Suffix trees for very large inputs

by Marina Barsky

1985 – M.Sc in Biology, Moscau State University, Russia

2006 – M. Sc in Computer Science, University of Victoria, Canada

A Dissertation Submitted in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

© Marina Barsky, 2010

University of Victoria

All rights reserved. This dissertation may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

Suffix trees for very large inputs

by

Marina Barsky

1985 – M.Sc in Biology, Moscau State University, Russia

2006 – M. Sc in Computer Science, University of Victoria, Canada

Supervisory Committee

Dr. Alex Thomo, Main Supervisor

(Department of Computer Science)

Dr. Ulrike Stege, Co-Supervisor

(Department of Computer Science)

Dr. Chris Upton, Co-Supervisor

(Department of Biochemistry and Microbiology)

Dr. Valerie King, Departmental Member

(Department of Computer Science)

Dr. John Taylor, Outside Member

(Department of Biology)

Supervisory Committee

Dr. Alex Thomo, Main Supervisor

(Department of Computer Science)

Dr. Ulrike Stege, Co-Supervisor

(Department of Computer Science)

Dr. Chris Upton, Co-Supervisor

(Department of Biochemistry and Microbiology)

Dr. Valerie King, Departmental Member

(Department of Computer Science)

Dr. John Taylor, Outside Member

(Department of Biology)

ABSTRACT

A suffix tree is a fundamental data structure for string searching algorithms. Unfortunately, when it comes to the use of suffix trees in real-life applications, the current methods for constructing suffix trees do not scale for large inputs.

As suffix trees are larger than their input sequences and quickly outgrow the main memory, the first half of this work is focused on designing a practical algorithm that avoids massive random access to the trees being built. This effort resulted in a new algorithm DiGeST which improves significantly over previous work in reducing random access to the suffix tree and performing only two passes over disk data. As a result, this algorithm scales to larger genomic data than managed before.

All the existing practical algorithms perform random access to the input string, thus requiring in essence that the input be small enough to be kept in main memory. The ever increasing amount of genomic data requires however the ability to build suffix trees for much larger strings. In the second half of this work we present another suffix tree construction algorithm, B^2ST that is able to construct suffix trees for input sequences significantly larger than the size of the available main memory. Both the

input string and the suffix tree are kept on disk and the algorithm is designed to avoid multiple random I/Os to both of them.

As a proof of concept, we show that B^2ST allows to build a suffix tree for 12 GB of *real* DNA sequences in 26 hours on a single machine with 2 GB of RAM. This input is *four times* the size of the Human Genome. The construction of suffix trees for inputs of such magnitude was never reported before.

Finally, we show that, after the off-line suffix tree construction is complete, search queries on entire sequenced genomes can be performed very efficiently. This high query performance is achieved due to a special disk layout of the suffix trees produced by our algorithms.

Contents

Supervisory Committee **ii**

Abstract **iv**

Table of Contents **vii**

List of Tables **x**

List of Figures **xi**

Acknowledgements **xiv**

Dedication **xvii**

1 Introduction **1**

1.1 DNA databases 1

1.2 A new type of data 2

1.3 A different type of index 4

2 Problem definition **10**

2.1 Introduction to suffix trees 10

2.2 Suffix tree storage requirements 14

2.3 Possible solutions to the memory problem 18

2.3.1 Index compression 18

2.3.2 Using disk space 20

3 Minimizing random access to the suffix tree 24

3.1 Related work 25

3.1.1 The Ukkonen algorithm and its disk-based version 25

3.1.2 The brute-force approach and the Hunt algorithm 33

3.1.3 Distributed and paged suffix trees 37

3.1.4 Top Down Disk based suffix tree construction (*TDD*) 41

3.1.5 The partition-and-merge strategy of *Trellis* 46

3.1.6 Summary of the existing algorithms 51

3.2 Our new algorithm: *DiGeST* 53

3.2.1 Design principles of a new I/O-efficient construction 53

3.2.2 The details of our algorithm 55

3.2.3 Experimental evaluation 73

3.3 Summary 79

4 Minimizing random access to the input string 80

4.1 Related work 81

4.1.1 Extension of the *TDD* - *ST-MERGE* 82

4.1.2 *Trellis* with string buffer 82

4.1.3 *DiGeST* and prefix buffering 84

4.1.4 *WAVEFRONT* and multiple scans 84

4.1.5 Summary of the existing algorithms 88

4.2 Our new algorithm: B^2ST 91

4.2.1 Re-using the techniques of *DiGeST* 91

4.2.2 A new technique: pairwise sorting 92

4.2.3 The details of our algorithm 93

4.2.4 Experimental Evaluation 100
4.3 Summary 104

5 Performance of disk-based suffix trees **106**
5.1 Layouts of the disk-based index 106
5.1.1 A forest of suffix trees 106
5.1.2 A new partitioning scheme: partitioning by intervals 108
5.1.3 Edge-labels in binary alphabet 110
5.2 Exact pattern matching 112
5.2.1 Problem definition 112
5.2.2 Exact pattern matching using disk-based suffix trees 112
5.2.3 Experimental evaluation 122
5.3 Summary 123

6 Future research **125**
6.1 Better algorithms for the construction of suffix trees 125
6.2 Suffix-tree based algorithms in disk settings 128
6.3 Conclusions 130

Bibliography **131**

List of Tables

Table 3.1 Input data sets for the evaluation of $DiGeST$ 74

Table 4.1 Construction time of different algorithms for an input of 3 GB kept on disk 102

Table 4.2 Input data sets for the evaluation of B^2ST 103

Table 4.3 Construction time for B^2ST for massive inputs 103

Table 5.1 Input data sets for the evaluation of the search performance . . 122

Table 5.2 The performance of the exact pattern matching 123

List of Figures

Figure 1.1 Suffix tree index 5
Figure 1.2 Search in the suffix tree 6
Figure 1.3 Generalized suffix tree for two input strings 7
Figure 1.4 Unique substrings 9

Figure 2.1 Suffix array with LCP 12
Figure 2.2 Suffix trie and suffix tree 13
Figure 2.3 Suffix tree array representation 14
Figure 2.4 Left-child right-sibling representation of the suffix tree 17
Figure 2.5 Compressed Suffix Trees 20
Figure 2.6 Data transfer speed for different memories 21

Figure 3.1 Example of the Ukkonen algorithm. Start 28
Figure 3.2 Example of the Ukkonen algorithm. Cascade A–C 30
Figure 3.3 Example of the Ukkonen algorithm. Cascade D–F 31
Figure 3.4 Pseudocode for the Ukkonen algorithm 32
Figure 3.5 Pseudocode for Hunt's algorithm 34
Figure 3.6 Example of Hunt's algorithm 36
Figure 3.7 Clifford's suffix tree construction: sparse suffix links 38
Figure 3.8 Example of Clifford's algorithm 40
Figure 3.9 Sample output of Clifford's algorithm 42
Figure 3.10 Pseudocode for the *TDD* algorithm 43

Figure 3.11 Example of the *TDD* algorithm 45
Figure 3.12 Pseudocode for *Trellis* 47
Figure 3.13 Example of the *Trellis* algorithm 48
Figure 3.14 Suffix tree construction methods (input string in main memory) 51
Figure 3.15 Graphical representation of *DiGeST* 56
Figure 3.16 Partitioning of an oversized input string 59
Figure 3.17 Pseudocode for the *DiGeST* algorithm 63
Figure 3.18 Suffix tree for the input string converted to the binary alphabet 66
Figure 3.19 Examples of adding new suffixes to the growing suffix tree . . 68
Figure 3.20 Euler tour 69
Figure 3.21 Suffix tree as an array of nodes 71
Figure 3.22 Chart of the construction time for real DNA 76
Figure 3.23 Chart of the construction time for synthetic DNA 77
Figure 3.24 Chart for comparative performance of *Trellis+* and *DiGeST* . 78

Figure 4.1 The main observation of *WAVEFRONT*. Example 85
Figure 4.2 The main observation of *WAVEFRONT*. Explanation 86
Figure 4.3 Example of the *WAVEFRONT* algorithm. Iteration 1 87
Figure 4.4 Example of the *WAVEFRONT* algorithm. Iteration 2 87
Figure 4.5 Suffix tree construction methods (input string on disk). 89
Figure 4.6 Partitioning for pairwise sorting. 93
Figure 4.7 Suffix array with LCP for a pair of partitions 94
Figure 4.8 Example of an output after pairwise partition sorting 94
Figure 4.9 Pseudocode for pairwise suffix sorting step of B^2ST 95
Figure 4.10 Pseudocode for the merge step of B^2ST 97
Figure 4.11 Pseudocode for refilling buffers in the merge step of B^2ST . . 98

Figure 4.12 Pseudocode for suffix comparison using order arrays in the merge step of B^2ST 99
Figure 4.13 Pseudocode for the pointer advancing in suffix arrays and order buffers in the merge step of B^2ST 99

Figure 5.1 Partitioning by intervals 109
Figure 5.2 The suffix tree for the binary representation of the input string 111
Figure 5.3 Disk-friendly pattern search in the suffix tree. Example 114
Figure 5.4 Pseudocode for exact pattern matching using PATRICIA search in disk-based suffix tree 116
Figure 5.5 Pseudocode for search in a partial tree performed in RAM . . 118
Figure 5.6 Pseudocode for a tree traversal during the blind search 119
Figure 5.7 Pseudocode for finding a leaf for verification after the blind search 120
Figure 5.8 Pseudocode for collecting all occurrences of a pattern in a current tree 121

ACKNOWLEDGEMENTS

I owe my deepest gratitude to all the people who helped me advance this research.

First and foremost, I would like to thank my three supervisors – Drs. Alex Thomo, Ulrike Stege and Chris Upton – for their acceptance, instant support and the degree of academic freedom they have afforded me in research and teaching.

To Alex Thomo, my main supervisor, I am grateful for personal encouragement, sound advice, good company, and lots of good ideas. He was always there for me, he was abundantly helpful and offered invaluable assistance, support and guidance. I am indebted to Dr. Thomo for teaching me how to convert ideas into state-of-the art algorithms and programs. He has been a friend and mentor, had confidence in me when I doubted myself, and helped me out on multiple professional and personal occasions. I would have been lost without his friendly personality, love for people and optimism.

To my co-supervisor Dr. Ulrike Stege I am grateful for the joy and enthusiasm she has for her research which served as a great motivation for me. She has provided the excellent role model as a successful woman professor and a natural expert in human relationships. Thank you for providing a stimulating and fun environment in which to learn and grow; and for the mentoring, support, encouragement, and patience. You have given me multiple opportunities for creativity, never giving up on me even in the thoughest times.

To Chris Upton I would like to thank for his contributions of time, challenging problems, and generous funding to make my research productive and enjoyable. Thank you for your involvement in introducing newly developed bioinformatics tools into the practice of your lab.

I would also like to convey thanks to Calisto Zuzarte from IBM Toronto lab and to Laurence Meadows from Mathematics of Information Technology and Complex

Systems (MITACS) for their encouragement and generous financial support.

Deepest gratitude is also due to the members of the supervisory committee – Dr. Valerie King and Dr. John Taylor – who reviewed my work on a very short notice and gave insightful comments which improved the quality of this work. Thank you for your friendship, encouragement, and valuable advice.

My sincere thanks to Tomas Bednar, code and system guru, for his kind assistance with operating systems, coding and finding the solutions for any technical problem. He has contributed immensely to my professional growth. Thank you also to Brian Douglas whose skill and dedication saved my hardware and data on multiple occasions.

I am thankful to Dr. Michael Witney, Dr. Hausi Muller, Dr. Venkatesh Srinivasan, Dr. Sudhakar Ganti and Dr. George Tzanetakis for their valuable advices in preparing this dissertation and defence.

I am also grateful to the secretaries of the Department of Computer Science. Wendy Beggs, Nancy Chan, Carol Harkness, Isabel Campos and Sharon Moulson deserve special mention for easying our life by solving every administrative problem with efficientcy and smile.

I wish to thank my entire family for providing a loving environment for me. I wish to thank my parents, Olga and Genady Barsky. They bore me, raised me, supported me, taught me, and loved me. I wish to express my love and gratitude to my beloved son Andrey – to whom I dedicate this thesis – and to Maria Kogan; for their patience and endless support, through the duration of my studies.

I thank to my Canadian friends Lenna and Grant Undershute for helping me get through the difficult times, and for all the emotional support, comraderie, entertainment, and caring they provided.

Finally, I would like to thank the National Sciences and Engineering Research Council of Canada (NSERC) for awarding me with a Postgraduate Scholarship. All

this generous support made me worry less about my finances and allowed me to concentrate on the research.

DEDICATION

To my beloved son Andrew

Chapter 1

Introduction

1.1 DNA databases

Nowadays, textual databases are among the most rapidly growing collections of data. One of these collections, the collection of sequenced genomes, is a textual database where the genomes of different organisms are represented as strings of characters over the 4-letter alphabet $\{a, c, g, t\}$ of DNA bases. The reason life-encoding sequences are put into a digital form is to enable computer programs to extract useful information from this raw data; something human, unassisted by software, could not do, no matter the amount of training.

Even for computer programs, both the size of the genomic sequences and their total number are large. From 1982 to the present, the size of the publicly available GenBank sequence database is doubling approximately every 18 months [7]. The Human genome project [12], officially completed in 2000, produced, in addition to the draft sequence of Human genome, a massive database of complete reference genomes of different species. As a result, the total size of the GenBank has reached 85 GB. The size of data in the Whole Genome Shotgun (WGS) sequencing project stands

currently at about 109 GB [74].

In addition, an unprecedented growth rate of genomic data is expected in the near future. Starting in 2008, the "1000 Genomes Project" has been launched [32] with the ultimate goal of collecting sequences of additional 1,500 Human genomes, 500 each of European, African, and East Asian origin. This will produce an extensive catalog of human genetic variations. The size of just the raw sequences in this catalog would be about 5 TB.

The search and comparison of sequences in such massive collections requires efficient and highly scalable algorithms. However, it is hard to develop such efficient solutions using raw sequences alone. Due to the massive and relatively static nature of the sequence data, it would be very useful to preprocess it into a comprehensive index where all the required information could be efficiently located.

1.2 A new type of data

The sequenced genomes represent a new type of digital data that differs from numerical or textual data existed before. Though these sequences can be regarded as strings, one cannot blindly adopt techniques used for indexing natural language texts, since there are fundamental differences between these two kinds of data. The main four features that distinguish DNA data from natural language texts are outlined below.

First of all, **the total size of inter-related information** is several orders of magnitude larger in DNA than in typical natural language texts. Even an entire book has a moderate size compared to the inter-related information in one "volume" of a genome - one chromosome. For example, the size of the Bible does not exceed two million characters [75], while the Human chromosome I is encoded in a sequence of 247 million characters [73].

The second major difference is **the size of tokens**. Here, a token is thought as a separate meaning-bearing entity. The size of tokens in natural languages is small, and is equal to the length of separate words or phrases. In contrast, even if we consider genes – the protein-coding units of genome – as DNA "tokens", these units are several orders of magnitude larger than a typical word or phrase. The non-coding regions, which make about 95% of genomic DNA [71], can hardly be broken into meaningful tokens. And these non-coding regions contain important regulatory information about the entire working organism.

While we do not have a clear division of DNA sequences into tokens (yet), we may consider to index each different substring that occurs in genomic databases. However, **the total number of different substrings to index** (2×10^{17} different substrings only in sequences of the Human genome, as measured by our experiments) is significantly larger than the total number of all possible words in one natural language (c.f. about $171,476$ words in the Oxford English dictionary).

The last distinctive feature of DNA sequences is **the number of different strings with the same meaning**. Multiple heavily "misspelled" tokens of DNA encode for the same output. For example, the GATATATA and TATATATT sequences in the $TATA$-box of a plant promoter bear the same meaning in a sense that they initiate the production of the same amount of the same protein (c.f. [35]).

Due to all these differences, the indexes that work well for natural language texts (such as *inverted indexes* [72] or *B-trees* [21]) cannot be efficiently used for indexing DNA sequences.

1.3 A different type of index

We want to create an index that will help locating all the positions of substrings that are equal to the query pattern q (*an exact pattern matching problem*). An index that indexes all different substrings of a given text is called *a full-text index* [47]. Examples of such indexes are *suffix trees* [48], *suffix arrays* [45], and *string B-trees* [18]. Note that currently none of the algorithms for construction of these indexes scales up for the massive inputs.

We have chosen to work towards the more efficient construction of **suffix trees**. A suffix tree not only indexes all distinct substrings of a given text (as a suffix array or string B-tree does), but also "exposes the internal structure of a string in a deeper way [26] (p. 89)", and as such provides efficient solutions to many string problems more complex than exact pattern matching. This includes the approximate pattern matching – the fundamental task in the biological sequence analysis.

An early, implicit form of suffix trees can be found in Morrison's [50] PATRICIA tree[1]. But it was Weiner [70] who initially proposed to use a suffix tree as an explicit index.

Informally, a suffix tree is a digital tree of all suffixes of a given string. Figure 1.1 depicts the suffix tree for string *ababc*. Each suffix is inserted into the tree: the suffix starting at position 0 – *ababc*, at position 1 – *babc*, at position 2 – *abc*, an so on.

Once the suffix tree is built, we can answer multiple combinatorial questions about the input string. For example, with the help of suffix tree we can count the total number of distinct substrings of the input string. This application is based on a fundamental property of a suffix tree: every distinct substring of the input string X is spelled out exactly once along a path from the root of the suffix tree. Thus an inventory of all the distinct substrings of X can be produced by listing all the strings

[1]PATRICIA stands for **P**ractical **A**lgorithm **T**o **R**etrieve **I**nformation **C**oded **I**n **A**lphanumeric

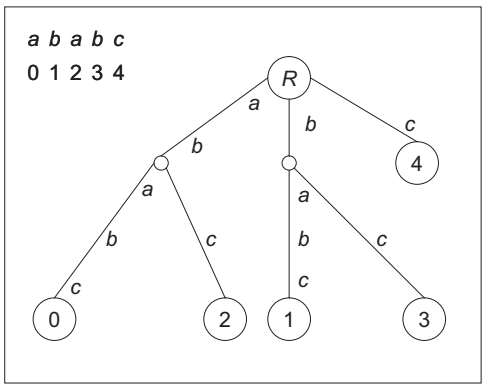

Figure 1.1: The suffix tree for string *ababc*. Each suffix of *ababc* is represented as a path from the root to the leaf node of this tree.

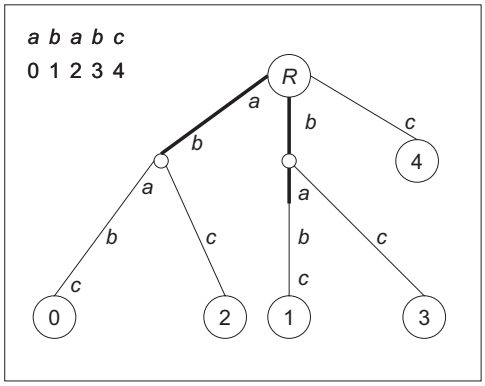

Figure 1.2: Examples of search in the suffix tree. The query strings ab and ba are found on the paths from the root of the tree – bold lines. Once the query string is located, the positions – $(0, 2)$ for ab and (1) for ba – are obtained from the leaves of the corresponding subtrees.

along each such path. The total number of distinct substrings may be quadratic in the input length, since in the worst case there may be as many as $\binom{N}{2}$ distinct substrings in the input of size N (that many ways to choose the pair of start and end positions). We can count the total number of all distinct substrings by simply adding up the lengths of the edges of the suffix tree in time linear in N. For example, there are 12 different substrings in the string *ababc*, as can be calculated from the suffix tree in Figure 1.1. Thus, the suffix tree represents a compact index of all distinct substrings of a given input string: it represents a quadratic number of substrings in a linear space (see Section 2.2).

The exact pattern matching using suffix trees is very efficient. In fact, each query pattern q can be located in X by following the path of symbols from the root of the suffix tree for X. The substring containment problem, namely whether q is a substring of X, can be solved in time proportional to the length of the query q and *independent* of the size of the pre-processed input. An example of a search is shown in Figure 1.2. In order to report all occurrences *occ* of q, the subtree induced by the end of the corresponding path is traversed, which results in a search with an optimal

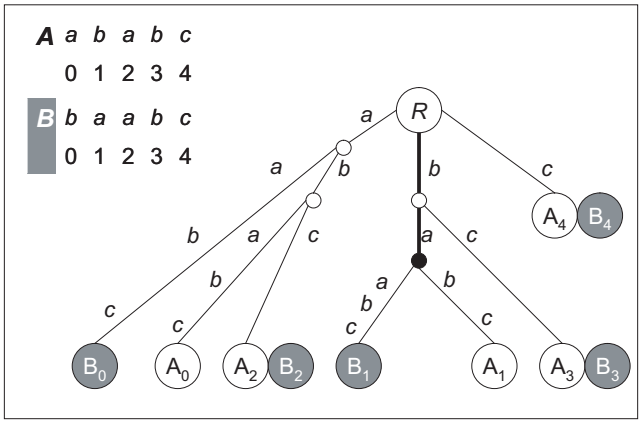

Figure 1.3: The generalized suffix tree for input strings $A = abbab$ and $B = babab$. All substrings common to both A and B can be found in time linear in the total input length. The common substrings end in the nodes that have leaves from both A and B in their corresponding subtrees, such as the node where the path for ba (shown in bold) ends.

performance of $O(|q| + occ)$.

If we are about to compare substrings of several genomic sequences, we can do this using a generalized suffix tree for a set of strings. An example of the generalized suffix tree for sequences $A = abbab$ and $B = babab$ is shown in Figure 1.3.

Using generalized suffix tree, we can find all substrings common to both input strings. In fact, the common substrings correspond to the labels of the paths ending in all nodes of the suffix tree whose leaves contain positions from both input strings in the induced subtree. For an example, consider the internal node reached by path ba highlighted in Figure 1.3.

In terms of genomic information, the sufficiently long substrings occurring in multiple genomes of different species point to conserved regions, which were preserved during the evolution. The possibility of finding these important common regions is greatly facilitated by the use of the suffix trees.

The same is true for unique substrings. The example in Figure 1.4 shows how to find substrings unique to a given input string. In terms of sequenced genomes,

discovering new unique substrings could help to map the positions of genes in the massive sequence by their proximity to these unique markers.

These are only some examples of the potential use of the suffix trees. In fact, theoretically optimal bounds were obtained for many non-trivial tasks such as computing matching statistics, locating all repetitive substrings, extracting palindromes and so on [26]. As Gusfield puts it in [26]:"Perhaps the best way to appreciate the power of suffix trees is ... to spend some time trying to solve [these] problems without using suffix trees. Without this effort and without some historical perspective, the availability of suffix trees may make certain problems appear trivial, even though linear-time algorithms for those problems were unknown before the advent of suffix trees. (page 122)".

However, all the benefits described in his book [26] appear to be illusory, since, when we started our work, the suffix trees for sufficiently large inputs could not be efficiently built [52].

Thus, this work is dedicated to the **engineering of practical algorithms for the construction of the suffix trees for very large inputs**.

The rest of the thesis is organized as follows. In Chapter 2 we give a formal definition of the suffix tree data structure and present the main challenges that we faced on the way to the fully-scalable efficient suffix tree construction. Then, in Chapter 3 we describe our new solution for building suffix trees for large inputs. In Chapter 4 we take a scalability of the suffix trees to the next level, by presenting a suffix tree construction algorithm for inputs several times larger than the main memory. In Chapter 5 we give basic instructions on the usage of our new scalable suffix trees. Finally, in Chapter 6 we present the remaining challenges that are to be overcome in future research.

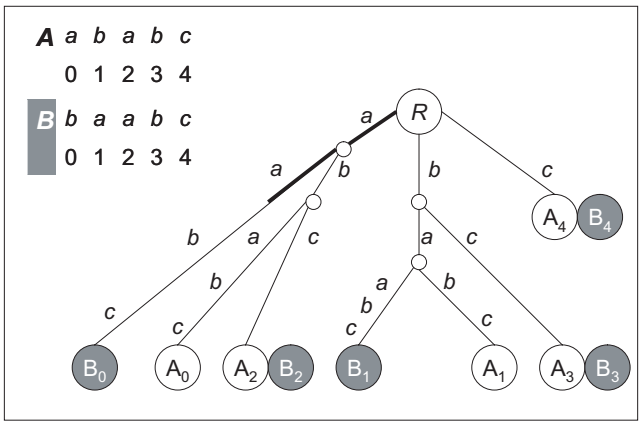

Figure 1.4: The substring aa, unique to the input string $B = babab$, is shown as a bold path in this generalized suffix tree for $A = abbab$ and $B = babab$. Any substring that ends on a leaf edge is unique, since it occurs only once, otherwise it would label a path to some internal node.

Chapter 2

Problem definition

In this chapter we describe the problems that we solve in this work. First, we give an introduction to suffix trees in Section 2.1. Next, in Section 2.2 we describe its space requirements, which cause problems in the construction and storage of these full-text indexes. In Section 2.3 we outline two possible solutions of the suffix tree memory problem: index compression and use of external storage (disk). Here we also present arguments for choosing the use of disk space over index compression. Then, in Section 2.3.2 we describe memory hierarchies and the requirements for disk-friendly suffix tree construction. We conclude by stating the two major bottlenecks for the suffix tree scalability, namely random access to the suffix tree and random access to the input string.

2.1 Introduction to suffix trees

The suffix tree was introduced by Weiner in 1973 [70]. We start out by taking a closer look at the suffix tree data structure and its representation.

First, we equip ourselves with some useful definitions.

We consider a *string* $X = x_0x_1 \dots x_{N-1}$ to be a sequence of N symbols. $N - 1$

symbols are over an alphabet Σ. The last symbol $x_{N-1} = \$$, which we attach to the end of X is unique and not in Σ (a so-called *sentinel*). Depending on the application, we regard \$ as having a lexicographical value lower or greater than any other symbol.

By $S_i = X[i, N-1]$ we denote a *suffix* of X beginning at position i, $0 \leq i < N$. Thus $S_0 = X$ and $S_{N-1} = \$$. Note that we can uniquely identify each suffix by its starting position.

Prefix P_i is a substring $[0, i]$ of X. The *longest common prefix* LCP_{ij} of two suffixes S_i and S_j is a substring $X[i, i+k]$ such that $X[i, i+k] = X[j, j+k]$, and $X[i, i+k+1] \neq X[j, j+k+1]$. For example, if $X = ababc$, then $LCP_{0,2} = ab$, and $|LCP_{0,2}| = 2$.

If we sort all the suffixes of string X in lexicographical order and record this order into an array SA of integers, then we obtain the *suffix array* of X [45]. SA holds all integers i in the range $[0, N]$, where i represents S_i. In more practical terms, suffix array SA is an array of positions sorted according to the lexicographic order of their corresponding suffixes. Note that the suffixes themselves are not stored in this array but are rather represented by their start positions. For example, for $X = ababc\$$ $SA = [5, 0, 2, 1, 3, 4]$. The suffix array can be augmented with the information about the longest common prefixes for each pair of suffixes represented as consecutive numbers in SA, as shown in the example of Figure 2.1.

A *trie* is a type of digital search tree [36]. In a trie, each edge represents a character from the alphabet Σ. The maximum number of children for each trie node is $|\Sigma|$, and sibling edges must represent distinct symbols. A *suffix trie* is a trie for all the suffixes of X. As an example, the suffix trie for $X = ababc$ is shown in Figure 2.2 [Left]. Beginning at the root node, each of the suffixes of X can be found in the trie: starting with *ababc*, *babc*, *abc*, *bc* and finishing with a *c*. Because of this organization, the occurrence of any query substring of X can be found by starting at the root and

X	a	b	a	b	a	a	a	b	b	c
	0	1	2	3	4	5	6	7	8	9

suffix start	4	5	2	0	6	3	1	7	8	9
	a	a	a	a	a	b	b	b	b	c
	a	a	b	b	b	a	a	a	b	
	a	b	a	a	b	a	b	b	c	
	b	b	a	b	c	a	a	b		
	…	…	…	…		…	…	…		
LCP	0	2	1	3	2	0	2	3	1	0

Figure 2.1: Suffix array with LCP for string $X = ababaaabbc$.

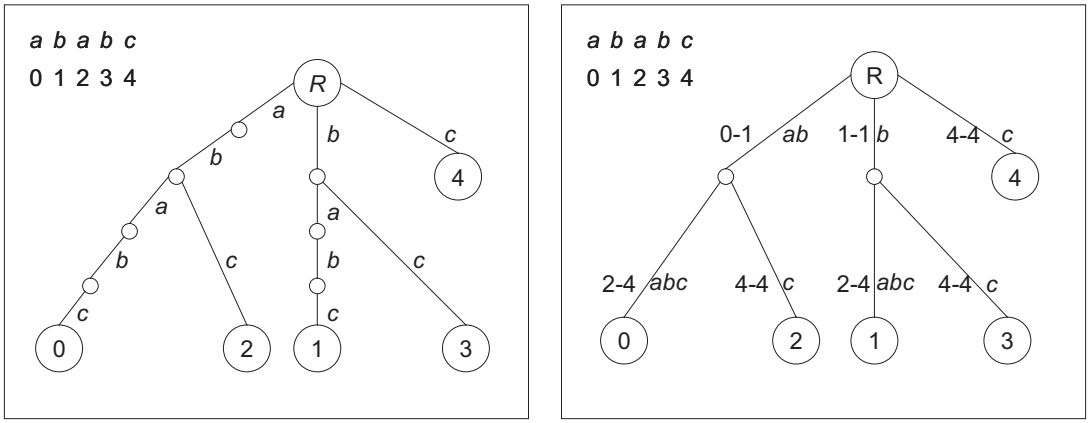

Figure 2.2: [**Left**] The suffix *trie* for $X = ababc$. Since c occurs only at the end of X, it can serve as a unique sentinel symbol. Note that each suffix of X can be found in the trie by concatenating character labels on the path from the root to the corresponding leaf node. [**Right**] The suffix tree for $X = ababc$. For clarity, the explicit edge labels are shown, which are represented as ordered pairs of positions in the actual suffix tree. Each suffix S_i can be found by concatenating substrings of X on the path from the root to the leaf node L_i.

following matches down the trie edges until the query is exhausted. In the worst case, the total number of nodes in the trie is quadratic in N. This situation arises, for example, if all the paths in the trie are disjoint, as for the input string *abcde*.

The number of edges in the suffix trie can be reduced by collapsing paths containing unary nodes into a single edge. This process yields a structure called the *suffix tree*. Figure 2.2 [Right] shows what the suffix trie for X looks like when converted to a suffix tree. The tree still has the same general shape, but far fewer nodes. The leaves are labeled with the start position in X of corresponding suffixes, and each suffix can be found in the tree by concatenating substrings associated with edge labels. In practice, these substrings are not stored explicitly but are represented as an ordered pair of integers indexing its start and end position in X. The total number of nodes in the suffix tree is constrained due to two facts: (1) there are exactly N leaves and (2) the degree of any external node is at least 2. There are therefore at most $N - 1$ internal nodes in the tree. Hence, the maximum number of nodes (and edges) is linear

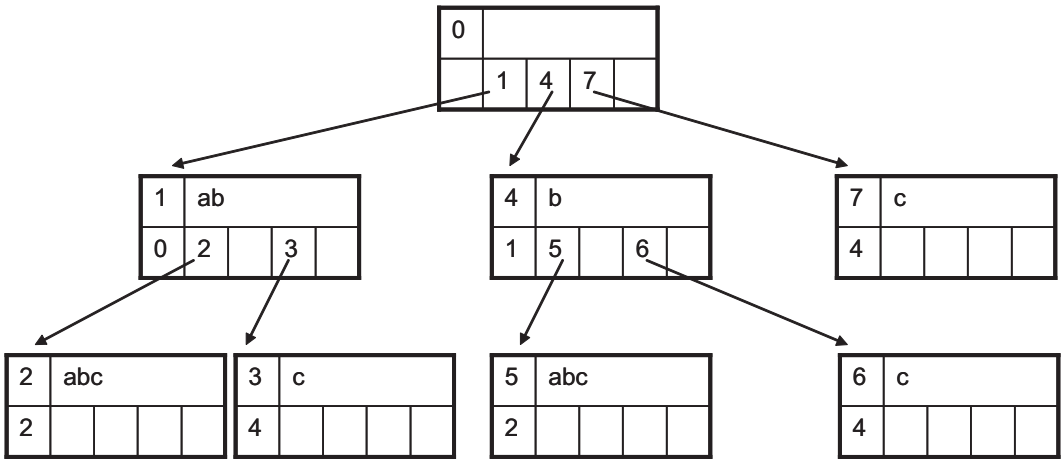

Figure 2.3: An array representation of the suffix tree for $X = ababc$. Each node contains an array of 4 child pointers. Note that not all the cells of this array are in use. The strings in the nodes are the labels of the incoming edges. They are shown for clarity only and are not stored explicitly.

in N. The tree's total space is linear in N for the case that each edge label can be stored in constant space. Fortunately, this is the case for an implicit representation of substrings by their positions.

In a nutshell, a *suffix tree* is a digital tree of symbols for the suffixes of X, where edges are labeled with the start and end positions in X of the substrings they represent. Note also that each internal node in the suffix tree represents the end of the longest common prefix for some pair of suffixes.

2.2 Suffix tree storage requirements

We discuss next the problem of suffix tree representation in memory in order to estimate the space requirements for suffix trees.

It is common to represent the node of a suffix tree together with the information about an incoming edge label. Each node, therefore, contains two integers representing the start and end position of the corresponding substring of X. In fact, it is

enough to store only the start position of this substring as the end position can be deduced from the start position of the child (for an internal node) or is simply N (for a leaf node). In a straightforward implementation, each node has pointers to all its child nodes. These child pointers can be represented as an array, as a linked list or as a hash table [26].

If the size of Σ is small, the child node pointers can be represented in form of an array of size $|\Sigma|$. Each i^{th} entry in this array represents the child node whose incoming label starts with the i^{th} character in a ranked alphabet. This is very useful for tree traversals, since the corresponding child can be located in constant time. Let us first consider the tree space for inputs where N is less than the largest 4-byte integer, i.e. $\log N < 32$. In this case, each node structure consists of $|\Sigma|$ integers for child node pointers plus one integer to represent the start position of the edge-label substring. Since there are at most $2N$ nodes in the tree, the total space required is $2N(|\Sigma| + 1)$ integers, which, for example, for $|\Sigma| = 4$ (DNA alphabet) yields $40N$ bytes of storage per N bytes of input. Such a representation is depicted in Figure 2.3.

For larger alphabets, an array representation of children is impractical and can be replaced by a linked list representation [26]. However, this requires an additional $log|\Sigma|$ search time spent at each internal node during the tree traversal, in order to locate the corresponding child. In addition, since the position of a child in a list does not reflect the first symbol of its incoming edge label, we may need to store an additional byte representing this first character.

Another possibility is to represent child pointers as a hash table [26]. This preserves a constant-time access to each child node and is more space-efficient than the array representation.

The linked-list based representation known as a "left-child right-sibling" was proposed by McCreight in [48]. In this implementation, the suffix tree is represented as

a set of node structures, each consisting of the start position of the substring labeling the incoming edge, together with two pointers – one pointing to the node's first child and the other one to its next sibling. Recall that the end position of the edge-label substring is not stored explicitly, since for an internal node it can be deduced from the start position of its first child, and for a leaf node this end position is simply N. This representation of the node's children is of type linked list, with all its space advantages and search drawbacks. The McCreight suffix tree representation is illustrated in Figure 2.4 [A]. Each suffix tree node consists of three integers, and since there are up to $2N$ nodes in the tree, the size of such a tree is at most $24N$. Again, for better traversal efficiency, we may store the first symbol along each edge label. Then the total size of a suffix tree will be at most $25N$ bytes for N bytes of input.

An even more space efficient storage scheme was proposed by Giegerich et al. [24]. In this optimization, the pointers to sibling nodes are not stored, but the sibling nodes are placed consecutively in memory. The last sibling is marked by a special bit. Now, each node stores only the start position of a corresponding edge-label plus the pointer to its leftmost child. As before, for efficiency of the traversal, each node may store an additional byte representing the start symbol of its edge label. The size of such a tree node is 9 bytes. For a maximum of $2N$ nodes this yields a maximum of $18N$ bytes of storage. Giegerich et al.'s [24] representation is depicted in Figure 2.4 [B].

Note that in all representations the leaf nodes do not contain child pointers. Thus, at the end of the construction we can store the leaf nodes in a separate array. Each element in the array of leaf nodes stores only the start position of the corresponding substring since the end position is implied to be N. In this case, the array representation occupies $24N$ bytes (for $\Sigma = 4$), the McCreight suffix tree occupies $20N$ bytes, and the Giegerich et al.'s representation occupies $12N$ bytes.

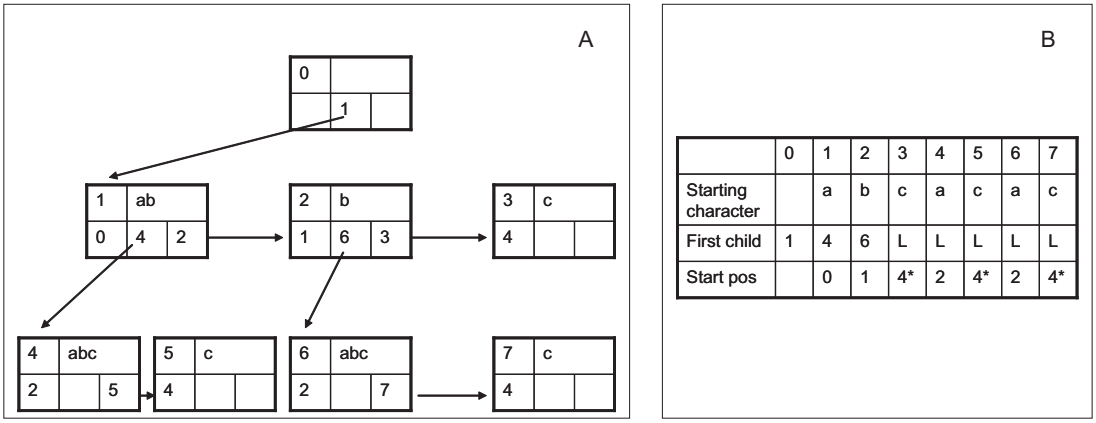

Figure 2.4: **[A].** Left-child right-sibling representation of the suffix tree for $X = ababc$. Each node contains 1 pointer to its first child and 1 pointer to the next sibling. **[B].** Giegerich et al.'s representation of the suffix tree, where all siblings are represented as consecutive elements in the array of nodes. The special symbol $\star$ indicates the bit representing the last sibling. Each node contains only a pointer to the first child and the start position of the incoming edge-label.

The suffix tree, theoretically, is a compact index, since it stores in a linear-space the total quadratic number of distinct substrings of X. However, this short survey of storage requirements clearly demonstrates the fact that the suffix tree index is very space-demanding. For example, for an input of 2 GB, the tree requires at least 24 GB of memory. Further, for inputs exceeding in size the largest 4-byte integer, the start positions and the child pointers need more than 4 bytes for their representation, namely $\log N$ bits for each number. In practice, for the inputs of a size in the tens of gigabytes the tree can easily reach $50N$ bytes.

Due to these excessive memory requirements, the power of suffix trees till recently has remained largely unharnessed.

2.3 Possible solutions to the memory problem

There are two main research directions for overcoming the memory bottleneck of suffix trees: the index compression and the use of disk space.

2.3.1 Index compression

Until 2007, the data structure by Giegerich et al. [24] was known as the most space efficient representation. Then Sadakane [59] fully developed the compressed suffix tree and its balanced parenthesis representation. The compressed representation allows the entire suffix tree to be stored in only $5N$ bits.

An example of the parenthesis representation of the suffix tree nodes for string $X = ababc$ is shown in Figure 2.5. The parentheses describe the tree topology. In order to store the information about the start position and the depth of each tree node, a special array and its unary encoding are used to bring the total memory requirements for the tree to $5N$ bits [59]. The compressed suffix tree supports all

regular suffix tree queries with a poly-log slowdown [59].

The algorithm for the compressed suffix tree construction was implemented (see [67]) and is available for indexing genomic sequences [66]. The algorithm requires a large amount of memory during the construction of the compressed tree (about 30 GB for indexing 3 GB of the Human genome). In [60], a new method for construction of compressed full-text indexes was introduced, which uses the disk space in the intermediate construction phase.

Compressed suffix trees and arrays represent conceptually new *self-indexing* structures, which do not require the input string for search and traversal. The resulting compressed self-index is smaller than the original input, and it must be kept entirely in the main memory during the search. For example, the compressed suffix tree for 3 GB of Human genome occupies only 2 GB of memory. These impressive results show that the entire fully-functional suffix tree can be stored using much less space than previously believed.

Thus, if the size of main memory permits, the compressed full-text index can be used for indexing genomes. However, the practical scalability of compressed suffix trees does not go beyond the inputs that are smaller than the main memory, since the index itself has size in the same order of magnitude as the input it is built upon, and it should be completely loaded into memory to be useful [67]. If the compressed index outgrows the main memory and is accessed from disk, then severe disk thrashing occurs due to extremely poor locality of references.

More important from a practical point of view is the fact that the algorithms for search in compressed indexes perform with a poly-log slowdown compared to the optimal algorithms on uncompressed indexes. For example, the traversal of a compressed suffix tree for 3 GB of Human genome was 90 times slower than the traversal of a conventional suffix tree [20]. The poly-log slowdown will become even

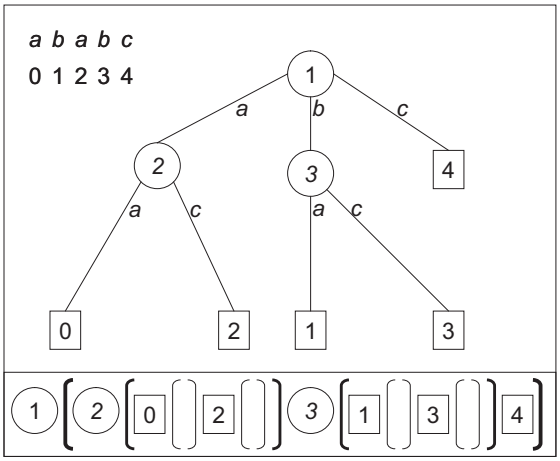

Figure 2.5: The parenthesis tree representation is a main high-level idea for the suffix tree compression [59].

more prominent with the growth of the input size.

More about compressed suffix trees can be found in recent papers [19, 58]. Aiming to develop a *practical* solution for DNA indexing, we do not consider the compressed suffix trees in this work, but rather concentrate on the idea of using external memory for tree construction and search.

2.3.2 Using disk space

We now turn to the idea of using disk during for the construction of the suffix tree.

Recall from Section 2.2 that for the most space efficient [39] representation of the suffix tree for an input of size 6 GB, we need 60 GB of space. Real genomic data, however, is often much larger than 6 GB. For example, the genome of Lilium longiflorum (trumpet lily) alone is 90 GB [25] in size, and converting this input string into a suffix tree requires at least 900 GB of memory.

Consider the idea of using disk space to store intermediate and resulting data structures during the suffix tree construction and queries, without ever loading the entire index into the main memory. This solution is quite attractive since disk space is cheap and virtually unlimited: we can hold on disk several TB of data.

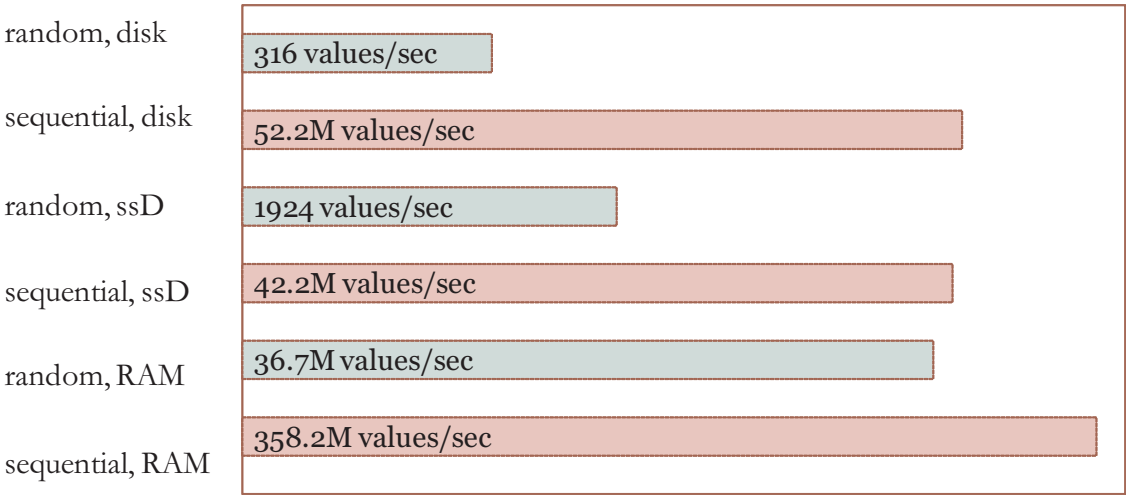

Figure 2.6: Data transfer speed for different memories [29].

However, in order for the suffix tree construction algorithm to be efficient we need to take into account some properties of the real computer memory hierarchies.

There are several categories of memory in a computer ranging from small and fast to cheap and slow. The access to data on a disk is 10^5-10^6 times slower than the access to data in main memory [68].

In order to model these speed differences in the design of algorithms using external memory, the external memory computational model, or **d**isk **a**ccess **m**odel (DAM), was proposed [69]. DAM represents the computer memory in the form of two layers with different access characteristics: the fast main memory of a limited size M, and a slow and arbitrarily large secondary storage memory (disk). In addition, for disks, it takes about as long to fetch a consecutive block of data as it does to fetch a single byte. That is why in the DAM computational model the asymptotic performance is evaluated as the total number of block transfers between a disk and main memory.

Although the DAM computational model is a workable approximation, it does not always accurately predict the performance of algorithms that use disk space. This is because DAM does not take into account the following important disk access property. The cost of a random disk access is the sum of seek time, rotational delay

and transfer time. The first two dominate this cost in the average case, and as such, are the bottleneck of a random disk access. However, if the disk head is positioned exactly over the piece of data we want, then there are neither seek time nor rotational delay components, but only transfer time. Hence if we access data sequentially on disk, then we only pay seek time and rotational delay for locating the first block of our data, but not for the subsequent blocks. The difference in cost between sequential and random access becomes even more prominent if we also consider read-ahead-buffering optimizations which are common in current disks and operating systems [13].

In fact, the real time of data transfer from disk and from RAM differs significantly depending on the access patterns. Figure 2.6 represents the results of experiments on data transfer speeds for different memories (from [29]). These results show that the random access to the main memory is even slower than the sequential access to disk. Thus, in order to design a truly efficient algorithm for a suffix tree construction that uses disk space, we need to strive to avoid random accesses to the data on disk as much as possible.

The suffix tree construction algorithms we describe in this work are intended for suffix trees that are much larger than the available main memory. Thus, **the first challenge is to reduce random access to the suffix tree** during its construction.

As the input data becomes more massive, we are facing **the second challenge which is to reduce the random access to the input string** during tree construction.

Thus, our refined research goal becomes:

Design a suffix tree construction algorithm that:

- avoids random access to the suffix tree during its construction
- avoids random access to the input string
- avoids random access to both tree and string during the search

Chapter 3

Minimizing random access to the suffix tree

In this chapter we present our new algorithm for constructing generalized disk-based suffix trees. This algorithm is simple, easy to implement and maintain, efficient, and more scalable than the state–of–the–art.

We start in Section 3.1 with a detailed overview of existing algorithms for disk-based suffix tree construction. In Section 3.2.1 we describe techniques used in the design of our algorithm. The detailed description of *DiGeST* is given in Section 3.2.2. Finally, in Section 3.2.3 we present experimental results for building suffix trees for several sequenced genomes. These results show that our new algorithm scales to very large inputs. The inputs, however, should entirely fit into the main memory.

In this chapter we consider the case when the random access to the input string occurs in main memory, but the suffix tree is incrementally written to disk.

3.1 Related work

All the algorithms presented in this section are intended for the case when the input string is kept entirely in main memory. In other words, none of them takes into account the random access to the input string, improving upon random access to the suffix tree being built. Since the suffix tree is much larger than the input, placing suffix tree on disk is the first logical step towards the scalability of the suffix tree construction.

We start by describing the linear-time construction algorithms that work well in main memory, but cannot be successfully extended to a disk version. In the remainder of the section, we present the review of the recent developments in practical disk-based suffix tree construction, with an emphasis on the efficiency and the scalability of these algorithms as applied to genomic data.

3.1.1 The Ukkonen algorithm and its disk-based version

The suffix tree for input string X of length N can be built in time $O(N)$. Linear-time algorithms were developed in [70, 48, 65]. In [23] it was shown that all three of them are based on similar algorithmic techniques, namely the use of *suffix links* (to be defined later in this section). The simplest and most frequently used out of these three algorithms is the Ukkonen algorithm [65]. It is tempting to use this asymptotically optimal algorithm for an external memory implementation. However, looking closely at Ukkonen's algorithm, we observe that it assumes that random access both to the input string and to the tree takes constant time. Unfortunately, in practice, when any of these data structures outgrows the main memory and is accessed directly on disk, the access time to disk-based arrays varies significantly depending on the relative location of the data on disk (see Section 2.3.2). The total number of random disk

accesses for this algorithm is, in fact, $O(N)$. This is extremely inefficient and causes the so-called disk thrashing problem, which let the authors in [52] conclude that suffix trees in secondary storage are inviable.

The random access behavior of the Ukkonen algorithm [65] was improved for external memory settings in [6]. We describe next the Ukkonen algorithm and show how it was extended for external memory.

For a given string X, Ukkonen's algorithm starts with the empty tree (that is, a tree consisting just of a root node) and then progressively builds an intermediate suffix tree ST_i for each prefix $X[0, i]$, $0 \leq i < N$. In order to convert a suffix tree ST_{i-1} into ST_i, each suffix of ST_{i-1} is extended with the next character x_i. We do this by visiting each suffix in order, starting with the longest suffix and ending with the shortest one (empty string). The suffixes inserted into ST_{i-1} may end in three types of nodes: leaf nodes, internal nodes or in the middle of an edge (at a so-called *implicit* internal node). Note that if a suffix of ST_{i-1} ends in a leaf node, we do not have to explicitly extend it with the next character. Instead, we consider each leaf node as an *open* node, i.e. the end position of its incoming edge-label is updated implicitly to the current value of i in ST_i, and at the end eventually becomes N.

Consider the example in Figure 3.1. It shows the three first iterations of the suffix tree construction for $X = ababcababd$. In the second iteration, we implicitly extend the a-child of a root node with b, and we add a new edge for b from the root (extending an empty suffix).

Thus, in each iteration, we need to update only suffixes of ST_{i-1} that end at explicit or implicit internal nodes of ST_{i-1}. We find the end of the longest among such suffixes at the *active point*. The active point is the (explicit or implicit) internal node where the previous iteration ended. If the node at the active point already has a child starting with x_i, the active point advances one position down the corresponding

edge. This means that all the suffixes of ST_i already exist in ST_{i-1} as the prefixes of some other suffixes. In the case that there is no outgoing edge starting with the new character, we add a new leaf node as a child of our explicit or implicit internal node (active point). Here, an implicit internal node becomes explicit. In order to move to the extension of the next suffix, which is shorter by one character, we follow the chain of *suffix links*. A suffix link is a directed edge from each internal node of the suffix tree (source) to some other internal node whose incoming path is one (the first) character shorter than the incoming path of the source node. The suffix links are added when the sequence of internal nodes is created during multiple splits of the edges.

To illustrate, consider the last iteration of the Ukkonen algorithm – extending an intermediate tree for $X = ababcababd$ with the last character d. We extend all the suffixes of ST_8 (Figure 3.2 [A]) with this last character. The active point is originally two characters below the node labeled by $\star$ in Figure 3.2 [A], and the implicit internal node is indicated by a black triangle. The active point is converted to an explicit internal node with two children: one of them is the existing leaf with incoming edge label *cababd* and the other one is a new leaf for suffix S_5 (Figure 3.2 [B]). Then, we follow the suffix link from the $\star$-node to the $\star\star$-node, and we add a new leaf by splitting an implicit node two characters below the $\star\star$-node. This results in the tree of Figure 3.2 [C] with a leaf for suffix S_6. Next, the suffix link from the $\star\star$-node leads us to the root node, and two characters along the corresponding edge we find the $\star$-node and add to it a new edge starting with d and leading to a leaf node for suffix S_7 (Figure 3.3 [D]). We continue in a similar manner and add the corresponding child starting with d both to the $\star\star$-node (Figure 3.3 [E]) and to the root (Figure 3.3 [F]). This illustrates how suffix links help to find all the insertion points for the new leaf nodes, making the amortized time of this algorithm $O(N)$:

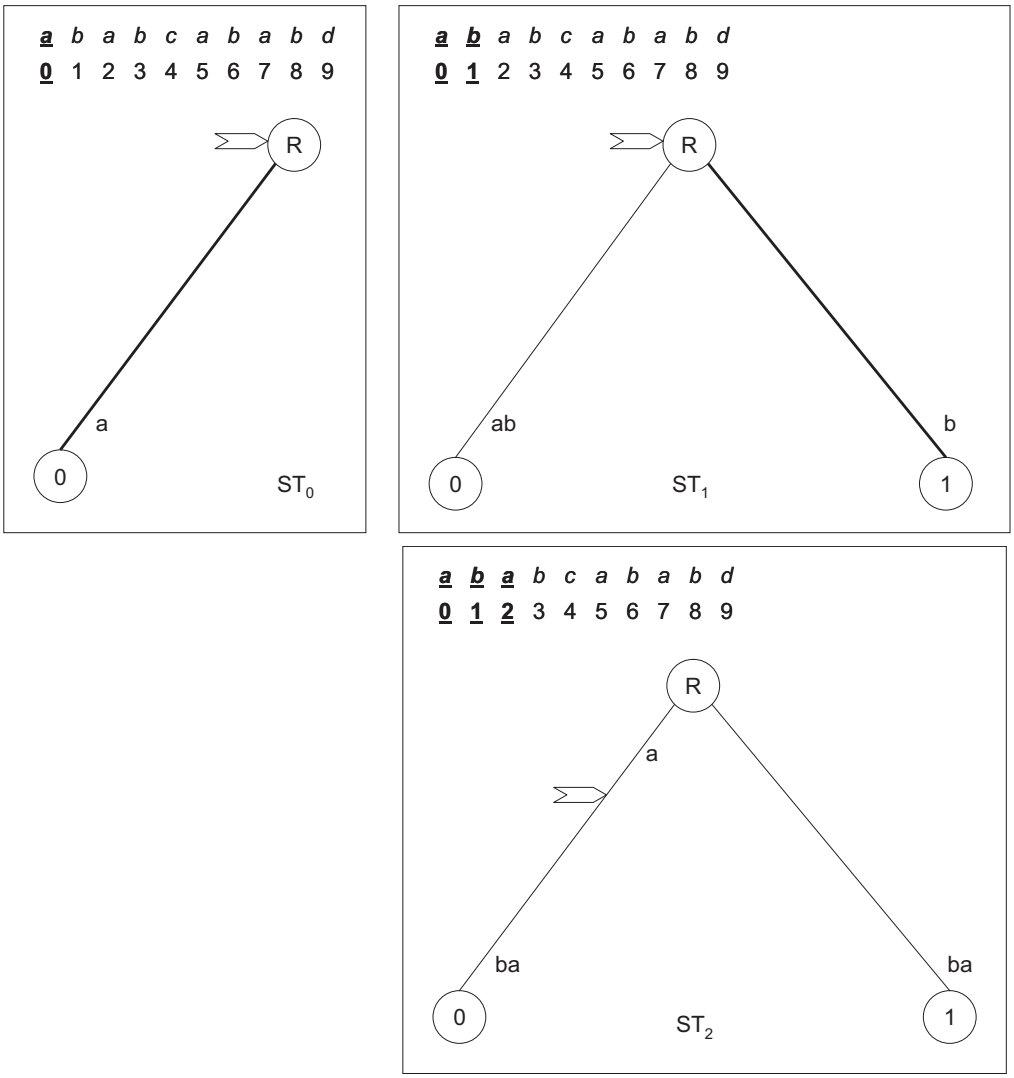

Figure 3.1: The three first steps of the Ukkonen algorithm. An arrow indicates the active point at the end of each iteration. Note that the extension of the edges ending at leaf nodes with the next character is performed implicitly: the edge length is just extended by 1.

there is a constant number of steps per leaf creation, and exactly N leaves.

The pseudocode in Figure 3.4 shows the procedure *update* for converting ST_{i-1} into ST_i [53]. Each call of *next_smaller_suffix()* finds the next suffix by following a suffix link.

If we look at the pseudocode of Figure 3.4 from the disk access point of view, we see that locating the next suffix requires a tree traversal to the arbitrary(non-sequential) position, one per leaf created. Hence, when the tree ST_{i-1} is to be stored on disk, a node access requires an entire random disk I/O. This access time depends on the disk place of the next access point. Moreover, since the edges of the tree are not labeled with actual characters, it is important that we access the input string in order to compare the test_char with the characters of X encoded as non-sequential positions in the suffix tree edges. Unfortunately, the arbitrary tree traversals lead to a very impractical performance [6], since the algorithm spends all its time moving the disk head from one random disk location to another.

In [6], Bedathur and Haritsa studied the patterns of node accesses during the suffix tree construction based on Ukkonen's algorithm. They found that the higher tree nodes are accessed much more frequently than the deeper ones. This gave rise to the buffer management method known as *TOP-Q*. In this on-disk version of Ukkonen's algorithm, the nodes that are accessed often, have a priority of staying in the memory buffer, and the other nodes are eventually read from disk. This significantly improved the hit rate for accessed nodes when compared to rather straightforward implementations. However, in practical terms, in order to build the suffix tree for the sequence of the Human chromosome I (approximately 247 MB), the *TOP-Q* runs for 96 hours, as was recently evaluated using a modern machine[1][64], and cannot be considered a practical method for indexing large inputs.

[1]We refer to the average machine currently available (Pentium 4 with 2.8 GHz clock speed and 2 GB of main memory) as the modern machine.

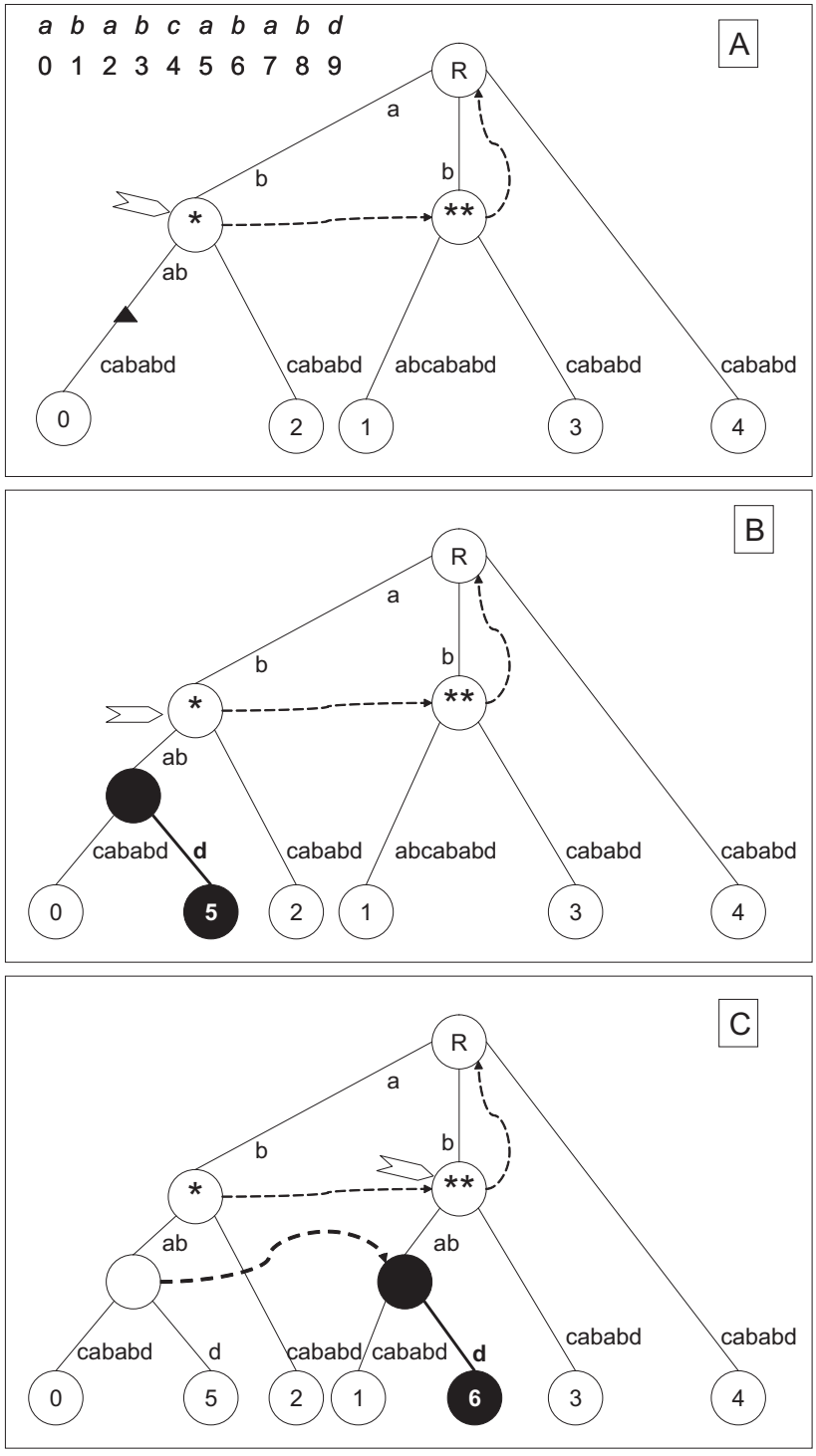

Figure 3.2: The last steps of the Ukkonen algorithm applied to $X = ababcababd$. In this cascade of leaf additions the ST_8 is updated to ST_9. The place for the next insertion is found following the suffix links (dotted arrows).

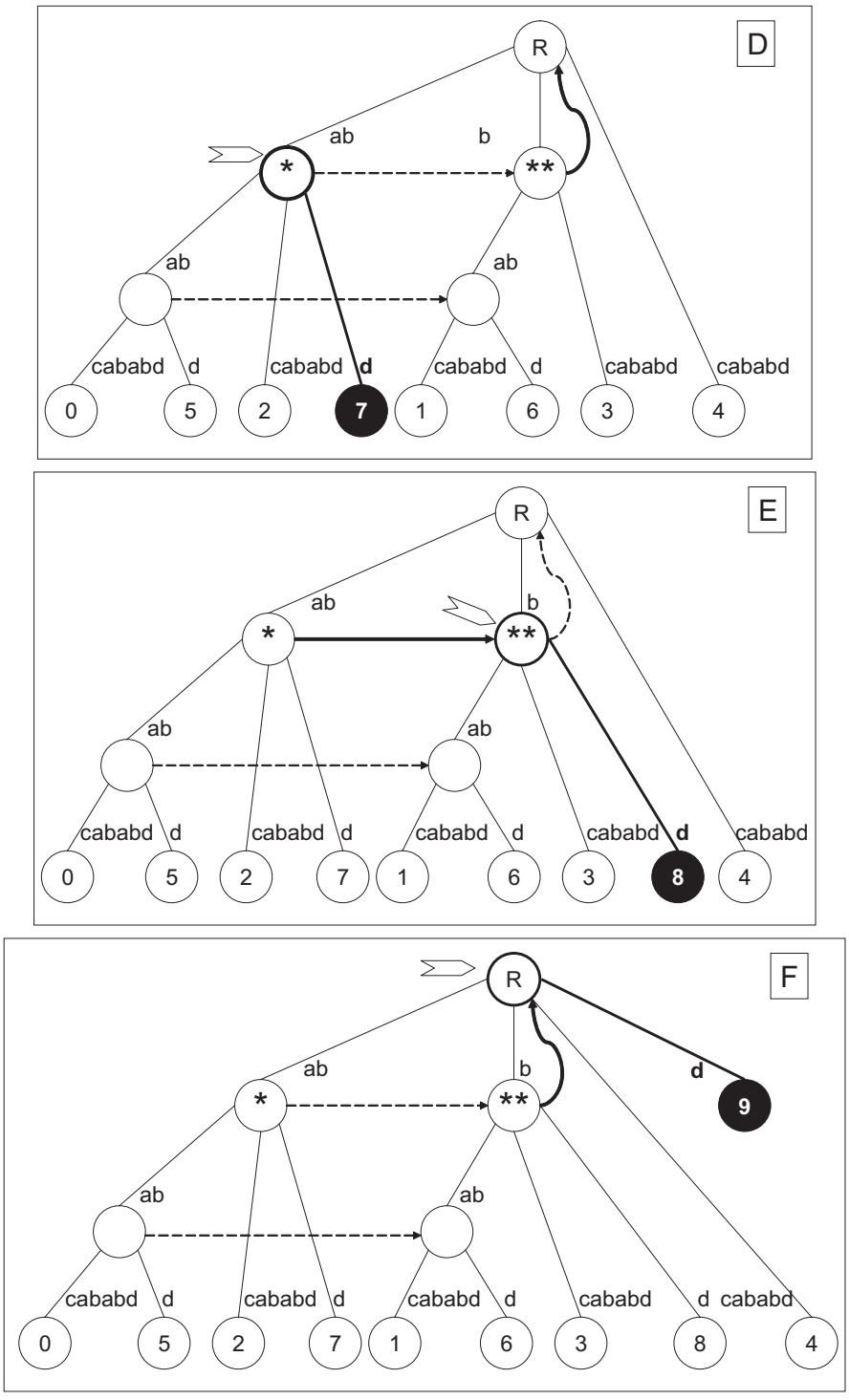

Figure 3.3: (Continued from Figure 3.2). The last steps of the Ukkonen algorithm applied to $X = ababcababd$. In this cascade of leaf additions the ST_8 is updated to ST_9. The place for the next insertion is found following the suffix links (dotted arrows).

```
Ukkonen MAIN (INPUT: string X of length N,
                    OUTPUT: suffix tree for X in RAM)
1  active_point=root
2  for i from 0 to N
3          call Update ( Prefix [0,i] )

4  Update (Prefix [0,i] )
5          curr_suffix_end = active_point
6          test_char = X [i]
7          done = false
8          while not done
9                  if curr_suffix_end in explicit node
10                         if the node has no descendant starting with
                                                                test_char
11                                 create new leaf
12                         else
13                                 advance active_point down the
                                                    corresponding edge
14                                 done = true
15                 else
16                         if the implicit node's next char is not test_char
17                                 create explicit node
18                                 create new leaf
19                         else
20                                 advance active_point down the
                                                    corresponding edge
21                         done = true
22                 if curr_suffix_end in root node (empty string)
23                         active_point=root
24                         done = true
25                 else
26                         curr_suffix_end = next_smaller_suffix()
27         active_point= curr_suffix_end //follow the suffix link
```

Figure 3.4: Pseudocode for Ukkonen's algorithm for the suffix-tree construction. The random accesses to the tree are performed in line 27. The random accesses to the input string are performed in line 16.

Next we describe a brute-force approach for the suffix tree construction, which runs in $O(N^2)$ time in the worst case. Amazingly enough, several fast practical methods for external memory were developed using this approach, due to the much better locality of tree accesses and the fact that the running time for most real-life inputs never reaches the quadratic asymptote. The average running time is $O(N \log N)$ as was shown in [3], and it is indeed $O(N \log N)$ for most real-life inputs.

3.1.2 The brute-force approach and the Hunt algorithm

An intuitive method of constructing a suffix tree ST is the following: for a given string X we start with a tree consisting of only a root node. We then successively add paths corresponding to each suffix of X from the longest to the shortest. This results in the algorithm depicted in Figure 3.5 [Top].

Here, ST_{i-1} represents the suffix tree after the insertion of all suffixes $S_0, \ldots S_{i-1}$. The *Update* operation inserts a path corresponding to the next suffix S_i yielding ST_i. In order to insert suffix S_i into the tree, we first locate some implicit or explicit node corresponding to the longest common prefix of S_i with some other suffix S_j. To locate this node, we perform $|LCP_{ij}|$ character comparisons. After this, if the path for LCP_{ij} ends in an implicit internal node, it is transformed into an explicit internal node. In any case, we add to this internal node a new leaf corresponding to suffix S_i. Once the end of the LCP_{ij} is found, we add a new child in constant time. Finding the end of LCP_{ij} in the tree defines the overall time complexity of the algorithm. The end of a LCP can be found in one step in the best case but in N steps in the worst case for each of N inserted suffixes. This can, in the worst case, lead to $O(N^2)$ total character comparisons.

However, Apostolico and Szpankowski have shown in [3] that on average the brute-force construction requires only $O(N \log N)$ time. Their analysis is based on the

```
Brute-force MAIN (INPUT: string X of length N,
                  OUTPUT: suffix tree of X in RAM)
1  for i from 0 to N
2        call Update ( Suffix [i,N] )

Update (Suffix [i,N] )
3        find LCP of Suffix [i,N] matching characters from the root
4        if LCP ends in explicit node
5                add child leaf labeled by X[i+LCP+1,N]
6        else
7                create explicit node at depth LCP from the root
8                add child leaf labeled by X[i+LCP+1,N]
```

```
Hunt MAIN (INPUT: string X of length N)
1  for each prefix PR of length prefix_len
2   for i from 0 to N
3        if S_i has prefix PR then
4                call Update ( Suffix [i,N], PR )
5   OUTPUT sub-tree for prefix PR to disk

Update (Suffix [i,N], PR )
6   if X[i,i+prefix_len] equals PR
7        find LCP of Suffix [i,N] matching characters from the root
                                                    of the sub-tree
8        if LCP ends in explicit node
9                add child leaf labeled by X[i+LCP+1,N]
10       else
11               create explicit node at depth LCP from the root
12               add child leaf labeled by X[i+LCP+1,N]
```

Figure 3.5: [**Bottom**]. The pseudocode for Hunt et al.'s algorithm [28] for the suffix-tree construction based on the brute-force algorithm shown at the [**Top**]. In Hunt's algorithm, the entire subtree for prefix PR is built in main memory (lines 2-4). The disk writes are sequential (line 5), and there is no disk reads since the entire input string is in RAM. Note that there are PR scannings of the entire input string X (nested loop in lines 1,2).

assumption that the symbols of X are independent and randomly selected from an alphabet according to a given probability distribution. The average $O(N \log N)$ construction time also holds for many real-life inputs, unless they contain substantially long repetitions.

Based on this brute-force approach, the first practical external memory suffix tree construction algorithm was developed in [28]. Hunt et al.'s incremental construction trades an ideal $O(N)$ performance for locality of access to the tree during its construction. The output tree is in fact represented as a forest of several suffix trees. The suffixes in each such tree share a common prefix. Each tree is built independently and requires scanning of the entire input string for each such prefix. The idea is that the suffixes that have, for example, prefix *aa* fall into a different subtree than those starting with *ab*, *ac* and *ad*. Hence, once the tree for all suffixes starting with *aa* is built, it is never accessed again. The tree for each prefix is constructed independently in main memory, and then is written to disk.

The total number of sub-trees p is computed as the ratio of the space required for the tree of the entire input string, ST_{total}, to the size of the available main memory M, i.e. $p = |ST_{total}|/M$. Then, the length of the prefix for each sub-tree can be computed as $\log_{|\Sigma|} p$, where $|\Sigma|$ is the size of the alphabet. This works well for non-skewed input data but fails if for a particular prefix there is a significantly larger amount of suffixes. This is often the case in DNA sequences with a large amount of repetitive substrings. In order to fit a tree for each possible prefix into main memory, we can increase the length of the prefix. This, in turn, exponentially increases the total number of sub-trees, and therefore, the total number of input string scans.

The construction of the sub-tree for prefix *ab* and input string $X = ababcababd$ is shown in Figure 3.6. Note that the sub-tree is significantly smaller than the suffix tree for the entire input string. The pseudocode is given in Figure 3.5 [Bottom].

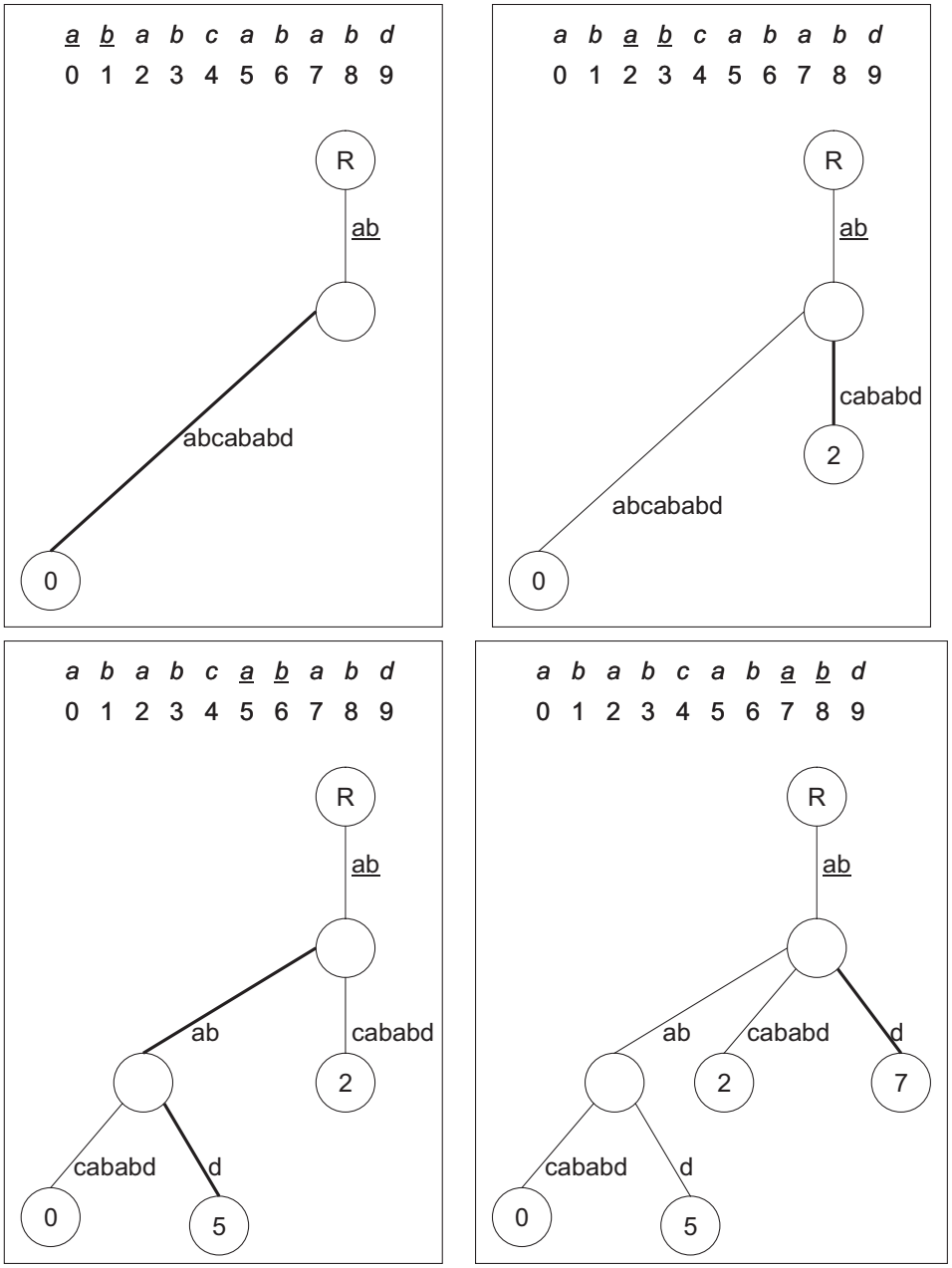

Figure 3.6: The steps of building the sub-tree for prefix *ab* and input string $X =$ *ababcababd* with the algorithm by Hunt et al. [28]

We remark that we iterate through the input string as many times as the total number of different prefixes. The construction of a tree for each prefix is performed in main memory. At the end, the suffix tree for each prefix is written to disk. Note also that in order to perform the brute-force insertion of each suffix into the tree we need to arbitrarily access the input string X, which therefore has to reside in memory. Since the input string is at least an order of magnitude smaller than the tree, this method efficiently addresses the problem of random accesses to the tree in secondary storage, but cannot be extended to inputs that are larger than the main-memory instantiation for holding X.

The algorithm performs much faster than the *TOP-Q* algorithm, despite the fact that its internal time is quadratic in the length of the input string. This is because for p prefixes all p passes over the input string are performed in main memory, and the tree is traversed in main memory as well. Thus, the algorithm performs only $O(p)$ random disk accesses: namely, when writing the tree for each prefix. For the Human DNA of size up to 247 MB (Human chromosome I) input, the suffix tree with Hunt et al.'s algorithm can be constructed in 97 minutes [64] compared to *TOP-Q*'s 96 hours for the same input on the same machine.

We remark that the performance of Hunt et al.'s algorithm degrades drastically if the input string does not fit the main memory and should be kept on disk. In this case it needs $O(pN)$ random disk accesses, that is how often it accesses the input string.

3.1.3 Distributed and paged suffix trees

A similar idea of processing suffixes of X separately for each prefix was developed in [11, 10]. The distributed and paged suffix tree (*DPST*) by Clifford and Sergot [11], which was proposed first in context of distributed computation, has all the properties

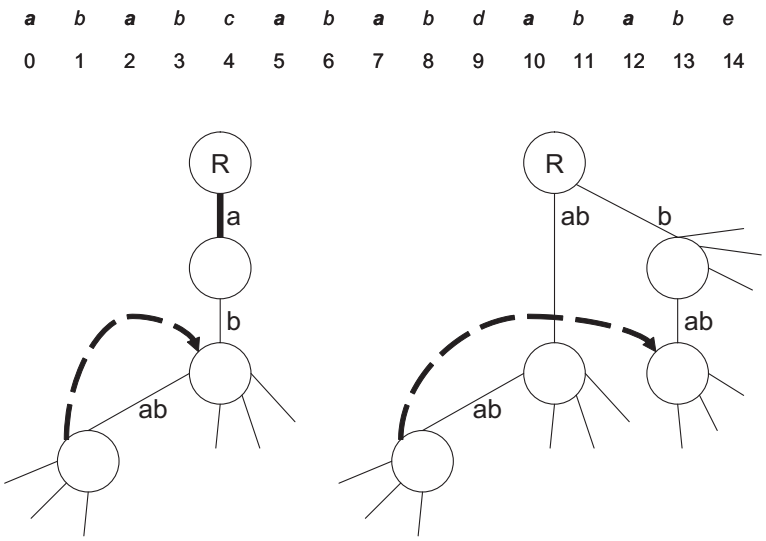

Figure 3.7: Difference between the sparse suffix links [**Left**] and the traditional suffix links [**Right**].

to be efficiently implemented to run using external memory. As before, the suffixes of X are grouped by their common prefix whose length depends on the size N of X and the amount of the available main memory. The number of suffixes in each subtree is small enough for the tree to be entirely built in main memory. Therefore, random disk access to the sub-tree during its construction is avoided. The main difference from Hunt et al.'s algorithm of the previous section is that the sub-tree for each particular prefix is built in an asymptotic time *linear* in N and not quadratic. In order to do so, the *DPST* algorithm uses the idea developed in [2] to build the suffix tree on words. The main ideas in [2] are similar to the Ukkonen algorithm [65] described in Section 3.1.1. However, the Ukkonen algorithm relies heavily on the fact that *all* suffixes of X are inserted, whereas the suffix tree on words is built only for some suffixes of X, namely the ones starting at positions marked by delimiters.

DPST applies the ideas of suffix tree on words by considering the particular prefix as the delimiter for producing the sub-tree for this prefix. It introduces the idea of *sparse suffix links* (SSL) instead of regular suffix links. A SSL in a particular subtree leads from each internal node v_i with incoming path label w to another internal node v_j in the same sub-tree whose incoming path-label corresponds to the largest possible suffix of w found in the same sub-tree (or to the root if the largest such suffix is an empty string).

We explain the difference between the sparse suffix link and the regular suffix link in the following example. Suppose we have a sub-tree for a prefix a for $X =$ *ababcababdababe* (see Figure 3.7). In the regular suffix tree, the suffix link from the internal node with an incoming path label *abab* leads to the node with the incoming path-label *bab*. However, in the sub-tree for prefix a, there is no suffix starting with *bab*. The longest suffix of *abab* that can be found in this sub-tree is *ab*, and the SSL leads to the internal node with the incoming path label *ab*.

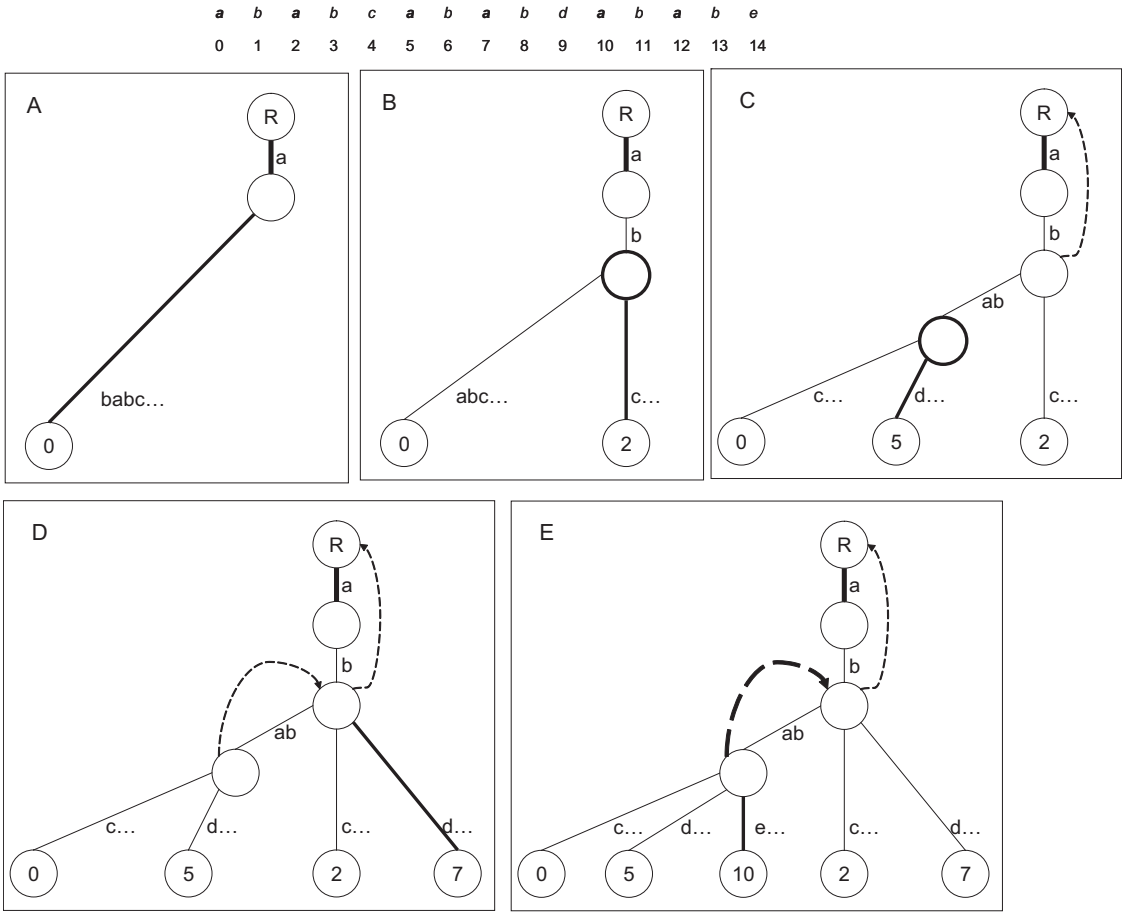

Figure 3.8: Steps of the construction of the sub-tree for prefix a by the distributed and paged suffix tree construction algorithm of Clifford and Sergot. Input string $X = ababcababdababe$.

Let us follow an example for the sub-tree construction for $X = ababcababdababe$ and prefix a in Figure 3.8. This sub-tree will contain only the suffixes of X starting at positions $0, 2, 5, 7, 10, 12$. Thus, we need to insert only these six suffixes to the tree. First, we insert suffix S_0 by creating leaf L_0. Next, we add S_2 by finding that $X[1] = X[3]$ and $X[2] \neq X[4]$. We split an edge and add leaf L_2. Now it is the turn for suffix S_5. Since the first four characters of S_5 correspond to some path in the tree, but $X[9] = d$ does not. Therefore, we add leaf L_5 and create an internal node with incoming path label *abab*. We see that the longest suffix of *abab* in this sub-tree is *ab*. We create a sparse suffix link from internal node for *abab* (marked by $\star\star$ in Figure 3.9) to the one for *ab* (marked by $\star$). When we create a new leaf out of the $\star\star$-node for suffix S_{10}, we follow the SSL and create the same e-child from the $\star$-node (Figure 3.9).

The use of these sparse suffix links for adding new leaves to the sub-tree allows to perform the construction of each sub-tree in time linear in N. The $DPST$ runs in time $O(NP)$ where P is the total number of different prefixes. Despite the superior asymptotic internal running time w.r.t. the previous algorithm, the practical performance and the scalability of the $DPST$ as implemented in [11] were inferior to the program by Hunt et al. [28] for the real DNA data used in their experiments.

3.1.4 Top Down Disk based suffix tree construction (*TDD*)

Quadratic in the worst case, but a more elaborated approach of the Top Down Disk based suffix tree construction algorithm (*TDD*) [64] takes the performance of the on-disk suffix tree construction to the next level. The base of the method is the combination of the *wotdeager* algorithm of Giegerich et al. [24] and Hunt et al.'s prefix partitioning described above. Being still an $O(N^2)$ brute-force approach, *TDD* manages more efficiently the memory buffers and is a cache-conscious method which

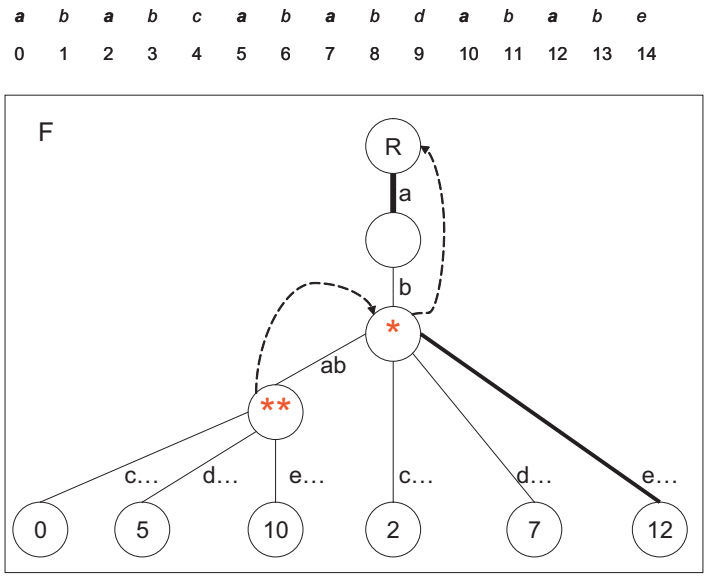

Figure 3.9: Sample output of the *DPST* algorithm by Clifford and Sergot.

```
TDD MAIN (INPUT: string X)
1  for each prefix PR of length prefix_len
2          collect from X start positions of suffixes starting with PR into array
3          sort suffixes by the first character
4          OUTPUT to disk groups with 1 suffix as leaf nodes of the tree
5          push groups with more than 1 suffix into the stack
6          while stack is not empty
7                  pop suffixes of the same group from the stack
8                  find LCP of all suffixes in the group
9                  OUTPUT to disk internal node at the depth LCP
10                 advance position of each suffix by LCP
11                 sort suffixes by the first character
12                 OUTPUT to disk groups with 1 suffix
                                        as leaf nodes of the tree
13                 push groups with more than 1 suffix into the stack
```

Figure 3.10: Pseudocode for the *TDD* algorithm [64]. All the suffix tree nodes are written directly to disk. There are some random accesses to the tree in lines 4,9,12. However, due to the top–down processing the repeating accesses to the written nodes are rare.

performs very well for many practical inputs.

The first step of *TDD* is the partitioning of the input string in a way similar to that of the algorithm by Hunt et al.. Now, the tree for each partition is built as follows. The suffixes of each partition are first collected into an array where they are represented by their start positions. Next, the suffixes are grouped by their first character into *character groups*. The number of different character groups gives the number of children for the current tree node. If for some character there is a group consisting only of one suffix, then this is a leaf node and is immediately written to the tree. If there is more than one suffix in the group, the LCP of all the suffixes is computed by sequential scans of X from different arbitrary positions, and an internal node at the corresponding depth is written to the tree. After advancing the position of each suffix by $|LCP|$, the same procedure as before is repeated recursively. The pseudocode for the *TDD* algorithm is given in Figure 3.10.

To illustrate the algorithm, let us observe several steps of the *TDD* suffix tree construction depicted in Figure 3.11. Suppose that we have partitioned all the suffixes of X by a prefix of length 1. This gives four partitions: a, b, c and d. We show how *TDD* builds the suffix tree for partition b. The start positions of suffixes starting with b are $\{1, 3, 6, 8\}$. Since the prefix length is 1, the characters at positions $\{2, 4, 7, 9\}$ are sorted lexicographically. This produces three groups of suffixes: a-group: $\{2, 7\}$, c-group: $\{4\}$ and d-group: $\{9\}$. Since the c-group and d-group contain one suffix each, the suffixes in these groups produce leaf nodes and are immediately added to the tree. The a-group contains two suffixes, and is therefore a branching node. $|LCP_{2,7}| = 2$, and therefore the length of the child starting with a equals 2. At this depth, the internal node branches at positions $\{4, 9\}$, which after sorting result into two leaf nodes: the children starting with c and d respectively.

The main distinctive feature of the *TDD* construction is the order in which the tree nodes are added to the output tree. Observe that the tree is written in a top-down fashion, and the nodes that were expanded in the current iteration are not accessed anymore. This reduces the number of non-sequential accesses to the partially built tree and the new nodes can be written directly to the disk. The number of random disk accesses is $O(P)$ as in Hunt et al.'s algorithm. However, the size of each partition may be much bigger than before since now the main memory buffer for the suffix tree data structure does not have to hold an entire sub-tree.

This pattern of accessing the tree was shown to be very efficient for cached architectures of the modern computer. It was even shown that the *TDD* algorithm outperforms the linear-time algorithm by Ukkonen for some inputs in case when all the data structures fit the main memory. For the same input of 247 millions of symbols (Human chromosome I) [73], which took about 97 minutes with the suffix-by-suffix insertion of Hunt, *TDD* builds the tree in 18 minutes [64].

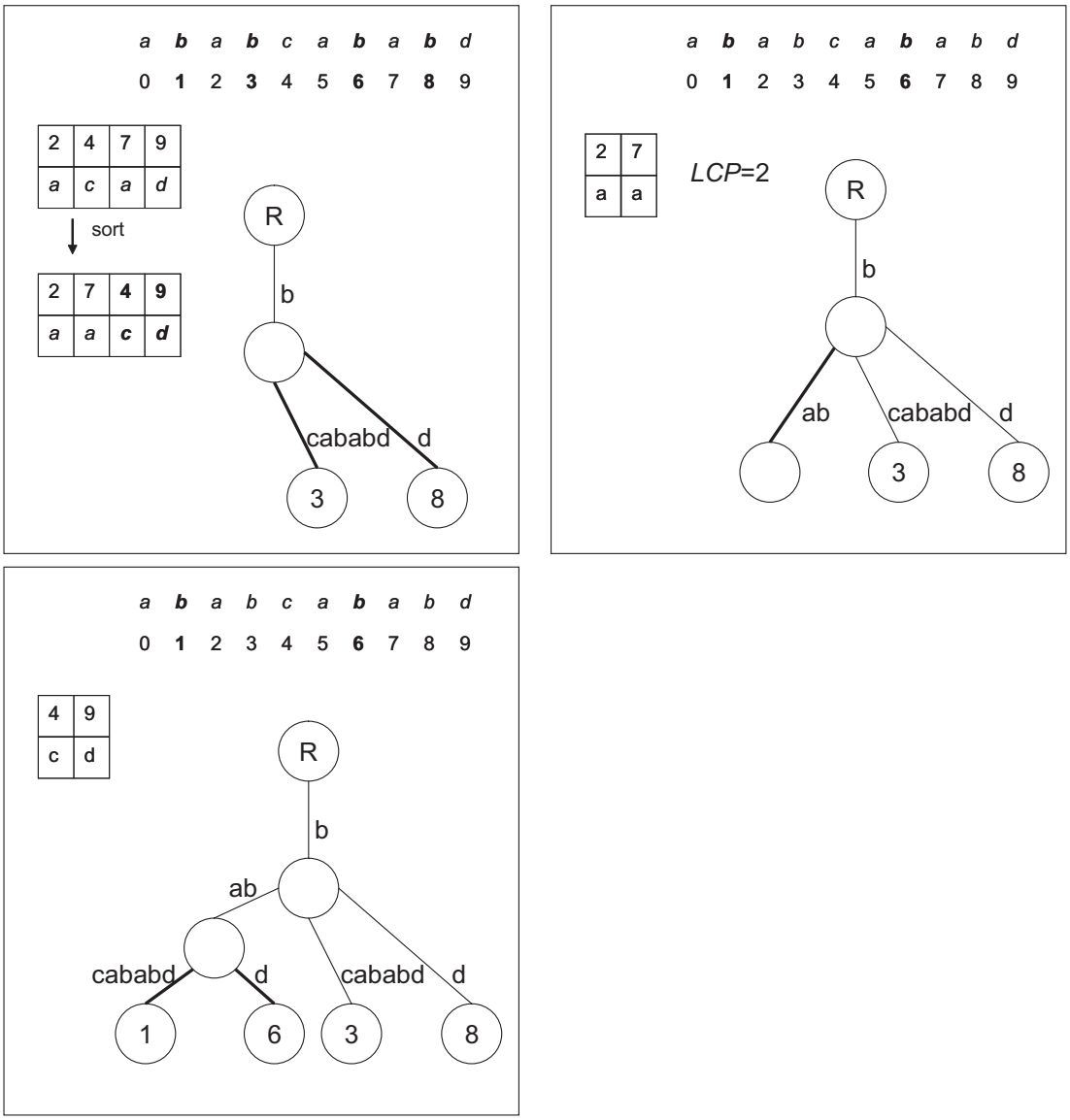

Figure 3.11: The steps of the *TDD* algorithm [64] for building the sub-tree for prefix b and input string $X = ababcababd$.

As before, the algorithm performs massive non-sequential accesses to the input string when it does the character-by-character comparisons starting at different arbitrary positions. The input string for the *TDD* algorithm cannot be larger than the main memory.

Another problem of *TDD* is the suffix tree on-disk layout. The trees for different partitions are of different sizes, and some of them can be significantly bigger than the main memory. This may pose problems when loading the subtree into main memory for querying. If the entire subtree cannot be loaded into and traversed in the main memory, the depth first traversal of such a tree requires multiple random accesses to different levels of on-disk nodes.

3.1.5 The partition-and-merge strategy of *Trellis*

The oversized subtrees caused by data skew can be eliminated by using a set of different-length prefixes, as shown in [55]. In practice, the initial prefix size is chosen so that the total number of prefixes P will allow to process each of the P sub-trees in main memory. For example, we can hold in our main memory in total T_{max} suffix tree nodes. The counts in each group of suffixes sharing the same prefix are computed by a sequential scan of input string X. If a count exceeds T_{max}, then we re-scan the input string from the beginning collecting counters for an increased prefix length. Based on the final counts, none of which exceeds T_{max}, the suffixes are combined into approximately even-sized groups. As an example consider the case when suffixes starting with prefix *ab* occur twice more often than the suffixes starting with *ba* and *bb*. We can combine suffixes in partitions *ba* and *bb* into a single partition *b* with approximately the same number of suffixes as contained in partition *ab*. The maximum number of suffixes in each prefix partition is chosen to ensure that the size of the tree for suffixes that share the same prefix will never exceed the size of the

```
Trellis MAIN (INPUT: string X)
1       partition X into K substrings
2       for each substring X_i
3               build suffix tree ST_X_i
4               for each prefix PR in collection of variable-length prefixes
5                       find sub-tree starting with PR
6                       OUTPUT this sub-tree
                                into a separate file on-disk

7       for each prefix PR in collection of variable-length prefixes
8               READ from disk all K sub-trees starting with PR
9               merge sub-trees into 1 sub-tree for prefix PR
10              OUTPUT this sub-tree back to disk
```

Figure 3.12: Pseudocode for the *Trellis* algorithm. Note KP iterations (including disk I/Os) in the nested loop in lines 7,8.

main memory. This is done to ensure that each such subtree can be built and queried in main memory.

Based on this new partitioning scheme, Phoophakdee and Zaki [55] proposed another method for creating suffix trees on disk – the *Trellis*[2] algorithm. The main innovative idea of this method is the combination of prefix partitioning and horizontal partitioning of the input into consecutive substrings, or *chunks*. In theory, substring partitioning does not work for any input, since the suffixes in each substring partition do not run till the end of the entire input string. However, this horizontal partitioning works for many practical inputs. Consider for example the Human genome sequence of the size of about 3 GB in length. In fact, there is not a single string representing Human genome, but rather 23 sequences of DNA in 23 different Human chromosomes, with the largest sequence being only about 247 MB in size. Those chromosome sequences represent natural partitions of the entire genome.

[2] *Trellis* stands for **E**xternal **S**uffix **TR**ee with Suffix **L**inks for **I**ndexing Genome-Sca**L**e Sequences.

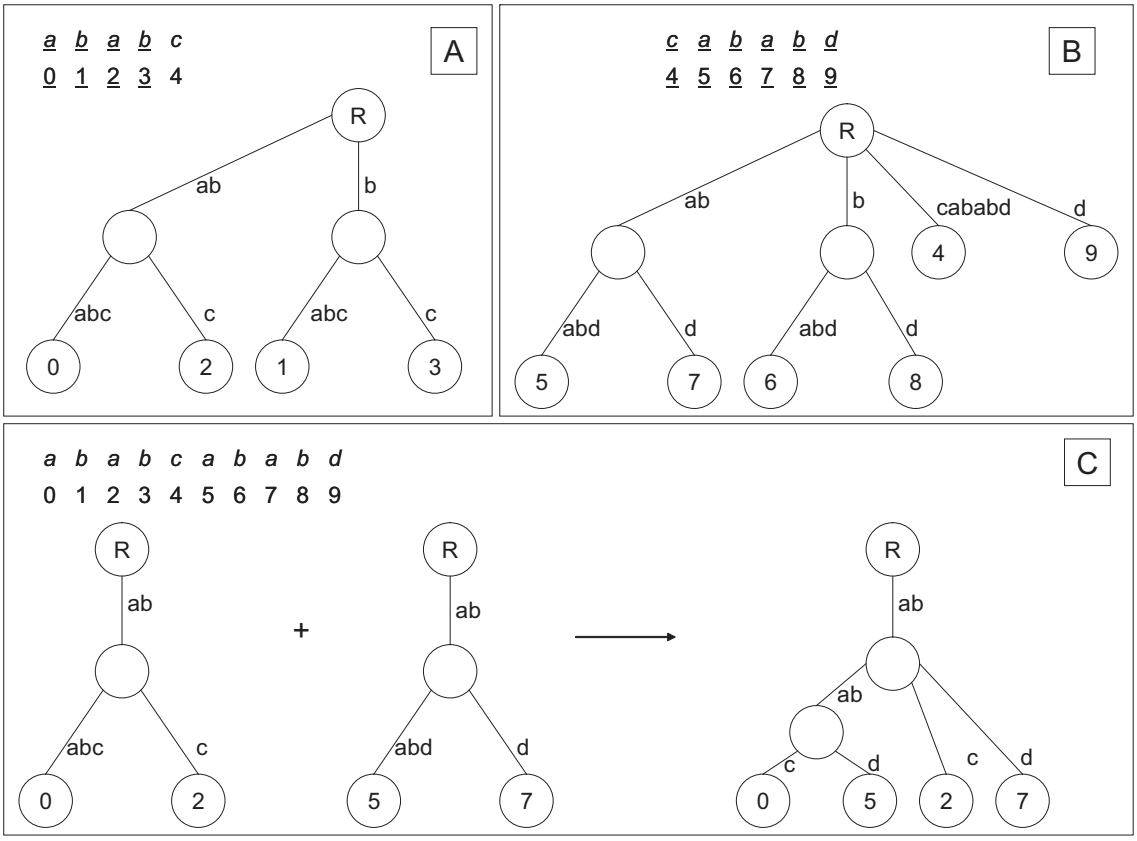

Figure 3.13: The steps of the *Trellis* algorithm applied to input string $X = ababcababd$. **A**. Building the suffix tree for substring $X_1 = abab(c)$. **B**. Building the suffix tree for substring $X_2 = cababd$. **C**. Merging the sub-trees for prefix ab. The total size of the tree structures at each step allows to perform each step in main memory.

If the size of each natural chunk of the input does not allow us to build its suffix tree entirely in main-memory, the chunk is split into several slightly overlapping substrings. A small "tail" – the prefix of the next partition – is appended to the end of each such substring except the last one. In practice, for real-life DNA sequences, the length of such a tail was set at $50,000$ characters for *Trellis*. This was intended to resolve the comparison of suffixes that start near the end of a partition.

After partitioning the input into chunks of appropriate size, *Trellis* builds an independent suffix tree for each chunk. It does not output the entire suffix tree to disk, but rather writes to disk the different sub-trees of the in-memory tree. These sub-trees correspond to the different variable-length prefixes. Once trees for each chunk are built and written to disk, *Trellis* loads into memory the subtrees for all the chunks that share the same prefix. Then it merges these subtrees into the shared-prefix-based subtree for an entire input string. The pseudocode for the *Trellis* algorithm is shown in Figure 3.12.

As an example, let us apply the *Trellis* method to our input string $X = ababcababd$. Let the collection of prefixes for a prefix-based partitioning be $\{ab, ba, c, d\}$. Next, we partition X into two substrings $X_1 = abab$ with "tail" c, and $X_2 = cababd$. Note the overlapping symbol c which is used as a sentinel for suffixes of X_1. We build in memory the suffix tree for X_1, which is shown in Figure 3.13 [A], and we output it to disk in the form of two different subtrees: one for prefix ab and the second for prefix ba. The same procedure is performed for X_2 (Figure 3.13 [B]). Then, we load into main memory the subtrees for, say, prefix ab and we merge those sub-trees into the common ab-subtree for the entire X.

The merge of subtrees for different chunks is performed by a straightforward character-by-character comparison, which leads to the same $O(N^2)$ worst-case internal time as the brute force algorithms described above.

Trellis was shown to perform at speed comparable to *TDD*. Further, *Trellis* almost never fails due to insufficient main memory. This failure may occur because either the trees for particular chunk or the subtrees for a particular prefix are larger than the main memory, which still accidentally happens with the *Trellis* implementation.

If we have K chunks and P prefixes in the variable-length prefixes collection, the number of random disk accesses is $O(KP)$. Since both K and P depend on the length of the input string N, the execution time of *Trellis* grows quadratically with the increase of N, and is therefore not scalable for larger inputs.

During the character-by-character comparison in the merge step, the input string is arbitrarily accessed at different positions all over the input string. Therefore, the scalability of *Trellis* does not go beyond the size of the main memory designated for the input.

Method name	Underlying main memory algorithm	Asymptotic internal running time	Max. number of random disk I/Os	Sample construction time[5] for ≈ 50 MB of Human chromosome I	Scalability: (the largest input handled with 2 GB of RAM) [6]
TOP-Q [6]	Suffix tree construction by Ukkonen [65]	N	N	7 h [55]	≈ 50 MB: 4 hours [6]
DPST [11]	Suffix tree on words [2] for each partition	$N \times P$	P	Not reported	≈ 187 MB: time not reported
Hunt et al. [28]	Brute force for each partition	$N^2 \times P$	P	12 min [64]	≈ 256 MB: 13 h [28]
TDD [64]	*Wotdeager* [24] for each partition	$N^2 \times P$	P	2 min [64]	≈ 2 GB: 2 h [64]
Trellis [55]	Ukkonen [65] for each partition, brute force for merge	$N + (\frac{N}{P})^2$	$K + KP$	2 min [55]	≈ 3 GB: 4 h [55]

Figure 3.14: The key features of the practical algorithms for the external memory suffix tree construction. Here, N denotes the total input size, P the number of partitions by common prefixes, K the number of substring partitions.

3.1.6 Summary of the existing algorithms

The main features of the practical external memory algorithms for suffix tree construction including some performance and scalability benchmarks are summarized in the table of Figure 3.14. The fastest and the most scalable algorithms [55, 64] are able to build the suffix tree for a largest string fitting into main memory in a matter

[5]These sample construction performances give an order of magnitude but are not directly comparable since they were obtained using different machines (2 GB of main memory each.)

[6]The time for *TDD* was obtained by using a DNA encoding with 4 bits per character, since 3 GB of DNA do not fit the main memory of 2 GB [64]. *Trellis* uses a 2-bits-per-DNA-character encoding and do not include the non-DNA characters such as "N" which stands for the undefined DNA symbol. *DiGeST* maps the new positions after "slicing out" the unknown characters to the original positions in the raw sequence. Note that if we index all the characters which appear in genomic sequences, we will face the problem of invalid common substrings of significant length which consist of long stretches of "N"s and do not represent actual common substrings.

of several hours on a single machine.

However, when trying to create a suffix tree for several genomes we faced the problem of increased running time of the presented algorithms such as *TDD* and *Trellis*, even though the entire input string was completely contained in main memory. This occurs due to the fact that all the above algorithms, while partitioning the output suffix trees by common prefixes, do repeated scanning of disk-based structures as many times as the number of prefixes, which in turn depends on N.

This shows the need for designing a new suffix tree construction algorithm, described in the following section.

3.2 Our new algorithm: *DiGeST*

Next we present our new method for disk-friendly suffix tree construction. Our new algorithm *DiGeST*[3] performs at a speed comparable with that of *TDD* and *Trellis*, and scales for larger inputs. While also *DiGeST* requires the input string to fit in the main memory, from an external memory point of view, the algorithm is very efficient: it performs only two scans over the disk data and furthermore accesses the disk mainly sequentially. From an internal (main memory) running time point of view, this algorithm still belongs to the group of brute-force algorithms with an asymptotically quadratic running time. Recall, however, that on average and for real-life inputs, the running time for brute-force suffix insertion has been found to be $O(N \log N)$ [3], which was confirmed by our experiments with genomic sequences.

3.2.1 Design principles of a new I/O-efficient construction

In the following we describe some techniques that we used in the design of our disk-friendly suffix tree construction algorithm.

Lexicographic intervals

Considering the sequential disk I/Os as a main goal of a new algorithm, we observe the following. The locality of references during the suffix tree construction can be significantly improved if we insert sorted suffixes into the suffix tree. In this case, if we collect and write to disk the suffix tree for all suffixes in a given lexicographic interval, then we do not access the completed part of the tree anymore. Thus, the first step is to obtain an array of lexicographically sorted suffixes, a *suffix array* (formally defined in Section 2.1).

[3] *DiGeST* stands for **Di**sk-based **Ge**neralized **S**uffix **T**ree. We published this work in [4]

Inserting sorted suffixes into a suffix tree

Once we obtain an array of lexicographically sorted suffixes we can insert in order each suffix into a growing suffix tree. This process of converting a suffix array with LCP information into a suffix tree exhibits good locality of references since the suffix array and the suffix tree for a given lexicographic interval are accessed sequentially. In order to insert the next suffix we create one internal node at depth of $|LCP|$ characters from the root of the suffix tree, and add a new leaf for this newly inserted suffix. This process is performed in main memory. Once the suffix tree for a given lexicographic interval is complete, it is written to disk and never accessed again.

Multi-way merge sort

In order to sort suffixes, we have chosen to use the well-known paradigm of the two-phase multi-way merge-sort [21] as the basis for our suffix sorting.

Multi-way merge sort works as follows. It partitions an input array of values into K sub-arrays. The size of each sub-array is bounded by the available main memory. In the first phase, the algorithm sorts the elements of each sub-array (using any efficient main-memory sorting algorithm). The sorted sub-arrays, called *runs*, are written to disk. The main idea of multi-way merging is to merge the sorted lists from multiple runs at the same time. The algorithm allocates K input buffers, one input buffer for each active (unfinished) run, and one output buffer. Each buffer has a pointer to the next element of the corresponding run. Then, all the currently pointed values of each run are compared, and the smallest (belonging to a buffer i, say) is transferred to an output buffer. The pointer in buffer i is advanced by 1. If buffer i is now exhausted of elements, we read the next blocks from the corresponding run. Once no data remains in that run, it is considered no longer active. When the output buffer is full, it is written to the end of the output file. When only one active run remains, the

algorithm finishes up by copying all the remaining elements to the end of the output file.

We have used a similar merge-sort approach in our algorithm. Note that unlike the conventional merge sort applied to numbers or short strings, we are sorting suffixes, which are long overlapping strings. This in general requires special sorting and merging techniques developed for suffix sorting. However, we have used in our suffix merge step the comparison of suffixes from different partitions, trading the efficiency of suffix sorting for a better patterns of random disk I/Os.

Prefix buffering

Since the input is accessed in an arbitrary manner, and the random access to large arrays even in main memory is expensive (see Section 2.3.2), we can buffer a small prefix of each suffix next to its start position in the suffix array. Now, having these characters close to the place of actual comparison we first compare the characters in these small buffers. Only if all of them are identical we consult the input string. This way the vast majority of the comparisons is resolved without random access to the input string. These buffered prefixes significantly improve the performance of the merging phase.

Combining these four techniques we designed and implemented a simple and fast algorithm, described in the following section.

3.2.2 The details of our algorithm

Our method for building a suffix tree on disk consists of the following main components (see Figure 3.15):

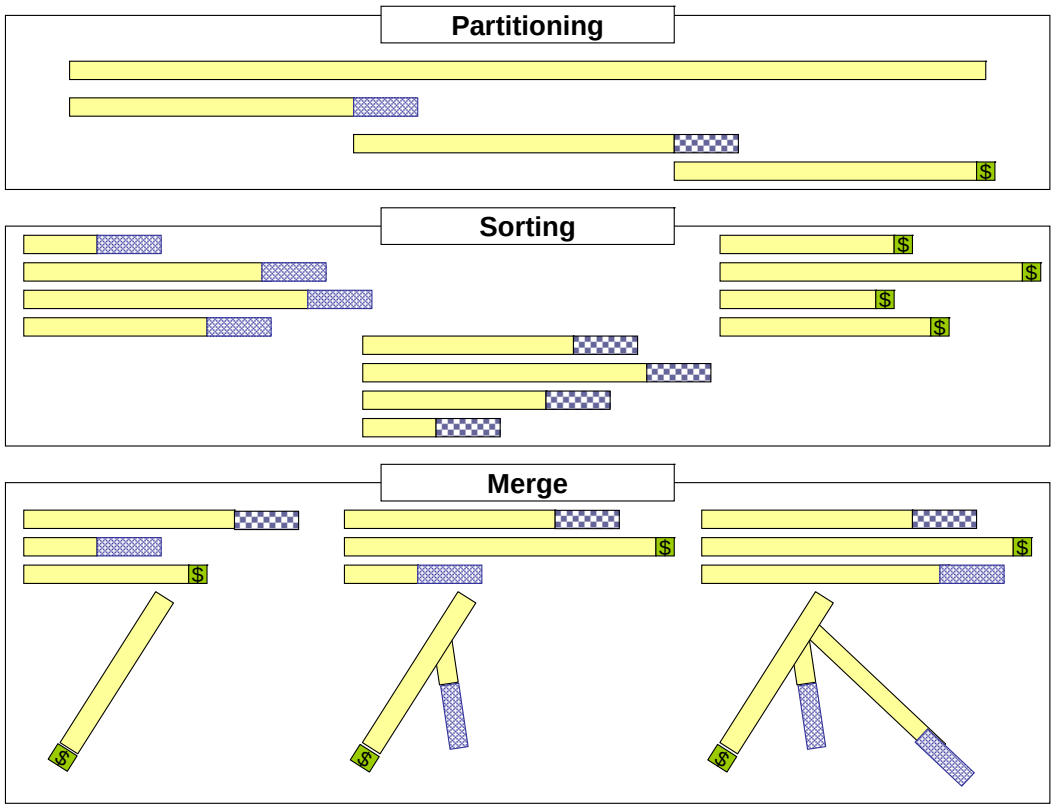

Figure 3.15: Graphical representation of *DiGeST*. Texture patterns at the end of each suffix correspond to tails, added to preserve the correct lexicographical order of the suffixes relative to the entire input string. A detailed description of each step is presented in Sections 3.2.2, 3.2.2, and 3.2.2 respectively.

1. **Preprocessing.** In this step we check the set of the input strings. If any of the strings in this set cannot be processed into a suffix array entirely in main memory, we create from this string several input files of approximately the same size. Note that most genomic inputs are naturally partitioned into chromosomes or separate genomes of a moderate size. At the end of this phase we have on disk K input partitions of such a size that each of them can be processed into its suffix array in main memory. In this step we also encode the input string (DNA) as a sequence of bits with two bits per character. we do this in order to fit a larger input into main memory, since our algorithm requires random access to the input strings.

2. **Sorting suffixes.** We sort suffixes in each partition using Larsson's in-place suffix sorting algorithm [44]. The output of this algorithm is an array where only start positions of sorted suffixes are stored. In addition, we also attach to each suffix start position a short prefix of this suffix[4]. At the end of this phase, for each partition, we obtain on disk a suffix array with attached prefixes.

3. **Merging prefix-attached suffix arrays.** Consecutive pieces of each of the K suffix arrays are read from the disk into input buffers. A "competition" is run among the top elements of each buffer and the "winning" (lexicographically smallest) suffix migrates to an output buffer organized as a suffix tree. When the output buffer is full we empty it to disk.

These main steps are summarizes in a high-level pseudocode of Figure 3.17. We describe next our new method in more detail.

[4]These prefixes significantly improve the performance of the merging phase.

Step 1. Preprocessing of the input

The size of the partitions is determined based on the amount of available memory. To build a suffix array for each partition using Larsson's algorithm [44], we need $8\times$ *partition size* bytes of RAM. In addition, we need an output buffer to collect the suffix positions along with 64-bit long prefixes attached. We found that we can quickly process partitions of size 100–250 MB using 2 GB of RAM. Note that, even though we could fit more into the available memory, Larsson's algorithm involves random accesses to the input string, and thus, its performance degrades for bigger partition sizes due to cache misses. After choosing the size of the partitions, we determine their number K.

In general, if one of the natural partitions (input strings) is too large, it is partitioned into several artificial partitions, such that each partition has length at most p. If we need to partition an oversized input string of size N_1 into k_1 consecutive substrings of size N_1/k_1, then we add a "tail" to each such partition (except the last). Let X_u and X_{u+1} be two consecutive partitions of some input string X_1. We attach a prefix t of X_{u+1} as a tail to X_u. Prefix t is the shortest prefix of X_{u+1} such that it does not occur anywhere as a substring of X_u. This "tail" serves as a sentinel for the suffixes of partition X_u, and its positions are not included into the suffix array for X_u. The uniqueness of the "tail" ensures that each suffix S_i of a partition X_u $(0 \le i < |X_u|, 0 \le u < K)$ will receive the same sorting rank as the entire suffix S_{up+i} of X (p is the size of the largest partition allowed by the algorithm).

Consider the following example in Figure 3.16. It shows the partitioning of input string $X = ababaaabbabbbabaabab$ into four partitions. The main memory can hold a suffix array for not more than $N = 8$. To facilitate the example, we represent our input in a binary alphabet (a stands for 0-bit, b stands for 1-bit). In this illustration we also refer to the partition numbers as A, B, C, and D in order to distinguish them

Partition A					Partition B					Partition C					Partition D				
a	b	a	b	a	a	a	b	b	a	b	b	b	a	b	a	a	b	a	b
1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5

Figure 3.16: Partitioning of an oversized input string $X = ababaaabbabbbabaabab$ into its four partitions. The tail of partition B is substring *bbb*, which serves as a sentinel for suffixes of $X_B = aabba$. The combined size of each partitions with its suffix array must be less than the size of main memory M.

from the numbers representing character positions. Note that the tail of partition X_B is substring *bbb*, which never occurs inside $X_B = aabba$. If we take partition B without this tail, then suffix $SB_5 = a <_{lex} SB_1 = aabba$, while in fact if the suffix starting at position 5 of B would be taken till the end of X, then $S_1 = aabba \ldots <_{lex} S_5 = abbb \ldots$.

We next prove that attaching such a tail t is necessary and sufficient to ensure that all suffixes in the partition are in correct order relative to the entire input string.

Proposition 1. *Let*

1. *Y be a partition of the input string X*
2. *t be the tail appended to Y (not a substring of Y)*
3. *SY_i, SY_j be two suffixes of Y starting at (global) positions i and j, respectively, (and running till the end of Y),*
4. *S_i, S_j be the suffixes of X starting at positions i and j, respectively, (and running till the end of X).*

Then, for the concatenation $SY_i \cdot t$ and $SY_j \cdot t$: $SY_i \cdot t <_{lex} SY_j \cdot t$ if and only if $S_i <_{lex} S_j$.

Proof.

Without loss of generality suppose that $i < j$.

Only if. We want to show that if $S_i <_{lex} S_j$ then $SY_i \cdot t <_{lex} SY_j \cdot t$.

Since $i < j$, $SY_i \cdot t \neq_{lex} SY_j \cdot t$ (as both are of different lengths). Suppose that $S_i <_{lex} S_j$ but $SY_i \cdot t >_{lex} SY_j \cdot t$. Since the tails are equal, it must be that $SY_i[i+|LCP|_{ij}+1] > SY_j[i+|LCP|_{ij}+1]$. But in this case $S_i >_{lex} S_j$ (contradiction).

If. Since t is not a substring of Y, $SY_j \cdot t$ cannot be a prefix of $SY_i \cdot t$.

As such, let c_{i+k} and c_{j+k} be the first characters in $SY_i \cdot t$ and $SY_j \cdot t$, respectively, where $SY_i \cdot t$ and $SY_j \cdot t$ differ. Then, if $c_{i+k} <_{lex} c_{j+k}$ then $S_i <_{lex} S_j$.

And if $c_{i+k} >_{lex} c_{j+k}$ then $S_i >_{lex} S_j$. □

Note that if a tail t cannot be found, we cannot guarantee a correct sorting of suffixes in each partition. However, in practice, we have not yet encountered such a case.

In practice, for real-life DNA sequences, the length of such a tail is negligibly small compared to the size of the partition itself (it never exceeded 1000 characters in our experiments with DNA databases).

In most cases, the sequenced genomes are already partitioned into natural partitions - the chromosomes. The size of the largest Human chromosome, chromosome I, is just 247 MB. The input strings which do not exceed 250 MB are left in their original separate files.

Step 2. Sorting suffixes

The sorting of suffixes in each partition is performed in main memory using Larsson's *quicksufsort* algorithm [44] and its implementation from [43].

Larsson improves the practical behavior of the Manber-Myers algorithm [45] with $O(N \log N)$ time complexity by avoiding the scanning of the entire array in each of the algorithm's $logN$ passes and using a ternary-split Quicksort as a sorting subroutine. Our choice in using this algorithm for sorting suffixes was influenced by

the experimental results of [57]. Note that one can use any efficient suffix sorting algorithm for main memory (cf. [47, 38, 30, 34, 27]) in the first phase, such as the algorithm by Manzini and Ferragina [47], the fastest in practice but with asymptotic complexity $O(N^2 logN)$. The running time of *DiGeST* is dominated by disk I/Os, so the choice of the in-memory sub-routine can be arbitrary as long as it guarantees a good performance.

Once the suffixes are sorted, we write their start positions to disk. For each partition, we have a list of suffix start positions. The order of these positions is according to the lexicographical order of their corresponding suffixes. Further, next to each start position, we store the 32 character (64 bits) prefix of the suffix in the form of two four-byte numbers. This is possible because the alphabet of DNA has only four letters, and thus, each letter can be compactly represented by two bits. The same encoding was used by *Trellis+*, the program we compare our results to in Section 3.2.3.

Step 3. Merging prefix-attached suffix arrays

Merging suffix arrays into a suffix tree incurs multiple random accesses to the input string. In order to increase the amount of the input that fit the available main memory we encode each distinct letter of the meaningful DNA alphabet $\{a, c, g, t\}$ as a 2-bit string[5]. Thus, our alphabet is reduced to $\Sigma = \{0, 1\}$. Note that a string over any alphabet can always be reduced to the binary alphabet by representing each character as a sequence of bits and then concatenating these binary sequences.

[5]We note that in real DNA sequences there are unidentified (or "unknown") characters denoted by n, which does not belong to the $\{a, c, g, t\}$ alphabet. For example, Human chromosome 22 contains a large block of of such symbols at its beginning. We discard these characters from our binary encoded input. *Trellis* also discards such characters from the input. We remark that this does not create a problem when using suffix trees. For this, one can record the cut positions and easily map the characters in the transformed string to characters in the original string.

We denote by $N_b = 2N$ the length of the entire DNA input encoded in binary as above.

Our merge works as follows. We use one input buffer for each of the K sorted arrays of suffixes (*runs*). The input buffers are loaded with suffixes from the corresponding runs. Then a competition is run among the top elements of each input buffer. The winning element migrates to an output buffer organized as a suffix tree.

For the competition we use a priority queue implemented as a heap of size K (total number of input partitions). In order to determine the relative order of the suffixes from different runs, we first compare their attached prefixes, stored as two 4-byte integers. Only if both of them are equal, we access the input string at the corresponding positions. We found that this happens in practice only for a very small fraction of the suffixes, approximately 2.5% in real DNA sequences and never happened for synthetic[6] DNA sequences of length 3 GB. Without storing such prefixes, different suffix comparisons would cause multiple random accesses to the input string, which is not desirable due to excessive cache misses.

The smallest element removed from the heap is added to a growing suffix tree in the output buffer. We determine the length of the longest common prefix LCP of a new suffix with the previously added suffix that is lexicographically smaller. And again, the prefixes attached to the elements of the heap help us determine the LCP of these two suffixes without accessing the input string in the vast majority of the cases. Once we have the information about the next suffix, say S_2 and its $|LCP|$ with S_1, we can create a corresponding internal node and a leaf for suffix S_2.

In the following we explain the pseudocode for our merge algorithm presented in Figure 3.17.

1. Create K input buffers – line 7.

[6]Sequences generated by a pseudorandom drawing of symbols from DNA alphabet

```
DiGeST MAIN (INPUT: string X)
1       partition X into K substrings
2       for each partition X_i
3               build suffix array SA_X_i
4               OUTPUT SA_X_i to disk
5       call Merge( )

Merge( INPUT: SA_X_i, 0< i<K as files on disk )
7       allocate K input buffers and 1 output buffer
8       for each input buffer
9               load part of SA_X_i into input buffer
10      create heap of size K
11      read first element of each input buffer into a heap

12      while heap is not empty
13              transfer the smallest suffix of substring j
                        from the top of heap into output buffer
14              find LCP of this suffix with the last inserted suffix
15              create a new internal node
                        and a new leaf in the output suffix tree
16              if output buffer is full
17                      OUTPUT suffix tree to disk

18              if input buffer j is empty
19                      if not end of SA_X_i
20                              fill input buffer j with the next
                                                        suffixes
21              if input buffer j is not empty
22                      insert next suffix from input buffer j into heap
```

Figure 3.17: Pseudocode for the *DiGeST* algorithm. The main-memory operations are performed in the constant working space. The interactions with disk are sequential I/Os – lines 9, 17.

2. Fill these K buffers with suffixes from the corresponding K sorted suffix arrays – lines 8,9.

3. Initialize a heap H with the first suffixes of each of the K input buffers – lines 10,11.

4. Remove the top element (the lexicographically smallest suffix) from H. Let this top element belong to partition X_i – line 13.

5. Initialize the suffix tree of the output buffer by creating one leaf node for this first suffix. Store this suffix and its 64-bit prefix in a variable *lastAdded*.

6. Insert into H the next suffix from the buffer corresponding to P_i. If the end of buffer is reached, then upload suffixes from the corresponding on-disk run – lines 19–20.

7. Remove top element *curr* from H – line 13.

8. Find $|LCP|$ between *curr* and *lastAdded* – line 14. Further record the bit of *curr* in position $curr + LCP + 1$. Call this bit *firstBitAfterLCP*.

9. Find a corresponding edge at depth LCP from a root of the suffix tree. If there was no node at depth $|LCP|$, then split the edge reached, creating a new internal node ν. Create a new leaf child of ν representing the rest of suffix *curr*. If there was a node, add a new child which starts from *firstBitAfterLCP*.(line 15).

10. If the output buffer is full, then

    (a) flush it to disk (lines 16,17),

    (b) store the 64-bit prefix of *curr* and a reference to the flushed tree in array of *dividers*,

(c) go to Step 4.

Otherwise, go to Step 6 – lines 21,22.

The suffix tree in the output buffer

Our growing suffix tree is created from a merged array of sorted suffixes and the LCP information between adjacent suffixes. First, we explain how to find efficiently the point of edge split while adding a new suffix to the current suffix tree in the output buffer.

We call the path of the suffix tree corresponding to the lexicographically largest suffix the *boundary path*.

In the next proposition we show that, while adding new nodes to the current suffix tree, the edge to split for each next suffix cannot be found anywhere except on the boundary path of the tree built so far.

Proposition 2. *Let*

1. *T be the suffix tree currently being built in the output buffer,*
2. *S_1 be the suffix last added to T,*
3. *S_2 be the suffix to be added next into T in Step 9.*

Then, split point ν (the parent node for leaf S_2) lies on the boundary path of T.

Proof. There does not exist a suffix S_3 such that $X[S_1, j] <_{lex} X[S_3, j] <_{lex} X[S_2, j]$ for any $1 \leq j \leq N$. This is true because otherwise S_3 would be the next suffix to be inserted after S_1.

Since S_1 corresponds to the (lexicographically) greatest suffix previously added to T, the boundary path of T corresponds to $X[S_1, N]$, and thus, covers all the prefixes of S_1 including substring $X[S_1, S_1+LCP]$. Hence, in order to locate substring

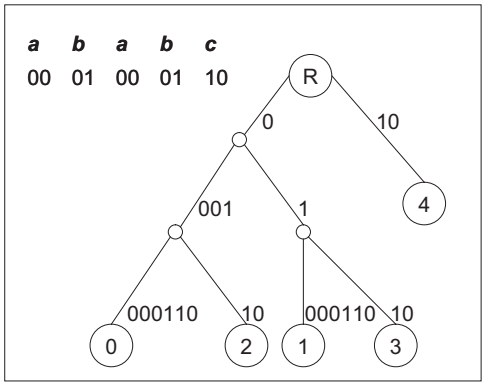

Figure 3.18: Suffix tree over binary alphabet for $X = ababc$ with edges labeled for clarity with the corresponding bit sequences.

$X[S_1, S_1 + LCP] =_{lex} X[S_2, S_2 + LCP]$, we need to follow the boundary path of T. Therefore split point ν lies on the boundary path. □

We consider each suffix to be inserted into the tree as a sequence of bits. Note that every string over any alphabet Σ can be reduced to the binary alphabet by representing each character as a sequence of $b = log|\Sigma|$ bits and then concatenating these binary sequences. Therefore, we build a suffix tree over a binary alphabet. Note that, since in each step we add one suffix to the tree, treating the suffix as a sequence of bits does not increase the total number of leaves in the suffix tree: the tree has one leaf node and one internal node for each inserted suffix. All that changes is the length of the edge labels. The tree over the binary alphabet is illustrated in Figure 3.18 which shows the suffix tree for $X_B = 0001000110$ which is an binary representation of $X = ababc$.

Any internal node in this suffix tree has exactly two children. This allows using two child pointers only (per node) and representing the entire suffix tree as an array of the constant-sized nodes.

The next proposition shows how we make use of the binary alphabet during Step 9 of the above merge procedure, without additional random access to the input string.

Each internal suffix tree node in our implementation has exactly two children, namely a 0-child and a 1-child.

Proposition 3. *Let*

1. *T be a suffix tree of lexicographically sorted suffixes currently being built in the output buffer,*
2. *S_1 be the suffix last added to T,*
3. *S_2 be the suffix to be added next into T in Step 9.*

Then the insertion of S_2 splits an existing edge such that it creates the 1*-child leading to the leaf S_2.*

Proof. The suffix starting at S_2 has a common prefix of length LCP with the suffix starting at S_1. Then the character $X[S_2 + LCP + 1]$ is greater than $X[S_1 + LCP + 1]$. Since our alphabet is binary, we have $X[S_2 + LCP + 1] = 1$. □

The only case a new 0-child can lead to the leaf corresponding to suffix S_2 is when $S_1 + LCP = N_b$. That is, suffix S_1 is a prefix of suffix S_2 and the leaf corresponding to S_2 will be a child of the node corresponding to S_1, which is transformed from a leaf to an internal node.

Note that, if using $\{a, c, g, t\}$ as alphabet instead of $\{0, 1\}$, then it is not possible to build the suffix tree as described above without incurring random accesses to input string S. This is because when splitting an edge for adding a leaf corresponding to S_2, we would have to check the character $X[S_1 + LCP + 1]$.

Algorithmically, the actual place of a new internal node is found as outlined in [49], page 162.

Specifically, the suffixes are added to the suffix tree in lexicographic order, i.e., as in Figure 3.19, the leaves are inserted from left to right. Thus a new leaf for suffix

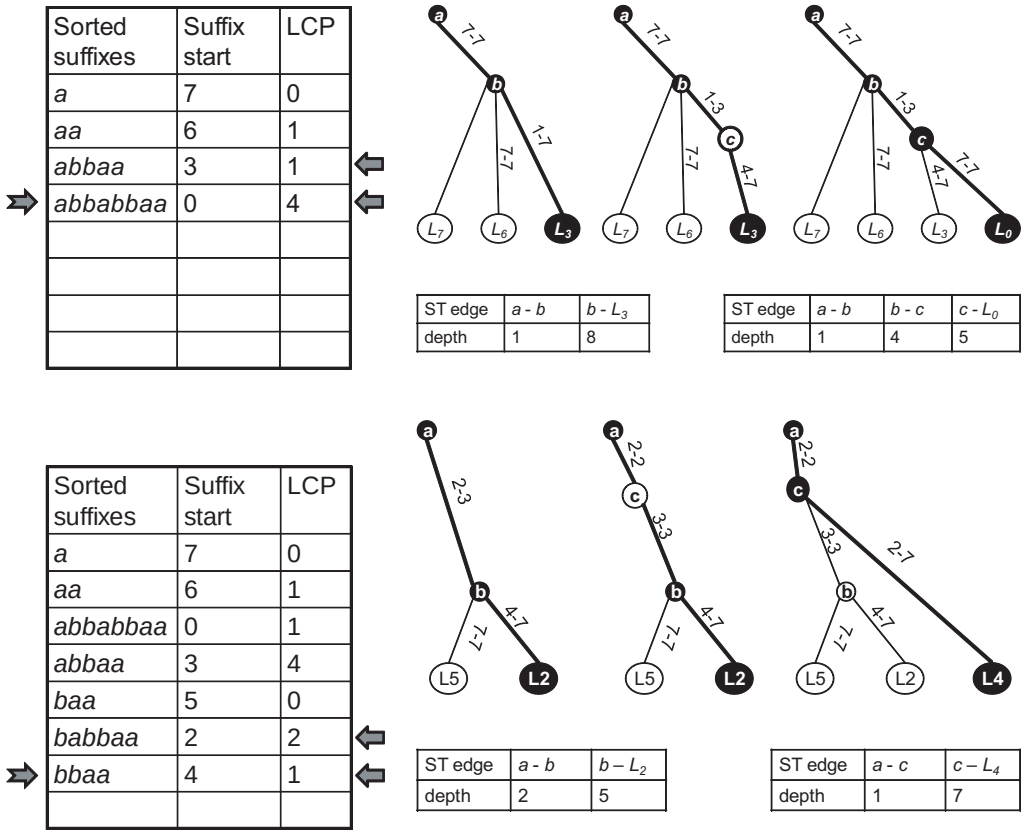

Figure 3.19: Two examples of adding new suffixes to the growing suffix tree for input string *abbabbaa*. [**Top**] Adding the suffix starting at position 0. In this case, the LCP of this suffix with the previous suffix, starting at position 3, is larger than the LCP between the latter and the suffix starting at position 6. The last edge bL_3 on the boundary path (abL_3) splits, and new internal node c and leaf L_0 are created. [**Bottom**] Adding a suffix starting at position 4. Here, the LCP of this suffix with the previous suffix, starting at position 2, is smaller than the LCP between the latter and the suffix starting at position 5. In this case, we first need to locate the corresponding edge ab on the boundary path abL_2 (for example using binary search). Then edge ab splits and new internal node c and leaf L_4 are created. The sorted suffixes are shown in the tables left of the trees. The boundary path arrays are shown below the trees.

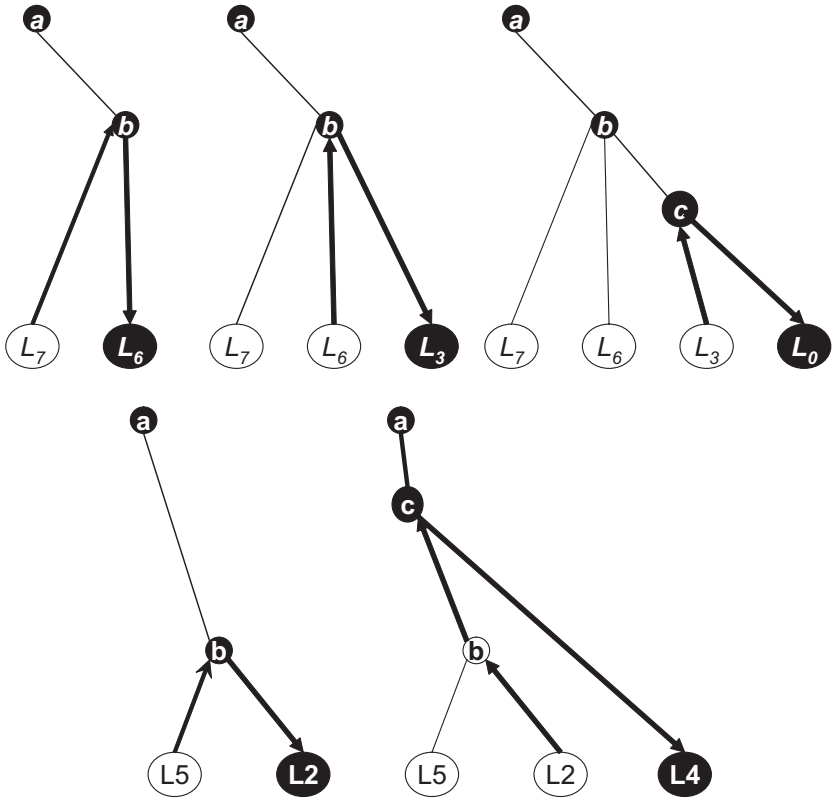

Figure 3.20: This figure repeats an example of Figure 3.19, but presents it as an Euler tour. Note that each node is accessed not more than twice – once from above and once from below – thus giving an efficient linear-time suffix tree construction from a suffix array with LCP.

S_2 always becomes a rightmost child of some node ν on the border path of the suffix tree. Furthermore, the $|LCP|$ tells the depth of ν. The depth here is the number of characters from the root to a node. All the nodes of the border path are kept in a stack with the leaf on top. For a new leaf, nodes are popped from the stack until the edge at the corresponding depth is reached. If needed, a new node ν is created at depth $|LCP|$ from the root by splitting this found edge. The new node and the new leaf are pushed to the stack for the subsequent processing. Two examples of adding a new leaf are shown in Figure 3.19.

By inserting in this order we simulate a traversal of the suffix tree, known as an

Euler tour (see Figure 3.20). Since such a traversal never visits the same node of the suffix tree more than twice, the entire conversion of the merged suffix array with LCP into a suffix tree runs in time $O(N)$. Thus, we can easily convert the suffix array with LCP into a corresponding suffix tree.

Since each internal node has exactly 2 children, each suffix tree node occupies a constant space. Due to this fact, the output buffer is designed as an array of tree nodes. Each tree node is a structure containing: the positions (referring to the same array) of the left and right children of the current node, and the start position i and the end position j of the substring $X[i, j]$ which labels the incoming edge of this node. Since we are working with inputs whose size exceeds what can be stored in a 4-byte integer, we have to represent the pair of positions inside the input as a combination of the partition number and the offset inside this partition. We use 1 integer for the partition id, 2 integers (8 bytes) for the children's positions and 2 integers (8 bytes) for the substring $X[i, j]$'s start and end position inside the partition. The total size of each node is therefore 20 bytes, and having exactly $2N$ nodes the total tree size is $40N$ bytes. Thus, the size of our trees is 40 times larger than the input it is built upon.

Figure 3.21 represents the insertion of the two first suffixes into a tree for a binary string $X_b = 0001000110$ which represents $X = ababc$. For clarity, the array elements are aligned as tree nodes and the corresponding suffix of an incoming edge for each node is explicitly shown. Note that only suffixes that start at even positions of X_b are inserted into a tree, since they represent the actual suffixes of X.

The construction of the tree in this example proceeds as follows. The lexicographically smallest suffix S_0 (0001000110) is the first to be inserted into the tree. It is the left child of the root. We create a root node and set its left child value to 1 which is the next available slot in the output buffer array. The incoming edge of the

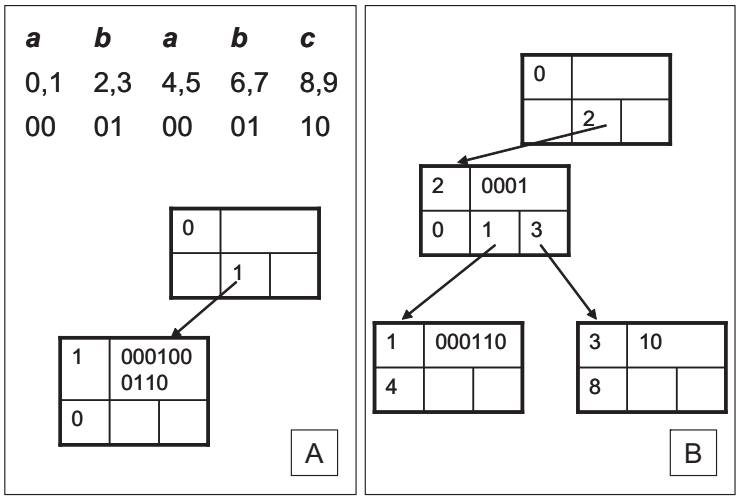

Figure 3.21: Suffix tree as an array of nodes. The two first steps of building the suffix tree for $X_B = 0001000110$, which represents $X = ababc$, in output buffer. **A**. Adding the lexicographically smallest suffix $S_0 = 0001000110$. **B**. Adding suffix $S_4 = 000110$ by splitting the edge at distance $|LCP_{0,4}| = 4$ from the root node.

new leaf node represents the substring starting at position 0 and ending at N. The next suffix to be inserted is suffix S_4, and its $|LCP|$ with the previous suffix is 4 (in bits), which is determined by the comparison of prefixes attached to each suffix start position. We break the edge of the previously added leaf node by converting it into a new internal node at depth 4 from the root and adding two new leaves corresponding to the previous suffix S_0 and a new suffix S_4.

We continue in a similar way filling the output buffer with lexicographically ordered suffixes. When the buffer is full, it is flushed to disk, the array pointer in the output buffer is reset to position zero, and the suffix tree construction starts from the beginning, overriding the previous values of the buffer. All trees written to disk are of equal size, and thus, the problem of data skew is now completely avoided.

Before flushing the current suffix tree to disk, its ID and the prefix corresponding to the boundary path in this tree are added to the collection of *dividers*. The purpose of these dividers is to uniquely identify the suffixes in each tree in order to quickly locate the partial tree of interest during pattern search as described later, in Section 5.1.2.

Let T_i be one of the suffix trees obtained by the algorithm. The leaves of tree T_i correspond to suffixes of a certain lexicographical range as captured by dividers d_{i-1} and d_i.

Each tree is small enough to be quickly loaded into the main memory to perform pattern search or full tree traversal. Since the number of dividers is small, and we store only their first 64 bit prefixes, they can be loaded entirely into the main memory. For example, for the Human genome the total number of dividers is approximately $11,500$, if each tree contains about $256,000$ suffixes.

Concluding Remarks

1. With our method, the total number of suffixes remains N (not $N_b = 2N$) as we sort only N suffixes (in the sorting phase), and merge these suffixes creating one leaf for each suffix.

2. Any internal node in a suffix tree has at least two children. As we use the binary alphabet, the number of children is limited to be at most two. This allows using two child pointers only (per node) and the use of a very efficient implementation of the suffix tree as an array of nodes, without memory re-allocation. In fact, the same output buffer is overwritten by the subsequent subtrees and it also can be easily moved from and to disk as an array of constant-size structures.

3. Due to equivalence of the construction of a suffix tree and a suffix array with LCP, exactly the same merge algorithm can be used for creating on-disk suffix arrays with LCP.

3.2.3 Experimental evaluation

In this section, we present the performance evaluation of *DiGeST* in comparison with *Trellis+*, that was previously considered the fastest algorithm for building suffix trees on disk. The source code of *Trellis+* was obtained from [54]. *DiGeST* was implemented in C and compiled with the GNU gcc compiler, version 4.1.2. All experiments were performed on a machine with an Intel Core Duo 2.66 Ghz CPU, 2 GB RAM and 4MB L2 cache under Ubuntu 7.04, 32-bit Linux. Both programs were compared for different input lengths with the same amount of available main memory (namely, 2 GB). Recall that, to be efficient, all recent algorithms, including *TDD*, *Trellis* and *DiGeST*, require that the input string resides in main memory. The 2-bit per character compression of the input made it possible to build the suffix tree for the input

Table 3.1: Input data sets used in our experiments. For example, line 1 of the table says that data set 1 consists of sequences of chromosomes 1, 2, 20 and 21 of both Human and chimpanzee, with the total input length of approximately 0.9 GB. Each chromosome was stored in a separate input file. Their total size is shown in the last column of the table. The last line represents a data set generated from three entire genomes of Human (22 chromosomes plus X-chromosome), chimpanzee (22 chromosomes plus X-chromosome, excluding M-chromosome) and zebrafish (all 25 chromosomes).

	Chromosomes of			Total
Data set	Human	chimpanzee	zebrafish	size, GB
1	1,2,20,21	1,2,20,21		0.9
2	2–6	2–6		1.8
3	1–8	1–8		2.7
4	1–12	1–12		3.6
5	1–18	1–18		4.5
6	1–23	1–23		5.4
7	1–23	1–23	1–25	6.3

string of 6 GB with 2 GB of total RAM. This tree is of size approximately 180 GB. In practice, RAM cannot be extended to such sizes to build the suffix tree entirely in main memory.

First, we evaluated the performance of *DiGeST* versus *Trellis+* for the Human genome which is of the size of about 3 GB. *DiGeST* was able to build the suffix tree in 1.5 hours, while *Trellis* took 2.5 hours.

In the following, we present further results demonstrating the performance of *DiGeST*. Namely, we evaluated the performance of *DiGeST* versus *Trellis* for the following DNA types:

1. Collection of the corresponding chromosome pairs of the genomes of several species, namely Human, chimpanzee, and zebrafish (obtained from [76]). We grouped the corresponding chromosome pairs into inputs of approximate total size from 1 GB to 6 GB. We believe that such combinations correspond to the

goal of comparative genomics.

2. Synthetic DNA sequences that we generated using a uniform random distribution of characters.

The details of the input data sets for type 1 are presented in Table 3.1.

The running times for *DiGeST* and *Trellis+* are given in Figure 3.22 and Figure 3.23 for the data sets of Table 3.1 and synthetic DNA sequences, respectively.

Observe that *DiGeST* significantly outperforms *Trellis+* for inputs greater than 1 GB. For example, for real DNA inputs of total size 3.5 GB *DiGeST* outperforms *Trellis+* by about 40%.

Our gain in performance is due to the fact that *DiGeST* performs mostly sequential disk I/Os, whereas *Trellis+*, in its tree merge phase, performs multiple random disk reads of the trees built in the previous phase.

We believe that, for inputs of size greater than 4 GB, the number of tree merges of *Trellis+* and the number of partition trees to load for each merge will cause a great deal of random I/Os.

On the other hand, *DiGeST* scales because it never reads the same piece of disk data more than once, and writes the suffix trees corresponding to the lexicographically partitioned suffixes performing only one random disk access per tree.

Next we study the behavior of *DiGeST* and *Trellis+* with respect to the type of input. In Figure 3.24, we fix an input size of approximately 3 GB which is about the size of the Human genome and plot the results of *DiGeST* and *Trellis+* for the three types of data. Namely, we consider (1) synthetic DNA with uniform distribution of characters, (2) Human genome, and (3) a subset of two genomes of similar species, Human and chimpanzee. Human genome has more and longer repetitions than synthetic DNA. On the other hand, the Human and chimpanzee genomes are quite similar and have even more and longer common substrings.

Data Sets	1 (0.9GB)	2 (1.8GB)	3 (2.7GB)	4 (3.6GB)	5 (4.5GB)	6 (5.4GB)	7(6.3GB)
Trellis	46	92	148	216			
DiGeST	34	68	102	135	174	208	244

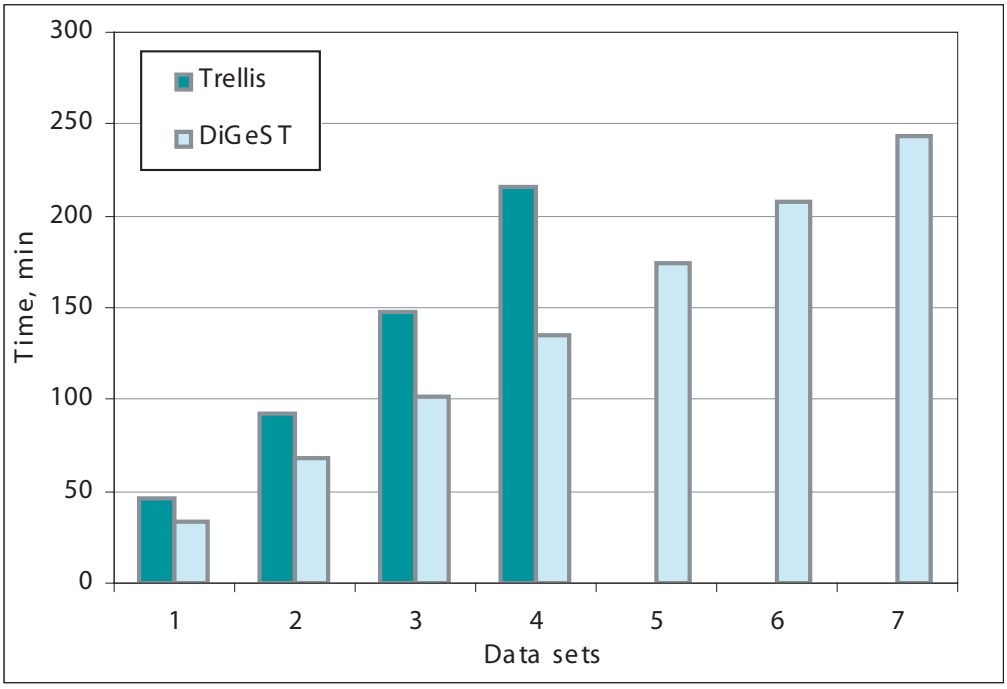

Figure 3.22: Chart of the suffix tree construction time for DNA data sets from Table 3.1. **X axis:** data set ID corresponding to the inputs of roughly 1, 2, 3, 3.5, 4.5, 5.5 and 6.5 GB. **Y axis:** total construction time (minutes). Observe that, for real DNA inputs of total size 3.5 GB *DiGeST* outperforms *Trellis+* by about 40%. For larger sizes only *DiGeST* is able to produce results.

Data Sets	1 (0.9GB)	2 (1.8GB)	3 (2.7GB)	4 (3.6GB)	5 (4.5GB)	6 (5.4GB)	7(6.3GB)
Trellis	39	79	126	191			
DiGeST	33	54	86	118	155	188	229

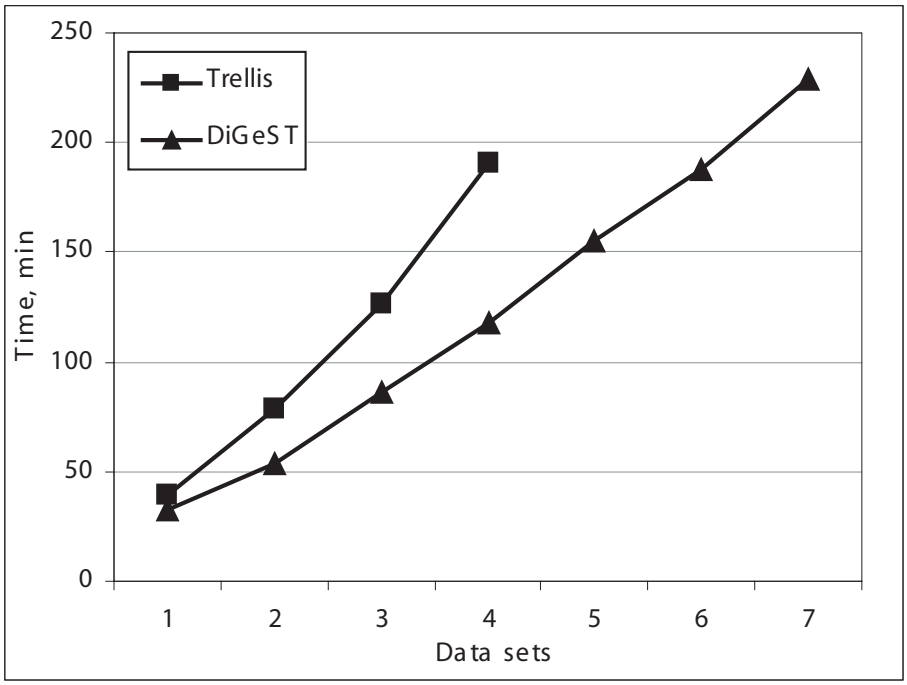

Figure 3.23: Chart of the suffix tree construction time for synthetic DNA of the same lengths as in Figure 3.22. **X axis:** data set ID corresponding to the inputs of roughly 1, 2, 3, 3.5, 4.5, 5.5 and 6.5 GB. **Y axis:** total construction time (minutes). Similar to the previous figure, only *DiGeST* is able to produce results for sequences larger that 4 GB.

Figure 3.24: Chart for the comparative performance of *Trellis+* and *DiGeST*. Suffix tree construction times (in minutes) for *Trellis+* and *DiGeST* on three different input types of size approximately 3 GB each.

Both *DiGeST* and *Trellis+* perform slightly better on synthetic DNA than on the other data sets used in the experiments.

3.3 Summary

In this chapter we described *DiGeST*, our new algorithm for the construction of suffix trees on disk. The algorithm is simple, easy to maintain and implement, efficient and scalable. *DiGeST* can handle several gigabytes of DNA input, significantly outperforming the state-of-the-art programs *TDD* and *Trellis* in terms of efficiency and scalability. With this new algorithm, we could build suffix trees of size hundreds GB in a matter of several hours on an average modern desktop computer, assuming that the entire input fits into a given RAM.

Next we discuss the problem of random access to the input string, which arises when our input is too massive outgrowing the main memory and must be kept on disk during the suffix tree construction.

Chapter 4

Minimizing random access to the input string

All the algorithms for disk-friendly suffix tree construction described so far, including our new algorithm *DiGeST*, reduce the random access to the suffix tree. They demonstrate severe performance degradation once the input string outgrows the main memory and is accessed directly on disk. This is because these algorithms by design assume that the massive random access to the input string is performed in RAM. Once the input string is on disk, this approach translates into a prohibitive number of random disk I/Os. As stated in [64], the construction of suffix trees for inputs that are even slightly larger than the main memory may take weeks and months.

The problem of efficiently constructing suffix trees for extra large strings is very important, because if the entire input string is in main memory, one might be able to find efficient algorithms that scan and search the string without using a (disk-based) index. In this case, the overhead of the on-disk index construction and handling may not pay off. That is, for strings that cannot be entirely loaded into the main memory, the index will be the most beneficial.

The problem is difficult since the best practical algorithms [28, 55, 4, 63] rely on multiple scanning of the input string from non-sequential start positions. In addition, recall that the edges of the suffix tree are not labeled with the actual substrings but rather contain pointers to the input string. Hence, if an algorithm requires a comparison of the characters in the input string with the characters of the edge – label substring, we inevitably need to access the input string at multiple arbitrary locations.

In the following we present our solution to this problem. First, in Section 4.1 we review the algorithms designed to improve the patterns of random access to the input string. The performance and scalability of these algorithms clearly demonstrate that the problem is far from being solved. Next, in Section 4.2.1 we discuss why the techniques which we were successfully applied for *DiGeST* do not extend for the case of inputs in excess of main memory. In Section 4.2.3 we present our new suffix tree construction algorithm B^2ST[1], specifically designed to handle inputs several times larger than the main memory. As a proof of concept, we show in Section 4.2.4 that our method allows to build suffix trees for 12 GB of *real* DNA sequences in 26 hours on a single average machine with 2 GB of RAM. This input is *four times* the size of the Human genome, and the construction of suffix trees for inputs of such magnitude was never reported before.

4.1 Related work

This section summarizes the main techniques that were applied to reduce the random access to the input string during the suffix tree construction.

[1] B^2ST stands for **B**ig tree, **B**ig string **S**uffix **T**ree. We have published this work in [5].

4.1.1 Extension of the *TDD - ST-MERGE*

In [64], the Top Down Disk based suffix tree construction algorithm *TDD* was extended for the case when the input string does not fit the main memory. The authors proposed the *Suffix Tree Merge (ST-Merge)* algorithm, which works as follows. The input of size N is partitioned into K partitions, such that N/K is smaller than the size of the main memory. A suffix tree for each partition is built using the *TDD* algorithm. After this, suffix trees from different partitions are merged into a single tree. In the merge step, the suffix-tree edges from the different partitions are grouped by their first character. Next, their *longest common prefix* (LCP) is calculated by scanning the corresponding parts of the input. The result of this calculation produces a new internal node in the growing suffix tree. Then the process continues for the sub-groups of suffixes that differ in the character next to LCP.

This algorithm was expected to have better locality of references in the access to the input string than the original *TDD* algorithm. However, during the merge performed by *ST-merge* multiple random accesses to the input string are performed. The experimental evaluation reported in [64] has shown that the *ST-merge* algorithm runs an infeasible amount of time even for moderate input sizes. For example, the construction of the suffix tree for an input of size 20MB using 6MB of main memory (allocated for the input string buffer) took about 8 hours. The performance for larger inputs was not reported. In fact, the improvement over the original *TDD* algorithm was insignificant.

4.1.2 *Trellis* with string buffer

Interesting original ideas to overcome the input string bottleneck were proposed by the authors of the *Trellis* algorithm in [56] where they developed a new version of the algorithm – *Trellis with String Buffer* (*Trellis+SB*). Similar to *ST-merge*, the *Trellis*

algorithm is based on a partition and merge strategy. It first builds suffix trees for partitions of the input string. Then it breaks the suffix tree of each partition into sub-trees according to the precomputed set of prefixes. Next, the sub-trees from different partitions which share the same prefix are merged together. The total size of sub-trees for each common prefix allows all such sub-trees to be merged in main memory. In the merge phase, however, the edges of the multiple subtrees need to be compared. Recall that the edges do not contain actual substrings. This comparison requires massive random accesses to the entire input string. This requires the entire input string to reside in main memory, otherwise the performance severely degrades.

To improve this behavior during the merge, in *Trellis+SB* some parts of the input string are kept in the main memory. The rest is read from disk when required. Since suffix-tree edges contain positions of the corresponding substring inside the input string, *Trellis+SB* replaces these positions, whenever possible, by positions in one small representative partition. This small representative part of the input is kept in memory during each merge and increases the buffer hit rate. This technique works only if the representative partition can be found for the entire input, which is not always the case.

Another technique used by *Trellis+SB* is the buffering of some initial characters for each leaf node, next to this node. This buffering proved to be efficient since most of the comparisons are resolved in the first $\log N$ characters of each suffix.

The combination of these techniques allowed in practice to reduce the number of accesses to the on-disk input string by 95%. Note that the remaining 5% for an input of 10 GB correspond to 500 million of random disk I/Os. The authors report that they were able to build the suffix tree for 3 GB of the Human genome, using 512MB of main memory, in 11 hours on an Apple Power Mac G5 with a 2.7GHz processor and 4 GB of total RAM. The performance for larger inputs was not reported.

4.1.3 *DiGeST* and prefix buffering

Similar results were obtained for our *DiGeST* algorithm [4], which merges suffix arrays built for input partitions, using a multi-way merge sort and organizing the output buffer in form of a suffix tree. As explained in Chapter 3, in order to reduce the access to the input string in the merge phase, for each position in the suffix arrays of partitions a 32–character prefix was attached. This prefix served the comparison of suffixes of different partitions in the merge phase of the algorithm. The references to the input string were reduced by 98% for the DNA data used in the experiments in [4].

However, even 2% of remaining random accesses significantly degrade the perfomance of *DiGeST* when the input string is kept on disk.

4.1.4 *WAVEFRONT* and multiple scans

Recently[2], the most naive algorithm by Hunt et al.[28] described in Section 3.1.2 was extended to the case of inputs larger than the main memory. The *WAVEFRONT* algorithm [22] shows better performance for this case than any algorithm described so far.

The algorithm builds a suffix tree in a tiled fashion maintaining a constant working set size. This is achieved by multiple scans over the input string, trading the increase in the total running time for sequential disk I/Os. Since this is the only algorithm, which specifically addresses the problem of the random I/Os to the input string, we describe it here in more detail.

As in Hunt's algorithm, a separate suffix tree is built for each different prefix. The length of this prefix is chosen so that the number of suffixes that share this prefix allows the suffix tree for a given prefix fit entirely into the main memory. If there are

[2]The algorithm was developed at the same time as our B^2ST.

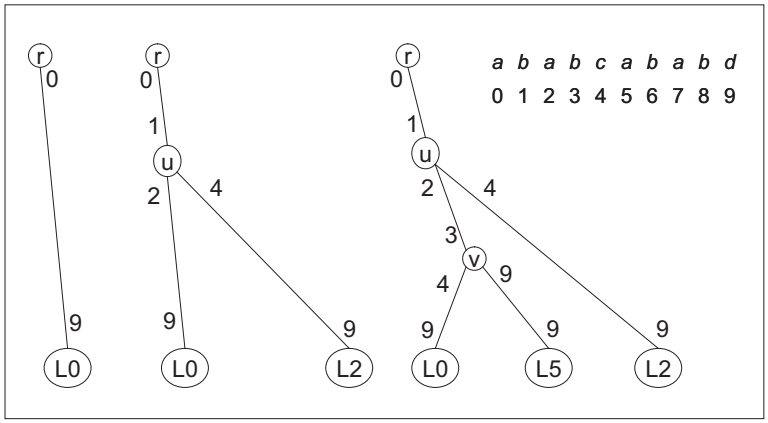

Figure 4.1: The main observation of the *WAVEFRONT* algorithm [22]: the positions on the incoming edges of any node in the suffix tree are always greater than the positions on the incoming edge of its parent. The three first steps of the suffix insertion into the suffix tree for prefix a. The incoming edge of node v is labeled by positions $[2, 3]$, which are greater than the edge-label positions $[0, 1]$ for node u. This holds for all the labels in this figure.

P different prefixes, then the algorithm performs P independent constructions.

What differs from Hunt's algorithm is the input string which is kept on disk. The input string of size N is partitioned into consecutive partitions of size one block B each. There are N/B such input partitions.

The following observation builds the basis of this algorithm: the positions that label an incoming edge of any suffix tree node are always greater than the positions on the incoming edge of its parent. For an example see Figure 4.1. In general, this holds for the brute–force suffix insertion, which was used in this algorithm, i.e. when each current suffix S_j starts to the right of any previously added suffix S_i: $j > i$.

The explanation of this fact is presented in Figure 4.2. When the new suffix S_j starting at position j has been added to the tree, the incoming edge of L_i is broken to create new internal node u and new leaf L_j: u is now a parent of two children: L_i and a new leaf node L_j. The positions on the incoming edge of a changed node L_i are greater than the positions on the incoming edge of node u, since both these labels

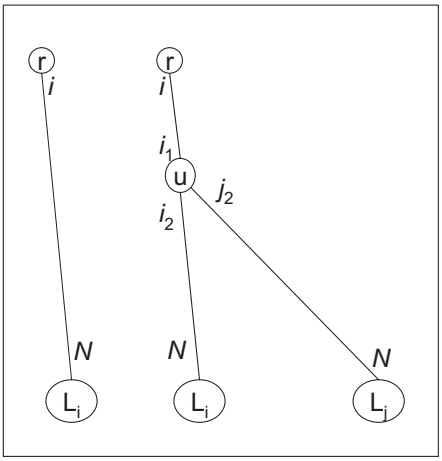

Figure 4.2: The main observation of the *WAVEFRONT* algorithm [22]: $j_2 > i_1$.

represent two consecutive parts of the same suffix S_i. Since the next suffix S_j starts after S_i ($j > i$ by construction) and it shares $i_1 - i + 1$ first characters with suffix S_i, so the positions on the edge to L_j are greater than those on the edge leading to L_i, and thus are greater than the positions on the incoming edge of u – its parent node.

The algorithm completes the construction of the suffix tree for a given prefix in N/B iterations. In the first iteration, only characters in the first block partition of the input string are kept in main memory. This allows to resolve the tree topology for all the nodes whose incoming edges are labeled with the substrings belonging to this first block. Since the positions are increasing from parent to child node, we have resolved all the entire tree topology closer to the top of the suffix tree. Next block of the input string is loaded into main memory and the construction of the tree proceeds in the bottom direction. Note that the entire set of suffixes of X, which share a common prefix, is scanned in each of N/B iterations.

An example of *WAVEFRONT* is presented in Figures 4.3 and 4.4. Each figure represents one iteration in the construction of the suffix tree for all the suffixes of string $X = ababcababd$ which start from a prefix a. In this example, the size of each block is 3 bytes, and only 3 characters of the input string can be held in main memory

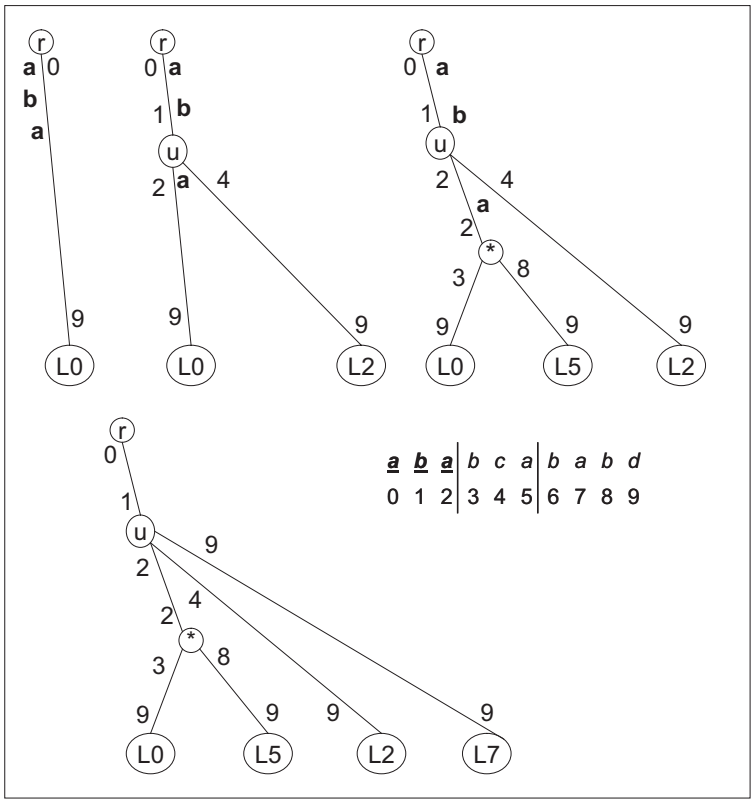

Figure 4.3: The first iteration of the *WAVEFRONT* algorithm. Only the 3 first characters of X are accessible. The topology of the edges labeled by the positions of these first 3 characters are resolved.

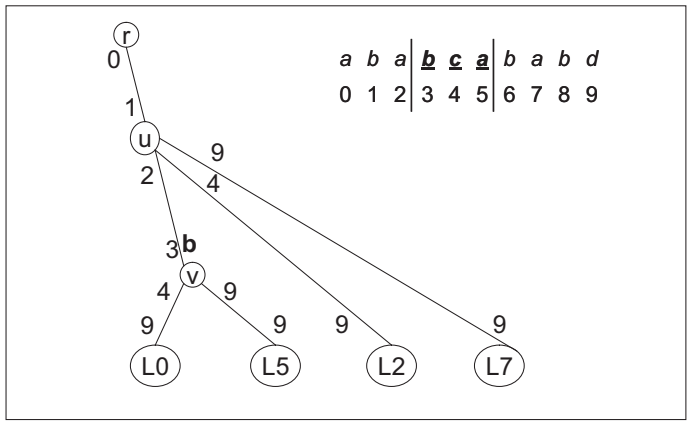

Figure 4.4: The second iteration of the *WAVEFRONT* algorithm. The second block of the input string is in main memory. The corresponding edges and nodes are updated.

in addition to the suffix tree being built. In the first iteration, only characters at positions from 0 to 2 are known. We insert each suffix in order scanning each inserted suffix at most B characters in each iteration. We build the suffix tree based on our knowledge of the first 3 characters. Here, for suffix S_2 the complete topology is found already after the first iteration, since this suffix differs from all other suffixes already in its third character. For the suffix S_5 the temporary internal node $*$ is created. In the next iteration, when the second block of the input string is in memory and we compare the unresolved edge labels, this node would be converted into a complete node v, as shown in Figure 4.4.

Thus, for each prefix and for each block of the input string the entire input is scanned in a sequential order. There are total pN/B scans of the input of size N, where p, the total number of different prefixes, roughly corresponds to the *input-to-memory ratio* $r = N/M$: $p = CN/M$ where C is a constant, which represents the double size of each suffix tree node, and in practice is about 50. There are, therefore, $50N^2/MB$ sequential scans, in each scan at most B characters are scanned starting from N/p positions. The running time of *WAVEFRONT* on a single machine is therefore $O(N^3B/MB) = O(50rN^2)$.

The *WAVEFRONT* algorithm performs at a speed comparable to this of *Trellis+* and is able to index 3 GB of the Human genome in 11 hours with the memory restricted to 512 MB, on a single machine. The results for larger inputs are not reported.

4.1.5 Summary of the existing algorithms

The summary of the algorithms is presented in Figure 4.5. From the evaluation of all these methods, it follows that the suffix tree construction algorithms for inputs in

[1]Using only 512 MB of RAM.

[2]The implementation of *ST-Merge* is not available [56]. This is time of *TDD* instead of *ST-merge*

Method	Based on	Performance for 3 GB[1]	Max. input handled
ST-MERGE [64]	*TDD* [63]	128 h[2]	50 MB [64]
Trellis+ [56]	*Trellis* [55]	11 h	3 GB
DiGeST [4]	*DiGeST* [4]	11 h	3 GB
WAVEFRONT [22]	Hunt et al. [28]	11 h	3 GB

Figure 4.5: The summary of the suffix tree construction algorithms for inputs larger than the main memory.

secondary storage remain impractical and call for a better solution.

In the following section, we present our new **B***ig tree,* **B***ig string* **S***uffix* **T***ree* construction algorithm (B^2ST) [5], which proposes a different solution to this problem.

4.2 Our new algorithm: B^2ST

4.2.1 Re-using the techniques of *DiGeST*

B^2ST is based on the following ideas, some of which were already introduced for the $DiGeST$ algorithm of Chapter 3.

Suffix trees built from suffix arrays

As we already know from Section 3.2.2, the suffix tree ST for string X can be constructed given its suffix array SA augmented with LCP information. In practice, this process exhibits a good locality of references and therefore a good behavior in external memory settings. When we incrementally insert sorted suffixes into the current suffix tree, we perform sequential reading of the suffix array and sequential writing of suffix trees for consecutive lexicographic intervals, without performing random disk I/Os to both these data structures. This shows that both SA and ST can be kept on disk, and only their sequential parts can be loaded and manipulated in main memory.

Consequently, the first step in our algorithm for the suffix tree construction is obtaining a suffix array for the entire input string X, augmented with the LCP information. For this, we want to lexicographically sort all suffixes of X.

Problems with suffix merging

Suffix sorting differs from conventional string sorting in that the elements to be sorted are N overlapping strings, and the length of each such string is $O(N)$. This implies that a comparison-based sorting algorithm, which requires $O(N \log N)$ comparisons, may take $O(N^2 \log N)$ time. Moreover, if we treat suffixes as if they were regular strings we have an even bigger problem: when comparing a pair of suffixes we need to scan the corresponding sequences of symbols in X starting at two positions along

the string X. These positions are mostly non-consecutive. When X is on disk, this translates into a prohibitive number of random disk I/Os.

Suppose that we want to use the external memory two-phase multi-way merge-sort *(2PMMS)* [21]. We partition X into slightly overlapping substrings (partitions) and lexicographically sort the suffixes in each partition. We can do this in main memory by using any of the best algorithms for in-memory suffix sorting [57]. Then, we output to disk the suffix arrays for suffixes in different partitions.

A problem arises when we want to merge these suffix arrays. In the simple case of merging sorted lists of keys, the relative order of elements from any two different lists is determined by comparing these elements. However, in our case, all we have are the starting positions of suffixes from different partitions, since this is all the information we can store in suffix arrays. This does not help in determining the relative lexicographical order of suffixes, since our sorting keys are the substrings of X (and not their starting positions).

A naïve approach would compare two suffixes from different partitions by performing random accesses to X, which is on disk. This would lead to $O(N)$ *random* disk I/Os, a prohibitive amount even for small N. Be reminded that the size N of our input string is several times larger than the available main memory.

4.2.2 A new technique: pairwise sorting

Our main approach to suffix sorting is that in the merge phase of the multi-way merge sort we do not compare the actual input string characters, but rather deduce the necessary information about the relative order and the LCP of any two suffixes from the special structures that we call *pairwise order arrays*. These arrays, for each pair of partitions, are created in the first phase of the merge-sort, and in the merge phase they are uploaded sequentially from their disk runs. We never load the

Partition A					Partition B					Partition C					Partition D				
a	b	a	b	a	a	a	b	b	a	b	b	b	a	b	a	a	b	a	b
1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5

Figure 4.6: Partitioning of input string $X = ababaaabbabbbabaabab$ into four partitions. The combined size of each partition pairs with its tails must be less than the size of main memory M.

entire input string into main memory, and we completely avoid random I/Os to the disk-based input string.

4.2.3 The details of our algorithm

Algorithm B^2ST proceeds in three steps: input partitioning, sorting of suffixes for each pair of partitions and merging all suffixes into a disk-resident suffix tree.

Step 1: Input Partitioning

Our algorithm first partitions the input string X of size N into K partitions such that $K = 2r$ (recall that $r = N/M$ is the input to memory ratio). The partitioning method for oversized input strings is essentially the same as in Section 3.2.2 for $DiGeST$. The difference here is that for $DiGeST$ the maximum size of each partition was chosen to accommodate its suffix array in main memory, and for B^2ST we require the combined size of any pair of partitions $2p$ including their tails to be less than M.

At the end of this step we have on disk K input partitions of size at most p such that $2p < M$. An example is shown in Figure 4.6, where the memory size M is about 16 bytes. It can hold only 2 partitions of size not more than 8 bytes each.

Partition A					Partition B									
a	b	a	b	a	a	a	b	b	a	b	b	b		
1	2	3	4	5	1	2	3	4	5	1	2	3		

SA_{AB} (suffix array)										
suffix start	5	1	3	1	2	5	4	2	4	3
	a	a	a	a	a	a	b	b	b	b
	a	a	b	b	b	b	a	a	a	b
	a	b	a	a	b	b	a	b	b	a
	b	b	a	b	a	b	a	a	b	b
	…	…	…	…	…		…	…	…	…
LCP	0	2	1	3	2	3	0	2	3	1
partition bit	A	B	A	A	B	B	A	A	B	B

Figure 4.7: Suffix array with LCP for pair of partitions A and B for the input in Figure 4.6. The total length of both partitions is less than the size of main memory: $|X_A| + |X_B| < M$.

R_{AB}										
LCP	0	2	1	3	2	3	0	2	3	1
partition bit	A	B	A	A	B	B	A	A	B	B

SA_A				
5	3	1	4	2

written to disk

Figure 4.8: Example of an output after pairwise partition sorting. Two structures are extracted from suffix array SA_{AB}: (1) the suffix array of partition A and (2) the order array R_{AB} storing the relative order of suffixes in A and B. These two structures are written to disk.

```
pairwiseSorting MAIN (INPUT: K partitions of string X)
1     for (u=0; u<K-1; i++)
2           for (v=1; v<K; v++)
3                 concatenate X_u X_v and load into RAM
4                 build suffix array with LCP SA_uv
5                 during sequential scan of SA_uv
6                 if v==K-1  //last chunk
7                       OUTPUT to disk SA_v
8                 OUTPUT to disk SA_u
9                 OUTPUT to disk R_uv          //order array
```

Figure 4.9: Pseudocode for pairwise suffix sorting. The computation of suffix arrays with LCP for a pair of partitions (line 4) is performed in a constant working space. The reading (line 3) and writing (lines 7–9) from/to disk are sequential.

Step 2: Suffix sorting in partition pairs

In this step we generate suffix arrays for each pair of partitions. We concatenate every possible pair u, v of partitions with their tails ($0 \leq u < k-1$, $u+1 \leq v < k$, $u < v$) into string $X_u X_v$. We load this input into the main memory and build the suffix array SA_{uv} with attached LCP information for each suffix. An LCP entry of SA_{uv} is the *length* of the longest common prefix of each suffix in SA_{uv} with its immediate predecessor.

The pseudocode for this step is shown in Figure 4.9. The example in Figure 4.7 shows how the suffix array with LCP look like for the pair of partitions A, B. We know from the description of *DiGeST* that we can build a suffix array with LCP using disk space for any input, given this input fits entirely into main memory.

From each SA_{uv} ($u < v$), we extract two structures: (1) the suffix array SA_u for partition X_u and (2) an "order array" R_{uv} of size $|X_u| + |X_v|$. The *order array* R_{uv} contains the LCP entries of SA_{uv} plus the partition ID information. Since each R_{uv} contains an information only about two partitions, we only need to use *one bit* to

represent the partition ID in R_{uv}. Specifically, we use 0 for u and 1 for v ($u < v$). Figure 4.8 shows SA_A and R_{AB} extracted from SA_{AB} in Figure 4.7.

At the end of this step we have on disk K suffix arrays for K partitions (of total size of $O(N)$), plus $K(K-1)/2$ order arrays for each possible pair of partitions (of total size kN).

This is all the information we need to efficiently perform the next step - the merge. As a result of this merge we produce the suffix tree for the entire input string X. We do this without loading the entire input string into main memory. In fact, we never access X anymore.

Step 3: Merging

In order to merge the suffix arrays of different partitions, we use the information from the order arrays. Notably, all these arrays are accessed sequentially.

More specifically, the merge works as follows. As in the classical *2PMMS*, we have k input buffers for each of the k disk-based suffix arrays created in Step 2. We denote the buffer for a suffix array SA_u by SA_BUF_u.

In addition, we use $k(k-1)/2$ input buffers for order arrays. We denote the buffer for an order array R_{uv} by R_BUF_{uv}.

Finally, we have an output buffer, ST_BUF, where we collect the nodes of the merged suffix tree before emptying it to disk. The total size of all the buffers matches the size of the available main memory.

The pseudocode for the merge step is presented in Figure 4.10.

First we fill the input buffers with the elements of the corresponding arrays (Calls to refill routines in lines 2,3 of B^2ST *merge MAIN.*). We associate a pointer with each SA_BUF and R_BUF pointing to the current element of the buffer. Originally, the pointers are set to the first element of each buffer.

```
B²ST merge MAIN (INPUT: SA_X_u , R_uv 0< u<K 1<v<K, u<v as files on disk)
1       lastTransferred = null
2       for each partition u
                call refillSuffixArrayBufer (u, SA_BUF_u)
3       for each partition pair u, v
                call refillOrderArrayBufer (u, v, R_BUF_uv)
4       for each SA_buf_u
6               insert SA_buf_u[0] into heap //calls to compareSuffix
7       while heap is not empty
8               remove smallest suffix S_i of partition u
                        from the top of the heap
9               rebalance heap //calls to compareSuffix

10              lcp = 0
11              if lastTransferred is not null
12                      v = lastTransferred.partitionID
13                      lcp = LCP from R_buf_uv [current_pointer]

14              create leaf for S_i using lcp in ST_buf
15              call advancePointers (u)

16              lastTransferred=S_i

17              if ST_buf is full
18                      store S_i (max suffix)
                                as a pointer to the current tree
19                      OUTPUT ST_buf to disk
20                      lastTransferred = null

21              S_j = get next suffix from SA_buf_u
22              if S_j is not null
23                      insert S_j into heap
```

Figure 4.10: Pseudocode for the merge step of B^2ST. The algorithm uses a constant working space, and it reads (lines 2,3,15) and writes (line 19) data from/to disk sequentially.

```
refillSuffixArrayBufer (partition ID u, SA_BUF_u of size m)
1    read next m start positions
         from disk suffix array SA_u into SA_BUF_u

refillOrderArrayBufer (first partition ID u, second partition ID v,
                                                  R_BUF_uv of size m/K)
2    read next m/K (LCP+partitionBit) entries
         from disk order array R_uv into R_BUF_uv
```

Figure 4.11: Pseudocode for refilling buffers in the merge step of B^2ST. Each time a piece of the on-disk array is read sequentially into a corresponding main-memory buffer.

We start the processing by comparing the first elements of each suffix array buffer. The first element of each buffer is inserted into a heap. In order to compare two entries of, say, SA_BUF_u and SA_BUF_v $(u < v)$, while inserting to and rebalancing the heap (lines 6,9,23 of B^2ST *merge MAIN*) we consult the partition bit in buffer R_BUF_{uv} under the current pointer, as shown in pseudocode for routine *compareSuffix* in Figure 4.12. If the bit is 0, we conclude that the current suffix of partition u is lexicographically smaller than the current suffix of partition v, and vice versa.

The top element of the heap, the smallest suffix (belonging to, say, a partition u) migrates to the suffix-tree output buffer (line 8, Figure 4.10). The pointer for buffer SA_BUF_u is advanced by 1. The pointers for all the order array buffers containing information about partition u are also advanced by 1, as shown in pseudocode of Figure 4.13. This means we have determined the order of the current suffix of partition u, and we need to consider the next element both in SA_BUF_u and in all relevant order buffers. We insert into the heap the next suffix of partition u, and we continue in a similar way until all suffixes are merged.

At any point, when one of the input buffers is processed till the end, we refill it with the elements of the corresponding on-disk array. This happens after advancing

```
compareSuffix (S_i from partition u,  S_j from partition v)
1       if (u = = v)
2               return -1  //S_i <_lex S_j,  since they are sorted
                           //in increasing order inside each partition
3       if (u < v)
4               if (partitionBit in R_buf_uv[current pointer] = = 0)
5                       return -1               //S_i <_lex S_j
6               else
7                       return 1                //S_i >_lex S_j
8       if (u > v)
9               if (partitionBit in R_buf_vu[current pointer] = = 0)
10                      return 1                //S_j <_lex S_i
11              else
12                      return -1               //S_j >_lex S_i
```

Figure 4.12: Pseudocode for suffix comparison which uses the pairwise suffix information from the order arrays created during pairwise suffix sorting. These arrays are read sequentially.

```
advancePointers (partition ID u)
1       SA_buf_u.current_pointer++
2       if reached the end of SA_buf_u
3               call refillSuffixArrayBufer (u, SA_buf_u)
4       for (i=0; i<u; i++)
5               R_buf_iu.current_pointer++
6               if reached the end of R_buf_iu
7                       refillOrderArrayBufer (i, u, R_buf_iu )
8       for (i=u; i<K; i++)
9               R_buf_ui.current_pointer++
10.             if reached the end of R_buf_ui
11.                     refillOrderArrayBufer (u, i, R_buf_ui )
```

Figure 4.13: Pseudocode for the pointer advancing in the suffix arrays and order buffers.

pointers (see *advancePointers* routine in Figure 4.13). If no data remains in the on-disk array, this array is considered no longer active. When only one active SA_BUF remains, the algorithm finishes up by just adding all the remaining suffixes of this array to the (output) suffix tree.

Note, that the disk-resident suffix arrays and the order arrays are read sequentially, which would not be the case if we were consulting the input string X to resolve a relative order for arbitrary suffix start positions of different partitions.

To summarize, during the merge we determine the relative order and the LCP between suffixes from different partitions from the information collected in the pairwise suffix sorting step. The advantage of this is that the information in the order arrays can be accessed sequentially and thus can be kept on disk. Otherwise we would need to compare substrings of X starting at arbitrary positions.

Suffix Tree Output Buffer

The suffix tree in the output buffer is built exactly as described in Section 3.2.2 for *DiGeST*.

Thus, all that differs in B^2ST is the way in which we determine the order and the LCP of two lexicographically consecutive suffixes while adding them to the growing suffix tree. This is, however, the crucial difference for the efficiency of the suffix tree construction, as we show in the following experimental evaluation.

4.2.4 Experimental Evaluation

We implemented B^2ST in C and compiled with a GNU gcc compiler, version 4.1.2. All experiments were performed on a machine with an Intel Core Duo 2.66 Ghz CPU, 2 GB RAM and 4 MB L2 cache under Ubuntu 7.04, 32-bit Linux.

The first step in our implementation was to choose an algorithm which creates a suffix array with LCP information for each pair of partitions. Our objective was to evaluate the main idea of B^2ST, assuming the required suffix arrays for each pair of partitions are given. For this purpose we used the modification of *DiGeST* [4] which outputs a suffix array with LCP values instead of suffix trees (see Section 3.2.2). *DiGeST* can build suffix arrays for very large inputs, requiring only that the input string fits into main memory. Thus, suffix arrays for 4 GB of DNA are efficiently built using a main memory string buffer of 1 GB assuming the DNA string is first compressed using 2 bits per character.

We first evaluated the performance of our algorithm in comparison with algorithms *TDD*, *Trellis+SB* and *WAVEFRONT*. We obtained the source code for *Trellis+SB* and *TDD* from [54] and [62] respectively. In [22] the *WAVEFRONT* algorithm was shown to perform at exactly the same speed as *Trellis+SB* on a single machine. We included these results into our comparison. Since neither *TDD* nor *Trellis+SB* or *WAVEFRONT* implementations were not reported to handle inputs larger than 3 GB, we designed a comparative experiment for 3 GB of Human genome DNA input.

Both *Trellis+SB* and B^2ST work on compressed inputs (using two bits per DNA character). This means that the 3 GB of DNA occupies 750MB of space. In order to simulate the case when the input string does not fit into main memory, we restricted the total available memory to these algorithms to 600MB. We have used the maximum of available main memory (2 GB) for *TDD* which works with uncompressed inputs. The results are shown in Table 4.1.

TDD builds the suffix tree for the above 3 GB input in 128 hours. We were unable to reproduce the results of *Trellis+SB* reported in [56]. The value in Table 4.1 is the result reported in [56] for similar settings on a comparable machine. The same holds for the *WAVEFRONT* implementation.

Table 4.1: Comparison of suffix tree construction times for Human genome (3 GB input on disk) performed by different suffix tree construction algorithms. The input is larger than the total allocated main memory, and is accessed from disk.

Program	Construction time
TDD	128 hours
Trellis+SB	11 hours
WAVEFRONT	11 hours
B^2ST	3 hours

For B^2ST we divided the 3 GB into partitions of 1 GB each and built the suffix array for partition pairs of a total size of 2 GB. As already mentioned, we used for this 600 MB of main memory. We then merged the arrays using the technique described in Section 4.2.3 into suffix trees of a total size of 59 GB.

The sorting of suffixes for the 3 pairs of 3 partitions took 118 minutes, while the merge took only 13 minutes. This shows that our new algorithm B^2ST achieves a drastic performance improvement over the other algorithms for strings that do not fit the main memory.

This confirms that performing sequential scans as we do in B^2ST pays off compared to just focusing on reducing the number of random disk I/O's to the input string, as the other algorithms do.

Next we evaluated the scalability of our algorithm. Using 1.5 GB of main memory to hold input string we constructed suffix trees for 6, 8, 10 and 12 GB of genomic data. These data sets were generated from combinations of complete eukaryotic genomes obtained from [76]. The exact description of inputs is presented in Table 4.2.

The performance results are shown in Table 4.3. The size of each partition is 2 GB. The partition pair of size 4 GB was compressed into 2 bits per character string, and processed into the suffix array with LCP. For our largest input, 12 GB, we had

Table 4.2: Input data sets used in our experiments. The complete genomic sequences of the listed organisms were concatenated to produce input of approximately 6, 8, 10 and 12 GB.

Data set	Size	Genomes of
1	6.2 GB	Human, chimpanzee, and zebrafish
2	8.4 GB	Human, chimpanzee, zebrafish, and cow
3	9.7 GB	Human, chimpanzee, zebrafish, mouse, and chicken
4	11.7 GB	Human, chimpanzee, zebrafish, cow, mouse, and chicken

Table 4.3: Suffix tree construction time of B^2ST for real genomic DNA (approximately 6, 8, 10 and 12 GB) using only 2 GB of main memory.

Input data set	Number of partitions	Number of partition pairs	Step 1. Pairwise sorting	Step 2. Merge	Total time
1 (˜6 GB)	3	3	7 h 21 min	27 min	7 h 58 min
2 (˜8 GB)	4	6	12 h 10 min	34 min	12 h 42 min
3 (˜10 GB)	5	10	18 h 20 min	42 min	19 h 14 min
4 (˜12 GB)	6	15	24 h 22 min	59 min	25 h 31 min

6 partitions and 15 partition pairs. The time taken to build the suffix arrays of these 15 pairs was about 25 hours and produced an intermediate on-disk output of size 234 GB. Despite this, the merge phase completed in only 59 minutes, scanning all this on-disk data in sequential manner and produced 2514 suffix tree files of total 215 GB, of size about 100 MB each.

This example shows that we need a large temporary disk space for scaling up the B^2ST algorithm. Specifically, we need $D = k^2p = kN$ bytes of disk space to store the order arrays for all partition pairs. Since the number of partitions is $k = N/M$, from $D = N^2/M$ we can determine the size of the largest input that we can process with M bytes of internal memory and D bytes of disk space. If we substitute the common values for modern computers, $D = 10^{12}$ (1 TB), and $M = 4 \times 10^9$ (4 GB), then we can build suffix trees using such a machine for up to 60 GB of input. Note, however,

that the construction of suffix trees even for 10 GB of input was never achieved and reported before.

As for the execution time, it is clear that the construction of suffix arrays for a pair of partitions can be done in parallel, since each such sorting is independent of the others. The scanning of $O(kN)$ intermediate disk structures in the merge step is very efficient due to the sequential reading. So, by using our B^2ST algorithm, indexing a large amount of DNA data with suffix trees becomes a feasible routine task.

4.3 Summary

In this chapter we presented B^2ST, an efficient external-memory suffix tree construction algorithm for very large inputs. The B^2ST algorithm is designed to store all its inputs, outputs and intermediate data structures on disk, making it a truly external memory algorithm. Our B^2ST algorithm minimizes random access to the input string, and accesses the disk-based data structures sequentially.

We have shown that B^2ST scales to much larger inputs than the previous algorithms. Our algorithm is able to build a disk-based suffix tree for virtually unlimited size of input strings, thus filling the ever growing gap between the increase of main memory in modern computers and the much faster increase in the size of genomic databases.

For example, using our implementation of B^2ST we could build the suffix tree for a DNA sequence of total size of about 12 GB in about 26 hours on a single machine using only 2 GB of main memory.

We have shown that B^2ST is several times more efficient than the previously proposed algorithms *TDD* [64], *Trellis+SB* [56] and *WAVEFRONT* [22] specifically designed for input strings larger than the main memory. This is because B^2ST

performs sequential access to both the input string and tree being built.

Since the suffix array construction and the LCP computation for each pair of partitions can be done in time linear in its length $2N/k$ (for example by using algorithms [31] and [33]), and since we have $K(K-1)/2$ different pair combinations, the running time of the sorting step is $O(KN)$, where $K = 2r$. In other words, the time is proportional to the total input size and the input-to-memory ratio r (i.e. how many times our input exceeds the available main memory). The suffix arrays and the order arrays produced in this step require $O(KN)$ temporary disk space. In the merge step, the suffix tree of size $O(N)$ is constructed in time linear in N, this requires, however, a complete scan of the intermediate order arrays of size $O(KN)$. Thus, the total running time of the B^2ST algorithm is $O(KN)$.

For comparison, the running time of the *WAVEFRONT* algorithm [22] is $O(KN^2)$, which indicates the potential scalability of both algorithms.

Chapter 5

Performance of disk-based suffix trees

We start this chapter by describing the on-disk layouts of the suffix trees produced by our algorithms. Next we present performance benchmarks for pattern search in massive inputs with and without suffix trees. The results presented in this chapter clearly indicate great potential of the disk-based suffix trees when analysis very long strings, and therefore also for the mining of genomic data.

5.1 Layouts of the disk-based index

5.1.1 A forest of suffix trees

We remark that most of the algorithms found in the literature (described in Sections 3.1 and 4.1) do not deliver a single suffix tree on disk, but rather a forest of suffix trees. This is useful both from the construction and the query points of view. Regarding the query efficiency, if a single suffix tree is of a size much larger than the available main memory, then searching for a query pattern q may incur $|q|$ random I/Os plus one

random I/O for each occurrence, which sums up to $O(|q| + occ)$ random disk I/Os.

Thus, important practical requirements for the output suffix trees are that each partial tree could be accessed with 1 disk I/O, loaded and traversed entirely in main memory. Furthermore, each tree must have some unique identifier to be located quickly.

These requirements gave raise to several tree partitioning schemes. The one most commonly used is the *partitioning by prefixes*. For each prefix, there is a separate tree which contains all the suffixes sharing this prefix. If the length of each prefix is $|p|$, then the total number of prefixes $P = O(|\Sigma|^{|p|})$. Note that, in order to search for a pattern, we need to find the corresponding prefix, load the corresponding sub-tree performing one sequential read and then find all the occurrences of this pattern in this sub-tree.

(1) *Constant-size prefixes* are used in the algorithm by Hunt et al. [28] and in the *TDD* algorithm [64]. This partitioning scheme works well in practice except when the (real–life) input data is so skewed that for some prefixes the trees are very small, whereas for others so large that they can not be entirely held in the available main memory.

(2) *Variable-length prefixes* are used in *Trellis* [55] and *WAVEFRONT* [22] to improve over the data skew problem. This partitioning scheme is described in detail in Section 3.1.5. In order to determine the collection of variable-length prefixes such that all the output trees are of approximately equal size, multiple scans of the input are performed. However, even after that, some trees may be significantly smaller than others.

Partitioning by prefixes has the disadvantage: for very large inputs, the number of prefixes can be very large. For instance, with equal-length prefixes, the total number of prefixes P grows exponentially in the prefix length. Recall that such algorithms as

the one by Hunt et al. [28], *TDD*[64] or *WAVEFRONT* [22] perform P iterations over the entire input of size N. A single sequential scan by itself is fast, but the number of scans should not be arbitrarily large. This holds the aforementioned algorithms from scaling up for inputs larger than 3 GB. For example, the authors of *TDD* do not recommend the length of prefixes to be larger than 8 characters [63], since otherwise the processing time increases dramatically.

To reiterate, the search for pattern q in a forest of prefixed suffix trees formally follows the following scheme. First, locate the corresponding tree based on the prefix of q. Next load into memory the suffix tree corresponding to this prefix. Finally, perform the search for the exact pattern first by determining the path for the characters of q starting at the root and then by collecting the leaves in the sub-tree induced by the end of this path.

5.1.2 A new partitioning scheme: partitioning by intervals

In our tree construction algorithms described in Chapters 3 and 4, we build the trees from lexicographically sorted suffixes and write the output trees to disk according to the lexicographic intervals of the suffixes in each output tree.

Once a number of suffixes collected in the output buffer reaches a pre-defined number of nodes, the tree is written to disk. The 32-bit prefixes of the lexicographically smallest and the largest suffixes in that tree is recorded in the collection of *dividers*, which divide the resulting forest of trees by their lexicographic intervals. The search starts by locating the proper divider and then loading into memory the entire tree corresponding to this interval. An example is presented in Figure 5.1. All the trees are of equal size. This solves the problem of data skew. In addition, we avoid multiple scans required in order to determine a collection of the variable-length prefixes.

Let us now look at a suffix–tree forest produced by *DiGeST* or B^2ST. Such a

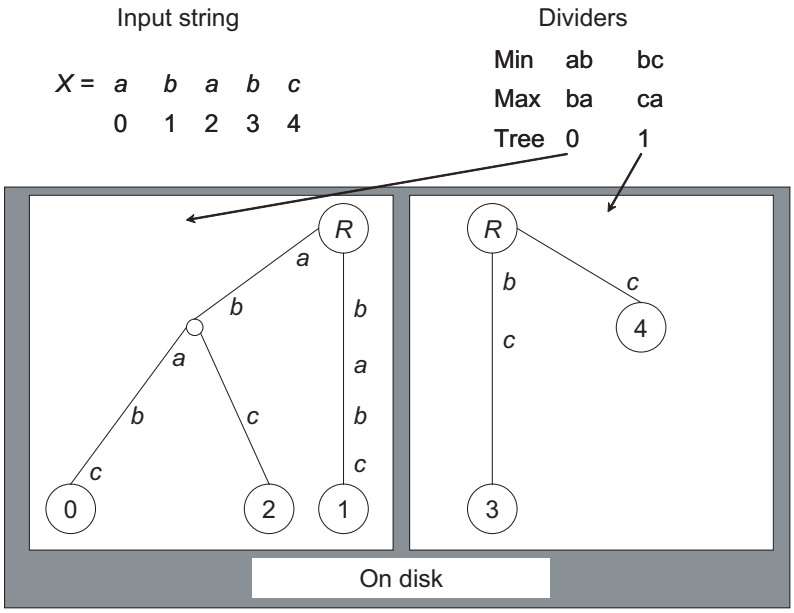

Figure 5.1: Forest of suffix trees for $X = ababc$, produced by our algorithms. The collection of dividers is kept in main memory. With each interval we associate a pointer to a corresponding suffix tree. Such a suffix tree can be loaded from disk by a sequential read.

forest is a collection of suffix trees, each of which is of small enough size to be quickly loaded into main memory using a sequential disk read. The collection of dividers can be loaded entirely in main memory due to the small size of this collection. There is a trade-off between the number of partitions and their size: the number of suffixes in each partition should be adjusted for fast reads. For our search performance experiments, we experimentally determined the optimal size of each tree to be of $512,000$ nodes per tree, accounting for $256,000$ suffixes, and each tree occupies 11.7 MB of disk space. Even for an input of 10 GB, the total number of trees does not exceed $35,000$, and thus the collection of dividers which points to the corresponding trees can easily be kept in main memory during the search.

5.1.3 Edge-labels in binary alphabet

In order to store each interval suffix tree in constant space, we used binary encoding of the suffixes. Note that a string over any alphabet Σ can always be reduced to the binary alphabet by representing each character as a sequence of $b = log|\Sigma|$ bits and then concatenating these binary sequences. The problem of *renaming*, which is a generic reduction from strings over an unbounded alphabet to binary strings, was studied in [16]. It was shown that such a reduction can be done in linear time.

For a binary alphabet, any internal node in the suffix tree has exactly two children. This allows using two child pointers only (per node) and representing the entire suffix tree as an array of the constant-sized nodes. If the entire input string is considered as a sequence of bits, only *valid* suffixes are added to the tree. These are the suffixes starting at positions i such that $i \bmod b = 0$, where b is the number of bits used to represent each character of Σ. As such, the number of tree nodes equals exactly $2N$: the tree has one leaf node and one internal node per inserted suffix.

We had successfully used a binary representation during the conversion of a suffix

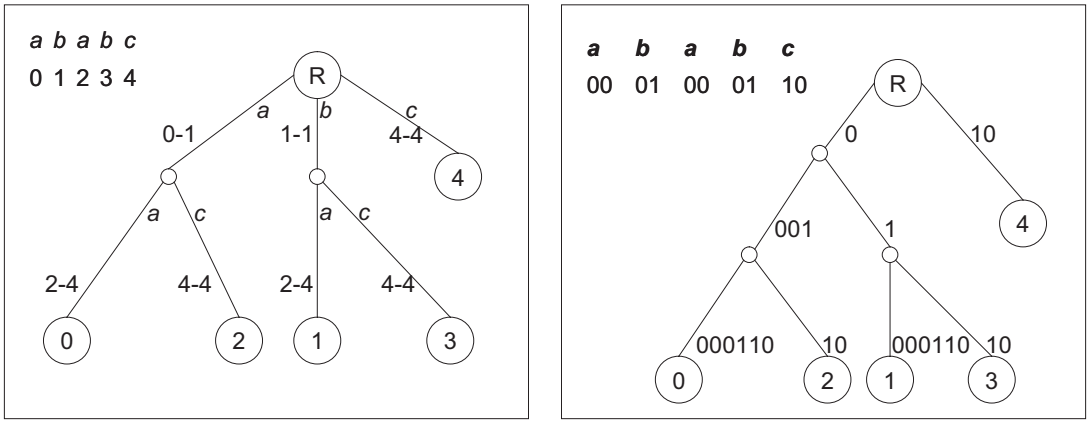

Figure 5.2: [**Left**] Suffix tree for $X = ababc$ given for comparison. [**Right**] Suffix tree for the same input string where each suffix is converted to a sequence of bits. Each character is encoded using 2 bits.

array with LCP to a suffix tree (see Section 3.2.2) when we determined the first character of a newly created internal node without consulting the input string. This constant space representation of a suffix tree also showed to be very efficient while moving the tree from disk to main memory or vice versa. All the reads and writes can be performed using the same constant working space, which is allocated only once during the entire program.

Figure 5.2 shows equivalent suffix trees over the original and the binary alphabets for input string $X = ababc$. Each node has exactly two child pointers plus one integer representing start position of incoming edge-label. Since there are exactly $2N$ nodes in such a tree, total size of the tree is $24N$ bytes. Note that this is independent of the size of the alphabet.

This binary representation of the suffix tree supports many common string queries. For example, in order to find occurrences of a pattern in a string X we can treat the pattern as a sequence of bits and match these bits along the path starting at the root. Also, if we are looking for a longest repeating substring (LRS) of X, and the alphabet contains characters represented by b bits each, we determine the internal node of the greatest depth, say d, from the root. Then we calculate an LRS (with

respect to the original alphabet) as $LRS = \lfloor d/b \rfloor$.

5.2 Exact pattern matching

5.2.1 Problem definition

Problem 1. Exact pattern matching *is to determine at what positions a short string (called* pattern*) occurs as a substring of a larger string (called* text*).*

The solution of the exact pattern matching problem often arises as a sub-task in multiple string searching algorithms (c.f. [26]). The Knuth-Morris-Pratt [37] or the Boyer-Moore algorithms [8] can locate a pattern q of length $|q|$ in a text X of length N in time $O(N + |q|)$, i.e. they are linear in the size of the text. However, for very large inputs considered in this work, this is an unsatisfactory performance, due to the fact that N is extremely large.

After the off-line pre-processing of the text into its suffix tree, the pattern can be located in time $O(|q| + occ)$, where occ is the number of occurrences. However, when the suffix tree resides on disk, we also need to account for the number of random disk I/Os incurred during the search.

5.2.2 Exact pattern matching using disk-based suffix trees

We remind the reader that the suffix tree does not store explicitly the labels of its edges. Instead, the edge labels are represented by an ordered pair of integers denoting its start and end positions in the input string.

Let us assume that we have constructed a suffix tree for an input string significantly larger than the main memory using our B^2ST algorithm, with the input string on disk during the suffix tree construction. In this case, we would like to keep the

input string on disk also during query executions. Note that, to search for a query string q in this tree using a traditional suffix tree traversal [26] we (naïvely) compare the characters of q to the characters of X as indicated by the positions of the edge labels. Such a search, unfortunately, requires multiple random accesses to the input string, and therefore is quite inefficient when X is on disk. In the worst case, this takes as many random accesses to the input string as the length of the query $|q|$.

However, massive random access to X during a search can be avoided if we follow the PATRICIA search algorithm originally described in [50]. The edges outgoing from an internal node are indexed according to the character specified by their start positions. The search consists of two phases.

In the first phase, we trace a downward path from the root of the tree to locate a corresponding suffix S_i. We remark that we do not match all the characters of this path to the characters of our query: we start out from the root and only compare some of the characters of q with the branching characters found in the arcs traversed until we either reach leaf L_i, or no further branching is possible. In the latter case, we choose L_i to be any descending leaf from the last node traversed, say node v.

In the second phase, we read substring $X[i, i + |q|]$ from the input string X which is on disk, by that performing only one random access to the input, instead of $|q|$ as in the usual suffix tree search. We compare $X[i, i + |q|]$ to q; if both are identical, we report an occurrence of q in X and collect all the remaining occurrences from the leaf nodes below v (if v is not a leaf).

Consider, for example, the suffix tree for $X = ababcababd$ and the two queries $q_1 = aaab$ and $q_2 = abab$ shown in Figure 5.3. By matching only the first and the third characters of q_1 or q_2, and after that verifying the queries against suffix S_0, we perform only one random access to the input string per query. Such an efficient (from the external memory point of view) search does not yet exist for alternative indexing

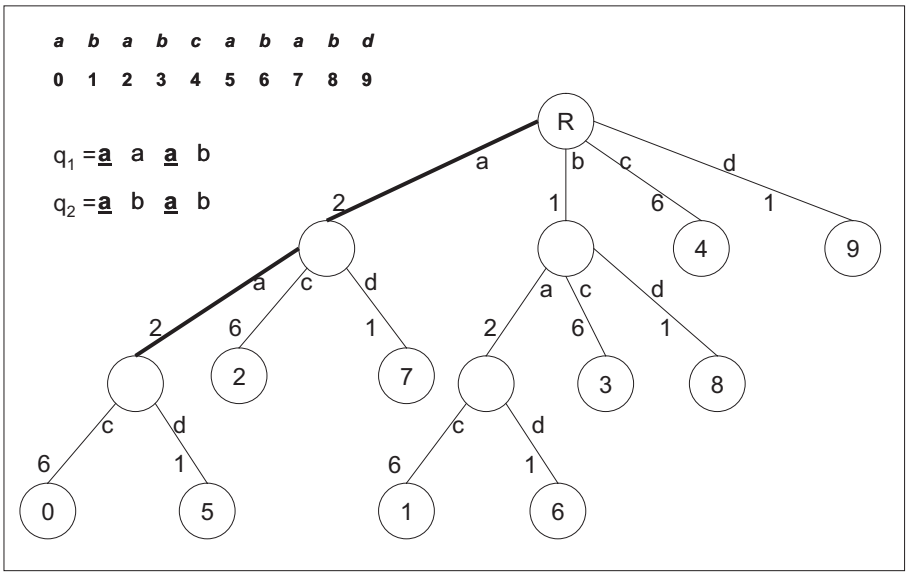

Figure 5.3: A disk-friendly pattern search in the suffix tree for string $X = ababcababd$. The first characters of each edge are implied by the positions of a child node in the array of children. The length of each edge is shown, which is deduced from the start and end positions of edge-label substrings. For query $q_1 = aaab$ we match $q_1[0]$ and $q_1[3]$, and then we retrieve the leaf L_0 as well as the substring $X[0, 3]$. Verification fails, since $aaab \neq abab$. Pattern $q_1 = aaab$ is not a substring of X. For query $q_2 = abab$ we follow the same path. This time the verification is successful, since $q_2 = X[0, 3]$. We report all the occurrences of q_2 in X by collecting leaves 0 and 5. In each case, only one random access to the input string was performed.

structures, such as suffix arrays.

We next describe the pseudocode of pattern search in detail in order to give a code template for tree traversals for further applications of our disk-based indexes.

The pseudocode of the algorithm *diskPatternSearch* used in our pattern search experiments is presented in Figures 5.4 and consists of the following steps.

1. Find the corresponding suffix tree by scanning the collection of dividers (in main memory) – line 3 in Figure 5.4.

2. Load the corresponding tree into main memory (**1 random I/O**) – line 4 in Figure 5.4, and perform search in this tree – call to function *searchInCurrentTree* in line 5. The pseudocode of this function is presented in Figure 5.5. It may happen that the pattern can be found in more than one tree, so the same function will be called for each such tree.

3. Perform PATRICIA match of the pattern to a path from the root of the suffix tree. This match is done as follows. Since we can infer the first character of each outgoing tree label according to the index of a child in the children array, we can match it to the corresponding character of the query. Note that since we have matched only selected characters, the path in the tree found in this step may not completely match the query. However, if we fail to find such a path, then we can immediately conclude that our query string does not occur as a substring of the input strings, and the search stops here – lines 17, 18 of *searchInCurrentTree* in Figure 5.5.

   Before verification we do not have complete information about the edge-labels traversed during search. Therefore, we may say that we search "blindly" (a *blind search*). The traversal for a blind search is presented in Figure 5.6.

```
1   occurrences // global array to collect occurrences
2   diskPatternSearch MAIN (INPUT: pattern q, suffix trees in treeFiles,
                                pointers to each tree dividers)
3       treeID:=locateTree (q, dividers)
4       currentTree:=loadTree (treeFiles, treeID)
5       call searchInCurrentTree (q, currentTree)
6       if count (occurrences) = 0 then
7           return
8       treeID ++
9       while treeID < count (dividers) do
10          if q ≤ dividers [treeID].maxSuffix then
11              currentTree:=loadTree (treeFiles,
                                treeID)
12              call searchInCurrentTree (q,
                                currentTree)
13          else
14              return
```

Figure 5.4: Pseudocode for exact pattern matching using PATRICIA search in disk-based suffix tree. **INPUT:** pattern q, suffix trees for text X in separate files on disk, and the collection of dividers, which maps the lexicographic intervals to the corresponding suffix tree files. **OUTPUT:** all occurrences of pattern p in text X are collected into an array of occurrences, defined as a global variable in line 1. Line 3: *locateTree* finds tree ID according to the lexicographic interval of q. Line 4: *loadTree* reads the entire tree into main memory buffer. Line 5: call to *searchInCurrentTree* presented in Figure 5.5. Line 6: *count* returns count of elements in the array.

4. Retrieve any leaf node from the sub-tree of the node reached by the path found in step 3. There might be more than one such leaf, but the suffixes they represent share the same prefix corresponding to the path found above. Therefore, we look for a leaf representing the smallest between these suffixes – by calling function *findFirstLeaf* from the line 19 of the current tree search in Figure 5.5. The pseudocode for *findFirstLeaf* is presented in Figure 5.7.

5. Obtain the start position of the suffix corresponding to the leaf found in the previous step, and read the input string from disk. (**1 random disk I/O**). Compare its characters against the query string – line 20 in Figure 5.5. If this verification fails (the prefix of this suffix does not match the query string), we conclude that the query string does not occur in X, and the search stops here – line 23.

6. In case of successful verification, collect all the leaves from the sub-tree of the node found in step 3 using a depth first traversal of this sub-tree – call to the recursive function *collectOccurrences* in line 21 in Figure 5.5. The pseudocode of *collectOccurrences* is given in Figure 5.8. The leaves collected present information about the suffixes of the input strings which all have a prefix equal to the query string. Thus we have found all the occurrences of the query string in the input strings.

The entire pattern search requires in most practical cases only 2 random disk I/Os: in step 2 (uploading the corresponding suffix tree) and in step 5 (reading the substring of the input string to verify matching). For some patterns which occur multiple times in the input, the search can spread to several trees corresponding to consecutive intervals.

```
15    searchInCurrentTree (pattern q, suffix tree in RAM currentTree)
16        node:=blindSearch (q, currentTree)
17        if node = NOT_FOUND then
18            return
19        leaf:=findFirstLeaf (node, currentTree)
20        if verify (q, leaf.startPos, leaf.fileNumber) = true then
21            call collectOccurrences (currentTree, node)
22        else
23            return
```

Figure 5.5: Pseudocode for pattern search in the suffix tree after the entire tree is loaded into the main memory. Line 16: call recursive routine *blindSearch* (Figure 5.6). Line 19: if the blind search was successful, call *findFirstLeaf* (Figure 5.7). Line 20: *verify* – load prefix of a suffix corresponding to the found leaf from the disk input file and compare it to query q character–by–character. Line 21: call *collectOccurrences* (Figure 5.8).

```
24      blindSearch (pattern q, suffix tree in RAM currentTree)
25          root:=currentTree [0]
26          firstBit:= bits (q) [0]
27          childIndex:=root.children [firstBit]
28          if childIndex=0 then
29                  return NOT_FOUND
30          currentNode:= currentTree [childIndex]
31          return traverseBlindly (q, currentTree, currentNode, 0, 0)
```

```
32      traverseBlindly (pattern q, suffix tree in RAM currentTree,
                pointer to currentNode in currentTree , current position
                in q to match bitPositionInQ,
                number of characters from the root depth)
33          edgeLength:=currentNode.edgeLength
34          if edgeLength ≥ count (bits (q)) then
35                  return currentNode
36          bitPositionInQ: += edgeLength
37          depth:= bitPositionInQ
38          nextBit:= bits (q) [bitPositionInQ]
39          childIndex:=currentNode.children [nextBit]
40          childNode:=currentTree [childIndex]
41          return traverseBlindly (q, currentTree, childtNode,
                                        bitPositionInQ, depth )
```

Figure 5.6: Pseudocode for a tree traversal during the blind search. Lines 26, 34, 38: *bits* is an array of bits, obtained from encoding of pattern q into a binary alphabet.

```
42      findFirstLeaf (suffix tree in RAM currentTree,
                       current node in this tree currentNode)
43          if currentNode.children[0] then
44              childIndex:= currentNode.children[0]
45              childNode:=currentTree [childIndex]
46              return findFirstLeaf (currentTree, childNode)
47          if currentNode.children[1] then
48              childIndex:= currentNode.children[1]
49              childNode:=currentTree [childIndex]
50              return findFirstLeaf (currentTree, childNode)
51          // currentNode is a leaf
52          return currentNode
```

Figure 5.7: Pseudocode for finding a leaf for verification after the blind search.

```
53      collectOccurrences (suffix tree in RAM currentTree,
                            current node in this tree currentNode)
54          currentNodeIsALeaf:=true
55          if currentNode.children[0] then
56              childIndex:= currentNode.children[0]
57              childNode:=currentTree [childIndex]
58              call collectOccurrences (currentTree,
                                         childNode)
59              currentNodeIsALeaf:=false
60          if currentNode.children[1] then
61              childIndex:= currentNode.children[1]
62              childNode:=currentTree [childIndex]
63              call collectOccurrences (currentTree,
                                         childNode)
64              currentNodeIsALeaf:=false
65          if currentNodeIsALeaf then
66              occurrences [counter++]=(currentNode.startPos,
                                         currentNode.fileID)
```

Figure 5.8: Pseudocode for collecting all occurrences of a pattern in a current tree. No access to the input string is required. Line 66: A leaf is reached. The occurrences are collected into a global array *occurrences* in form of a pair (start position, file number).

Table 5.1: Input data sets used in our pattern search experiments.

Data set set	Number of files	Total size	Description
1	23	3 GB	Human genome (build 18) (from [76])
2	100	8.4 GB	4 eukaryotic genomes (from [76]): Human, chimpanzee, zebrafish and chicken
3	6643	113 MB	viral genomes (from [77])

5.2.3 Experimental evaluation

The C-implementation of the search described in the previous section was compiled with GNU gcc compiler, version 4.1.2. All experiments were performed on a machine with an Intel 4-core 2.93 GHz CPU, 2 GB RAM and 4 MB L2 cache under Ubuntu 9.04, 64-bit Linux.

First, we evaluated the search performance for different sizes of each partial suffix tree. It turned out, that the loading of the corresponding tree into main memory dominates the overall running time of the search program. In our experimental settings the best size of each sub-tree was 11.7 MB ($512,000$ nodes per tree, which accounts for $256,000$ suffixes). Larger trees took more time to load from disk. The further decrease of the sub-tree size not only leads to an increase of the number of dividers, but also leads to the search in multiple sub-trees for each query.

In order to evaluate the search performance on real data sets, we have constructed suffix trees for the following sets of sequenced genomes, described in Table 5.1.

Next, we report the query performance for these data sets in Table 5.2.

The results in Table 5.2 show that the query performance is extremely high. Notably, even reporting of all $255,998$ occurrences of the query of length 10 in the Human genome takes in total not more than 30 ms. For comparison, the linear sequential search by Unix *grep* program (an optimized implementation of the Boyer–

Table 5.2: The performance of the exact pattern matching using suffix trees for massive inputs.

Data set	Query length	Max occurrences	Time per query		
			Min, ms	Max, ms	Ave, ms
1	10	255998	17	29	20
	100	3	19	116	46
	1000	1	17	80	36
2	10	466445	20	161	35
	100	5	51	223	134
	1000	1	52	212	112
3	10	15534	9	14	12
	100	1	6	12	11
	1000	1	9	13	11

Moor algorithm [8]) in the same settings takes as long as 1 min 20 seconds, which is 40 times slower than using the disk-based suffix tree index.

Thus, the exact pattern search in a disk-based suffix trees is very efficient.

5.3 Summary

We have presented the query performance results for disk-based suffix trees, constructed by our two new fast and scalable algorithms *DiGeST* and B^2ST. The output suffix trees were laid out on disk according to lexicographic intervals. These results clearly indicate the great practical potential of the suffix tree indexes for exact pattern matching on a genome scale. In case of very large inputs and even larger suffix trees, the running time of search is dominated by the number of random disk I/Os to the tree and to the input string. The blind search described above with 2 random disk I/Os per search makes the exact pattern matching using suffix trees, from the random disk access point of view, better than the search using any other known full-text index. This power of the suffix trees can be harnessed for solving numerous

problems of great practical importance such as searching for patterns with wildcards or regular expressions, a problem that often arises in bioinformatics when searching for a profile of a promoter inside the entire sequenced genome.

Chapter 6

Future research

We conclude the thesis outlining the problems that remain to be efficiently solved in order to use disk-based suffix trees to their full potential. Our future research consists of finding better solutions for two major problems: improving the efficiency of the suffix tree construction and enabling the use disk-based suffix trees for more complex tasks, such as computing common substrings, approximate pattern matching and so on.

6.1 Better algorithms for the construction of suffix trees

In parallel to the development of practical software for suffix tree construction on disk, promising theoretical results for this problem were obtained.

As a matter of fact, an algorithm that runs in $O(SORT(N))$ time in the theoretical Disc Access computational model (DAM)[1] was developed in [17]. here, $SORT(N)$ means that we can build the suffix tree for an input string of any size with a time complexity equal to that of several external-memory sorts, applied to integers. This

algorithm recursively builds separate suffix trees for suffixes starting at even and odd positions and then merges them into a suffix tree for X. The merge phase is the real bottleneck of this algorithm, and it is not clear whether it can possibly be implemented in practice, as was also noted in [61]. The details of this external memory algorithm are extremely complex, preventing so far an implementation of this algorithm.

More promising results were obtained for building a suffix array of X. As we know from [17] and Section 4.2.1, not only we can convert each suffix tree into a suffix array in linear time using a depth-first traversal of the suffix tree, but we can also convert a suffix array into a suffix tree in linear time, assuming that the suffix array is augmented with the longest common prefix information for consecutive sorted suffixes. This is performed by simulating an Euler tour of the tree under construction using LCP information (see Section 3.2.2).

Following this direction, in order to develop a practical algorithm to construct suffix trees for input strings of any size, we need three essential steps to be efficient from an external memory point of view: building the suffix array, generating an array of LCPs and converting this enhanced suffix array into the suffix tree.

As for the suffix array construction, the optimal results obtained for the DAM computational model look very promising. The *SKEW* algorithm, proposed in [30] and generalized into the *DC* (Difference Cover) algorithm in [31], is a simple and elegant algorithm that builds suffix arrays on disk in $O(SORT(N))$ time. This algorithm was first implemented in the *DC-3* program [15] and has demonstrated a promising practical performance for large inputs, though it never scaled for the size of Human genome [64]. A better implementation of this algorithm can be used as a first step to overcome the input string size bottleneck on the way to fully-scalable suffix trees.

The conversion of a suffix array into a suffix tree turned out to be disk-friendly,

since reads of the suffix array and writes of the suffix tree can be performed sequentially.

However, the suffix array needs to be be augmented with LCP information in order to be converted into a suffix tree. There exist linear-time, space-efficient and easy-to-implement LCP computation algorithms (see for example [33, 46]) that, however, perform random access to at least one intermediate array of size N. These algorithms would severely degrade in performance once N is larger than the main memory, therefore they need to be modified for such a case.

Theoretical results for computing LCP in external memory settings in time $O(SORT(N))$ were presented in [9]. These results are based on range minima queries, performed using special tree-like data structures and an external memory sort of queries to minimize random disk I/Os. These results were used by the authors of the *DC* algorithm [31] to show how to compute the LCP enhancement for their suffix array. So far, nobody has been able to efficiently implement this idea.

We also would like to move the best performing practical algorithms towards a better asymptotic time complexity. The important drawback of algorithms with a good practical performance (cf. [64, 55, 4]) is the internal asymptotically quadratic time of these algorithms. Though $O(N \log N)$ on average and for most inputs, this time increases dramatically if we construct a generalized suffix tree for similar DNA sequences. For example, in order to compare DNA sequences of different genomes of the same species (c.f "the 1000 genomes project" [32]), when building a generalized suffix tree for these sequences, all the algorithms presented in [64, 55, 4] will perform poorly. The only algorithm not suffering from this problem is B^2ST. However, it performs in time $O(rN)$, where $r = N/M$ is the input-to-memory ratio. Due to this increased running time, we might need to perform this algorithm on multiple processors, in order for the suffix tree construction to scale to the size of the entire

Genebank (more than 100 GB of input).

Implementing a parallel version of our algorithm is thus the first task in the priority list of our future work.

6.2 Suffix-tree based algorithms in disk settings

With a look at the potential applications of the on-disk suffix trees, we notice another important problem. If we want to find an approximate occurrence of the query string q in X, then we are searching for substrings of X which match q with some errors. This process is called an *approximate pattern matching*. An approximate pattern matching using the algorithm by Navarro and Baeza-Yates [51, 52] cannot be performed with the same efficiency as an exact pattern matching using disk-based trees. The proposed algorithm requires the comparison of actual characters of the string in order to find the edit distance between the different substrings of q and the substrings corresponding to the edge-labels. Therefore, the approximate pattern matching algorithm proposed in [52] will not work efficiently in external memory settings in the case that the input string cannot be held in main memory. This requires the development of new approaches for approximate pattern matching. The adaptability and the efficiency of other algorithms using on-disk suffix tree layouts is yet to be investigated.

The range minima queries used for computation of the LCP during tree construction, are also used for finding the lowest common ancestor of two suffixes in the suffix tree. Since no practical implementation of this problem for disk settings exists, the algorithms based on the constant-time lowest common ancestor (LCA) retrieval are not (yet) applicable for the disk-resident suffix trees. These algorithms include an important algorithm by Landau and Vishkin [42] for an error-bounded approximate pattern matching, and finding common substrings for a set of multiple strings in time

$O(N)$. Recent advances in the adaptation of LCA queries to the DAM computational model are presented in [14]. These are interesting theoretical results, whose implementation will require multiple calls to a sub-routine of external-memory sort, and this can make the practical efficiency of these methods quite questionable. All the LCA algorithms proposed for DAM are quite involved and difficult to implement. Thus, this would be an interesting area for an ambitious researcher.

The last note about differences between on-disk and in-memory suffix trees is on the usefulness of suffix links. Recall that suffix links connect each internal node representing some substring αy (where α is one character long) of X with some other internal node where substring y ends. The suffix links, a by-product of the linear time construction algorithms, can be useful by themselves. An example is finding occurrences of all substrings of the query q in the input string X. In this case, after locating some prefix $q[0, i]$ of the query string, we follow the suffix link from the lowest internal node of the path found in the tree, and check the next substring starting from the node at the other end of the suffix link. By this we save as many character comparisons as the depth of this internal node from the root. Though not used during most of the suffix tree construction algorithms presented here, suffix links can be recovered in a post-processing step [55, 22]. The suffix link recovery step is as expensive as the entire suffix tree construction due to numerous random I/Os to different partial trees. We believe that these recovered suffix links in the external memory settings are of a limited use, since a link can lead to a different subtree laid out in distant disk locations. This means that an assumed "constant-time" jump following a suffix link can cause in fact an entire random disk access. Note that finding all substrings of the query will require the same number of disk accesses using suffix links as checking all different suffixes of q using the PATRICIA search in the corresponding subtrees. As for other algorithms that make use of suffix links (see for

example [41, 40]), it seems that they might require new non-trivial adaptations when moved from in-memory to on-disk settings.

6.3 Conclusions

In this thesis we presented two new algorithms for the suffix tree construction using external memory. Both algorithms perform well in practice and can be immediately used for indexing all the substrings in the database of sequenced genomes. We believe that these algorithms are important steps towards a fully scalable solution for constructing suffix trees on disk for inputs of any type and size. Once this is done, a whole world of new possibilities will be opened, especially in the field of biological sequence analysis.

Bibliography

[1] A. Aggarwal and J. S. Vitter, *The input/output complexity of sorting and related problems*, Commun. ACM **31** (1988), no. 9, 1116–1127.

[2] A. Andersson, N.J. Larsson, and K. Swanson, *Suffix trees on words*, CPM, LNCS 1075, 1996, pp. 102–115.

[3] A. Apostolico and W. Szpankowski, *Self-alignments in words and their applications*, J. Algorithms **13** (1992), no. 3, 446–467.

[4] M. Barsky, U. Stege, A. Thomo, and C. Upton, *A new method for indexing genomes using on-disk suffix trees*, CIKM, 2008, pp. 649–658.

[5] ———, *Suffix trees for very large genomic sequences*, CIKM, 2009, pp. 1417–1420.

[6] S.J. Bedathur and J.R. Haritsa, *Engineering a fast online persistent suffix tree construction*, ICDE, 2004.

[7] D. Benson, I. Karsch-Mizrachi, D. Lipman, J. Ostell, and D. Wheeler, *Genbank*, Nucleic Acids Research **34** (2006), 17–20.

[8] R. Boyer and J. Moore, *A fast string searching algorithm*, Commun. ACM **20** (1977), no. 10, 762–772.

[9] Y. Chiang, M. Goodrich, E. Grove, R. Tamassia, D. Vengroff, and J. Vitter, *External-memory graph algorithms*, ACM–SIAM Symposium on Discrete Algorithms, 1995.

[10] R. Clifford, *Distributed suffix trees*, J. Discrete Algorithms **3** (2005), no. 2–4, 176–197.

[11] R. Clifford and M.J. Sergot, *Distributed and paged suffix trees for large genetic databases*, CPM, 2003, pp. 70–82.

[12] International Human Genome Sequencing Consortium, *Initial sequencing and analysis of the human genome*, Nature **409** (2001), no. 6822, 860–921.

[13] A. Crauser and P. Ferragina, *A theoretical and experimental study on the construction of suffix arrays in external memory*, Algoritmica **32** (2002), no. 1, 1–35.

[14] E. Demaine, G. Landau, and O. Weimann, *On cartesian trees and range minimum queries*, ICALP (1), 2009, pp. 341–353.

[15] R. Dementiev, J. Krkkinen, J. Mehnert, and P. Sanders, *Better external memory suffix array construction*, Workshop on Algorithm Engineering and Experiments, 2005.

[16] M. Farach and S. Muthukrishnan, *Optimal logarithmic time randomized suffix tree construction*, ICALP, LNCS 1099, 1996, pp. 550–561.

[17] M. Farach-Colton, P. Ferragina, and S. Muthukrishnan, *On the sorting-complexity of suffix tree construction*, J. of the ACM **47** (2000), no. 6, 987–1011.

[18] P. Ferragina and R. Grossi, *The string B-tree: a new data structure for string search in external memory and its applications*, J. of the ACM **46** (1999), no. 2, 236–280.

[19] J. Fischer, V. Mäkinen, and G. Navarro, *An(other) entropy-bounded compressed suffix tree*, CPM, LNCS 5029, 2008, pp. 152–165.

[20] J. Fischer, V. Mäkinen, and N. Välimäki, *Space efficient string mining under frequency constraints*, ICDM, 2008, pp. 193–202.

[21] H. Garcia-Molina, J.D. Ullman, and J.D. Widom, *Database system implementation*, Prentice-Hall, Inc, 1999.

[22] A. Ghoting and K. Makarychev, *Serial and parallel methods for I/O efficient suffix tree construction*, SIGMOD, 2009, pp. 827–840.

[23] R. Giegerich and S. Kurtz, *From ukkonen to mccreight and weiner: A unifying view of linear-time suffix tree construction*, Algorithmica **19** (1997), no. 3, 331–353.

[24] R. Giegerich, S. Kurtz, and J. Stoye, *Efficient implementation of lazy suffix trees*, Software—Practice and Experience **33** (2003), no. 11, 1035–1049.

[25] T. Ryan Gregory, *The evolution of the genome*, Academic Press, 2005.

[26] D. Gusfield, *Algorithms on strings, trees, and sequences: Computer science and computational biology*, Cambridge University Press, 1997.

[27] W-K. Hon, K. Sadakane, and W.-K. Sung, *Breaking a time-and-space barrier in constructing full-text indices*, IEEE Symposium on Foundations of Computer Science, 2003, p. 251.

[28] E. Hunt, M.P. Atkinson, and R.W. Irving, *A database index to large biological sequences*, The VLDB Jornal **7** (2001), no. 3, 139–148.

[29] A. Jacobs, *The pathologies of big data*, Commun. ACM **52** (2009), no. 8, 36–44.

[30] J. Kärkkäinen and P. Sanders, *Simple linear work suffix array construction*, ICALP, 2003.

[31] J. Kärkkäinen, P. Sanders, and S. Burkhardt, *Linear work suffix array construction*, J. of the ACM **53** (2006), no. 6, 918–936.

[32] J. Karow, *Group unveils "1,000 genomes" study to map genetic variants using new sequencing tools*, `http://www.genomeweb.com/sequencing/`.

[33] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park, *Linear-time longest-common-prefix computation in suffix arrays and its applications*, CPM, LNCS 2089, 2001, pp. 181–192.

[34] D. Kim, J. Sim, H. Park, and K. Park, *Linear-time construction of suffix arrays*, CPM, LNCS 2676, 2003, pp. 189–199.

[35] K. Kiran, SA. Ansari, R. Srivastava, N. Lodhi, CP. Chaturvedi, SV. Sawant, and R. Tuli, *The TATA-box sequence in the basal promoter contributes to determining light-dependent gene expression in plants*, Plant Physiol. **142** (2006), no. 1, 364–76.

[36] D. Knuth, *The art of computer programming, volume 3: Sorting and searching*, Addison-Wesley, 1998.

[37] D. Knuth, J. Morris Jr, and V. Pratt, *Fast pattern matching in strings*, SIAM J. Comput. **6** (1977), no. 2, 323–350.

[38] P. Ko and S. Aluru, *Space efficient linear time construction of suffix arrays*, J. of Discrete Algorithms **3** (2005), no. 2–4, 143–156.

[39] S. Kurtz, *Reducing space requirement of suffix trees*, Software Practice and Experience **29** (1999), no. 13, 1149–1171.

[40] S. Kurtz, A. Phillippy, A.L. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S.L. Salzberg, *Versatile and open software for comparing large genomes*, 2004, pp. doi:10.1186/gb–2004–5–2–r12.

[41] S. Kurtz and C. Schleiermacher, *Reputer: fast computation of maximal repeats in complete genomes*, Bioinformatics **15** (1999), no. R12, 426–427.

[42] G. Landau and U. Vishkin, *Introducing efficient parallelism into approximate string matching and a new serial algorithm*, STOC, 1986, pp. 220–230.

[43] N.J. Larsson, *Strings, compression, and orchestra*, `http://www.larsson.dogma.net/research.html`, April 2010.

[44] N.J. Larsson and K. Sadakane, *Faster suffix sorting*, Tech. Rep. LUCS-TR:99-214, Dept. of Comp. Sc., Lund University, Sweden, 1999.

[45] U. Manber and E. Myers, *Suffix arrays: A new method for on-line string searches*, SIAM J. of Computing **22** (1993), no. 5, 935–948.

[46] G. Manzini, *Two space saving tricks for linear time lcp array computation*, SWAT, LNCS 3111, 2004, pp. 372–3837.

[47] G. Manzini and P. Ferragina, *Engineering a lightweight suffix array construction algorithm*, Algorithmica **40** (2004), 33–50.

[48] E. M. McCreight, *A space-economical suffix tree construction algorithm*, J. of the ACM **23** (1976), no. 2, 262–272.

[49] U. Meyer, P. Sanders, and J. F. Sibeyn (eds.), *Algorithms for memory hierarchies, advanced lectures [dagstuhl research seminar, march 10-14, 2002]*, Lecture Notes in Computer Science, vol. 2625, Springer, 2003.

[50] D. Morrison, *Patricia – practical algorithm to retrieve information coded in alphanumeric*, J. of the ACM **15** (1968), no. 4, 514–534.

[51] G. Navarro and R. Baeza-Yates, *A new indexing method for approximate string matching*, Technical Report TR/DCC-98-14, Department of Computer Science, Univ. of Chile, 1998.

[52] ———, *A hybrid indexing method for approximate string matching*, J. of Discrete Algorithms **1** (2000), no. 1, 205–209.

[53] M. Nelson, *Fast string searching with suffix trees*, `http://marknelson.us/1996/08/01/suffix-trees/`, April 2010.

[54] B. Phoophakdee, *Source code for* TRELLIS, `http://www.cs.rpi.edu/$\sim$zaki/software/trellis/`, April 2009.

[55] B. Phoophakdee and M.J. Zaki, *Genome-scale disk-based suffix tree indexing*, SIGMOD, 2007.

[56] ———, *Trellis+: An effective approach for indexing massive sequence*, Pacific Symposium on Biocomputing, 2008.

[57] S.J. Puglisi, W.F. Smyth, and A.H. Turpin, *A taxonomy of suffix array construction algorithms*, ACM Comput. Surv. **39** (2007), no. 2, 4.

[58] L. Russo, G. Navarro, and A. Oliveira, *Dynamic fully-compressed suffix trees*, CPM, LNCS 5029, 2008, pp. 191–203.

[59] K. Sadakane, *Compressed suffix trees with full functionality*, Theory Comput. Syst. **41** (2007), no. 4, 589–607.

[60] J. Sirén, *Compressed suffix arrays for massive data*, SPIRE, 2009, pp. 63–74.

[61] W. Smyth, *Computing patterns in strings*, Addison-Wesley, 2003.

[62] S. Tata, *Source code for* TDD, `http://www.eecs.umich.edu/tdd/`, April 2009.

[63] S. Tata, R.A. Hankins, and J.M. Patel, *Practical suffix tree construction*, VLDB, 2004, pp. 36–47.

[64] Y. Tian, S. Tata, R. Hankins, and J. Patel, *Practical methods for constructing suffix trees*, The VLDB Journal **450** (2007), 219–232.

[65] E. Ukkonen, *On-line construction of suffix trees*, Algorithmica **14** (1995), no. 3, 249–260.

[66] N. Välimäki, W. Gerlach, K. Dixit, and V. Mäkinen, *Compressed suffix tree – a basis for genome-scale sequence analysis*, Bioinformatics **23** (2007), 629–630.

[67] ———, *Engineering a compressed suffix tree implementation*, Workshop on Experimental Algorithms, LNCS 4525, 2007, pp. 217–228.

[68] J. Vitter, *External memory algorithms and data structures: dealing with massive data*, ACM Computing Surveys **33** (2001), no. 2, 209–271.

[69] J. Vitter and M. Shriver, *Algorithms for parallel memory: Two-level memories*, Algorithmica **12** (1994), 110–147.

[70] P. Weiner, *Linear pattern matching algorithm*, IEEE Symposium on Switching and Automata Theory, 1973, pp. 1–11.

[71] A. Woolfe, *Highly conserved non-coding sequences are associated with vertebrate development*, PLoS Biology **3** (2005), no. 1, e–journal.

[72] J. Zobel, A. Moffat, and K. Ramamohanarao, *Inverted files versus signature files for text indexing*, ACM Transactions on Database Systems **23** (1998), no. 4, 453–490.

[73] *Homo sapiens chromosome 1 (36.3)*, http://www.ncbi.nlm.nih.gov/mapview/, April 2009.

[74] NCBI Genebank *overview*, http://www.ncbi.nlm.nih.gov/Genbank/, April 2010.

[75] *The on-line Bible*, http://www.onlinebible.net/win31.html, April 2010.

[76] *USCS genome browser*, http://hgdownload.cse.ucsc.edu/downloads.html, April 2010.

[77] *Viral bioinformatics resource center*, http://biovirus.org/viruses.asp, April 2009.