The Context Trees of Block Sorting Compression

N. Jesper Larsson

Department of Computer Science, Lund University,
Box 118, S-221 00 LUND, Sweden (jesper@dna.lth.se)

ABSTRACT. The *Burrows-Wheeler transform* (BWT) and *block sorting compression* are closely related to the *context trees* of PPM. The usual approach of treating BWT as merely a permutation is not able to fully exploit this relation.

We show that an explicit context tree for BWT can be efficiently generating by taking a subset of the corresponding suffix tree, identify the central problems in exploiting its structure, and trace the influence of the context tree on the common *move-to-front* schemes. We experimentally obtain limits for compression using the constructed trees, and, as an attempt at utilizing the full context tree, present a compression scheme that represents the context tree explicitly as part of the compressed data.

We argue that a conscious treatment of the context tree should be able to achieve the full compression performance of PPM while maintaining the computational efficiency of BWT. Thus, BWT with explicit context trees is a strong candidate for powerful general compression, especially for large data files.

1. Introduction

Block sorting compression was originally presented by Burrows and Wheeler in 1994 [**3**]. The method consists of a transform (henceforth referred to as the Burrows-Wheeler transform, BWT) which reorganizes the input string to concentrate repetitions. The transformed string can then be compressed by a simple locally adaptive statistical compression scheme to yield compression ratios close to the best known modelling schemes.

While BWT may at first glance appear to be a magical new algorithm, Cleary, Teahan, and Witten [**4**] observed that its effect is quite similar to PPM [**4, 5, 11**]. We take that similarity one step further in giving the *context tree*, which is implicit in BWT, a concrete form. We present a computationally efficient method to construct the tree, explore its power of capturing characteristics of the source, identify the central points in using it for compression, and finally suggest a possible direction towards an efficient complete compression algorithm, presenting a description of an experimental program, with preliminary compression results.

This text is organized as follows: first, we describe the BWT transform, including its time complexity, and discuss previous work. Section 3 identifies the implicit context tree of BWT and gives an efficient algorithm to make it explicit. Section 4 reconsiders the common *move-to-front* encoding from the perspective of the context tree. In Section 5 we discuss explicit use of the context tree in BWT compression and exemplify by presenting an experimental algorithm. We conclude that compression using the context tree is a good candidate for achieving the full benefits of tree model methods such as PPM, while maintaining tight complexity bounds.

2. Background

We first present the basics of BWT as a starting point for the following text, as well as discussions of previous work. Although our formulations are somewhat different, the basis of this section is primarily Burrows and Wheeler [**3**].

2.1. Block Sorting Transform. We assume that the last symbol of the input is a special *end-of-file* symbol \$. With this assumption, sorting *cyclic shifts* of the input, as in the common formulation of BWT, is equivalent to sorting *suffixes* of the same string.

Let the input be a string $t = t_1 \dots t_n$ of symbols. We assume that symbols are represented as integers in the range $[\,1, k\,]$. The output of the transform is the pair $(t', \mathbf{i})$ where t' is a string of length n and $\mathbf{i}$ an integer in the range $[\,1, n\,]$. The transform is performed as follows:

1. Sort all the suffixes of t. Represent the sorted sequence as a vector $S = (S_1, \dots, S_n)$ of numbers in the range $[\,1, n\,]$ such that i precedes j in S iff the suffix that begins in position i of t lexicographically precedes that which begins in position j.
2. Let $\mathbf{i}$ be the number such that $S_{\mathbf{i}} = 1$.
3. For $i \in [\,1, n\,]$, let $t'_i = t_{S_i - 1}$, where we define $t_0 = t_n = \$$.

The effect of the transform is that symbols followed by the same substrings in t are placed in consecutive positions in t'. Referring to the suffix following a position in t as the *context* of that position, we can say that the more similar the contexts of two positions, the closer the symbols in those positions in t. (In PPM, the symbols *preceding* a context defines its context. If desired, this can be emulated in BWT, simply by reversing t. However, the difference is normally of no importance.)

Consequently, if t contains repeating patterns, some parts of t'—that originate from similar contexts—comprise only symbols from a small part of the input alphabet. By transferring t' instead of t to the decompressor, we can exploit its regularities efficiently with a simple locally adaptive compression method.

The decompression program needs to reverse the transform to obtain the original string. This remarkably fast and simple procedure can be formulated as follows:

1. For $c \in [\,1, k\,]$, let n_c be the number of occurrences of symbol c in t'.
2. Set $C[\,1\,]$ to 0. For $i = 1, \dots, k-1$, set $C[\,i+1\,]$ to $C[\,i\,] + n_i$.
3. For $i = 1, \dots, n$, set $P\big[C[\,t'_i\,]\big]$ to i and increment $C[\,t'_i\,]$.
4. Set i to $\mathbf{i}$. For $j = 1, \dots, n$, let $t_j = t'_i$ and advance i to $P[\,i\,]$.

2.2. Sorting Algorithms and Time Complexity. The key advantage of BWT compression is its moderate requirements on computational resources compared to methods with similar compression performance. Throughout this work, we make an effort to maintain this advantage and avoid processes that notably increase time or space complexity. We now discuss the time complexity of BWT itself.

The computionally critical part of the transform is the suffix sorting. It is important to note that since the elements to be sorted are strings, comparisons potentially take linear time. Therefore, a normally commendable comparison based algorithm may well require $\Omega(n^2 \log n)$ time, so a specialized method is needed.

Existing BWT implementations typically use ad hoc combinations of sorting algorithms, often paired with a run length encoding scheme to handle common degeneration cases [**3, 7, 12, 14**]. However, as noted by Burrows and Wheeler [**3**], this can be improved upon by building a suffix tree (see Section 3.1), which is then traversed in sorted order and the sorted sequence obtained from the leaves.

The suffix tree implementation yields linear time for the transform. If the alphabet size is too large to be realistically regarded as constant, this time bound can be achieved by representing the edges of the tree with hashing [**10**]. This requires an additional sorting step, which is done in linear time by bucket sorting the edges and rebuilding the tree in sorted order [**1**].

Furthermore, even if hashing can not be used, the recently presented algorithm by Farach [**6**] provides linear time suffix tree construction for all alphabets relevant for BWT. Thus, the time complexity of BWT is deterministically $\Theta(n)$.

2.3. Move-to-front and Related Coding. A large majority of the previous work on BWT relies on *move-to-front coding* to exploit the local repetitiveness of the transformed string [**3, 7, 12, 14**]. The symbols of the input alphabet are placed in a conceptual list, and the position of a symbol in this list, counting from the head, is used to encode the symbol when encountered. Encoded symbols are immediately moved to the head of the list.

This subsidiary transformation of t' produces another string t'' of integers in the range $[\,1, k\,]$, for which the distribution is highly skewed (provided that t is compressible): low numbers are more common than high numbers. Now, t'' can be compressed with simple zero-order statistical compression, such as Huffman or arithmetic coding.

Arnavut and Magliveras [**2**] devised a slightly different technique named *inversion frequencies*. While move-to-front coding replaces each symbol c with the number of *distinct* symbols encoded since the last occurrence of c, inversion-frequency coding replaces c with the *total* number of symbols *greater than* c encoded since the last occurrence of c. The results were shown to be similar to move-to-front coding.

3. Context Trees

We elaborate the properties of the reorganization performed in BWT by relating to context trees, known from PPM (sometimes referred to as context *tries*). The close relation between PPM and BWT was briefly noted by Cleary, Teahan, and Witten [**4**].

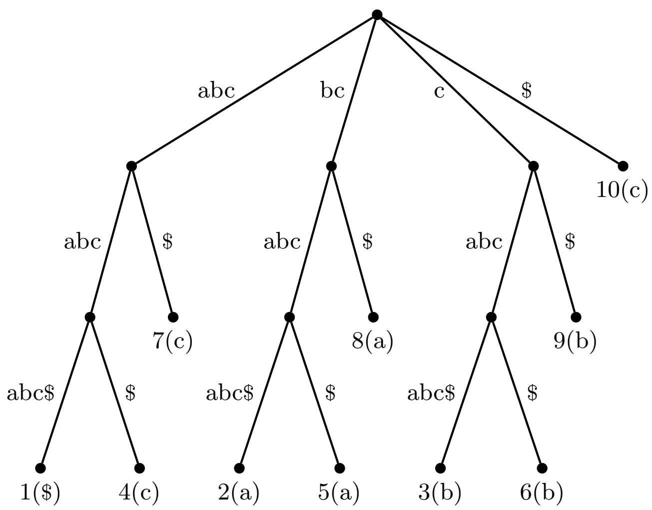

FIGURE 1. The suffix tree of the string 'abcabcabc$'. Below each leaf is shown the number of the suffix it corresponds to and, in parenthesis, the corresponding symbol that would be emitted by the BWT.

3.1. More on Suffix Trees. A context tree can be viewed as a trie (also known as digital tree) storing substrings of the input string t. Edges of the tree are labelled with symbols, and each node (state) corresponds to the string spelled out by the labels on the path from the root to that node. When the trie contains *all* substrings of the input string (as in PPM* [**4**]), and, in addition, is path compressed (i.e. all paths of single-child nodes contracted) this results in what is commonly known as a *suffix tree* [**6, 9, 10, 13**] (see Figure 1).

As noted in Section 2.2, a suffix tree can be used to produce the BWT string t': the tree is traversed left to right, and for each leaf encountered, the symbol preceding the corresponding position of t is emitted as the next symbol of t'. However, the suffix tree is not only a useful tool for the transform, it is also an excellent hierarchical model of similarities between contexts. The leaves of the tree correspond to the contexts. The lowest common ancestor of a pair of nodes, particularly the depth of that ancestor, manifests the similarity between the corresponding pair of contexts, i.e. the length of their common prefix.

For each internal node, we consider the set of frequency counts for the symbols of the input alphabet emitted by the BWT for leaves in its subtree. The root holds the counts for the whole string, which would be used in a simple zero-order encoding, while an internal node corresponding to a string w (where w is the string spelled out by the labels on the path from the root) holds the counts for symbols occurring in the context w. Thus, the suffix tree incorporates exactly the structure of a context tree.[1]

---

[1]Note however, that since our contexts are the strings *after* each position, the tree representation is "backwards" compared to most PPM descriptions.

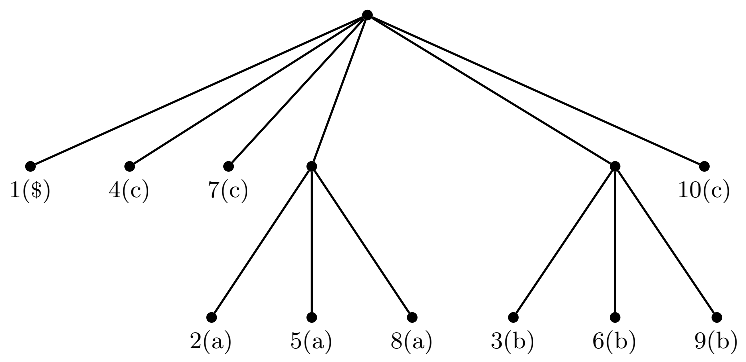

FIGURE 2. A pruned context tree corresponding to the suffix tree in Figure 1.

3.2. Pruning the Tree. Frequency counts in each internal node as described in the previous section means keeping an absolute maximum of statistics about the context properties of the string. This is generally much more than what is actually needed to fully characterize the source.

As an extreme example, consider a single-state source—there is obviously no gain in using more than one set of counts in modelling this. We should recognize this condition, and remove all internal nodes of that context tree, except the root. Generally, we should remove all internal nodes that do not exhibit any significant change in distribution compared to its parent (see Figure 2). Eventually, for large n, the number of internal nodes should converge towards a number that reflects the number of states in a tree model of the source (provided that there is some tree model that captures the source).

To find an approximation of the optimal context tree, we use a greedy method that recursively prunes the tree bottom-up and left-to-right. This has the advantage of being simple and fast, and consuming little space. At each point in time, we only need to maintain frequency counts for nodes on the path from the root to the node currently being processed. This limits space requirements to the height of the tree times the size of the alphabet. We can limit space requirements even further by simply removing all nodes below a certain, constant, depth. This does not notably infer on the final product (it is extremely rare that nodes below a depth of about seven are maintained), but yields an important improvement in worst case space complexity.

In principal, the pruning algorithm works as follows: At each node, we calculate the optimal code length for encoding symbols both including and excluding that node. If keeping the node does not yield a smaller total code length, we remove it.

In addition to maintaining counts over the input alphabet, we also need to take into account the discrepancy in which symbols are used in different subtrees. Again in terms borrowed from PPM, we employ an *escape* mechanism to account for the cost of introducing new events in a state. The first time a symbol occurs, it charges an increase of the escape count instead of the count of the symbol itself.

More specifically, the greedy pruning algorithm prunes the subtree rooted at an internal node u as follows:

1. For each leaf child of u, check which symbol the transform should produce corresponding to that leaf. Then for each symbol c, set $n_{c,u}$ to the number of times c was encountered in this process.
2. Repeat steps 3 to 8 for each internal-node child v of u.
3. Recursively prune the subtree rooted at v.
4. For each symbol c, let $e_c = 1$ if c occurs in the subtree rooted at v, and $e_c = 0$ otherwise. This is to account for escape events in the subtree.
5. Calculate the optimal code length $\mathbf{h}_u$ for encoding, as an independent sequence, the symbols corresponding to leaf children of u, using $n_{c,u} + e_c$ as frequency counts.
6. Analogously calculate the optimal code length $\mathbf{h}_\Sigma$ for encoding symbols in the combination of u and v, using $n_{c,u} + n_{c,v}$ as frequency counts.
7. If $\mathbf{h}_\Sigma < \mathbf{h}_u + \mathbf{h}_v$, then delete v and let all children of v become children of u. Update the $n_{c,u}$ by adding to them their corresponding $n_{c,v}$.
8. Otherwise, update the $n_{c,u}$ by adding to them their corresponding e_c.

Calculation of code lengths is expressed as follows: Let $U = \{\, c \mid n_{c,u} + e_c > 0 \,\}$, and $n_U = \sum_{c \in U} n_{c,u} + e_c$. Summing code length for escapes and symbols in u, we have

$$\begin{aligned}\mathbf{h}_u &= |U| \log \frac{n_U}{|U|} + \sum_{c \in U} (n_{c,u} + e_c - 1) \log \frac{n_U}{n_{c,u} + e_c - 1} \\ &= l(n_U) - l(|U|) - \sum_{c \in U} l(n_{c,u} + e_c - 1),\end{aligned}$$

where $l(n) \equiv n \log n$. The calculation of $\mathbf{h}_\Sigma$ is analogously reduced to a sum of $l(n)$ terms. The function $l(n)$ can be efficiently implemented through a simple halving procedure, which can be speeded up further by a lookup table. We may therefore realistically assume that these calculations are dominated by set operations, which yields a worst case complexity of $O(n \log k)$ for the greedy pruning algorithm (where k is the alphabet size), with a straightforward implementation using (possibly implicit) binary trees.

3.3. Code Length Measurements. To illustrate how the context tree captures the statistics of a file, Table 1 shows experimental results of code length, using the files of the Calgary corpus as input. Note that these are not compression results, since information about the tree structure is not included (see Section 5), but rather lower bounds for what can be achieved by the greedy-pruned context tree.

The measurements show how much redundancy the context tree is able to capture for different kinds of data. Perhaps the most interesting point is that since the pruned context tree approximates the optimal context tree, this can be regarded as an estimate of the actual entropy for the corresponding source, assuming that the tree model assumption holds. Thus, the values are an approximation of the lower bound for *any* tree model based compression method, including PPM, when compressing data with these characteristics.

	size	nodes	bits		size	nodes	bits
bib	111261	10248	1.80	paper1	53161	6265	2.26
book1	768770	59701	2.19	paper2	82199	8338	2.22
book2	610856	50286	1.87	pic	513216	7499	0.78
geo	102400	5442	4.37	progc	39611	4939	2.29
news	377109	40568	2.28	progl	71646	7958	1.59
obj1	21504	1972	3.45	progp	49379	5922	1.62
obj2	246814	27037	2.27	trans	93695	10328	1.42

TABLE 1. Results of the pruning algorithm for the Calgary corpus. *Size* is the original file size in bytes, *nodes* the number of internal nodes maintained by the pruning, and *bits* is the calculated code length in bits per symbol.

4. The Relationship between Move-to-front Encoding and Context Trees

We now review the move-to-front encoding described in Section 2.3 from a context tree perspective, in order to shed light on some important points regarding its performance.

The transform of t' into t'' serves to replace the local repetitions of t' by a globally skewed distribution that would ultimately submit to compression using static frequency counts. However, static encoding is a poor choice. While lower numbers are indeed generally more common than high numbers in t'', their probabilities vary due to the following facts:

- The move-to-front process has no notion of depth changes in the context tree. While BWT places similar contexts close to each other, many not so similar contexts still end up in consecutive positions. The extreme case occurs when all the contexts beginning with a particular symbol are exhausted—the next position corresponds to a completely different context, e.g., a character followed by 'baaa' may be placed directly after a character followed by 'azzz'.
- The degree of regularity varies between contexts. As an example, in English text the characters followed by the string 'the ' are extremely regular (almost all spaces), while the characters followed by ' the' are much less predictable. In information theoretic terms: different states of the source have different entropy. Again, a simple left-to-right view is unable to take context changes into account.

Existing implementations essentially all deal with these inherent disadvantages in the same way: they employ highly adaptive statistics. The simplest method is the common approach of periodically scaling down frequency counts, typically halving them. This gives local probability distributions an advantage over old statistics.

Despite the apparent crudeness of this approach—throwing away large amounts of the collected statistics—it can give quite astonishing results. Fenwick [**7**] reports the same average as the PPM* algorithm [**4**] for the Calgary corpus. The key to this performance lies in the extreme degree of repetition in t' for some files, which produces long runs of zeroes in the move-to-front transform. This is a global property of those files, which remains in spite of the loss of detail in the estimates.

5. Context Tree BWT Compression Schemes

The ultimate goal of our exploration of the BWT context tree is of course to find a competitive compression scheme. However, while the possibilities appear to be immense, it is far from clear what is the best way of exploiting the context tree.

An interesting option, that has a clear potential of competing with move-to-front encoding in computational requirements, is to include a representation of the structural properties of the tree as part of the compressed data, and then encoding the BWT transformed string left-to-right, dynamically updating frequency counts as in PPM. Below we discuss a simple implementation using this strategy. It works well for large files (where the tree representation comprises a small part of the data), but it appears that a more sophisticated tree encoding is required for this method to be a general improvement over move-to-front encoding.

5.1. Further Pruning. When the tree is to be explicitly represented we need to reconsider the pruning strategy. Now, the tree that models the data optimally is not necessarily the best choice, since the size of the tree is a factor. We need to weigh the cost of representing each node against the gain of utilizing that node.

Consequently, the pruning algorithm should be modified so that it maintains a node only if the gain in code length is larger than the cost of representing that node. However, the cost of representing a node is not easily predicted. It depends, naturally, on our choice of representation of the tree, but also on the structure of the whole tree. Our experimental algorithm employs the simplest possible strategy: the cost of representing each node is estimated as a constant, whose value is empirically determined. Furthermore, we impose a lower limit on the number of leaves in a subtree; all nodes with less than some constant number of nodes are removed.

5.2. Encoding the Tree. A pruned context tree has properties which makes it highly compressible. One quickly noted attribute that is easy to take advantage of, is that a large majority of the nodes are leaves. Less obviously exploitable are the structural repetitions in the tree: small subtrees are essentially copies of larger subtrees with some nodes removed.

In the current implementation we use the following simplistic encoding method: we traverse the tree in order, obtaining the number of children of each node. These numbers are encoded as exponent-mantissa pairs, where the exponents are compressed with a first-order arithmetic encoder whose state is based on the size of the parent.

5.3. Encoding the Symbols. In encoding the symbols corresponding to the leaves of the tree, we have to choose a strategy for transferring the frequency counts to the decoder. One possibility is to encode them explicitly, as we do the structure of the tree. Another, which we have chosen in the current implementation, is to use the tree only for state selection and encode new symbols by escaping to shorter contexts, as in PPM compression.

The crucial difference compared to PPM is that of computational efficiency and simplicity: Since we encode left-to-right in the tree, we only need to maintain frequency count for one branch of the tree at a time. Furthermore, escaping to a

	size	nodes	bits (tree + sym)		size	nodes	bits (tree + sym)
bib	111261	2308	2.26 (0.28 + 1.98)	paper1	53161	1384	2.84 (0.35 + 2.49)
book1	768770	7777	2.37 (0.15 + 2.22)	paper2	82199	1634	2.65 (0.28 + 2.37)
book2	610856	8793	2.17 (0.21 + 1.96)	pic	513216	652	0.80 (0.02 + 0.78)
geo	102400	899	4.69 (0.13 + 4.56)	progc	39611	1071	2.92 (0.36 + 2.56)
news	377109	7350	2.76 (0.26 + 2.50)	progl	71646	1778	2.13 (0.33 + 1.80)
obj1	21504	516	4.27 (0.32 + 3.95)	progp	49379	1256	2.22 (0.34 + 1.88)
obj2	246814	6303	2.94 (0.33 + 2.61)	trans	93695	2775	2.06 (0.37 + 1.69)

TABLE 2. Results of the experimental compression program. *Size* is the original file size in bytes, and *nodes* the number of internal nodes maintained. *Bits* is the average number of bits per compressed symbol, divided into *tree* and *sym* (symbol) to show the relative code space for tree encoding used by the program for these files.

shorter context is simple, since the shorter context is the *parent* of each node—we do not need the suffix links, or escape lists, of PPM implementations.

Now, we have the same choices as in PPM regarding strategies of escape probability estimation, inheritance, exclusion etc. Again because the tree is traversed in order, most conceivable choices are easily and efficiently implemented, which opens extensive possibilities for refinement. Our current implementation uses no inheritance, an escape estimate similar to PPMD [**8**], full exclusion, and update exclusion.

5.4. Preliminary Results. Table 2 shows the current results of our experimental compression program. The limits chosen for the pruning algorithm were five bits as the minimum gain to retain a node, and a minimum of eight leaves for subtrees rooted at internal nodes.

It is clear from the table that for these files our current experimental program is no general improvement over the best known BWT implementations—only the largest file, book1, yields a total improvement over the move-to-front results achieved by Fenwick [**7**]. In particular, the the tree encoding scheme must be improved in order to achieve favourable compression ratios for files as small as these (although for a few files, Fenwick's implementation performs better even disregarding the *tree* part). The small number of internal nodes retained by the pruning indicates that this improvement should certainly be possible through a more sophisticated tree encoding.

For very large files, the representation of the tree should eventually be negligible, provided that the source can be represented as a tree model, and that the number of internal nodes of the context tree converges towards a constant that reflects the number of states in this model (see Section 3.2).

6. Final Comments

Data compression using BWT has an advantage over other tree model based methods in its moderate requirements on computational resources. We assert that this advantage can be maintained with a much more sophisticated modelling method than the move-to-front transform. Our context tree approach reveals the possibility of using BWT to obtain a tight time complexity while taking advantage of sophisticated techniques developed for PPM.

However, finding the optimal combination of these two approaches remains as an open problem. Particularly, if the approach of representing the context tree explicitly is used, the structure of the tree must be further analyzed, and a sophisticated encoding scheme designed, if the method is to be competitive for small files.

It should be noted that while we have approached the context trees of BWT with suffix trees as a starting point, the process of pruning the suffix tree to obtain a useful context tree is by no means the only possibility. On the contrary, the small number of internal nodes maintained by the extended pruning indicates that a top-down method of constructing the tree (which could be made to consume less memory) should certainly be considered. This is particularly the case when large blocks of data are treated, since the suffix tree may then require considerable (although linear) storage space.

In conclusion, we conjecture that BWT compression with an explicit treatment of context trees is a strong candidate for achieving the full compression performance of PPM and similar methods, with a program that requires only moderate computational resources.

References

1. Arne Andersson, N. Jesper Larsson, and Kurt Swanson, *Suffix trees on words*, Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching, Lecture Notes in Computer Science, vol. 1075, Springer-Verlag, June 1996, pp. 102–115.
2. Ziya Arnavut and Spyros S. Magliveras, *Block sorting and compression*, Proceedings of the IEEE Data Compression Conference, March 1997, pp. 181–190.
3. Michael Burrows and David J. Wheeler, *A block-sorting lossless data compression algorithm*, Research Report. 124, Digital Systems Research Center, Palo Alto, California, May 1994.
4. John G. Cleary, W. J. Teahan, and Ian H. Witten, *Unbounded length contexts for PPM*, Proceedings of the IEEE Data Compression Conference, March 1995, pp. 52–61.
5. John G. Cleary and Ian H. Witten, *Data compression using adaptive coding and partial string matching*, IEEE Transactions on Communications **COM-32** (1984), 396–402.
6. Martin Farach, *Optimal suffix tree construction with large alphabets*, Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science, October 1997, pp. 137–143.
7. Peter Fenwick, *Block sorting text compression*, Proceedings of the 19th Australasian Computer Science Conference (Melbourne, Australia), January–February 1996.
8. Paul Glor Howard, *The design and analysis of efficient lossless data compression systems*, Ph.D. thesis, Deptartment of Computer Science, Brown University, Providence, Rhode Island, June 1993, CS-93-28.
9. N. Jesper Larsson, *Extended application of suffix trees to data compression*, Proceedings of the IEEE Data Compression Conference, March–April 1996, pp. 190–199.
10. Edward M. McCreight, *A space-economical suffix tree construcion algorithm*, Journal of the ACM **23** (1976), no. 2, 262–272.
11. Alistair Moffat, *Implementing the PPM data compression scheme*, IEEE Transactions on Communications **COM-38** (1990), no. 11, 1917–1921.
12. Julian Seward, *bzip2 program*, `http://www.muraroa.demon.co.uk/`, 1997.
13. Esko Ukkonen, *On-line construction of suffix trees*, Algorithmica **14** (1995), no. 3, 249–260.
14. David Wheeler, *An implementation of block coding*, Tech. report, Cambridge University Computer Laboratory, October 1995.