Structures of String Matching and Data Compression

N. Jesper Larsson

Department of Computer Science
Lund University

Department of Computer Science
Lund University
Box 118
S-221 00 Lund
Sweden

Copyright © 1999 by Jesper Larsson
CODEN:LUNFD6/(NFCS-1015)/1–130/(1999)
ISBN 91-628-3685-4

Abstract

This doctoral dissertation presents a range of results concerning efficient algorithms and data structures for string processing, including several schemes contributing to sequential data compression. It comprises both theoretic results and practical implementations.

We study the *suffix tree* data structure, presenting an efficient representation and several generalizations. This includes augmenting the suffix tree to fully support *sliding window* indexing (including a practical implementation) in linear time. Furthermore, we consider a variant that indexes naturally *word-partitioned* data, and present a linear-time construction algorithm for a tree that represents only suffixes starting at word boundaries, requiring space linear in the number of words.

By applying our sliding window indexing techniques, we achieve an efficient implementation for dictionary-based compression based on the LZ-77 algorithm. Furthermore, considering predictive source modelling, we show that a PPM* style model can be maintained in linear time using arbitrarily bounded storage space.

We also consider the related problem of *suffix sorting*, applicable to *suffix array* construction and *block sorting compression*. We present an algorithm that eliminates superfluous processing of previous solutions while maintaining robust worst-case behaviour. We experimentally show favourable performance for a wide range of natural and degenerate inputs, and present a complete implementation.

Block sorting compression using *BWT*, the *Burrows-Wheeler transform*, has implicit structure closely related to *context trees* used in predictive modelling. We show how an explicit BWT context tree can be efficiently generated as a subset of the corresponding suffix tree and explore the central problems in using this structure. We experimentally evaluate prediction capabilities of the tree and consider representing it explicitly as part of the compressed data, arguing that a conscious treatment of the context tree can combine the compression performance of predictive modelling with the computational efficiency of BWT.

Finally, we explore *offline dictionary-based compression*, and present a semi-static source modelling scheme that obtains excellent compression, yet is also capable of high decoding rates. The amount of memory used by the decoder is flexible, and the compressed data has the potential of supporting direct search operations.

Between theory and practice, some talk as if they were two – making a separation and difference between them. Yet wise men know that both can be gained in applying oneself whole-heartedly to one.

Bhagavad-Gītā 5:4

Short-sighted programming can fail to improve the quality of life. It can reduce it, causing economic loss or even physical harm. In a few extreme cases, bad programming practice can lead to death.

P. J. Plauger,
Computer Language, Dec. 1990

Contents

Foreword 7

Chapter One Fundamentals 9
1.1 Basic Definitions 10
1.2 Trie Storage Considerations 12
1.3 Suffix Trees 13
1.4 Sequential Data Compression 19

Chapter Two Sliding Window Indexing 21
2.1 Suffix Tree Construction 22
2.2 Sliding the Window 24
2.3 Storage Issues and Final Result 32

Chapter Three Indexing Word-Partitioned Data 33
3.1 Definitions 34
3.2 Wasting Space: Algorithm A 36
3.3 Saving Space: Algorithm B 36
3.4 Extensions and Variations 40
3.5 Sublinear Construction: Algorithm C 41
3.6 Additional Notes on Practice 45

Chapter Four Suffix Sorting 48
4.1 Background 50
4.2 A Faster Suffix Sort 52
4.3 Time Complexity 56
4.4 Algorithm Refinements 59
4.5 Implementation and Experiments 63

Chapter Five Suffix Tree Source Models 71
5.1 Ziv-Lempel Model 71
5.2 Predictive Modelling 73
5.3 Suffix Tree PPM* Model 74
5.4 Finite PPM* Model 76
5.5 Non-Structural Operations 76
5.6 Conclusions 78

Chapter Six Burrows-Wheeler Context Trees 79
6.1 Background 80
6.2 Context Trees 82
6.3 The Relationship between Move-to-front Coding and Context Trees 86
6.4 Context Tree BWT Compression Schemes 87
6.5 Final Comments 89

Chapter Seven Semi-Static Dictionary Model 91
7.1 Previous Approaches 93
7.2 Recursive Pairing 94
7.3 Implementation 95
7.4 Compression Effectiveness 101
7.5 Encoding the Dictionary 102
7.6 Tradeoffs 105
7.7 Experimental Results 106
7.8 Future Work 110

Appendix A Sliding Window Suffix Tree Implementation 111

Appendix B Suffix Sorting Implementation 119

Appendix C Notation 125

Bibliography 127

Foreword

Originally, my motivation for studying computer science was most likely spawned by a calculator I bought fourteen years ago. This gadget could store a short sequence of operations, including a conditional jump to the start, which made it possible to program surprisingly intricate computations. I soon realized that this simple mechanism had the power to replace the tedious repeated calculations I so detested with an intellectual exercise: to find a general method to solve a specific problem (something I would later learn to refer to as an *algorithm*) that could be expressed by pressing a sequence of calculator keys. My fascination for this process still remains.

With more powerful computers, programming is easier, and more challenging problems are needed to keep the process interesting. Ultimately, in algorithm theory, the bothers of producing an actual program are completely skipped over. Instead, the final product is an explanation of how an idealized machine could be programmed to solve a problem efficiently. In this abstract world, program elements are represented as mathematical objects that interact as if they were physical. They can be chained together, piled on top of each other, or linked together to any level of complexity. Without these *data structures*, which can be combined into specialized tools for solving the problem at hand, producing large or complicated programs would be infeasible. However, they do not exist any further than in the programmer's mind; when the program is to be written, everything must again be translated into more basic operations. In my research, I have

tried to maintain this connection, seeing algorithm theory not merely as mathematics, but ultimately as a programming tool.

At a low level, computers represent everything as sequences of numbers, albeit with different interpretations depending on the context. The main topic in this thesis is algorithms and data structures – most often tree shaped structures – for finding patterns and repetitions in long sequences, *strings*, of similar items. Examples of typical strings are texts (strings of letters and punctuation marks), programs (strings of operations), and genetic data (strings of amino acids). Even two-dimensional data, such as pictures, are represented as strings at a lower level. One area particularly explored in the thesis is storing strings compactly, *compressing* them, by recording repetition and systematically introducing abbreviations for repeating patterns.

The result is a collection of methods for organizing, searching, and compressing data. Its creation has deepened my insights in computer science enormously, and I hope some of it can make a lasting contribution to the computing world as well.

Numerous people have influenced this work. Obviously, my coauthors for different parts of the thesis, Arne Andersson, Alistair Moffat, Kunihiko Sadakane, and Kurt Swanson, have had a direct part in its creation, but many others have contributed in a variety of ways. Without attempting to name them all, I would like to express my gratitude to all the central and peripheral members of the global research community who have supported and assisted me.

The influence of my advisor Arne Andersson goes beyond the work where he stands as an author. He brought me into the research community from his special angle, and imprinted me with his views and visions. His notions of what is relevant research, and how it should be presented, have guided me through these last five years.

Finally, I wish to specifically thank Alistair Moffat for inviting me to Melbourne and collaborating with me for three months, during which time I was accepted as a full member of his dynamic research group. This gave me a new perspective, and a significant push towards completing the thesis.

Malmö, August 1999
Jesper Larsson

Chapter One

Fundamentals

The main theme of this work is the organization of sequential data to find and exploit patterns and regularities. This chapter defines basic concepts, formulates fundamental observations and theorems, and presents an efficient suffix tree representation. Following chapters frequently refer and relate to the material given in this chapter.

The material and much of the text in this current work is taken primarily from the following five previously presented writings:

- *Extended Application of Suffix Trees to Data Compression*, presented at the IEEE Data Compression Conference 1996 [42]. A revised and updated version of this material is laid out in chapters two and five, and to some extent in §1.3.
- *Suffix Trees on Words*, written in collaboration with Arne Andersson and Kurt Swanson, published in *Algorithmica*, March 1998 [4]. A preliminary version was presented at the seventh Annual Symposium on Combinatorial Pattern Matching in June 1996. This is presented in chapter three, with some of the preliminaries given in §1.2.
- *The Context Trees of Block Sorting Compression*, presented at the IEEE Data Compression Conference 1998 [43]. This is the basis of chapter six.
- *Offline Dictionary-Based Compression*, written with Alistair Moffat of the University of Melbourne, presented at the IEEE Data Compression Conference 1999 [44]. An extended version of this work is presented in chapter seven.

- *Faster Suffix Sorting*, written with Kunihiko Sadakane of the University of Tokyo; technical report, submitted [45]. This work is reported in chapter four. Some of its material has been presented in a preliminary version as *A Fast Algorithm for Making Suffix Arrays and for Burrows-Wheeler Transformation* by Kunihiko Sadakane [59].

1.1 Basic Definitions

We assume that the reader is familiar with basic conventional definitions regarding strings and graphs, and do not attempt to completely define all the concepts used. However, to resolve differences in the literature concerning the use of some concepts, we state the definitions of not only our specialized concepts, but also of some more general ones.

For quick reference to our specialized notations, appendix C on pages 125–126 summarizes terms and symbols used in each of the chapters of the thesis.

Notational Convention For notation regarding asymptotic growth of functions and similar concepts, we adopt the general tradition in computer science; see, for instance, Cormen, Leiserson, and Rivest [20].

All logarithms in the thesis are assumed to be base two, except where otherwise stated.

1.1.1 *Symbols and Strings* The input of each of the algorithms described in this thesis is a sequence of items which we refer to as *symbols*. The interpretation of these symbols as letters, program instructions, amino acids, etc., is generally beyond our scope. We treat a symbol as an abstract element that can represent any kind of unit in the actual implementation – although we do provide several examples of practical uses, and often aim our efforts at a particular area of application.

Two basic sets of operations for symbols are common. Either the symbols are considered *atomic* – indivisible units subject to only a few predefined operations, of which pairwise comparison is a common example – or they are assumed to be represented by *integers*, and thereby possible to manipulate with all the common arithmetic operations. We adopt predominantly the latter approach, since our primary goal is to develop practically useful tools, and in present computers everything is always, at the lowest level, represented as integers. Thus, restricting allowed operations beyond the set of arithmetic ones often introduces an unrealistic impediment.

We denote the size of the *input alphabet*, the set of possible values

of input symbols, by k. When the symbols are regarded as integers, the input alphabet is $\{1, \ldots, k\}$ except where otherwise stated.

Consider a string $\alpha = a_1 \ldots a_N$ of symbols a_i. We denote the length of α by $|\alpha| = N$. The substrings of α are $a_i \ldots a_j$ for $1 \leq i \leq N$ and $i - 1 \leq j \leq N$, where the string $a_i \ldots a_{i-1}$ is the *empty string*, denoted ϵ. The *prefixes* of α are the $N + 1$ strings $a_1 \ldots a_i$ for $0 \leq i \leq N$. Analogously, the *suffixes* of α are $a_i \ldots a_N$ for $1 \leq i \leq N + 1$.

With the exception of chapters two and five, where the input is potentially a continuous and infinite stream of symbols, the input is regarded as a fixed string of n symbols, appended with a unique terminator symbol \$, which is not regarded as part of the input alphabet except where stated. This special symbol can sometimes be represented as an actual value in the implementation, but may also be implicit. If it needs to be regarded as numeric, we normally assign it the value 0.

We denote the input string X. Normally, we consider this a finite string and denote $X = x_0 x_1 \ldots x_n$, where n is the size of the input, $x_n = \$$, and x_i, for $0 \leq i < n$, are symbols of the input alphabet.

Trees and Tries We consider only rooted trees. Trees are visualized with the root at the top, and the children of each node residing just below their parent. A node with at least one child is an *internal node*; a node without children is a *leaf*. The *depth* of a node is the number of nodes on the path from the root to that node. The maximum depth in a tree is its *height*. 1.1.2

A *trie* is a tree that represents strings of symbols along paths starting at the root. Each edge is labeled with a nonempty string of symbols, and each node corresponds to the concatenated string spelled out by the symbols on the path from the root to that node. The root represents ϵ. For each string contained in a trie, the trie also inevitably contains all prefixes of that string. (This data structure is sometimes referred to as a *digital tree*. In this work, we make no distinction between the concepts *trie* and *digital tree*.)

A trie is *path compressed* if all paths of single-child nodes are contracted, so that all internal nodes, except possibly the root, have at least two children. The path compressed trie has the minimum number of nodes among the tries representing a certain set of strings; a string α contained in this trie corresponds to an explicit node if and only if the trie contains two strings αa and αb, for distinct symbols a and b. The length of a string corresponding to a node is the *string depth* of that node.

Henceforth, we assume that all tries are either path compressed or that their edges are all labeled with single symbols only (in which case

depth and *string depth* are equivalent), except possibly during transitional stages.

A *lexicographic trie* is a trie for which the strings represented by the leaves appear in lexicographical order in an in-order traversal. A *non-lexicographic* trie is not guaranteed to have this property.

1.2 Trie Storage Considerations

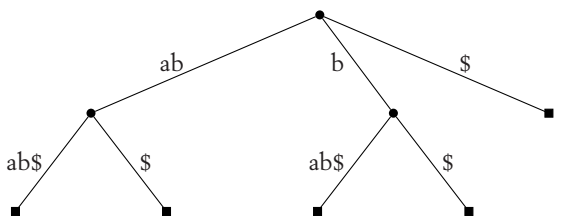
The importance of the trie data structure lies primarily in the ease at which it allows searching among the contained strings. To locate a string, we start at the root of the trie and the beginning of the string, and scan downwards, matching the string against edge labels, until a leaf is reached or a mismatch occurs. This takes time proportional to the length of the matched part of the search, plus the time to choose edges along the search path. The choice of edges is the critical part of this process, and its efficiency depends on what basic data structures are used to store the edges.

When choosing a trie implementation, it is important to be aware of which types of queries are expected. The ordering of the nodes is one important concept. Maintaining a lexicographic trie may be useful in some applications, e.g. to facilitate neighbour and range search operations. Note, however, that in many applications the alphabet is merely an arbitrarily chosen enumeration of unit entities with no tangible interpretation of *range* or *neighbour*, in which case a lexicographic trie has no advantage over its non-lexicographic counterpart.

Because of the characteristics of different applications, it is sometimes necessary to discuss several versions of tries. We note specifically the following possibilities:

1. Each node can be implemented as an array of size k. This allows fast searches, but for large alphabets it consumes a lot of space and makes efficient initialization of new nodes complex.
2. Each node can be implemented as a linked list or, for instance, as a binary search tree. This saves space at the price of a higher search cost, when the alphabet is not small enough to be regarded as constant.
3. The edges can be stored in a hash table, or alternatively, a separate hash table can be stored for each node. Using dynamic perfect hashing [22], we are guaranteed that searches spend constant time per node, even for a non-constant alphabet. Furthermore, this representation may be combined with variant 2.

An important fact is that a non-lexicographic trie can be made lexicographic at low cost by sorting all edges according to the first symbol of each edge label, and then rebuilding the tree in the sorted order.



Suffix tree for the string 'abab$'.

We state this formally for reference in later chapters:

Observation A non-lexicographic trie with l leaves can be transformed into a lexicographic one in time $O(l + s(l))$, where $s(l)$ is the time required to sort l symbols. *1A*

1.3 Suffix Trees

A *suffix tree* (also known as *position tree* or *subword tree*) of a string is a path compressed trie holding all the suffixes of that string – and thereby also all other substrings. This powerful data structure appears frequently throughout the thesis.

The tree has $n + 1$ leaves, one for each nonempty suffix of the \$-terminated input string. Therefore, since each internal node has at least two outgoing edges, the number of nodes is at most $2n + 1$. In order to ensure that each node takes constant storage space, an edge label is represented by pointers into the original string. A sample suffix tree indexing the string 'abab\$' is shown above.

The most apparent use of the suffix tree is as an index that allows substrings of a longer string to be located efficiently. The suffix tree can be constructed, and the longest substring that matches a search string located, in asymptotically optimal time. Under common circumstances this means that construction takes linear time in the length of the indexed string, the required storage space is also linear in the length of the indexed string, and searching time is linear in the length of the matched string.

An alternative to the suffix tree is the *suffix array* [47] (also known as *PAT array* [28]), a data structure that supports some of the operations of a suffix tree, generally slower but requiring less space. When additional space is allocated to supply a *bucket* array or a *longest common prefix* array, the time complexities of basic operations closely approach those of the suffix tree. Construction of a suffix array is equivalent to *suffix sorting*, which we discuss in chapter four

1.3.1 *Construction Algorithms* Weiner [68] presented the first linear time suffix tree construction algorithm. Shortly thereafter, McCreight [48] gave a simpler and less space consuming version, which became the standard. Also notable is Ukkonen's construction algorithm [67], the most comprehensible *online* suffix tree construction algorithm. The significance of this is explained in chapter two, which also presents a full description of Ukkonen's algorithm, with extensions.

The three mentioned algorithms have substantial similarities. They all achieve linear time complexity through the use of *suffix links*, additional backwards pointers in the tree that assist in navigation between internal nodes. The suffix link of a node representing the string $c\alpha$, where c is a symbol and α a string, points to the node representing α.

Furthermore, these algorithms allow linear time construction only under the assumption that the choice of an outgoing edge to match a certain symbol can be determined in amortized constant time. The time for this access operation is a factor in construction time complexity. We state this formally:

1B *Theorem* (Weiner) A suffix tree for a string of length n in an alphabet of size k can be constructed in $O(n\,i(k))$ time, where $i(k)$ bounds the time to locate a symbol among k possible choices.

This bound follows immediately from the analysis of any of the mentioned construction algorithms. Thus, these algorithms take linear time when the input alphabet is small enough to be regarded as a constant or – if a randomized worst case bound is sufficient – when hash coding is used to store the edges.

When hash coding is used, the resulting tree is non-lexicographic. Most of the work done on suffix tree construction seems to assume that a suffix tree should be lexicographic. However, it appears that the majority of the applications of suffix trees, for example all those discussed by Apostolico [6], do not require a lexicographic trie, and indeed McCreight asserts that hash coding appears to be the best representation [48, page 268]. Furthermore, once the tree is constructed it can always be made lexicographic in asymptotically optimal time by observation 1A.

Farach [23] took a completely new approach to suffix tree construction. His algorithm recursively constructs the suffix trees for odd- and even-numbered positions of the indexed string and merges them together. Although this algorithm has not yet found broad use in implementations, it has an important implication on the complexity of the problem of suffix tree construction. Its time bound does not depend on the input alphabet size, other than requiring that the input

is represented as integers bounded by n. Generally, this is formulated as follows:

Theorem (Farach) A lexicographic suffix tree for a string of length n can be constructed in $O(n + s(n))$ time, where $s(n)$ bounds the time to sort n symbols. *1C*

This immediately gives us the following corollary:

Corollary The time complexity for construction of a lexicographic suffix tree for a string of length n is $\Theta(n + s(n))$, where $s(n)$ is the time complexity of sorting n symbols. *1D*

Proof The upper bound is given by theorem 1C. The lower bound follows from the fact that in a lexicographic suffix tree, the sorted order for the symbols of the string can be obtained by a linear scan through the children of the root. □

1.3.2 *Suffix Tree Representation and Notation*

The details of the suffix tree representation deserves some attention. Choice of representation has a considerable effect on the amount of storage required for the tree. It also influences algorithms that construct or access the tree, since different representations require different access methods.

We present a suffix tree representation designed primarily to be compact in the worst case. We use this representation directly in chapter two, and in the implementation in appendix A. It is to be regarded as our basic choice of implementation except where otherwise stated. We use hashing to store edges, implying randomized worst case time when it is used. The notation used for our representation is summarized in the table on the next page.

In order to express tree locations of strings that do not have a corresponding node in the suffix tree, due to path compression, we introduce the following concept:

Definition For each substring α of the indexed string, *point*(α) is a triple (u, d, c), where u is the node of maximum depth that represents a prefix of α, β is that prefix, $d = |\alpha| - |\beta|$, and c is the $|\beta| + 1$st symbol of α, unless $\alpha = \beta$ in which case c can be any symbol. *1E*

Less formally: if we traverse the tree from the root following edges that together spell out α for as long as possible, u is the last node on that path, d is the number of remaining symbols of α below u, and c is the first symbol on the edge label that spells out the last part of α, i.e., c determines on which outgoing edge of u the point is located. For an illustration, consider the figure on page 17, where

Summary of suffix tree representation. The values of *leaf*, *spos*, *child*, *parent*, h, and g are computed, the others stored explicitly.

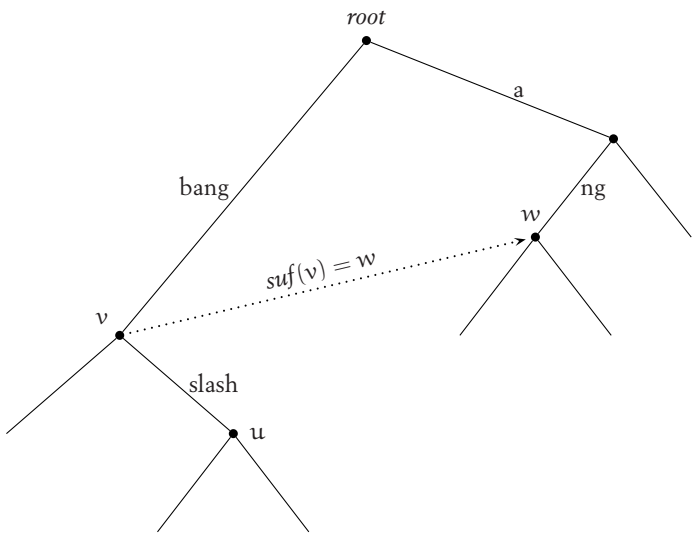
$\mathit{depth}(u)$	String depth of node u, i.e., total number of symbols in edge labels on the path from the root to u; stored explicitly for internal nodes only.
$\mathit{pos}(u)$	Starting position in X of the incoming edge label for node u; stored explicitly for internal nodes only.
$\mathit{fsym}(u)$	First symbol in the incoming edge label of leaf u.
$\mathit{leaf}(i)$	Leaf corresponding to the suffix $x_i \dots x_n$.
$\mathit{spos}(u)$	Starting position in X of the suffix represented by leaf u; $i = \mathit{spos}(u) \Leftrightarrow u = \mathit{leaf}(i)$.
$\mathit{child}(u, c)$	The child node of node u that has an incoming edge label of beginning with symbol c. If u has no such child, $\mathit{child}(u, c) = \mathit{nil}$.
$\mathit{parent}(u)$	The parent node of node u.
$\mathit{suf}(u)$	Node representing the longest proper suffix of the string represented by internal node u (the suffix link target of u).
$h(u, c)$	Hash table entry number for $\mathit{child}(u, c)$.
$g(i, c)$	Backward hash function, $u = g(i, c) \Leftrightarrow i = h(u, c)$.
$\mathit{hash}(i)$	Start of linked list for nodes with hash value i.
$\mathit{next}(u)$	Node following u in the linked list of nodes with equal hash values.

$\mathit{point}(\text{`bangsl'}) = (v, 2, \text{`s'})$.

All nodes are represented by numbers. Internal nodes can have their numbers assigned in any order, but leaves are numbered consecutively according to which suffixes they represent. This gives us constant time access to a leaf node given the starting position, as well as to the starting position given the node. If a leaf node v corresponds to the suffix $x_i \dots x_n$, we denote $\mathit{leaf}(i) = v$ and $\mathit{spos}(v) = i$. For instance, we can number the leaves $l_0, \dots, l_0 + n$ for any l_0, and define $\mathit{leaf}(i)$ to be node number $l_0 + i$.

We adopt the following convention for representing edge labels: each node u in the tree has two associated values $\mathit{pos}(u)$, which denotes a position in X where the label of the incoming edge of u is spelled out; and $\mathit{depth}(u)$, which denotes the string depth of u (the length of its represented string). Hence, the label of an edge (u, v) is the string of length $\mathit{depth}(v) - \mathit{depth}(u)$ that begins at position $\mathit{pos}(v)$ of X. For internal nodes, we store these values explicitly. For leaves, this is not needed, since the values can be obtained from the node numbering: if v is a leaf, the value corresponding to $\mathit{depth}(v)$ is $n + 1 - \mathit{spos}(v)$, and the value of $\mathit{pos}(v)$ is $\mathit{spos}(v) + \mathit{depth}(u)$, where u is the parent of v.

As noted by McCreight [48, page 268] it is possible to avoid storing *pos* values through a similar numbering arrangement for internal nodes as for the leaves, thus saving one integer of storage per internal node. However, we choose not to take advantage of this due to the limita-



Fragment of a suffix tree for a string containing 'bangslash'. In this tree, *point*('bangsl') is $(v, 2, \text{'s'})$, *child*(*root*, 'b') is the node v and *child*(v, 's') is the node u. The dotted line shows the suffix link of v.

tions it imposes on handling of node deletions, which are necessary for the sliding window support treated in chapter two.

By $\mathit{child}(u, c) = v$, and $\mathit{parent}(v) = u$, where u and v are nodes and c is a symbol, we denote that there is an edge (u, v) whose label begins with c.

Associated with each internal node u of the suffix tree, we store a suffix link as described in §1.3.1. We define $\mathit{suf}(u) = v$ if and only if u represents $c\alpha$, for symbol c and string α, and the node v represents α. In the figure above, the node v represents the string 'bang' and w represents 'ang'; consequently, $\mathit{suf}(v) = w$. The suffix links are needed during tree construction but are not generally used once the tree is completed.

For convenience, we add a special node *nil* and define $\mathit{suf}(\mathit{root}) = \mathit{nil}$, $\mathit{parent}(\mathit{root}) = \mathit{nil}$, $\mathit{depth}(\mathit{nil}) = -1$, and $\mathit{child}(\mathit{nil}, c) = \mathit{root}$ for any symbol c. We leave $\mathit{suf}(\mathit{nil})$ and $\mathit{pos}(\mathit{nil})$ undefined, allowing the algorithm to assign these entities any value. Furthermore, for a node u that has no outgoing edge such that its label begins with c, we define $\mathit{child}(u, c) = \mathit{nil}$.

We use a hashing scheme where elements with equal hash values are chained together by singly linked lists. The hash function $h(u, c)$, for internal node u and symbol c produces a number in the range $[0, H)$, where H is the number of entry points in the hash table. We require that a *backward* hash function g is defined so that the node u can be uniquely identified as $u = g(i, c)$, given i and c such that $i = h(u, c)$. For uniqueness, this implies that H is at least $\max\{n, k\}$.

Child retrieval in our edge representation. Numeric values of nodes are defined in the text.

child(u, c):
1 $i \leftarrow h(u, c)$, $v \leftarrow$ *hash*(i).
2 While v is not a list terminator, execute steps 3 to 5:
3 If v is a leaf, $c' \leftarrow$ *fsym*(v); otherwise $c' \leftarrow x_{pos(v)}$.
4 If $c' = c$, stop and return v.
5 $v \leftarrow$ *next*(v) and continue from step 2.
6 Return *nil*.

To represent an edge (u, v) whose edge label begins with symbol c, we insert the node v in the linked list of hash table entry point $h(u, c)$. By *hash*(i) we denote the first node in hash table entry i, and by *next*(u) the node following u in the hash table linked list where it is stored. If there is no node following u, *next*(u) stores a special *list terminator* value. If there is no node with hash value i, *hash*(i) holds the terminator.

Because of the uniqueness property of our hash function, it is not necessary to store any additional record for each item held in the hash table. To determine when the correct child node is found when scanning through a hash table entry, the only additional information needed is the first symbol of the incoming edge label for each node. For an internal node v, this symbol is directly accessible as $x_{pos(v)}$, but for the leaves we need an additional n symbols of storage to access these distinguishing symbols. Hence, we define and maintain *fsym*(v) for each leaf v to hold this value.

The *child*(u, c) algorithm above shows the child retrieval process given the specified storage. Steps 3 and 4 of this algorithm determine if the current v is the correct value of *child*(u, c) by checking if it is consistent with the first symbol in the label of (u, v) being c.

Summing up storage, we have three integers for each internal node, to store the values of *pos*, *depth*, and *suf*, plus the hash table storage which requires $\max\{n, k\}$ integers for *hash* and one integer per node for *next*. In addition, we need to store $n + 1$ symbols to maintain *fsym* and the same amount to store the string X. (For convenience, we store the *nil* node explicitly.) Thus, we can state the following regarding the required storage:

1F *Observation* A suffix tree for a string of n symbols from an alphabet of size k, with an appended end marker, can be constructed in expected linear time using storage for $5(n+1)+\max\{n, k\}$ integers and $2(n+1)$ symbols.

The hash function $h(u, c)$ can be defined, for example, as a simple *xor* operation between the numeric values of u and c. The dependence of this value on the symbols of X, which potentially leads to degenerate

hashing performance, is easily eliminated by assigning internal node numbers in random order. This scheme may require a hash table with more than $\max\{n, k\}$ entry points, but its size is still represented in the same number of bits as $\max\{n, k\}$.

The uniqueness of the hash function also yields the capability of accessing the parent of a node without using extra storage. If we let the list terminator in the hash table be, say, any negative value – instead of one single value – we can store information about the hash table entry in that value. For example, let the list terminator for hash table entry i be $-(i+1)$. We find in which list a node is stored, after following its *next* pointer chain to the end, signified by any negative value. This takes expected constant time using the following procedure:

To find the parent u of a given node v, we first determine the first symbol c in the label of (u, v). If v is a leaf, $c = \mathit{fsym}(v)$, otherwise $c = x_{\mathit{pos}(v)}$. We then follow the chain of *next* pointers from v until a negative value j is found, which is the list terminator in whose value the hash table entry number is stored. Thus, we find the hash value $i = -(j+1)$ for u and c, and obtain $u = g(i, c)$.

1.4 Sequential Data Compression

A large part of this thesis is motivated by its application in data compression. Compression is a rich topic with many branches of research; our viewpoint is limited to one of these branches: lossless sequential compression. This is often referred to as *text compression*, although its area of application goes far beyond that of compressing natural language text – it can be used for any data organized as a sequence.

Furthermore, we almost exclusively concentrate on the problem of *source modelling*, leaving the equally important area of *coding* to other research. The coding methods we most commonly refer to are *entropy codes*, such as Huffman and arithmetic coding, which have the purpose of representing output data in the minimum number of bits, given a probability distribution (see for instance Witten, Moffat, and Bell [70, chapter two]). A carefully designed coding scheme is essential for efficient overall compression performance, particularly in connection with predictive source models, where probability distributions are highly dynamic.

Our goal is to accomplish methods that yield good compression with moderate computational resources. Thus, we do not attempt to improve compression ratios at any price. Nor do we put much effort into finding theoretical bounds for compression. Instead, we concentrate on seeking efficient source models that can be maintained in

time which is linear, or very close to linear, in the size of the input. By careful application of algorithmic methods, we strive to shift the balance point in the tradeoff between compression and speed, to enable more effective compression at reasonable cost. Part of this work is done by starting from existing methods, whose compression performance is well studied, and introducing augmentations to increase their practical usefulness. In other parts, we propose methods with novel elements, starting from scratch.

We assume that the reader is familiar with the basic concepts of information theory, such as an intuitive understanding of a *source* and the corresponding definition of *entropy*, which are important tools in the development of data compression methods. However, as our exploration has primarily an algorithmic viewpoint, the treatment of these concepts is often somewhat superficial and without mathematical rigour. For basic reference concerning information theoretic concepts, see, for instance, Cover and Thomas [21].

Chapter Two

Sliding Window Indexing

In many applications where substrings of a large string need to be indexed, a static index over the whole string is not adequate. In some cases, the index needs to be used for processing part of the indexed string before the complete input is known. Furthermore, we may not need to keep record all the way back to the beginning of the input. If we can release old parts of the input from the index, the storage requirements are much smaller.

One area of application where this support is valuable is in data compression. The motive for deletion of old data in this context is either to obtain an adaptive model or to accomplish a space economical implementation of an advanced model. Chapter five presents applications where support of a dynamic indexed string is critical for efficient implementation of various source modelling schemes.

Utilizing a suffix tree for indexing the first part of a string, before the whole input is known, is directly possible when using an *online* construction algorithm such as Ukkonen's [67], but the nontrivial task of moving the endpoint of the index forward remains.

The contribution of this chapter is the augmentation of Ukkonen's algorithm into a full *sliding window* indexing mechanism for a window of variable size, while maintaining the full power and efficiency of a suffix tree. The description addresses every detail needed for the implementation, which is demonstrated in appendix A, where we present source code for a complete implementation of the scheme.

Apart from Ukkonen's algorithm construction algorithm, the work of Fiala and Greene [26] is crucial for our results. Fiala and Greene

presented (in addition to several points regarding Ziv-Lempel compression which are not directly relevant to this work) a method for maintaining valid edge labels when making deletions in a suffix tree. Their scheme is not, however, sufficient for a full linear-time sliding window implementation, as several other complications in moving the indexed string need to be addressed.

The problem of indexing a sliding window with a suffix tree is also considered by Rodeh, Pratt, and Even [57]. Their method is to avoid the problem of deletions by maintaining three suffix trees simultaneously. This is clearly less efficient, particularly in space requirements, than maintaining a single tree.

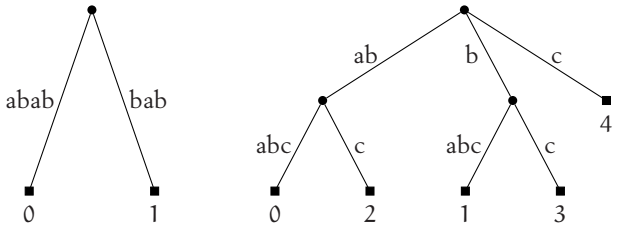
2.1 Suffix Tree Construction

Since the support of a sliding window requires augmentation inside the suffix tree construction algorithm, it is necessary to recapitulate this algorithm in detail. We give a slightly altered, and highly condensed, formulation of Ukkonen's online suffix tree construction algorithm as a basis for our work. For a more elaborate description, see Ukkonen's original paper [67].

We base the description on our suffix tree implementation, and notation, described in §1.3.2. One detail regarding the given representation needs to be clarified in this context. To minimize representation of leaves, we have stipulated that incoming edges of leaves are implicitly labeled with strings that continue to the end of the input. In the current context, the end of the input is not defined. Instead, we let these labels dynamically represent strings that continue to the end of the *currently* indexed string. Hence, there is no one-to-one mapping between suffixes and leaves of the tree, since some suffixes of the indexed string may be represented by internal nodes or points between symbols in edge labels.

Ukkonen's algorithm is incremental. In iteration i we build the tree indexing $x_0 \dots x_i$ from the tree indexing $x_0 \dots x_{i-1}$. Thus, iteration i needs to add, for all suffixes α of $x_0 \dots x_{i-1}$, the i strings αx_i to the tree. Just before αx_i is to be added, precisely one of the following three cases holds:

1. α occurs in precisely one position in $x_0 \dots x_{i-1}$. This means that it is represented by some leaf s in the current tree. In order to add αx_i we need only increment the string depth of s.
2. α occurs in more than one position in $x_0 \dots x_{i-1}$, but αx_i does not occur in $x_0 \dots x_{i-1}$. This implies that α is represented by an internal point in the current tree, and that a new leaf must be created for αx_i.



Suffix trees for the strings 'abab' (left) and 'ababc' (right). Leaf numbers are shown below the corresponding leaves.

In addition, if *point*(α) is not located at a node but inside an edge label, this edge has to be split, and a new internal node introduced, to serve as the parent of the new leaf.

3 αx_i occurs in $x_0 \ldots x_{i-1}$ and is therefore already present in the tree.

Note that if, for a given x_i in a specific suffix tree, case 1 holds for $\alpha_1 x_i$, case 2 for $\alpha_2 x_i$, and case 3 for $\alpha_3 x_i$, then $|\alpha_1| > |\alpha_2| > |\alpha_3|$.

For case 1, all work is avoided in our representation. The labels of leaf edges are defined to continue to the end of the currently indexed string. This implies that the leaf that represented α after iteration $i-1$ implicitly gets its string depth incremented by iteration i, and is thus updated to represent αx_i.

Hence, the point of greatest depth where the tree may need to be altered in iteration i is *point*(α), for the longest suffix α of $x_0 \ldots x_{i-1}$ that also occurs in some other position in $x_0 \ldots x_{i-1}$. We call this the *active point*. Before the first iteration, the active point is $(\mathit{root}, 0, *)$, where $*$ denotes any symbol. Other points that need modification can be found from the active point by following suffix links, and possibly some downward edges.

Finally, we reach the point that corresponds to the longest αx_i string for which case 3 holds. This concludes iteration i; all the necessary insertions have been made. We call this point, the point of maximum string depth for which case 3 holds, the *endpoint*. The active point for the next iteration is found simply by moving one step down from the endpoint, just beyond the symbol x_i along the current path.

The figure above shows an example suffix tree before and after the iteration that expands the indexed string from 'abab' to 'ababc'. Before this iteration, the active point is $(\mathit{root}, 2, \text{'a'})$, the point corresponding to 'ab', located on the incoming edge of *leaf*(0). During the iteration, this edge is split, points $(\mathit{root}, 2, \text{'a'})$ and $(\mathit{root}, 1, \text{'b'})$ are made into explicit nodes, and leaves are added to represent the suffixes 'abc', 'bc', and 'c'. The two longest suffixes are represented by the leaves that were already present, whose depths are implicitly incremented. The active point for the next iteration is $(\mathit{root}, 0, *)$, corresponding to the empty string.

We maintain a variable *front* that holds the position to the right of the string currently included in the tree. Hence, $front = i$ when the tree spans $x_0 \ldots x_{i-1}$.

The *insertion point* is the point where new nodes are inserted. Two variables *ins* and *proj* are kept, where *ins* is the closest node above the insertion point and *proj* is the number of projecting symbols between that node and the insertion point. Consequently, the insertion point is $(ins, proj, x_{front-proj})$.

At the beginning of each iteration, the insertion point is set to the active point. The C*anonize* subroutine on the facing page is used to ensure that $(ins, proj, x_{front-proj})$ is a valid point after *proj* has been incremented, by moving *ins* along downward edges and decreasing *proj* for as long as *ins* and the insertion point are separated by at least one node. The routine returns *nil* if the insertion point is now at a node, otherwise it returns the node r, where (ins, r) is the edge on which the active point resides.

The complete procedure for one iteration of the construction algorithm is shown on the facing page. This algorithm takes constant amortized time, provided that the operation to retrieve $child(u, c)$ given u and c takes constant time (proof given by Ukkonen [67]), which is true in our representation of choice.

2.2 Sliding the Window

We now give the indexed string a dynamic left endpoint. We maintain a suffix tree over the string $X_M = x_{tail} \ldots x_{front-1}$, where *tail* and *front* are integer variables such that at any point in time $0 \leq front - tail \leq M$ for some maximum length M. For convenience, we assume that *front* and *tail* may grow indefinitely. However, since the tree does not contain any references to $x_0 \ldots x_{tail-1}$, the storage for these earlier parts of the input string can be released or reused. In practice, this is most conveniently done by representing indices as integers modulo M, and storing X_M in a circular buffer. This implies that for each $i \in [0, M)$, the symbols x_{i+jM} occupy the same memory cell for all nonnegative integers j, and consequently only M symbols of storage space is required for the input.

Each iteration of suffix tree construction, performed by the algorithm shown on the facing page, can be viewed as a method to increment *front*. This section presents a method that, in combination with some slight augmentations to the previous *front* increment procedure, allows *tail* to be incremented without asymptotic increase in time complexity. By this method we can maintain a suffix tree as an

Canonize:

1 While $proj > 0$, repeat steps 2 to 5:
2 $r \leftarrow child(ins, x_{front-proj})$.
3 $d \leftarrow depth(r) - depth(ins)$.
4 If r is a leaf or $proj < d$, then stop and return r,
5 otherwise, decrease *proj* by d, and set $ins \leftarrow r$.
6 Return *nil*.

Subroutine that moves *ins* down the tree and decreases *proj*, until *proj* does not span any node.

1 Set $v \leftarrow nil$, and loop through steps 2 to 16:
2 $r \leftarrow$ *Canonize*.
3 If $r = nil$ and $child(ins, x_{front}) \neq nil$, break out of loop to step 17.
4 If $r = nil$ and $child(ins, x_{front}) = nil$, set $u \leftarrow ins$.
5 If r is a leaf, $j \leftarrow spos(r) + depth(ins)$; otherwise $j \leftarrow pos(r)$
6 If $r \neq nil$ and $x_{j+proj} = x_{front}$, break out of loop to step 17.
7 If $r \neq nil$ and $x_{j+proj} \neq x_{front}$, execute steps 8 to 13:
8 Assign u an unused node.
9 $depth(u) \leftarrow depth(ins) + proj$.
10 $pos(u) \leftarrow front - proj$.
11 Delete edge (ins, r).
12 Create edges (ins, u) and (u, r).
13 If r is a leaf, $fsym(r) \leftarrow x_{j+proj}$; otherwise, $pos(r) \leftarrow j + proj$.
14 $s \leftarrow leaf(front - depth(u))$.
15 Create edge (u, s).
16 $suf(v) \leftarrow u$, $v \leftarrow u$, $ins \leftarrow suf(ins)$, and continue from step 2.
17 $suf(v) \leftarrow ins$.
18 $proj \leftarrow proj + 1$, $front \leftarrow front + 1$.

One iteration of suffix tree construction. The string indexed by the tree is expanded with one symbol. Augmentations necessary for sliding window support are given in §2.2.6.

index for a sliding window of varying size at most M, while keeping time complexity linear in the number of processed symbols. The storage space requirement is $\Theta(M)$.

2.2.1

Preconditions Removing the leftmost symbol of the indexed string involves removing the longest suffix of X_M, i.e. X_M itself, from the tree. Since this is the longest string represented in the tree, it must correspond to a leaf. Furthermore, accessing a leaf given its string position is a constant time operation in our tree representation. Therefore it appears, at first glance, to be a simple task to obtain the leaf v to remove as $v = leaf(tail)$, and delete the leftmost suffix simply by removing v and incrementing *tail*.

This simple operation does remove the longest suffix from the tree, and it is the basis of our deletion scheme. However, to correctly maintain a suffix tree for the sliding window, it is not sufficient. We have to ensure that our deletion operation retains a complete and valid suffix tree, which is specified by the following preconditions:

- Path compression must be maintained. If removing one node leaves its parent with only one remaining child, the parent must also be removed.
- Only the longest suffix must be removed from the tree, and all other strings retained. This is not trivial, since without an input terminator, several suffixes may reside on the same edge.
- The insertion point variables *ins* and *proj* must be kept valid.
- Edge labels must not slide outside the window. As *tail* is incremented we must make sure that $pos(u) \geq tail$ still holds for all internal nodes u.

The following sections explain how our deletion scheme deals with these preconditions.

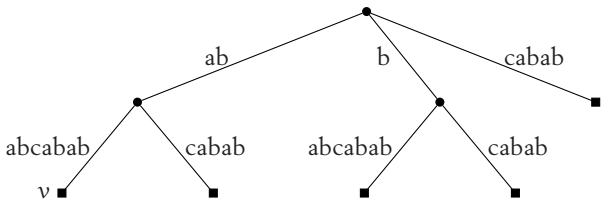
2.2.2 *Maintaining path compression* Given that the only removed node is $v = leaf(tail)$, the only point where path compression may be lost is at the parent of this removed leaf. Let $u = parent(v)$. If u has at least two remaining children after v is deleted, the path compression property is not violated. Otherwise, let s be the single remaining child of u; u and s should be contracted into one node. Hence, we remove the edges $(parent(u), u)$ and (u, s), and create an edge $(parent(u), s)$. To update edge labels accordingly, we move the starting position of the incoming edge label of s backwards by d positions.

Removing u cannot decrease the number of children of $parent(u)$, since s becomes a new child of u. Hence, violation of path compression does not propagate, and the described procedure is enough to keep the tree correctly path compressed.

When u has been removed, the storage space occupied by it should be marked unused, so that it can be reused for nodes created when the front end of the window is advanced.

Since we are now deleting internal nodes, one issue that needs to be addressed is that deletion should leave all suffix links well defined, i.e., if $suf(x) = y$ for some nodes x and y, then y must not be removed unless x is removed. However, this follows directly from the tree properties. Let the string represented by x be $c\alpha$ for some symbol c and string α. The existence of x as an internal node implies that the string $c\alpha$ occurs at least twice in X_M. This in turn implies that α, the string represented by y, occurs at least twice, even if $c\alpha$ is removed. Therefore, y has at least two children, and is not removed.

2.2.3 *Avoiding unwanted suffix removals* When we delete $v = leaf(tail)$, we must ensure that no other string than $x_{tail} \ldots x_{front-1}$ is removed from the tree. This is violated if some other suffix of the currently indexed string is located on the edge $(parent(v), v)$.



Suffix tree of 'ababcabab' illustrating §2.2.3. Deleting leaf v would remove suffixes 'ababcabab' and 'abab'.

The tree shown above indexes the string 'ababcabab'. Deleting v from this tree would remove the longest suffix, but it would also cause the suffix 'abab' to be lost since this is located on the incoming edge of v.

Fortunately, there is a simple way to avoid this. First note the following general string property:

Lemma Assume that A and α are nonempty strings for which the following properties hold: *2A*

1 α is the longest string such that $A = \delta\alpha = \alpha\theta$ for some nonempty strings δ and θ;

2 if $\alpha\mu$ is a substring of A, then μ is a prefix of θ.

Then α is the longest suffix of A that also occurs as a substring in some other position of A.

Proof Trivially, by assumption 1, α is a suffix of A that also occurs as a substring in some other position of A. Assume that it is not the longest one, and let $\chi\alpha$ be a longer suffix with this property. This implies that $A = \phi\chi\alpha = \beta\chi\alpha\gamma$, for nonempty strings ϕ, χ, β, and γ.

Since $\alpha\gamma$ is a substring of A, it follows from assumption 2 that γ is a prefix of θ. Hence, $\theta = \gamma\theta'$ for some string θ'. Now observe that $A = \alpha\theta = \alpha\gamma\theta'$. Letting $\alpha' = \alpha\gamma$ and $\delta' = \beta\chi$ then yields $A = \delta'\alpha' = \alpha'\theta'$, where $|\alpha'| > |\alpha|$, which contradicts assumption 1. □

Assume that some nonempty string would be inadvertently lost from the tree if v was deleted, and let α be the longest string that would be lost. If we let $A = X_M$, the two premises of lemma 2A are fulfilled. This is clear from the following observations:

1 Only prefixes of the removed string can be lost. Hence, α is both a prefix and a suffix of X_M. If a longer string with this property existed, it would be located further down in the tree along the path to v, and it would therefore be lost as well. This cannot be the case, since we defined α as the longest lost string.

2 There cannot be any internal node in the tree below $\mathit{point}(\alpha)$, since it resides on the incoming edge of a leaf. Therefore, for any two strings following α in X_M, one must be a prefix of the other.

Hence, both premises of lemma 2A hold, and we conclude that the longest potentially lost suffix α is also the longest suffix that occurs as a substring elsewhere in X_M.

This in turn implies that $point(\alpha)$ is the active point of the next iteration. Therefore, we can determine if a suffix would be lost by checking if the active point is located on the incoming edge of v, the leaf that is to be deleted. We call C*anonize* and check whether the returned value is equal to v. If so, instead of deleting v, we replace it by a leaf that represents α, namely $leaf(front - |\alpha|)$, where we calculate $|\alpha|$ as the string depth of the active point.

This saves α from being lost, and since all potentially lost suffixes are prefixes of X_M and therefore also of α, the result is that all potentially lost suffixes are saved.

2.2.4 *Keeping a valid insertion point* The insertion point indicated by the variables *ins* and *proj* must, after deletion, still be the correct active point for the next front increment operation. In other words, we must ensure that the point $(ins, proj, x_{front-proj}) = point(\alpha)$ still represents the longest suffix that also appears as a substring in another position of the indexed string. This is violated if and only if:

- the node *ins* is deleted, or
- removal of the longest suffix has the effect that only one instance of the string α is left in the tree.

The first case occurs when *ins* is deleted as a result of maintaining path compression, as explained in §2.2.2. This is easily overcome by checking if *ins* is the node being deleted, and, if so, backing up the insertion point by increasing *proj* by $depth(ins) - depth(parent(ins))$ and then setting $ins \leftarrow parent(ins)$.

The second case is closely associated with the circumstances explained in §2.2.3; it occurs exactly when the active point is located on the incoming edge of the deleted leaf. The effect is that if the previous active point was $c\beta$ for some symbol c and string β, the new active point is $point(\beta)$. To see this, note that, according to the conclusions of §2.2.3, the deleted suffix in this case is $c\beta\gamma$, for some nonempty string γ. Therefore, while $c\beta$ appears only in one position of the indexed string after deletion, the string β still appears in at least two positions. Consequently, the new active point in this case is found following a suffix link from the old one, by simply setting $ins \leftarrow suf(ins)$.

2.2.5 *Keeping Labels Inside the Window* The final precondition that must be fulfilled is that edge labels do not become out of date when *tail* is incremented, i.e. that $pos(u) \geq tail$ for all internal nodes u.

One immediately apparent method is as follows. Each newly added leaf corresponds to a suffix $x_i \dots x_{front}$, for some $i \geq tail$, of the currently indexed string. Each time a leaf is added, we can traverse the path between the root and that leaf, and update the incoming edge label of each internal node u on that path by setting $pos(u) \leftarrow i + depth(u)$. This ensures that all labels on the path from the root to any current leaf, i.e., any path in the tree, are kept up to date. However, this would yield superlinear time complexity, and we must find a way to restrict the number of updates to keep the algorithm efficient.

The idea of the following scheme should be attributed to Fiala and Greene [26]; our treatment is only slightly extended, and modified to fit into our context.

When $leaf(i)$, the leaf representing the suffix $x_i \dots x_{front}$, is added, we let it pass the position i on to its parent. We refer to this operation as the leaf *issuing a credit* to its parent.

We assign each internal node u a binary counter $cred(u)$, explicitly stored in the data structure. This *credit counter* is initially zero as u is created. When a node u receives a credit, we first refresh its incoming edge label by updating the value of $pos(u)$. Then, if $cred(u)$ is zero, we set it to one, and stop. If $cred(u)$ was already one, we reset it to zero, and let u pass a credit on to its parent. This allows the parent, and possibly nodes higher up in the tree, to have the incoming edge label updated.

When a node is deleted, it may have been issued a credit from its newest child (the one that is not deleted), which has not yet been passed on to its parent. Therefore, when a node u is scheduled for deletion and $cred(u) = 1$, we let u issue a credit to its parent. However, this introduces a complication in the updating process: several waiting credits may aggregate, causing nodes further up in the tree to receive an *older* credit than it has already received from another of its children. Therefore, before updating a *pos* value, we compare its previous value against the one associated with the received credit, and use the newer value.

By *fresh credit*, we denote a credit originating from one of the leaves currently present, i.e., one associated with a position larger than or equal to *tail*. Since a node u has $pos(u)$ updated each time it receives a credit, $pos(u) \geq tail$ if u has received at least one fresh credit. The following lemma states that this scheme guarantees valid edge labels.

Lemma (Fiala and Greene) Each internal node has received a fresh credit from each of its children. *2B*

Proof Any internal node of depth $h-1$, where h is the height of the

tree, has only leaves as children. Furthermore, these leaves all issued credits to their parent as they were created, either directly or to an intermediate node that has later been deleted and had the credit passed on. Consequently, any internal node of maximum depth has received a credit from each of its leaves. Furthermore, since each internal node has at least two children, it has also issued at least one fresh credit to its parent.

Assume that any node of depth d received at least one fresh credit from each of its leaves, and issued at least one to its parent. Let u be an internal node of depth $d - 1$. Each child of u is either a leaf or an internal node of depth at least d, and must therefore have issued at least one fresh credit each to u. Consequently, u has received fresh credits from all its children, and has issued at least one to its parent.

Hence, internal nodes of all depths have received fresh credits from all its children. □

To account for the time complexity of this scheme, we state the following:

2C *Lemma* (Fiala and Greene) The number of label update operations is linear in the size of the input.

Proof The number of update operations is the same as the number of credit issue operations. A credit is issued once for each leaf added to the tree, and once when two credits have accumulated in one node. In the latter case, one credit is consumed and disappears, while the other is passed up the tree. Consequently, the number of label updates is at most twice the number of leaves added to the tree. This in turn, is bounded by the total number of symbols indexed by the tree, i.e., the total length of the input. □

2.2.6 *The Algorithms* The deletion algorithm conforming to the conclusions in §2.2.2 through §2.2.5, including the *Update* subroutine used for passing credits up the tree, is shown on the facing page.

The child access operation in step 10 is guaranteed to yield the single remaining child s of u, since all leaves in the subtree of s are newer than v, and s must therefore have issued a newer credit than v to u, causing $pos(u)$ to be updated accordingly.

The algorithm that advances *front* on page 25 needs some augmentation to support deletion, since it needs to handle the credit counters for new nodes. This is accomplished with the following additions:

At the end of step 12: $cred(u) \leftarrow 0$.
At the end of step 15: $Update(u, front - depth(u))$.

Subroutine that issues a credit to node v. The parameter i is the position of the suffix being added.

Update(v, i):
1 While $v \neq$ *root*, repeat steps 2 to 6
2 $u \leftarrow$ *parent*(v).
3 $i \leftarrow \max\{i, pos(v) - depth(u)\}$.
4 $pos(v) \leftarrow i + depth(u)$.
5 $cred(v) \leftarrow 1 - cred(v)$.
6 If $cred(v) = 1$, stop; otherwise $v \leftarrow u$, and continue from step 1.

Deletion algorithm. Removes the longest suffix from the tree and advances *tail*.

1 $r \leftarrow$ *Canonize*, $v \leftarrow$ *leaf*(*tail*).
2 $u \leftarrow$ *parent*(v), delete edge (u, v).
3 If $v = r$, execute steps 4 to 6:
4 $i \leftarrow$ *front* $-$ (*depth*(*ins*) $+$ *proj*).
5 Create edge (*ins*, *leaf*(i)).
6 *Update*(*ins*, i), *ins* $\leftarrow$ *suf*(*ins*).
7 If $v \neq r$, $u \neq$ *root*, and u has only one child, execute steps 8 to 16:
8 $w \leftarrow$ *parent*(u).
9 $d \leftarrow depth(u) - depth(w)$.
10 $s \leftarrow child(u, x_{pos(u)+d})$.
11 If $u =$ *ins*, set *ins* $\leftarrow w$ and *proj* $\leftarrow$ *proj* $+ d$.
12 If $cred(u) = 1$, *Update*$(w, pos(u) - depth(w))$.
13 Delete edges (w, u) and (u, s).
14 Create edge (w, s).
15 If s is a leaf, $fsym(s) \leftarrow x_{pos(u)}$; otherwise, $pos(s) \leftarrow pos(s) - d$.
16 Mark u as unused.
17 *tail* $\leftarrow$ *tail* $+ 1$.

The algorithm as shown fulfills all the preconditions listed in §2.2.1. Hence, we conclude that it can be used to correctly maintain a sliding window.

Apart from the work performed by the *Update* routine, the deletion algorithm comprises only constant time operations. By lemmata 2B and 2C, the total time for label updates is linear in the number of leaf additions, which is bounded by the input length. Furthermore, our introduction of sliding window support clearly does not affect the amortized constant time required by the tree expansion algorithm on page 25 (cf. Ukkonen's time complexity proof [67]). Hence, we can state the following, in analogy with theorem 1B:

2D

Theorem The presented algorithms correctly maintain a sliding window index over an input of size n from an alphabet of size k in $O(n\,i(k))$ time, where $i(k)$ is an upper bound for the time to locate a symbol among k possible choices.

2.3 Storage Issues and Final Result

Two elements of storage required for the sliding window scheme are unaccounted for in our suffix tree representation given in § 1.3.2. The first is the credit counter. This binary counter requires only one bit per internal node, and can be incorporated, for example, as the sign bit of the suffix link. The second is the counter for the number of children of internal nodes, which is used to determine when a node should be deleted. The number of children of any internal node apart from the root in our algorithm is in the range $[1, k]$ at all times. The root initially has zero children, but this can be treated specially. Hence, maintaining the number of children requires memory corresponding to one symbol per internal node.

Consequently, we can combine these observations with observation 1F to obtain the following conclusion:

2E *Theorem* A sliding window suffix tree indexing a window of maximum size M over an input of size n from an alphabet of size k can be maintained in expected $O(n)$ time using storage for $5M + \max\{M, k\}$ integers and $3M$ symbols.

Chapter Three

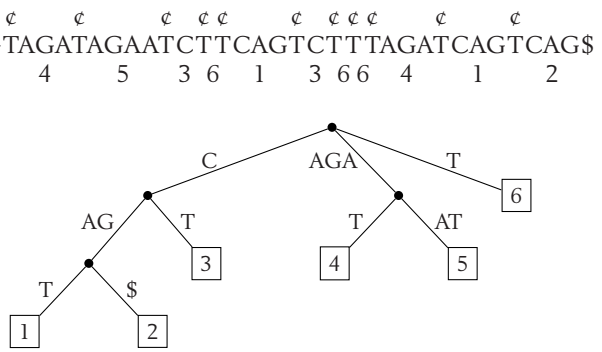
Indexing Word-Partitioned Data

Traditional suffix tree construction algorithms rely heavily on the fact that *all* suffixes are inserted, in order to obtain efficient time bounds. Little work has been done for the common case where only certain suffixes of the input string are relevant, despite the savings in storage and processing times that are to be expected from only considering these suffixes.

Baeza-Yates and Gonnet [9] have pointed out this possibility, by suggesting inserting only suffixes that start with a word, when the input consists of ordinary text. They imply that the resulting tree can be built in $O(n\mathcal{H}(n))$ time, where $\mathcal{H}(n)$ denotes the height of the tree, for n symbols. While the expected height is logarithmic under certain assumptions [64, theorem 1 (ii)], it is unfortunately linear in the worst case, yielding an algorithm that is quadratic in the size of the input.

One important advantage of this strategy is that it requires only $O(m)$ space for m words. Unfortunately, with a straightforward approach such as that of the aforementioned algorithm, this is obtained at the cost of a greatly increased time complexity. We show that this is an unnecessary tradeoff.

We formalize the concept of words to suit various applications and present a generalization of suffix trees, which we call *word suffix trees*. These trees store, for a string of length n in an arbitrary alphabet, only the m suffixes that start at word boundaries. The words are separated by, possibly implicit, *delimiter* symbols. Linear construction time is maintained, which in general is optimal, due to the requirement of scanning the entire input.



A sample string where ¢ = T with its number string and word trie. The word numbers used, shown in the leaves of the trie, generate the number string, shown just below the original string.

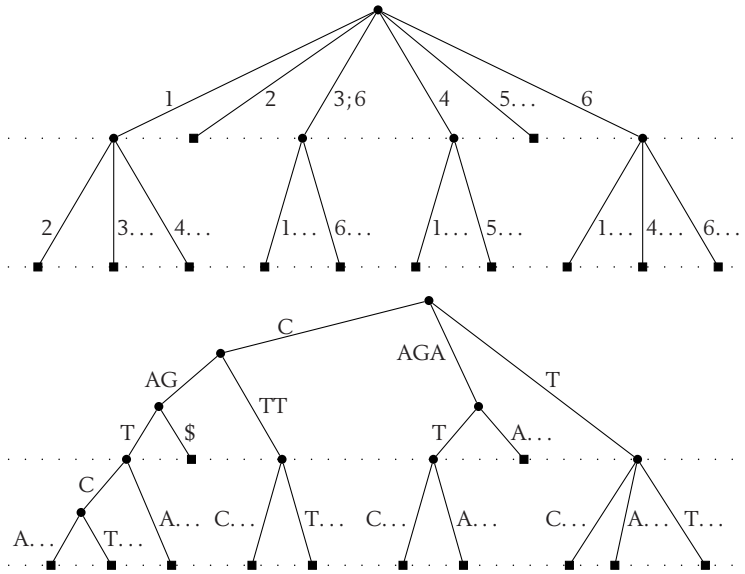
The related problem of constructing *evenly spaced suffix trees* has been treated by Kärkkäinen and Ukkonen [34]. Such trees store all suffixes for which the start position in the original text are multiples of some constant. We note that our algorithm can produce this in the same complexity bounds by assuming implicit word boundaries at each of these positions.

It should be noted that one open problem remains, namely that of removing the use of delimiters – finding an algorithm that constructs a trie of arbitrarily selected suffixes using only $O(m)$ construction space for m words.

3.1 Definitions

For convenience, this chapter considers the input to be drawn from an input alphabet which includes two special symbols which do not necessarily have a one-to-one correspondence to actual low-level symbols of the implementation. One is the end marker \$; the other is a *word delimiter* ¢. This differs slightly from the general definition given in §1.1.1, in that the \$ symbol is included among the k possible symbols of the input alphabet, and in the input string of length n.

Thus, we study the following formal problem. We are given an input string consisting of n symbols from an alphabet of size k, including two, possibly implicit, special symbols \$ and ¢. The \$ symbol must be the last symbol of the input string and may not appear elsewhere, while ¢ appears in $m - 1$ places in the input string. We regard the input string as a series of *words* – the m non-overlapping substrings ending either with ¢ or \$. There may of course exist multiple occurrences of the same word in the input string. We denote the number of *distinct* words by m'. We regard each ¢ or \$ symbol as being contained in the preceding word, which implies that there are no empty words;



The number suffix tree (explained in §3.3) with its expanded final word suffix tree below, for the sample string and word trie shown on the facing page. Dotted lines denote corresponding levels.

the shortest possible word is a single ¢ or \$. The goal is to create a trie structure containing m strings, namely the suffixes of the input string that start at the beginning of words.

The figures on this spread constitute an example where the input consists of a DNA sequence, and the symbol T is viewed as the word delimiter. (This is a special example, constructed for illustrating the algorithm, *not* a practical case.) The lower tree on this page is the word suffix tree for the string displayed on the preceding page. These figures are more completely explained throughout this chapter.

Our definition can be generalized in a number of ways to suit various practical applications. The ¢ symbol does not necessarily have to be a single symbol, we can have a *set* of delimiting symbols, or even sets of delimiting *strings*, as long as the delimiters are easily recognizable.

All tries discussed (the word suffix tree as well as some temporary tries) are assumed to be path compressed. In order to reduce space requirements, edge label strings are represented by pointers into the original string. Thus, a trie with m leaves occupies $\Theta(m)$ space.

We assume that the desired data structure is a *non-lexicographic* trie and that a randomized algorithm is satisfactory, except where otherwise stated. This makes it possible to use hashing to represent trees all through the construction. However, in §3.4 we discuss the creation of lexicographic suffix trees, as well as deterministic construction algorithms.

3.2 Wasting Space: Algorithm A

We first observe the possibility of creating a word suffix tree from a traditional $\Theta(n)$ size suffix tree. This is a relatively straightforward procedure, which we refer to as *Algorithm A*. Delimiters are not necessary when this method is used – the suffixes to be represented can be chosen arbitrarily. Unfortunately however, the algorithm requires much extra space during construction.

Algorithm A is as follows:

1. Build a traditional *non-lexicographic* suffix tree for the input string with a traditional algorithm, using hashing to store edges.
2. Refine the tree into a word suffix tree: remove the leaves that do not correspond to any of the desired suffixes, and perform explicit path compression.
3. If so desired, perform a sorting step to make the trie lexicographic.

The time for step 1 is $O(n)$ according to theorem 1B; the refinement time in step 2 is bounded by the number of nodes in the original tree, i.e. $O(n)$; and step 3 is $O(m + s(m))$, where $s(m)$ denotes the time to sort m symbols, according to observation 1A.

Hence, if the desired final result is a non-lexicographic tree, the construction time is $O(n)$, the same as for a traditional suffix tree. If a sorted tree is desired however, we have an improved time bound of $O(n + s(m))$ compared to the $\Theta(n + s(n))$ time required to create a lexicographic traditional suffix tree on a string of length n. We state this in the following observation:

3A *Observation* A word suffix tree for a string of n symbols in m words can be created in $O(n)$ time and $O(n)$ space, and made lexicographic in extra time $O(m + s(m))$, where $s(m)$ is the time to sort m symbols.

The disadvantage of Algorithm A is that it consumes as much space as traditional suffix tree construction. Even the most space-economical implementation of Ukkonen's or McCreight's algorithm requires several values per node in the range $[0, n]$ to be held in primary storage during construction, in addition to the n symbols of the string. While this is infeasible in many cases, it may well be possible to store the final word suffix tree of size $\Theta(m)$.

3.3 Saving Space: Algorithm B

We now present *Algorithm B*, the main word suffix tree construction algorithm, which in contrast to Algorithm A uses only $\Theta(m)$ space.

The algorithm is outlined as follows. First, a non-lexicographic trie

with m' leaves is built, containing all distinct words: the *word trie*. Next, this trie is traversed and each leaf – corresponding to each distinct word in the input string – is assigned its in-order number. Thereafter, the input string is used to create a string of m numbers by representing every word in the input by its in-order number in the word trie. A *lexicographic* suffix tree is constructed for this string. Finally, this number-based suffix tree is expanded into the final non-lexicographic word suffix tree, utilizing the word trie.

We now discuss the stages in detail.

3.3.1 *Building the Word Trie* We employ a recursive algorithm to create a non-lexicographic trie containing all distinct words. Since the delimiter is included at the end of each word, no word can be a prefix of another. This implies that each word will correspond to a leaf in the word trie. We use hashing for storing the outgoing edges of each node. The construction is performed top-down by the following algorithm, beginning at the root, which initially contains all words:

1 If the current node contains only one word, stop.
2 Set the variable i to 1.
3 Check if all contained words have the same ith symbol. If so, increment i by one, and repeat this step.
4 Let the incoming edge to the current node be labeled with the substring consisting of the $i-1$ symbol long common prefix of the words it contains. If the current node is the root, and $i > 1$, create a new, unary, root above it.
5 Store all distinct ith symbols in a hash table. Construct children for all distinct ith symbols, and split the words, with the first i symbols removed, among them.
6 Apply the algorithm recursively to each of the children.

Each symbol is examined no more than twice, once in step 3 and once in step 5. For each symbol examined, steps 3 and 5 perform a constant number of operations. Furthermore, steps 2, 4, and 6 take constant time and are performed once per recursive call, which is clearly less than n. Thus, the time for construction is $O(n)$.

3.3.2 *Assigning In-Order Numbers* We perform an in-order traversal of the trie, and assign the leaves increasing numbers in the order they are visited, as shown in the figure on page 34. At each node, we take the order of the children to be the order in which they appear in the hash table. It is crucial for the correctness of the algorithm (the stage given in §3.3.5), that the following property holds:

3B *Definition* An assignment of numbers to strings is *semi-lexicographic* if and only if for all strings, α, β, and γ, where α and β have a common prefix that is not also a prefix of γ, the number assigned to γ is either less or greater than both numbers assigned to α and β.

For an illustration of this, consider the word trie shown on page 34. The requirement that the word trie is semi-lexicographic ensures that consecutive numbers are assigned to the strings AGAT and AGAAT, since these are the only two strings with the prefix AGA.

The time for this stage is the same as for an in-order traversal of the word trie, which is clearly $O(m')$, where $m' \leq m$ is the number of distinct words.

3.3.3 *Generating a Number String* We now create a string of length m in the alphabet $\{1, \ldots, m'\}$.

This is done in $O(n)$ time by scanning the original string while traversing the word trie, following edges as the symbols are read. Each time a leaf is encountered, its assigned number is output, and the traversal restarts from the root.

3.3.4 *Constructing the Number-Based Suffix Tree* We create a traditional *lexicographic* suffix tree from the number string. For this, we use an ordinary suffix tree construction algorithm, such as McCreight's or Ukkonen's. Edges are stored in a hash table. The time needed for this is $O(m)$.

Since hashing is used, the resulting trie is non-lexicographic. However, it follows from observation 1A that it can be made lexicographic in $O(m)$ time using bucket sorting. In the lexicographic trie, we represent the children at each node with linked lists, so that the right sibling of a node can be accessed in constant time.

As an alternative, the suffix tree construction algorithm of Farach (see §1.3.1) can be used to construct this lexicographic suffix tree directly in $O(m)$ time, which eliminates the randomization element of this stage.

3.3.5 *Expanding the Number-Based Suffix Tree* Each node of the number-based suffix tree is now replaced by a local trie, containing the words corresponding to the children of that node. First, we preprocess the word trie for lowest common ancestor retrieval in $O(m')$ time, using for example the method of Harel and Tarjan [30]. This allows lowest common ancestors to be obtained in constant time. The local tries are then built left-to-right, using the fact that since the assignment of

numbers to words is semi-lexicographic and the number-based suffix tree is lexicographic, each local trie has the essential structure of the word trie with some nodes and edges removed. We find the lowest common ancestor of each pair of adjacent children in the word trie, and this gives us the appropriate insertion point (where the two words diverge) of the next node directly.

More specifically, after preprocessing for computation of lowest common ancestors, we build a local trie at each node. The node expansion (illustrated in the figure on page 35) is performed in the following manner:

1 Insert the first word.
2 Retrieve the next word in left-to-right order from the sorted linked list of children. Compute the lowest common ancestor of this word and the previous word in the word trie.
3 Look into the partially built trie to determine where the lowest common ancestor of the two nodes should be inserted, if it is not already there. This is done by searching up the tree from the last inserted word until reaching a node that has smaller height within the word trie.
4 If necessary, insert the internal (lowest common ancestor) node, and insert the leaf node representing the word.
5 Repeat from step 2 until all children have been processed.
6 If the root of the local trie is unary, remove it to maintain path compression.

Steps 1 and 6 take constant time, and are executed once per internal node of the number-based suffix tree. This makes a total of $O(m')$ time for these steps. Steps 2, 4, and 5 also take constant time, and are executed once per node in the resulting word suffix tree. This implies that their total cost is $O(m)$. The total work performed in step 3 is essentially an in-order traversal of the local subtree being built. Thus, the total time for step 3 is proportional to the total size of the final tree, which is $O(m)$. Consequently, the expansion takes a total of $O(m)$ time.

Main Algorithm Result The correctness of the algorithm is easily verified. The crucial point is that the number-based suffix tree has the essential structure of the final word suffix tree, and that the expansion stage does not change this. 3.3.6

Theorem A word suffix tree for an input string of size n containing m words can be built in $O(n)$ expected time, using $O(m)$ storage space. 3C

3.4 Extensions and Variations

Although the use of randomization, in the form of hashing, and non-lexicographic suffix trees during construction is sufficient for a majority of practical applications, we describe extensions to Algorithm B in order to meet stronger requirements.

3.4.1 *Building a Lexicographic Trie* While many common applications have no use for maintaining a lexicographic trie, there are cases where this is necessary. (A specialized example is the number-based suffix tree created in §3.3.4).

If the alphabet size k is small enough to be regarded as a constant, it is trivial to modify Algorithm B to create a lexicographic tree in linear time: instead of hash tables, use any ordered data structure – most naturally an array – of size $O(k)$ to store references to the children at each node.

If hashing is used during construction as described in the previous section, Algorithm B can be modified to construct a lexicographic trie simply by requiring the number assignments in §3.3.2 to be lexicographic instead of semi-lexicographic. Thereby, the number assignment reflects the lexicographic order of the words exactly, and this order propagates to the final word suffix tree. A lexicographic number assignment can be achieved by ensuring that the word trie constructed in §3.3.1 is lexicographic. Observation 1A states that the trie can be made lexicographic at an extra cost which is asymptotically the same as for sorting m' symbols, which yields the following:

3D *Theorem* A lexicographic word suffix tree for an input string of size n containing m words of which m' are distinct can be built in $O(n + s(m'))$ expected time, using $O(m)$ storage space, where $s(m')$ is the time required to sort m' symbols.

For the general problem, with no restrictions on alphabet size, this implies an upper bound of $O(n \log\log n)$ by applying the currently best known upper bound for integer sorting [3].

3.4.2 *A Deterministic Algorithm* A deterministic version of Algorithm B can be obtained by representing the tree with deterministic data structures only, such as binary search trees. Also, when these data structures maintain lexicographic ordering of elements (which is common, even for data structures with the best known time bounds) the resulting tree becomes lexicographic as a side effect. We obtain a better worst case time, at the price of an asymptotically inferior expected

performance.

We define $i(m, m')$ to denote the time to insert m symbols into ordered dictionaries each bounded in size by m', where $m' \leq m$ is the number of distinct words. In a straightforward manner, we can replace the hash tables in §3.3.1 and §3.3.4 with deterministic data structures. Since no node may have more than m' children, the resulting time complexity is $O(n + i(m, m'))$.

Theorem A lexicographic word suffix tree for an input string of size n containing m words of which m' are distinct can be built deterministically in $O(n + i(m, m'))$ time and $O(m)$ space, where $i(m, m')$ is the time required to insert m symbols into ordered dictionaries each bounded in size by m'. *3E*

Using binary search trees, $i(m, m') = O(m \log m')$. There are other possibilities, for example we could implement each node as a fusion tree [27], which implies

$$i(m, m') = O(m \log m' / \log\log m'),$$

or as an exponential search tree [2], which implies

$$i(m, m') = O(m\sqrt{\log m'}), \text{ or}$$

$$i(m, m') = O(m \log\log m' \log\log k),$$

where the latter bound is the more advantageous when the alphabet size is reasonably small.

3.5 Sublinear Construction: Algorithm C

In some cases, particularly when the alphabet is small, we may assume that the n symbols in the input string occupy $o(n)$ machine words. Then it may be possible to avoid the apparently inescapable $\Omega(n)$ cost due to reading the input.

This theme can be altered in many ways, the details depend on the application. The purpose of this – somewhat technical – section is to show that a cost of $\Omega(n)$ is not a theoretical necessity.

We start by studying the case when the positions of the delimiters are known in advance. Then we describe an application where the input string can be scanned and delimiters located in $o(n)$ time.

If the alphabet size is k, then each symbol occupies $\log k$ bits and the total length of the input is $N = n \log k$ bits stored in N/w machine words, where w is the number of bits in a machine word. (In this section, it is important to distinguish between *words* in the input string

and hardware-dependent *machine words*.)

We first observe the following:

3F *Lemma* A lexicographic trie containing strings of a-bit symbols can be transformed into the corresponding lexicographic trie in a b-bit alphabet in linear time, where a and b are not larger than the size of a machine word.

Proof This transformation can be made in two steps, where we first transform the trie of a-bit symbols into a binary trie, which is then transformed into the final b-bit trie.

For the first part, we compute the lowest common ancestor in the binary trie for each pair of neighbouring strings, by finding the position of their first differing bit. This position is found in constant time using the technique of Fredman and Willard [27]. When lowest common ancestors for each pair of adjacent leaves are known, we can construct a binary path compressed trie in the same manner as the node expansion stage of Algorithm B.

The binary trie, in turn, is easily transformed into a trie of the desired degree in linear time during a single traversal, by constructing each new node from b levels of the binary trie. (For a detailed description, we refer to Andersson, Hagerup, Nilsson, and Raman [3]). □

The following algorithm, which we refer to as *Algorithm* C, builds a word suffix tree, while temporarily viewing the string as consisting of n' b-bit pseudo-symbols, where $n' = o(n)$. It is necessary that this transformation does not cause the words to be comprised of fractions of pseudo-symbols. Therefore, in the case where a word ends at the ith bit of a pseudo-symbol, we pad this word implicitly with $b - i$ bits at the end, so that the beginning of the next word may start with an unbroken pseudo-symbol. This does not influence the structure of the input string, since each distinct word can only be replaced by another distinct word. Padding may add at most $m(b-1)$ bits to the input. Consequently,

$$n' = O\left(\frac{N + m(b-1)}{b}\right) = O\left(\frac{N}{b} + m\right)$$

We are now ready to present Algorithm C:

1 Construct a non-lexicographic word trie in the b-bit alphabet in time $O(n')$, as in §3.3.1. The padding of words does not change the important property of direct correspondence between the words and the leaves of the word trie.

2 Sort the edges of this trie, yielding a lexicographic trie in the b-bit alphabet in $O(m' + s_b(m'))$ time, by observation 1A, where $s_b(m')$ is

the time to sort m' b-bit integers.

3 Assign in-order numbers to the leaves, and then generate the number string in time $O(n')$, in the same manner as in §3.3.3.

4 Convert this word trie into a word trie in the original k-size alphabet, utilizing lemma 3F. (This does not affect the in-order numbers of the leaves).

5 Proceed from the number-based suffix tree construction stage (§3.3.4) of Algorithm B.

The first four steps take time $O(n' + s_b(m'))$, and the time for the completion of the construction from §3.3.4 is $O(m)$. Thus the complexity of Algorithm C is $O(n' + m + s_b(m'))$. Thereby we obtain the following theorem:

Theorem When the positions of all delimiters are known, a lexicographic word suffix tree on a string comprising m words of which m' are distinct, can be constructed in time *3G*

$$O\left(\frac{N}{b} + m + s_b(m')\right)$$

for some integer parameter $b \leq w$, where N is the number of bits in the input, w is the machine word length, and $s_b(m')$ is the time to sort m' b-bit integers.

Note that theorem 3G does not give a complete solution to the problem of creating a word suffix tree. We still have to find the delimiters in the input string, which may take linear time. We illustrate a possible way around this for one application:

Example: Huffman Coded Text Suppose we are presented with a Huffman coded text and asked to generate an index on every suffix starting with a word. Furthermore, suppose that word boundaries are defined to be present at every position where a non-alphabetic symbol (a space, comma, punctuation etc.) is followed by an alphabetic symbol (a letter), i.e. we have implicit ¢ symbols in these positions. The resulting word suffix tree may be a binary trie based on the Huffman codewords, or a trie based on the original alphabet. Here we assume the former.

We view the input as consisting of b-bit pseudo-symbols, where $b = (\log n)/2$. The algorithm is divided in two main parts:

1 *Create a code table:* We start by creating a table containing 2^b entries, each entry corresponding to one possible pseudo-symbol. For each entry, we scan the corresponding pseudo-symbol and examine its contents by decoding the Huffman codewords contained in it. If

there is an incomplete Huffman codeword at the end, we make a note to the length of this codeword. We denote the decodable part of the pseudo-symbol a *chunk*. While decoding the contents of a table entry, we check if any word boundaries are contained in the decoded chunk. If so, this is noted in the table entry. Furthermore, we check if the last symbol in the chunk is non-alphabetic, in which case we note that this symbol, together with the first symbol in the next chunk, may define a word boundary.

The time to create and scan the table is at most proportional to the total number of bits it contains, which is $2^b \cdot b$.

2 *Scan the input and locate delimiters:* We use p as a pointer into the input string, and scan the input for delimiters with the following procedure:

1 Set $p \leftarrow 1$.

2 Read a pseudo-symbol (b bits), starting at position p.

3 Use the pseudo-symbol as an address in the code table. Examine if any word boundaries are contained in the corresponding decoded chunk. Let i be the length of the chunk. We have two cases:

- **a** $i \geq b/2$. Update p to point at the first bit after the chunk and repeat from step 2.
- **b** $i < b/2$. Continue reading bits in the input string one at a time while traversing the Huffman tree until the end of a symbol is found. Update p to point at the first bit after this symbol and go repeat from step 2.

Assuming that b consecutive bits can be read in $O(1)$ time, the time consumption for step 2 is constant. This step is performed a total number of $O(\lceil N/b \rceil)$ times.

Case a of step 3 takes constant time plus the number of found word boundaries. Hence the total cost of this case is $O(N/b + m)$.

Case b of step 3 occurs when more than the last $b/2$ bits are occupied by a single symbol. It consumes time proportional to the number of bits in the symbol's codeword each time it occurs. Hence, the total cost of case b equals the total number of bits occupied by codewords of length more than $b/2$.

The length of a Huffman coded text asymptotically approaches the entropy of the text. Therefore, we may assume that the length of the codeword for a symbol with frequency f approaches $-\log f$. This yields the following:

3H *Observation* Given a Huffman coded input string of n symbols, a symbol whose Huffman codeword occupies i bits occupies a total of $O(ni/2^i)$ bits in the coded string.

Since the number of symbols occupying i bits cannot exceed the alphabet size, k, the total number of bits taken by codewords of length i or longer is $O(kNi/2^i)$. Hence, the total number of bits taken by codewords of length $b/2$ or longer is $O(kNb/2^b)$, which gives us a bound on the cost of case b in step 3.

The total cost for finding delimiters becomes

$$O\left(2^b \cdot b + \frac{N}{b} + m + \frac{kNb}{2^b}\right) = O\left(\sqrt{n}\log n + \frac{N}{\log n} + m + \frac{kN\log n}{\sqrt{n}}\right)$$

The first term can be canceled since $N \geq n$ and the last term can be canceled if $k = O(\sqrt{n}/(\log n)^2)$. We then get a cost of

$$O\left(\frac{N}{\log n} + m\right)$$

Next, applying theorem 3G with the same choice of b, we find that

$$s_b(m') = O(m' + 2^b) = O(m' + \sqrt{n})$$

by using bucket sorting; this cost is negligible. The space used by this algorithm is $O(m + \sqrt{n})$, the last term being due to the table. This yields:

Observation For a Huffman coded input string of n symbols coded in N bits, where the alphabet size k satisfies $k = O(\sqrt{n}/(\log n)^2)$, a word suffix tree on m natural words can be constructed in time $O(N/\log n + m)$ with construction space $O(m + \sqrt{n})$. *31*

It should be noted that even if the alphabet is very large, the complexity of our algorithm would be favourable as long as symbols with long Huffman codewords are rare, i.e. when the entropy of the input string is not too high.

3.6 Additional Notes on Practice

Space Overhead As noted in §1.3, a *suffix array* is a space efficient alternative to the suffix tree. Asymptotically, our space requirement is better than that of a suffix array, but, an asymptotic advantage may of course sometimes be neutralized by high constant factors. 3.6.1

However, the potential increase in constant factors from using our data structure is not particularly large. Recall that we have n symbols, m words, and m' distinct words. The space taken by our construction algorithm equals the space required to construct a traditional suffix tree of m symbols, plus the space required to store m' words in the word trie, (including lowest-common-ancestor links). In many prac-

Examples of natural language text.

	n	m	m'
Mark Twain's *Tom Sawyer*	387 922	71 457	7 389
August Strindberg's *Röda rummet*	539 473	91 771	13 425

tical cases (for example, see the table above which lists two typical examples of natural language), m' is considerably smaller than m and we can neglect the space required by the word trie.

Thus, the word suffix tree, whose final size is bounded by $O(m)$, has competitive space requirements compared to the linear-sized suffix array, unless the word lengths are very small.

3.6.2 *Examples of Applications* The word suffix tree is indeed a natural data structure, and it is surprising that efficient construction of word suffix trees has previously received very little attention. We now discuss several practical cases where word suffix trees would be desirable.

With natural languages, a reasonable word partitioning would consist of standard text delimiters: space, comma, carriage return, etc. We could also use implicit delimiters, as in the example in the preceding section. Using word suffix trees, large texts can be manipulated with a greatly reduced space requirement, as well as increased processing speed [9]. The table above indicates that the number of words, m, in common novels, is much less than the length of the work in bytes, n. This difference is even greater when one considers the number of distinct words, m'.

An application directly related to this is natural language text modelling as considered by Teahan. As a way of saving space in a PPM* context trie data structure (see §5.2.1) that is used as a *word model*, he suggests including only contexts that start at a word [65, page 187 ff.]. This corresponds exactly to a word suffix tree with the space character as the word delimiter. Teahan concludes that this provides substantial storage savings.

In the study of DNA sequences, we may represent a large variety of genetic substructures as words, from representations of single amino acids, up to entire gene sequences. In many such cases, the size of the overlying DNA string is substantially greater than the number of substructures it contains. As an example, there are merely tens of thousands of human genes, whilst the entire length of human DNA contains approximately three billion nucleotides.

The word suffix tree is of particular importance in the case where the indexed string is not held in primary storage while the tree is utilized. Using an alternative trie representation that stores only the first

symbol of each edge and the length of the label explicitly in the trie, allows search operations with a single access to secondary storage. With this representation, only $O(m)$ cells of primary storage are required, regardless of the length of the search string. However, a search operation may reach a leaf where it would have failed in the tree with full edge label representation; the full string must subsequently be compared against the potentially matching position in the indexed string.

Chapter Four

Suffix Sorting

Suffix sorting is the problem of lexicographically ordering all the suffixes of a string. The suffixes are represented by integers denoting their starting positions. We present a novel algorithm that removes much of the overhead of previous solutions, and yet maintains robust behaviour for all kinds of input, with a worst case time complexity of $O(n \log n)$. We present a practical implementation in detail and give experimental results that demonstrate the favourable performance of our algorithm.

Suffix sorting has at least two important applications. One is construction of a *suffix array* (see §1.3). Another is in data compression with the *Burrows-Wheeler transform*, BWT, where suffix sorting is a computational bottleneck, and an efficient sorting method is crucial for any implementation of this compression scheme. A detailed description of BWT can be found in chapter six.

Suffix sorting differs from ordinary string sorting in that the elements to sort are overlapping strings, whose lengths are linear in the input length n. This implies that a comparison-based algorithm, which requires $\Omega(n \log n)$ comparisons, may take $\Omega(n^2 \log n)$ time for suffix sorting, and analogously a non-specialized radix sorting algorithm may take $\Omega(n^2)$ time. Fortunately, these bounds can be surpassed with specialized methods.

Linear time suffix sorting can be achieved by building a suffix tree and obtaining the sorted order from its leaves. However, a suffix tree involves overhead, particularly in space requirements, which commonly makes it too expensive to use for suffix sorting alone. In experiments, we find our proposed algorithm to outperform suffix tree

implementations for natural data, even for very large files, and to be competitive even for degenerate cases – despite the fact that suffix trees have superior asymptotic time complexity. In addition, our algorithm requires less space than a suffix tree.

Manber and Myers [47] presented an elegant radix-sorting based algorithm that takes at most $O(n \log n)$ time. They also suggested augmentations to allow string matching operations in time bounds close to those of the suffix tree, at the cost of additional space. Although our proposed algorithm is strongly related to that of Manber and Myers (it requires the same amount of space, has the same asymptotic worst case time complexity, and relies on the same suffix ordering observations), our algorithm gains a substantial advantage through reduction of superfluous processing. Our experiments clearly show that our approach yields a substantially faster algorithm for almost any input.

Our algorithm exhibits an excellent robustness when processing large or repetitive inputs, matched only by suffix trees. Thus, although a general string sorting algorithm optimized for short strings may have a slight advantage for inputs with little repetition, we assert that our algorithm is clearly a better choice in general, since ordinary string sorting degenerates catastrophically for some input distributions.

In §4.1 we recapitulate the Manber-Myers algorithm and other approaches connected with our algorithm, which we present in its basic version in §4.2. In §4.3 we analyze time complexity. In §4.4 we present various refinement techniques. In §4.5 we present a practical implementation that includes the refinements, and results of an experimental comparison with other suffix sorting implementations.

This work was performed in collaboration with Kunihiko Sadakane, who has previously presented the basic ideas of the proposed algorithm in preliminary work [59]. This extended work presents an algorithm that has been improved in both time and space requirements, and contributes a tight time complexity analysis.

Problem Definition We apply our normal notation regarding input, considering a string $X = x_0 x_1 \ldots x_n$ of $n + 1$ symbols, where $x_n = \$$. We regard $\$$ as having a value below all other symbols. By S_i, for $0 \leq i \leq n$, we denote the suffix of X beginning in position i. Thus, $S_0 = X$, and $S_n = \$$ is the first suffix in lexicographic suffix order.

The output of suffix sorting is a permutation of the S_i, contained in an integer array I. Throughout the algorithm, this array holds all integers in the range $[0, n]$, where i represents S_i. Ultimately, these numbers are placed in order corresponding to lexicographic suffix order, i.e., $S_{I[i-1]}$ lexicographically precedes $S_{I[i]}$ for all $i \in [1, n]$. We

refer to this final content of the array I as the *sorted suffix array*.

Thus, suffix sorting in more practical terms means sorting the integer array I according to the corresponding suffixes. We interchangeably refer to the integers in the I array and the suffixes they represent; i.e., *suffix* i, where i is an integer, denotes S_i.

Manber and Myers also consider calculation of *longest common prefix* (LCP) information, within the time bounds of the algorithm. We conjecture that this can be efficiently computed as a byproduct of our algorithm as well, but do not consider it further, for the following reasons. The LCP array, as well as other augmentations that allow faster access in the suffix array, increase space requirements to the extent that a compact suffix tree implementation (consider for example the representation of Kurtz [40], McCreight [48, page 268], Andersson and Nilsson [5], or the one given in §1.3.2) would often be a better alternative. Furthermore, LCP information is unnecessary for many applications. It is, for example, of no use in implementing the Burrows-Wheeler transform. Lastly, a linear time LCP calculation algorithm is given by Kasai, Arimura, and Arikawa [36], surpassing our sorting bound as well as previous ones.

Alphabet Size Considerations Much confusion concerning time complexity of suffix sorting originates from insufficient consideration of the input alphabet size.

It is well known that general sorting with only pairwise comparisons has time complexity $\Theta(n \log n)$, matching the worst case complexity of the Manber-Myers algorithm as well as ours. However, when the input consists of integers in a restricted range, radix techniques may be used. Indeed, the Manber-Myers algorithm is radix based, and requires that the input consists of integers bounded by n. To lift this restriction, the algorithm must be preceded by a transform comprising symbol sorting. Our algorithm does not require this augmentation.

The suffix order can also be obtained by traversing a lexicographic suffix tree of the input string. Thus, according to theorem 1C, linear-time suffix sorting is possible for $O(n)$ alphabets by taking the detour over suffix tree construction. (See also §6.1.2, which discusses the time complexity of BWT.)

4.1 Background

This section presents the background material for our algorithm as well as previous work and alternative approaches to suffix sorting.

Suffix Sorting in Logarithmic Number of Passes One obvious idea for a suffix sorting algorithm is to start by sorting according to only the first symbol of each suffix, then successively refining the order by expanding the considered part of each suffix. If one additional symbol per suffix is considered in each pass, the number of passes required in the worst case is $\Omega(n)$. However, fewer passes are needed if we exploit the fact that each proper suffix of the whole string is also a suffix of *another* suffix. 4.1.1

The key for reducing the number of passes is a doubling technique, originating from Karp, Miller, and Rosenberg [35], which allows the positions of the suffixes after each sorting pass to be used as the sorting keys for preceding suffixes in the next pass.

Define the h-*order* of the suffixes as their order when lexicographically sorted according to their initial h symbols of each suffix. The h-order is not necessarily unique when $h < n$. Note the following:

Observation (Manber and Myers) Sorting the suffixes using, for each suffix S_i, the position in the h-order of S_i as its primary key, and the position of $S_i + h$ in the same order as its secondary key, yields the $2h$-order. *4A*

To use this observation, we first sort the suffixes according to the first symbol of each suffix, using the actual contents of the input; i.e., x_i is the sorting key for suffix i. This yields the 1-order. Then, in pass j, for $j \geq 1$, we use the position that suffix $i + 2^{j-1}$ obtained in pass $j-1$ (where pass 0 refers to the initial sorting step) as the sorting key for suffix i. This doubles the number of considered symbols per suffix in each pass, and only $O(\log n)$ passes in total are needed.

Manber and Myers [47] use this observation to obtain an $O(n \log n)$ time algorithm through bucket sorting in each pass. An auxiliary integer array, which we denote V, is employed to maintain constant-time access to the positions of the suffixes in I.

The main implementation given by Manber and Myers uses, in addition to storage space for X, I, and V, an integer array with n elements, to store counts. However, the authors sketch a method for storing counts in temporary positions in V with maintained asymptotic complexity.

A substantially cleaner solution with reduced constant factors has been presented as source code by McIlroy and McIlroy [49]. Some properties of their implementation are discussed in §4.4.3.

Ternary-Split Quicksort The well known Quicksort algorithm [31] recursively partitions an array into two parts, one with smaller elements 4.1.2

than a *pivot element* and one with larger elements. The parts are then processed recursively until the whole array is sorted.

Where traditional Quicksort partitioning mixes the elements equal to the pivot into – depending on the implementation – one or both of the parts, a ternary-split partition generates *three* parts: one with elements smaller than the pivot, one with elements equal to the pivot, and one with larger elements. The *smaller* and *larger* parts are then processed recursively while the *equal* part is left as is, since its elements are already correctly placed.

This approach is analyzed and implemented by Bentley and McIlroy [13]. The comparison-based sorting subroutine used in our algorithm is directly derived from their work.

4.1.3 *Ternary String-Sorting and Trees* Bentley and Sedgewick [14] employ a ternary-split Quicksort for the problem of sorting an array of strings, which results in the following algorithm. Start by partitioning the whole array based on the first symbol of each string. Then process the *smaller* and *larger* parts recursively in exactly the same manner as the whole array. The *equal* part is also sorted recursively, but with partitioning starting from the *second* symbol of each string. Continue this process recursively: each time an *equal* part is being processed, move the position considered in each string forward by one symbol.

The result is a fast string sorting algorithm which, although it is not specialized for suffix sorting, has been used successfully for this application in the widely spread Burrows-Wheeler implementation *Bzip2* [62].

Our proposed algorithm does not explicitly make use of this string sorting method, but the techniques are related. This is apparent from our time complexity analysis in §4.3. Bentley and Sedgewick consider the implicit *ternary tree* that emerges from their algorithm when regarding each partitioning as a node with three outgoing edges, one for each part of the splitting. We use this tree as a tool for our analysis.

4.2 A Faster Suffix Sort

Usually in suffix sorting, the final sorted positions of most of the suffixes are determined by only the first few symbols of each suffix. This is true for common real-life data (see §4.5.2) as well as random strings. As a result, a specialized suffix sorting method, such as the Manber-Myers algorithm, is often outperformed in practice by an ad hoc string sorting method, optimized for sorting short strings.

To improve the Manber-Myers algorithm, we need to remove un-

necessary scanning and idle reorganizing of already sorted suffixes. Still, we wish to maintain the robust worst case behaviour for repetitive strings which *do* also occur in practice. Furthermore, we do not want to increase the amount of auxiliary space, which would be necessary if a suffix tree was used.

We now present a suffix sorting algorithm that accomplishes this. The various techniques explained in §4.1 are components of our algorithm. This section describes a basic version of the algorithm, which we refer to as *Algorithm S*. In §4.4 we describe refinements to the algorithm that improve both running time and storage space.

Our algorithm inherits the use of observation 4A to double the number of considered symbols over a number of sorting passes, as well as the array V to gain constant time access to suffix positions, from Manber and Myers (see §4.1.1). To refrain from scanning the whole array in each pass, we mark which sections of the suffix array are already finished and skip over them when sorting. We use ternary-split Quicksort (§4.1.2) as our sorting subroutine.

The following concepts allow us to express the rules of individual sorting passes:

Definition When suffixes are sorted lexicographically according to the first h symbols of each suffix, we say that: *4B*

- a maximal sequence of adjacent suffixes in I that have the same initial h symbols is a *group*;
- a group containing at least two suffixes is an *unsorted group*;
- a group containing only one suffix is a *sorted group*; and
- a maximal sequence of adjacent sorted groups is a *combined sorted group*.

We number the groups so that the numbers reflect the order in which the groups appear in I. This is necessary to allow group numbers to be used as sorting keys for preceding suffixes. It is convenient to define the number of a group $I[f \ldots g]$ as one of the numbers $f \ldots g$. For reasons that become apparent in §4.4, we choose the following group numbering:

Definition A group occupying the subarray $I[f \ldots g]$ has *group number* g. *4C*

During sorting, the array V stores group numbers. $V[i] = g$ reflects that suffix i is currently in group number g.

Furthermore, we employ a conceptual array L that holds the lengths of unsorted groups and combined sorted groups in positions corresponding to their leftmost elements. To distinguish between them, we

Algorithm S, the basic version of our proposed algorithm.

1. Place the suffixes, represented by the numbers $0, \ldots, n$, in I. Sort the suffixes using x_i as the key for i. Set h to 1.
2. For each $i \in [0, n]$, set $V[i]$ to the group number of suffix i.
3. For each unsorted group or combined sorted group occupying the subarray $I[f \ldots g]$, set $L[f]$ to its length or negated length respectively.
4. Process each unsorted group in I with ternary-split Quicksort, using $V[i + h]$ as the key for suffix i.
5. Mark splitting positions between non-equal keys in the unsorted groups.
6. Double h. Create new groups by splitting at the marked positions, updating V and L accordingly.
7. If the contents of I is a single combined sorted group, then stop. Otherwise, go to 4.

store positive numbers for unsorted groups and negative numbers – the negated lengths – for combined sorted groups. Thus, if the subarray $I[f \ldots g]$ is an unsorted group, we have $L[f] = g - f + 1$; if it is a combined sorted group, $L[f] = -(g - f + 1)$ instead. In §4.4.1, we show how the relevant information of L can be superimposed on the I array, so that no storage space needs to be allocated for L.

Note the difference in treatment of sorted groups between V and L: L holds lengths of *combined* sorted groups; V holds group numbers for unit length sorted groups.

The first step of the algorithm places the suffixes – represented as numbers 0 through n – into the I array, sorted according to the first symbol of each suffix. This step consists of integer sorting, where the keys are drawn from the input alphabet. After this step, the contents of I are in 1-order. We initialize V and L accordingly.

Then a number of passes for further sorting follow. At the beginning of the jth such pass, the contents of the I array are in h-order where $h = 2^{j-1}$. Note the following:

4D *Observation* When the contents of I are in h-order, each suffix in a sorted group is uniquely distinguished from all other suffixes by its first h symbols.

This implies that all suffixes in sorted groups are already in their final location, and only unsorted groups need to be rearranged.

We sort the unsorted groups using the group number of suffix $i + h$ as the key for suffix i, which, by observation 4A, places the contents of I in $2h$-order. We then split groups between suffixes with non-equal keys, updating V and L. When setting the lengths in L, we combine adjacent groups so that they can be efficiently skipped over in subsequent passes.

h	i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	x_i	t	o	b	e	o	r	n	o	t	t	o	b	e	$
	I[i]	13	2	11	3	12	6	1	4	7	10	5	0	8	9
	V[I[i]]	0	2	2	4	4	5	9	9	9	9	10	13	13	13
	L[i]	−1	2		2		−1	4				−1	3		
1	V[I[i] + h]		4	4	7	0		2	10	12	2		7	12	7
	I[i]		2	11	12	3		1	10	4	7		0	9	8
	V[I[i]]		2	2	3	4		7	7	8	9		12	12	13
	L[i]	−1	2		−3			2		−3			2		−1
2	V[I[i] + h]		8	0				4	3				2	2	
	I[i]		11	2				10	1				0	9	
	V[I[i]]		1	2				6	7				12	12	
	L[i]	−11											2		−1
4	V[I[i] + h]												8	0	
	I[i]												9	0	
	V[I[i]]												11	12	
	L[i]	−14													
	I[i]	13	11	2	12	3	6	10	1	4	7	5	9	0	8

Example run of Algorithm S with the input string 'tobeornottobe'. Time flow is from the top down. Sections with h values show the keys used when sorting the entries that have equal values of V[I[i]]. Other sections show the parts of the contents of X, I, V, and L that are accessed at each sorting stage.

Algorithm S is shown on the preceding page. Its time complexity is analyzed in §4.3. The crucial point of this algorithm is the utilization of observation 4D in step 4: the group lengths stored in L allow us to skip over sorted groups completely while we continue to process unsorted groups. For marking of groups in step 5, we can use, for instance, the sign bits of I. (With the refinement shown in §4.4.2, the necessity of this marking disappears.)

Note that step 4 does not check that $i + h$ is in the legal range – at most n – when referring to $V[i + h]$. This is not necessary, because of the unique \$ symbol that terminates X. All suffixes $n - h + 1, \ldots, n$ have length at most h, and the \$ symbol is therefore included in the considered part of these suffixes, which implies that their positions in the sorted suffix array must already have been uniquely determined. They are therefore all in sorted groups, and we never attempt to access their sorting keys.

The chart above shows a run of Algorithm S with the string 'tobeornottobe' as input. The top section of the chart shows X, the input with the unique \$ symbol attached to the end. The second section shows the result of sorting the suffixes according to their first symbols.

Negative numbers in L[0], L[5] and L[10] denote that suffixes I[0], I[5] and I[10] are already in their final positions.

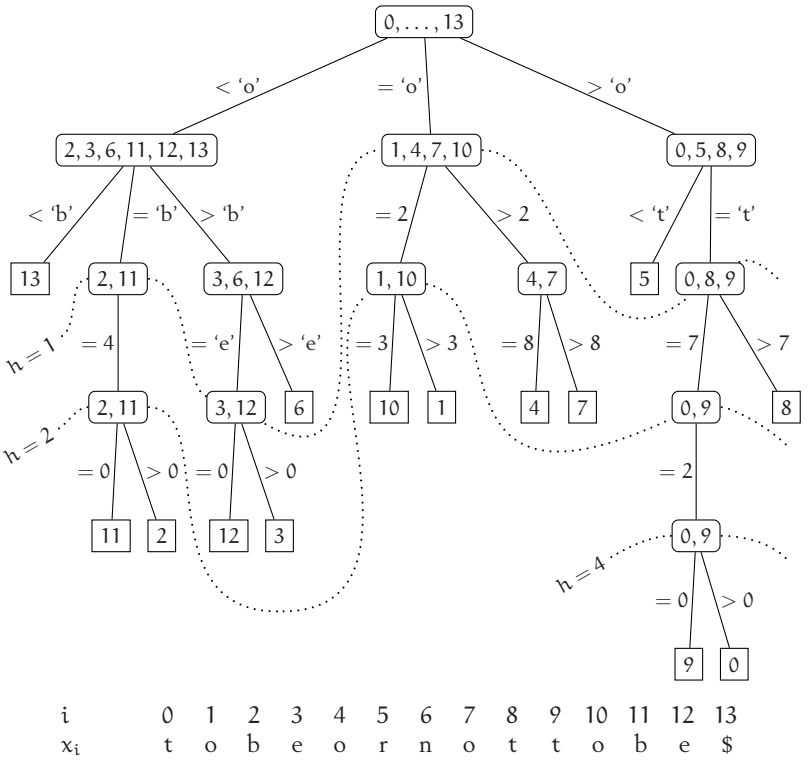
The next, single-line, section of the chart shows the keys used for the $h = 1$ sorting pass. In this pass, the sorting key of suffix i is $V[I[i]+1]$. Suffixes in groups 2, 4, 9, and 13 (i.e., subarrays $I[1\ldots2]$, $I[3\ldots4]$, $I[6\ldots9]$, and $I[11\ldots13]$) are sorted separately, according to these keys. The result, shown in the next section of the chart, is that suffixes are sorted according to their first two symbols. Groups have been split by updating $L[i]$ and $V[i]$ for i ranging over the just sorted groups.

Analogously, the next sorting pass, for $h = 2$, processes still unsorted groups (2, 7, and 12) by sorting according to $V[I[i]+2]$, and obtains the suffix order according to the first four symbols of each suffix. Finally, the single remaining unsorted group (12) is sorted according to $V[I[i]+4]$, again doubling the number of considered symbols. This concludes the suffix sorting, since the longest repeated string in the input is shorter than eight symbols, and leaves the I array holding the sorted suffix array as shown at the bottom of the chart.

4.3 Time Complexity

Consider Algorithm S on page 54. The time for the first sorting step is between $O(n)$ and $O(n \log n)$ depending on the sorting method used. Initialization of V and L in steps 2 and 3 are both performed in linear time in a left-to-right sweep. The asymptotically dominant part of the algorithm is thus the loop comprising steps 4–7, which is performed up to $\log n$ times. Clearly, the time for each run through this loop can be bounded by $n \log n$ – the time to sort the contents of I with a comparison-based sorting method – yielding an upper bound of $O(n(\log n)^2)$ for the total time complexity. However, the more detailed complexity analysis that follows shows that a worst case bound of $O(n \log n)$ is possible.

Our sorting subroutine is Quicksort with a ternary-split partition, such as the split-end partition of Bentley and McIlroy (see §4.1.2). We assume that the true median is chosen as pivot element to guarantee that the array is partitioned as evenly as possible. This requires that the median is located in linear time, for example using the algorithm of Schönhage, Paterson, and Pippenger [61], as part of the partitioning routine. In practice, this is rarely desirable, due to increased constant factors, and hardly necessary. There exists a range of pivot-choice methods which balances guaranteed worst-case versus expected performance [13].



An implicit ternary tree that corresponds to the sorting process illustrated on page 55. Suffixes processed in each partition are listed inside the corresponding nodes. Outgoing edges are labeled with relation operations and pivot keys that determine the results of partitioning. The dotted curves mark transitions between sorting passes.

For simplicity, we assume in the following analysis that the same method is used for the initial sorting in step 1 as in later passes. Employing a different sorting algorithm for initial sorting (considered in §4.4) may improve the practical behaviour of the algorithm, but does not influence the asymptotic worst case time complexity.

We view the sorting process as construction of an implicit ternary tree, which is analogous to the search tree discussed by Bentley and Sedgewick [14]. In this tree, each call to the partitioning routine corresponds to a node. The initial partitioning of the whole array corresponds to the root of the tree. Each node has three subtrees: a middle subtree which corresponds to the subarray containing elements equal to the pivot after the partitioning, and left and right subtrees corresponding to the subarrays holding smaller and larger elements respectively. All internal nodes have nonempty middle subtrees, while their left or right subtrees are empty for subarrays with less than three distinct keys. The tree has $n + 1$ leaves, corresponding to all the elements in sorted order.

An example ternary tree is shown on the preceding page. It corresponds to the same input and sorting process as the chart on page 55. Note that a different choice of pivot elements would lead to a different tree – even if the difference is only in how the median of an even number of elements is determined.

The following lemma bounds the height of the ternary tree:

4E *Lemma* The length of a path from the root to any leaf in the ternary tree is at most $2\lceil \log n \rceil + 3$.

Proof Consider first the number of middle-subtree roots on a walk from the root to a leaf in the tree. At the first such node encountered, only the first symbol of each suffix is considered by the sorting. Then, at each subsequent middle-subtree root encountered, the number of symbols considered by the sorting is twice as large as at the previous one. Consequently, the full length of any suffix is considered after encountering at most $\lceil \log n \rceil + 1$ middle-subtree roots, at which time sorting is done.

Now consider the left- and right-subtree roots. For each such node encountered on a walk from the root to a leaf, the number of leaves in its subtree is at most half compared to the previous one, since partitioning is done as evenly as possible. Thus, we are down to a single leaf after encountering at most $\lfloor \log n \rfloor + 1$ left- or right-subtree roots.

Summing the root and the maximum number of middle-, left-, and right-subtree roots on a path, we have a path length of at most $\lceil \log n \rceil + \lfloor \log n \rfloor + 3 \leq 2\lceil \log n \rceil + 3$. □

We now consider the amount of work that corresponds to each depth level of the ternary tree.

4F *Lemma* Partitioning operations corresponding to all the nodes of any given depth of the tree takes at most $O(n)$ time.

Proof Partitioning a subarray takes time linear in its size. The initial array, whose partitioning corresponds to the root, has $n + 1$ elements, and since no overlapping subarrays are ever assigned to different subtrees of any node, the total number of elements in all subarrays at any given depth is at most $n + 1$. The total time for partitioning at this depth is thus $O(n)$. □

We can now state the following tight bound:

4G *Theorem* Suffix sorting with Algorithm S can be done in $O(n \log n)$ worst case time.

Proof Partitioning asymptotically dominates sorting time; splitting and combining groups is done in linear time on subarrays which are

already sorted.

From lemma 4F, the total partitioning cost is at most $O(n)$ times the height of the ternary tree. Lemma 4E implies that the height of the tree is $O(\log n)$, and consequently the total partitioning time is $O(n \log n)$. □

4.4 Algorithm Refinements

This section lists a number of refinements that reduce the time and space requirements of Algorithm S. These are incorporated in the practical implementation described in §4.5.1.

Eliminating the Length Array The only use of the information stored in the array L is to find right endpoints of groups in the *scanning-and-sorting* phase of the algorithm (step 4 in Algorithm S on page 54). For combined sorted groups, this is needed in order to skip over them in constant time, and for unsorted groups to use the endpoint as a parameter to the sorting subroutine. However, the endpoint of unsorted groups is directly known without using L, since it is equal to the group number according to definition 4C, and can therefore be obtained from V. 4.4.1

Consequently, we need only find alternative storage for the lengths of combined sorted groups to be able to get rid of the L array. For this, note that once a suffix has been included in a combined sorted group, the position in I where it resides is never accessed again. Therefore, we can reuse the subarrays of I that span sorted groups for other purposes, without compromising the correctness of the algorithm.

Of course, overwriting parts of the I array with other information means that it does not hold the desired output, the sorted suffix array, when the algorithm terminates. However, the information needed to quickly reconstruct this is present in V. When the algorithm finishes, all parts of the suffix array are sorted groups, and since V holds group numbers of unit-length sorted groups, it is in fact at this point the inverse permutation of the sorted suffix array. Hence, setting $I[V[i]] \leftarrow i$ for all $i \in [0, n]$ reconstructs the sorted suffix array in I.

This allows us to use the first positions of each combined sorted group for storing its length. To distinguish it from the suffix numbers of other positions, we store the negated length. When we probe the beginning of the next group in the left to right scanning-and-sorting step, we check the sign of the number $I[i]$ in this position. If it is negative, $I[i \ldots i - I[i] + 1]$ is a combined sorted group; otherwise $I[i \ldots V[I[i]]]$ is an unsorted group.

4.4.2 *Combining Sorting and Updating* After each call to the sorting routine, Algorithm S scans the processed parts twice, in order to update the information in V and L. This is true both for the initial sorting step and for each run through the loop in steps 4–7. We now show how this additional scanning can be eliminated.

First, note that concatenating adjacent sorted groups, to obtain the maximal combined sorted groups, can be delayed and performed as part of the scanning-and-sorting step (step 4) of the *following* iteration. This change is straightforward.

Furthermore, all other updates of group numbers and lengths can be incorporated in the sorting subroutine. This change requires some more consideration, since changing group numbers of some suffixes affects sorting keys of other suffixes. Therefore, updating group numbers before all unsorted groups have been processed must be done in such an order that no group is ever, not even temporarily, given a lower group number than a group residing in a higher part of the I array. With the ternary-split sorting routine we use, this poses no difficulty. We give the sorting routine the following schedule:

1 Partition the subarray into three parts: smaller than, equal to, and larger than the pivot.
2 Recursively sort the *smaller* part.
3 Update group number and size of the *equal* part, which becomes a group of its own.
4 Recursively sort the *larger* part.

Since the group numbers stored in V never increase – splitting groups always only involves *decreasing* group numbers – this keeps the sorting keys consistent.

This change may still influence the sorting process, but only in a positive direction. Some elements may now be directly sorted according to the keys they would otherwise obtain *after* the current sorting pass, and this effect may propagate through several groups. Although this does not affect the worst case time complexity, it causes a nontrivial improvement in time complexity for some input distributions.

4.4.3 *Input Transformation* If we assume that the input alphabet is small enough for a symbol to be represented as a nonnegative integer (which is invalid for only a few, less than practical, machine models), we can start by transferring the contents of X to V, and perform the initial sorting in step 1 using V[i] as the key for suffix i. This has the following potential advantages, which to some degree all originate from McIlroy and McIlroy [49]:

- By setting $h = 0$, we can use the exact same sorting subroutine for initial sorting as for subsequent sorting passes.
- Since we no longer access X, we do not need to keep it in primary storage during sorting. Indeed, if we do not wish to retain X, we can overlay V on X, eliminating the memory usage for this array completely.
- When transferring symbols from X to V, the alphabet can undergo any transformation as long as the order between the suffixes is maintained.

The implementation of McIlroy and McIlroy requires an alphabet transformation that represents the unique \$ symbol with zero, and maps the original symbols to integers in the range $[1, k')$, where $k' - 1$ is the number of distinct symbols in the input. This transformed alphabet facilitates bucket sorting – essential in this implementation, since it is based on the Manber-Myers algorithm.

We now develop alphabet transforms that our algorithm can benefit from even though we do not use bucket sorting (except possibly for initial sorting, see §4.4.4). We assume for the remainder of this section that the input consists of integers in the range $[l, k)$, not counting the \$ symbol. In other words, k is the size of the input alphabet, and $l \in [0, k)$ is a lower bound for the lowest-numbered symbol that occurs in a specific input string.

The possibility to introduce an explicit representation of the \$ symbol is a small but convenient effect of alphabet transformation. The simplest way to achieve this is to set $V[i]$ to $x_i - l + 1$ for all $i \in [0, n)$ when transferring from X, and set $V[n]$ to zero. Now, the rest of the algorithm does not have to pay any attention to range or alphabet limits.

A transform with direct impact on time complexity, related to a variation described by Manber and Myers [47, page 944], is possible when the input range is small enough for several symbols to be aggregated into one integer. Let K denote $k - l + 1$, the upper bound on the size of the set of occurring symbols in the input, including \$, and let r be the largest integer such that $K^r - 1$ can be held in one machine word. Now, for all $i \in [0, n]$, set

$$V[i] \leftarrow \sum_{j=1}^{r} x_{i+j-1} \cdot K^{r-j}$$

where we define $x_i = 0$ for $i \geq n$.

This has the effect that initial sorting, where $V[i]$ is used as the key for suffix i, concerns not only the first symbol of each suffix, but the first r symbols. Therefore, subsequent sorting can start with h set to r instead of 1, and the number of sorting passes is reduced.

The transform can be computed in linear time independent of r through the alternative form

$$V[i+1] \leftarrow (V[i] \bmod K^{r-1}) \cdot K + x_{i+r}$$

for $i > 0$. If K is rounded up to the nearest power of two, the multiplication and modulo operations can be replaced by faster *shift* and *and* operations.

Since r is highly dependent on K and thereby on k and l – the limits of the input alphabet range – it can be fruitful to tighten these limits as much as possible before computing the transform. Checking the minimum and maximum symbol values that actually occur in the input and adjusting k and l accordingly is a simple task that commonly yields a noticeable improvement.

A further improvement can be gained in many cases by compacting the alphabet prior to the symbol aggregating transform. Denote the set of symbols that occur in the input $\Sigma = \{s_1, \ldots, s_{|\Sigma|}\}$, where $s_i < s_j$ if and only if $i < j$. Replacing each symbol s_i in the input with its ordinal number i allows us to set $l = 0$ and $k = |\Sigma|$. If only a small subset of the allowed input alphabet is used, this can result in a substantially larger value of r than would otherwise be possible.

With a maximum used range size $K_0 \leq k$ for the original alphabet, we can, unless K_0 is very large, compute the preparatory compaction transform efficiently using an auxiliary array of size K_0 (which may be overlaid on I). Positions in the array corresponding to used symbol numbers are marked, and ordinal numbers then accumulated in the same array. The time complexity is $O(n + K_0)$.

4.4.4 *Initial Bucket Sorting* The initial sorting step is quite separate from the rest of the algorithm and is does not need to use the same sorting method as later passes. Since this step must process all of the input in one single sorting operation, a substantial improvement can be gained by using a linear-time bucket sorting algorithm, instead of a comparison-based algorithm that requires $\Omega(n \log n)$ time.

At this stage, the array I does not yet contain any data. Therefore, if the alphabet size is at most $n+1$, we can use I as an auxiliary bucketing array, not requiring any extra space. If the input alphabet is larger than $n+1$ and cannot be readily renumbered, we cannot use this technique. However, in practice, this is unusual unless n is very small, in which case there is no need for a sophisticated sorting algorithm. (Note also that the Manber-Myers suffix sorting algorithm and similar techniques cannot function at all if the alphabet size is larger than $n + 1$.)

An even more substantial improvement can be gained by combin-

ing bucket sorting with transformation of the input alphabet as described in §4.4.3. In this case, when choosing the value of r – the number of original symbols to aggregate into one – we require not only that $K^r - 1$ can be held in one machine word, but also that it is at most n. The resulting transformed alphabet can be larger than the original one, but still allows bucket sorting without allocating extra space. Thus, using only linear-time preprocessing, we allow the initial order of the suffixes to be sorted according to the first r symbols of each suffix. This commonly takes a substantial load off the main sorting routine.

4.5 Implementation and Experiments

This section describes a practical implementation of the proposed suffix sorting algorithm, and an experimental comparison between this and other suffix sorting methods.

Implementation We describe an implementation of our algorithm that includes the refinements of §4.4, and present source code in the C programming language [38]. Since the details for implementation of alphabet transformation (described in §4.4.3) and bucket sorting (described in §4.4.4) are not central to this work, we omit the source code for the functions that perform those operations. The full implementation, including alphabet transformation and bucket sorting, is found in appendix B. 4.5.1

The main suffix sorting routine is shown on the next page. The parameters to this function are two pointers x and p to arrays that are to be used as the V and I arrays of the algorithm, and integers representing n, the input length, and the input alphabet limits k and l (see §4.4.3). When this function is called, the input should already have been transferred to the V array (which thus holds nonnegative integers in the range $[l, k)$, representing the input string), but the alphabet not yet transformed, other than possibly with the initial compaction described in the last two paragraphs of §4.4.3. On return, the contents of this array has been transformed to the inverse of the sorted suffix array held in the I array.

The *suffixsort* function first sets global variables that allow the arrays to be accessed by other functions, then enters the alphabet transformation and initial sorting phase.

The *transform* function called in this phase implements techniques described in §4.4.3. It transforms the alphabet and changes the contents of V accordingly, while maintaining the lexicographic order between suffixes:

The function *suffixsort*. Parameter x points to an array representing the input; p to an array that is to hold the suffix array. On return, x holds the inverse of x. V, I, h, and r are global variables in the program.

```
void suffixsort(int *x, int *p, int n, int k, int l)
{
   int *pi, *pk;
   int i, j, s, sl;

   V=x; I=p;                    /* set global values.*/
   if (n >= k-l) {              /* if bucketing possible,*/
      j=transform(V, I, n, k, l, n);
      bucketsort(V, I, n, j);   /* bucketsort on first r positions.*/
   } else {
      transform(V, I, n, k, l, INT_MAX);
      for (i=0; i<=n; ++i)
         I[i]=i;                /* initialize I with suffix numbers.*/
      h=0;
      sort_split(I, n+1);       /* quicksort on first r positions.*/
   }
   h=r;                         /* symbols aggregated by transform.*/

   while (I[0] >= -n) {         /* while not single combined group.*/
      pi=I;                     /* pi is first position of group.*/
      sl=0;                     /* sl is neg. length of sorted groups.*/
      do {
         if ((s=*pi) < 0) {
            pi-=s;              /* skip over sorted group.*/
            sl+=s;              /* add negated length to sl.*/
         } else {
            if (sl) {
               *(pi+sl)=sl;     /* combine sorted groups left of pi.*/
               sl=0;
            }
            pk=I+V[s]+1;        /* pk-1 is end of unsorted group.*/
            sort_split(pi, pk-pi);
            pi=pk;              /* next group.*/
         }
      } while (pi <= I+n);
      if (sl)                   /* if I ends with a sorted group.*/
         *(pi+sl)=sl;           /* combine sorted groups at the end.*/
      h=2*h;                    /* double sorted-depth.*/
   }
   for (i=0; i<=n; ++i)         /* reconstruct array from inverse.*/
      I[V[i]]=i;
}
```

- $V[n]$ is set to zero, representing the \$ symbol, and the previous n cells of the V array are assigned positive integers.
- r symbols of the original alphabet are aggregated into one, where r is the maximum integer such that $K^r \leq q$, K is the smallest power of two such that $K > k - l$, and q is the last parameter in the call to *transform*. The value of r is kept as a global variable.

  The transformed alphabet is $\{0, \ldots, j-1\}$ for some alphabet size $j \leq q+1$, where 0 represents the unique \$ symbol and q is a parameter to the *transform* function. The value returned by this function is j. (To simplify the bucket sorting routine, our *transform* implementation also under some circumstances *compacts* the alphabet after symbol aggregation, so that all integers less than j occur at least once in V.)

We adapt the use of *transform* to the sizes of the input and the input alphabet. If n is large enough for the I array to hold all the symbol buckets for the given alphabet range, i.e., if $n \geq k-l$, we call *transform* with the q parameter set to n. This guarantees that bucketing is still possible for the transformed alphabet. We then use bucket sorting for initialization of I through a call to a separate function *bucketsort*.

If the given alphabet range is larger than n we do not use bucket sorting, since this would require extra space. In this case, we may just as well use the largest possible symbol aggregation, so we call the *transform* function with q value INT_MAX. Then we initialize the I array with the numbers 0 through n, and use our main ternary-split Quicksort subroutine *sort_split* for initial sorting. By setting h to zero before the call to *sort_split*, we get the desired effect that the contents of $V[i]$ is used as the sorting key for suffix i.

This concludes the initialization phase. The suffix array has been sorted according to the first r symbols of each suffix, i.e., we can set h to r. The contents of I are suffix numbers for unsorted groups, and negative group length values for sorted groups, according to the scheme described in §4.4.1. (At this point, the sorted group length values are all -1, since the groups have yet to be combined.)

The main *while* loop of the routine runs for as long as the I array does not consist of a single combined sorted group of length $n+1$, i.e., until the first cell of I has got the value $-(n+1)$. The inner part of the loop consists of combining sorted groups that emerged from the previous sorting pass with each other, and with previously combined sorted groups, and refining the order in unsorted groups through calls to the function *sort_split*. This process follows the description in §4.4.1 and §4.4.2.

Finally, I, now filled with negative numbers denoting lengths of sorted sequences, is restored to the sorted suffix array from its inverse permutation, which the algorithm has produced in V. If the application of suffix sorting is Burrows-Wheeler transformation, this step can be replaced by an analogous one that computes the transformed string instead.

The ternary-split Quicksort routine is shown on the next page. The implementation is directly based on *Program 7* of Bentley and McIlroy [13] with two exceptions, the sorting method for the smallest subarrays, and the incorporation of group updates. The choice of pivot element is in a separate function *choose_pivot*. Our implementation uses the same *ninther* strategy as Bentley and McIlroy. Other possibilities are, for instance, using the true median (as we assumed for guaranteed worst case performance in §4.3) or a random choice.

§ 4.5.1

The function *sort_split*. Parameters are beginning of a subarray and its number of elements. The function *choose_pivot* returns the key for one element in the subarray.

```
static void sort_split(int *p, int n)
{
   int *pa, *pb, *pc, *pd, *pl, *pm, *pn;
   int f, v, s, t, tmp;
#  define KEY(p)                (V[*(p)+(h)])
#  define SWAP(p, q)            (tmp=*(p), *(p)=*(q), *(q)=tmp)

   if (n<7) {
      select_sort_split(p, n);  /* special sorting for smallest arrays.*/
      return;
   }
   v=choose_pivot(p, n);
   pa=pb=p; pc=pd=p+n-1;
   while (1) {
      while (pb<=pc && (f=KEY(pb))<=v) {
         if (f==v) { SWAP(pa, pb); ++pa; }
         ++pb;
      }
      while (pc>=pb && (f=KEY(pc))>=v) {
         if (f==v) { SWAP(pc, pd); --pd; }
         --pc;
      }
      if (pb>pc) break;
      SWAP(pb, pc); ++pb; --pc;
   }
   pn=p+n;
   if ((s=pa-p)>(t=pb-pa)) s=t;
   for (pl=p, pm=pb-s; s; --s, ++pl, ++pm) SWAP(pl, pm);
   if ((s=pd-pc)>(t=pn-pd-1)) s=t;
   for (pl=pb, pm=pn-s; s; --s, ++pl, ++pm) SWAP(pl, pm);

   s=pb-pa; t=pd-pc;
   if (s>0) sort_split(p, s);
   update_group(p+s, p+n-t-1);
   if (t>0) sort_split(p+n-t, t);
}
```

Group updates are handled in the last section of the routine, between the recursive calls, as explained in §4.4.2. This is implemented as the separate function *update_group*, shown on the facing page. This function takes as parameters pointers to the first and last positions of a subarray that is to constitute a group of its own, and updates the corresponding group numbers in V – unless the result is a unit-length group in which case it is registered as sorted through a -1 value in V.

For fast handling of very small subarrays, we use a nonrecursive sorting routine for subarrays with less than 7 elements, implemented as a separate function. Since group updating is difficult in insertion sorting – the common algorithm to use in this situation – we use a variant of selection sorting that picks out one new group at a time, left to right, by repeatedly finding all elements with the smallest key value and moving them to the beginning of the subarray. This is easily combined with group updating.

```
static void update_group(int *pl, int *pm)
{
   int g=pm-I;                  /* new group number.*/
   V[*pl]=g;                    /* update group number.*/
   if (pl==pm) *pl=-1;          /* one element, sorted group.*/
   else do                      /* more than one element, not sorted.*/
      V[*(++pl)]=g;             /* update group numbers.*/
   while (pl<pm);
}
```

The function *update_group*. Called with first and last positions of a subarray to be a single group.

4.5.2

Experimental Results We report suffix sorting time for various inputs. We use a Sun Ultra 60 workstation (360 MHz Ultrasparc II CPU and 2 GB primary storage) running Solaris 2.6. The programs were compiled with the Gnu C compiler version 2.7.2.3, with option −O3 for maximum optimization. The reported times are user times, measured with the *rusage* command.

The list of programs included in the comparison is shown on the next page. The *htr2ar*, *tr2ar*, and *bese* programs were kindly supplied by Stefan Kurtz of the University of Bielefeld. The first two of these are based on suffix trees implemented using Kurtz's space reduction techniques [40]. The *htr2ar* code originates from an application with limited input length; it is unable to handle our largest test files.

The *mcil* program is the implementation by McIlroy and McIlroy [49], referred to in §4.1.1 and §4.4.3. It uses a variant of the Manber-Myers algorithm [47], with improvements that yield better performance than a direct implementation of that algorithm. The implementation originally contains error checks and calculation of parameters that we regard as inputs. These computations, which would lead to unjustly large execution times, have been removed in our experiments. Because of the input requirements of this implementation, the same input alphabet computation as for *qss2* is incorporated in *mcil*.

As example input, we use a set of large files, listed on the next page. The files are chosen to demonstrate the behaviour of the programs for different kinds of natural data as well as degenerate cases. The files that are part of the Calgary or Canterbury corpora are available via *ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus/* or *http://corpus.canterbury.ac.nz/fileset.html* respectively.

The table on page 69 shows sorting time of the algorithms, listed with average and maximum LCP length for each file, which gives a good estimate of the repetitiveness of the files. (Maximum LCP is equivalent to the longest repeated string.) The top section of the table lists the results for the full sized natural data files, and the lower sections list results for generated and truncated files of equal length,

Algorithm implementations participating in the comparison.

program	algorithm
htr2ar	Kurtz's suffix tree implementation with hash table representation (IHTI).
tr2ar	Kurtz's suffix tree implementation with linked list representation (ILLI).
mcil	Suffix sorting implementation by McIlroy and McIlroy using an improved version of the Manber-Myers algorithm.
bese	String sorting algorithm of Bentley and Sedgewick (see §4.1.3) with an initial bucket sorting step. Implementation by Kurtz.
qss0	Our algorithm with input alphabet size 256.
qss1	Our algorithm with input alphabet limits k and l set according to the input (see §4.4.3).
qss2	Our algorithm with compacted input alphabet (see §4.4.3).

Input data set used for algorithm comparison.

file	contents	size
maini	All articles of the Japanese newspaper *Mainichi* during 1995.	109 442 894
patent	A collection of Japanese patent claims.	89 229 120
reuters	The Reuters corpus.	27 636 766
html	A collection of html files from servers in Japan.	125 595 037
calg	Concatenation of the original Calgary corpus files except *pic* (13 files).	2 628 406
cant	Concatenation of the Canterbury corpus files except *ptt5*.	2 297 568
pic	A Calgary corpus file (the same as *ptt5* of the Canterbury Corpus).	513 216
ecoli	The file E.coli of the large Canterbury corpus.	4 638 690
bible	The file bible.txt of the large Canterbury corpus.	4 047 392
world	The file world192.txt of the large Canterbury corpus.	2 473 400
aaaa64k	The letter 'a' repeated $64 \times 1\,024$ times.	65 536
aaaa2M	The letter 'a' repeated two million times.	2 000 000
—2M	First two million bytes of the corresponding file.	2 000 000
—8M	First 8 191 kB of the corresponding file.	8 387 584

which give normalized timing results. Within each section, the files are listed in order of increasing average LCP.

The table shows that the simple, non-specialized, string sorting implementation *bese* is the fastest when average LCP is small, but not much faster than the *qss* programs that implement our algorithm. When repeated strings are longer, the *qss* programs are more efficient, and for extremely repetitive input, the suffix tree implementations have an advantage. For the most repetitive files, *bese* degenerates to

file	avg LCP	max LCP	*htr2ar*	*tr2ar*	*mcil*	*bese*	*qss0*	*qss1*	*qss2*
cant	9.0	738	8.4	15.7	24.1	**3.7**	4.0	4.0	4.2
bible	14.0	551	20.4	13.8	72.6	**9.1**	12.0	10.7	10.7
calg	14.6	1706	12.5	11.8	43.2	**5.0**	5.7	5.7	5.8
ecoli	17.4	2815	29.2	17.6	101.1	**8.5**	17.3	13.5	9.8
maini	20.1	5918	—	1109.2	5499.9	**415.8**	537.1	539.4	536.7
world	23.0	559	11.1	7.6	39.1	8.0	6.7	**6.0**	6.1
patent	41.4	8923	—	545.7	3663.7	398.6	**390.1**	385.9	392.2
reuters	50.9	4975	—	120.3	713.4	161.6	115.0	103.6	**103.3**
html	606.4	99125	—	953.2	6450.5	3521.3	**585.0**	586.1	585.9
pic	2353.4	36316	1.6	**0.8**	3.3	53.3	0.9	0.9	0.9
maini8M	19.3	4701	40.2	50.9	205.4	**21.3**	24.0	23.8	21.4
patent8M	38.1	2027	39.9	33.9	160.6	29.5	**25.5**	25.6	26.1
reuters8M	50.3	4967	36.7	31.0	199.0	41.7	29.0	**25.9**	26.5
html8M	849.6	73344	38.8	40.7	238.2	301.6	25.4	**25.4**	25.9
cant2M	8.3	228	7.4	15.3	14.6	**3.1**	3.2	3.2	3.3
maini2M	10.0	1032	9.3	10.7	33.0	**3.5**	4.1	4.1	4.2
calg2M	11.0	1029	9.9	9.0	32.4	**3.6**	4.3	4.3	4.4
ecoli2M	12.9	1345	11.7	7.1	34.2	**3.1**	6.0	4.7	3.5
bible2M	14.7	551	9.4	6.3	30.6	**4.1**	5.0	4.5	4.4
world2M	22.9	559	8.8	6.3	30.2	6.5	5.1	**4.7**	4.8
patent2M	31.6	1439	9.2	7.0	29.6	5.4	4.5	**4.5**	4.6
reuters2M	47.1	4967	8.6	6.3	36.4	8.1	5.0	**4.6**	4.7
html2M	252.1	27110	9.0	8.9	36.9	21.0	4.0	**4.0**	4.1
aaaa2M	999999.5	1999999	4.4	**1.8**	11.4	—	5.8	5.1	5.2
aaaa64k	32767.5	65535	0.1	**0.1**	0.2	92.8	0.1	0.1	0.1

Sorting times in seconds. Average and maximum LCP, *longest common prefix* length for adjacent suffixes in sorted order, is listed at the left for each file. Files are in order of increasing average LCP. Lowest time for each file is in bold face. The three lower sections list files with homogenized sizes.

quadratic time complexity. Since the *bese* program is unable to handle the *aaaa2M* file, we include the smaller file *aaaa64k* to illustrate the extremely poor behaviour of *bese* for this kind of data.

It is interesting to note that *mcil* is slower than the *qss* programs for all the files, even though *mcil* implements the Manber-Myers algorithm which is also specialized for suffix sorting and has the same worst case time complexity as our algorithm. Indeed, these experiments indicate that the Manber-Myers algorithm performs very badly for large files, even for natural, non-degenerate, input data. When maximum LCP is large, *mcil* becomes slow, since the number of passes in this algorithm is the logarithm of maximum LCP length, and each pass has to process the full input string. In our algorithm, the speed is not much influenced by maximum LCP, because in later passes most suffixes are already sorted and skipped.

Note that the difference between *qss* and *mcil* is fairly small for *aaaa2M*, whose average and maximum LCP are both large, which causes the unsorted parts to shrink slowly. For *ecoli* on the other hand, the difference between these algorithms is large, since average LCP is small but maximum LCP is large.

Although *htr2ar* is the only program that uses an algorithm with expected linear worst case performance, it is not the fastest for any of the inputs. The other suffix tree implementation, *tr2ar*, uses linked lists for storing edges, which means that the input alphabet is a factor in its time complexity. This program is slightly faster than those using our algorithm for the most repetitive natural data file *pic*, and the fastest without comparison for the generated file *aaaa2M*, whose input alphabet size is one.

Input alphabet compaction clearly helps when the input alphabet is small. This is noticeable particularly for *ecoli*, which is in the four symbol alphabet of DNA sequences, causing *qss2* to be much faster than *qss0* and *qss1*.

Our algorithm is the fastest for files whose average LCP is neither terribly small nor large. Moreover, it exhibits robust behaviour over all the inputs: the difference in speed between our algorithms and the fastest one is small for all files.

Chapter Five

Suffix Tree Source Models

Lossless compression of a string involves maintaining, in some form, statistics for the occurrences of its substrings. Indeed, this is commonly the computationally dominant part of the compression algorithm. Thus, the necessity for efficient string data structures in sequential data compression is clear. In this chapter presents several instances where suffix trees contribute to this field. In particular, we emphasize use of the sliding window scheme given in chapter two.

In the context of dictionary-based compression, our techniques provide a robust expected linear-time complexity, independent of the input – a property that common implementation techniques do not have. For predictive modelling, our contribution is even more notable, as it assists in making schemes that are among the most theoretically prominent available for efficient practical use.

5.1 Ziv-Lempel Model

The dictionary-based family of algorithms originating from Ziv and Lempel [71, 72] comprise perhaps the conceptually simplest source model of all. The idea of these schemes is to incrementally construct a *dictionary* – a set of *phrases*, strings that occur in the input – and produce an output consisting of references to the dictionary. Apart from an initial part that typically consists of the individual symbols of the input alphabet, the dictionary is constructed exclusively from already processed parts of the input, which implies that the dictionary

need not be explicitly transferred. Instead, the compression and decompression algorithms share the same rules for creating new phrases, causing them to build analogous dictionaries, and sharing the same set of phrases at all times.

In the LZ-77 family of algorithm, originating from the first presented algorithm of Ziv and Lempel [71], the idea is to let the dictionary comprise *all* strings in the previous part of the input. In each iteration, the previous part of the input is searched for a string that matches the following part (usually, the longest match is used), and then the position and length of the match are output. If no match is found, the next symbol of the input is transferred explicitly to the output.

Since primary storage is never unlimited, the dictionary cannot be allowed to grow indefinitely in practice. The algorithm must be augmented in some manner to be able to handle large inputs. One possibility is to block the input into smaller parts, restarting the model from scratch at each new block. However, this may yield considerably worse compression, since the beginning of each block is compressed using a very small dictionary. As a more attractive approach to handling strings of unlimited length, it is common to store the latest part (typically several thousand symbols) of the processed part of the input in a buffer, and limit the search for the longest match to this buffer.

A suffix tree can locate the longest matching substring of its indexed string in time proportional to the length of the match, and can be constructed in linear time. Hence, continuously maintaining a suffix tree for the buffer supplies an ideal situation for locating the longest previous string matching the input.

Rodeh, Pratt, and Even [57] consider this possibility, and observe that it is possible to implement a linear-time LZ-77 algorithm by utilizing a suffix tree. They also consider moving the indexed string along the input to support a finite buffer, by pacing over the string with three simultaneous suffix trees, each of maximum size proportional to the buffer size. Asymptotically, this solves the finite buffer problem, but it introduces substantial overhead in time and, particularly, space requirements. Indeed, in a survey of string searching algorithms for LZ-77 compression, Bell and Kulp [11] rule out suffix trees because of the inefficiency of deletions.

Using our sliding window scheme given in chapter two, this inefficiency can be eliminated. The index is incrementally expanded to include newly processed parts of the input using the *front* increment procedure consisting of the procedure on page 25 with the augmentations on page 30, and the back end of the input is moved forward the

same number of positions using the *tail* increment procedure given on page 31, once the size of the index has reached the buffer size.

The time required for searching is proportional to the total length of the matching strings located with the suffix tree, which does not exceed the total length of the input. Thus, we have the following:

Theorem 5A An LZ-77 algorithm using a buffer of maximum size M implemented using our sliding window indexing scheme processes an input of size n in expected $O(n)$ time, using $O(M)$ storage space.

An LZ-77 implementation that uses a simple hashing scheme, which is common, and advocated by Bell and Kulp [11], does not have this robust worst case complexity. When the input is repetitive, many equal substrings are entered into the hash table, causing a large number of collisions. Since a number of strings sharing the same hashed sample need to be scanned in each step of the algorithm, this may lead to $\Omega(M^2)$ time complexity – independently of the hashing scheme.

5.2 Predictive Modelling

Some of the most effective results in data compression have been achieved by statistical source modelling in combination with arithmetic coding. Specifically, PPM, prediction by partial matching, has generated notable results. The original PPM algorithm was given by Cleary and Witten [19]. A plethora of improvements and analyses has been presented since [1, 15, 18, 32, 51].

The idea of PPM is to regard the last few symbols of the input stream as a *context*, and maintain statistical information about each context in order to predict the next symbol; i.e., to estimate a probability distribution for which symbol follows the current context. The length of the string used as a context is referred to as the *order*.

For each context, a table of symbol counts is dynamically maintained, and the code applied whenever that context occurs is based on the statistics of this table. The higher the count of a certain symbol in the current context, the larger the code space allocated to it. The low level encoding is usually performed with arithmetic coding.

When a symbol appears in a context for the first time, its count in that context is zero. Still, it must be possible to encode the symbol in that context, so some amount of code space must be reserved for previously unseen events. Therefore, each context also keeps an *escape* count, used to encode a *new symbol* event in that context. After an escape occurs, the algorithm falls back to the context of nearest smaller order. A (-1)-order context, where all symbols (or, possibly, only *pre-*

viously unseen symbols) are assumed to be equally likely, is maintained for symbols that have never occurred in the input stream.

5.2.1 *PPM with unbounded contexts* The most general and flexible PPM variant is the one named PPM∗ by Cleary and Teahan [18], which maintains statistics for *all* contexts that have occurred in the previous part of the input. Previous to PPM∗, the maximum order has usually been set to some small number – primarily to keep the number of states from growing too large, but also because a decrease in compression performance can be observed when the order is allowed to grow large (usually to more than about six). This is because contexts of high order make the algorithm less stable; the chance of the current context not having seen the upcoming symbol is larger. However, the performance of PPM∗ demonstrates that with a careful strategy of choosing contexts, allowing the order to grow without bounds can yield a significant improvement.

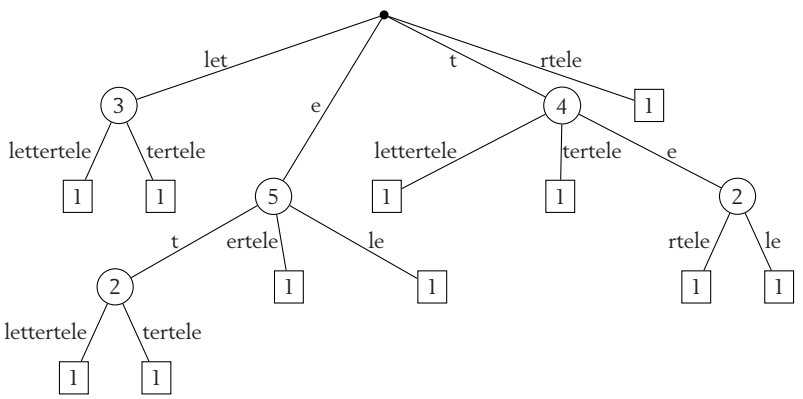
All substrings that have occurred in the input stream are stored in a trie and each node in the trie corresponds to a context. A *context list*, a linked list of all nodes whose corresponding contexts match the last part of the input stream, is maintained. For instance, if the part of the input processed thus far ends with '. . . abc', and the string 'abc' has also occurred in some previous position of the input, the context list holds the nodes corresponding to 'c', 'bc', 'abc', and possibly some longer previously occurred contexts that match the current one.

The context to use for encoding is chosen among the ones on the context list. The exact rules for which context should be used may differ. Escaping is also performed along the context list by moving one step in the direction of shorter contexts. Furthermore, in the implementation of Cleary and Teahan, the context list is used in maintaining the trie, for finding the positions that need to be updated as the model expands.

5.3 Suffix Tree PPM∗ Model

Cleary and Teahan observe that collapsing paths of unary nodes into single nodes, i.e. path compression, can save substantial space. We make some further observations that lead us to the conclusion that the suffix tree operations described in chapter two are suitable to maintain the data structure for a PPM∗ model. Again, our data structure is based on the representation given in §1.3.2.

- *A context trie is equivalent to a suffix tree* A path compressed context trie is a trie storing all substrings of the processed part of the input.



Suffix tree context trie corresponding to the input 'letlettertele'. Numbers are context counts.

Thus, this context trie *is* a suffix tree indexing this string, according to our definition in §1.3.

- *Suffix links provide context lists* The context list of the PPM∗ scheme corresponds to a chain of nodes in the suffix tree connected by suffix links. Using suffix links, it is not necessary to maintain a separate context list, since all possible lists are already present in the tree. We only need to decide which is the first node in the context list, the one corresponding to the longest context.
- *Linear number of counts is sufficient* The symbols that have nonzero counts in the table associated with a context are exactly the symbols for which the node corresponding to that context has children. Hence, if $child(u, c) = v$, the count for symbol c in context u can be stored in v. As for contexts that correspond to points residing on edges, as opposed to in explicit nodes, there is no need for additional tables of counts for the following reason. If two strings (contexts) belong to the same node, this implies that one is a prefix of the other, and that there are no branches between them. Hence, in the currently considered part of the input, they always appear with one as the prefix of the other, which implies that they occur the same number of times. Therefore, the reasonable strategy is clearly to let them have the same count, and only one instance of the count needs to be stored.

  Hence, we can use the online suffix tree maintenance techniques of Ukkonen (see §2.1) for the context trie. This is illustrated in the figure above, which shows a suffix tree for the string 'letlettertele', augmented to serve as a context trie by noting the context count in each node. For example, the number 3 in the top left internal node corresponds to the number of times the letter 'l' appears in the string, reflecting that this node is $child(root, \text{'l'})$.

  This leads us to conclude the following:

5B *Theorem* A PPM* context trie for input length n can be maintained in expected $O(n)$ time, using $O(n)$ storage space.

Note, however, that linear space in the size of the input is still not feasible for large files, or indeed, for processing unlimited streams of data. Note also that the bound concerns only the structural part of the context model; the time for statistical updates remains to be accounted for.

The connection between PPM context tries and suffix trees has also been considered by Bunton [15]. However, her work is not concerned with keeping the data structure strictly linear, and the time and space complexities of her techniques are not fully analyzed.

5.4 Finite PPM* Model

A suffix tree that is allowed to grow without bounds, until it covers the whole input, is still not a practical source model in general. For large files or long input streams, primary storage cannot even hold the complete input, let alone a suffix tree to index it.

To bound the size of the data structure to a finite amount of storage, we propose maintaining a source model that holds only the contexts appearing in the last M symbols of the processed part of the input, for some finite number M which may depend on the amount of available primary storage.

We accomplish this with the sliding window techniques of chapter two. Thus, contexts corresponding to strings occurring in the latest M symbols are always maintained, while older contexts are progressively "forgotten". Note, however, that we need not lose all information from previous parts of the input when deleting old contexts. The counts of remaining contexts can still be influenced by previous occurrence of these contexts. Analogously to theorem 5B, we have:

5C *Theorem* Maintaining a PPM* context trie limited to contexts occurring in a sliding window of maximum size M is accomplished in $O(M)$ space, and expected linear time in the size of the input.

5.5 Non-Structural Operations

Our time complexity bounds concern only maintaining the structure of the source model. In addition to this, the cost of choosing the context to use, incrementing counts of contexts as they occur in the input, and for coding remain to be accounted for. Although we do not

have final conclusions regarding these topics, and consider them for the most part to be beyond the scope of our research, we briefly state the problems that must be considered in accounting for time complexity of the overall implementation.

Context Choice and Count Updating The problems of choosing a starting context, updating counts in the tree, and handling escape events has been subject to some research [1, 15, 18]. However, these variables of the model have been considered almost exclusively regarding their implications on compression performance, while their consequences for time complexity are largely ignored. 5.5.1

The proposed updating strategy of Cleary and Teahan, which involves frequently following the context list to the end, does break linear time complexity, but its exact time requirements are not fully understood. Furthermore, it is far from clear that some slightly relaxed updating strategy requiring only amortized constant time per iteration cannot yield equally good compression. Plausibly, there is a tradeoff between speed and prediction when choosing an updating strategy, whose characteristics remain to be studied.

Coding Our suffix tree source model data structure provides statistics as individual counts. An arithmetic encoder requires a *range* of cumulative counts to be allocated for each symbol. If only individual counts are maintained, the children of the node corresponding to the context used for encoding must be scanned to compute the range. This potentially introduces a factor of $\Omega(k)$ in the time complexity of the algorithm. 5.5.2

In practice, this cost can be decreased with *move-to-front* techniques, and rarely becomes as large as a factor k. Furthermore, it can be reduced to $O(\log k)$ by storing cumulative counts in a binary tree [24, 33, 52].

The matter is more complicated if symbol *exclusion* is to be employed in connection with escape events. When a symbol is coded just after an escape, it cannot be one of the symbols that existed in the previous context – if it was, it would have been encoded in that context. Therefore, we can exclude the ranges that would have been assigned to symbols existing in previous contexts, to gain some code space for the others and decrease the size of the compressed output. It is common to use exclusion with PPM, since it yields a notable compression improvement.

Now, the problem is to compute a range from the *intersection* of two sets of symbols, which is far more complicated. Incorporating this

possibility in the data structure, while maintaining efficient asymptotic worst-case time complexity is an open problem.

5.6 Conclusions

The tight connection between pattern matching and data compression offers many possibilities for improving the practical usefulness of existing compression schemes by applying efficient data structures. Our findings show that sliding window indexing with suffix trees is a powerful tool for supporting finite source models for sequential compression.

Another interesting aspect is the insight into the fundamental relation between two quite different source modelling schemes that can be gained through considering the suffix tree. Equivalence between predictive modelling schemes and dictionary-based compression has been shown in various settings; see, for instance, Bell and Witten [12]. Our application of the same fundamental data structure to both of these compression techniques serves as a further illustration of this.

Chapter Six

Burrows-Wheeler Context Trees

Block sorting compression was originally presented by Burrows and Wheeler in 1994 [16]. Its central element is a transform which we refer to as the *Burrows-Wheeler transform*, *BWT*, that reorganizes the input string to concentrate repetitions. The transformed string can be compressed with a simple locally adaptive statistical compression scheme. Even in its most rudimentary form, BWT compression matches substantially more complex modelling schemes in compression performance, and with advances in research as well as practical implementations [10, 60, 62], its importance is growing rapidly.

While BWT may at first glance appear to be a magical new algorithm, Cleary and Teahan [18] observe that its effect is quite similar to that of PPM (see §5.2). In this chapter, we take that similarity one step further in giving the *context tree*, which is implicit in BWT, a concrete form. An important aspect is the connection between this tree and the suffix tree of the input string. 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 work springs from the observation that previous work in block sorting compression maintains much of the traditional *online* approach

BWT algorithm, producing, from the input X, a transformed string X′ and a number **i**.

1 Sort all the suffixes of X. Represent the sorted sequence as a vector $S = (s_0, \ldots, s_n)$ of numbers in the range $[0, n]$ such that i precedes j in S iff the suffix that begins in position i of X lexicographically precedes that which begins in position j.
2 Let **i** be the number such that $s_i = 1$.
3 For $i \in [0, n]$, let $x'_i = x_{s_i - 1}$, where we define $x_{-1} = x_n = \$$.

Illustration of BWT (the algorithm above) for the input string 'abcabcabc$'. The output is the number $\mathbf{i} = 3$ and the x_i column.

i	s_i	x_i	suffix i in sorted order
0	9	c	$
1	6	c	abc$
2	3	c	abcabc$
$3 = \mathbf{i}$	0	$	abcabcabc$
4	7	a	bc$
5	4	a	bcabc$
6	1	a	bcabcabc$
7	8	b	c$
8	5	b	cabc$
9	2	b	cabcabc$

of data compression, i.e. it allows the decompressor (and to some extent also the compressor) to work incrementally in one pass, updating parameters depending on only the previous part of the message. However, BWT is inherently block structured, and hence there is no apparent reason to prefer online strategies in this case. Through the exploration of the context tree, we move towards considering the full structure of the BWT, and not merely regarding it as a permutation operation.

6.1 Background

We begin with a recapitulation the basics of BWT, and a discussion of previous work. Although our formulations are somewhat different, the basis of this section is primarily Burrows and Wheeler [16].

6.1.1 *Block Sorting Transform* We assume that the input is a string X as specified in §1.1.1. The transform produces a string $X' = x'_0 \ldots x'_n$ which comprises the same symbols as X in a different order. (The algorithm and an illustration of it are shown above). The effect is that symbols followed by the same substrings in X are placed in consecutive positions in X′. Referring to the suffix following a position in X 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 X′. Note

1 For $c \in [0, k]$, let n_c be the number of occurrences of symbol c in X'.
2 Set $C[0]$ to 0. For $i = 1, \ldots, k$, set $C[i] \leftarrow C[i-1] + n_{i-1}$.
3 For $i = 0, \ldots, n$, set $P[C[x'_i]] \leftarrow i$ and increment $C[x'_i]$.
4 Set i to $\mathbf{i}$. For $j = 0, \ldots, n$, let $x_j = x'_i$ and set $i \leftarrow P[i]$.

The reverse BWT algorithm. Symbol 0 is the \$ symbol.

that interpretation of contexts is different than the one for PPM described in §5.2, where the symbols *preceding* a position define its context. If desired, the same behaviour can be emulated in BWT, simply by reversing X. However, the difference is normally of no importance.

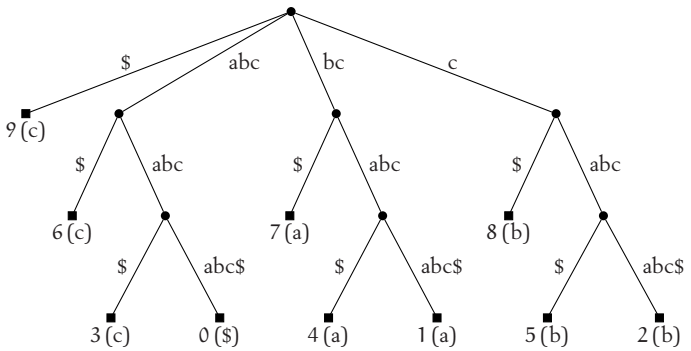
If X contains repeating patterns, some parts of X' – that originate from similar contexts – comprise only symbols from a small part of the input alphabet. By transferring the symbols to the decompressor in the order of X' instead of X, we can exploit its regularities efficiently with a simple locally adaptive compression method.

The decompression program needs to reverse the transform to obtain the original string. This remarkably fast and simple procedure is shown above.

Sorting Algorithms and Time Complexity The key advantage of BWT compression is its moderate requirements in computational resources, compared to other methods with similar compression performance. We make an effort to maintain that advantage throughout this work and avoid processes that notably increase time or space complexity. We now discuss the time complexity of BWT itself. 6.1.2

Normally, the computationally critical part of the transform is the suffix sorting, a subject thoroughly treated in chapter four. The transform, as well as the reverse transform, also requires storage and scanning of the frequency array C. This contributes an $\Omega(k)$ term to both time and space complexity, but in practice this is usually a minor component. Furthermore, this term can be avoided by preceding the transform with an alphabet compaction phase that produces a new alphabet of size $O(n)$. However, this compaction, which requires sorting the original symbols of the input, is worthwhile only if k is very large.

Existing BWT implementations typically use ad hoc combinations of sorting algorithms, often paired with a run length encoding scheme to handle common degenerate cases [16, 25, 62, 69]. A better alternative in general is the $O(n \log n)$ time sorting algorithm presented in chapter four, which is shown to perform very well in practice. However, as noted by Burrows and Wheeler [16], this can be asymptotically improved by building a suffix tree, which is then traversed in sorted order and the sorted sequence obtained from the leaves (see



Suffix tree for 'abcabcabc$'. Below each leaf is shown the number of the corresponding suffix and, in parenthesis, the symbol that the BWT would emit.

§6.2.1). The traverse takes linear time. Thus we immediately have the following corollary of theorem 1C:

6A *Corollary* The time complexity of BWT is $\Theta(n + s(n))$, where n denotes the input length and $s(n)$ the time to sort n symbols.

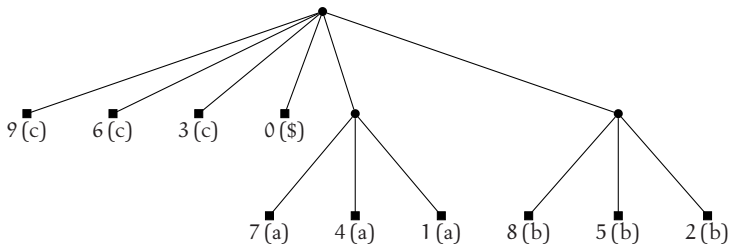
6.1.3 *Move-to-front and Related Coding* A large majority of the previous work on BWT relies on *move-to-front coding* to exploit the local repetitiveness of the transformed string [16, 25, 62, 69]. The symbols of the input alphabet are placed in a conceptual list, and the position of a symbol in this list, counting from the head starting from zero, is used to encode the symbol when encountered. Encoded symbols are immediately moved to the head of the list.

This subsidiary transformation of X' produces another string X'' of integers in the range $[0, k]$, for which the distribution is highly skewed (provided that X is compressible): low numbers are more common than high numbers. Now, the symbols of X'' can be predicted with simple zero-order statistical source model, and entropy encoded with, for example, Huffman or arithmetic coding.

Arnavut and Magliveras [8] 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.

6.2 Context Trees

We elaborate on the properties of the reorganization performed in BWT by relating it to context trees of PPM – equivalent to the suffix



Pruned context tree that corresponds to the suffix tree on the facing page.

tree source models, or context *tries*, discussed in the PPM∗ setting in chapter two. The close relation between PPM and BWT was briefly noted by Cleary and Teahan [18].

More on Suffix Trees As noted in §6.1.2, a suffix tree can be used to produce the BWT string X′: the tree is traversed left to right, and for each leaf encountered, the symbol preceding the corresponding position of X is emitted as the next symbol of X′ (see the figure on the preceding page). 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. 6.2.1

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 to that node) holds the counts for symbols occurring in the context w. Thus, the suffix tree incorporates exactly the structure of a suffix tree source model in PPM∗, as described in §5.2. (Note however, that since our contexts are the strings *after* each position, the tree representation is "backwards" compared to most PPM descriptions.)

Pruning the Tree Maintaining 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. 6.2.2

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 the figure on the preceding page). 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 a finite tree model that adequately 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 affect 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, we increase the escape count instead of the count for 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 sym-

bols 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. Denote $U = \{\, c \mid n_{c,u} + e_c > 0 \,\}$, 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.

Code Length Measurements To illustrate how the context tree captures the statistics of a file, the table on the next page shows experimental results using the files of the Calgary corpus as input (available at *ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus/*). Note that the reported code lengths are not compression results, since information about the tree structure is not included (see §6.4), but rather lower bounds for performance of the greedy-pruned context tree. 6.2.3

The measurements show how much redundancy the context tree is able to capture for different kinds of data. Perhaps the most interesting interpretation of these data is as an approximation of the lower bound for *any* tree model based compression method, including PPM, when compressing data with these characteristics – provided that the pruned context tree is an adequate approximation of the optimal context tree. However, the relatively large code lengths – not a general improvement over the best predictive models – indicate that a pruning strategy that brings the context tree closer to the global optimum should be desirable for maximum compression.

Measured 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* the calculated code length in bits per symbol.

	size	nodes	bits
bib	111 261	10 248	1.80
book1	768 770	59 701	2.19
book2	610 856	50 286	1.87
geo	102 400	5 442	4.37
news	377 109	40 568	2.28
obj1	21 504	1 972	3.45
obj2	246 814	27 037	2.27
paper1	53 161	6 265	2.26
paper2	82 199	8 338	2.22
pic	513 216	7 499	0.78
progc	39 611	4 939	2.29
progl	71 646	7 958	1.59
progp	49 379	5 922	1.62
trans	93 695	10 328	1.42

6.3 *The Relationship between Move-to-front Coding and Context Trees*

We now review the move-to-front encoding described in §6.1.3 from a context tree perspective, in order to shed light on some important points regarding its performance.

The transform of X′ into X″ serves to replace the local repetitions of X′ by a globally skewed distribution that would ultimately submit to compression using a static code. However, static coding is a poor choice. While lower numbers are indeed generally more common than high numbers in X″, 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 [25] reports the same average as the PPM* algorithm [18] for the Calgary corpus. The key to this performance lies in the extreme degree of repetition in X′ 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.

6.4 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. In this section 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.

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. 6.4.1

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 leaves below are removed.

6.4.2 *Encoding the Tree* A pruned context tree is 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. A more sophisticated tree encoding method could be based on existing specialized tree compression methods, such as those of Katajainen and Mäkinen [37].

6.4.3 *Encoding the Symbols* For encoding the symbols symbols of the transformed string, 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 is 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.

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 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.

In this setting, 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 [32], full exclusion, and update exclusion.

6.4.4 *Experimental Results* The table on the facing page shows the results of our experimental compression program. The limits chosen for the

	size	nodes	bits	(tree + sym)
bib	111 261	2 308	2.26	(0.28 + 1.98)
book1	768 770	7 777	2.37	(0.15 + 2.22)
book2	610 856	8 793	2.17	(0.21 + 1.96)
geo	102 400	899	4.69	(0.13 + 4.56)
news	377 109	7 350	2.76	(0.26 + 2.50)
obj1	21 504	516	4.27	(0.32 + 3.95)
obj2	246 814	6 303	2.94	(0.33 + 2.61)
paper1	53 161	1 384	2.84	(0.35 + 2.49)
paper2	82 199	1 634	2.65	(0.28 + 2.37)
pic	513 216	652	0.80	(0.02 + 0.78)
progc	39 611	1 071	2.92	(0.36 + 2.56)
progl	71 646	1 778	2.13	(0.33 + 1.80)
progp	49 379	1 256	2.22	(0.34 + 1.88)
trans	93 695	2 775	2.06	(0.37 + 1.69)

Results of the experimental compression program. *Size* is original file size in bytes, *nodes* number of internal nodes maintained, and *bits* average number of bits per compressed symbol. *Tree* and *sym* show individual code space for tree and symbol encoding.

pruning algorithm were five bits as the minimum gain to retain a node, and a minimum of eight leaves for each subtrees rooted at an internal node.

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 [25]. In particular, 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 number of internal nodes of the context tree which models the source converges towards a constant, reflecting the states in the source (see §6.2.2).

6.5 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.

Chapter Seven

Semi-Static Dictionary Model

Dictionary-based modelling is a mechanism used in many practical compression schemes. For example, the various members of the two Ziv-Lempel families (see also §5.1) parse the input message into a sequence of phrases selected from a dictionary, and obtain their compression since a reference to the phrase can be more compact than the phrase itself. Despite the inherent disadvantage in prediction capability compared to symbol-based methods – the conditioning context used to guide probability predictions is, in essence, reset to the empty string at the start of each phrase – the paradigm is attractive because of the elegant balance it achieves between speed, memory usage, simplicity, and compression ratio.

In most implementations of dictionary-based compression the encoder operates *online*, incrementally inferring its dictionary of available phrases from previous parts of the message, and adjusting its dictionary after the transmission of each phrase. Doing so allows the dictionary to be transmitted implicitly, since the decoder simultaneously makes similar adjustments to its dictionary after receiving each phrase.

An alternative approach – the topic explored in this chapter – is to use the full message (or a large block of it) to infer a complete dictionary in advance, and include an explicit representation of the dictionary as part of the compressed message. Intuitively, the advantage of this *offline* approach to dictionary-based compression is that with the benefit of having access to all of the message, it should be possible to optimize the choice of phrases so as to maximize compression performance. Indeed, we demonstrate that, particularly on large files,

very good compression can be attained by an offline method without compromising the fast decoding that is a distinguishing characteristic of dictionary-based techniques.

Several nontrivial sources of overhead, in terms of both computation resources required to perform the compression, and bits generated into the compressed message, have to be carefully managed as part of the offline process. In this investigation we develop a compression scheme *Re-pair* which is a combination of a simple but powerful phrase derivation method and a compact dictionary encoding. The scheme is highly efficient, particularly in decompression, and has characteristics that make it a favourable choice when compressed data is to be searched directly.

It should also be noted that while offline compression involves the disadvantage of having to store a large part of the message in memory for processing, the difference between doing this and storing the growing dictionary of an online compressor is illusory. Indeed, incremental dictionary-based algorithms maintain an equally large part of the message in memory as part of the dictionary; similarly, online predictive symbol-based context models occupy space that may be linear in the size of that part of the message on which prediction is based.

Our scheme is offline only while inferring the dictionary, and during decompression bits are read and phrases written in a fully interleaved manner. Moreover, during decoding only a relatively compact representation of the dictionary must be stored. Thus, during decompression, our approach has a space advantage over both incremental dictionary-based schemes and over context-based source models.

Notation In this chapter we use the *symbol* concept in a more general sense than in the rest of the thesis. We allow our algorithm to introduce new symbols; thus, symbols are not restricted to be only input items. To distinguish input symbols from created ones, we denote them *characters*. Thus, there are k possible distinct characters – the symbols of the input alphabet – but a larger variable number, k', of distinct symbols currently used internally in the algorithm.

Dictionary-Based Compression The goal of dictionary-based modelling to derive a set of *phrases* (normally, but not always, substrings of the message being encoded) in such a way that replacing the occurrences of these phrases in the message by references to the table of phrases decreases the length of the message. Furthermore, since in an offline method the phrase table must be transmitted as part of the compressed message, the derivation scheme used should allow a compact

encoding of the phrase set. This latter requirement does not apply to incremental dictionary-based methods, and they may create their dictionary without concern for how it might be represented.

7.1 Previous Approaches

Extensive treatment of offline substitution methods in the so-called *macro model* is given by Storer [63, chapter 5]. In addition to presenting several practical schemes, this survey also proves the intractability of *optimal* offline substitution.

An early exploration of phrase derivation is by Rubin [58]. He suggests several strategies, and gives experimental results. The basic idea for our scheme, as well as for some other similar approaches to dictionary derivation [17, 55], is clearly related to the *incremental encoding* schemes suggested by Rubin. However, his treatment of computational complexity and dictionary encoding techniques is superficial.

To facilitate a compact encoding of the phrase table we employ a *hierarchical* scheme where longer phrases are encoded through references to shorter ones. This is in some ways similar to the LZ-78 mechanism [72], and the extension to that developed by Miller and Wegman [50]. The drawback of the aggressive phrase construction policies of LZ-78 mechanisms is that the dictionary is diluted by phrases that do not in fact get productively used, and compression suffers. In our proposal, described in detail in §7.2, every phrase is used either to directly code at least two distinct parts of the source message, or as a building block of a longer phrase that is itself used twice or more.

Our derivation scheme is also loosely related to the grammar-based compression method *Sequitur* of Nevill-Manning and Witten [56]. In Sequitur, the input message is processed incrementally, and rules in a context-free grammar are created and then revised in a symbol-by-symbol manner, with the decoder inferring the rules from the compressed message stream. But because Sequitur processes the message in a left-to-right manner, and maintains its two invariants (uniqueness and utility) at all times, it does not necessarily choose as grammar rules the phrases that might eventually lead to the most compact representation. Hence, Sequitur is best categorized as an online algorithm with strong links to the LZ-78 family, and the obvious question is whether a holistic approach to constructing a grammar to represent the message can yield better compression.

Our scheme also has some points in common with the compression regime described by Manber [46]. To obtain fast searching of compressed text, Manber considers a simple compression mechanism

based upon character digrams, and then compresses a search string using the same rules, so that the two compressed representations can be directly compared using standard pattern matching algorithms. We also replace frequent pairs, but continue the process recursively until no more pairs of symbols can be reduced. Hence the name of our program, *Re-pair*, for *recursive pairing*.

Apostolico and Lonardi [7] present an offline compression scheme with a phrase derivation scheme that uses a suffix tree. The suffix tree is augmented to maintain statistics for contexts without overlap, which requires superlinear $O(n \log n)$ construction time. However, although this scheme involves offline phrase derivation, the transmission of the dictionary is performed incrementally. Thus, it is not fully offline in the sense of our algorithm, and does not offer the same potential for random access searching in the compressed data.

The same is true for the work of Nakamura and Murashima [55]. They independently propose a compression scheme that comprises the same phrase generation scheme as ours, but has a different approach to the representation of the dictionary as well as and message encoding. Similarly to the other mentioned previous approaches, the dictionary is transmitted adaptively.

Another independent work based on a similar phrase generation scheme is that of Cannane and Williams [17]. Their approach is specialized for processing very large files using limited primary storage. It involves scanning through the input in multiple passes during dictionary construction. Hence, their algorithm requires a multiply longer encoding time than a single-pass encoding algorithm would, but on the other hand does not require that large inputs are split into separate blocks.

7.2 Recursive Pairing

The phrase derivation algorithm used in Re-pair consists of replacing the most frequent pair of symbols in the source message by a new symbol, reevaluating the frequencies of all of the symbol pairs with respect to the extended alphabet, and then repeating the process until there is no pair of adjacent symbols that occurs twice. Algorithm R on the facing page captures this mechanism. Although this simple scheme is not among the more well known compression algorithms, similar techniques have, as noted in §7.1, appeared as components of several independent works [17, 55, 58].

The message is reduced to a new sequence of symbols, each of which represents either a unit symbol or a pair of recursively defined

1 Identify symbols a and b such that ab is the most frequent pair of adjacent symbols in the message. If no pair appears more than once, stop.
2 Introduce a new symbol A and replace all occurrences of ab with A.
3 Repeat from step 1.

Algorithm R, the basic pair replacement mechanism.

symbols. That is, each of these final symbols is a phrase; the phrase set is organized in the form of hierarchical graph structure with unit symbols at the lowest level. A zero-order entropy code for the reduced message is the final step in the compression process; and the penultimate step is, of course, transmission of the dictionary of phrases.

We have not specified in which order pairs should be scheduled for replacement when there are several pairs of equal maximum frequency. While this does influence the outcome of the algorithm, in general it appears to be of minor importance. The current implementation resolves ties by choosing the least recently accessed pair for replacement, which avoids skewness in the hierarchy by discriminating against recently created pairs.

7.3 Implementation

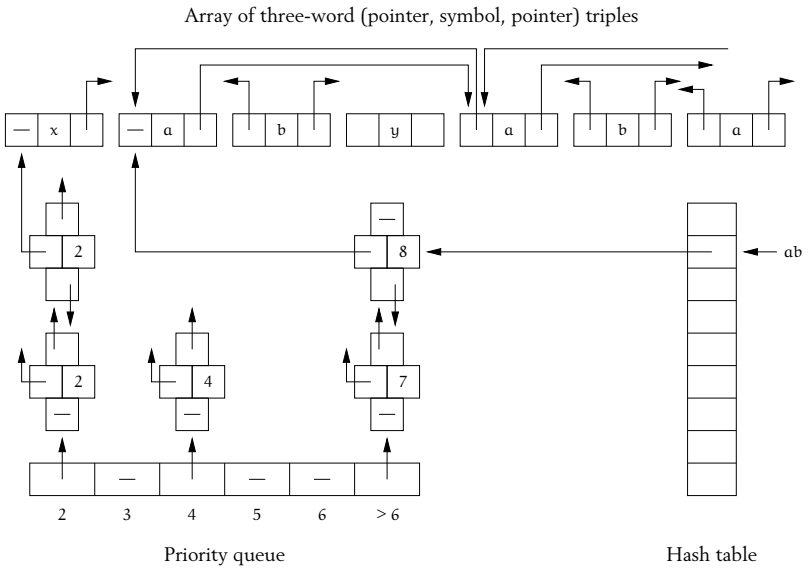
We sketch a phrase derivation implementation that takes $O(n)$ time and space. Many options are available, but for brevity only a single set of choices is described here, and a number of alternatives are omitted.

7.3.1

Data Structures Our implementation involves three data structures to access pairs in the input sequence:

- An array storing the sequence of symbol numbers – initially, the characters of the input message. Each record in the array contains three words: one that holds the symbol number, and two that are used as threading pointers.
- A hash table with an entry for each *active pair*, which denotes a combination of two adjacent symbols in a pair that is still under consideration for replacement by a single symbol; and a pointer to the first appearance of each active pair in the symbol array.
- A specialized priority queue, implemented as an array of roughly $\sqrt{n}$ linked lists recording the active pairs that occur less than that number of times, and one final list recording the more frequent ones.

The figure on the next page shows the full structure of our suggested implementation. The two pointers of each record in the sequence array are used to thread records together in a series of doubly linked lists, one for each active pair. In combination with the hash ta-



Data structures during phrase construction. Pair ab is assumed to be one of two symbol pairs that appear more than six times; with the first appearance of ab being the one illustrated, in context xaby. Pair xa is assumed to appear twice, with the one shown being the first.

ble, this gives us direct access to all positions of the sequence array that hold a given active pair.

As pairs are aggregated, some positions of the array become empty, as one of the two records combined is left vacant. To allow skipping over sequences of adjacent records in constant time, the empty space is also threaded: in the first record in a sequence of empty records, the forward thread points to the first nonempty record beyond this sequence. Analogously, in the last record, the backward thread points to the last nonempty record before the sequence.

The hash table and priority queue make use of the same set of underlying records, each of which holds a counter for the number of occurrences of that active pair, and the pointer to the first location at which that pair occurs.

Note that the count of any existing active pair never increases. When the count of a pair decreases as a result of its left or right part being absorbed in a pair replacement, that pair either remains on the final priority list or is moved to a list residing at a lower index in the array. Moreover, the count of any new active pairs introduced during the replacement process cannot exceed the count of the pair being replaced. Hence, the maximum count is a monotonically decreasing entity, and locating the next most frequent active pair can be done in constant time per pairing operation. When the last list of frequent items is non-empty, it is scanned in $O(\sqrt{n})$ time to find the

greatest frequency pair, and when this is identified at least $\sqrt{n}$ pairs are replaced as a result. Once all the pairs on the final list have been dealt with, the rest of the priority array is walked from its last position (roughly $\sqrt{n}$) down to position one, using $O(n)$ total time.

The priority queue is initialized in linear time by scanning the original sequence and updating counts and entry lists through hash table lookups. The total time consumed by all executions of step 1 of Algorithm R on page 95 is thus $O(n)$.

7.3.2 *Pair Replacement Operation* To account for the replacement operation in step 2 of Algorithm R, observe that since the length of the sequence decreases for each replacement, the total number of replacements is $O(n)$. Replacement of a single appearance of pair ab by a new symbol A involves the following sequence of operations, each of which must be accomplished in constant time:

I Locate the first or next sequence entry associated with ab. Identify the adjacent symbols x and y to establish the context $xaby$.

II Decrement the counts of the adjacent pairs xa and by. If any of the pairs reaches a count of one, delete its priority queue record.

III Replace ab in the sequence, leaving xAy.

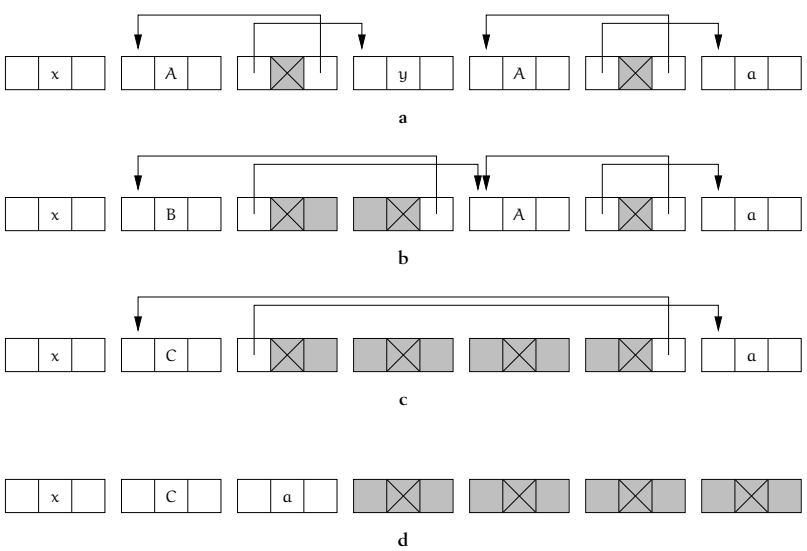
IV Increase the counts of the pairs xA and Ay. This involves creating records for them and adding them to the hash table and priority queue if necessary (see §7.3.4).

Care must be taken for sequences of identical symbols, since these introduce overlapping pairs. For example, replacing aa with A in the subsequence $aaaa$ should yield two occurrences of A, not three. If the initial scanning for pairs as well as replacements is done in strict left-to-right order (which is natural), this is a simple matter of remembering the last few positions encountered in scanning or replacing pairs, and excluding any pair that overlaps one that was just counted.

Operations I and III can be accomplished in $O(1)$ time using the threading pointers of the sequence array.

In steps II and IV, entries are moved from one linked list in the priority queue to another. These movements take $O(1)$ time, because each pair record includes the index of the list that contains it, and the lists are doubly linked. Hence, total processing time is $O(1)$ per symbol.

7.3.3 *Memory Space* Initially, each symbol in the input message is stored as a three-word triple in an array of $3n$ words. One word is used to store a symbol number, and the other two are pointers threading together equal pairs of symbols in the sequence.



Spanning the gaps between sequence records.
a: after pair ab is reduced to A (in two places, see the figure on page 96);
b: after Ay is reduced to B;
c: after BA is reduced to C;
d: after a compaction phase. The normal thread pointers are omitted.

The priority queue structure requires an array of $\lceil\sqrt{n+1}\rceil - 1$ words, plus a record for each distinct pair that appears twice or more in the source message. Each record stores the frequency of that pair, plus a pointer to the first appearance in the sequence of that pair. Moreover, the lists are doubly linked, so two further words per record are required for the list pointers.

Prior to any pair replacements there can be at most k^2 distinct pairs in the priority lists. Thereafter, each pair replacement causes at most one new item to be added to the priority lists – both left and right combinations (xA and Ay in the figure on page 96) might be new, but each must occur twice before they need to be added to the priority lists, and so in an amortized sense, it is at most one new active pair per pair replaced. Each of these new records requires a further four words of space.

With careful attention to detail, it is possible to limit the amount of extra memory required by priority list nodes to just n words.

Suppose that the pair reduction process is commenced with $3n$ words in the sequence array, and $4k^2$ words in use for the k^2 initial priority list items. When $n/4$ pair replacements have taken place, and at most $n/4$ new priority list items (taking a total of n words of memory) have been created, the processing is temporarily suspended, and a *compaction* phase (illustrated above) commenced. The purpose of compaction is to pack all sequence records still being used into a single section of the sequence array, and free the memory occupied by empty

sequence records for other use. Since $n/4$ replacements have taken place, the first compaction phase frees a block of at least $3n/4$ words.

Pair replacement then resumes. The memory freed is sufficient for the construction of another $3n/16$ priority list nodes, which in turn can only happen after a minimum of $3n/16$ further pair replacements have taken place. A second compaction, this time over only 3/4 of the length of the original $3n$-word array, then takes place, and frees up $9n/16$ words, which is enough to allow the replacement process to resume and reduce another $9n/64$ pairs.

This alternating sequence of compactions and reductions continues until all pairs have been resolved, and, by construction, the ith compaction will take place (at the earliest) after $3^{i-1}n/4^i$ pair replacements, and will be required to pack $3^{i-1}n/4^{i-1}$ three-word records into $3^i n/4^i$ three-word spaces, and in doing so frees space for

$$n \cdot \left(\frac{3^{i-1}}{4^i} - \frac{3^i}{4^{i+1}} \right) \cdot \frac{3}{4} = n \cdot \frac{3^i}{4^{i+1}}$$

list records, since each sequence record is 3/4 the size of a list record. That is, the memory freed by one compaction is exactly sufficient to accept all newly created priority lists records generated prior to the next compaction, and apart from the n words added during the first phase, no more memory needs to be allocated.

Moreover, since the time taken by each compaction operation is linear in the number of records scanned, the total cost of all compactions is:

$$O\left(n \cdot \sum_{i=0}^{\infty} (3/4)^i \right) = O(n).$$

Pair Record Considerations Our arrangement supposes that records are not created for new pairs unless it is clear that they will appear in the reduced sequence more than once. For this reason, when replacing ab with A, we scan the list of occurrences for ab twice: 7.3.4

In the first pass, we do not increment any counts. Instead, we check, for each occurrence of ab in the context $xaby$, if there is already a hash table entry for xA and Ay respectively. If not, we need to find out if the current position is the first or second appearance of that new pair along the ab list. We allocate one special bit per hash table entry to record this. At the first appearance of xA we set this bit in the hash table entry for xa, and for Ay analogously in the entry for by. If either of these hash table entries does not exist, we know immedi-

ately that the corresponding new pair cannot occur twice, and skip it. (For example, if xa is not in the hash table, this means that it occurs only once in the sequence, and therefore xA will occur only once as well). If we find the bit already set, we know that this is the second appearance of that pair and allocate a new priority queue entry for xA or Ay, which we link into the hash table and priority queue. The first-occurrence bit can then be reset.

In the second pass, we increment counts in the priority queue entries for pairs that have entries in the hash table. The processing in the first pass guarantees that these are the active pairs.

The hash table structure contains a pointer to the priority list record for each pair, from whence the pair itself can be identified by following the pointer in that record into the symbol sequence array. The number of entries in the hash table never exceeds $n/2$, since each entry corresponds to a pair of adjacent symbols in the message that appears at least twice, and there are at most n symbols in the message. If it is supposed that a peak loading of $1/2$ is appropriate, the hash table must have space for n pointers.

To allow deletion to be handled, linear probing is used to resolve collisions [39, page 526]. When a record is deleted, rather than simply tag it as such in the hash table, all of the records between its location and the next empty cell are reinserted. The cost of this miniature rehashing is asymptotically less than the square of the cost of an unsuccessful search, which is $O(1)$ expected time for a given table loading. Hence, all of lookup, insertion, and deletion require $O(1)$ expected time.

A fourth data structure not already described is the hierarchical phrase graph. Each record in this directed acyclic graph requires two words of memory, indicating the left and right components of this particular phrase, and is required at exactly the same time as the priority list item for that particular pair is being processed. Hence, that space can be reused, and no additional space is required.

7.3.5 *Total Dictionary Space* Summed over all data structures, the memory required is never more than $5n + 4k^2 + 4k' + \lceil\sqrt{n+1}\rceil - 1$ words, where n is the number of symbols in the source message, k is the cardinality of the source alphabet; and k' is the cardinality of the final dictionary. This requirement is dominated by the $5n$ component except in pathological situations in which k' might be large.

Hence, we can summarize our findings regarding the complexity of the dictionary construction algorithm in the following statement:

Theorem 7A Dictionary creation through recursive pair replacement with an input of n symbols from an alphabet of size k, generating a total of k' phrases, is accomplished in expected $O(n)$ time, using $5n + 4k^2 + 4k' + \lceil\sqrt{n+1}\rceil - 1$ words of primary storage.

In the decoder, two words of memory are required for each phrase in the hierarchy. As is demonstrated by the experiments in §7.7, this is a very modest requirement.

7.4 Compression Effectiveness

We now consider the manner in which the phrase derivation scheme described in §7.2 achieves compression, when phrases in the input are represented as references to the table, using a zero-order entropy code. Transmission of the dictionary of phrases is disregarded in this section.

Symbolwise Equivalent 7.4.1 To understand the structure of a dictionary-based model, it is helpful to consider the structure of its *symbolwise equivalent* model – a model that process one character at a time with an entropy coder [12, 41].

Consider the final sequence of phrases. Suppose that there are k' distinct phrases in the sequence, and n' phrases in total. Then each occurrence of a phrase that appears l times in the final sequence generates approximately $-\log(l/n')$ bits in the compressed message, since the final phrases are entropy coded. Let one such phrase, of length r, be described by $x_1x_2\ldots x_r\$$, where \$ represents an *end of phrase* symbol, and let l be its frequency. Let $N(c \mid s)$ be the number of phrases in the final sequence (of the n') which have sc as a prefix. For example, $N(a \mid \epsilon)$ is the count of the number of phrases that commence with character 'a' (ϵ is the empty string) and $N(b \mid a)$ is the number of phrases that commence with 'ab'.

Now consider the expression

$$-\log\frac{N(x_1 \mid \epsilon)}{n'} - \log\frac{N(x_2 \mid x_1)}{N(x_1 \mid \epsilon)} - \cdots - \log\frac{N(\$ \mid x_1x_2\ldots x_r)}{N(x_r \mid x_1x_2\ldots x_{r-1})}\ ,$$

which telescopes to

$$-\log\frac{N(\$ \mid x_1x_2\ldots x_r)}{n'} = -\log\frac{l}{n'}\ ,$$

since $N(\$ \mid x_1x_2\ldots x_r) = l$, the number of times the phrase $x_1x_2\ldots x_r$ appears. That is, the overall code for each phrase can be interpreted as a zero-order code for the first symbol, with probabilities evaluated

relative to the commencing letters of the set of phrases; followed by a first-order probability for the second symbol, with probabilities evaluated in the context of first letters of the set of phrases, and so on.

7.4.2 *Sources of Redundancy* Given the symbolwise equivalent, it is unlikely that the Re-pair mechanism can outperform a well-tuned context-based model, since the latter uses a high-order prediction for every character in the message, whereas, like other dictionary-based methods, Re-pair essentially resets its prior context to the empty string at the start of each phrase. However, the same improvements that have been suggested for other dictionary-based models can be used if compression effectiveness is to be maximized. For example, Gutmann and Bell [29] suggest that the probability for each phrase be conditioned upon the last character of the previous phrase. (A full first-order model on phrases is, of course, pointless.)

Another way in which compression effectiveness can be improved is to note that, by virtue of the way in which phrases are constructed, the final sequence contains no repeated symbol pairs, nor any pairs that constitute phrases in the dictionary. That is, if phrase pair AB has previously appeared in the final message, or if $C = AB$ is in the phrase table (for some C), then, when phrase A appears, the next phrase cannot be B. In this case phrase B, and any others that match the criteria, can be *excluded* from consideration at the next coding step, and the remaining probabilities adjusted upwards, in the same way that in PPM-style methods characters can be excluded because they are known to not be possible (see §5.5.2).

Both conditioning and exclusions are complex to implement, and if compression effectiveness (rather than compression efficiency) is the goal, then a full context-based mechanism is a better basic choice of algorithm. Our current implementation includes neither of these two improvements.

7.5 Encoding the Dictionary

The hierarchical organization of the phrase table offers a natural way to encode it compactly as backward-referring pairs. This is achieved by encoding the phrases in *generations*. The first generation is the set of phrases that consist of two primitive symbols, the second generation the phrases constructed by combining first-generation objects, etc.

Let the primitive symbols be generation 0, and the number of items up to and including generation i be k_i. Define k_i for $i < 0$ to be zero, $k = k_0$ to be the size of the input alphabet, and s_i to be the size of

generation i. That is, $k_i = \sum_{j \le i} s_j$. Finally, note that each phrase in generation i can be assumed to have at least one of its components in generation $i-1$, as otherwise it could have been placed in a earlier generation. Therefore, the universe from which the phrases in generation i are drawn has size $k_{i-1}^2 - k_{i-2}^2$. Items are numbered from 0, so that the primitives have numbers 0 through $k-1$, and generation i has numbers k_{i-1} through $k_i - 1$. Each item is a pair (l, r) of integers, where l and r are the ordinal symbol numbers of the left and right components.

Given this enumeration, the task of transmitting the dictionary becomes the problem of identifying and transmitting the generations; and to transmit the ith generation, a subset of size $s_i = k_i - k_{i-1}$, drawn from the range $k_{i-1}^2 - k_{i-2}^2$, must be represented. This section considers three strategies for the low-level encoding of the generations: arithmetic coding with a Bernoulli model, spelling out the pairs literally, and binary interpolative encoding.

7.5.1 *Bernoulli Model* If the s_i combinations that comprise the ith generation are randomly scattered over the $k_{i-1}^2 - k_{i-2}^2$ possible locations, then an arithmetic coder and a Bernoulli model will code the ith generation in

$$\log \binom{k_{i-1}^2 - k_{i-2}^2}{k_i - k_{i-1}} \text{ bits.}$$

For efficiency reasons we do not advocate the use of arithmetic coding for this application; nevertheless, calculating the cost of doing so over all generations gives a good estimate of the underlying entropy of the dictionary, and is reported as a reference point in the experimental results in §7.7. (The cost of transmitting the input alphabet is disregarded in this estimate).

7.5.2 *Literal Pair Enumeration* On the other hand, the most straightforward way of encoding the generations is as pairs of numbers denoting the ordinal numbers of the corresponding left and right elements, encoding $(l_1, r_1), (l_2, r_2), \ldots$ as the number sequence $l_1, r_1, l_2, r_2, \ldots$. A few optimizations to limit the range of these integers, and thereby reduce the required number of bits to encode them, are immediately obvious:

- *Numbers are contained in previous generations.* The maximum element when encoding generation i is k_{i-1}.
- *Pairs must have one of their elements in the immediately prior generation.* If, in generation i, the left element of a pair is less than or equal to k_{i-2}, then the right element is greater than k_{i-2}.

- *The pairs in each generation may be coded in lexicographically sorted order.* The left elements will then appear in monotonically increasing order, and when the left element is the same as the previous one, the right element is strictly larger than in the previous pair.

  Given these observations, the left elements of the pairs in the sequence grow slowly, with long sequences of equal elements, while the right elements are more varied. Experimentally, the most efficient encoding – of those tested – is to use a zero-origin gamma code (see, for instance, Witten, Moffat, and Bell [70, §3.3] for details of this representation) for transmitting the input alphabet as well the differences between successive left elements in the pair sequence; and a binary code for the corresponding right elements, tracking the remaining range (for the current left element), so that a minimal number of bits can be used.

7.5.3 *Interpolative Encoding* There are many other ways of representing a subset of values over a constrained range [70, Chapter 3]. When the subset is expected to be non-random over the range – as is the case here because, intuitively at least, some symbols are more likely to form pairs than others – the interpolative coding method of Moffat and Stuiver [53] can be used. In this method a sorted list of integer values in a known range is represented by first coding the middle item as a binary number, and then recursively transmitting the left and right sublists, both within the narrowed range established by the now-available knowledge of the value of the middle item. When the middle item lies towards one of the ends of the range, all subsequent codes in that section of the list will thus be shorter than if a normal gap-based mechanism such as Golomb coding had been used. In extreme cases of clustering, values can be transmitted in less than one bit each.

To actually encode the phrases with interpolative coding the two-dimensional pairs data must be converted to single numbers. A direct approach is to enumerate the possible pairs using the same lexicographically sorted ordering as the literal pairs. This means that pair (l, r) in generation i is assigned the number

$$\phi(l, r) = \begin{cases} l\,(k_{i-1} - k_{i-2}) + r - k_{i-2} & \text{for } l < k_{i-2}, \\ l k_{i-1} + r - k_{i-2}^2 & \text{for } l \geq k_{i-2}. \end{cases}$$

The resulting enumeration, which we call *horizontal slide*, is shown in the left half of the figure on the facing page.

The function $\phi(l, r)$ is not symmetric in its arguments, and any two-dimensional clusters in the matrix are broken up into several parts.

l								
6	33	34	35	36	37	38	39	
5	26	27	28	29	30	31	32	
4	19	20	21	22	23	24	25	
3	12	13	14	15	16	17	18	
2				8	9	10	11	
1				4	5	6	7	
0				0	1	2	3	
	0	1	2	3	4	5	6	r

l								
6	7	15	23	30	35	38	39	
5	6	14	22	29	34	37	36	
4	5	13	21	28	33	32	31	
3	4	12	20	27	26	25	24	
2				19	18	17	16	
1				11	10	9	8	
0				3	2	1	0	
	0	1	2	3	4	5	6	r

Pair enumeration of the horizontal and chiastic slides respectively, when $k_{i-1} = 7$ and $k_{i-2} = 3$.

Furthermore, the lower left part of the matrix can be expected to have the higher density, and the interpolative coding should be able to exploit this clustering. This leads to the enumeration shown in the right part of the figure above, which we refer to as the *chiastic slide*. With this scheme, (l, r) in generation i gets number

$$\chi(l,r) = \begin{cases} 2l\,(k_{i-1} - k_{i-2}) + k_{i-1} - r - 1 & \text{for } l < k_{i-2}, \\ (2r+1)(k_{i-1} - k_{i-2}) + l - k_{i-2} & \text{for } r < k_{i-2}, \\ l\,(2k_{i-1} - l) + k_{i-1} - r - k_{i-1}^2 - 1 & \text{for } k_{i-2} \le l \le r, \\ r\,(2k_{i-1} - r - 2) + k_{i-1} - l - k_{i-1}^2 - 1 & \text{for } k_{i-2} \le r < l. \end{cases}$$

Calculating $\chi(l, r)$ is a costly operation if performed for each pair, and the closed form for $\chi^{-1}(x)$, for decoding, includes division as well as a square root. Fortunately, since the encoding is performed generation-wise and numbers are strictly increasing, values can be precomputed or accumulated, and incremental processing is fast. In particular, decoding requires only a constant number of multiplications per generation plus a constant number of additions and subtractions per pair.

7.6 *Tradeoffs*

The Re-pair mechanism offers a number of tradeoffs between time and space. This section briefly canvasses some of these.

Encoder The description of Algorithm R in §7.2 stipulated that pair replacement should continue until no pair occurs twice, but this *all the way* threshold can be modified if faster encoding, or a tighter bound on the dictionary space requirement, is required. At the potential expense of compression effectiveness, we can bring pair replacement to 7.6.1

a premature halt, stopping when either the dictionary reaches a predetermined maximum size or when the count of the most frequent pair reaches a certain value, and transmit the sequence as it stands at that time. The decoding algorithm is unaffected by this change, and acts in exactly the same manner as previously.

A second tradeoff is between encoding space and time. If more memory space can be allocated then encoding will be faster, since longer intervals between compaction phases will be possible. In the limit, if $4n$ words can be allocated to the priority list records ($8n$ words in total for all structures), then no compactions at all are required, even on pathological input sequences.

7.6.2 *Decoder* The decoder offers a particularly convenient tradeoff regarding throughput and memory usage. The simplest – and most compact – decoding data structure is to simply reproduce the phrase hierarchy, and then, for each symbol number decoded, undertake an in-order traversal of the hierarchy, and output a character as each leaf is encountered. While compact, this structure leads to relatively slow decoding. The alternative is to expand all of the phrases to form strings, and then output strings directly as symbol numbers are decoded. This form of decoding is more akin to the mechanism used by LZ-77 decoders such as *Gzip*, and extremely fast decompression results. Moreover, the transition between these two extremes is adjustable. For a given amount of memory – a parameter that is set at decode time, and independent of the encoding process – the most frequent (or recent) strings can be held in full, and others at least partially expanded via recursive processing of the phrase tree. One possible implementation of this tradeoff is to retain a sliding window of recently decoded text (which can be combined with output buffering), in the style of LZ-77 compression mechanisms. Phrases that reappear within the scope of the window can then be decoded by simply copying characters out of the window, rather than recursively expanding the phrase.

7.7 Experimental Results

Our prototype implementation of the Re-pair mechanism, with input partitioned into 1 MB blocks, yields the compression results shown on the facing page. The six test files are the three files of the *Large Canterbury Corpus* or LCC (files and compression results available at *http://corpus.canterbury.ac.nz/*); the file *WSJ-20 MB* which is 20 MB extracted from the *Wall Street Journal* (English text, including SGML markup); the file *Random-1* which consists of a random sequence of

file	*size (kB)*	*p.-ent.*	*lit. p.*	*hori.*	*chi.*	*stat.*	*total*
E.coli	4 529	0.19	0.21	0.14	0.12	2.00	2.12
bible.txt	3 952	0.38	0.38	0.36	0.36	1.53	1.89
world192.txt	2 415	0.42	0.42	0.39	0.38	1.40	1.78
average							1.93
WSJ-20MB	20 480	0.43	0.43	0.41	0.39	1.83	2.22
Random-1	128	0.40	0.82	0.44	0.44	8.13	8.57
Random-2	128	5.33	5.93	5.23	5.02	0.00	5.02

Results for the *Large Canterbury Corpus* and three other files, using a 1 MB blocksize. Explained in §7.7.

8-bit bytes; and the file *Random-2* that contains a sequence of 65 536 random 8-bit bytes, followed by an exact repetition of that sequence.

The *file* column in the table shows the original file sizes in kilobytes; *p.-ent.* is the phrase table entropy estimate calculated as described in §7.5.1; the *lit. p.*, *hori.*, and *chi.* columns show the space required for phrase tables encoded as *literal pairs* (§7.5.2), and with interpolative coding (§7.5.3) using the *horizontal slide* and *chiastic slide* respectively, measured in bits per symbol of the original file. The *stat.* column gives the space for the sequence part of each compressed file compressed with a static minimum-redundancy code, and *total* is the sum of the *chi.* and *stat.* columns.

The final column in the table above shows the overall compression attained when the chiastic slide dictionary representation is combined with a semi-static minimum-redundancy coder for the reduced message [54, 66]. As expected, compression is good, but not quite as good as the PPM context-based mechanism. As a reference point, Gzip and a fifth-order PPM obtain average compression of 2.30 and 1.70 bits per character respectively over the LCC; and 2.91 and 1.76 bits per character on the file *WSJ-20MB*. On the two random files PPM achieves 9.35 and 5.13 bits per character.

The illustration on the next page shows the phrases isolated for the first few verses of the Bible, based upon a dictionary built for a block consisting of the first 1 MB of the file. Quite remarkably, in the same 1 MB section, the longest phrase constructed (and used a total of five times) was

> *offered: \nHis offering was one silver charger, the weight whereof was an hundred and thirty shekels, one silver bowl of seventy shekels, after the shekel of the sanctuary; both of them full of fine flour mingled with oil for a meat offering: \nOne golden spoon of ten shekels, full of incense: \nOne young bullock, one ram, one lamb of the first year, for a burnt offering: \nOne kid of the goats for a sin offering: \nAnd for a sacrifice of peace offer-*

Phrase representation for the first few verses of the King James Bible.

```
In the  beginning  God  creat  ed the  heaven  and the earth
. And the  earth  was  without  form  , and  void  ; and
darkness  was  upon the face of the  deep  . And the
Spirit of God  moved  upon the face of the water  s. \nAnd  God
said, Let  there be light  : and there was  light  . \n And God
saw the  light  , that  it was good  : and God  divided the
light from the darkness  . \n And God  called the  light  D  ay
, and the  darkness  he called  N  ight
. And the evening and the morning were the  first
day. \n And God said, Let  there be a  firmament
in the midst of the  water  s, and  let it  divide the  waters
from the water  s. \n And  God made the  firmament  , and
divided the  water  s which were  under the  firmament
from the water  s which were  above the  firmament
: and it was so. \nAnd God called the  firmament  He  aven
. And the evening and the morning were the  second
day. \nAnd God said, Let the waters  under the  heaven
be gathered together  unto one  place, and  let the  dry land
appear  : and it was so. \nAnd God called the  dry land
```

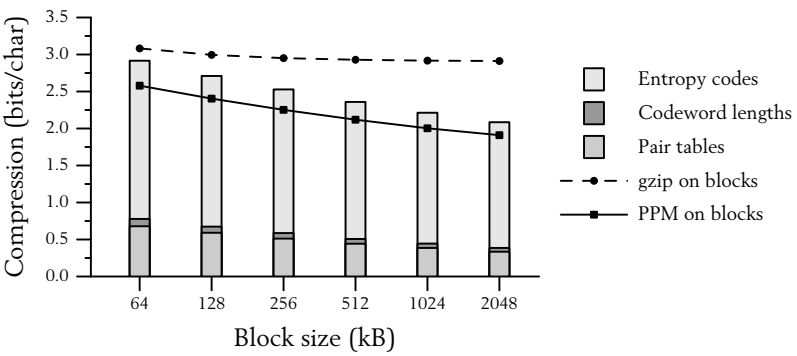
Phrase statistics when using 1 MB blocksize.

file	*max. pairs*	*av. phr.*	*longest*	*av. len.*
E.coli	39 778	16 587	1 800	6.2
bible.txt	28 681	26 994	548	9.3
world192.txt	29 112	24 072	393	10.2
WSJ-20MB	32 069	31 318	1 265	7.8
Random-1	38 878	14 471	4	9.1
Random-2	48 262	53 931	65 534	26 214

ings, two oxen, five rams, five he goats, five lambs of the first year: this was the offering of

in which *\n* indicates a newline character.

The table above gives some detailed statistics for the Re-pair mechanism, again using 1 MB blocks. The *max. pairs* column shows the maximum number of pairs formed during the processing of any of the blocks; *av. phr.* is the average number of phrases constructed per block; *longest* is the length in characters of the longest phrase constructed in any of the blocks; and *av. len.* column is the average number of characters in each symbol of the reduced message. The number of pairs formed is considerably smaller than the length of the block, and the phrases isolated on the non-random files can be very long indeed.



Compression attained as a function of blocksize for file *WSJ-20 MB*.

The diagram above shows the compression rate attained by Re-pair on file *WSJ-20 MB* as a function of blocksize. The three components for Re-pair are the cost of specifying the pairs; the cost of transmitting the codeword lengths; and the cost of coding the reduced message.

Even with a relatively small blocksize, the compression obtained is as good as that of Gzip (which always operates with a blocksize of 64 kB), and for large blocksizes the compression approaches the target set by the fifth-order PPM implementation (here using 32 MB for its model, and escape method D [32]). Because of the memory overheads incurred by the current Java version of the encoder we have been unable to apply Re-pair to the entire 20 MB file, but expect, when we are in a position to do so, that the resultant compression will be excellent, and perhaps comparable to the 1.76 bits per character attained by the PPM implementation.

The prototype implementation of Re-pair was undertaken in Java, compiled and tested with Sun JDK version 1.1.6, to take advantage of the favourable extensibility properties of this object-oriented environment during the development of our program. This experimental program, which was written before the implementation described in § 7.3 was completed (and hence does not conform to that representation), and makes extensive use of dynamic memory, runs slowly compared to other compressors, which are written in C and compiled into efficient machine code. However the Re-pair decoder has been implemented in C as well as Java, and executes approximately 50 times faster, when compiled with the Gnu C compiler (achieving a decoding times far smaller than PPM). Hence, we believe that a C implementation of the encoder will operate at speeds comparable to Gzip and PPM encoding.

Execution time of the three compression processes on the file *WSJ-20 MB* are shown on the next page. These times are CPU seconds for

Experimental time for encoding and decoding.

method	*language*	*encoding*	*decoding*
Gzip	C	40	3
PPM	C	64	70
Re-pair	Java	3 181	254
Re-pair	C	—	5

encoding and decoding the file on a 266 MHz Intel Pentium II with 256 MB RAM and 512 kB cache. For Re-pair, the times listed include the cost of the entropy coder, a program that was supplied by Andrew Turpin of Melbourne University, and based on techniques presented by Turpin and Moffat [66]. This requires 6 seconds for encoding and 1 second for decoding. The PPM implementation uses a fifth-order context; Gzip was used with the −9 option.

7.8 Future Work

A number of areas for further development remain. One track that is worth following is exploiting the fact that our dictionary is static so as to yield efficient access operations when searching in the compressed data.

We are also interested in exploring the possible correlations between blocks of text in a large file. It may be that the dictionary used in one block can most economically be transmitted as a variant of the dictionary used in the previous block, rather than encoded from scratch.

Also, a less resource demanding implementation of the Re-pair program than our prototype compressor clearly needs to be written. Our findings in § 7.3 in combination with an efficient compiler should yield such a program.

Finally, following the lead shown by Sequitur, it will be interesting to assess the extent to which the dictionary construction technique used in Re-pair generates a sensible structural decomposition for complex sequences of a non-textual nature.

Appendix A

Sliding Window Suffix Tree Implementation

This appendix presents an implementation of a suffix tree sliding window index (described in chapter two) as source code in the C programming language [38]. It can also be used as an implementation of Ukkonen's construction algorithm [67] of a regular suffix tree, simply by not calling the function *advancetail*, which advances the left endpoint of the window.

The interface to this code comprises the following functions:

- *initslide* Initialization routine. A pointer to a memory area used as a circular buffer is passed to this function. The user is responsible for filling up this area with input data, and calling *advancefront* and *advancetail* for sliding the window over the buffer. The size of the buffer is also the maximum size of the sliding window.
- *releaseslide* Releases the memory allocated by *initslide*.
- *advancefront* Moves the right endpoint of the window, expanding the tree, by the given number of positions. The corresponding positions of the circular buffer must have been filled with input data before this function is called.
- *advancetail* Moves the left endpoint of the window by the given number of positions, resulting in nodes being removed from the tree.
- *longestmatch* Uses the tree to search for the longest string in the window that matches a given pattern.

The user must ensure that the size of the window stays in the legal range $[0, M]$, where M is the size of the circular buffer.

The implementation conforms to the representation presented in §1.3.2. It follows the algorithms given in chapter two, but includes a few low-level speed optimizations.

For further details, see comments in the source code.

```
#include <stdlib.h>
#include <limits.h>
#include <time.h>

#define RANDOMIZE_NODE_ORDER 1
#define K (UCHAR_MAX+1)

typedef unsigned char SYMB;
enum { SLIDE_OK, SLIDE_PARAMERR, SLIDE_MALLOCERR };

/* Node numbering:

   Node 0 is nil.
   Node 1 is root.
   Nodes 2...mmax-1 are non-root internal nodes.
   Nodes mmax...2*mmax-1 are leaves.*/

struct Node {
   int pos;                     /* edge label start.*/
   int depth;                   /* string depth.*/
   int suf;                     /* suffix link; sign bit is cred.*/
   SYMB child;                  /* number of children minus one, except
                                   for the root which always has
                                   child==1.*/
};

static int mmax;                /* max size of window.*/
static int hashsz;              /* number of hash table slots.*/
static SYMB *x;                 /* the input string buffer.*/
static struct Node *nodes;      /* array of internal nodes.*/
static int *hash;               /* hash table slot heads.*/
static int *next;               /* next in hash table or free list.*/
static int freelist;            /* list of unused nodes.*/
static SYMB *fsym;              /* first symbols of leaf edges*/

static int ins, proj;           /* active point.*/
static int front, tail;         /* limits of window.*/
static int r, a;                /* preserved values for canonize.*/

/* Sign bit is used to flag cred bit and end of hash table slot.*/
#define SIGN            INT_MIN

/* Macros used to keep indices inside the circular buffer (avoiding
   modulo operations for speed). M0 is for subtractions to stay
   nonnegative, MM for additions to stay below mmax.*/
#define M0(i)           ((i)<0 ? (i)+mmax : (i))
#define MM(i)           ((i)<mmax ? (i) : (i)-mmax)

/* Hash function. If this is changed, the calculation of hashsz in
   initslide must be changed accordingly.*/
#define HASH(u, c)      ((u)^(c))
#define UNHASH(h, c)    ((h)^(c))
```

```
/* Macro to get child from hashtable, v=child(u, c). This macro does not
   support the implicit outgoing edges of nil, they must be handled
   specially.*/
#define GETCHILD(v, u, c) {                                        \
   v=hash[HASH(u, c)];                                             \
   while (v>0) {                                                   \
      if ((v<mmax ? x[nodes[v].pos] : fsym[v-mmax])==(c))          \
         break;                                                    \
      v=next[v];                                                   \
   }                                                               \
}

/* Macro to get parent from hashtable. c is the first symbol of the
   incoming edge label of v, u=parent(v).*/
#define GETPARENT(u, v, c) {                    \
   int gp_w=(v);                                \
   while ((gp_w=next[gp_w])>=0)                 \
      ;                                         \
   u=UNHASH(gp_w&~SIGN, c);                     \
}

/* Macro to insert edge (u, v) into hash table so that child(u, c)==v.*/
#define CREATEEDGE(u, v, c) {                   \
   int ce_h=HASH(u, c);                         \
   next[v]=hash[ce_h];                          \
   hash[ce_h]=(v);                              \
}

/* Macro to remove the edge (u, v). c is the first symbol of the edge
   label. Makes use of the fact that the hash and next arrays are located
   next to each other in memory.*/
#define DELETEEDGE(u, v, c) {                   \
   int de_w, de_i, de_h=HASH(u, c);             \
   de_w=hash[de_i=de_h];                        \
   while (de_w!=(v)) {                          \
      de_i=de_w+hashsz;                         \
      de_w=next[de_w];                          \
   }                                            \
   hash[de_i]=next[v];                          \
}

/* Function initslide:

   Initialize empty suffix tree. The buffer parameter should point to an
   array of size max_window_size which is used as a circular buffer. */
int initslide(int max_window_size, SYMB *buffer)
{
   int i, j, nodediff, nodemask;

   mmax=max_window_size;
   if (mmax<2)
      return SLIDE_PARAMERR;
   x=buffer;                    /* the global buffer pointer.*/

   /* calculate the right value for hashsz, must be harmonized with the
      definition of the hash function.*/
   if (mmax>K) {                /* i=max{mmax, K}-1; j=min{mmax, K}-1.*/
      i=mmax-1;
      j=K-1;
```

```
   } else {
      i=K-1;
      j=mmax-1;
   }
   while (j) {                  /* OR in all possible one bits from j.*/
      i|=j;
      j>>=1;
   }
   hashsz=i+1;                  /* i is now maximum hash value.*/

   /* allocate memory.*/
   nodes=malloc((mmax+1)*sizeof *nodes);
   fsym=malloc(mmax*sizeof *fsym);
   hash=malloc((hashsz+2*mmax)*sizeof *hash);
   if (! nodes || ! fsym || ! hash)
      return SLIDE_MALLOCERR;
   next=hash+hashsz;            /* convenient for DELETEEDGE.*/

#if RANDOMIZE_NODE_ORDER
   /* Put nodes into free list in random order, to avoid degenaration of
      hash table. This method does NOT yield a uniform distribution over
      the permutations, but it's fast, and random enough for our
      purposes.*/
   srand(time(0));
   nodediff=(rand()%mmax)|1;
   for (i=mmax>>1, nodemask=mmax-1; i; i>>=1)
      nodemask|=i;              /* nodemask is 2^ceil(log_2(mmax))-1.*/
   j=0;
   do {
      i=j;
      while ((j=(j+nodediff)&nodemask)>=mmax || j==1)
         ;
      next[i]=j;
   } while (j);
   freelist=next[0];
#else
   /* Put nodes in free list in order according to numbers. The risk of
      the hash table is larger than if the order is randomized, but this
      is actually often faster, due to caching effects.*/
   freelist=i=2;
   while (i++<mmax)
      next[i-1]=i;
#endif

   for (i=0; i<hashsz; ++i)
      hash[i]=i|SIGN;           /* list terminators used by GETPARENT.*/

   nodes[0].depth=-1;
   nodes[1].depth=0;
   nodes[1].suf=0;
   nodes[1].child=1;            /* stays 1 forever.*/

   ins=1;                       /* initialize active point.*/
   proj=0;
   tail=front=0;                /* initialize window limits.*/
   r=0;

   return 0;
}
```

```
/* Function releaseslide:*/
void releaseslide()
{
   free(nodes);
   free(fsym);
   free(hash);
}

/* Macro for canonize subroutine:

   r is return value. To avoid unnecessary access to the hash table, r is
   preserved between calls. If r is not 0 it is assumed that
   r==child(ins, a), and (ins, r) is the edge of the insertion point.*/
#define CANONIZE(r, a, ins, proj) {             \
   int ca_d;                                    \
   if (proj && ins==0) {                        \
      ins=1; --proj; r=0;                       \
   }                                            \
   while (proj) {                               \
      if (r==0) {                               \
         a=x[MO(front-proj)];                   \
         GETCHILD(r, ins, a);                   \
      }                                         \
      if (r>=mmax)                              \
         break;                                 \
      ca_d=nodes[r].depth-nodes[ins].depth;     \
      if (proj<ca_d)                            \
         break;                                 \
      proj-=ca_d; ins=r; r=0;                   \
   }                                            \
}

/* Macro for Update subroutine:

   Send credits up the tree, updating pos values, until a nonzero credit
   is found. Sign bit of suf links is used as credit bit.*/
#define UPDATE(v, i) {                          \
   int ud_u, ud_v=v, ud_f, ud_d;                \
   int ud_i=i, ud_j, ud_ii=MO(i-tail), ud_jj;   \
   SYMB ud_c;                                   \
   while (ud_v!=1) {                            \
      ud_c=x[nodes[ud_v].pos];                  \
      GETPARENT(ud_u, ud_v, ud_c);              \
      ud_d=nodes[ud_u].depth;                   \
      ud_j=MO(nodes[ud_v].pos-ud_d);            \
      ud_jj=MO(ud_j-tail);                      \
      if (ud_ii>ud_jj)                          \
         nodes[ud_v].pos=MM(ud_i+ud_d);         \
      else {                                    \
         ud_i=ud_j; ud_ii=ud_jj;                \
      }                                         \
      if ((ud_f=nodes[ud_v].suf)>=0) {          \
         nodes[ud_v].suf=ud_f|SIGN;             \
         break;                                 \
      }                                         \
      nodes[ud_v].suf=ud_f&~SIGN;               \
      ud_v=ud_u;                                \
   }                                            \
}
```

```
/* Function advancefront:

   Moves front, the right endpoint of the window, forward by positions
   positions, increasing its size.*/
void advancefront(int positions)
{
   int s, u, v;                 /* nodes.*/
   int j;
   SYMB b, c;

   while (positions--) {
      v=0;
      c=x[front];
      while (1) {
         CANONIZE(r, a, ins, proj);
         if (r<1) {             /* if active point at node.*/
            if (ins==0) {       /* if ins is nil.*/
               r=1;             /* r is child of ins for any c.*/
               break;           /* endpoint found.*/
            }
            GETCHILD(r, ins, c);
            if (r>0) {          /* if ins has a child for c.*/
               a=c;             /* a is first symbol in (ins, r) label.*/
               break;           /* endpoint found.*/
            } else
               u=ins;           /* will add child below u.*/
         } else {               /* active point on edge.*/
            j=(r>=mmax ? MM(r-mmax+nodes[ins].depth) : nodes[r].pos);
            b=x[MM(j+proj)];    /* next symbol in (ins, r) label.*/
            if (c==b)           /* if same as front symbol.*/
               break;           /* endpoint found.*/
            else {              /* edge must be split.*/
               u=freelist;      /* u is new node.*/
               freelist=next[u];
               nodes[u].depth=nodes[ins].depth+proj;
               nodes[u].pos=M0(front-proj);
               nodes[u].child=0;
               nodes[u].suf=SIGN; /* emulate update (skipped below).*/
               DELETEEDGE(ins, r, a);
               CREATEEDGE(ins, u, a);
               CREATEEDGE(u, r, b);
               if (r<mmax)
                  nodes[r].pos=MM(j+proj);
               else
                  fsym[r-mmax]=b;
            }
         }
         s=mmax+M0(front-nodes[u].depth);
         CREATEEDGE(u, s, c);   /* add new leaf s.*/
         fsym[s-mmax]=c;
         if (u!=1)              /* don't count children of root.*/
            ++nodes[u].child;
         if (u==ins)            /* skip update if new node.*/
            UPDATE(u, M0(front-nodes[u].depth));
         nodes[v].suf=u|(nodes[v].suf&SIGN);
         v=u;
         ins=nodes[ins].suf&~SIGN;
         r=0;                   /* force getting new r in canonize.*/
      }
```

```
      nodes[v].suf=ins|(nodes[v].suf&SIGN);
      ++proj;                   /* move active point down.*/
      front=MM(front+1);
   }
}


/* Function advancetail:

   Moves tail, the left endpoint of the window, forward by positions
   positions, decreasing its size.*/
void advancetail(int positions)
{
   int s, u, v, w;              /* nodes.*/
   SYMB b, c;
   int i, d;

   while(positions--) {
      CANONIZE(r, a, ins, proj);
      v=tail+mmax;              /* the leaf to delete.*/
      b=fsym[tail];
      GETPARENT(u, v, b);
      DELETEEDGE(u, v, b);
      if (v==r) {               /* replace instead of delete?*/
         i=MO(front-(nodes[ins].depth+proj));
         CREATEEDGE(ins, mmax+i, b);
         fsym[i]=b;
         UPDATE(ins, i);
         ins=nodes[ins].suf&~SIGN;
         r=0;                   /* force getting new r in canonize.*/
      } else if (u!=1 && --nodes[u].child==0) {
         /* u has only one child left, delete it.*/
         c=x[nodes[u].pos];
         GETPARENT(w, u, c);
         d=nodes[u].depth-nodes[w].depth;
         b=x[MM(nodes[u].pos+d)];
         GETCHILD(s, u, b);     /* the remaining child of u.*/
         if (u==ins) {
            ins=w;
            proj+=d;
            a=c;                /* new first symbol of (ins, r) label*/
         } else if (u==r)
            r=s;                /* new child(ins, a).*/
         if (nodes[u].suf<0)    /* send waiting credit up tree.*/
            UPDATE(w, MO(nodes[u].pos-nodes[w].depth))
         DELETEEDGE(u, s, b);
         DELETEEDGE(w, u, c);
         CREATEEDGE(w, s, c);
         if (s<mmax)
            nodes[s].pos=MO(nodes[s].pos-d);
         else
            fsym[s-mmax]=c;
         next[u]=freelist;      /* mark u unused.*/
         freelist=u;
      }
      tail=MM(tail+1);
   }
}
```

```
/* Function longestmatch:

   Search for the longest string in the tree matching pattern. maxlen is
   the length of the pattern (i.e. the maximum length of the match);
   *matchlen is assigned the length of the match. The returned value is
   the starting position of the match in the indexed buffer, or -1 if the
   match length is zero.

   The parameters wrappos and wrapto support searching for a pattern
   residing in a circular buffer: wrappos should point to the position
   just beyond the end of the buffer, and wrapto to the start of the
   buffer. If the pattern is not in a circular buffer, call with zero
   values for these parameters.*/
int longestmatch(SYMB *pattern, int maxlen, int *matchlen,
                 SYMB *wrappos, SYMB *wrapto)
{
   int u=1,                     /* deepest node on the search path.*/
      ud=0,                     /* depth of u.*/
      l=0,                      /* current length of match.*/
      e=0,                      /* positions left to check on incoming
                                   edge label of u.*/
      start=-1,                 /* start of the match.*/
      p, v, vd;
   SYMB c;

   while (l<maxlen) {
      c=*pattern;
      if (e==0) {               /* if no more symbols in current label.*/
         if (u>=mmax)
            break;              /* can't go beyond leaf, stop.*/
         GETCHILD(v, u, c);     /* v is next node on search path.*/
         if (v<1)
            break;              /* no child for c, stop.*/
         if (v>=mmax) {         /* if v is a leaf.*/
            start=v-mmax;       /* start of string represented by v.*/
            vd=M0(front-start); /* depth of v.*/
            p=MM(v+ud);         /* first position of edge label.*/
         } else {               /* v is an internal node.*/
            vd=nodes[v].depth;  /* depth of v.*/
            p=nodes[v].pos;     /* first position of edge label.*/
            start=M0(p-ud);     /* start of string represented by v.*/
         }
         e=vd-ud-1;             /* symbols left in current label.*/
         u=v;                   /* make the switch for next iteration.*/
         ud=vd;
      } else {                  /* symbols left to check in same label.*/
         p=MM(p+1);             /* next position in current label.*/
         if (x[p]!=c)
            break;              /* doesn't match, stop.*/
         --e;                   /* one less symbol left.*/
      }
      ++l;                      /* match length.*/
      if (++pattern==wrappos)   /* wrap if reached end of buffer.*/
         pattern=wrapto;
   }
   *matchlen=l;
   return start;
}
```

Appendix B

Suffix Sorting Implementation

In this appendix, we present the full C implementation of the suffix sorting algorithm described in chapter four.

The function *suffixsort* should be passed a pointer to an array of integers representing the input string, which is replaced by the suffix array. A second array of integers must be supplied to hold the inverse array. Furthermore, limits for the input alphabet must be supplied, which should be as tight as possible for the algorithm to operate with maximum efficiency. For details, see comments inside the code.

```
#include <limits.h>

static int *I,                  /* group array, becomes suffix array.*/
   *V,                          /* inverse array, ultimately inverse I.*/
   r,                           /* symbols aggregated by transform.*/
   h;                           /* length of already sorted prefixes.*/

#define KEY(p)          (V[*(p)+(h)])
#define SWAP(p, q)      (tmp=*(p), *(p)=*(q), *(q)=tmp)
#define MED3(a, b, c)   (KEY(a)<KEY(b) ?                        \
        (KEY(b)<KEY(c) ? (b) : KEY(a)<KEY(c) ? (c) : (a))       \
        : (KEY(b)>KEY(c) ? (b) : KEY(a)>KEY(c) ? (c) : (a)))

/* Function update_group:

   Subroutine for select_sort_split and sort_split. Sets group numbers
   for a group whose lowest position in I is pl and highest position is
   pm.*/
static void update_group(int *pl, int *pm)
{
   int g;

   g=pm-I;                      /* group number.*/
   V[*pl]=g;                    /* update first position group number.*/
   if (pl==pm)
      *pl=-1;                   /* one element, sorted group.*/
   else
```

```
      do                        /* more than one elt, unsorted group.*/
         V[*++pl]=g;            /* update group numbers.*/
      while (pl<pm);
}


/* Function select_sort:

   Quadratic sorting method to use for small subarrays. To be able to
   update group numbers consistently, a variant of selection sorting is
   used.*/
static void select_sort_split(int *p, int n) {
   int *pa, *pb, *pi, *pn;
   int f, v, tmp;

   pa=p;                        /* start of group being picked out.*/
   pn=p+n-1;                    /* last position of subarray.*/
   while (pa<pn) {
      for (pi=pb=pa+1, f=KEY(pa); pi<=pn; ++pi)
         if ((v=KEY(pi))<f) {
            f=v;                /* f is smallest key found.*/
            SWAP(pi, pa);       /* place smallest element at beginning.*/
            pb=pa+1;            /* position for elements equal to f.*/
         } else if (v==f) {     /* if equal to smallest key.*/
            SWAP(pi, pb);       /* place next to other smallest elts.*/
            ++pb;
         }
      update_group(pa, pb-1);   /* update group values for new group.*/
      pa=pb;                    /* continue sorting rest of subarray.*/
   }
   if (pa==pn) {                /* check if last part is single elt.*/
      V[*pa]=pa-I;
      *pa=-1;                   /* sorted group.*/
   }
}


/* Function choose_pivot:

   Subroutine for sort_split, taken from algorithm by Bentley & McIlroy
   whose main part is below.*/
static int choose_pivot(int *p, int n) {
   int *pl, *pm, *pn;
   int s;

   pm=p+(n>>1);                 /* small arrays, middle element.*/
   if (n>7) {
      pl=p;
      pn=p+n-1;
      if (n>40) {               /* big arrays, pseudomedian of 9.*/
         s=n>>3;
         pl=MED3(pl, pl+s, pl+s+s);
         pm=MED3(pm-s, pm, pm+s);
         pn=MED3(pn-s-s, pn-s, pn);
      }
      pm=MED3(pl, pm, pn);      /* midsize arrays, median of 3.*/
   }
   return KEY(pm);
}
```

```
/* Function sort_split:

   Sorting routine called for each unsorted group. Sorts the array of
   integers (suffix numbers) of length n starting at p. The algorithm is
   a ternary-split quicksort taken from Bentley & McIlroy, "Engineering a
   Sort Function", Software -- Practice and Experience 23(11), 1249-1265
   (November 1993). This function is based on Program 7.*/
static void sort_split(int *p, int n)
{
   int *pa, *pb, *pc, *pd, *pl, *pm, *pn;
   int f, v, s, t, tmp;

   if (n<7) {                   /* special sort for smallest arrays.*/
      select_sort_split(p, n);
      return;
   }

   v=choose_pivot(p, n);
   pa=pb=p;
   pc=pd=p+n-1;
   while (1) {                  /* split-end partition.*/
      while (pb<=pc && (f=KEY(pb))<=v) {
         if (f==v) {
            SWAP(pa, pb);
            ++pa;
         }
         ++pb;
      }
      while (pc>=pb && (f=KEY(pc))>=v) {
         if (f==v) {
            SWAP(pc, pd);
            --pd;
         }
         --pc;
      }
      if (pb>pc)
         break;
      SWAP(pb, pc);
      ++pb;
      --pc;
   }
   pn=p+n;
   if ((s=pa-p)>(t=pb-pa))
      s=t;
   for (pl=p, pm=pb-s; s; --s, ++pl, ++pm)
      SWAP(pl, pm);
   if ((s=pd-pc)>(t=pn-pd-1))
      s=t;
   for (pl=pb, pm=pn-s; s; --s, ++pl, ++pm)
      SWAP(pl, pm);
   s=pb-pa;
   t=pd-pc;
   if (s>0)
      sort_split(p, s);
   update_group(p+s, p+n-t-1);
   if (t>0)
      sort_split(p+n-t, t);
}
```

```
/* Function bucketsort:

   Bucketsort for first iteration.

   Input: x[0...n-1] holds integers in the range 1...k-1, all of which
   appear at least once. x[n] is 0. (This is the corresponding output of
   transform.) k must be at most n+1. p is array of size n+1 whose
   contents are disregarded.

   Output: x is V and p is I after the initial sorting stage of the
   refined suffix sorting algorithm.*/
static void bucketsort(int *x, int *p, int n, int k)
{
   int *pi, i, c, d, g;

   for (pi=p; pi<p+k; ++pi)
      *pi=-1;                    /* mark linked lists empty.*/
   for (i=0; i<=n; ++i) {
      x[i]=p[c=x[i]];            /* insert in linked list.*/
      p[c]=i;
   }
   for (pi=p+k-1, i=n; pi>=p; --pi) {
      d=x[c=*pi];                /* c is position, d is next in list.*/
      x[c]=g=i;                  /* last position equals group number.*/
      if (d>=0) {                /* if more than one element in group.*/
         p[i--]=c;               /* p is permutation for the sorted x.*/
         do {
            d=x[c=d];            /* next in linked list.*/
            x[c]=g;              /* group number in x.*/
            p[i--]=c;            /* permutation in p.*/
         } while (d>=0);
      } else
         p[i--]=-1;              /* one element, sorted group.*/
   }
}


/* Function transform:

   Transforms the alphabet of x by attempting to aggregate several
   symbols into one, while preserving the suffix order of x. The alphabet
   may also be compacted, so that x on output comprises all integers of
   the new alphabet with no skipped numbers.

   Input: x is an array of size n+1 whose first n elements are positive
   integers in the range l...k-1. p is array of size n+1, used for
   temporary storage. q controls aggregation and compaction by defining
   the maximum value for any symbol during transformation: q must be at
   least k-l; if q<=n, compaction is guaranteed; if k-l>n, compaction is
   never done; if q is INT_MAX, the maximum number of symbols are
   aggregated into one.

   Output: Returns an integer j in the range 1...q representing the size
   of the new alphabet. If j<=n+1, the alphabet is compacted. The global
   variable r is set to the number of old symbols grouped into one. Only
   x[n] is 0.*/
static int transform(int *x, int *p, int n, int k, int l, int q)
{
   int b, c, d, e, i, j, m, s;
   int *pi, *pj;
```

```
   for (s=0, i=k-l; i; i>>=1)
      ++s;                      /* s is number of bits in old symbol.*/
   e=INT_MAX>>s;                /* e is for overflow checking.*/
   for (b=d=r=0; r<n && d<=e && (c=d<<s|(k-l))<=q; ++r) {
      b=b<<s|(x[r]-l+1);        /* b is start of x in chunk alphabet.*/
      d=c;                      /* d is max symbol in chunk alphabet.*/
   }
   m=(1<<(r-1)*s)-1;            /* m masks off top old symbol of chunk.*/
   x[n]=l-1;                    /* emulate zero terminator.*/
   if (d<=n) {                  /* compact if bucketing possible.*/
      for (pi=p; pi<=p+d; ++pi)
         *pi=0;                 /* zero transformation table.*/
      for (pi=x+r, c=b; pi<=x+n; ++pi) {
         p[c]=1;                /* mark used chunk symbol.*/
         c=(c&m)<<s|(*pi-l+1);  /* shift in next old symbol in chunk.*/
      }
      for (i=1; i<r; ++i) {     /* handle last r-1 positions.*/
         p[c]=1;                /* mark used chunk symbol.*/
         c=(c&m)<<s;            /* shift in next old symbol in chunk.*/
      }
      for (pi=p, j=1; pi<=p+d; ++pi)
         if (*pi)
            *pi=j++;            /* j is new alphabet size.*/
      for (pi=x, pj=x+r, c=b; pj<=x+n; ++pi, ++pj) {
         *pi=p[c];              /* transform to new alphabet.*/
         c=(c&m)<<s|(*pj-l+1);  /* shift in next old symbol in chunk.*/
      }
      while (pi<x+n) {          /* handle last r-1 positions.*/
         *pi++=p[c];            /* transform to new alphabet.*/
         c=(c&m)<<s;            /* shift right-end zero in chunk.*/
      }
   } else {                     /* can't bucket, don't compact.*/
      for (pi=x, pj=x+r, c=b; pj<=x+n; ++pi, ++pj) {
         *pi=c;                 /* transform to new alphabet.*/
         c=(c&m)<<s|(*pj-l+1);  /* shift in next old symbol in chunk.*/
      }
      while (pi<x+n) {          /* handle last r-1 positions.*/
         *pi++=c;               /* transform to new alphabet.*/
         c=(c&m)<<s;            /* shift right-end zero in chunk.*/
      }
      j=d+1;                    /* new alphabet size.*/
   }
   x[n]=0;                      /* end-of-string symbol is zero.*/
   return j;                    /* return new alphabet size.*/
}


/* Function suffixsort:

   The main suffix sorting routine. Makes suffix array p of x. x becomes
   inverse of p. p and x are both of size n+1. Contents of x[0...n-1] are
   integers in the range l...k-1. Original contents of x[n] is
   disregarded, the n-th symbol being regarded as end-of-string smaller
   than all other symbols.*/
void suffixsort(int *x, int *p, int n, int k, int l)
{
   int *pi, *pk;
   int i, j, s, sl;
```

```
   V=x;                          /* set global values.*/
   I=p;

   if (n>=k-l) {                 /* if bucketing possible,*/
      j=transform(V, I, n, k, l, n);
      bucketsort(V, I, n, j);    /* bucketsort on first r positions.*/
   } else {
      transform(V, I, n, k, l, INT_MAX);
      for (i=0; i<=n; ++i)
         I[i]=i;                 /* initialize I with suffix numbers.*/
      h=0;
      sort_split(I, n+1);        /* quicksort on first r positions.*/
   }
   h=r;                          /* no of syms aggregated by transform.*/

   while (*I>=-n) {
      pi=I;                      /* pi is first position of group.*/
      sl=0;                      /* sl is neg. length of sorted groups.*/
      do {
         if ((s=*pi)<0) {
            pi-=s;               /* skip over sorted group.*/
            sl+=s;               /* add negated length to sl.*/
         } else {
            if (sl) {
               *(pi+sl)=sl;      /* combine sorted groups before pi.*/
               sl=0;
            }
            pk=I+V[s]+1;         /* pk-1 is last pos. of unsorted group.*/
            sort_split(pi, pk-pi);
            pi=pk;               /* next group.*/
         }
      } while (pi<=I+n);
      if (sl)                    /* if the array ends with sorted group.*/
         *(pi+sl)=sl;            /* combine sorted groups at end of I.*/
      h=2*h;                     /* double sorted-depth.*/
   }

   for (i=0; i<=n; ++i)          /* reconstruct from inverse.*/
      I[V[i]]=i;
}
```

Appendix C

Notation

General

n	input length, p. 11
k	input alphabet size, p. 11
X	input string, p. 11
x_i	symbol number i of input, p. 11
ϵ	empty string, p. 11
\$	string terminator, p. 11

Suffix Tree Representation (More detailed descriptions are given in the table on page 16.)

$\mathit{depth}(u)$	string depth of node u, p. 16
$\mathit{pos}(u)$	incoming edge label position of internal node u, p. 16
$\mathit{fsym}(u)$	first symbol of incoming edge label of leaf u, p. 18
$\mathit{leaf}(i)$	leaf representing suffix starting in position i, p. 16
$\mathit{spos}(u)$	starting position of suffix represented by leaf u, p. 16
$\mathit{child}(u, c)$	child of node u for symbol c, p. 17
$\mathit{parent}(u)$	parent of node u, p. 17
$\mathit{suf}(u)$	suffix link target of internal node u, p. 17
$h(u, c)$	hash value of node $\mathit{child}(u, c)$, p. 17
$g(i, c)$	node with hash value i for symbol c, p. 17
$\mathit{hash}(i)$	linked list of nodes with hash value i, p. 18
$\mathit{next}(u)$	successor of node u in hash table list, p. 18

Appendix C

Sliding Window Indexing

ins	lowest node above or at active point, p. 24
proj	positions between *ins* and active point, p. 24
front	first position to the right of indexed string, p. 24
tail	leftmost position of indexed string, p. 24
M	maximum length of indexed string, p. 24

Word-Partitioned Indexing

¢	word delimiter, p. 34
m	number of words, p. 34
m'	number of distinct words, p. 34

Suffix Sorting

S_i	suffix starting at position i, p. 49
I	position array, ultimately suffix array, p. 49
V	group array, ultimately inverse suffix array, p. 51
L	group length array, p. 53
LCP	longest common prefix of adjacent suffixes, p. 50
h-order	order of suffixes when sorted on h symbols, p. 51
group	subarray of equal suffixes when I is in h-order, p. 53
group number	last position of group, p. 53
K	max. number of occurring symbols in X, p. 61
r	number of symbols combined in transform, p. 61

BWT Compression

X'	BWT transformed string, p. 80
x'_i	symbol number i in BWT transformed string, p. 80
X''	move-to-front-transformed string, p. 82

Semi-Static Modelling

k'	dynamic number of symbols used, p. 92
k_i	number of symbols in generations $0 \ldots i$, p. 102

Bibliography

1 J. Åberg, Yu. M. Shtarkov, and B. J. M. Smeets, *Towards understanding and improving escape probabilities in PPM*, Proceedings of the IEEE Data Compression Conference, March 1997, pp. 22–31.

2 Arne Andersson, *Faster deterministic sorting and searching in linear space*, Proceedings of the 37th Annual IEEE Symposium on Foundations of Computer Science, October 1996, pp. 135–141.

3 Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman, *Sorting in linear time?*, Journal of Computer and System Sciences **57** (1998), no 1, 74–93.

4 Arne Andersson, N. Jesper Larsson, and Kurt Swanson, *Suffix trees on words*, Algorithmica **23** (1999), no 3, 246–260.

5 Arne Andersson and Stefan Nilsson, *Efficient implementation of suffix trees*, Software – Practice and Experience **25** (1995), no 2, 129–141.

6 Alberto Apostolico, *The myriad virtues of subword trees*, Combinatorial Algorithms on Words (Alberto Apostolico and Zvi Galil, eds.), NATO ASI Series, vol. F12, Springer-Verlag, 1985, pp. 85–96.

7 Alberto Apostolico and Stefano Lonardi, *Greedy off-line textual substitution*, Proceedings of the IEEE Data Compression Conference, March–April 1998, pp. 119–128.

8 Ziya Arnavut and Spyros S. Magliveras, *Block sorting and compression*, Proceedings of the IEEE Data Compression Conference, March 1997, pp. 181–190.

9 Ricardo Baeza-Yates and Gaston H. Gonnet, *Efficient text searching of regular expressions*, Proceedings of the 16th International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science, vol. 372, Springer-Verlag, 1989, pp. 46–62.

10 Bernhard Balkenhol, Stefan Kurtz, and Yuri M. Shtarkov, *Modifications of the Burrows and Wheeler data compression algorithm*, Proceedings of the IEEE Data Compression Conference, March 1999, pp. 188–197.

11 Timothy Bell and David Kulp, *Longest-match string searching for Ziv-Lempel compression*, Software – Practice and Experience **23** (1993), no 7, 757–771.

12 Timothy C. Bell and Ian H. Witten, *The relationship between greedy parsing and symbolwise text compression*, Journal of the ACM **41** (1994), no 4, 708–724.

13 Jon L. Bentley and M. Douglas McIlroy, *Engineering a sort function*, Software – Practice and Experience **23** (1993), no 11, 1249–1265.
14 Jon L. Bentley and Robert Sedgewick, *Fast algorithms for sorting and searching strings*, Proceedings of the eighth Annual ACM–SIAM Symposium on Discrete Algorithms, January 1997, pp. 360–369.
15 Suzanne Bunton, *On-line stochastic processes in data compression*, Ph.D. thesis, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, USA, December 1996.
16 Michael Burrows and David J. Wheeler, *A block-sorting lossless data compression algorithm*, Research Report. 124, Digital Systems Research Center, Palo Alto, California, USA, May 1994.
17 Adam Cannane and Hugh E. Williams, *General-purpose compression for efficient retrieval*, Tech. Report TR-99-6, Department of Computer Science, RMIT University, Melbourne, Australia, June 1999.
18 John G. Cleary and W.J. Teahan, *Unbounded length contexts for PPM*, Computer Journal **40** (1997), no 2/3, 67–75.
19 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.
20 Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to algorithms*, The MIT Press/McGraw-Hill, 1990.
21 Thomas M. Cover and Joy A. Thomas, *Elements of information theory*, John Wiley & Sons, 1991.
22 Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan, *Dynamic perfect hashing: Upper and lower bounds*, SIAM Journal on Computing **23** (1994), no 4, 738–761.
23 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.
24 Peter Fenwick, *A new data structure for cumulative frequency tables*, Software – Practice and Experience **24** (1994), no 3, 327–336.
25 Peter Fenwick, *Block sorting text compression*, Proceedings of the 19th Australasian Computer Science Conference (Melbourne, Australia), January–February 1996.
26 Edward R. Fiala and Daniel H. Greene, *Data compression with finite windows*, Communications of the ACM **32** (1989), no 4, 490–505.
27 Michael L. Fredman and Dan E. Willard, *Surpassing the information theoretic bound with fusion trees*, Journal of Computer and System Sciences **47** (1993), 424–436.
28 Gaston H. Gonnet and Ricardo A. Baeza-Yates, *Handbook of algorithms and data structures*, Addison-Wesley, 1991.
29 Peter C. Gutmann and Timothy C. Bell, *A hybrid approach to text compression*, Proceedings of the IEEE Data Compression Conference, March 1994, pp. 225–233.
30 Dov Harel and Robert E. Tarjan, *Fast algorithms for finding nearest common ancestors*, SIAM Journal on Computing **13** (1984), no 2, 338–355.
31 C.A.R. Hoare, *Quicksort*, Computer Journal **5** (1962), no 1, 10–15.
32 Paul Glor Howard, *The design and analysis of efficient lossless data compression systems*, Ph.D. thesis, Department of Computer Science, Brown University, Providence, Rhode Island, USA, June 1993, CS-93-28.
33 Douglas W. Jones, *Application of splay trees to data compression*, Communications of the ACM **31** (1988), no 8, 996–1007.
34 Juha Kärkkäinen and Esko Ukkonen, *Sparse suffix trees*, Proceedings of the 2ndAn-

nual International Conference on Computing and Combinatorics, Lecture Notes in Computer Science, vol. 1090, Springer-Verlag, June 1996, pp. 219–230.
35 Richard M. Karp, Raymond E. Miller, and Arnold L. Rosenberg, *Rapid identification of repeated patterns in strings, trees and arrays*, Proceedings of the 5th Annual IEEE Symposium on Foundations of Computer Science, May 1972, pp. 125–136.
36 Toru Kasai, Hiroki Arimura, and Setsou Arikawa, *Virtual suffix trees: Fast computation of subword frequency using suffix arrays*, Proceedings of the 1999 Winter LA Symposium, February 1999, in Japanese.
37 Jyrki Katajainen and Erkki Mäkinen, *Tree compression and optimization with applications*, International Journal of Foundations of Computer Science **1** (1990), no 4, 425–447.
38 Brian W. Kernighan and Dennis M. Ritchie, *The C programming language*, second ed., Prentice Hall, 1988.
39 Donald E. Knuth, *Sorting and searching*, second ed., The Art of Computer Programming, vol. 3, Addison-Wesley, 1998.
40 Stefan Kurtz, *Reducing the space requirement of suffix trees*, Tech. Report 98-03, Computer Science, Faculty of Technology, University of Bielefeld, Germany, 1998.
41 Glen G. Langdon, *A note on the Ziv-Lempel model for compressing individual sequences*, IEEE Transactions on Information Theory **IT-29** (1983), no 2, 284–287.
42 N. Jesper Larsson, *Extended application of suffix trees to data compression*, Proceedings of the IEEE Data Compression Conference, March–April 1996, pp. 190–199.
43 N. Jesper Larsson, *The context trees of block sorting compression*, Proceedings of the IEEE Data Compression Conference, March–April 1998, pp. 189–198.
44 N. Jesper Larsson and Alistair Moffat, *Offline dictionary-based compression*, Proceedings of the IEEE Data Compression Conference, March 1999, pp. 296–305.
45 N. Jesper Larsson and Kunihiko Sadakane, *Faster suffix sorting*, Tech. Report LU-CS-TR:99-214, LUNDFD6/(NFCS-3140)/1–20/(1999), Department of Computer Science, Lund University, Sweden, May 1999.
46 Udi Manber, *A text compression scheme that allows fast searching directly in the compressed file*, ACM Transactions on Information Systems **15** (1997), no 2, 124–136.
47 Udi Manber and Gene Myers, *Suffix arrays: A new method for on-line string searches*, SIAM Journal on Computing **22** (1993), no 5, 935–948.
48 Edward M. McCreight, *A space-economical suffix tree construction algorithm*, Journal of the ACM **23** (1976), no 2, 262–272.
49 Peter M. McIlroy and M. Douglas McIlroy, *ssort.c*, Source Code, 1997, http://cm.bell-labs.com/cm/cs/who/doug/source.html.
50 Victor S. Miller and Mark N. Wegman, *Variations on a theme by Ziv and Lempel*, Combinatorial Algorithms on Words (Alberto Apostolico and Zvi Galil, eds.), NATO ASI Series, vol. F12, Springer-Verlag, 1985, pp. 131–140.
51 Alistair Moffat, *Implementing the PPM data compression scheme*, IEEE Transactions on Communications **COM-38** (1990), no 11, 1917–1921.
52 Alistair Moffat, *An improved data structure for cumulative probability tables*, Software – Practice and Experience **29** (1999), no 7, 647–659.
53 Alistair Moffat and Lang Stuiver, *Exploiting clustering in inverted file compression*, Proceedings of the IEEE Data Compression Conference, April 1996, pp. 82–91.
54 Alistair Moffat and Andrew Turpin, *On the implementation of minimum-redundancy prefix codes*, IEEE Transactions on Communications **45** (1997), no 10, 1200–1207.
55 Hirofumi Nakamura and Sadayuki Murashima, *Data compression by concatenations of symbol pairs*, Proceedings of the IEEE International Symposium on Information Theory and its Applications (Victoria, BC, Canada), September 1996, pp. 496–499.

56 Craig G. Nevill-Manning and Ian H. Witten, *Compression and explanation using hierarchical grammars*, Computer Journal **40** (1997), no 2/3, 103–116.
57 Michael Rodeh, Vaughan R. Pratt, and Shimon Even, *Linear algorithm for data compression via string matching*, Journal of the ACM **28** (1981), no 1, 16–24.
58 Frank Rubin, *Experiments in text compression*, Communications of the ACM **19** (1976), no 11, 617–623.
59 Kunihiko Sadakane, *A fast algorithm for making suffix arrays and for Burrows-Wheeler transformation*, Proceedings of the IEEE Data Compression Conference, March–April 1998, pp. 129–138.
60 Michael Schindler, *Szip*, Program, 1998, http://www.compressconsult.com/.
61 A. Schönhage, M. Paterson, and N. Pippenger, *Finding the median*, Journal of Computer and System Sciences **13** (1976), no 2, 184–199.
62 Julian Seward, *Bzip2*, Program, 1997–1999, http://www.muraroa.demon.co.uk/.
63 James A. Storer, *Data compression: Methods and theory*, Computer Science Press, 1988.
64 Wojciech Szpankowski, *A generalized suffix tree and its (un)expected asymptotic behaviors*, SIAM Journal on Computing **22** (1993), no 6, 1176–1198.
65 W. J. Teahan, *Modelling english text*, Ph. D. thesis, Department of Computer Science, University of Waikato, Hamilton, New Zealand, October 1998.
66 Andrew Turpin and Alistair Moffat, *Housekeeping for prefix coding*, IEEE Transactions on Communications, to appear.
67 Esko Ukkonen, *On-line construction of suffix trees*, Algorithmica **14** (1995), no 3, 249–260.
68 Peter Weiner, *Linear pattern matching algorithms*, Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory, 1973, pp. 1–11.
69 David Wheeler, *An implementation of block coding*, Tech. report, Cambridge University Computer Laboratory, October 1995.
70 Ian H. Witten, Alistair Moffat, and Timothy C. Bell, *Managing gigabytes: Compressing and indexing documents and images*, second ed., Morgan Kaufmann, 1999.
71 Jacob Ziv and Abraham Lempel, *A universal algorithm for sequential data compression*, IEEE Transactions on Information Theory **IT-23** (1977), no 3, 337–343.
72 Jacob Ziv and Abraham Lempel, *Compression of individual sequences via variable-rate coding*, IEEE Transactions on Information Theory **IT-24** (1978), no 5, 530–536.