* which stores *firstchild*$(\overline{w})$, *rightbrother*$(\overline{w})$, *depth*$(\overline{w})$, and *headposition*$(\overline{w})$. It also stores *suffixlink*$(\overline{w})$, whenever *depth*$(\overline{w}) \leq 2^{11} - 1$. The successors of a branching node are therefore found in a list whose elements are linked via the *firstchild*, *rightbrother*, and T_{leaf} references. To speed up the access to the successors, each such list is ordered according to the first character of the edge labels.

To guarantee constant time access from a small node b_i to the large node b_r, all records consist of integers (the general assumption is that an integer occupies 4 bytes or equivalently 32 bits). The integers are stored in table T_{branch}, ordered by the head positions of the corresponding branching nodes. All branching nodes are referenced by their *base address* in T_{branch}. The base address is the index of the first integer of the corresponding record. Since there are at most n large nodes in ST, the maximal base address is $3n-3$. A reference is either a base address or a leaf number. To distinguish these, we store a base address as an integer with offset $n+1$, i.e., base address i is stored as $n+1+i$. So a reference is smaller than $4n$, and if $n \leq 2^{21} - 1$, then it occupies 23 bits. Each depth and each head position occupies at most 21 bits.

Consider the range of the distance values. In the worst case, take e.g. $x = a^n$, there is only one chain of length $n-1$, i.e., the maximal distance value is $n-2$. However, this case is very unlikely to occur. To save space, we delimit the maximal length of a chain to 65536. As a consequence, after at most 65535 consecutive small nodes an "artificial" large node is introduced, for which we store a large record. In this way, we delimit the distance value to be at most 65535, and thus the distance occupies 16 bits, which are stored with the two integers occupied by a small record. Thus we trade a delimited distance value for the saving of one integer for each small record.

Now let us consider how to store the values of a large record. The first two integers of a large record store the *firstchild* reference and the *rightbrother* reference, as in a small record. We need just one extra integer to store the remaining values of a large record: Consider some large node, say $\overline{w}$, and let $\overline{v}$ be the rightmost child of $\overline{w}$. There is a sequence consisting of one *firstchild* reference and at most $k-1$ *rightbrother*/T_{leaf} references which link $\overline{w}$ to $\overline{v}$. If $\overline{v} = \overline{S_j}$ for some $j \in [1, n+1]$, then $T_{leaf}[j]$ is a nil reference. Otherwise, if $\overline{v}$ is a branching node, then *rightbrother*$(\overline{v})$ is a nil reference. Of course, it only requires one bit to mark a reference as a nil reference. Hence the integer used for the nil reference contains unused bits, in which we store *suffixlink*$(\overline{w})$. As a consequence, retrieving the suffix link of $\overline{w}$ requires traversing the list of successors of $\overline{w}$ until the nil reference is reached, which encodes the suffix link of $\overline{w}$. This *linear retrieval* of suffix links takes $O(k)$ time in the worst case. However, despite linear retrieval, the suffix tree can still be constructed in $O(kn)$ time, since suffix links are retrieved at most n times during suffix tree construction (see [McCreight, 1976, Kurtz, 1998]).

Experiments show that linear retrieval may slow down suffix tree construction in practice. For this reason, we use the following method which makes linear retrieval of suffix links an exception: Whenever the depth of a large node does not exceed $2^{11} - 1 = 2047$, we mark this fact and use the remaining bits of the corresponding large record to also store the suffix link. This can later be retrieved in constant time. For those large nodes whose depth exceeds 2047, linear traversal of suffix links is required. But those nodes are usually very rare, and if they occur, then the number of their successors is expected to be small. Hence the linear retrieval of suffix links is expected to be fast.

A small record stores two references ($2 \cdot 23$ bits), a distance value (16 bits), one *small/large bit* to mark whether the first integer is part of a small or a large record, and one *nil bit* to mark a reference as a nil reference. Altogether, a small record occupies 64 bits which fit into two integers. A large record, say for a large node $\overline{w}$, stores two references, one nil bit, one small/large bit, and one *small depth bit* which tells whether the depth is at most $2^{11}-1$. Moreover, there are 21 bits required for the head position, and 11 or 21 bits for the depth, depending on whether the small depth bit is set or not. Thus a large record requires 81 or 91 bits, which fit into three integers. If the depth of $\overline{w}$ is at most $2^{11} - 1$, there are 15 unused bits in the large record. These are used to store the suffix link. The remaining 8 bits of the suffix link for $\overline{w}$ are stored in the integer $T_{leaf}[headposition(\overline{w})]$. Recall that this stores a reference (23 bits) and one nil bit.

Let σ be the number of small records and λ be the number of large records. Thus table T_{branch} requires $2\sigma + 3\lambda$ integers. Table T_{leaf} occupies n integers, and hence the space requirement of our implementation technique is $n+2\sigma+3\lambda$ integers. The implementation technique of [Kurtz, 1998] requires $n + 2\sigma + 4\lambda$ integers (for $n \leq 2^{27} - 1$), while a previous implementation technique (see [McCreight, 1976]) requires $2n + 5(\sigma + \lambda)$ integers. In the worst case $\lambda = n$ and $\sigma = 0$.

The proposed suffix tree representation can be constructed in linear time, using the algorithm of [McCreight, 1976]. The basic observation is that this algorithm constructs the branching nodes of ST in order of their head positions, which is compatible with our implementation technique. For details, see [Kurtz, 1998].

An alternative representation of the suffix tree uses a hash table to store the edges, as recommended in [McCreight, 1976]. Unfortunately, this representation does not directly allow the depth first traversal to run in linear time. As already remarked in [Larsson, 1998], an additional step is required to sort the edges lexicographically. This can be done by a bucket sorting algorithm, and thus requires linear time. In [Kurtz, 1998] it is shown that in practice this approach requires about 60% more space than the proposed linked list implementation, and it leads to a faster sorting procedure only if the alphabet is very large.

1.4 DEPTH FIRST TRAVERSAL

Due to the one-to-one correspondence between the leaves of ST and the non-empty suffixes of $x\$$, the BWT can be read from ST by a simple depth first traversal. This processes the edges outgoing from some branching node $\overline{w}$ in order $<_{\overline{w}}$ which is defined by $\overline{w} \xrightarrow{au} \overline{wau} \quad <_{\overline{w}} \quad \overline{w} \xrightarrow{cv} \overline{wcv} \iff a < c$. It is obvious that such a depth first traversal visits leaf $\overline{S_i}$ before leaf $\overline{S_j}$ if and only if $S_i < S_j$. Thus the suffix order $\varphi(1), \varphi(2), \ldots, \varphi(n+1)$ on $x\$$ is just the list of suffix numbers encountered at the leaves during the traversal. The linked list implementation of Section 1.3 allows the depth first traversal to run in $O(n)$ time. The only extra space required is for a stack storing references to the predecessors of a branching node. The stack occupies at most $r_{\max}$ integers where $r_{\max}$ is the length of the longest repeated substring of x.

The depth first traversal constructs $\widetilde{x}$ from left to right. Whenever it visits a leaf $\overline{S_j}$, $j > 1$, it has found the next character x_{j-1} of $\widetilde{x}$. It stores this character and proceeds with the right brother of $\overline{S_j}$ (if it exists). Thus x_{j-1} is accessed immediately before $T_{leaf}[j]$. Now recall that the integer $T_{leaf}[j]$ stores a reference and a nil bit, occupying 24 bits together. The 8 bits storing a part of the suffix link of the father (if this is a large node and $\overline{S_j}$ is the rightmost child) are not needed during the depth first traversal. For this reason, we store character x_{j-1} (which occupies 8 bits) in the unused bits of $T_{leaf}[j]$. This can be done very efficiently in one sweep over x and T_{leaf} before the depth first traversal. As a consequence, x is no longer accessed in a "random" fashion, which improves the cache coherence of the program and therefore its running time in practice. Moreover, during the traversal the space for the input string x can be reclaimed to store $\widetilde{x}$.

1.5 EXPERIMENTAL RESULTS

We used the programming language C to implement the techniques proposed here. The resulting program computes the BWT, and is referred to by *stbwt*. In order to compare *stbwt* with the Manber-Myers and the Benson-Sedgewick algorithm, we modified the original code of [Manber and Myers, 1993] and [Bentley and Sedgewick, 1997], since these only compute the suffix order. The program derived from [Manber and Myers, 1993], referred to by *mamy*, requires $8n$ bytes. We developed two programs based on [Bentley and Sedgewick, 1997]: *bese1* applies the Benson-Sedgewick algorithm to all suffixes of the input string. It requires $4n$ bytes. *bese2* first uses bucket sort to presort all suffixes according to their first $l = \lfloor \log_k n \rfloor$ characters. Then it applies the Benson-Sedgewick algorithm independently to all groups of suffixes whose prefix of length l is identical. This presorting step runs in linear time, but it requires $4n$ extra bytes. Thus the space requirement of *bese2* is $8n$ bytes. Unfortunately, the program of Sadakane is not available, and so we cannot compare it to *stbwt*. However, experiments in [Sadakane, 1998] show that Sadakane's algorithm is on average slightly slower than a suffix tree based method implemented by Larsson.

			mamy	*bese1*	*bese2*	*stbwt*	
file	*length*	*k*	*time*	*time*	*time*	*time*	*space*
bib	111261	81	4.13	0.60	0.49	0.71	8.87
book1	768771	82	35.72	6.08	4.39	8.62	8.92
book2	610856	96	28.93	4.45	3.30	5.67	8.96
geo	102400	256	2.38	0.36	0.30	1.87	6.83
news	377109	98	27.39	2.80	2.24	4.54	8.84
obj1	21504	256	0.39	0.21	0.20	0.11	7.14
obj2	246814	256	10.99	1.56	1.33	2.46	8.80
paper1	53161	95	1.15	0.20	0.17	0.28	9.09
paper2	82199	91	2.45	0.34	0.27	0.51	9.01
pic	513216	159	29.61	190.86	192.18	2.44	8.67
progc	39611	92	0.73	0.15	0.12	0.20	8.93
progl	71646	87	2.32	0.48	0.43	0.34	9.69
progp	49379	89	1.52	0.53	0.50	0.21	9.81
trans	93695	99	6.35	1.03	0.96	0.44	10.06
	3141622		154.04	209.66	206.87	28.40	8.83

Table 1.1 Running times (in seconds) and Space Requirement (bytes/input character)

We applied all four programs to the 14 files of the Calgary Corpus. Table 1.1 shows the lengths and the alphabet sizes of the files and the running times in seconds on a computer with a Pentium MMX Processor (166 MHz, 32 MB RAM). The last column shows the total space requirement for *stbwt* in bytes per input character. In each row, the shortest running time is shown in a grey box. The last row gives the total file length, the total running times, and the average space requirement for *stbwt*. The table shows that *mamy* is the slowest program. Except for the file *pic* it is always considerably slower than the other programs. *bese1* is always slower than *bese2*. Both are faster than *stbwt* for the same 9 files, but the advantage is small (mostly within a factor of two). However, *bese1* and *bese2* are very slow for the file *pic* which contains long repeated substrings. This clearly reveals the poor worst case behavior of the Benson and Sedgewick algorithm. For most files, *stbwt* requires about n bytes more space than *mamy* and *bese2*. For *pic* and *obj1* it requires even less space.

Acknowledgements. We thank Gene Myers for providing a copy of his program code.

References

[Balkenhol and Kurtz, 1998] Balkenhol, B. and Kurtz, S. (1998). Universal Data Compression Based on the Burrows and Wheeler Transformation: Theory and Practice. Technical Report, Sonderforschungsbereich: Diskrete Strukturen in der Mathematik, Universität Bielefeld, 98-069. `http://www.`

mathematik.uni-bielefeld.de/sfb343/preprints/.

[Balkenhol et al., 1999] Balkenhol, B., Kurtz, S., and Shtarkov, Y. (1999). Modification of the Burrows and Wheeler Data Compression Algorithm. In *Proceedings of the IEEE Data Compression Conference, Snowbird, Utah*, pages 188–197. IEEE Computer Society Press.

[Bentley and Sedgewick, 1997] Bentley, J. and Sedgewick, R. (1997). Fast Algorithms for Sorting and Searching Strings. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 360–369. http://www.cs.princeton.edu/~rs/strings/.

[Burrows and Wheeler, 1994] Burrows, M. and Wheeler, D. (1994). A Block-Sorting Lossless Data Compression Algorithm. Research Report 124, Digital Systems Research Center. http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-124.html.

[Farach, 1997] Farach, M. (1997). Optimal Suffix Tree Construction with Large Alphabets. In *Proceedings of the 38th Annual Symposium on the Foundations of Computer Science, FOCS 97*, New York. IEEE Comput. Soc. Press. ftp://cs.rutgers.edu/pub/farach/Suffix.ps.Z.

[Giegerich and Kurtz, 1997] Giegerich, R. and Kurtz, S. (1997). From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction. *Algorithmica*, **19**:331–353.

[Kurtz, 1998] Kurtz, S. (1998). Reducing the Space Requirement of Suffix Trees. Report 98–03, Technische Fakultät, Universität Bielefeld. http://www.TechFak.Uni-Bielefeld.DE/techfak/~kurtz/publications.html.

[Larsson, 1998] Larsson, N. (1998). The Context Trees of Block Sorting Compression. In *Proceedings of the IEEE Data Compression Conference, Snowbird, Utah, March 30 - April 1*, pages 189–198. IEEE Computer Society Press.

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

[McCreight, 1976] McCreight, E. (1976). A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM*, **23**(2):262–272.

[Sadakane, 1998] Sadakane, K. (1998). A Fast Algorithm for Making Suffix Arrays and for Burrows-Wheeler Transformation. In *Proceedings of the IEEE Data Compression Conference, Snowbird, Utah, March 30 - April 1*, pages 129–138. IEEE Computer Society Press.

[Ukkonen, 1995] Ukkonen, E. (1995). On-line Construction of Suffix-Trees. *Algorithmica*, **14**(3).

[Weiner, 1973] Weiner, P. (1973). Linear Pattern Matching Algorithms. In *Proceedings of the 14th IEEE Annual Symposium on Switching and Automata Theory*, pages 1–11, The Univsersity of Iowa.