I would like to dedicate this thesis to my Mum, Marianna and Dad, László

Köszönöm, hogy felneveltetek, és hogy idáig juthattam

Efficient Computational Approach to Identifying Overlapping Documents in Large Digital Collections

Krisztian Monostori

Submitted in fulfillment of
the requirements for the degree of
Doctor of Philosophy

School of Computer Science and Software Engineering,
Monash University

30 December, 2002

Abstract

With the rapid growth of the Internet, large collections of digital documents have become available. These documents may be used for various purposes including education, research, entertainment, and many others. Given this diversity of objectives in using these documents, we need tools that are capable of identifying overlap and similarity within a potentially very large set of documents. Applications of such tools include plagiarism detection, search engines, comparative analysis of literary works, and clustering collections of documents.

This thesis studies different approaches to identifying overlap between electronic documents. Existing approaches are compared based on different attributes including accuracy, performance, and the degree of protection. A novel two-stage approach is proposed in this thesis. It selects a set of candidate documents in the first phase by applying chunking and indexing methods. The second phase uses exact comparison methods based on suffix trees.

Suffix trees have been identified as an efficient data structure for exact string-matching problems. One of the main arguments against the widespread use of suffix trees is their extensive space-consumption requirements. We propose a new data structure, which has the same versatility as a suffix tree but requires less space than any other representation known to date. This new structure is called a suffix vector because of the way it is organised in memory. We show how this structure can be constructed in linear time and we also prove that this data structure requires the least space among those representations that have the same versatility. Not only does the new representation save space but it is also more time-efficient because it eliminates certain redundancies of a suffix tree. Therefore, some phases of algorithms running on the structure can be eliminated, too.

This thesis also analyses how existing data structures can be used for document comparison. Sparse suffix trees and directed acyclic graphs generated from suffix trees are discussed in the context of document comparison applications. Some algorithms have been modified to suit the above mentioned data structures.

We have built a prototype system, called MatchDetectReveal (MDR) that implements the algorithms we proposed. The MDR system is capable of efficiently identifying overlapping documents in a large set and uses suffix trees and suffix vectors in its core component – the matching engine.

To speed up the comparison of documents we have developed a parallel algorithm to process documents. We have used a general-purpose cluster of commodity workstations with general-purpose middleware to test our algorithms as well as a special-purpose cluster with purpose-built software based on the message-passing model. Results of these tests are presented in this thesis and they demonstrate both time- and space-efficiency of the proposed algorithms.

The results of this thesis have been presented at 13 international and national conferences in Australia, New Zealand, Europe, and North America.

This thesis consists of 8 chapters, and is 224 pages long. It contains 87 figures and 13 tables and has a list of 116 references.

This thesis contains no material which has been accepted for the award of any other degree or diploma in any university and, to the best of my knowledge and belief, contains no material previously published or written by another person, except when due reference is made in the text of the thesis.

[redacted]

Contents

List of Figures

List of Tables

Outcomes of the Thesis Work

Published full papers:

- Monostori K., Zaslavsky A., Schmidt H. Parallel Overlap and Similarity Detection in Semi-Structured Document Collections. *Proceedings of 6th Annual Australasian Conference on Parallel and Real-Time Systems (PART '99), Melbourne, Australia, 1999.* pp 92-103, 1999.
- Monostori K., Zaslavsky A., Schmidt H. Parallel and Distributed Document Overlap Detection on the Web. *Workshop on Applied Parallel Computing – PARA2000, 18-21 June 2000, Bergen, Norway.* pp 206-214, 2000.
- Monostori K., Zaslavsky A., Schmidt H. MatchDetectReveal: Finding Overlapping and Similar Digital Documents. *Information Resources Management Association International Conference (IRMA2000), 21-24 May, 2000 at Anchorage Hilton Hotel, Anchorage, Alaska, USA.* pp 955-957, 2000.
- Monostori K., Zaslavsky A., Schmidt H. Efficiency of Data Structures for Detecting Overlaps in Digital Documents. *Proceedings of the 24th Australasian Computer Science Conference, Bond University, Gold Coast, Queensland, 29 January - 2 February, 2001.* pp 140-147, 2000.
- Zaslavsky A., Bia A., Monostori K. Using copy-detection and text comparison algorithms for cross-referencing multiple editions of literary works. *Proceedings of the 5th European Conference on Research and Advanced Technology for Digital Libraries, September 4-9 2001, Darmstadt, Germany,* pp 103-114, 2001.
- Monostori K., Zaslavsky A., Bia A. Using the MatchDetectReveal System for Comparative Analysis of Texts. *Proceedings of the Sixth Australasian Document Computing Symposium (ADCS 2001), Pacific Bay Resort, Coffs Harbour, 7 December, 2001,* pp 51-58, 2001.
- Monostori K., Zaslavsky A., Vajk I. Suffix Vector: A Space-Efficient Suffix Tree Representation. *Proceedings of the International Symposium on Algorithms and Computation, Christchurch, New Zealand, Dec 19-21, 2001,* pp 707-718, 2001.

- Monostori K., Zaslavsky A., Schmidt H. Suffix Vector: Space- and Time-Efficient Alternative To Suffix Trees. *Proceedings of the 25th Australasian Computer Science Conference, Monash University, Melbourne, Victoria, 28 January - 1 February, 2002.* pp 157-166, 2002.
- Finkel R. A., Zaslavsky A., Monostori K., Schmidt H. Signature extraction for overlap detection in documents. *Proceedings of the 25th Australasian Computer Science Conference, Monash University, Melbourne, Victoria, 28 January - 1 February, 2002.* pp 59-64, 2002.
- Monostori K., Finkel R., Zaslavsky A., Hodász G., Pataki M. Comparison of Overlap Detection Techniques. *The 2002 International Conference on Computational Science, Amsterdam, The Netherlands, 21 - 24 April, 2002.* (I) pp 51-60, 2002.

Published short papers:

- Monostori K., Schmidt H., Zaslavsky A. Document Overlap Detection System for Distributed Digital Libraries. *ACM Digital Libraries 2000 (DL00), 2-7 June, 2000 in San Antonio, Texas, USA.* pp 226-227, 2000.

Published posters:

- Monostori K., Schmidt H., Zaslavsky A. MatchDetectReveal: Finding Overlapping and Similar Digital Documents. *The 10th Australasian Conference on Information Systems, Wellington, New Zealand, 1999.* pp 1223, 1999.
- Monostori K., Schmidt H., Zaslavsky A. Identifying Overlapping Documents in Semi-Structured Text Collections. *Australasian Computer Science Conference, Canberra, Australia, 2000.*
- Monostori K., Zaslavsky A., Schmidt H. Digital Documents in Educational Environment: Misuse, Appropriation, and Detection Issues. *Fourth Australasian Computing Education Conference, 4 - 6 December 2000, Monash University, Melbourne.* pp 254-255, 2000.

CHAPTER ONE

Introduction

1

1.1 Introduction

Proliferation of personal computers and workstations, increased quality of printers and display devices, the ever-increasing storage capacity of servers as well as desktop computers, and the world-wide network connecting computers have made electronic publishing technologically feasible and have led to increased use of digital libraries [CMPS94].

Electronic documents are available in large numbers on the Internet. The World Wide Web dates from the late 1980s [BR99a] but only for a few years has it been widely used in many areas of everyday life. The Internet has become the primary publishing medium for researchers as well as commercial organisations. The size of the Web grows exponentially [BR99a], and according to NetSizer [Net01, October 2001] there are currently 127.914 million hosts on the Internet.

Documents on the Internet are easily downloadable and they can be used for different legal or illegal purposes. There are documents not intended to be freely published, however someone can easily purchase a book in electronic format, and publish it on the Internet. Documents may be copied within the Internet and documents downloaded from the Internet may go into print.

This thesis discusses and proposes different algorithms and techniques for computationally efficient identification of overlapping documents. A new two-stage approach is proposed where the first stage uses the usual procedure of identifying candidate documents, and the second stage uses exact string matching to identify exactly matching chunks of documents. We propose to use the suffix tree structure along with the matching statistics algorithm to detect copies of chunks. Suffix trees are very space-consuming structures, so we propose three alternatives to reduce space. These alternatives are discussed in Chapters 4, 5 and 6. Sparse suffix-tree representations, which are discussed in Chapter 4, can most efficiently be used for comparing natural-language text. At the same time the DAG representation (Chapter 4) and the suffix-vector representation (Chapters 5 and 6) are more general representations. The suffix vector representation is a fundamentally new representation of a suffix tree, which is as versatile as any other suffix tree representation, so it can be used in any area where suffix trees are used.

This chapter discusses different issues regarding copying electronic documents, using the Internet as a distribution media, and it also gives an overview of the thesis.

1.2 Copying Electronic Documents

In the following subsections we give a detailed discussion of different reasons for copying electronic documents, and we also describe how copy-detection methods, which are the main focus of this thesis, could be applied to issues discussed below.

1.2.1 Illegal Copies of Electronic Documents

We all know how useful a full-text electronic version of a book is. We can search the text, we can print parts of it if we do not need the whole book. We can cite portions of the text by a simple drag-and-drop operation. Why is it then that most of the books are not available in electronic format? One of the reasons could be that people might misuse the aforementioned advantages of electronic documents. Someone purchases an electronic book and then puts it on his/her Web site, or posts it to mailing lists or bulletin boards. This behaviour is against copyright laws, and is an important reason why publishers are reluctant to make their publications available in electronic form [GS95a]. To tackle this problem with a copy-detection system the publisher can submit its electronic document to the system, which would identify any partial or full match with existing documents on the Internet.

1.2.2 Replicas of Popular Documents on the Internet

There are many documents on the Internet that are replicated on purpose with the permission of the legal copyright holder of the document. Exact copies of the following documents are stored on multiple sites [BGM97]:

- FAQ (Frequently Asked Questions) or RFC (Request for Comments) documents [RFC01]
- On-line documentation for popular programs
- Documents stored on mirror sites
- Legal documents

Partial copies of documents can also be found on the Internet. Documents may have substantial overlap because they are [BGM97]:

- Different versions of the same document
- The same document with different formatting
- The same document with site-specific links, customisations
- Combined with some other text to create a larger document
- Split into smaller documents

A copy-detection system could find these replicas or near replicas of documents and help users in one of three ways:

- A search-engine user could be prevented from reading the same document more than once just because the search-engine returned documents from mirrored sites [GS95a]. The copy-detection system can be used as an additional filter to eliminate duplicates or near duplicates.
- It can also be used in a reverse fashion and it can present the user with only the differences between files, so the user only has to read additional information that he or she has not yet read.
- It can present the user with alternative sites to download a given document, for example for reducing download time. For this solution full as well as partial overlap detection is required, too.

1.2.3 Overlap among Documents in a Local File System

This problem is very similar to the problem described in Section 1.2.2. The only difference is that local files are copied for different reasons. As the capacity of hard disk storage is always just not enough, everyone has been faced with the problem of "cleaning" his/her hard disk. One may have very similar documents just because they are different versions of the same document; they are temporary documents, which have not been deleted; they are different programs containing the same procedure, etc. A copy-detection system could cluster such documents and one could decide which files are necessary to keep and which files can be discarded. If such a system also has access to a central archive, one is able to discover some files that have already been stored in that central location, so there is no need to store them redundantly on the local machine.

1.2.4 Plagiarism

Everyone who has been in academia for some time has faced the problem of students copying each other's work or using documents downloaded from the Internet to submit as their own work. Students have a handy tool for writing research papers, because most of the scientific publications are available in electronic format. However, this is a two-edged sword and it is tempting for students to download ready-to-submit documents from the Internet or download relevant documents and use different portions of them to assemble a "new" document that they can submit. Even if we are almost certain that the given document is not a student's genuine work, unless we confront him/her with the original documents it is impossible to prove that he/she is a plagiarist. Manually finding these documents is very hard even with today's sophisticated search engines.

Plagiarism is not limited to students. There are several reported cases discussed by Kock [Koc99] and Deninng [Den95], that researchers have chosen the easy way to have the required number of publications rather than publishing their own work. The "publish or perish" law might have put too much pressure on them. In the rest of this section we summarize the case of plagiarism reported by Kock [Koc99]. Not only does it reveal some frequently used "intelligent" ways of plagiarism but it also tells a story of how hard it is to take even an obvious case to court, and catching cheaters still has low success.

In 1997 Kock, who is a professor at Temple University in the USA, discovered that someone, who is disguised under the name of Plag in the article, has used Kock's previously published paper and submitted it to a journal. Kock's colleague happened to review the paper and e-mailed Kock that there was a paper, which heavily referenced his work. It was not a coincidence since Plag plagiarised Kock's paper. Among the references Kock also found some papers that are almost impossible to get hold of, like manuals for some special courses. Of the 51 paragraphs in Plag's paper, 38 were almost sentence-by-sentence copies of Kock's article. The paper discussed the results of a survey, and Plag was smart enough to change localities and numbers. Kock sought legal advice, but US lawyers did not find the money involved big enough to take up the case before court. Kock finally went public in a conference, and Plag got his well-deserved punishment from the research community.

Kock's discovery was accidental, but a copy-detection system would catch more Plag-like people. Kock's paper also warns us that such a system must be prepared for sophisticated methods of plagiarism. Though moving away from the Web as a distribution medium would make it harder for plagiarists, it would also impede bona fide researchers, who use the Web as the primary medium for research resources.

Not only does the Internet make it easier for plagiarists but it also makes it possible to create a plagiarism-detection system that would find these documents on the Internet. The methods used for plagiarism detection are very similar to those described in the previous subsections, and catching plagiarists is one of the main motives behind copy-detection systems, which was also our initial goal when we started this project.

1.3 Outline of Thesis

This thesis studies the theoretical foundations of document overlap detection and investigates how different algorithms may or may not be used efficiently for this purpose. In this thesis we propose a two-stage comparison technique, which is more reliable and cost effective than previous approaches. The first stage uses the usual procedure of identifying candidate documents, and in the second stage candidate documents are compared by exact-matching methods that use suffix trees. We have developed two alternative space-efficient suffix tree representations. Besides the theoretical background, we have also developed a prototype system to test different algorithms. As part of the development some other crucial components have been developed:

- the Visualiser component to visualise the results

- the Document Generator component to create sufficient numbers of documents for testing
- an On-line Submission component that enables Web-based submission of documents

The structure of the thesis is the following. In Chapter 2 we discuss existing copy-detection systems and their underlying algorithms. These systems are built for different purposes and use different algorithms. These algorithms are analysed for time and space efficiency as well as applicability to different problems discussed in the previous sections. We also analyse string-comparison algorithms, which form the basis of the second stage of our approach. In this chapter we introduce the suffix-tree structure, which is a versatile data structure. We also discuss different areas of string matching where suffix trees can be applied. Suffix trees can be built in linear time using different construction algorithms. Three fundamental suffix-tree construction algorithms are presented. We focus on Ukkonen's algorithm [Ukk95] because it is the algorithm that we use to build our modified suffix tree, and this is the algorithm that is modified to construct our suffix vector structure.

The matching statistics algorithm is a linear-time algorithm that is able to find the length of the longest overlapping chunk in a string starting from each position. We use the matching statistics values to detect overlapping chunks. The matching statistics algorithm is also described in this chapter.

The main contributions of this thesis are the modified suffix trees presented in Chapter 4 and the suffix vector representations discussed in Chapters 5 and 6. Chapter 3 discusses the different suffix tree representations that have been developed to date. They are compared on their versatility and space requirements. The most important issue of versatility is the suffix link information in the tree. There are many algorithms that make use of the suffix link extension of the suffix tree, including the matching statistics algorithm. All data structures that we use have this suffix link information. We discuss Kurtz's Improved Linked List Implementation [Kur99] in detail, because this is the most space-efficient suffix tree representation known to date that also keeps suffix link information. In later chapters we show that our suffix vector representation is as versatile as Kurtz's representation, but requires less space on average than Kurtz's representation. We briefly discuss representations that do not keep suffix link information but may be more efficient in algorithms where this information is not required.

Chapter 4 discusses modifications to the suffix tree structure that allows them to be stored in less space. However, these representations still resemble the original tree representation, unlike the suffix vector representation, which is a fundamentally new approach and is discussed in Chapters 5 and 6. One possible modification to the tree is to include only suffixes starting at the beginning of words, because we are not interested in overlaps starting in the middle of a word. This approach has been proposed before [KU96, ALS99] but it has not been discussed in the context of the matching statistics algorithm, whose running time can significantly be reduced by utilising this structure [MZS01]. We analyse these issues and we show that this modified tree along with its suffix links, which have different meaning in this representation, can speed up the matching statistics algorithm. Another possible way of reducing the space requirement of a suffix tree is to convert it into a directed acyclic graph. We show how suffix link information can be kept in this structure and how the matching statistics algorithm can run on these structures. We also discuss the benefits of using these structures for document comparison rather than the original suffix tree [MZS01].

Chapters 5 and 6 discuss the suffix-vector structure. This structure is a fundamentally new structure that is as versatile as any other suffix tree representation but requires the least space on average. This data structure can be used on any string, not only natural language text, as it does not make use of word separators in the text as opposed to the sparse suffix tree of Chapter 4.

Chapter 5 introduces the new suffix vector representation [MZV01]. It is first introduced at the high level and some general features of the structure are described. Then we discuss alternative physical representations. We have developed two representations: general suffix vector and compact suffix vector. The general representation can be constructed in linear time from scratch, while the compact suffix vector can be converted from any other suffix tree or suffix vector representation in linear time, thus giving a linear-time overall construction time. The space requirement of the compact representation is compared to that of Kurtz's representation in this chapter. It is shown that the compact suffix vector requires less space on average than Kurtz's Improved Linked List Implementation [Kur99]. Statistical information on the suffix vector is analysed to illustrate which categories of texts could most probably benefit from the suffix vector representation and which categories of texts have only marginal benefit over Kurtz's representation.

Chapter 6 is mainly concerned with the algorithms that run on the suffix vector structure [MZS02]. First, a linear-time construction algorithm of the general representation is discussed. We formally prove the linear-time bound of the algorithm, and present practical results to support the claim. The conversion algorithm from the general representation into the compact representation is also discussed in this chapter. The space-requirement of any representation is only one issue; another very important issue is how fast we can access information stored in the data structure. Since both Kurtz's and our compact representations are very low-level representations, we have to analyse how many operations are required to retrieve certain pieces of information from the tree because this will affect the running time of the algorithms that run on the structure. This chapter also contains the analysis of the space requirements of the general suffix vector representation.

Chapter 7 discusses our prototype copy-detection system, MatchDetectReveal (MDR) [MZS00b, MZS00c]. First, the general architecture of the system is presented with a short introduction to each component. The search-engine and matching engine components are based on the algorithms discussed in Chapters 2, 4, 5, and 6. The Document Generator component is used to generate random documents for testing the overlap detection, where the amount of overlap along with other parameters can be set. The Visualiser component presents the results to the user in a Web browser [MZB01]. It takes the results of the comparison from the Matching Engine as an input and generates HTML files that make use of JavaScript features to present the overlapping between documents. A case study of comparing different documents of the Miguel de Cervantes Digital Library [BP01] is also presented in this chapter [ZBM01]. Copy-detection on a large scale is very resource consuming, which suggests the application of parallel and distributed approaches for copy-detection [MZS99, MZS00a]. Two test-beds have been used: the Monash Parallel Parametric Modelling Engine [Clu99], which is a local cluster at the School of Computer Science and Software Engineering that uses the EnFusion software (formerly known as Clustor) to distribute and load-balance jobs; and a local cluster with 4 machines dedicated to parallel comparison of documents. The Windows implementation of the standard MPI Library [WMPI01] has been used for interprocess communication between processes. Different document and job distribution strategies are analysed here.

Chapter 8 summarises the results of the thesis and revisits the main contributions.

CHAPTER TWO

Document Comparison Systems – Literature Review

2

2.1 Introduction

In this chapter we give an overview of existing document overlap detection systems. These systems have been built for different purposes, but the underlying problem is the same: try to identify related overlapping or identical documents. Commercial systems include Plagiarism.org [Pla99], EVE [EVE00], IntegriGuard [Int01], PaperBin [Pap01], CopyCatch [Cop01], and Glatt [Gla99]. Documentation on research prototype systems is widely published; this chapter summarises the different approaches used in these packages and compares them based on some fundamental issues, such as chunking primitives, fingerprint-selection algorithms, comparison algorithms, and selection criteria. The systems we analyse include the SCAM (Stanford Copy Analysis Method) system [GS95a], the Koala system [Hei96], the "shingling approach" of [BGM97], and the file-system clustering method of the sif tool [Man94]. We also discuss some alternative methods for safeguarding intellectual property [CMPS94, BLMO94a, BLMO94b, SLL97].

Exact string-matching algorithms also are analysed in this chapter. We define the document overlap problem and analyse existing algorithms based on their applicability to the problem. We point out that the document overlap problem can most efficiently be solved by using suffix trees. Thus we introduce suffix trees in this chapter but we leave the discussion of different physical representations of suffix trees until Chapter 3.

2.2 Copy-prevention mechanisms

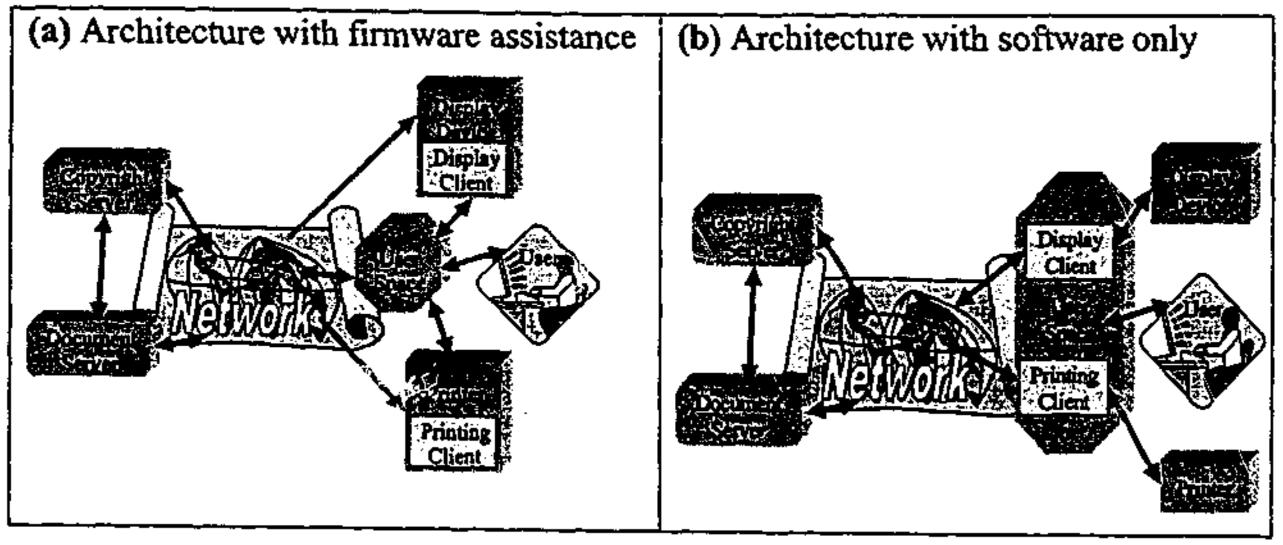
As already discussed in the previous chapter, we have two fundamental approaches (copy prevention and copy detection) to safeguarding intellectual property. One possibility is to prevent violations from occurring. This method makes sure that it is not possible or at least makes it hard enough to copy electronic documents. By "hard enough" we mean that producing a copy of a document should be at least as expensive as purchasing an original copy.

One method suggests a designated access point to an electronic document, such as a stand-alone machine with a CD-ROM drive [GS95b]. If we make sure that text cannot be copied from that CD-ROM, then it is impossible to create an illegal copy of the document. For example, if this machine does not have either a floppy drive or a network connection then the material cannot be copied.

Choudhury et al. [CPMS94] propose a document-distribution architecture that uses asymmetric encryption schemes. In this scheme a public key is used to encrypt the document. This document is then decrypted by a private key on the client end. Two architectures are proposed: The first uses hardware-based encryption/decryption, while the second scheme relies on software components for cryptography. Both architectures share the following components [CPMS94]:

- **Document server**: provides encrypted documents to user; trusted by publisher
- **Copyright server**: authenticates users; trusted by publisher
- **Display client**: decrypts and displays documents; software trusted by publisher
- **Printing client**: decrypts and prints document; software trusted by publisher

In the first step, the user determines the unique identifier for the document with the aid of some sort of searching service. The user then requests the document, the encrypted document is delivered to the user and the software on the client side either displays or prints the document after decrypting it. The hardware-based and software-based architectures are depicted in *Figure 2.1*.


(a) Architecture with firmware assistance

(b) Architecture with software only

Figure 2.1. Hardware-based (a) and Software-based (b) Document Distribution

The hardware-based approach is superior to the software-based approach because in the software-based approach the document or some kind of a representation of the document (e.g. bitmap) exists on the client computer and might be stored with the help of a suitable program.

As another alternative, Griswold [Gri93] describes active documents as one of the ways of protecting intellectual property. Active documents are documents presented by special-purpose software. The text of the document is only available in an encoded format on the disk and only that special software is able to decode the document. Since the document is presented by software, actions that we can perform on the text depend on the software, and it can prevent us from copying text from the document.

The above-mentioned three methods are the most popular methods for copy-prevention, but we believe that these methods are too cumbersome for bona fide users and suppress the inherent advantages of electronic documents, thus copy-detection methods are more favourable than copy-prevention methods.

2.3 Digital Watermarking

Digital watermarking [BLMO94a, BLMO94b] is a copy-detection method applied in electronic publishing. Copy-detection does not try to hinder the distribution of documents but rather tries to detect illegal copies. One of the issues in copy-detection methods is how to identify the original distributor of the document. That is, once an illegal copy is found, the question is then who purchased the original copy of the document and made it available to other users. Digital watermarking can be considered as a complementary tool for copy-detection mechanisms. Since the main focus of this thesis is how to find copies of documents, and not how to identify who spread the given document, we discuss one method of watermarking proposed by Brassil et al. [BLMO94a, BLMO94b] as a representative example of such methods.

The essence of the watermarking method is to place undetectable codewords in documents similar to the method of placing watermarks in bank notes. This codeword can represent a unique document identifier, and this identifier can be assigned to a given customer who purchased the document. These codewords can be hidden in the document by slightly altering the layout of the text. These alterations must be reliably decodable yet not perceptible to the user. These criteria are conflicting, so it is not trivial to design a proper method. Common to these methods is the alteration of some kind of textual feature. Brassil et al. [BLMO94a] study three different methods: line-shift coding, word-shift coding and feature coding.

Line-shift coding vertically shifts the locations of text lines. Originally the space between lines is evenly distributed all over the document, but users would not notice a slight alteration in the distance between two consecutive lines. Brassil et al. [BLMO94a] tested a method that kept every second line intact and every line between the unchanged lines was either moved up or down. The unchanged lines serve as control lines and the lines shifted up or down carry the coding.

Word-shift coding horizontally shifts words within their lines. It makes use of the fact that most texts are left- and right-justified, and the space between words is calculated by an algorithm. One possible application of this method is selecting the largest spacing within a line and decreasing it by some amount, while the smallest spacing is increased by the same amount, thus maintaining the length of the text line.

Feature coding alters the actual bitmap representation of a given character depending on the codeword. One possibility is to increase or decrease the height of a character by a small amount. Good candidate characters for this method are b, d, and h because of the visibility criteria for changes.

Brassil et al. [BLMO94a] test the algorithm in the presence of some noise. Noise sources in this case are the inaccuracy of printers, scanners, and photocopy machines. They thoroughly analyse the line-shifting method and discuss two alternatives. In the baseline-detection method they try to identify the baseline of a given text line, while the centroid-detection method uses the centre of mass of a text

line profile. According to Brassil et al. [BLMO94a] centroid detection is more reliable and to reduce the effect of noise we also can apply error-correcting coding schemes.

Of course, once users know that watermarks are used in a document they may try to destroy those watermarks. Such destruction is possible, but the goal here is not to develop a "bullet-proof" method. We only require that destroying such watermarks should be at least as costly as obtaining the original documents legally.

2.4 Copy-Detection Systems

As we pointed out in Section 2.2, we believe that copy-prevention methods are too cumbersome for bona fide users, so copy-detection systems are more favourable. There have been several copy-detection methods developed, and although they had different final goals they have much in common. In this section, we first define some general concepts for copy-detection problems and then existing systems are analysed based on these concepts.

Before classifying copy-detection problems let us define some general goals that we expect from our copy-detection method:

- **Performance**: We expect our system to be as fast as possible. Most of the systems use some index, which has to be built. Thus performance not only means query performance, but we also have to analyse how long it takes to build that index.
- **Storage capacity**: The index should not require excessive space. It means that we have to analyse how much space is required to store that special-purpose index and the effect that different index sizes have on performance and accuracy.
- **Accuracy**: Appropriate accuracy measures must be introduced. Accuracy is linked to the likelihood that overlapping documents will be found. We also have to analyse how the accuracy of the system is related to other measures, such as performance and storage capacity.
- **Protection**: How hard it is to do some modifications to the text and deceive the system. In an ideal situation it is at least as hard to circumvent the system as it is to write an original document, for example, if our target is plagiarism detection.

General concepts for copy-detection are introduced by Brin et al. [BDG95]. First, we define a document as a body of text with some structural information, such as word, sentence, and paragraph boundaries. We ignore figures embedded in text, though we note that similar figures are also some evidence of plagiarism. We restrict our discussion to textual overlap. Formatting information is removed from the document. As a result of this process the document is represented in its canonical form. The conversion to the canonical form must be unambiguous. Otherwise two identical documents might have different canonical forms.

Brin et al. also have defined violation tests [BDG95]. Violation tests are decisions made by humans, which are subjective. Let us suppose we have two documents *d* and *r*, or a given document *d* and a set of documents *R*. Examples of violation tests are: Plagiarism(*d*,*r*) is true if *d* plagiarised *r*; Containment(*d*,*r*) holds if document *d* is contained in document *r*. We also can define violation tests for sets of documents: Plagiarism(*d*,*R*) holds if *d* plagiarised from any of the documents contained in *R* (possibly from more than one).

A copy detection mechanism implements operational tests that approximate these violation tests [BDG95]. For example, if an operational test discovers that 95% of the sentences in document *d* appear in document *r*, then this operational test approximates the Containment violation test. Operational tests are the result of some computer algorithms, thus they are objective.

These operational tests try to identify the overlap between documents and make a decision based on some threshold. To identify overlap, we have to decide what is the smallest unit of text that we consider an overlap. First, we have to define a text unit: Documents can be divided into some components, such as chapters, sections, subsections, paragraphs, sentences, words, letters; each of these types of division is a unit type, and particular instances of these types are units. A chunk is a contiguous sequence of units. These chunks may overlap. For example, let us suppose that we have a document ABCDEFGH where letters may represent any unit types. We may have a chunking strategy that produces the following chunks: AB, CD, EF, GH. Another strategy may consider ABCD, CDEF, and EFGH as chunks.

Chunks are usually not stored as text but a hash value is calculated for each chunk and an inverted file is used as an index. Because of the large number of possible chunks in a text, some algorithms keep only a certain number of chunks. The chunks that are kept are usually selected by using a fingerprinting algorithm, such as Rabin's fingerprint [BDG95]. Because of the complex procedures of registration, it is important to evaluate not only the query performance, that is how fast we can find documents that overlap with a given document, but we also have to analyse how long it takes to add a document to the registered set of documents.

When we have the index of chunks and we have to decide whether a given document (a query document) is plagiarised or it has "substantial overlap" with other documents, or it is contained in another document, we have to evaluate a decision function. This decision function can be a simple threshold value, or a more complex value, such as from the relative frequency model [GS95b], which is similar to techniques used in information retrieval ranking algorithms [BR99b].

In the following subsections we discuss how the aforementioned issues are addressed in different prototype systems. Subsection 2.4.1 compares different chunking strategies. Subsection 2.4.2 analyses different strategies to select chunks to be stored in the index. In Subsection 2.4.3 we introduce the different decision functions used in the approximation of violation tests. Subsection 2.4.4 considers many-to-many comparisons, and Subsection 2.4.5 deals with the problem of converting documents to a canonical form.

2.4.1 Chunking Strategies

Before we discuss different chunking strategies we have to decide on the smallest number of consecutive text units that we require in order to consider it a copy rather than an accidental overlap. One extreme is to consider that if the same word appears in two documents, it is an overlap. This would generate many false matches. This granularity is obviously too fine. The other extreme is to consider the whole document as a chunk. In this case we can only detect identical copies of documents and partial overlap detection is impossible. In this case we would miss almost all documents. This granularity is obviously too coarse. Different systems use different units, but knowing that the average word length in English language is approximately 5-6 characters, we can compare strategies based on words with strategies based on characters.

The Koala system [Hei96] uses 20 consonants, which translates into approximately 30-45 characters. The "shingling approach" [BGM97] considers 10 consecutive words, which are approximately 50-60 characters. The sif tool [Man94] uses 50 bytes (they use bytes rather than characters since their main focus is not only textual files but binary files, too). The SCAM system [GS96] analyses the effect of different chunk sizes: one word, five words, ten words, and sentences. Our algorithm, employed in the second stage of the comparison, uses 60 characters as a threshold value. Different research prototypes show that the threshold value should be somewhere between 40 and 60 characters. This value is an empirical value, which seems to work in other prototypes as well as in our system.

The chunks of a given document comprise the fingerprint of the document. As discussed earlier, these chunks are usually hashed for space-efficiency. In the ideal case we would index all possible α-length chunks of a document where α is the selected threshold. In a document of length l there are l-α-1 possible α-length chunks [Hei96]. Here we note that a suffix tree is a data structure that stores all possible suffixes of a given string. That includes all possible α-length chunks.

We could consider non-overlapping chunks of α-length character sequences. The problem with this approach is that adding an extra word or a character to the document would shift all boundaries. Thus two documents differing only by a word would produce two totally different fingerprints.

Manber [Man94] suggests two methods for selecting chunks. The first method identifies anchors in the text and builds a chunk around that anchor. By using anchors we avoid the problem of shifted boundaries. An arbitrary example of an anchor is the string 'acte'. Since anchors are independent from their actual position in the text, identical chunks would generate the same fingerprints provided they contain an anchor. We can use different anchors in such a way that they are uniformly distributed all over the document, although it is hard to find a good set of anchors that work well with different documents. The other method proposed by Manber considers all 50-byte chunks and uses a selection-algorithm to choose representative fingerprints.

The "shingling approach" of Broder et al. [BGM97] uses words as the building blocks of chunks. Every ten-word chunk is considered. As an example consider the following sentence:

Copy-detection methods use some kind of a hash-function to reduce space requirements.

Ten-word chunks generated from this "document" will include:

- copy-detection methods use some kind of a hash-function to reduce
- methods use some kind of a hash-function to reduce space
- use some kind of a hash-function to reduce space requirements

These ten-word chunks are called shingles. If we consider every ten-word chunk it will require too much space, so Broder et al. [BGM97] consider a selection strategy to select representative chunks of a document. Selection algorithms are discussed in Section 2.4.2.

Garcia-Molina et al. [GS96] compare different chunking strategies. For the finest granularity, word chunking is considered with some enhancements, such as eliminating stopwords. Another possibility is to use sentences as chunks to be indexed. The problem with sentences is that sentence boundaries are not always easy to detect. Garcia-Molina et al. tested their algorithm on informal texts, such as discussion group documents, and because of their informality they often lack punctuation. Sentence chunking also suffers when identifying partial sentence overlap. To overcome the boundary-shifting problem they introduce hashed breakpoint chunking. Instead of having strict positional criteria to detect chunk boundaries, such as every 10th word is a chunk boundary, they calculate a hash value on each word and whenever this hash value modulo k is 0, it is considered as a chunk boundary. They studied the performance of their prototype system with $k=10$, and $k=5$. The expected length of a chunk with this hashed breakpoint chunking is k words. As discussed earlier $k=10$ is closer to the empirically defined threshold value of 40-60 characters. The problem with this approach is that the chunk sizes may vary. We can easily have one-word chunks and also 15-20 word chunks. A more detailed analysis of this chunking strategy can be found in Chapter 7.

As the comparison shows [GS96], the finer the granularity, the more false positives are computed (beta error). The coarser the granularity, the more false negatives are computed (alpha error). This result illustrates the expectation that in the case of small chunks, accidental overlaps occur more frequently, and in the case of longer chunks, we miss some overlapping documents.

2.4.2 Selecting Chunks to Store

Heintze [Hei96] points out that the space-requirement for full fingerprinting, that is, considering each possible k-length chunk of a document, is too large. Some selection method must be put in place to keep a representative fingerprint of the possible chunks in a document. Full fingerprinting requires less space in the case of the "shingling approach" [BGM97], which makes use of the fact that natural language texts are comprised of words and we are not interested in overlaps that start in the middle of words. We also utilise this feature of natural language texts in our algorithm and we will show that significant space- and time-reduction can be achieved when we use suffix trees. If we only consider chunks starting at the beginning of words then the number of possible chunks (full fingerprinting) is in the order of the number of words rather than the number of characters. However, we have to note that the sif tool [Man94] targets a broader area of copy-detection and considers binary files as well as text files. In the case of a binary file we do not have such a natural chunk boundary as a word boundary in a text file.

The SCAM system [GS96] studies different chunking strategies and different selection strategies. In the case of word chunking, the size of the vocabulary is limited given the limited number of English words. Other chunking strategies like sentence chunking and any number of consecutive words or characters have virtually unlimited numbers of possible chunks. In the word chunking strategy they study three different approaches. The first approach considers every single word by only filtering away the words in the WAIS stoplist (391 words) [GS96]. They call this strategy "Word+0" chunking. "Word+100" and "word+300" strategies ignore the 100 and 300 most frequent words respectively as well as the stopwords in the WAIS list. In the case of sentence chunking and hashed breakpoint chunking SCAM keeps all possible chunks. However, chunks are non-overlapping, as opposed to overlapping chunks considered in other systems. When considering the whole document as a chunk we do not have to select chunks since we have only one chunk to represent the document. Garcia-Molina et al. [GS96] study the effect of chunk size on storage requirements, performance, and accuracy. Accuracy issues are discussed in Section 2.4.3. The storage requirement for the index is between 37 and 79MB for a 120MB dataset. If we consider the sizes of the vocabulary and the number of postings separately, we can conclude that the smaller the chunking primitive the less the size of the vocabulary, but the number of postings is higher. The number of postings significantly decreases if we eliminate more and more frequent words. Of course, we cannot go to the extreme of eliminating all words.

There are two main issues that affect the time required to build the index. One issue is locality in chunk referencing, which affects the number of buffer misses, and the actual size of the entire index. As the results show, chunking strategies that keep all chunks are too space consuming and in the case of such a large dataset as the Internet, it is prohibitive to have an index, which is 30-60% of the actual size of documents.

The sif tool [Man94] proposes two different methods for identifying 50-byte chunks. However the anchor-based strategy is basically discarded because it is hard to achieve a uniform distribution of chunks. The second strategy considers all 50-byte chunks and calculates a hash-value for each chunk. Here we note that the word "fingerprint" is used in two different ways in the literature. Some authors refer to fingerprint as a representative set of hash-values of a document. These hash values are calculated on chunks. On the other hand, some authors [Man94, BGM97] refer to the certain hash-value associated with the chunk when they talk about fingerprints. In this thesis we will use the former meaning, and the hash values calculated on different chunks are referred to as hash values. In the sif tool the hash value for the chunk starting at the first position, that is position 0, is:

$$F_0 = (t_0 \cdot p^{49} + t_1 \cdot p^{48} + \ldots + t_{49}) \bmod M \tag{2.1}$$

where p and M are constants. This formula is useful because we do not have to calculate the whole polynomial again if we want to get the value for position 1; we only have to use the following formula:

$$F_1 = (p \cdot F_0 + t_{50} - t_0 \cdot p^{49}) \bmod M \tag{2.2}$$

With this method hash values of consecutive chunks are easily calculated. Manber [Man94] proposes to keep only those chunks that have their last k bits 0. In the sif tool M=2^{30} , k=8, and a prime number for p are used. This means that, on average, every 256th chunk is considered. If we ignore this many chunks the accuracy of the system may suffer when only small portions of the files overlap.

The Koala system [Hei96] proposes to use a fixed size selective fingerprinting method, which has an equal number of chunks in a representative fingerprint, regardless of the size of the document. Although this method has the advantage that the size of the index is dependent on the number of documents rather than the overall size of documents, it raises some problems when file sizes in the dataset vary significantly. In the case of large documents, a larger proportion of chunks will be discarded. In their experiments, for documents stored in the repository, 100 chunks are selected. For probing, that is, when a document is compared against the database, 1000 chunks are selected in a way that they are a strict superset of the 100 chunks selected for registration. This ensures that exact copies of documents are detected by the system. This method raises some protection issues. If the selection algorithm is known, then a user need only make 100 changes and the system is circumvented. We can change this selection algorithm from time to time and have different 100 chunks selected out of 1000 considered at probing. This will not have any effect on probing, as it will still be a subset of the chunks selected for comparison. In the Koala system, the hash function creates a 28-bit value for each chunk. One option is to keep the chunks with the lowest *n* values as a selection strategy. To have more accurate results we can consider the frequency of chunks and keep the least frequent ones. The problem with this strategy is that relative frequency in a document is not always a good indicator of overall frequency. Heintze [Hei96] proposed a method that uses the first five letters of a chunk to compute the frequency.

Broder et al. [BGM97] consider 10-word chunks. 10-word chunks are ordered according to some function, e.g. a hash function. Two methods are proposed for the selection of a representative fingerprint. In the first method the *n* smallest elements of a permutation of the chunks are selected:

$$F(A) = MIN_n(\Pi(S(A))) \tag{2.3}$$

where *S(A)* is the set of shingles for document *A*, and *Π* is a permutation of these shingles. It has a fixed size and basically has the same problems as discussed in the case of the Koala system. Another method to select chunks is to have a number associated with each chunk, e.g. a hash function and only numbers, that are divisible by a carefully selected number *m*, are kept:

$$V(A) = MOD_m(\Pi(S(A))) \tag{2.4}$$

This is very similar to the method discussed in the case of the Koala system. However Broder et al. [BGM97] also introduce a method to limit the number of chunks in *V(A)*. For example, consider a document of size between $100{*}2^i$ and $100{*}2^{i+1}$:

$$V_i(A) = MOD_{2^i}(\Pi(S(A))) \tag{2.5}$$

The expected size of $V_i(A)$ is always between 50 and 100. Since $V_{i+1}(A)$ can easily be computed from $V_i(A)$, we can calculate $V_i(A)$ and $V_i(B)$, in case of document *A* and *B*, based on the size of the longer document. The system uses a 40-bit hash function based on Rabin fingerprints [Rab81] and the latter method for chunk selection with *m*=25, meaning that only every 25th chunk is kept.

2.4.3 Decision Function

As discussed earlier in this chapter, our goal is to approximate such violation tests as resemblance, plagiarism, and containment. For such an approximation we need to associate a decision function with our operational tests. Since our operational tests determine how many chunks overlap in a given document with chunks in other documents, the most common decision function is a threshold value. If *V(A)* is the fingerprint of document *A*, that is, a representative set of chunks in document *A*, and the same holds for *B*, then our resemblance measure is:

$$\text{Resemblance}(A, B) = \frac{|V(A)\ \text{I}\ V(B)|}{|V(A)\ \text{Y}\ V(B)|} \tag{2.6}$$

$|V(A)\ \text{I}\ V(B)|$ is the number of chunks appearing in both document *A* and *B*, and $|V(A)\ \text{Y}\ V(B)|$ is the number of chunks appearing in either document *A* or document *B*.

The containment measure is:

$$\text{Containment}(A, B) = \frac{|V(A)\ \text{I}\ V(B)|}{|V(A)|} \tag{2.7}$$

This measure only holds if *V(A)* and *V(B)* are not fixed size fingerprints but are dependent on the document size, for example in formula (2.4). To generalise this idea we may consider all registered documents as one big document and the same measures apply. Different applications may define different thresholds. For example, the plagiarism decision function may be a 10% resemblance value applied for the whole registered document set. In a file system application where we want to find similar documents, the threshold may be higher, say 80%.

The "shingling approach" [BGM97] and the sif tool [Man94] use 50% as the resemblance threshold. Their main target is to create clusters of similar files and they have found that this threshold meets their needs. Heintze [Hei96] does not give a threshold, but rather shows the resemblance values for the test document set. Garcia-Molina et al. [GS96] study three different scenarios and set three different thresholds. They study netnews articles, so they set the three different scenarios as follows:

- **Exact Copies**: identical articles.
- **High Overlap Replies**: response to a given article with most of the original document contained in the response.
- **Some Overlap Replies**: similar to above but only small portion of original document is contained in the response.

For different chunking schemes they set different thresholds. For word chunking these thresholds are 100%, 66%, 33%, while in the case of sentence and modulo schemes they are 100%, 50%, 5%. The chosen values reveal that in the case of word chunking accidental overlap is much more likely than in the case of longer chunks. For word-based chunking schemes SCAM proposes a new overlap measure, which is based on the cosine measure [BR99a] widely used in information retrieval. The problem with the cosine measure is that it only works well if the word frequencies are of similar magnitude. To correct this problem Garcia-Molina et al. introduce the notion of a closeness set *c(R,S)* for documents *R* and *S*. Words contained in the closeness set must satisfy the following condition:

$$\varepsilon - \left(\frac{F_i(R)}{F_i(S)} + \frac{F_i(S)}{F_i(R)} \right) > 0 \qquad (2.8)$$

where $F_i(D)$ is the relative frequency of the *i*-th word in document *D*, and ε is a tunable parameter. A small ε will decrease false positives but it will also miss documents with minor overlap. The higher the ε value is, the more false positives reported. The value of ε is set to 2.5 in the SCAM system since they have found it to work well in practice. After defining the closeness set they define the subset measure as:

$$subset(R,S) = \frac{\sum_{w_i \in c(R,S)} \alpha_i^2 \cdot F_i(R) \cdot F_i(S)}{\sum_{i=1}^{N} \alpha_i^2 \cdot F_i^2(R)} \qquad (2.9)$$

In this formula only words in the closeness set are considered. Then the similarity measure is:

$$sim(R,S) = \max\{subset(R,S), subset(S,R)\} \qquad (2.10)$$

These corrections are necessary, as simply having similar numbers of certain words in two documents does not necessarily mean that those documents are similar.

Documents reported by these decision functions must be checked by a human to decide whether these decision functions are good approximations of the violation tests they try to approximate.

2.4.4 Clustering Documents

Two of the above-discussed systems target document- or more generally file-clustering. The additional problem of clustering is how to compare many documents to many documents. The "shingling approach" [BGM97] and the sif tool [Man94] use the same approach for this problem. They calculate the hash values for the selected chunks of each document and then use the "divide, merge, sort" approach [BGM97] to find matching chunks. Manber [Man94] also addresses the problem of how to represent these clusters because document resemblance is not necessarily a transitive relation. That is, while document *A* resembles document *B*, and document *B* resembles document *C*, document *A* does not necessarily resemble document *C*. This representation problem is outside the scope of this thesis.

2.4.5 Canonical Forms of Documents

Converting a document to its canonical form is the very first step of every copy-detection scheme. We deferred the discussion of canonical forms to this section because different copy-detection schemes have different requirements in order for their algorithm to work.

The first step is to convert the document to pure text format. The sif tool [Man94] does not apply this step because they deal with binary files as well as with text files, and binary files, of course, cannot be converted to ASCII format in a meaningful way. For transparent handling of files they do not convert text files, either.

Converting documents to pure text is not a trivial process. Heintze [Hei96] discusses the problem of converting PostScript files to pure text and finds that standard tools are usually unreliable and as tools evolve their output changes, so further processing of the text is required. Heintze found that sometimes even word boundaries are not preserved and finds that it is easier to consider only the consonants of a text rather than the entire sequence of characters. Of course, those schemes that use chunking primitives based on word or sentence boundaries must detect those boundaries, meaning that the conversion process must preserve word boundaries. The most common trivial conversion steps, such as converting to lowercase/uppercase and ignoring multiple whitespaces, are used in all systems. With the exception of sentence-based chunking schemes, we also can ignore punctuation and convert them to, for example, space characters.

Word-based chunking schemes also ignore certain stopwords. Stopwords are the most common words such as 'and', 'is', 'the', etc. that are not likely to contribute to the semantic meaning of the text. Some schemes also recognise that there are common chunks, which are characteristic of a given document set just because they are on the same topic. Most copy-detection schemes ignore very frequent chunks, like the "word+k" schemes in the SCAM system [GS96]. The Koala [Hei96] system ignores those chunks that appear in more than four documents but only up to 10 chunks. Otherwise, by registering the same document five times the sixth copy of the same document would not generate any match with the first five. The reasons why we might have common chunks in unrelated documents are discussed by Heintze [Hei96] and Broder et al. [BGM97]:

- HTML comment tags generated by a HTML editor
- Shared header or footer in HTML files
- Mechanically generated pages with artificially different URLs and internal links

- The beginning parts of documents contain "problematic strings", such as addresses, names, funding agencies, acknowledgements, etc.

The last problem is addressed by ignoring the first 1000 characters of each document in the Koala system [Hei96] and the results confirm the assumption that this part of the text does not contribute much to the semantics of the document.

A canonical form of the document is the result of the above mentioned conversion steps. The canonical form may vary from system to system since the text must be ready to be processed by a certain algorithm. The conversion algorithm used in our system addresses most of the issues discussed here and is presented in Chapter 7.

2.5 Exact String Matching Algorithms

The previous sections of this chapter demonstrate that the systems proposed so far have drawbacks, which need to be addressed. One problem with the methods discussed is the potential for false positives. Since each chunk extracted from the text is hashed to a binary value, it is possible that two different chunks may produce the same hash value even if the original chunks are different. Not only does this stem from imperfect hashing functions, but also the mere amount of chunks to be hashed means that we may end up with more chunks than available hash values. Of course, choosing larger hash values that accommodate more chunks would increase the storage capacity required. We have to find a trade-off that allows storing large amounts of chunks without sacrificing accuracy.

Another problem is finding the exact overlap between the documents. If we only store a fingerprint of a document we need a second stage that identifies the exact positions of identical chunks in the documents. With each chunk we could store their starting positions but that also would increase the storage capacity required.

We propose a second stage that compares documents, which have been identified to be similar in the first stage. It means that we can use even smaller hash values, thus less space, as any false positives will be filtered out in the second stage. Applying a second stage also means that positions of chunks need not be stored in the database supporting the first stage.

Let us formalise the problem we want to solve in the second stage. Let us suppose that the two documents we want to compare are *G* and *C* (genuine document and candidate document). We want to find each chunk in *G* with a minimum length of *t* that overlaps with any chunk in *C*. More formally we can say that given two strings *G* and *C* with the length of *m* and *n* respectively, we have to divide *G* and *C* into substrings $G=t_1s_1t_2s_2t_3s_3...t_ks_kt_{k+1}$ and $C=p_1q_1p_2q_2p_3q_3...p_rq_rp_{r+1}$ where for each s_i there is a *j* such that $s_i=q_j$ and $\Sigma(|s_i|)$ is maximal.

This section analyses different exact string-matching problems and algorithms that may help to solve the stated problem. A comprehensive overview of string-matching algorithms is given by Gusfield [Gus97] and Aho [Aho90]. We will show that the problem is most efficiently solved by using suffix trees. Thus this data structure is discussed in detail in this section.

2.5.1 Basic String-Matching Problems

One of the basic string-matching problems is the exact matching problem: "given a string *P* called the pattern and a longer string *T* called the text, the exact matching problem is to find all occurrences, if any, of pattern *P* in *T*" [Gus97].

There are three basic approaches to this problem, which are referred to as the Boyer-Moore algorithm [BM77], Knuth-Morris-Pratt algorithm [KMP77], and the Aho-Corasick algorithm [AC75]. All of them – with certain extensions – are linear-time algorithms. The Aho-Corasick algorithm generalizes the Knuth-Morris-Pratt algorithm to handle multiple patterns but the Boyer-Moore algorithm has the best running time in practice.

The naive method would align the beginning of the pattern with each possible character of the text and compare characters one by one until a mismatch is found or all characters in the pattern have been matched. Obviously this algorithm runs in O(*nm*) time in the worst case (*n* and *m* are the length of *T* and *P* respectively).

All of the above mentioned algorithms attempt to solve the problem by applying so called shift-rules, which shift the pattern by more than one character if a mismatch is found. *Figure 2.2* illustrates the concept of shift-rules.

Different variants of the above algorithms have widely been published in the literature [Hor80, HS91, Smi91, Smi94, Sun90, AG86]. Other approaches referred to as arithmetic methods have also been developed [BG92, FSK82, FP74].

```
      (a)                       (b)
T: xabxyabxyabxz      T: xabxyabxyabxz
P: abxyabxz           P: abxyabxz
    abxyabxz               abxyabxz
     abxyabxz                   abxyabxz
```

Figure 2.2. Shifting of the Pattern. (a) Naive Algorithm (b) Shift-Rule

The applicability of algorithms solving the exact matching problem is quite limited in our overlap problem. We may define each possible substring of *G* as the pattern *P* and apply the algorithms above. Let the length of *G* be *n* and the length of *C* be *m*. There are $\frac{n(n+1)}{2}$ possible substrings of G and the comparison of each pattern runs in O(*m*) time in the worst case. Thus the worst case running time of this approach is $O(mn^2)$. Even if we consider all suffixes of *G* rather than all substrings of *G* the running time is still *O(nm)*. A suffix of a string *G* is a substring whose last character is the last character of string *G*. We will show that *O(n+m)* running time is achievable by other methods. This approach also suffers when we have to compare *G* to a set of documents C_1, C_2, ..., C_k. In this case we would have to run *k* instances of the above-described algorithm. Thus the running time would be *k* times more.

Obviously the algorithms designed to solve the basic exact matching problems cannot efficiently solve our overlap problem. We have presented the exact matching problems and the algorithms solving the problem to form a base for other problems and algorithms as well as to show that the overlap problem is not trivial to solve.

2.5.2 Longest Common Subsequence of Two Strings

This is the problem of determining how "close" two files are to each other. In order to demonstrate the problem and algorithms, below we define the terms subsequence, common subsequence, and longest common subsequence as they are defined in [Aho90].

A subsequence of a string *T* is a sequence of characters obtained by deleting zero or more characters from *T*. A common subsequence of two strings *P* and *T* is a string that is a subsequence of both *P* and *T*, and a longest common subsequence is a common subsequence that is of greatest length. Here we note that in some literature the term "subsequence" and "substring" are used interchangeably. We follow the definitions used in most of the literature and differentiate between the two terms. The difference is that a substring is a consecutive range of characters while a subsequence is generated by deleting some characters and concatenating the remaining.

As an example consider the longest common subsequence of *P='abcbba'* and *T='cbabbcba'*. The longest common subsequence of these two strings is *'abcba'*. The longest common subsequence problem is closely related to the edit distance problem [Aho90], which defines how many edit operations are needed to convert one of the strings to the other. *Figure 2.3* shows how the two strings may be aligned, and from that, edit operations are self-explanatory.

```
  ab cbba
  || || |
cbabbcb a
```

Figure 2.3. Alignment of P and T

We can solve the problem with a recursive algorithm [EGGI92]. The running time of the recursive algorithm is O(2^n) but it can be reduced to O(nm) by dynamic programming [Hir77], where n and m are the lengths of the two strings.

Myers [Mye86] and Ukkonen [Ukk85] proposed an algorithm which treats the problem as a cheapest-cost path in a graph. A refinement of Myers' algorithm uses suffix trees and runs in $O(n \log n + d^2)$ time, where d is the edit distance between the two strings [Mye88]. According to Epstein et al. [EGGI92] the fastest algorithm so far runs in O($n$log s + c loglog min$(c,mn/c)$) time where c is the number of "corners" in the matrix built during the algorithm and $s=m+n$. This is much better than O(mn) but still not linear and running time can be very large in the case of large documents.

Now let us discuss how this algorithm suits our needs. The problem is similar to our overlap problem but it considers the order of characters and tries to find overlaps in the original order of characters/chunks. As an example let us consider the longest common subsequence of the following two strings *P="this is a text to find in any of the documents submitted from now on"* and *T="that is what we need in the final text and this is a text to find"*. The longest common subsequence of these two strings is *"th is ate nd in f te dts st o n "*, which is not what we are looking for. We need an algorithm which reports *"this is a text to find"* as an overlapping chunk. Using words as symbols is not a solution either because it would look for words in their original order e.g. having two strings $P=S_1S_2$ and $T=S_2S_1$ these algorithm might find either S_1 as a matching chunk or S_2 but definitely not both of them. S_1 and S_2 are substrings of P and T. The main problem is that the longest common subsequence of two strings takes the order of characters (or words or chunks) into consideration.

2.5.3 The Longest Common Substring of Two Strings

We have discussed the difference between the notions of "subsequence" and "substring" in the previous section. We will argue in this section that problems concerned with substrings are more closely related to our overlap problem than subsequence problems.

A substring of a string is a consecutive sequence of characters. Let $S[i]$ denote the character at position i in S (numbering of positions starts at 0). $P=S[i..j]$ is a substring of S if $S[i+k]=P[k]$ for each k $0 \leq k \leq j-i$.

The longest common substring R of two strings P and T is the substring of both P and T that is of greatest length. As an example the longest common substring of *P="this is a text to find in any of the documents submitted from now on"* and *T="that is what we need in the final text and this is a text to find"* is *"this is a text to find"*. These are the same example strings used in the previous subsection and one can see that the longest common subsequence algorithm failed to deliver the expected result while the longest common substring algorithm produced the result we needed.

The longest common substring problem can be solved by building a generalized suffix tree for both P and T, and then finding the deepest node in the tree. When we have a closer look at the algorithm we can realize that this algorithm reveals all matches between the two documents. Since suffix tree related problems and solutions form the basis of this thesis we leave a more detailed discussion of suffix trees and the longest common substring algorithm to the next section. This algorithm runs in linear time, that is O($n+m$). We also will show that a generalization of the problem will allow us to compare one document to many documents in O($m+n_1+n_2+...+n_k$) time where m is the length of the original document while $n_1,n_2,...,n_k$ are the lengths of the candidate documents. It is interesting to note here that in 1970 Don Knuth conjectured that a linear time algorithm for the longest common substring problem would be impossible [Gus97].

2.6 Suffix Trees

In this section we introduce suffix trees and show their applicability to our problem area. In this section we only deal with suffix trees at a high, theoretical level, abstracted from the actual physical representation. Physical representations of suffix trees are presented in Chapter 3. Suffix trees are versatile data structures that can be used to solve many string-matching problems. Apostolico et al. wrote a paper titled "The Myriad Virtues of Subword Trees" [AG86], which shows how versatile this structure is. Gusfield [Gus97] describes numerous problems that can be solved by suffix trees including exact set matching, longest common substring, recognizing DNA contamination, common substrings of more than two strings, all pairs suffix-prefix matching, etc.

Variants of the suffix tree structure are also referred to as position trees [AHU74][MR75], complete inverted files [BBE$^+$84][BBE$^+$85], subword trees [AG86], and PATRICIA trees [Mor86]. Section 2.6.1 introduces a suffix tree at a high-level and then we show how different algorithms may run on the structure.

2.6.1 Suffix Tree at a High Level

There are different definitions and even different names for the index structure we call suffix tree. In this subsection we give the definition as it is given in [Gus97]:

"A suffix tree *T* for an *m*-character string *S* is a rooted directed tree with exactly *m* leaves numbered *0* to *m-1*. Each internal node, other than the root, has at least two children and each edge is labelled with a nonempty substring of *S*. No two edges out of a node can have edge-labels beginning with the same character. The key feature of the suffix tree is that for any leaf *i*, the concatenation of the edge-labels on the path from the root to leaf *i* exactly spells out the suffix of *S* starting at position *i*. That is, it spells out *S[i..m-1]*"[1].

This definition does not guarantee that a suffix tree for a given string exists. In order to guarantee the existence and uniqueness of the suffix tree we have to add a unique termination symbol to the string. We will refer to the unique termination symbol by *'$'*. From here on we assume that this termination symbol is part of the string, thus whenever we refer to *S* we refer to a string terminated by the unique termination symbol. As an example, *Figure 2.4* shows the suffix tree of *S='abcdabdbcdabb$'*.

[1] This definition is taken from Gusfiled's book [Gus97], but the indices are modified because we index strings from *0* to *m-1*.

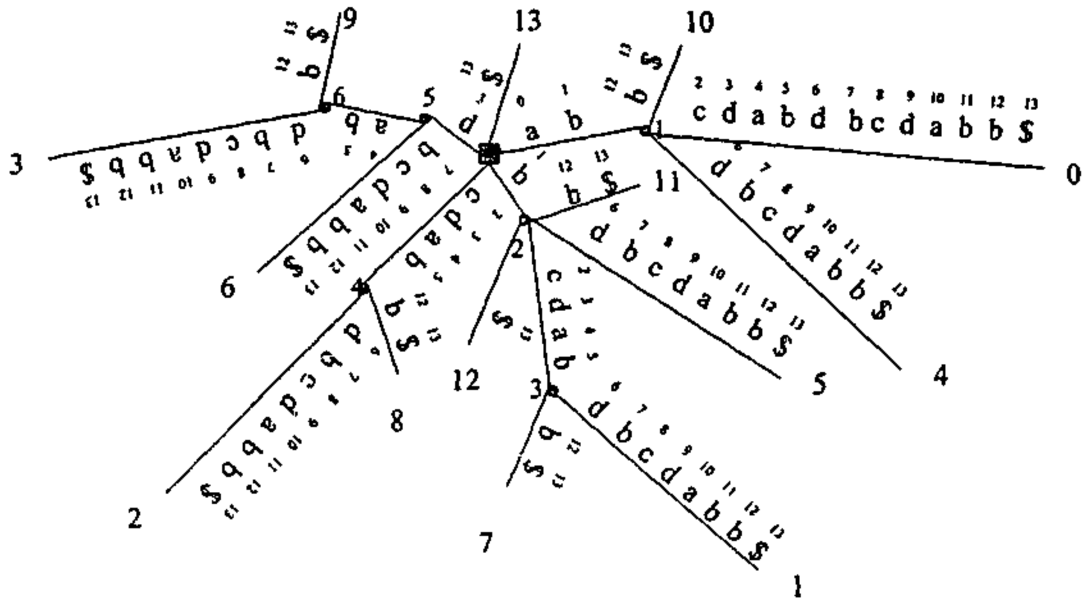


Figure 2.4. The Suffix Tree of *S='abcdabdbcdabb$'*

We define some additional notions that we will use throughout this thesis:

The *label of a path* is the concatenation of the characters on the edges from the root to the end of the path. For example, the label of the path to *node 3* is *'bcdab'*. We will also use the term *label of a node*, which refers to the label of the path from the root to the given node.

The *depth of a node* is the number of characters in the path from the root to the given node. The depth of *node 3* in our example tree is 5.

Before introducing the algorithms that run on a suffix tree, we show how simply the exact matching problem may be solved by using a suffix tree. Let us assume that we search for the pattern *P='ab'* in *T='abcdabdbcdabb$'*. We build a suffix tree for *T*, which can be done in linear time (for proof see Section 2.6.2) and then starting from the root we follow the path labelled by *'ab'*. We get to *node 1* and we check the leaves under *node 1*, which will give us all the positions where *'ab'* appears: 0, 4, 10.

2.6.2 Suffix Tree Construction in Linear Time

A suffix tree stores all the suffixes, consequently, all substrings of a given string. Any substring of that string can be retrieved in time proportional to the length of the substring. This feature makes the use of suffix trees very appealing but the question is what are the time- and space-constraints this structure exhibits.

Firstly, the suffix tree as presented in the Section 2.6.1 cannot be stored in space proportional to its length. Since the overall length of all suffixes of an n-character string is $\frac{n(n+1)}{2}$, it means that it would require storage proportional to n^2 rather than n. To overcome this problem we can store only the start and end positions of each edge and edge labels can be retrieved from the string if required. Of course, this representation assumes that the string is also stored in its entirety. The number of edges in a suffix tree is proportional to n [Gus97], which allows linear-space storage of a suffix tree. Not only is linear-space storage important because of space-consumption but also an algorithm building a structure requiring non-linear space cannot theoretically run in linear time. Of course, the linear-space requirement does not guarantee a linear-time construction algorithm but fortunately there have been linear-time construction algorithms developed [Gus97]. The linear-space representation of our example suffix tree is depicted in *Figure 2.5.*

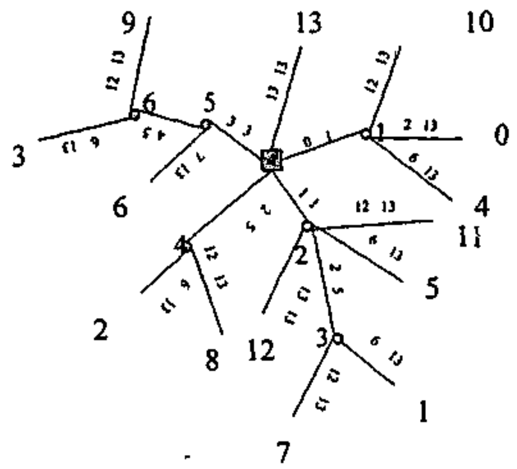


Figure 2.5. Linear-Space Representation of a Suffix Tree

The naive algorithm would insert each suffix one by one. Insertion of a suffix takes time proportional to its length, thus the running time of the naive algorithm is O(n^2). There are three fundamental linear-time construction algorithms frequently cited in the literature: Weiner's algorithm [Wei73], McCreight's algorithm [McC76], and Ukkonen's algorithm [Ukk95]. Giegerich et al. give a unifying view of these algorithms [GK97]. Other alternatives of these algorithms have also been studied in the literature (see [CS85] as an example).

In the following subsections we introduce all three algorithms and compare them. The detailed description of the algorithms along with the proofs can be found in Weiner's [Wei73], McCreight's [McC76] and Ukkonen's [Ukk95] papers.

2.6.2.1 Weiner's Suffix Tree Construction Algorithm

Weiner's suffix tree construction algorithm [Wei73] constructs the tree by inserting the suffixes from right to left. Firstly, *S[n-1]* is inserted then *S[n-2..n-1]*, and so on. When *S[0..n-1]* is inserted the suffix tree construction is over and the tree has been created. In Step i (i runs from n-*1* down to *0*) when *S[i+1..n-1]* has already been inserted we have to find the longest common prefix of *S[i..n-1]* and the suffixes already inserted into the tree. Let us denote this prefix by Head(i). Head(i) is obviously already in the tree, thus we insert the remaining part of the string. That is, we have to remove Head(i) from the beginning of *S[i..n-1]* and create a leaf with this label running out of the position of Head(i). The position of Head(i) may or may not be at a node. If it is not at a node, a new node needs to be created and the current edge must be split into two.

Finding Head(i) naively would not allow linear time construction of the tree. Weiner introduces two vectors at each node, which aid in finding Head(i): the *indicator vector I* and the *link vector L*. Both vectors are of length equal to the size of the alphabet (including the termination symbol '*$*'). The indicator vector is a bit vector allowing values of 0 and 1 at each position, while the link vector contains pointers to other nodes. Let $I_v(x)$ specify the entry of the indicator vector at node v indexed by character x. At any given stage of the algorithm $I_v(x)=1$ if and only if there is a path from the root in T_{i+1} labelled $x\alpha$ if the path label of v is α. T_j, in general, denotes the jth character of a string (indexing starts from 0).

Similarly $L_v(x)$ is the link vector at node v indexed by character x. At any given stage of the algorithm $L_v(x)$ points to node $\underline{v}$ if and only if $\underline{v}$ has path label $x\alpha$ if the path label of v is α.

Step $i+1$ finishes with the insertion of leaf $i+1$. In the next step the algorithm walks up from the leaf until it finds the first node v with $I_v(S[i])=1$. It then walks further up until it finds a node $\underline{v}$ with $L_{\underline{v}}(S[i])$ not null. v and $\underline{v}$ may be the same node. We jump to the node pointed by $L_{\underline{v}}(S[i])$ and find the route starting with the first character of the path between $\underline{v}$ and v. Then we traverse down as many characters as there are on the path between $\underline{v}$ and v and the insertion of leaf i must be done at this position. It is also possible that we reach the root node without finding either v or $\underline{v}$ or both of them. These degenerate cases can be handled in the algorithm by properly setting the vectors at the root node.

The running time of the algorithm is proportional to the number of nodes visited during the algorithm. It can be proven that both traversing down and up are limited by *2n*, thus the algorithm runs in linear time. For a correct time analysis of the algorithm see [Gus97] or [Wei73]. The pseudo code of Weiner's algorithm is depicted in *Figure 2.6.*

```
For i:=n-1 DownTo 0
  1. Start at leaf i+1 and walk toward the root
     searching for the first node v where $I_v(S[i])=1$.
  2. If the root is reached and $I_v(S[i])=0$ Head(i)
     ends at the root. Go to Step 4.
  3. Continue walking upward until the first node with
     $L_{\underline{v1}}(S[i])$ not null is found.
      3a. If the root is reached and $L_{v1}(S[i])$ is null
      then $t_i$ is the number of characters between the
      root and v. Search for the edge that starts with
      S[i] and walk down $t_i+1$ characters.
      3b. If not 3a. Let $L_{v1}(S[i])$ point to $v_2$. Walk
      down as many characters as the number of
      characters between $v_1$ and v on the path from $v_2$
      starting with the first character on the path
      between $v_1$ and v. That is the position of
      Head(i).
  4. If no node exists at Head(i) create one and from
     the node at Head(i) create a new leaf numbered i.
  5. Update the indicator and link vectors.
End For
```

Figure 2.6. Pseudo Code of Weiner's Algorithm

The updates to the indicator and link vectors do not violate the linear-time bound of the algorithm because in each step only the nodes visited may need to be updated. The content of the link vector and indicator vector of each node is shown in *Table 2.1.*

Node	Indicator Vector				Link Vector			
	a	B	c	d	a	b	c	d
1	0	0	0	1	0	0	0	6
2	1	1	0	1	1	0	0	0
3	0	1	1	0	0	0	0	0
4	1	0	0	1	0	0	0	0
5	0	1	0	0	0	4	0	0
6	0	0	1	0	0	0	5	0

Table 2.1. Indicator and Link Vector Values of the Example Tree

2.6.2.3 McCreight's Suffix Tree Construction Algorithm

McCreight's algorithm [McC76] inserts suffixes starting from the longest suffix *S[0..n-1]* to the shortest one *S[n-1..n-1]*. Again a naive algorithm would run in O(n^2) time, thus we need to find a way to locate *Head(i)* once *Head(i-1)* has been located. In the case of this algorithm Head(*i*) is the longest common prefix of suffix *i* (*S[i..n-1]*) and any suffix between *0* and *i-1*. Once Head(*i*) is located we can insert the remaining characters in the tree.

The algorithm makes use of the so called suffix links between nodes. Suffix links have a reverse role compared to the links established in Weiner's algorithm. If the path label of a node v is $a\beta$ where a is a single character and β is a sequence of characters (possibly empty) then the suffix link of node v points to a node labelled β.

Each step is divided into the following three substeps.

Substep A finds the deepest node on the path to Head(*i-1*) that already has a suffix link. Let us denote this node by v_1. Head(*i-1*) then can be divided into three substrings: $\chi\alpha\beta$. $\chi\alpha$ is the path label of the node v_1. If no such node is found $\chi\alpha$ is the empty string otherwise χ is the first character and α is the rest of the characters. If we follow the suffix link of v_1 we get to node v_2, whose label is α.

In **Substep B** we do a "rescanning" of characters β from node v_2. It is certain that β can be read from this position because Head(*i-1*)=$\chi\alpha\beta$, that is $\chi\alpha\beta$ is part of T_{i-1} and T_{i-2}, which means that $\alpha\beta$ is part of T_{i-1}. Note that the rescanning needs to analyse only the first characters of the edges on the path of β because we know that β is part of the tree. It means that the number of steps required by rescanning is proportional to the intermediate nodes visited during the traversal of β. Here we have to create a new node unless there is already a node at this position. Let us denote this node by *d*. The suffix link of the node labelled Head(*i-1*) will point to this node.

Substep C is called the "scanning" phase. Let us denote the remaining part of the characters in Head(*i*) by *γ*. The length of *γ* is not known beforehand, thus we have to analyse each character down the subtree until we "fall out" of the tree. At the position where we fall out, we create a new node and the remaining part of suffix *i* will label the leaf created from the new node. Note that the introduction of the unique termination symbol ensures that the matching will fall out at some stage.

It can be proven that the overall number of intermediate nodes encountered in the rescanning steps of the entire algorithm is proportional to *n*. Similarly, the overall number of characters examined in the "scanning" phase is proportional to *n* for the entire algorithm. Other operations in each step require only constant time, thus the overall running time of McCreight's algorithm is O(*n*).

The pseudo code in *Figure 2.7* summarizes the steps we have to follow in McCreight's algorithm.

```
For i=0 To n-1
  A. Walk up to the deepest node with a suffix link on
     the path to the current node. Jump to the node
     pointed to by the suffix link.
  B. Traverse down the remainder of Head(i-1) from
     this node by analysing only the start of each
     edge.
  C. Traverse down the remainder of suffix i until the
     matching falls out. Create a new node here with
     the remaining of characters labelling leaf i.
End For
```

Figure 2.7. The Pseudo Code of McCreight's Algorithm

Figure 2.8 depicts our example string with the suffix links. The dashed arrows represent the suffix links.

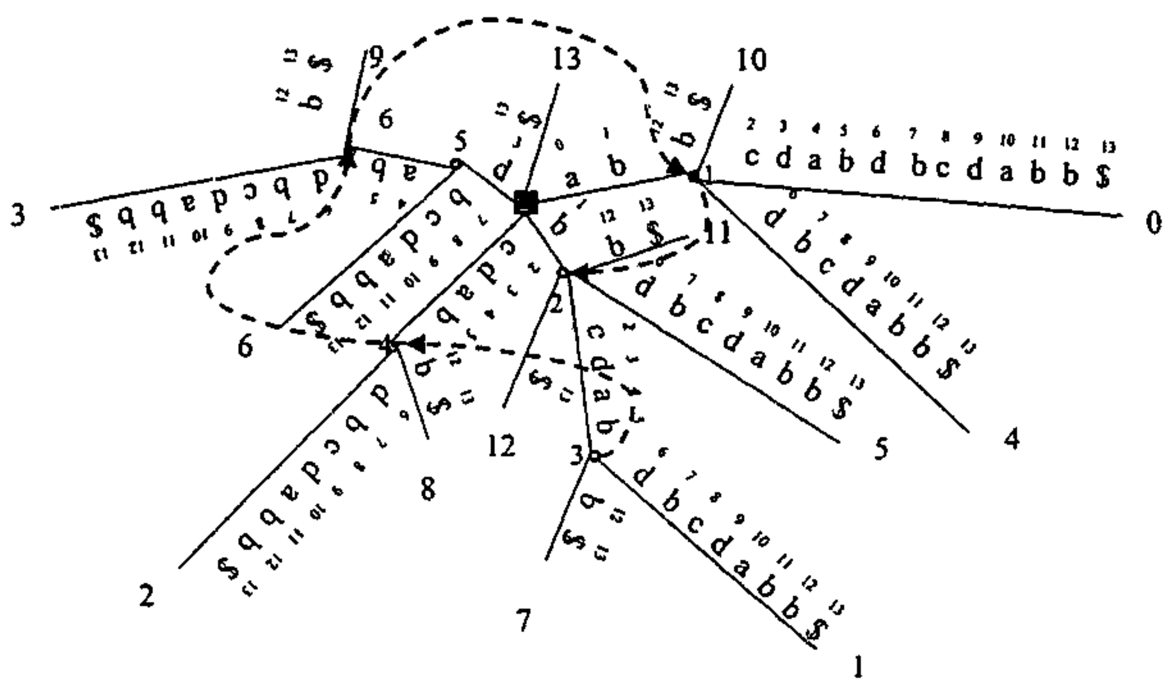


Figure 2.8. A Suffix Tree with Suffix Links

2.6.2.3 Ukkonen's Suffix Tree Construction Algorithm

Of the three algorithms discussed in this thesis, Ukkonen's is the latest, published in 1995 [Ukk95]. It is the most elegant algorithm and it shares some characteristics with the matching statistics algorithm [CL94] though this latter one was published earlier. This is the algorithm that we have used in our prototype system for suffix tree construction. This algorithm also forms the basis of the construction algorithms used for building the proposed alternative representations described in Chapters 4, 5, and 6.

Ukkonen's algorithm constructs an implicit suffix tree T_i in each phase *i*. An implicit suffix tree is a tree obtained from the suffix tree of *S* by removing every copy of the unique termination symbol. Note that in an implicit suffix tree there is no one-to-one relationship between leaves and suffixes of a string. T_i is the implicit suffix tree of string *S[0..i]*. The true suffix tree of the whole string *S* is T_{n-1} provided that *S[n-1]* is the unique termination symbol. The naive algorithm would create the suffix tree in O(n^3) time. Of course, this time bound will shrink to O(*n*) when we introduce the various shortcuts applied in each phase.

Each phase is divided into extensions. In extension *j* of phase *i* suffix *j* of *S[0..i]* is inserted into the tree (*S[j..i]*). By the end of phase *i*, the implicit suffix tree, T_i is created. Fortunately, not all extensions must explicitly be done. Let *β* denote the string *S[j..i-1]*. We have to locate *β* and then extend it by *S[i]*. Note that *β* is definitely in the tree since *S[j..i-1]* is a suffix of T_{i-1} and T_i is created from T_{i-1}. The following three extension rules may be defined [Gus97]:

Rule 1. Path β ends at a leaf. In this case *S[i]* must be added to the end of the leaf.

Rule 2. No path from the end of β starts with *S[i]* but at least one labelled path continues from the end of β. A new leaf edge with label *(i,i)* must be created and if necessary a new node, too. The index of the newly created leaf will be *j*.

Rule 3. Some path from the end of β starts with *S[i]*. In this case we do not have to do anything because an implicit suffix tree is allowed to have a suffix that ends in the middle of an edge.

The key in the algorithm is how to find β. With the help of suffix links β can be found in constant time on average for the entire algorithm. In the previous extension we inserted *S[i]* at the end of $a\beta$ where a denotes the single character at position *j-1*: *a=S[j-1]*. We find the deepest node with a suffix link on the path to $a\beta$. Let us denote this node by v_1 and the node where its suffix link points to by v_2. v_2 is definitely on the path to β and β can be reached similarly to the rescanning step of McCreight's algorithm. These steps are referred to as the skip/count steps and it has been proven that the overall number of skip/count steps is proportional to n for the entire algorithm. If β can be found in constant time the entire algorithm runs in O(n^2) time.

There are a few more shortcuts that reduce this time bound to the required linear time. In any phase when **Rule 3** applies we know that it will apply to all further extensions of the same phase. It is true because **Rule 3** applies if *S[j..i]* is found in the implicit suffix tree T_{i-1}. If *S[j..i]* is the prefix of some suffix of *S[j..i-1]* we know that all strings from *S[j+1..i]* through *S[i..i]* are also prefixes of some suffix of *S[j..i-1]*. Since suffix links only need to be updated when a new node is created we can stop a phase as soon as **Rule 3** applies. We say that all the extensions after **Rule 3** is applied are done implicitly as opposed to extensions that we have to do explicitly by inserting a leaf with label *S[i..i]*.

Our next observation is that once a leaf is created in any stage of the algorithm execution, it will remain a leaf for the entire algorithm run duration. Using this observation we can label the newly created leaves by *(i,e)* instead of *(i,i)* where *e* is the current end pointer. This shortcut will implicitly do all the extensions where **Rule** 1 applies. If we know the length of the string in advance we may label the leaf by *(i,n-1)* because the algorithm would never try to examine a character beyond *e*. However, as we will point out in the next subsection the length of the string is not always known in advance.

Explicit extensions require constant time and at most one explicit extension is shared between two consecutive phases, thus the running time of the algorithm is O(n). The pseudo code of Ukkonen's algorithm is depicted in *Figure 2.9*.

```
Add root node with an edge labelled (0,e)
$j_i$ := 1
for i:=1 to n-1 do
 e := i
 Starting at $j_i$ compute successive extensions until
     Rule 3 applies (j*)
 Set $j_{i+1}$ to j*-1 to prepare for the next phase
end for
```

Figure 2.9. The Pseudo Code of Ukkonen's Algorithm

2.6.2.4 Comparison of the Three Fundamental Suffix Tree Construction Algorithms

Both McCreight's and Ukkonen's algorithms use suffix links to speed up the search between phases. Weiner's algorithm uses a link vector and an indicator vector. The length of both vectors is the number of characters in the alphabet (let us denote this value by m). It means that for each node we need to store *m*size_of_pointer+m* bits where *size_of_pointer* is the size of node pointers in bits. In case of using suffix links we only need *size_of_pointer* extra bits. The suffix link is the reverse of a link depicted in a link vector. The suffix link representation allows using less space because each node may only point to one node via a suffix link but each node may be pointed to by as many suffix links as the number of characters in the alphabet. This problem is illustrated in *Table 2.1*, which shows that the link vector is very sparse.

Both McCreight's and Weiner's algorithms require the end of the string to be present before processing can start. It is not required by Ukkonen's algorithm because it builds the tree from left to right. That is why it is often referred to as an on-line algorithm, which means that it can start processing the string as soon as the first character arrives at the processing site. Of course, in this case we cannot use *n-1* as end pointers on leaves as discussed in the previous subsection. A more detailed comparison of the three algorithms was discussed by Giegerich and Kurtz [GK97].

Here we also note that suffix links are directly used in the matching statistics algorithm introduced in the next section, thus McCreight's and Ukkonen's algorithms are better choices for that problem.

2.7 Algorithms on Suffix Trees

In this section we introduce algorithms that use the suffix tree structure and help us solve the overlap problem. We have shown that a suffix tree can be built in time proportional to the number of characters in the text. In this chapter we present two algorithms that solve the overlap problem. The first one requires a generalized suffix tree for the two texts while the second one only requires one suffix tree, that is, it is more favourable.

2.7.1 The Longest Common Substring of Two Strings

This problem was defined in Section 2.5.3. To solve the problem with the aid of suffix trees we can build a generalized suffix tree for the two strings, which is built from the concatenation of the two strings. In this tree each leaf has two indices. One denotes which document it is from (1 or 2) and the other one is the usual suffix index within that file. Another option is to use absolute indexes and store the "cut-off" index that separates the two strings. Let us suppose that the two strings are *P* and *T* and their lengths are *m* and *n*, respectively.

We can build a generalized suffix tree in O(*m*+*n*) time by concatenating the two strings and using two different termination symbols. After the tree has been built, we can do a depth-first traversal and mark each node with a 1 (or 2) if there is a leaf in its subtree which comes from document 1 (or 2). This can be done in O(*n*+*m*) time. During this search we can store the depth and the address of the deepest node labelled both 1 and 2. From this node there must be a leaf, which is labelled 1, or an edge running into a node labelled 1, and also a leaf, which is labelled 2, or an edge running into a node labelled 2. Note that we cannot have an edge running into a node labelled both 1 and 2 because that node would be deeper than our node and we are dealing with the deepest node labelled both 1 and 2. If we use Ukkonen's algorithm to build the tree the edge indices will actually denote occurrences, thus an edge running out of a *d*-character-deep node with a start position of *i* denotes an occurrence starting at *i-d*. We find two edges that belong to the two different files and calculate the positions.

As an example let us suppose that the two strings to be compared are *P='abcdab'* and *T='bcdabb'*. We terminate *P* by *'#'* and *T* by *'$'*. Then we concatenate the two strings: *S='abcdab#bcdabb$'*. The generalized suffix tree for *S* is depicted in *Figure 2.10*. *P* is referenced as *document 1*, and *T* as *document 2*.

We used a general index (running from *0* to *13*) to label edges. Indices *0* through *6* are from the first document and indices *7* through *13* are from the second document. For every leaf, we give the number of the document the given leaf comes from in brackets. In this example each node is labelled 1,2 and the deepest one is *node 3* with depth 5. The leaf with *document 1* has a start index of 6, which means that it appears in *P* starting at position 1 (6-5). The leaf with *document 2* has a start index of *12*, which means that it appears in *S* starting at position 7 (12-5), which is equivalent to 0 in *T*.

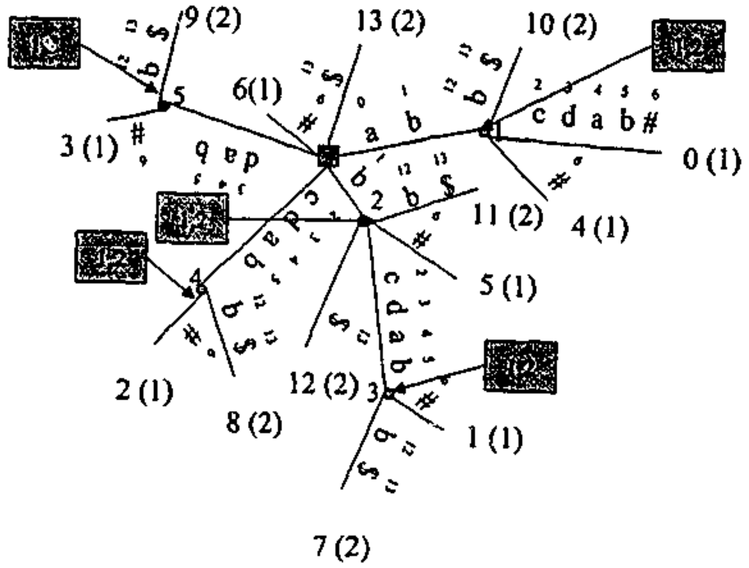


Figure 2.10. Generalized Suffix Tree

Before we show how this method can be generalized to solve our overlap problem we introduce the matching statistics values, which can easily answer our overlap problem. Given two strings *T* and *P*, the matching statistics value $ms_T(i)$ is the longest chunk starting at position *i* in *T* that appears anywhere in *P*. Formally

$ms_T(i)=v$ if $T[i..i+v-1]=P[k..k+v-1]$ for some $0\leq k\leq m-1$, and there is no $0\leq j\leq m-1$ such that $T[i..i+v]=P[j..j+v]$. Note that there may be more than one k value satisfying this condition. We may only need one of those values because we are not interested in how many times it appears in P since we are examining document T. We store one of these positions and call it the matching chunk position: $mc_T(i)=k$. Obviously finding the matching statistics value reveals all chunks of T that overlap with document P.

The longest common substring problem can be generalized to solve the matching statistics problem. In the next section we show a more elegant and more space-efficient solution, thus here only the outline of this generalization is given. We do a depth-first search on the tree and we keep a list of pointers about suffixes of document T in the subtree of the current node. If the current node is labelled 1 and 2 we find an edge belonging to *document 2* and set the matching statistics values of the suffixes in the list to the depth of the node. The matching chunk position value can also be set accordingly.

The longest common substring problem can further be generalized to more than two strings with the extension that each string must have a unique termination symbol and we are looking for nodes labelled with all possible document indices.

2.7.2 Matching Statistics Algorithm

Finding the matching statistics values, which we introduced in the previous subsection, is a general problem. Landau and Viskin [LV89] gave a solution that uses a generalized suffix tree and Galil and Giancarlo [GG88] gave an automata-based solution, which requires scanning the string 4 times.

The most elegant and space-efficient solution was given by Chang et al. [CL94]. The matching statistics algorithm builds a suffix tree for P and compares T to the tree in linear time.

To find the $ms_T(0)$ the algorithm starts matching T from position 0 to the tree beginning at the root. When there is no further match possible we record the matching statistics value in $ms_T(0)$. Let us suppose that this value is z, which means that $T[0..z-1]$ was found in the tree. If $T[0..z-1]$ is in the tree we also know that $T[1..z-1]$ is also in the tree and its position can be found by following the suffix link of the deepest node encountered on the path and then traversing down using the same skip/count trick as used in Ukkonen's construction algorithm. Once the position of $T[1..z-1]$ is located in the tree we can try to find further matches ($T[z]$). Then we can progress using the same method we used to find $ms_T(1)$. The $mc_T(i)$ values can be retrieved from the tree using the same idea as in the longest common substring problem. Let us denote the position of the last character that matched in the tree by j, then the $mc_T(i)$ value can be set to $j-z+1$.

Note that each character of T is examined only once, which means that this part of the algorithm runs in linear time. The skip/count steps are also proportional to the number of characters in T, which can be proven using the same reasoning as in Ukkonen's construction algorithm. Overall O(m) time is required to build the tree, and then O(n) time to find the matching statistics values. Thus the running time is O($m+n$).

The high-level pseudo code of the matching statistics algorithm is depicted in *Figure 2.11*. Suppose that the length of T is n and the length of P is m. Once we have the matching statistics values for T we can easily extract chunks that overlap. We only have to consider that an $ms_T(i)$ value of z represents the same chunk as $ms_T(i+1)=z-1$. Only the left-most of these positions need be recorded. The algorithm in *Figure 2.12* calculates the number of characters that overlap but it can easily be modified to extract the actual chunks. The array *ms* contains the matching statistics values.

```
Build suffix tree for P
Traverse down the suffix tree starting matching from T[0]
Let z denote the number of characters matching
Record $ms_T(0)$ and $mc_T(0)$
For i=1 To n-1
    Follow the suffix link of the deepest node on the
    path to the current position
    Apply the skip/count steps to find the position
    in the tree that represents the overlap starting
    at T[i] and of length z-1
    Match further characters if possible
    Record $ms_T(i)$ and $mc_T(i)$
    z=$ms_T(i)$
End For
```

Figure 2.11. The Pseudo Code of the Matching Statistics Algorithm

In Chapter 4 we will deal with the issue of the minimum length of a chunk to be considered in our overlap detection system because this algorithm identifies every single overlapping chunk no matter how long it is.

```
last_position=-1, last_value=0
for i=0 to n-1 do
  if ms[i]==0
    continue
  end if
  current_value= ms[i], current_position=i
  if (current_position-last_position)>last_value
    overlap=overlap+last_value
    last_value=current_value
    last_position=current_position
  else if current_value>last_value-(current_position-
                 last_position)
    overlap=overlap+(current_position-last_position)
    last_value=current_value
    last_position=current_position
  end if
end for
overlap = overlap + last_value
```

Figure 2.12. Calculation of the Overlap Value from Matching Statistics Values

2.8 Alternative Approaches

The systems and algorithms discussed in Section 2.4 are very similar to each other in that they all split up the text into chunks, select a certain percentage or a number of chunks as a fingerprint of a document, build an index on those chunks, and compare documents based on those indexes. In this section we introduce two alternative copy-detection approaches, which are fundamentally different from the systems discussed previously.

The CHECK system [SLL97] is based on the observation that comparing two documents on two different subjects are unnecessary. In the pre-processing phase the structure of the document is discovered and a document tree is constructed that represents the different structural units of the document, such as sections, subsections, subsubsections, and paragraphs. At each level of the tree, keywords are stored for the given unit. Keywords are selected from open-class words [SLL97]. Firstly, keywords at the document-level are compared and if the documents are considered to be similar the next level of the trees are compared. In the end of the process related paragraphs are identified. This method has the advantage that it is not strictly exact-matching based comparison. However, a keyword extraction method is not reliable and overlapping chunks smaller than a paragraph are difficult to detect. Keyword comparison at the top-level may be used as a way of identifying a set of candidate documents but after that a more comprehensive search must be completed to filter out unrelated documents.

Glatt [Gla99] is a commercial software product and it is not computationally intensive at all. It is based on the assumption that users more easily remember their own sentences than plagiarised sentences. It takes the text as an input and generates an output text with every fifth word substituted with a blank. These blanks must be filled out by the user, and if at least 70% of the words are correct the document is considered to be genuine. According to [Gla99] no students have been falsely accused yet. The problem with this approach is that users must be involved and cooperative, which is impossible, for example, in the case of submitted conference papers. The protection of the system is also low, since reading a plagiarised assignment multiple times reduces the chances of the system to catch the user. This system may reveal cheaters but without actually presenting the originals it is hard to prove plagiarism.

2.9 Summary

In this chapter we have presented different schemes for safeguarding intellectual property. Firstly we introduced copy-prevention mechanisms including using standalone CD-ROMs, special hardware for authorisation, and active documents. We believe that these approaches are too cumbersome for genuine users and they marginalise the real advantages of electronic documents.

The first copy-detection scheme we discussed was digital watermarking. Digital watermarking schemes make changes to the layout of the text, which cannot be recognised by humans. Line-shift coding, word-shift coding, and feature coding techniques were introduced. In case of infringement, special codewords hidden in the layout of the text reveal the person or company who purchased the original of that document.

In Section 2.4 we compared existing systems based on different criteria, such as chunking strategies, chunk selection schemes, decision functions, and text conversion methods. The performance, protection, and accuracy of different methods were compared.

We pointed out that these systems may produce false positives and the exact chunks are not always retrievable. Exact string-matching algorithms and suffix trees were analysed in Sections 2.5, 2.6, and 2.7. We concluded this chapter by identifying suffix trees as the most suitable data structure, thus we propose to use a two-stage approach where the second stage compares documents based on exact string matching. Suffix trees are the focus of this thesis. Thus physical storage issues and a new space-efficient representation are discussed in later chapters.

CHAPTER THREE

Suffix Tree Representations

3

3.1 Introduction

This chapter analyses suffix tree representations that have been proposed to date. Different representations have been developed with different applications in mind. For each representation we analyse its applicability for different string matching problems with special focus on the matching statistics algorithm (see Chapter 2). The key to the matching statistics algorithm is whether suffix links are stored in the representation. Some representations (e.g. Kurtz's [Kur99]) naturally include the suffix link information while others could or could not be supplemented with suffix link information.

For each representation we analyse their space requirement, which is always O(cn) (n is the length of the string) but the constant c is different for each representation. Since the matching statistics algorithm requires suffix links, which are not present in all implementations, we analyse the time complexity of the exact-matching problem. This problem is the most natural and at the same time the most basic application of suffix trees. Most suffix tree representations have their associated construction algorithms, which are not discussed here as they are beyond the scope of this thesis. Basic suffix tree construction algorithms have been discussed in Chapter 2.

To the best of our knowledge Kurtz [Kur99] has developed and demonstrated the most space-efficient practical suffix tree representation that stores suffix link information. In this chapter we extensively analyse that representation because in later chapters we compare our proposed suffix vector representation to Kurtz's suffix tree representation.

This chapter is organized as follows. Section 3.2 is a general discussion of suffix tree representations in their original form considering how edges and nodes could be represented. Section 3.3 introduces McCreight's [McC76] representation, which is one of the early attempts to store the tree efficiently. Section 3.4 discusses Kurtz's [Kur99] representation, which is based on McCreight's early representation but eliminates redundancies and accommodates a bit-level utilisation of storage space. This section also discusses another alternative representation developed by Giegerich et al. [GKS99] that does not store suffix link information (lazy suffix tree). Section 3.5 discusses a fundamentally different representation called suffix array [MM93], which does not have the versatility of a suffix tree but uses much less space. Section 3.6 discusses the suffix cactus representation [Kär95] while Section 3.7 analyses LC-tries [AN95]. Section 3.8 discusses Suffix Binary Search Trees [IL00]. Two other alternative representations are discussed in Section 3.9. Section 3.10 presents a comparative analysis of the representations discussed earlier in the chapter.

Throughout the chapter we use an example string (the same as used in previous chapters) to demonstrate the physical layout of different representations. Also, whenever it is applicable, we give the representation in table format.

3.2 Practical Implementation Issues

Linear storage requirements and linear construction time are only true in the case of a finite alphabet, which is a fair assumption in case of DNA sequences and English text. This also means that these representations and their corresponding construction algorithms always include a |Σ| factor where Σ denotes the alphabet and consequently |Σ| denotes the size of the alphabet.

The alphabet-size factor is represented in the number of edges running out of a node. If we have a finite-size alphabet the upper bound of the storage requirement of a node is constant. This brings up the most important issue in suffix tree representations, that is, how to store edges running out of each node. The following four subsections analyse the four straightforward choices: fixed size array, linked list, balanced tree, and hash-table [Gus97].

3.2.1 Fixed Size Array

If we have a finite alphabet we can store the edges running out of a node in an array of size |Σ|. Each position in the array represents a character in the alphabet. If there is an edge starting with the given character running out of that node, then the edge is stored in that position. If there is no edge running out from the node that starts with the given character, then that position of the array is empty.

A fixed size array implementation allows very fast search but wastes too much space if the number of edges running out of a node is much less than |Σ|. Search is fast because it only takes one lookup operation to find out whether there is an edge running out of the node starting with a given character.

3.2.2 Linked List

In this implementation, the edges running out of the node are implemented as a linked list. The lists may be in random order depending on the actual construction algorithm used. Traversal from the node can be performed by sequentially searching the list. This representation is more space-efficient than the fixed size array when the number of edges running out of a node is less than the size of the alphabet.

An alternative to this representation is an ordered linked list. If we keep the list in sorted order we can terminate the search as soon as we find a character in the list that is lexicographically higher than the character we are searching for. Note that the list is still a linked list. Thus a binary search cannot be performed in this structure.

3.2.3 Balanced Tree

This representation is basically a compromise between the fixed-size array and linked list representations. In a balanced tree [AHU74] additions and searches take O(log$k$) time where k is the number of children of a given node. Because of the additional programming and space overhead, this approach is only efficient for large k values.

3.2.4 Hash Table

This representation relies on some kind of a hash table that associates edges with nodes. The total number of edges is bounded by $2n$ in a tree. Thus we can define an absolute hash table, which returns the edge once a node and a character are given.

More formally, the hash table implements a function *f* from the set of ordered pairs of the form (node, character) to the set of nodes, with the property *f(v,x)=w* if there is an edge starting with character *x* between *node v* and *node w* and *f(v,x)=0* if there is no such edge. Using perfect hashing schemes [FKS84] the linear time bound can be preserved.

3.2.5 Using a Mixture of the Above Techniques

Different edge-representations can be used at different nodes of the tree. The nodes close to the root tend to have more edges than other nodes. For example, the root node has one edge for each different character appearing in the string. Based on this observation we may decide to use fixed size arrays or balanced trees at nodes close to the root while a linked list is usually a better candidate at lower level nodes where we do not expect many outgoing edges.

Also some levels may be compressed if they contain all possible followings. For example, there are 20^5 possible amino acid substrings of length 5 and each of them appears in some protein sequence [Gus97]. If we want to create a suffix tree on all proteins then we can compress the top 5 levels of the tree and index the edges with the first character. This idea is further studied in [AN95], which we discuss in Section 3.7.

3.3 McCreight's Suffix Tree Representation

McCreight was one of the pioneers in the area of suffix tree algorithms and representations. Not only did he present a construction algorithm in his early paper [McC76] but he also discussed some implementation issues. He suggested two alternative representations: one based on linked lists and the other based on hash tables.

In McCreight's representation each node is represented by the edge running into that node. The root node is the only node in the tree that has no incoming edges, so a special edge is created for that node. Each edge contains the start and end indices of the substring it represents, a pointer to the next node in the list of edges running out of the origin node of the edge, a pointer to the first edge of its destination node and a suffix link. The explanation of the representation is shown in *Figure 3.1*.

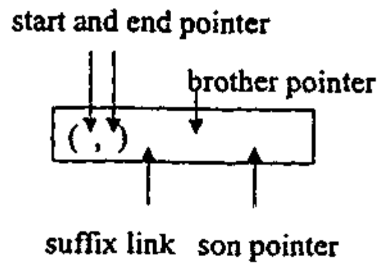


Figure 3.1. McCreight's Edge Representation

We use an example suffix tree throughout this chapter to demonstrate different suffix tree representations. This example suffix tree is built on the string *S='abcdabdbcdabb$'* that was depicted in *Figure 2.8*. McCreight's representation of this suffix tree is depicted in *Figure 3.2*.

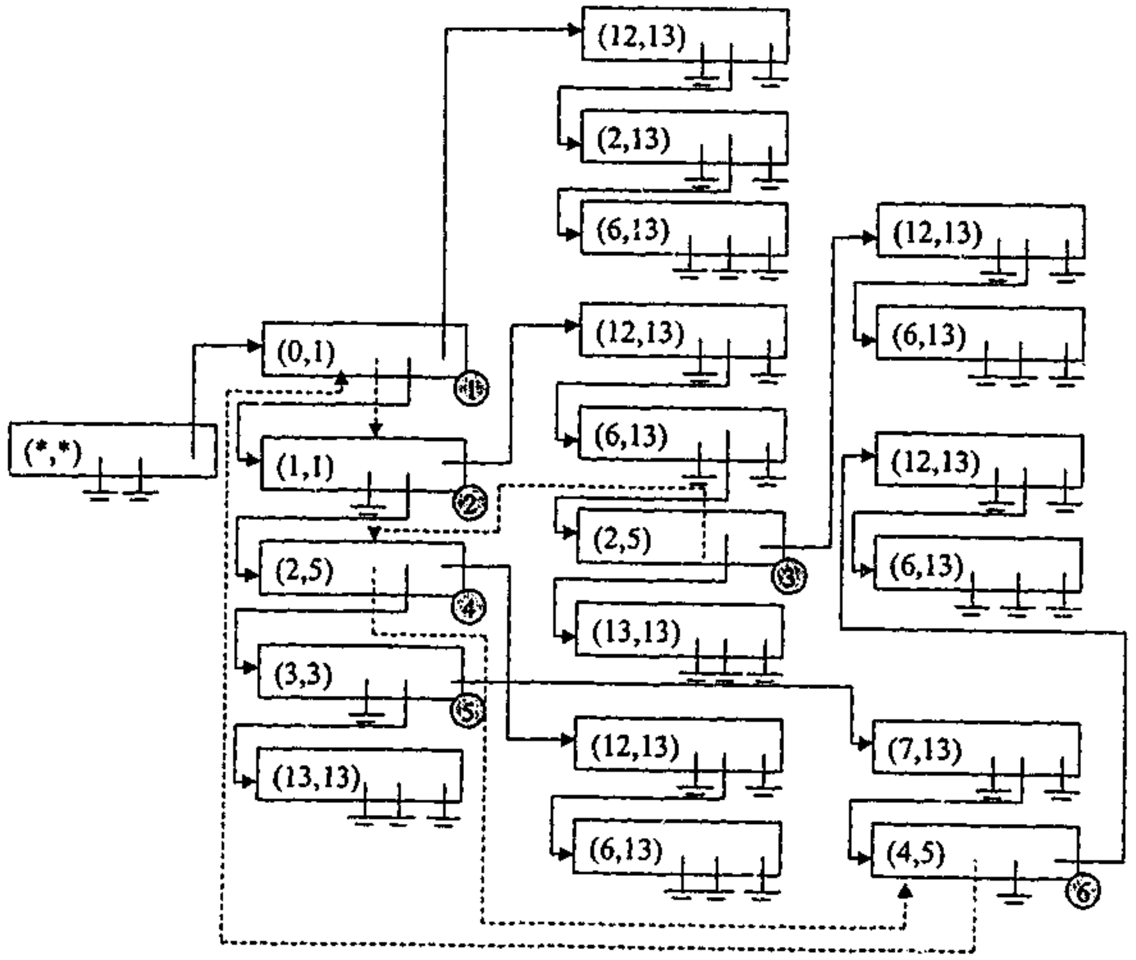


Figure 3.2. McCreight's Suffix Tree Representation

In *Figure 3.2* suffix links are represented by dashed arrows while other links (brother, son) are represented by solid arrows. As we have already mentioned the root is a special node, which is represented by an edge with no start and end pointers (*,*). Leaves are nodes with no son pointer. Nodes are represented by the edges running into them. For example node 1 in *Figure 2.8* corresponds to edge (0,1) in *Figure 3.2* because the incoming edge is (0,1). Its brother pointer points to the next edge in the list of edges running out of the root (1,1). Its son pointer points to the first edge running out of it (12,13). Other nodes are also depicted in the figure.

All throughout this chapter we assume a 32-bit memory space, which means that 32-bit pointers are used for both physical memory location pointers and indices of start and end positions of edges. We refer to this unit as one computer word (W – 4 bytes).

Now let us analyse the space-requirement of McCreight's representation. Each edge requires one start and one end position, one son pointer, one brother pointer and one suffix link, that is, 5 computer words, or 20 bytes per edge. The number of edges in the tree is the sum of the number of nodes plus the number of leaves because each edge runs into a single node or it is a leaf. The root node is an additional "edge" to represent but if we include the root node in the number of nodes count, the total space requirement of McCreight's representation is: $5*(q+n)*W$ where q is the number of nodes and n is the number of characters in the string. Of course, we have to store the string itself along with the tree. However, this is true for all representations. Thus we do not consider that space requirement in our comparison.

As a space-efficient alternative to McCreight's representation, leaves can be represented by only two pointers: the start index and the brother index. Storage of the end index is not necessary because the last character of a leaf is the last character of the string. Using this technique the space requirement can be reduced to $(2n+5q)*W$. Theoretically the upper bound for the number of nodes is n (e.g. an n-character long string of *'a'*-s). Thus the worst-case space requirement is $7*n*W$. In practice, depending on the type of text we index, the number of nodes is considerably less than n. The theoretical average for random text is $q=0.62n$ [Kur99].

McCreight's suffix tree representation can solve the exact matching problem in O(m) time if the pattern we are searching for is m characters long. In our comparison of suffix tree representations when we refer to the exact matching problem we only consider finding one occurrence and not all occurrences. In order to find all occurrences of P the total time needed is O(m+k) where k denotes the number of occurrences. Traversal of the tree can work in the way we described in Chapter 2 and once we find an occurrence we have to analyse the subtree of the position where we finished matching P. The leaves in the tree will give us the actual occurrences and if there are k occurrences the subtree can be traversed in O(k) time.

3.4 Kurtz's Suffix Tree Representation

In his paper [Kur99] Kurtz proposes four different implementations: simple linked list implementation (SLLI), simple hash table implementation (SHTI), improved linked list implementation (ILLI), and improved hash table implementation (IHTI). Both linked list implementations are superior to their hash table counterparts regarding space, thus we only discuss those implementations.

In Kurtz's SLLI each node requires 5 pointers:

1. *firstchild* points to the first child of the node (either a leaf or another node, it corresponds to the son pointer of McCreight's representation)
2. *branchbrother* refers to the next child in the list (corresponds to the brother pointer of McCreight's representation)
3. *depth* is the number of characters on the path from the root to the given node
4. *headposition* is the second left-most occurrence of the string represented by the node (the depth-headposition pair is a substitute for the start and end pointers)
5. *suffix link* is the suffix link of the given node (corresponds to the suffix link in McCreight's representation)

Leaves can be stored in an array of length n. Each leaf is stored at a position that corresponds to the suffix it represents. In other words leaf i is stored in the ith position in the array. There is one value stored with each leaf: the *branchbrother* value, which has the same meaning as the branchbrother value of a node. The start and end pointer values can be calculated from the depth and headposition values in the following way. Let us suppose that we have an edge pointing from node v to node w. The start pointer can be calculated as *w.headposition-v.depth* and the length of the edge is the difference in node depths (*w.depth-v.depth*), which determines the end position. In case the edge is a leaf edge the same formula applies to the start position by substituting *w.headposition* with j (if it is leaf j) and the end position is n-1 for all leaves. The choice of using headposition and depth instead of the more natural choice of start and end pointers will become clearer when we analyse the ILLI representation. By using this representation n computer words can be saved over McCreight's representation because only one computer word is required for each leaf. *Table 3.1* shows the leaf (T_{leaf}) and node (T_{branch}) tables of the SLLI for our example string.

T_{leaf}														
leaf #	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$T_{leaf}[j]$	10	nil	8	nil	0	11	nil	1	Nil	3	nil	nil	3	5

T_{branch}							
node #	Root	1	2	3	4	5	6
firstchild	13	4	12	7	2	**6**	9
branchbrother	Nil	nil	1	5	**2**	**4**	6
Depth	0	2	1	5	4	1	3
headposition	0	4	5	7	8	6	9
suffixlink	Nil	**2**	nil	**4**	**6**	nil	**1**

Table 3.1. The SLLI Implementation of the Example String

In *Table 3.1* nodes are referenced by the same numbers as used in *Figure 2.8.* References (firstchild, branchbrother, suffix link) in bold refer to other nodes, while other references refer to leaves. Thus they index the leaf table. In a physical implementation this distinction can be achieved by using one bit in the pointer to indicate whether the given pointer refers to a leaf or a node. It is true that we lose one bit and now we can only address 2^{31} locations but suffix tree implementations, developed so far, use at least 6 bytes per input symbol. Thus the maximum length of a string to be stored as a suffix tree in a 2^{32}-byte address space is less than 2^{30}, so 30 bits will suffice our needs.

The ILLI implementation exploits the redundancies of the above representation. Since no two headpositions of two different nodes are the same, an absolute ordering of nodes is possible based on headposition values. In that ordered sequence there are nodes whose headposition is one less than the node after, and there are nodes whose headposition differ by more than one from the following node. The latter ones are called large nodes and the rest of the nodes are small nodes. Kurtz shows that small nodes share some characteristics with the left-most large node following in the list. We can partition the sequence of nodes into chains of zero or more consecutive small nodes followed by a single large node. Let us denote one of these chains by $b_l,\ldots,b_r$ where b_r is the large node. The following holds for any small node b_i in the chain:

- $b_i.depth = b_r.depth+(r-i)$
- $b_i.headposition = b_r.headposition-(r-i)$
- $b_i.suffixlink = b_{i+1}$

$(r-i)$ is the distance between the position of the small node and the position of the large node. Based on these observations a small node can be stored in 2 computer words by storing the distance, firstchild, and branchbrother values while a large node can be stored in four computer words by storing firstchild, branchbrother, depth, headposition, and suffix link.

In Kurtz's representation every bit is efficiently used. In *Figure 3.3* we only give an overview of the bit layout of nodes. More details of the representation can be found in Kurtz's paper [Kur99]. Also, more in-depth analysis of Kurtz's representation is presented in Chapters 5 and 6.

Integers (1) and (2) are used for both large and small nodes, while (3) and (4) are only used for large nodes. By default 5 bits are used to store the distance, which only allows storing distances up to 32. If a longer sequence occurs a small node is artificially converted to a large node, thus splitting the long sequence. 10 bits are used to store the depth whenever possible. Nodes with depth values greater than 1023 are stored in a different way. In this case the suffix link is stored with the last edge running out of the node.

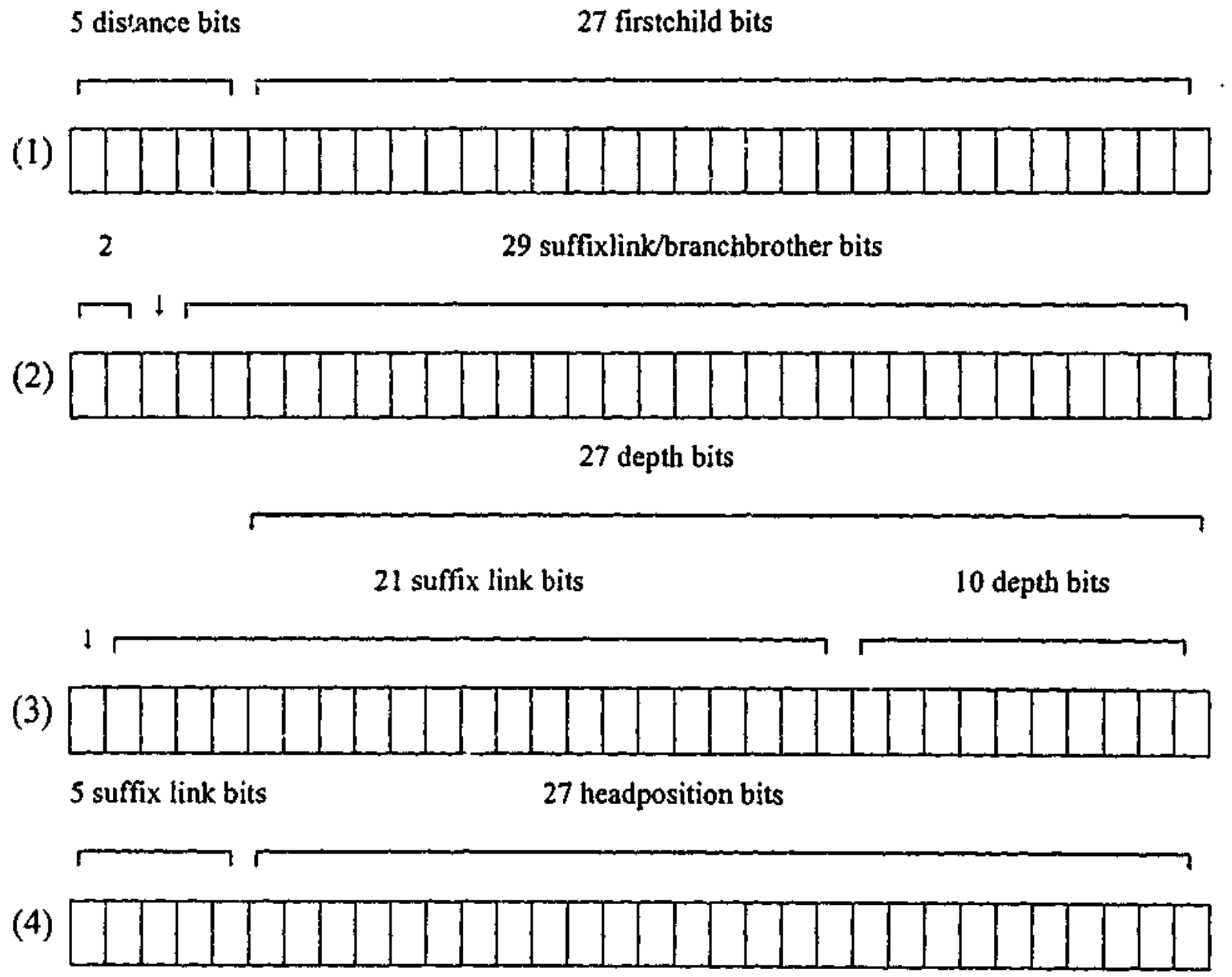


Figure 3.3 Bit-Level Node-Representation in ILLI

The exact matching algorithm runs in the same way on Kurtz's representation as in the general representation. The only difference is that edge indices are retrieved differently. We compare our representation to Kurtz's [Kur99] in Chapter 5 where the actual space requirements are also analysed.

Kurtz and others have developed another alternative representation [GKS99], which is not as versatile as the SLLI representation, because it does not store suffix link information. They call the structure a *lazy suffix tree* and it is based on a total order of the children of a branching node. This ordering scheme uses the index of the leaf with the smallest index value in the subtree of the node. Let us denote this value for node v by min*l(v)*. This representation only stores the left index for an edge. Let us suppose that we have an edge between node u and v then the left pointer of the node can be calculated as min$l(v)+|u|$. The right pointer is one character to the left of the left pointer of the smallest child running out of node v.

For each node we store the left pointer and a pointer to its first child. Children of the same node are stored at consecutive positions thus no link pointers are required. For leaves we only store the left pointer. We need two special bits: one to indicate the end of a child list and another one to indicate whether the given entry is a node or a leaf. It means that we need 2 computer words for a node and one computer word for a leaf. Giegerich et al. [GKS99] also give a construction algorithm for a lazy suffix tree, whose expected running time is $O(n\log_k n)$. *Figure 3.4* shows the lazy suffix tree representation of our example string.

0	9	1	12	2	16	3	18	13	2	6	12	1	12
1		2		4		5		13	0	4	10	3	
0	1	2	3	4	5	6	7	8	9	10	11	12	13

6	12	13	6	12	6	24	7	6	12	6	12
5	11	12	2	8	6		6	1	7	3	9
14	15	16	17	18	19	20	21	22	23	24	25

Figure 3.4. Lazy Suffix Tree Representation

In *Figure 3.4*, boxes with solid frames represent nodes, and shaded boxes represent the last child in the list. The lower row only indicates the mapping of nodes and leaves to the general representation (node numbers and leaf numbers). This row is not actually stored.

3.5 Suffix Arrays

Suffix arrays are good substitutes for suffix trees in some applications but they use a fundamentally different approach to represent the suffixes of a string. Suffix arrays were first introduced by Manber and Myers [MM93]. A suffix array is a one-dimensional array of size n (the length of the string), where the pointers to suffixes are stored in lexicographical order. The most naive exact matching algorithm would do a binary search in the array to find a given pattern. Given a pattern of length m the time complexity of this algorithm is $O(m\log n)$. In *Figure 3.5* we show the suffix array of our example string.

pos	0	1	2	3	4	5	6	7	8	9	10	11	12	13
SUFFIX	13	10	0	4	12	11	7	1	5	8	2	9	3	6

Figure 3.5. Suffix Array

Suffix arrays may be augmented with longest common prefix (lcp) values, which speed up the search. *Lcp(i,j)* by definition is the longest common prefix of the suffix starting at position i and the suffix starting at position j. If during a binary search we know the lcp values of the boundaries of the current interval, the running time of finding a pattern can be reduced to $O(m+\log n)$. We only need the lcp values for certain pairs in the array. Thus these extra values require an extra $2n$ computer words. The total space requirement of a suffix array with lcp values is $3*n*W$. *Figure 3.6* shows the pairs that the lcp values are required for, as well as the actual lcp values for our example string.

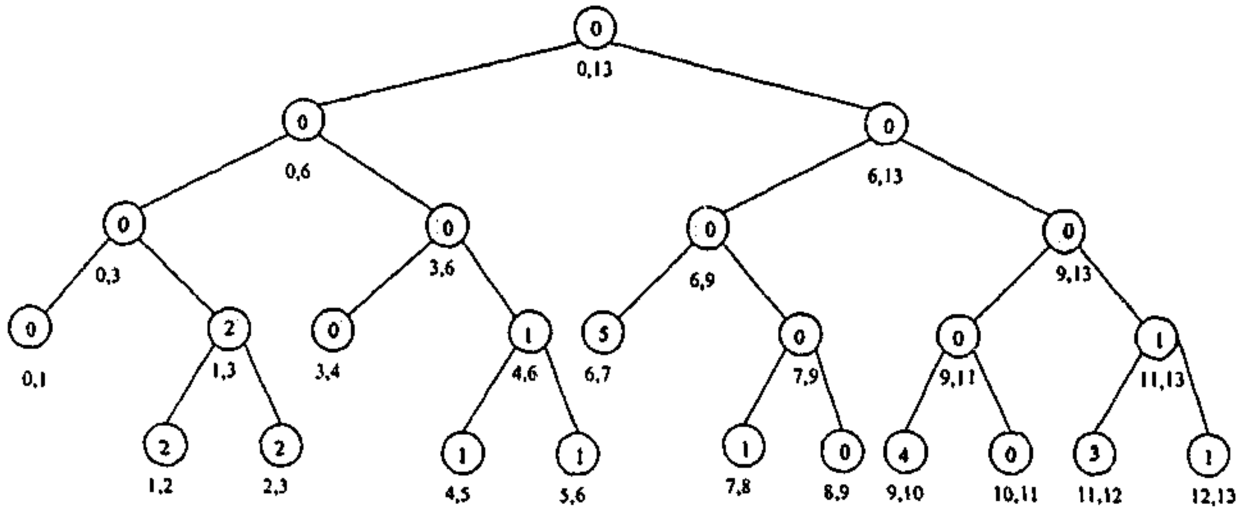


Figure 3.6. Lcp Values

Since we are doing a binary search on the array, these values are best represented in a binary tree. The pairs are depicted under the nodes (these values need not be stored because they can be derived from their positions in the tree) and the actual lcp values are depicted in the node. Suffix arrays can either be derived from a suffix tree representation or they can be built directly.

Suffix arrays have the advantage of being alphabet-size independent but the searching is slower and suffix links are not present in this representation. That means that our matching statistics problem cannot be solved in linear time using this representation. We also have to note that instead of storing lcp values in one computer word, we can store it in one byte provided that they do not exceed 255, which is the case in many texts. If we want to store lcp values in one byte we have to prepare for the rare exception when more than one byte is needed. By storing lcp values in one byte the space requirement of a suffix array is *6*n* bytes.

3.6 Suffix Cactus Representation

This section describes Kärkkäinnen's suffix cactus representation [Kär95], which can be viewed as a "cross between a suffix tree and a suffix array" [Kär95]. The representation includes the suffix array discussed in the Section 3.5, as well as two other arrays of the same length with some extra information.

At the high level, a suffix cactus can be viewed as a suffix tree where each node is catenated with one of its children. The simplest option is to catenate each node with its first child, but it supposes that there is an ordering among the children. A natural ordering is the alphabetical ordering of children. *Figure 3.7* shows the suffix cactus representation of our example string.

In a suffix cactus a branch is the result of the catenation of one or more edges. In *Figure 3.7*, branches are numbered 0-13 and they are shown in lexicographical order. In order to represent a suffix cactus we have to define a parent relation between branches. We define the root of a branch as the starting node of the branch. Now we say that the parent branch of a node is the branch containing the preceding sibling of its root. In the figure above, numbers 13, 0, 4, 10, etc. under the branches (represented by numbers in normal font) are leaf indices while numbers in bold depict the parent branch of a given branch. Now we can represent a suffix cactus with three arrays:

1. SUFFIX is the suffix array of *S*, that is the lexicographically ordered list of suffixes
2. DEPTH(*i*) is the depth of the root node of the given branch
3. SIBLING(*i*) is FIRSTCHILD(*i-1*) if *i-1* has children and NEXTSIBLING(*i*) if *i* has a next sibling.

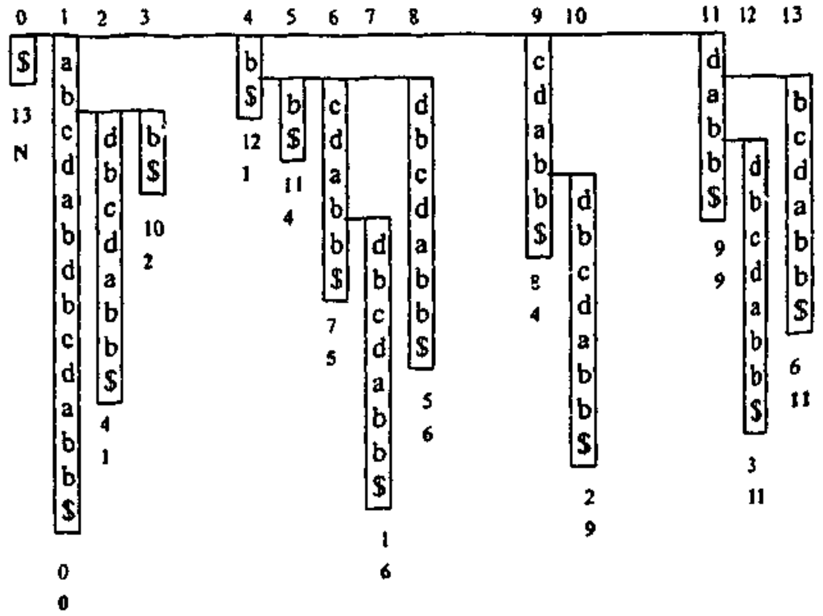


Figure 3.7. Suffix Cactus

These three arrays are shown in *Table 3.2* for our example string.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
SUFFIX(*i*)	13	0	4	10	12	11	7	1	5	8	2	9	3	6
DEPTH(*i*)	0	0	2	2	0	1	1	5	1	0	4	0	3	1
SIBLING(*i*)	0	1	4	3	2	9	6	8	7	5	11	10	13	12

Table 3.2. Array Representation of a Suffix Cactus

Kärkkäinnen [Kär95] suggests to store suffix and sibling values in one computer word each (two computer words in total) and one byte for depth values. Here the same comments apply to deep nodes as the comments about lcp values in the previous section. Depending on the number of bytes used for the depth values a suffix cactus implementation requires *9*n* or *12*n* bytes.

The exact matching algorithm can run in the same manner as it runs in case of a general suffix tree representation. However this representation also lacks suffix links, which means that the matching statistics algorithm cannot run on this structure in linear time.

3.7 Level-Compressed Tries

Andersson et al. [AN95] suggest representing the suffix tree as a trie. Every suffix tree can be represented as a binary tree if we encode the characters in a binary format. The number of bits used to encode a character depends on the size of the alphabet. *Figure 3.8* shows the top levels of the binary trie of our example string. For the sake of the LC-trie (Level-Compressed trie) example we left out the termination symbol, thus we have a four-letter alphabet, which can be encoded in two bits. We only show the top part of the trie because of space limitations and because the concepts are clearly exhibited in these top levels. We have encoded *'a'* as *'00'*, *'b'* as *'01'*, *'c'* as *'10'*, and *'d'* as *'11'*.

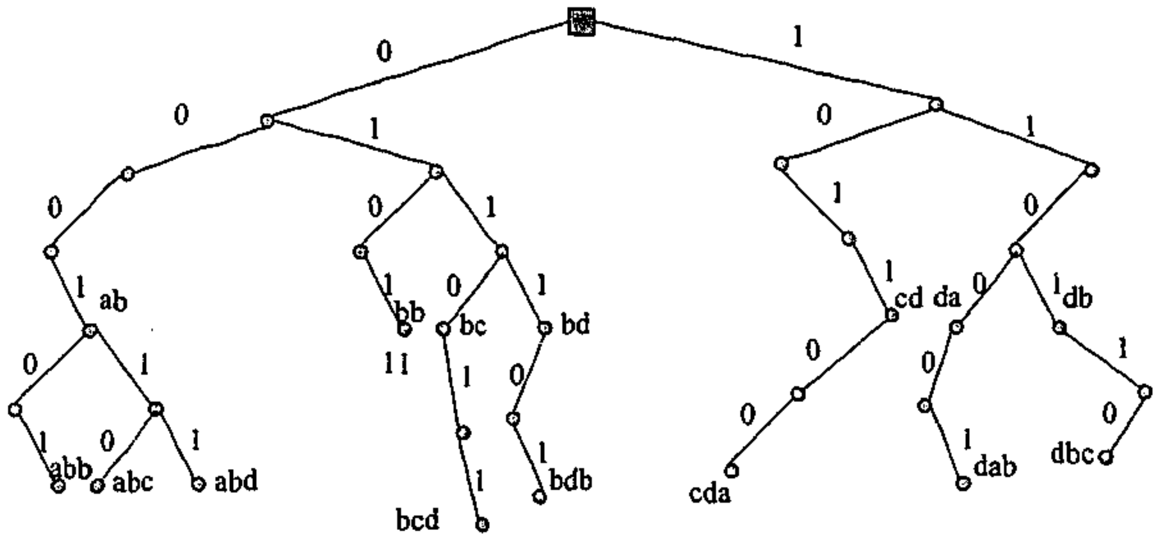


Figure 3.8. Binary Trie

There are two operations that may condense the trie:

- *path compression* – each node that has only one child can be merged with its child and a skip value is applied on the path. A path-compressed binary trie is a Patricia tree [Mor86]. As an example, the last three edges of the path running to node 'bdb' can be compressed into one edge with a skip value of 3.
- *level compression* – if the *i* highest levels of the trie are complete but level *i+1* is not, we replace the *i* highest levels by a single node of degree 2^i. This replacement is repeated top-down. In our example the top two levels are complete; thus they can be replaced by a single level of degree 4.

The top part of the LC-trie of our example string is shown in *Figure 3.9.* Skip values are shown in bold and we numbered each node for referencing.

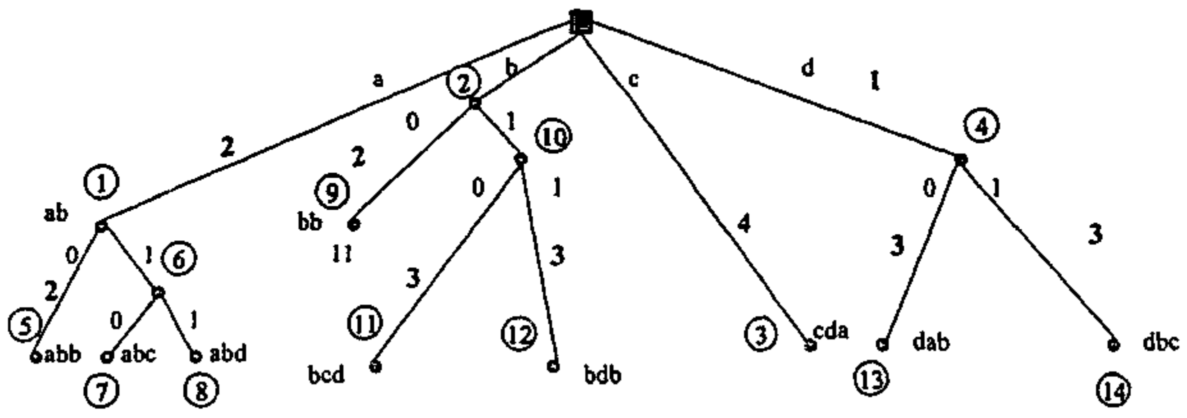


Figure 3.9. LC-trie

A node of an LC-trie can be represented by three numbers (all requiring a computer word each):

1. the number of positions to skip,
2. the position of the leftmost child,
3. and the branching factor (for level compressed nodes).

The first and third values are mutually exclusive for each node, thus they can be stored in one computer word, a bit of which flags whether it is a branch or a skip value. The LC-trie representation of a suffix tree requires *10*n* bytes on average if a computer word is used for the leftmost child value and a short integer (2 bytes) is used for the field shared between branch and skip values.

Exact matching in the LC-trie works in the same fashion as in other suffix tree representations but the matching statistics algorithm cannot run in linear time on this structure because it does not implement suffix links.

3.8 Suffix Binary Search Trees

Suffix binary search trees (SBST) and suffix AVL trees were introduced by Irving et al. [IL00]. Instead of storing the suffixes in an array as is done in a suffix array, all suffixes are stored in a binary tree with some extra information to help search:

- m_i is 0 if i is the root otherwise it is $\max_j lcp(S_i,S_j)$ for all ancestors j of node i (S_i and S_j are the suffixes starting at position i and j, respectively)
- d_i (1-bit value) is *left* or *right*, depending on whether node i is in the left or right subtree of node j found above

Figure 3.10 shows the SBST without the additional *m* and *d* values for our example string. As one can see the tree can be highly unbalanced, which affects searching time.

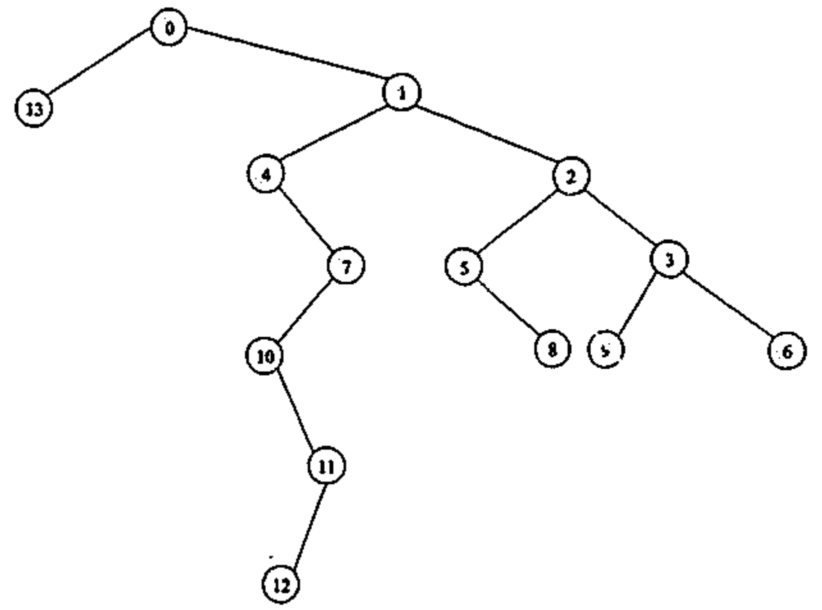


Figure 3.10. Suffix Binary Search Tree

In order to balance the tree, the suffix binary search tree can be converted into a suffix AVL tree. An AVL tree is a balanced binary tree where the height of the two subtrees (children) of a node differs by at most one. The tree can be maintained as an AVL tree during the construction. The suffix AVL tree of our example string with the auxiliary information (m_i and d_i)is shown in *Figure 3.11.*

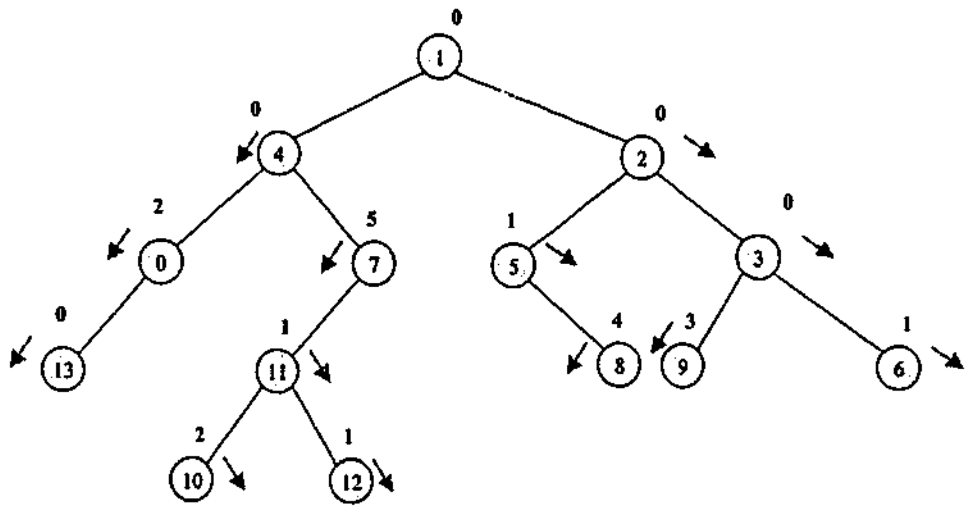


Figure 3.11. Suffix AVL Tree

The exact matching algorithm on the SBST runs in O(m+l) time where l is the length of the search path in the tree. In the worst case l is O(n) but the expected length is O(log$n$). In case of the suffix AVL tree the worst case bound for l is O(log$n$). The complexity of the construction algorithm for SBST is O(nh) where h is the height of the tree. Similarly to the search algorithm the worst case bound for h is O(n) but for random strings the expected height is O(log$n$), which is the worst case bound for AVL trees.

3.9 Alternative Representations

In this section we analyse two alternative representations, which have a slightly different perspective from the representations discussed above. In Section 3.9.1 we introduce a representation that deals with the issues of how to efficiently represent a suffix tree on disk, and Section 3.9.2 introduces a representation based on some encoding techniques.

3.9.1 Suffix Tree on Disk

Hunt et al. [HAI01] propose a suffix tree representation that can be stored efficiently on disk. The suffix tree representations discussed in Sections 3.3-3.8 all work well when the tree can fit into main memory. However as soon as we get to a stage where paging comes into effect the performance those representations degrade because of poor locality of suffix trees. Poor locality mainly stems from suffix links, which can cause a jump from one part of the tree to a totally different part.

Hunt et al. [HAI01] propose not to use suffix links during the construction and they split the suffix tree into buckets based on the first x characters of the suffix. Those buckets are built independently. The worst case running time of the algorithm is O(n^2) but because of the pseudo-random nature of DNA sequences the average behaviour in their tests showed O($n$log$n$) running time. The exact matching algorithm runs in O(m) time or if all k occurrences need to be found it runs in O(m+k) time.

3.9.2 Compressed Suffix Arrays

The compressed suffix array data structure was proposed by Grossi et al. [GV00]. It is based on the observation that strings of length n over a binary alphabet may produce at most 2^{n-1} different suffix arrays. This suggests that each suffix array can uniquely be assigned a number between 1 and 2^{n-1}. The key point of this kind of representation is how to look up individual values in the array. Grossi et al. [GV00] show that the lookup operation can be done in O(log$^{\varepsilon}n$) time for any fixed constant $\varepsilon > 0$.

This representation can be as compact as O(log$n$) bits but because of the complexity of the lookup and compress operations we do not consider them as viable options for our matching statistics algorithm.

3.10 Summary

In this chapter we have given an overview of existing suffix tree representations. We used an example string to demonstrate different ideas to store suffix trees more efficiently. In *Table 3.3* below we summarize the features of the representations discussed above, except for the representations analysed in Section 3.9. We analyse different representations based on the following criteria:

- average space requirement – n is the length of the string, and since each representation requires the actual string to be present it is not considered here. In McCreight's representation q is the number of nodes. The figures in brackets show space requirements of those representations which have limited string lengths because of smaller si.:e indices. Data is given in bytes;
- suffix link – whether or not suffix links are represented;
- construction via the suffix tree – whether it is suggested by the authors to construct the structure via the suffix tree;
- direct construction time – the time complexity of the direct construction algorithm;
- running time of the exact matching algorithm.

	Space	**Suffix Link**	**Construction via Tree**	**Construction Time**	**Exact Matching**
McCreight	8n+20q	Yes	NA	O(n)	O(m)
SLLI	10.1n	Yes	Yes	O(n)	O(m)
Lazy Suffix Tree	8.5n	No	No	O($n$log$_k n$)	O(m)
Suffix Array	12n (6n)	No	Yes	O($n$log$n$)	O(m+log$n$)
Suffix Cactus	12n (9n)	No	Yes	O($n$log$n$)	O(m)
LC-trie	10n	No	No	O(n)	O(m)
Suffix AVL Tree	10n	No	No	O($n$log$n$)	O(m+log$n$)

Table 3.3. Comparison of Suffix Tree Representations

As one can see from the above table, Kurtz's SLLI implementation is the most suitable for our matching statistics algorithm because of its space requirement and versatility. In Chapter 6 we propose a new structure called suffix vector, which performs even better in terms of space requirement and has the same versatility.

CHAPTER FOUR

Modified Suffix Tree

4.

4.1 Introduction

In the previous two chapters we have shown how suffix trees can be used to solve the document overlap problem. We also have analysed different suffix tree representations and shown that the space requirements of suffix trees often limit their applicability because they do not represent suffix links. In this thesis we study more space-efficient suffix tree implementations and propose a more space-efficient method to store suffix trees. Our application area is document overlap detection, which means that the applicability of data structures are studied from this perspective, though it is true that the data structures presented in this thesis can also be used in other applications.

Our quest for more space-efficient representations is set along the following dimensions. A number of approaches try to make use of the inherent structure of natural language texts and only store suffixes that start at the beginning of a word [MZS99]. Some other implementations are more general and can be used in any application of suffix trees [MZS01, MZV01].

The suffix vector, which is proposed in this thesis, is a general representation, which can be applied to any text, i.e. it does not make use of words [MZV01, MZS02]. We prove that the suffix vector representation is the most space-efficient representation known to us. It is discussed in detail in Chapters 5 and 6.

In this chapter we analyse other alternatives, which make use of word boundaries. We also analyse a structure called Directed Acyclic Graph (DAG) that eliminates some redundant information of suffix trees [MZS01]. In the following section we first show how a suffix tree can be used in a novel fashion to solve the matching statistics problem. This new method uses only one suffix tree built on the suspicious document and candidate documents can be compared to that tree. The original algorithm in a one-to-many comparison would require building a suffix tree for each candidate document.

In Section 4.3 we study how the suffix tree structure can be used by only inserting the suffixes that start at the beginning of words. This representation obviously saves space but we have to analyse how the structure can directly be constructed as well as how algorithms, especially the matching statistics algorithm, can run on this structure [MZS99].

In Section 4.4 we introduce Directed Acyclic Word Graphs (DAWG) and their compressed form of Compressed DAWG (CDAWG). Another alternative that is referred to as a Directed Acyclic Graph is also introduced. This data structure eliminates redundancy in the suffix tree. Not only does this save space but it also eliminates some redundant comparisons in the matching statistics algorithm [MZS01].

Performance analysis of the above-mentioned data structures is presented in respective sections. Our results are summarized in Section 4.5 and possible future extensions are also discussed.

4.2 The Matching Statistics Algorithm by Using Only One Suffix Tree

In Chapter 2 we have shown how the matching statistics algorithm can answer the document overlap problem. The algorithm presented there works well in a one-to-one comparison scenario. However, when we want to compare one file to many files, a suffix tree needs to be built for each document we want to compare to that one document. From here on we will refer to the one document to be compared to many documents as the *suspicious document*[2] while the other documents will be called the *candidate documents*. This naming scheme comes from the plagiarism detection application, though the algorithm presented here can also be applied in other applications.

[2] Throughout this thesis we use the words 'document', 'string', and 'text' interchangeably.

The following subsection introduces the problem and explains why we need to find an algorithm that works on one suffix tree while Subsection 4.2.2 presents our algorithm.

4.2.1 Matching Statistics Algorithm with Many Suffix Trees

As discussed in Chapter 2, the document overlap problem can be solved by building a suffix tree for each candidate document and running the matching statistics algorithm on those trees with the suspicious document. The result of this will be a matching statistics array for the suspicious document, which can answer the overlap problem by using the algorithm described in Section 2.7.2.

If we reverse the role of the suspicious document and the candidate document we can run the same algorithm but the result will not identify all overlaps of the suspicious document with regards to the candidate documents. The reason is that when identifying a matching statistics value for the candidate document (plain text) only one occurrence in the suspicious document (suffix tree) can be identified. This is the case unless we are willing to sacrifice the linear running time of the algorithm. Once an overlap is found we can traverse the subtree of the node our path is running into. This would give all the occurrences in the suspicious document. For each of these positions we could set the matching statistics values accordingly.

Here we note that in our document comparison system we can set a threshold for the minimum length of overlapping chunks to be recorded. This threshold is set to 60 characters for plagiarism detection but other thresholds can be defined for different applications. It means that most matches that should be recorded will finish on a leaf or a node with only few leaves in its subtree. It is true because deep nodes mean long overlaps within the suspicious document itself. Thus, we expect the overhead expenses of this approach to be marginal. That does not significantly degrade the running time of the algorithm in practice. Theoretically, this approach is not linear in time.

In order to illustrate the problem we give an example of a one-to-many comparison. Let us suppose that our suspicious document is the string that we have used to demonstrate suffix tree representations (*S='abcdabdbcdabb$'*). The two candidate documents to be compared to the suspicious one are C_1=*'cbcdad$'* and C_2=*'bbdb$'*. The suffix tree of S is depicted in *Figure 2.8*. In Step 1 (finding $ms_S(1)$ starting from letter *b*) in the comparison of C_1 to the suffix tree of *S*, we finish matching on the edge between nodes *2* and *3*. Here, if we are willing to sacrifice the linearity of the algorithm we can traverse the subtree of node *3* and find both occurrences (*7* and *1*). If we want to remain within the linear time bounds we can record the occurrence with the smallest index. It can directly be read from the edge labels: we have matched 4 characters and we finished matching at position *4*, so the starting position is 1. If we use this latter method, the resulting matching statistics and matching chunk position arrays at the end of the algorithm are shown in *Table 4.1*.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
S	a	b	c	d	a	b	d	b	c	d	A	b	b	$
$ms_S(i)$		4	3	2	1	3	2					2	2	1
$mc_S(i)$		1	2	3	4	1	2					0	3	6
C		C_1	C_1	C_1	C_1	C_2	C_2					C_2	C_2	C_1

Table 4.1. Matching Statistics Values by Using One Suffix Tree

As one can see from the above table, not all positions are filled out in the table. Had we run the matching statistics algorithm in its original form (comparing S to the suffix trees of C_1 and C_2, respectively) all positions would have been filled out properly. *Table 4.2* shows the matching statistics values for this latter case.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
S	a	b	c	d	a	b	d	b	c	d	A	b	b	$
$ms_S(i)$	1	4	3	2	1	3	2	4	3	2	1	2	2	1
$mc_S(i)$	4	1	2	3	4	1	2	1	2	3	4	0	3	6
C	C_1	C_1	C_1	C_1	C_1	C_2	C_2	C_1	C_1	C_1	C_1	C_2	C_2	C_1

Table 4.2. Matching Statistics Values by Using Many Suffix Trees

The problem illustrated above raises an interesting question if the algorithm is used for plagiarism detection. If a chunk, that is taken from some other document, appears many times in the suspicious document, does that mean one or many counts of plagiarism? One could say it is one count of plagiarism because once it was copied that "copy" was used. Others could argue that the chunk was plagiarised many times. This is not a technical issue, thus we do not analyse this question further. In the following subsection we show how the first matching statistics table could be converted into the second one in linear time by using only the suffix tree of S.

4.2.2 Converting the Matching Statistics Table

From *Tables 4.1* and *4.2* one can see that the difference can be resolved by analysing the suffix tree structure of S. The *'bcda'* overlap is present in two positions in *S*, which is represented by node *3* in the suffix tree. Node 3 is in the continuation of the path labelled by *'bcda'*. Generally we can say that given a matching statistics value of *m* at position *i*, the matching statistics value at position *j* must be set to

$$\max[ms_S(j), \min(ms_S(i),d)] \qquad (4.1)$$

where *d* is the depth of the longest common ancestor of leaves *i* and *j*. If we had to calculate (4.1) for each pair of *i* and *j* we would need $O(n^2)$ time. The length of *S* is *n*. Below we present an algorithm that correctly sets the matching statistics values in $O(n)$ time.

In the first step of the algorithm we create a tree with exactly the same structure of nodes and edges as the original suffix tree. Each leaf stores the following information:

- *leaf number*: the start position of the suffix represented by this leaf
- *matching statistics value*: the matching statistics value defined by the algorithm in the previous subsection (note that this may or may not be the proper matching statistics value)
- *matching chunk position*: the matching chunk position belonging to the above value
- *document identifier*: the identifier of the document the above two values belong to (e.g. C_1)

The last 3 values from this list are stored in nodes. These values are taken from the leaf in the given node's subtree with the highest matching statistics value. If there are more than one of those values, any of them can be selected. We also need to retain node depth information from the original suffix tree structure. The values stored in leaves can be directly derived from the initial matching statistics values while the values in nodes can be obtained by a depth-first traversal of the tree. The values for our example are shown in *Figure 4.1*.

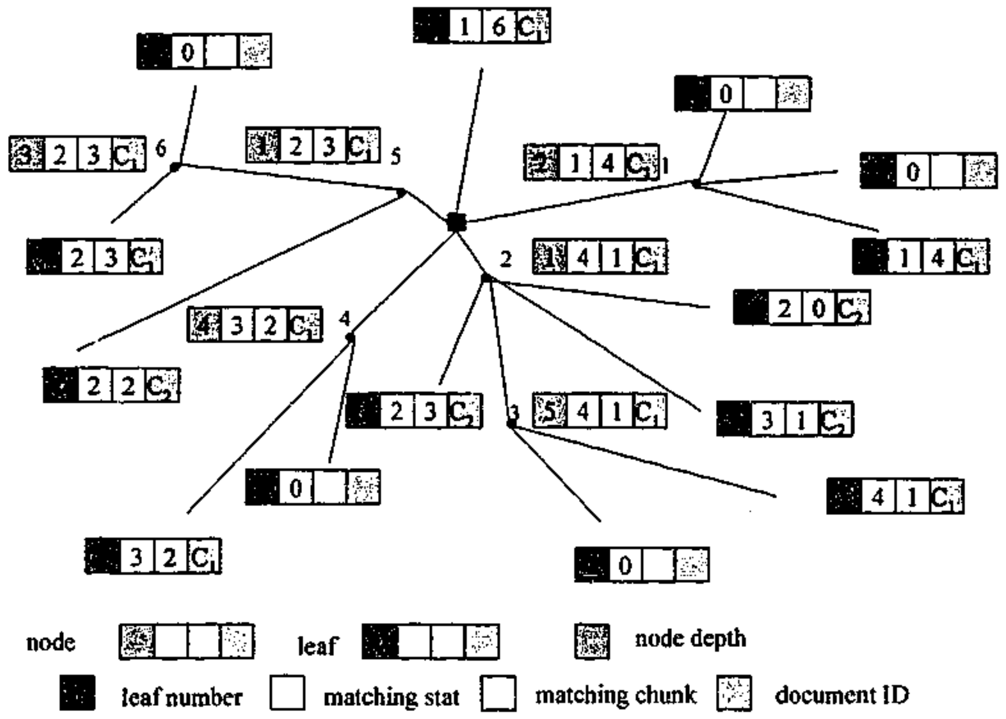


Figure 4.1. Auxiliary Tree

In the second step we do a breadth-first traversal of the tree. Let node w be a child node of node v. During the traversal the matching statistics value of node w is set to

$$\max[\min(ms_v,d_v),ms_w] \qquad (4.2)$$

where ms_v denotes the matching statistics value of node v, d_v denotes the depth of node v, while ms_w denotes the matching statistics value of node w. The corresponding matching chunk and document identifier values are set accordingly. When we get to a leaf we can apply the same formula by substituting ms_w with ms_l where ms_l is the matching statistics value of the given leaf:

$$\max[\min(ms_v,d_v),ms_l] \qquad (4.3)$$

We call this algorithm the Matching Statistics Value Assigning Algorithm (MSVAA). Its pseudo code is depicted in *Figure 4.2.*

```
1. Build a tree with the same structure as the suffix
   tree.
2. Assign leaf numbers, matching statistics values,
   matching chunk position values, and document
   identifiers to leaves.
3. With a depth-first traversal, set the same values in
   each node in a way that it stores the maximum value
   of its children.
4. With a breadth-first traversal, set the values in
   the child nodes and leaves of the current node by
   applying formula (4.3).
```

Figure 4.2. The Pseudo Code of the MSVAA Algorithm

Theorem 4.1. The MSVAA algorithm correctly sets the matching statistics values for each position in S.

Formally, we can say that if there is a matching statistics value of ms(j) set at position j which is greater than the one at position i (ms(i)), and S itself has an overlap of length d (S[$j..j+d-1$] = S[$i..i+d-1$]) which appears at both j and i, then the matching statistics value at i must be set to max[min(d,ms(j)),ms(i)]. To prove this theorem we assume that either ms(i)<ms(j) or ms(j)<ms(i) but we only prove the former case. The latter case can be handled similarly. We will assume that the values are not set correctly after we have run the algorithm and we will show that it is not possible. There are 3 cases.

Case 1. ms(i)<d<ms(j). Leaves i and j have the longest common ancestor of depth d. In the first step of the algorithm the matching statistics value of that node is set to ms(j) or if there is a leaf with a greater matching statistics value it is set to that value. Let us denote this value by k. In the second step of the algorithm the nodes on the path from the longest common ancestor to leaf i will be set to d because d is less than ms(j) or if there is a node with a greater value on the path from the longest common ancestor to leaf i it will be set to that value. Also the matching statistics value of leaf i will be set to that value, which is at least d, which contradicts with our original assumption for this case.

Case 2. ms(i)<ms(j)<d. Leaves i and j have the longest common ancestor of depth d. In the first step of the algorithm the matching statistics value of that node is set to ms(j) or if there is a leaf with a greater matching statistics value it is set to that value. Let us denote this value by k. In the second step of the algorithm the nodes on the path from the longest common ancestor to leaf i will be set to at least k, or in case $k>d$, it is set to d. Also the matching statistics value of leaf i will be set to k (or d), which contradicts with our original assumption. Similarly to Case 1 there may be a higher value on the path but it would still contradict with our original assumption for this case.

Case 3. $d<\text{ms}(i)<\text{ms}(j)$. In this case the already set value of ms(i) is greater than the number of overlapping characters between suffix i and suffix j. Thus the matching statistics value set at j has no effect on the matching statistics value set at i.□

In the case when we are only interested in overlapping chunks of length above a certain threshold, the algorithm needs to run only on nodes with depths greater than the given threshold value. *Figure 4.3* shows our auxiliary tree after the second step of the algorithm.

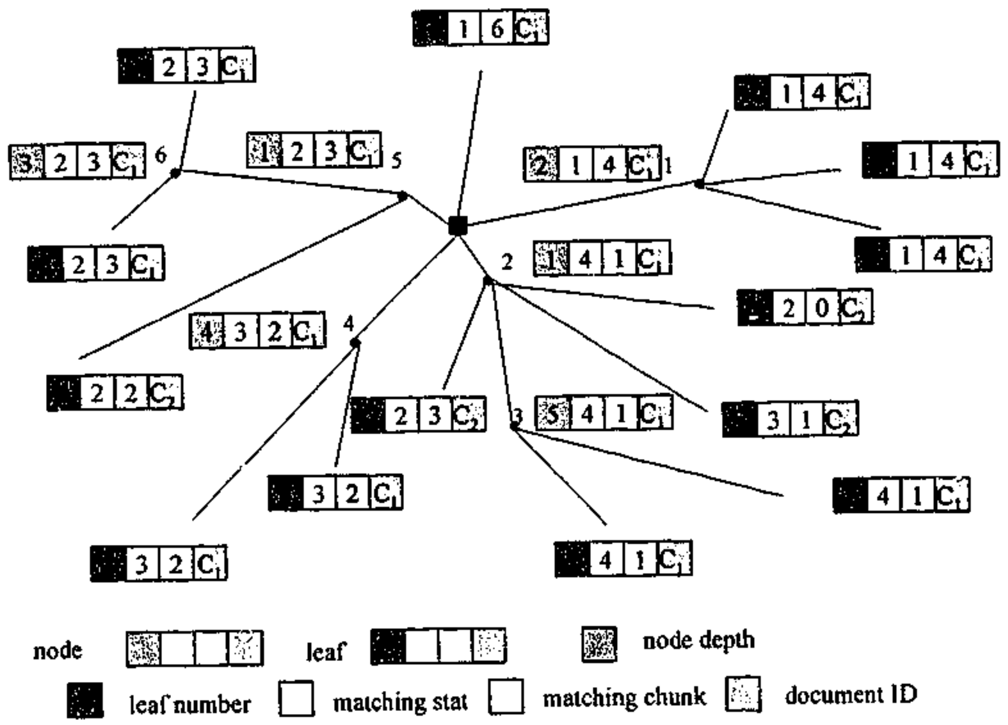


Figure 4.3. Auxiliary Tree after the Second Step of the Algorithm

In Theorem 4.1 we have proven that the values are properly set and it can be checked on our example string in *Figure 4.3*.

4.3 Sparse Suffix Trees

In this section we describe a modification of a suffix tree that only contains suffixes that start at the beginnings of words. Obviously, this data structure could save space but we have to analyse how it could be constructed and whether it could be used for the matching statistics algorithm [CL94].

These suffix trees have been studied in the literature and different names were suggested: sparse suffix trees [KU96] and word suffix trees [ALS99]. We use the term 'sparse suffix trees' in this section.

In the following subsection we formally define sparse suffix trees and analyse two different construction algorithms, while in Subsection 4.3.2 we discuss how sparse suffix trees may be used in the document overlap detection problem. Subsection 4.3.3 presents the results obtained in our test system.

4.3.1 Construction of Sparse Suffix Trees

A sparse suffix tree can be defined as a suffix tree containing only a subset of all suffixes of a string S. This is a general definition where suffix selection is arbitrary. In our application, and also in many other applications dealing with natural language texts, word boundaries are a natural choice of suffixes to be represented. We have found that the most convenient choice of suffix positions is the right-most non-alphanumerical character before the word. Thus we artificially insert a non-alphanumerical character at the beginning of the document if one is not already there. Here we also note that in our plagiarism-detection application we remove multiple whitespaces between words, which means that the term 'right-most' becomes irrelevant. A more detailed description of the text conversion performed in the pre-processing phase can be found in Chapter 7.

A straightforward construction method for a sparse suffix tree would build the suffix tree for S and then remove those suffixes that are irrelevant. Not only would this construction method use an unnecessarily large amount of space but also suffix link relationships between nodes would be violated. This latter effect would prevent us from using the structure for the matching statistics algorithm.

In [KU96] and [KD95] two algorithms have been presented that are based on Ukkonen's [Ukk95] and McCreight's [McC76] algorithms, respectively. The only difference between the original algorithms and the direct construction algorithms of the sparse suffix tree is that suffix links have different meanings. Otherwise the algorithms can progress in a similar fashion by scanning the text and only inserting suffixes that start at the beginning of words or more generally at any delimiter character.

From here on we restrict our discussion to suffix trees that contain suffixes starting at word boundaries and we refer to this structure as a sparse suffix tree. The ideas presented here for our definition of sparse suffix trees can be generalized to a more general definition of sparse suffix trees. Before we present the new definition of suffix links we formally define the term "word" in our representation. Let us denote the word delimiter by the '+' symbol. The substring $S[i..i+l]$ is a word if $S[i]$='+' and $S[i+l+1]$='+' while $S[j]\neq$'+' for any $i<j\leq i+l$.

Suffix links in a sparse suffix tree can be defined as follows. There is a suffix link pointing from node v to node z if and only if the label of node v is $\alpha\beta$ where α is a word and β is a substring of S, and the label of node z is β. In *Figure 4.4* we present a part of the suffix tree of *S='+they+were+the+last+to+arrive+but+they+were+not+late+'*.

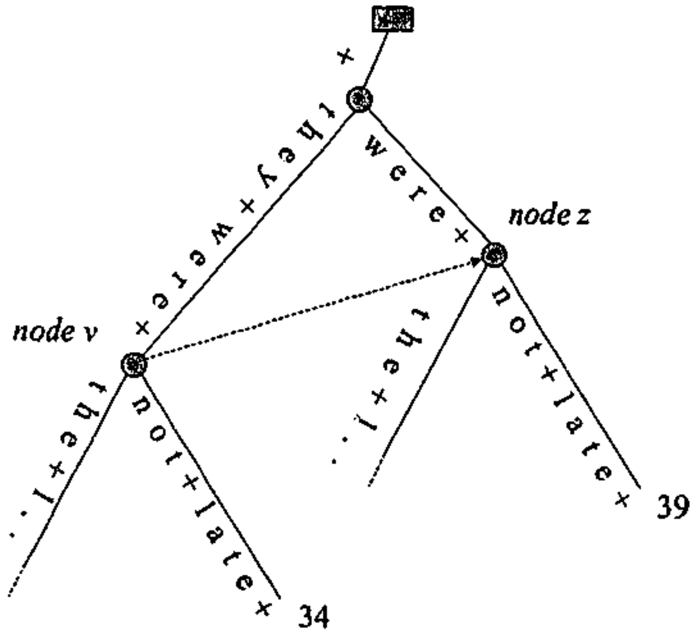


Figure 4.4. Suffix Link in a Sparse Suffix Tree

In the example the label of node v is *'+they+were+'* while the label of node z is *'+were+'*, which means that α=*'+they'* and β=*'+were+'*. We will show in the following section that this interpretation of a suffix link is very convenient for us and it can directly be utilised in the matching statistics algorithm in natural language text processing.

Andersson et al. [ALS99] describe a suffix tree construction algorithm which is explicitly different from the basic algorithms [McC76, Ukk95, Wei73]. We sketch the algorithm below and a detailed discussion of the algorithm can be found in Andersson's paper [ALS99].

In the first step of the algorithm we build a trie that contains all the different words of S. Andersson et al.'s choice is to append the delimiters at the end of each word, whereas in our approach we consider them as the first characters of the words. Either of these approaches produces words where no suffix of a word can be the prefix of another, which implies that each distinct word will correspond to a leaf in the trie.

Step 2 performs an in-order traversal of the trie, which means that children of a given node are visited in lexicographical order. Each leaf is assigned a number, which corresponds to its position during the in-order traversal.

In Step 3, a number string is generated, which lists the numbers assigned to each word in the string in the order of their appearance. This can be done by traversing the trie while scanning the string.

In Step 4 we create a suffix tree for the number string generated in the previous step. For this we can use any suffix tree construction algorithm.

Step 5 expands the number-based suffix tree into the sparse suffix tree. This is done by replacing each number with its corresponding word. In order to maintain the linear time bound of the algorithm, the word trie must be pre-processed for least common ancestors by using any suitable algorithm, e.g. the one published by Harel and Tarjan [HT84].

A time analysis of the above algorithm is also presented in Andersson's paper [ALS99]. When the positions of all delimiters are known, a lexicographic m-word suffix tree having m_l distinct words can be constructed in time

$$O\left(\frac{N}{b}+m+s_b(m_1)\right)$$

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

In our prototype system we have used Ukkonen's construction algorithm with the modifications described in [KU96]. This approach suited our needs better because it keeps the suffix link information in a way that can be utilised in the matching statistics algorithm. The latter approach by Andersson does not keep suffix link information.

Here we also note that in [KU96] the time-bound for searching the sparse suffix tree is also discussed. Of course, searching for substrings that start at suffixes represented in the tree is straightforward but a general search that finds any substring is more complicated and it is further analysed in [KU96].

Another issue about sparse suffix trees is that not all suffix tree representations support the insertion of only those suffixes that start at delimiters. As an example McCreight's [McC76] general representation can easily adopt sparse suffix trees while Kurtz's [Kur99] representation makes use of the fact that nodes are in the order of suffix links. Thus it cannot accommodate sparse suffix trees without major modifications.

4.3.2 Running the Matching Statistics Algorithm on a Sparse Suffix Tree

The matching statistics algorithm can use the suffix links of a sparse suffix tree in the following way. We start matching only from beginnings of words (recall our definition of word) and when we finish traversing a path due to a mismatch, we can follow the suffix link and continue matching from that node where we jumped to, because having found a matching chunk of $\alpha\beta$ we will follow by matching β.

As an example let us suppose that we want to compare the string *'+they+went+home+before+midnight'* to the suffix tree of *Figure 4.4*. We traverse down the tree until we find *'+they+went+home+'*, which gives a score of 16 for the matching statistic value at position 0. Then we have to find the matching statistics value at position 5, that is the beginning of the next word. We follow the suffix link, which places us on the path of *'+went+home+'*, so we can continue matching from this position, which in our case is impossible. So we set the matching statistics value to 11 (the length of *'+went+home+'*) for position 5.

As one would expect, the matching statistics algorithm using a sparse suffix tree can save all comparisons that are in connection with suffixes starting in the middle of words. Of course, this is very useful for natural language texts where such an overlap does not carry any useful information.

4.3.3 Performance Analysis of Sparse Suffix Trees

In this section we analyse the time required to build a sparse suffix tree as well as the running time of the matching statistics algorithm on the structure. We use an array-based implementation because in this experiment our priority is running time. Our suffix tree representation uses arrays to represent edges running out of nodes and also some extra information is stored on nodes to make the matching statistics algorithm run faster. This representation uses 23 bytes per input symbol on average for the sparse suffix tree, which allows us to store the sparse suffix tree of documents up to approximately 5MB on a machine with a main memory of 128MB. 5MB of pure English text translates into approximately 3000 pages and considering that any part of this text can instantly be accessed, one can understand why suffix trees are used in many areas of string matching. We have to emphasize that this is a very space-wasting implementation optimised for performance of the matching statistics algorithm. As we have discussed in Chapter 3 an efficient implementation uses around 10 bytes per input symbol for the original suffix tree.

This implementation suits the specific task we have used it for, namely one-to-many copy-detection tasks. Since we only have to build one suffix tree for the suspicious document, our only concern is that it should fit into main memory. For most of the documents we have analysed in our test this was true. Here, we also note that a more space-efficient alternative is discussed in Chapters 5 and 6.

Figure 4.5 depicts the relation between the size of the suffix tree and the time required to build the suffix tree (depicted by squares). Triangles depict the time required to calculate the matching statistics values if we have candidate documents of total size of 9.84M. Experiments were carried out on our test machine (Intel Pentium II 433MHz, 128M RAM, Windows NT Workstation).

The running time for building the suffix tree is proportional to the size of the document. Calculating matching statistics is theoretically independent from the size of the tree. In *Figure 4.5* there is a slight increase in time as the size of the tree is increasing. This contradicts with the linear time bound of the matching statistics algorithm, which states that the running time of the algorithm is independent from the size of the tree and only depends on the strings compared to the tree, which are the candidate-documents of 9.84M in our case. We can see that in the case of small trees the running time is slightly shorter. It is because in case of a small suffix tree the cache-hit ratio is higher during the matching statistics algorithm run. Thus there are more cases when the tree can be read from the cache rather than the main memory.

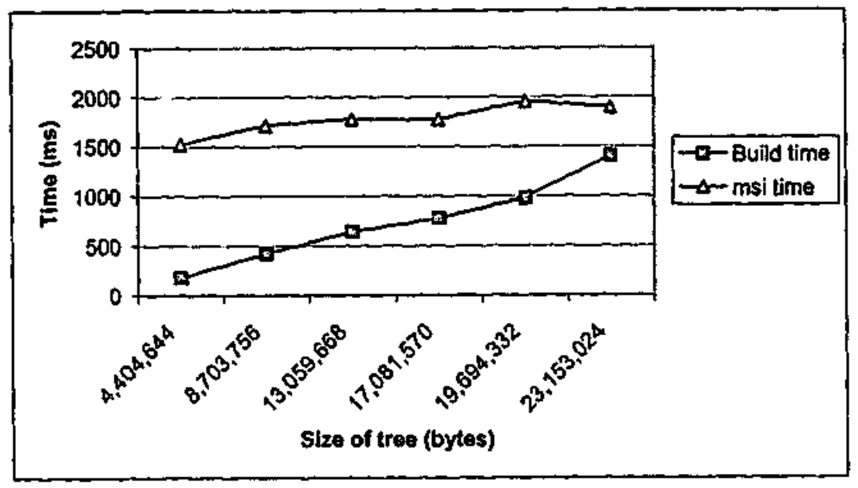


Figure 4.5. Running Time of the Construction and Matching Statistics Algorithm Usiug a Sparse Suffix Tree

The following two figures illustrate the comparison between the sparse suffix tree and the original suffix tree. *Figure 4.6* shows the running-time of the construction algorithm and the matching statistics algorithm for the same documents as used above but using the original suffix tree. *Figure 4.7* depicts the ratio of the running time of these algorithms between the original suffix tree and the sparse suffix tree (original/sparse).

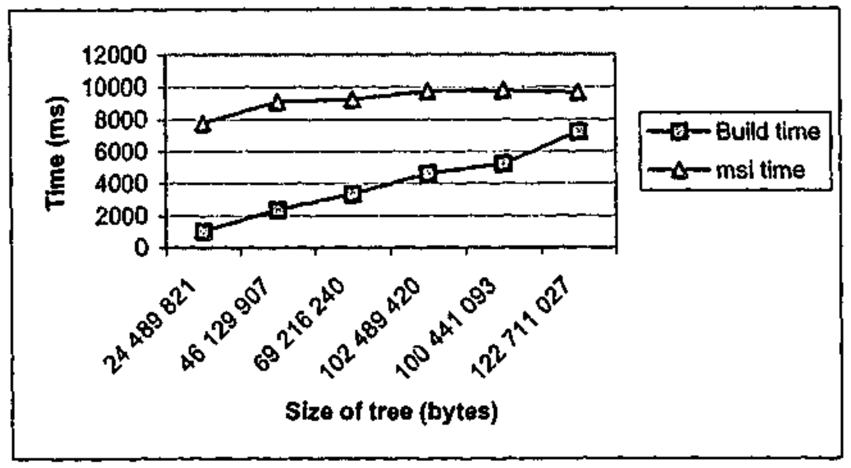


Figure 4.6. Running Time of the Construction and Matching Statistics Algorithm on the Original Tree

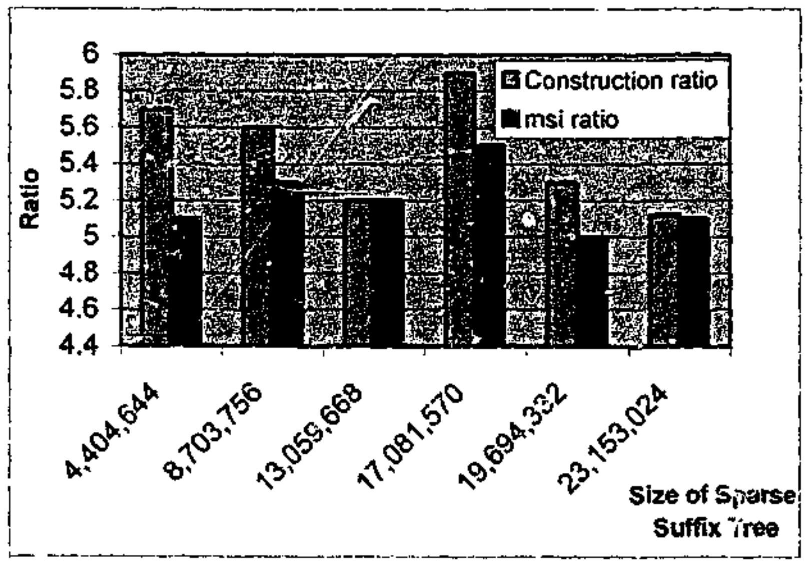


Figure 4.7. Ratio between Original and Sparse Suffix Trees

4.4 Directed Acyclic Graphs

Compact directed acyclic word graphs (CDAWG) could be viewed as an alternative structure of a suffix tree that contains all suffixes of a string. The area of CDAWGs has been studied extensively in the literature [CV97, BBE$^+$84, BBE$^+$85, Gus97]. We present the definitions of DAWGs and CDAWGs based on the definitions given in [CV97].

The *directed acyclic word graph* of a string *S*, denoted by DAWG(*S*), is the minimal deterministic automaton (not necessarily complete) that accepts all suffixes S_i $(0 \le i \le n)$ of *S*. The compact directed acyclic word graph of a string *S*, denoted by CDAWG(*S*) is the compaction of DAWG(*S*) obtained by keeping only terminal states and states that are strict classes of factors according to $\equiv_{S}$ and by labelling transitions accordingly. The definitions of classes of factors can be found in Crochemore et al.'s paper [CV97]. This definition is based on the DAWG representation. A CDAWG can be viewed as either a compact DAWG or a suffix tree where redundant nodes are merged. We follow the path of this latter derivation because it reveals the relationship between suffix trees and DAWGs more closely.

A CDAWG of a string can be derived from the suffix tree of the string by merging nodes with identical subtrees, removing the terminal symbols from leaf labels and diverting all leaves to the terminal state. If we need leaf information we have to keep the terminal symbol and we do not merge the leaves into a terminal state. This structure is no longer an automaton but it is still a directed acyclic graph, thus we will refer to this representation as a directed acyclic graph (DAG). *Figure 4.8* depicts the DAG of our example string *S='abcdabdbcdabb$'*.

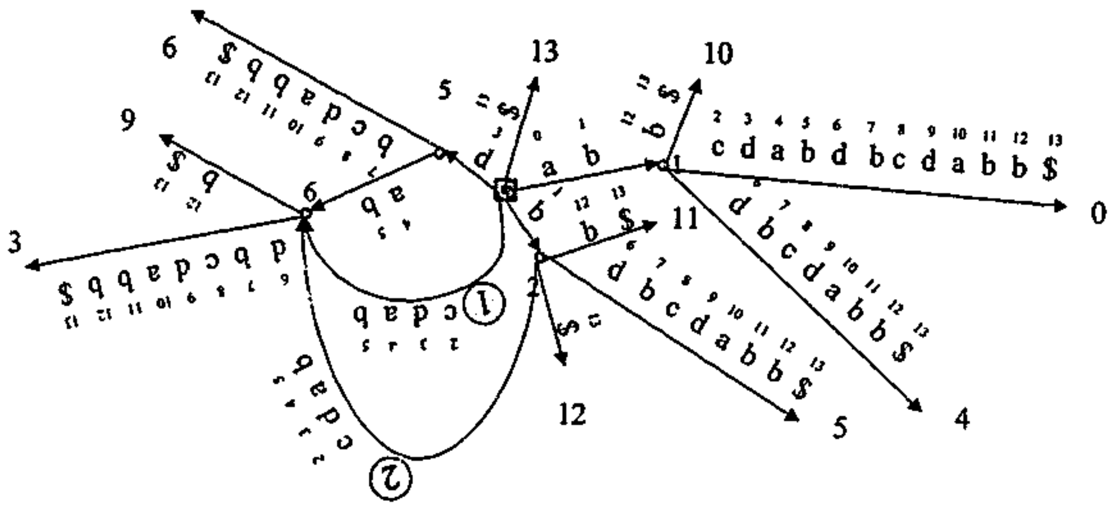


Figure 4.8. Directed Acyclic Graph Representation

In our example we could merge nodes *3* and *4* with node *6* because they had identical subtrees. Edges running into those nodes are diverted to node *6*. These edges have special values associated with them because routes containing these edges lead to leaves with incorrect leaf numbers. If we deduct the offset value accumulated on the path to the leaf we get the correct leaf number. As an example, if we follow the route of *'cdabdb'* we finish on leaf *3* but there was an offset of *1* on our route to the leaf, which means that the given substring can be found starting at position *2*.

CDAWGs and DAGs can both be constructed either via the suffix tree or via the DAWG representation. The latter approach is discussed in [CV97] while construction via the suffix tree is described in the following section. A direct construction algorithm has also been presented in [CV97].

4.4.1 Converting a Suffix Tree Into a DAG

Gusfield [Gus97] describes a linear-time algorithm to convert a suffix tree into a DAG. In *Figure 2.8* one can see that isomorphic subtrees are connected by suffix links. Gusfield [Gus97] formally proves that the subtree below *node p* and *node q* are isomorphic if and only if there is a directed path of suffix links between the two nodes and they have the same number of leaves. Since isomorphic subtrees are connected by suffix links there is an offset of indexes on leaves. The number of suffix link hops defines the value of the offset to be applied. The tree of *Figure 2.8* has been converted into a DAG depicted in *Figure 4.8*. Offset values can be applied to new links and having reached a leaf with an index on traversing down the tree we have to subtract the sum of the offset values encountered on the way from the actual index value of the leaf.

A suffix tree converted into a DAG can be used to solve some exact matching problems including deciding whether a pattern is a substring of another string, the longest common substring, exact set matching, etc. However with this representation we lose suffix links, which are heavily utilized in Chang's matching statistics algorithm [CL94]. We have to find a way to incorporate suffix links into this representation and also a conversion algorithm that converts a suffix tree into a DAG preserving suffix links. These issues are discussed in the following section. In *Figure 4.9* we present the compaction algorithm given by Gusfield [Gus97].

```
Identify the set Q of pairs (p, q) such that there is a
suffix link from p to q and the number of leaves in their
respective subtrees is equal.

While there is a pair (p, q) in Q and both p and q are in
the current DAG,
     Merge node p into q
End While
```

Figure 4.9. The Pseudo Code of the Conversion Algorithm

4.4.2 Converting a Suffix Tree into a DAG Preserving Suffix Links

There is a practical problem when applying the algorithm given in the previous section. Deciding whether *p* and *q* are in the current DAG is not straightforward. Our implementation language is C++ and nodes are represented as pointers. It is not obvious how to decide whether a given pointer points to a valid address or it points somewhere in main memory, which was previously freed. We could traverse down the tree for each *p* and *q* but it would sacrifice the linear time bound of the algorithm.

The other main problem is that the matching statistics algorithm heavily uses suffix links, which have to be retained in the new presentation. When we created our version of the conversion algorithm we had to preserve suffix links in the new representation [MZS01].

Our first observation is that many suffix links are removed in the conversion process. What is the equivalence in the DAG representation of following a suffix link in the suffix tree? If there is a suffix link in the suffix tree representation between *nodes p* and *q*, and *nodes p* and *q* are merged in the conversion process, it means that their subtrees are isomorphic. During the matching statistics algorithm we follow the suffix link to find the value of ms($i+1$) having found ms(i) but if the two subtrees are isomorphic then it is certain that no further matches will be found, so ms($i+1$) is simply ms(i)-1. We know that we are in a subtree that was previously connected by suffix links if there is an offset on the path that we have followed from the root. The sum of the offsets determines how many suffix links we could have followed without finding a different subtree, so if this value is x we already know the matching statistics ms($i+1$) through to ms($i+x$). After that we follow the algorithm as if we have already defined ms($i+x$) and we would like to define ms($i+x+1$).

It is also possible that after merging *nodes p* and *q* some suffix links under *nodes p* and *q* are lost. These suffix links are not needed at all in the DAG representation because traversing further down from *node p* will place us under *node q* after following a suffix link, and this case is equivalent to the case discussed above.

There is one more case that we have to discuss. It is possible that the destination of a suffix link is removed from the tree. There is still an equivalent node of that destination node but they are only equivalent with regards to their subtrees. If we divert the suffix link to that equivalent node, we have to analyse how it affects the matching statistics algorithm. We can place an offset on the suffix link similar to the one we place on edges. The offset value is the number of suffix link hops between the original destination node and the equivalent node. Following a diverted suffix link is equivalent to following the original suffix link in the original suffix tree, then repeating the steps as many times as the offset value. Matching statistics will always decrease by one on the way because subtrees are equivalent. If we place an offset on suffix links the algorithm is the same as the algorithm with the offset on edges.

Having discussed how suffix links are converted, we describe how our algorithm works in practice. In the first step it makes a depth-first traversal to identify an equivalent node for each node, if any. This can be done in linear time. After this, suffix links are diverted. If a suffix link points to a node which has an equivalent node, the suffix link is directed to the equivalent node with an offset of 1. It is also possible that the equivalent node has an equivalent node, too. In this case the offset is incremented by one and the suffix link is diverted to the latter node. This process can be followed until a node is found without an equivalent node. Normal edges can be diverted in the same way as suffix links. The only difference is that if an edge is diverted, the subtree of the node, that was originally pointed to by the edge, has to be removed. Instead of removing it immediately, we add this node to a list of nodes to be removed. The reason for this is that by removing a node we might break a path of equivalent nodes and at a stage we might address some invalid memory range. After all edges and suffix links are diverted we have to remove subtrees of nodes in the list. The pseudo code in *Figure 4.10* summarizes our algorithm.

```
Find number of leaves under each node
For each node
  Identify equivalent node
End For
For each node
  Divert suffix link to equivalent node
  Define offset value for suffix link
  For each edge running out of current node
    If equivalent exist
      Divert edge
      Define offset value for edge
      Add next node to remove_list
    End If
  End For
End For
For each node in the remove_list
  Remove subtree
End For
```

Figure 4.10. Pseudo Code of the Practical Conversion Algorithm

4.4.3 Performance Analysis of DAGs

As we have shown in the previous subsection, DAGs have a special feature among suffix tree representations. This feature is that some redundant information is eliminated from the tree. Not only does this save space but we can also save time when we run the matching statistics algorithm. We have built the DAG representation of those documents that we used in our analysis in Section 4.3.3. *Figure 4.11* shows the running time of the matching statistics algorithm on the DAG representation. We can see that the elimination of redundant comparisons saves about 20% time on average.

In the next two chapters we present our new suffix vector representation, which is also a data structure that eliminates redundant information. In Chapter 5 we show that more redundant information is eliminated by the suffix vector and it is also better in terms of space than the most space-efficient representations known to date.

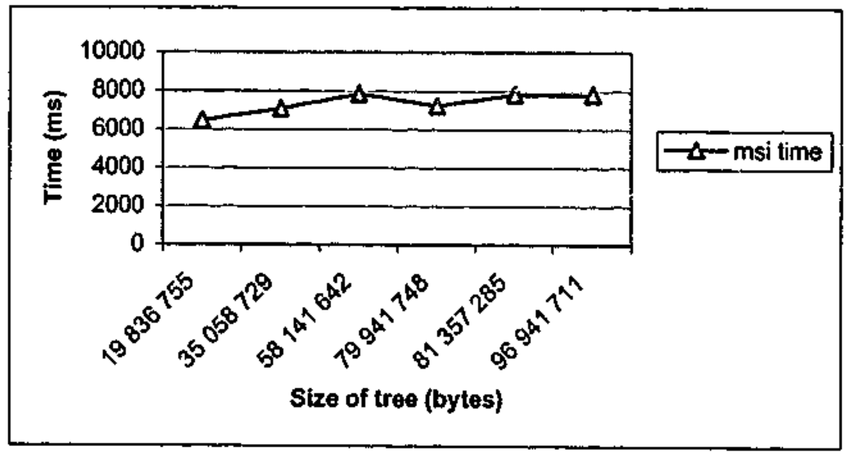


Figure 4.11. Matching Statistics Running Time on the DAG Representation

4.5 Summary

In this chapter we have shown how the matching statistics algorithm can work in a scenario when one suspicious document needs to be compared to many candidate documents. The original algorithm would require the construction of a suffix tree for each candidate document. We have shown that the matching statistics algorithm can be used in a reverse fashion, where the matching statistics algorithm is run on the candidate documents against the suffix tree, and when the comparisons have finished, the correct values can be set for the suspicious document by using an auxiliary tree.

We have also presented two alternative representations of suffix trees in this chapter. Not only do these alternatives save space but also the matching statistics algorithm can run faster on these structures. Sparse suffix trees utilize natural delimiters in English text, thus the application area of this representation is restricted to natural language texts. Directed Acyclic Graphs have a wider application area because they do not rely on word delimiters. DAGs eliminate some redundant information which is present in suffix trees.

These data structures have been widely studied in the literature but we know of no study that analysed these representations in the context of the matching statistics algorithm or more generally in the context of suffix links. Suffix links are introduced in sparse suffix trees in [KU96] and in CDAWGs in [CV97] but we also show how the matching statistics algorithm can benefit from this new interpretation of suffix links. Also, suffix links studied in the context of CDAWGs are not satisfactory for our matching statistics algorithm because they do not store the all important offset information, which tells us how many matching statistics values need not be calculated. The proposed modification of suffix links can be utilized in any other algorithm that uses suffix link information.

The Directed Acyclic Graph representation is a general representation that can be used anywhere where suffix trees can be used. In our performance analysis section we have shown that by merging nodes with isomorphic subtrees we can save approximately 20% space and the same ratio is maintained for the running time of the matching statistics algorithm. Of course, the modifications we have introduced for suffix links allows the structure to be used in any other suffix tree application but in this thesis we analyse data structures in the context of the matching statistics algorithm.

General sparse suffix trees can be built on any string with some delimiter characters introduced. However, the most appealing application is when they are used for natural language texts where word delimiters are natural boundaries. We have shown that the matching statistics algorithm can utilize this new representation efficiently.

The average word length in English texts is between 5 and 6 characters as observed in our experiments. This means that by storing only suffixes that start at the beginning of words we can save up to approximately 83% of the original representation. The same holds for the matching statistics algorithm, whose running time is reduced by the same factor because only values for positions of word beginnings are calculated.

It remains an open question how these two approaches could be combined. However this issue is outside of the scope of this thesis. In the following chapters we further analyse other possible space-efficient representations of a suffix tree. We propose the suffix vector representation [MZV01, MZS02], which is better in terms of space than the most space-efficient representations known to date. Our representation eliminates even more redundant information than the DAG representation, which allows the matching statistics algorithm to run even faster.

CHAPTER FIVE

Suffix Vector Representation

5

5.1 Introduction

In this chapter we propose an alternative data structure for suffix trees. We call this data structure a *suffix vector* because of the way it is organised in memory. We show that this data structure is more space-efficient than any other suffix tree data structure that has the same versatility. By same versatility we mean that suffix link information is also present because it is used in many algorithms including our document comparison algorithm based on the matching statistics algorithm [CL94].

In Section 5.2 we introduce the suffix vector at the high-level, which is abstracted from the actual physical representation in memory. We also show the connection between suffix trees and suffix vectors through an example string.

Section 5.3 describes some of the characteristics of the suffix vector and gives proofs when necessary. This will form the basis of different physical representations of suffix vectors introduced in later sections.

Section 5.4 introduces two alternative physical representations of a suffix vector. The *general suffix vector* is more space consuming than its alternative but it has the advantage that it can be directly built from a string in linear time. The second, compact representation is more space-efficient but cannot be directly built in linear time. It can be constructed from any suffix tree representation that stores suffix link information and also from the general suffix vector representation. Subsection 5.4.3 analyses the space requirement of the compact representation. The space requirement is compared to the most space-efficient suffix tree implementation known to date.

We use the same set of documents that Kurtz [Kur99] used for his comparison tests. The average space requirement of the compact suffix vector is better than that of Kurtz's suffix tree representation [MZV01]. We also analyse the details of the compact suffix vector representation to point out correlation between the structure of documents and the space requirement of suffix vectors.

Section 5.5 discusses one more alternative physical representation of a suffix vector that we call *functionally reduced suffix vector*. This representation is discussed separately from the other two representations because it is not as versatile as the other two. We do not store either suffix link or next node information in this representation, which limits the range of algorithms that can run on the structure. Kurtz et al. [GK99] also propose a representation that is limited in functionality but our representation uses less space than that of Kurtz's.

Section 5.6 summarises the results of this chapter and revisits the key issues presented in this chapter. The construction of the suffix vector and the running time of algorithms run on the structure are discussed in Chapter 6.

5.2 A Suffix Vector at a High-Level

Suffix vectors are proposed in this chapter as an alternative representation of suffix trees and directed acyclic graphs (DAG) derived from suffix trees. The basic idea of suffix vectors is based on the observation that we waste too much space on edge indices. Therefore we store node information aligned with the original string. Hence, edge labels can be read directly from the string. This section describes the proposed alternative structure and in later sections we analyse its space requirements.

Firstly we give an example string and the suffix tree representation of that string. Let the string be *S='abcdabdbcdabb$'*, the same string that we used in our previous examples. '*$*' is the unique termination symbol, which is necessary, otherwise the suffix tree cannot be constructed. The suffix tree for S is depicted in *Figure 5.1*, as a reference, it is the same tree as shown in *Figure 2.8*.

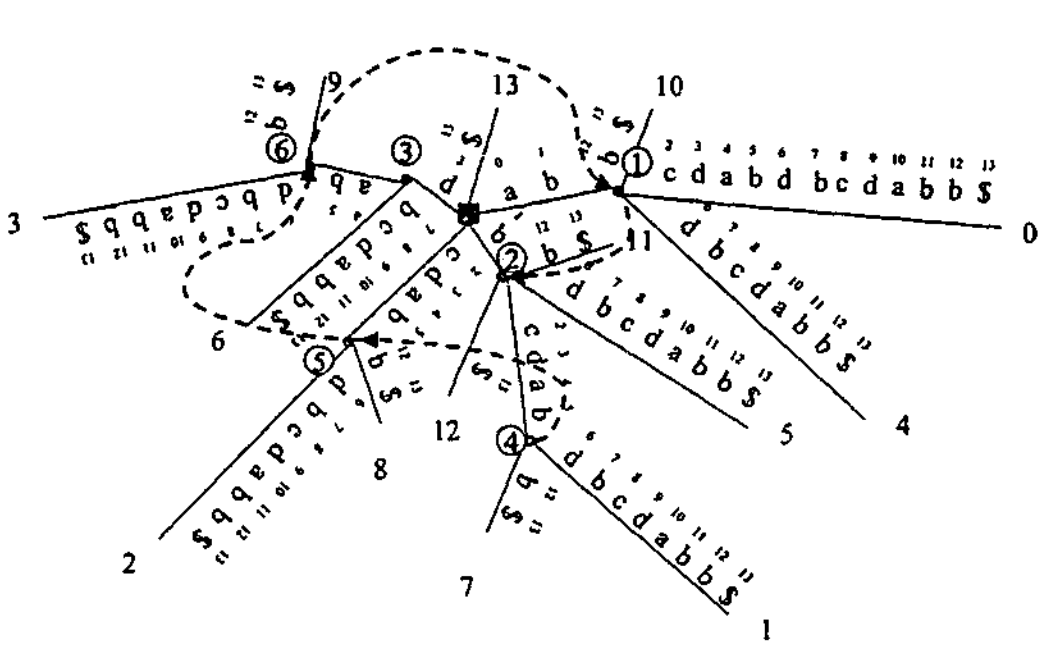


Figure 5.1. Suffix Tree of 'abcdabdbcdabb$'

Firstly we introduce a high-level suffix vector representation that is abstracted from the actual storage method. We show how the traversal of the tree works using this representation. Later we show how we can efficiently represent this structure in memory. As we have already mentioned, the basic idea of our new representation is based on storing nodes and edges aligned with the string. *Figure 5.2* shows the new representation.

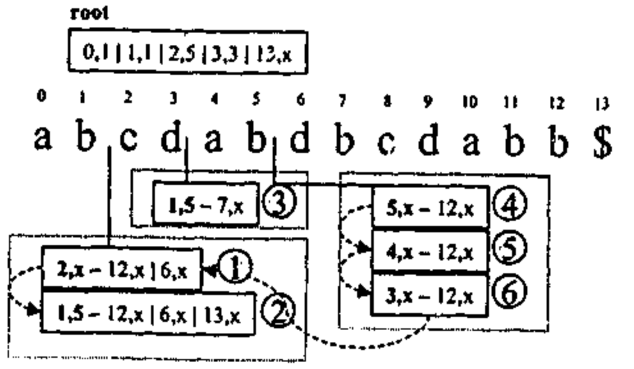


Figure 5.2. Suffix Vector of S

The root node is represented as a linked list and it shows where to start searching for a string. It has one pointer for each distinct character in the string (a,b,c,d,$). Nodes of the original tree are represented as linked lists in the vector aligned with the string. For example, *node 3* of the original tree is represented in the box at position 3. Each node has a "natural edge", that is, the continuation of the string, so the edge pointing from *node 3* to *node 6* contains characters 4 and 5 in the string. The first number in bold is the depth of the node. In the case of *node 3*, '*7,x*' means that after matching one character (position 3 *'d'*) we can either follow the string itself (this is the edge pointing from *node 3* to *node 6*), or we can jump to position 7 (this is *leaf 6*). The '*x*' means that if we jump to position 7 there are no more nodes, that is, this edge is a leaf. The second number in bold (**5**) in the pair (**1,5**) says that if we have reached this position after matching one character ('*d*') and we would follow matching '*a*' (the "natural edge"), the next node is at position 5. In the original tree the next node is *node 6*, which is depicted by the third row of the box at position 5. We need to be able to jump from one node to the next one for some algorithms. There are algorithms that do not require this information. For example, if we only need to find one occurrence of a pattern in the string we can find it without this information. In this case, the pointers of edges to the next node can be omitted. Also, next node pointers that point to the destination node of the natural edge can be omitted. However, in case we want to run the matching statistics algorithm, we need next node information because after following a suffix link we need to be able to jump from node to node.

As one can see, each node has one corresponding row in one of the boxes. *Node 1* is the first row in the box at position 1, *node 2* is the second row in the box at position 1, *node 3* (as discussed earlier) is the only row in the box at position 3, *node 4* is the first row in the box at position 5, *node 5* is the second row in the box at position 5, and node 6 is the third row in the box at position 5. Every node is stored at the smallest possible index, that is, at the first occurrence of the string running into that node.

To see how the algorithm finds a string, let us follow the matching of '*dabb*' in the string. We start from the root and find that we have to start at position 3. It is equivalent to analysing the edges running out of the root in the tree. After having matched '*d*', we try to match '*a*'. In the tree we have to check whether there is an edge starting with '*a*' running out of *node 3*, in the suffix vector we match the next character. In this case it matches '*a*'. If it had not matched we should have checked the possible followings after having matched one character. We find this information in the first row of the box at position 3. After '*a*' we have to match '*b*' on the edge in the tree and in the string in the suffix vector. They match, so we have to match the second '*b*'. They do not match. We have followed 3 characters up to now, so we have to check the possible followings from here. We can see that having matched 3 characters we could follow at position 12 – it corresponds to *leaf 9* in the tree. The '*x*' depicts that this is a leaf node, so that is the only possible match. We have matched 4 characters up to position 12, so the start position is 9.

We formally define a suffix vector below. A suffix vector V built on string S is a data structure that allows accessing each suffix S_i of the string in time proportional to the length of the suffix. Let the length of string S be n. A suffix vector is an n-element array of suffix boxes B. A suffix box may contain information on multiple nodes. A node is defined similarly to its definition in the case of a suffix tree. We have a *node* w with depth d in the box at position i if $S[i-d+1..i]=T$ is the first occurrence of T in the string and there exists a $j>i$ where $S[j-d+1..j]=T$ and $S[i+1]\neq S[j+1]$. The first occurrence of string T with length d is at position i, which means that $T= S[i-d+1..i]$ and there is no $k<i$ where $T= S[k-d+1..k]$. For each node in a box the depth of the node is to be stored. The actual representation has to make sure that information on a node with a given depth may be accessed in constant time. Suffix vectors may have two types of edges: natural and regular (normal) ones. A *natural edge* is an edge naturally represented in the vector. A natural edge E with starting position s and end position e is a substring in S with the following characteristics: there is a node with depth d at position $s-1$ and there is another node with depth $d+e-(s-1)$ at position e. Natural edges may correspond to multiple edges in the suffix tree representation. We also have to store a next node value nn for the natural edge. A given node has exactly one natural edge and one or more normal edges running out of that node. The edges running out of a given node represent different possible suffixes in a string similarly to the suffix tree representation. The natural edge is represented by the next node pointer nn associated with the node. For normal edges the start and end positions must be stored. Suffix links may also be stored in the vector. A label L of a node of depth d at position i is the substring $S[i-d+1..i]$. Let a denote a single character and w denote a sequence of characters. A suffix link points from *node x* to *node y* if *node x* is labelled aw and *node y* is labelled w. From the definition it follows that if the depth of *node x* is d then the depth of *node y* is $d-1$. Thus storing a suffix link for a *node x* we only need to store the position of the box where *node y* is stored. In the next section we will show that only one suffix link needs to be stored per box. There is one special node that is not stored in a vector box but it is rather stored separately from other nodes. This node is the *root* node R and it has one outgoing edge for each distinct character in S.

From the above definition one can see that there is a one-to-one mapping between the suffix vector and the suffix tree if suffix tree edges have the smallest possible edge indices, which is the case if the tree is constructed using Ukkonen's linear-time algorithm [Ukk95]. In the following section we analyse some features of a suffix vector, which allows space-efficient representation of the vector.

5.3 Suffix Vector Characteristics

In this section we discover some features of the suffix vector that allow us to actually store less information than we outlined in the previous section. The characteristics described here are independent of the actual physical storage method. Further space reduction may be achieved by using an efficient physical implementation. The details of a space-efficient physical implementation are discussed in later sections.

The following lemma will assist the proof of Theorem 5.1. The characteristics described in Theorem 5.1 will allow us to store only one suffix link value per box.

Lemma 5.1. There may be only one node with *depth d* represented in the box at position *i*.

Let us suppose that there are two different nodes *x* and *y*, both with depth *d* represented at position *i*. These two nodes must represent the same information, that is there is some $j>i$ where $S[j-d+1..j]=S[i-d+1..i]$ and $S[j+1]\neq S[i+1]$. If they represent the same piece of information they are the same node □.

Theorem 5.1. The suffix link of a node always points to another node represented in the same box at the same position, except for the node with the smallest node depth represented in the box.

Let the label of *node v* be *aw* where *a* is a single character and *w* is a sequence of characters. Let us suppose that the suffix link of *node v* points to *node z*. Let *d* be the depth of *node v*. *Node v* is represented at position *i*, thus $S[i-d+1]=a$ and $S[i-d+2..i]=w$. If there is a suffix link between *node v* and *node z* then *node z* has a label *w*. If *node v* is not the node with the smallest node depth then there is a *node y* represented at position *i*, which has a node depth *d-1* and its label is *w* by definition because $S[i-d+2..i]=w$. The suffix link of *node v* points to a node with label *w* and there is only one such node, which is stored in that box (see Lemma 1). Therefore, the suffix link of *node v* must point to *node y*, thus the equality relation *node z* ≡ *node y* follows □.

As one can see, there is still redundant information in the representation shown in *Figure 5.2*:

- *nodes 4, 5,* and *6* are identical
- *nodes 1* and *2* share some edges.

In the first case we can store only one node in the box. This node represents nodes with depths 3 through to 5 using extra information stored in that box (see *Figure 5.3*). We call these boxes *reduced boxes* because the information we have to store is reduced. We can utilise the second observation by eliminating redundant edge information.

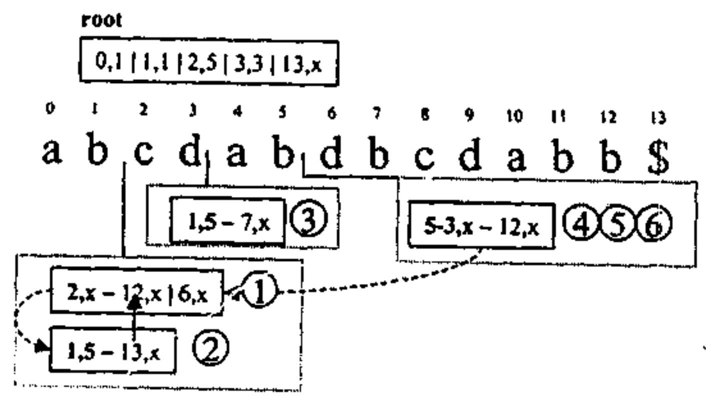


Figure 5.3. Eliminating Redundant Information

We formalise these observations below.

Let us assume that we have *nn* nodes in the box at position *i*. The depth of the deepest node is *d*. If the next node values for each node *d* through to *d-nn+1* are identical and these nodes have the same number of edges and also identical information on those edges, we only have to represent one node in that box. Along with that we need to set a flag, which denotes that this is a reduced box. Thus if we encounter this box during traversal, we know that more than one node is represented here.

The following lemma will assist us in formalising our second observation.

Lemma 5.2. The number of edges running out of nodes stored at the same position *i* monotonically increases as the depth decreases. Formally, let us denote the number of edges running out of the node with depth *d* at position *i* by *E(i,d)*. If $d_1<d_2$ then $E(i,d_1)\geq E(i,d_2)$.

We denote the node with depth d at position i by $i.d$. Let us assume that $i.d$ has m edges running out of that node. If $i.d$ has m edges running out of it, it means that there are $m+1$ occurrences of $S[i-d+1..i]$ in S. Since $S[i-d+2..i]$ is contained in $S[i-d+1..i]$ it has at least $m+1$ occurrences, thus there must be at least m edges running out of node $i.(d-1)$ □.

Based on the above observations we define three different and mutually exclusive cases:

- **Case A**: the node with depth $d-1$ has the same number of edges as the node with depth d and these are the same edges. In this case we simply set their first edge pointers to the same position.
- **Case B**: the node with depth $d-1$ has the same edges as the node with depth d plus some extra edges. In this case, the pointer of the node with depth d will point to the edge in the list of the node with depth $d-1$, where its own edges start.
- **Case C**: the node with depth $d-1$ has different edges to the node with depth d. In this case all the edges must be represented in a separate list. We call these nodes (with depth $d-1$) *large nodes*.

Figure 5.4 illustrates this concept and *Figure 5.3* shows how it applies to our example string.

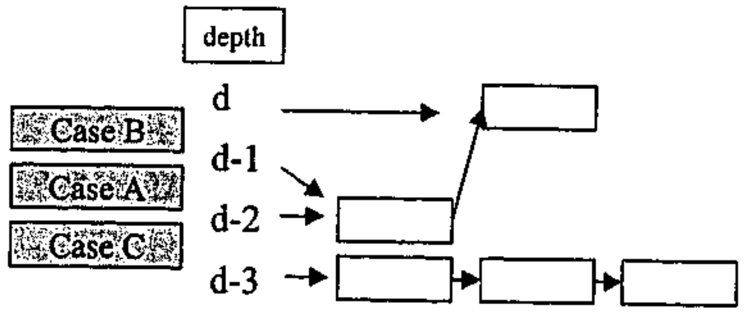


Figure 5.4. The Concept of Large Nodes

Some edges might be eliminated by using directed acyclic graphs (DAG) instead of suffix trees [Gus97] (see also Chapter 4). However, the above rules allow elimination of more edges. First, we show that all edges eliminated by DAGs are also eliminated in the suffix vector representation [MZS02]. In [Gus97] it is proven that two nodes (*node v* and *node y*) can be merged if they are connected by a path of suffix links and the number of nodes in the subtree of *node v* equals the number of nodes in the subtree of *node y*. It is shown in *Figure 5.5*.

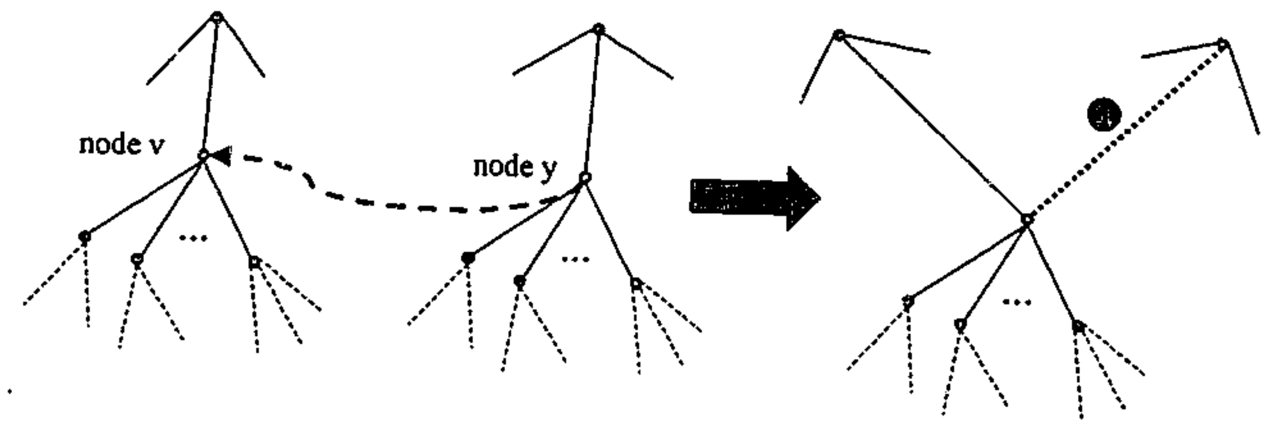


Figure 5.5. Merging Nodes of a Suffix Tree to Create a DAG

We now show that *nodes v* and *y* have exactly the same edges running out of them. Thus if they are eliminated in the DAG representation they are also eliminated in the suffix vector representation. There are three different cases.

Case 1. Let us assume that *nodes y* and *v* have exactly the same number of leaves in their respective subtrees, but there is an edge running out of *node v*, which is not present in the edge list of *node y*. Let us first assume that there are two edges with the same first character but they have different start positions. Let the labels of *nodes y* and *v* be $[y1..y2]$ and $[v1..v2]$, respectively. The depths of the nodes are d_y and d_v, respectively. Since there is a suffix link from *node y* to *node v* we know that $s[y_1+1..y_2]=S[v_1..v_2]$. Let the edge label of the node running out of *node y* be $[y_{11}..y_{22}]$ while the edge label running out of *node v* be $[v_{11}..v_{22}]$. We know that $y_{11}>v_{11}$ and $S[y_{11}]=S[v_{11}]$. This means that the substring $S[y_1+1..y_2]+S[y_{11}]$ appears in the string at position $S[y_{11}-d_y+1..y_{11}]$ and it also appears at $S[v_{11}-d_v..v_{11}]$. We also know that the first occurrence of the string $S[y_{11}-d_y..y_{11}]$ is at position $y_{11}-d_y$ otherwise y_{11} would not be the start position of the edge running out of that node. It means that there will be more edges in the subtree of *node v* than in the subtree of *node y* because under *node v* we will have a leaf with index $v_{11}-d_v$ as well as $y_{11}-d_y+1$. In the subtree of *node y* we will only have $y_{11}-d_y$, which corresponds to $y_{11}-d_y+1$ under *node v*. We do not have a leaf that corresponds to $v_{11}-d_v$. From Lemma 5.2 it follows that *node v* has at least as many leaves in its subtree as *node y* and those leaves will have their indices shifted by one, that is *node v* has more leaves than *node y* if there is a leaf number under *node v* that does not appear under *node y*. We have proven that we cannot have two edges starting with the same character, but having different indices, and running out of *node y* and *v* if they have the same number of leaves in their subtrees.

Case 2. Now let us assume that we have an edge $[v_{11}..v_{22}]$ running out of *node v* that is not present in the subtree of *node y*. Note that because of Lemma 5.2 the only way to have an edge running out of *node y* and not running out of *node v* is Case 1. If we have an edge $[v_{11}..v_{22}]$ running out of *node v* it means that the first occurrence of $S[v_{11}-d_v..v_{11}]$ is at position $v_{11}-d_v$, so there is a *leaf* $v_{11}-d_v$ in the subtree of *node v*. We know that the only way to have equal number of leaves is if *node y* has a *leaf* $v_{11}-d_v-1$ in its subtree. If there is such a leaf it means that the label of *node y* is $S[v_{11}-d_v-1..v_{11}-1]$ and there is an edge with a start position of v_{11} running out of node *y*. It contradicts with our original assumption.

Case 3. Let us assume that we have two edges with the same start positions but different end positions. That is $v_{11}=y_{11}$ but $v_{22}-v_{11}<y_{22}-y_{11}$. $v_{22}-v_{11}$ cannot be greater than $y_{22}-y_{11}$ because it would mean that there are two different positions (*m* and *n*) such that $w=S[y_{22}-d_y-(y_{22}-y_{11})+1..y_{22}]$ has only one possible following ($S[y_{22}+1]$) while *cw* has at least two different followings (*cwa* and *cwb*; *a*,*b*, and *c* denote single characters, while *w* denotes a sequence of characters). If *cw* has more than one following, then *w* must have more than one following. Let us denote the node at the end of edge $[v_{11}..v_{22}]$ by *node z* and the node at the end of edge $[y_{11}..y_{22}]$ by *node x*. Let us denote the start positions of two edges running out of *node z* by z_1 and z_2 and the depth of *node z* by d_z. We know that the leaves z_1-z_d and z_2-z_d are in the subtree of *node z*, thus in the subtree of *node v*. It means that leaves z_1-z_d-1 and z_2-z_d-1 must be under *node y*. In order for this to be true we must have a node at $y_{11}+v_{22}-v_{11}$ and it cannot be y_{22} because $y_{22}-y_{11}>v_{22}-v_{11}$. It contradicts with our original assumption.

In order to prove that these edges are only represented once in the suffix vector structure we have to prove that the end positions of the incoming edges of the two nodes are also the same. If they had different incoming edges they would be represented at two different positions in the vector, thus it would be impossible to share edge information. Once they are in the same box they are stored under each other because they are linked by a suffix link, so either they share edges (Case A) or they are represented as a single reduced box.

Let us assume that $y_2>v_2$, y_2 cannot be less than v_2 because we know that $S[v_1..v_2]=S[y_1+1..y_2]$. $S[v_1..v_2]$ is the first occurrence of $S[v_1..v_2]$ thus $v_2\leq y_2$ is true. Otherwise the label of *node v* would be $S[y_1+1..y_2]$. We know that for each *leaf i* under *node y* there is a corresponding *leaf i+1* under *node v*. *Leaf* v_1 is under *node v*. It means that there must be a *leaf* v_1-1 under *node y*. Otherwise they cannot have the same number of leaves. If there is a *leaf* v_1-1 under *node y* it means that *node y* could have been labelled $S[v_1-1..v_2]$, which contradicts with our original assumption.

We have shown that the extra information eliminated in the DAG representation is also eliminated in the proposed suffix vector representation. We can also prove that the suffix vector representation eliminates even more redundant information. There are two types of redundant information that are still present in the DAG representation. The first is when the edges running out of two nodes connected by a suffix link are identical but the number of leaves in their respective subtrees is different. Such a situation is shown in *Figure 5.6*.

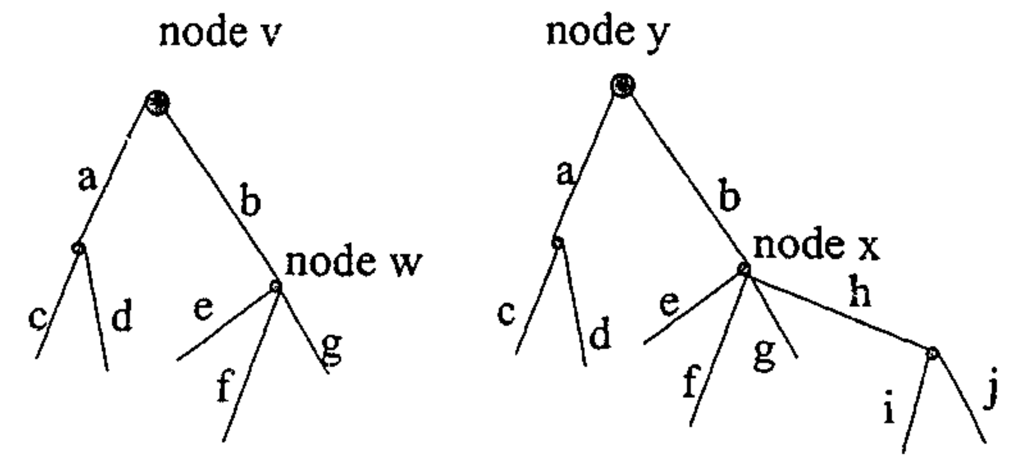


Figure 5.6. Redundant Information in a Suffix Tree

Nodes v and *y* have exactly the same edges running out of them though the numbers of leaves in their subtrees are different. If these two nodes have the same value for the end position of their incoming edges their edges will be represented only once in the suffix vector representation while multiple times in the DAG and suffix tree representation.

The other type of redundant information is exhibited by *node w* and *node x*. We can store the information that *node x* is the same as *node w* except that it also has an *edge h* running out of it. This is represented by Case B in *Figure 5.4*.

5.4 Physical Representation of Suffix Vectors

In this section we describe two alternative physical representations of a suffix vector. The first representation occupies more space but it can be directly built from a string in linear tim· (the construction of the general suffix vector is discussed in Chapter 6). The second representation is the compact physical representation but, because of the actual physical structures used, it cannot be directly built. However, the first representation can be converted to the second one in linear time with minimum overhead. From here on we will refer to the first representation as the *general suffix vector* while the second one as the *compact suffix vector* [MZS02].

In any suffix vector representation the building blocks are boxes. One box stores all the nodes represented at the same position. We have to be able to access those nodes in constant time otherwise the linearity of the algorithms run on the structure will be jeopardised. It means that in each box we have to store the number of nodes represented at that node as well as the depth of the deepest node. We also have to store a pointer to the first edge for each node and a next node value that stores the index of the next node in case the natural edge is followed. Following from Theorem 5.1 we only need to store one suffix link per box.

We start with a few observations that are utilised in either or both of the representations described here. In Section 5.5 we analyse the space requirement of different suffix vector representations and we will prove that the observations made here are true in a practical sense.

Observation 5.1. The node depth of a deepest node can usually fit into 7 bits. (The reason for using 7 bits will become clear when we describe the compact representation but generally we can say that we usually do not need the same range of values to store the deepest node information compared to the stored edge indices.)

Large node depth values represent long repetitions in the string. These are very rare in English texts and also very unlikely in random texts. Representations should not limit the possible node depth values but they may allow storing this information in one byte whenever it is possible.

Observation 5.2. The number of nodes at a given position (in a given box) can usually fit into 7 bits.

This observation is a direct consequence of Observation 5.1, as we cannot have more nodes in a given box than the depth of the deepest node because nodes with the same node depths are stored at different positions.

Observation 5.3. The length of an edge can usually fit into 1 byte.

This observation follows the reasoning of Observations 5.1 and 5.2. Long edges mean long overlaps in the text and you cannot have many long overlaps. If you have many long overlaps it means that you have a long text, so the proportion of edges that are long is still small.

5.4.1 General Suffix Vector Representation

In this representation, for each box we store the number of nodes represented in the box (4 bytes), the depth of the deepest node (4 bytes), an array of pointers to the first edges of each node (the length of the array equals to the number of nodes and each pointer is 4 bytes), an array of next node indices (same length and each index is 4 bytes) and a suffix link value. Using this representation the first edge of a given node can be accessed in constant time if we know the depth of the node.

For storing edges we make use of Observation 5.3. We store the start position of the edge in 4 bytes but the 2 most significant bits have special meaning. If the first bit is set to 1 it means that the edge is a leaf. If the second bit is set to 1, it means that the next node pointer is stored in 1 byte rather than 4 bytes. It does not actually store the position of the next node but rather stores the length of the edge. We also have to store a 4-byte pointer that points to the next edge in the linked list. Using these two special bits we can only use 30 bits to represent the start pointer, which limits the length of the string that we may be able to store. However, it is not a real constraint because using 4-byte pointers we can address 4GB memory space and the general suffix vector representation, as pointed out in the next chapter, uses at least 8 bytes per input symbol meaning that in a 2^{32}-byte memory space we can store a string with length less than 2^{29} characters. The structure of a box in the general representation is shown in *Figure 5.7.*

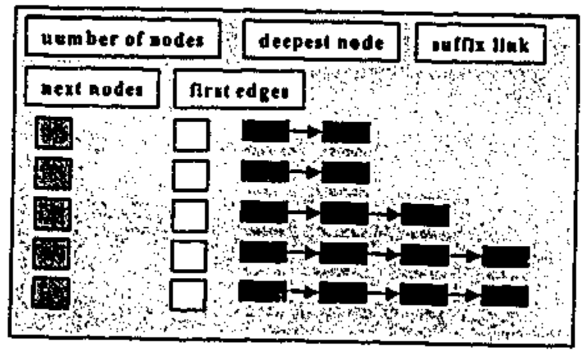


Figure 5.7. A Box in the General Suffix Vector

A reduced box would only store one edge-list and one next node value because they are the same for each node. In the general representation we do not make use of large nodes because they do not allow linear-time construction. The space-requirement of the general suffix vector representation is discussed in Chapter 6 along with the construction algorithm.

5.4.2 Compact Suffix Vector

A compact suffix vector cannot be directly constructed in linear time but it allows us to store our suffix vector in a much smaller space (up to 50% reduction). *Figure 5.8* depicts the compact suffix vector representation that utilises the characteristics discussed in the beginning of Section 5.4.

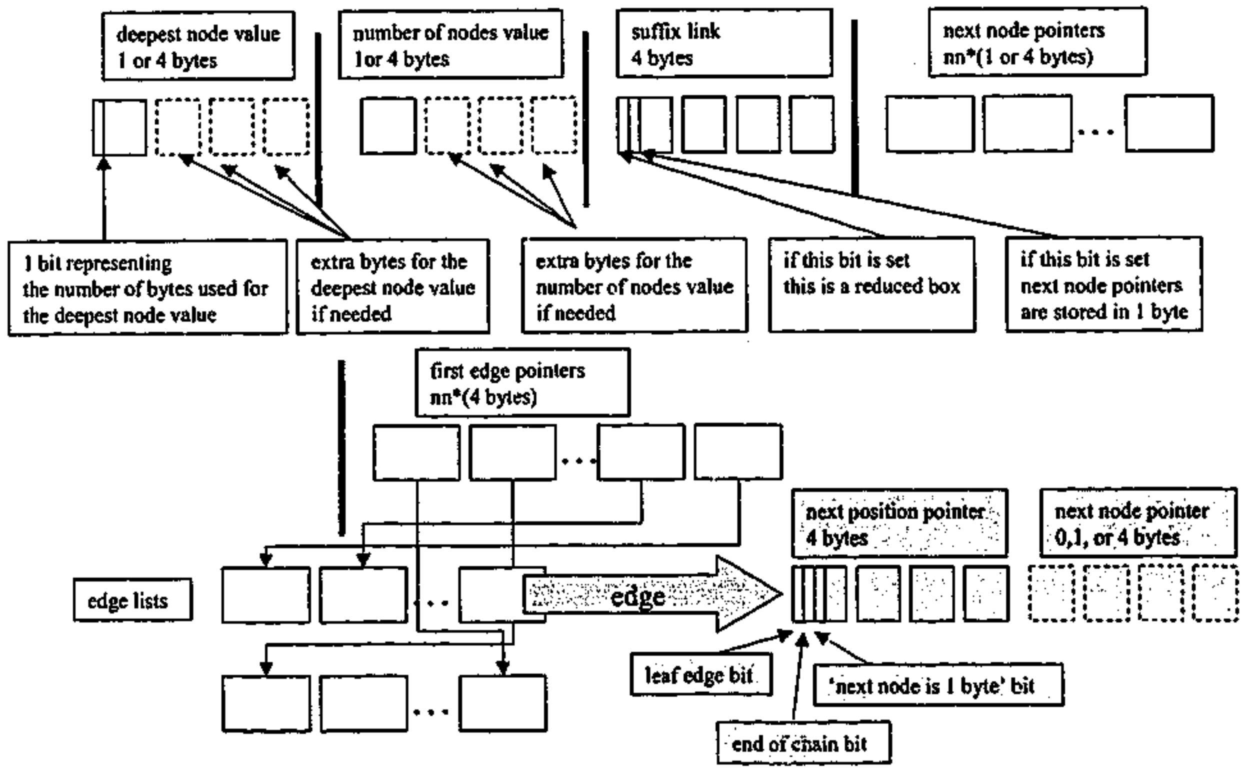


Figure 5.8. Space-Efficient Storage of a Suffix Vector

In each box we have to store three pieces of information that characterize the box rather than individual nodes, so this information must be stored once per box. The first value that we store is the deepest node value representing the depth of the deepest node stored at this position. From Observation 5.1 we know that the depth of the deepest node is usually small, so storing it constantly in 4 bytes is a waste of storage space. We use the first bit to denote the number of bytes needed to store the deepest node value (1 or 4 bytes). Let us denote this value by *l*. The best case for us is when the depth is under 128 because then it fits into the first byte (note that the first bit is used to flag the length of the field). It is very rare that chunks greater than 128 characters are repeated in any text. The number of nodes value uses the same number of bytes (*l*) based on Observation 5.2. It is possible that the number of nodes value is less than the deepest node value. Thus it fits into one byte when the deepest node value does not fit into one byte. However, using another bit to flag this case would unnecessarily complicate retrieval of data and would only save space rarely.

The next piece of information stored in the box is the suffix link value. From Theorem 5.1 it follows that every box needs to store at most one suffix link value. If the number of nodes value equals to the deepest node value it means that the smallest node depth in this box is one character. One-character-deep nodes do not need to store a suffix link. In this case it is not necessary to store a suffix link for the shortest node but we store one anyway because we use the first bit of the suffix link to flag whether this is a reduced box and the second bit to flag whether we have small next node pointers (1 byte) or large next node pointers (4 bytes). If the second bit is set it means that all next node pointers can be stored in 1 byte, so the following pointers for next nodes occupy one byte. Let *s* be 1 if this is a normal box and *number_of_nodes* if this is a reduced box and let *k* denote the length of the fields used for storing next node pointers (1 or 4 bytes).

Next we store the array of next node pointers. Next node pointers point to the node following from this position in case we have to follow the natural edge. Note that depending on the depth of the node stored at this position, the next node value may vary, thus we have to store a next node pointer for each node represented at this position. From Observation 5.3 we know that edges are usually short, which means that we can save space by storing the length of an edge rather than the actual position of the next node. From the length of the edge we can calculate the actual position. As we point out later in this section it is very rare to have edges longer than 256 characters, so we only consider two cases. If all natural edges are short then next node pointers are stored in one byte. If any of the natural edges is long, next node pointers are stored in 4 bytes. In the array we have as many pointers as the number of nodes. The size of the pointers depends on the size of edges as discussed above.

The next piece of information that we have to store is the array of pointers to the list of first edges. The first of these pointers is located at the offset of *(2*l+4+number_of_nodes/s*k)* bytes from the start position of the box. These are physical pointers to a given memory address. We need as many pointers as the number of nodes stored at this position. A pointer points to the address space where the list of edges running out of that node is stored. It is possible that a pointer points to an area and another pointer addresses edges within that area (this corresponds to the cases discussed in Section 5.3).

Each first edge pointer points to an area where the list of edges is stored. Below we will discuss how a list of edges is represented. Each edge must store a next position pointer, which tells the next position in the string where we can follow the matching of a pattern (this is the start position of the edge). We store this information in an integer (4 bytes). The 3 most significant bits of this value are saved for some additional information. The first bit flags whether this edge is a leaf or an intermediate edge. If it is a leaf there is no need to store a next node pointer, so in this case the edge is stored in one integer. The next bit flags whether this edge is the last one in the list or there are more edges to follow. Using this technique we do not need to store edges as a linked list connected by pointers. Rather we can have a fixed-length array and we check for each edge whether this is the last edge in the list. If it is not, we know that the next integer stores another edge. The third bit flags the number of bytes used to store the next node pointer. We follow the same reasoning that we followed in the case of the next node pointers for the box. Edges are usually short, so if they are shorter than 256 characters we store the length of the edge in one byte. If they are longer, then we store them in an integer (4 bytes). The difference here is that we can decide for each next node pointer whether we need one or four bytes. In case of the next node pointers of the natural edges, it is determined for the whole array. By using this technique we can always determine the address of the next edge in the list in constant time from the first 3 bits or we learn that this is the last edge in the list. Let l be 0 if this is a leaf and 1 if it is not a leaf. Let k denote the number of bytes used for the next node pointer for the given edge. The number of bytes needed to store the given edge can be calculated using the formula: $4+l*k$.

5.4.3 Space Requirement of a Suffix Vector

The most space-efficient suffix tree representation so far has been developed by Kurtz [Kur99]. He uses a collection of 42 files of different types to compare his representation to others. We compare our compact representation to his representation in this subsection. Kurtz uses files from the Calgary Corpus [BCW90], the Canterbury Corpus [AB97] as well as DNA sequences used in [LI93]. We do not consider binary files because they are not commonly used in suffix tree applications and there is not enough information to reproduce the generated files PIR500, R500k4, R500k20 [Kur99]. In Kurtz's collection there are English text files (light grey shading in *Table 5.1*: book1, book2, paper1, paper2, paper3, paper4, paper5, paper6, alice29, lcet10, plrabn12, bible, world192, bib, news, trans, cp, xargs, asyculik), formal text[3] (medium grey shading in *Table 5.1*: progc, progl, progp, fieldsc, grammar), and DNA sequences (dark grey shading in *Table 5.1*: ecoli, J03071, K02402, M13438, M26434, M64239, V00636, V00662, X14112). *Table 5.1* shows the space requirement of our and his representations. We have also included the space requirement of compact directed acyclic word-graphs [BBE$^+$85] because it is also a data structure that eliminates some redundant information of suffix trees and later we will point out the similarity between the space requirement of converted directed acyclic word-graphs and our representation.

Table 5.1 shows the total space requirement of the test files. The size of the file and the size of the suffix vector are given in bytes. Bytes per symbol is the average number of bytes needed for each character in the original document. In case of 29 files out of the total of 33 files in the data set, the suffix vector representation is the most space-efficient. For one file (trans) the converted directed acyclic graph representation is the most space-efficient, while Kurtz's representation has the best performance in terms of space requirements for 3 files (bible, m13438, v00636).

We can also analyse the results based on the type of text. There are 19 English text files. In case of 17 files the suffix vector representation is the most space-efficient. In one case the CDAWG representation has the best performance while in another case Kurtz's representation is the most space-efficient.

There are 5 formal text files and in all 5 cases the proposed suffix vector representation is the most space-efficient. The difference between Kurtz's representation and the suffix vector representation here is much higher than in case of English text files. Later on we will analyse why our representation is significantly better in case of formal text files. Here we also note that the CDAWG representation performs much better on formal text than other types of texts.

In case of DNA sequences, the suffix vector representation is the most space-efficient in 7 cases, while Kurtz's representation is the best in case of 2 files. The space requirement of the suffix vector representation is very similar to Kurtz's representation and significantly better than the CDAWG representation.

[3] "formal text" refers to its representation with a grammar and structure, e.g. a programming language

File name	File size	Compact Suffix Vector size	Bytes/symbol Compact Suffix Vector	Bytes/symbol Kurtz	Bytes/symbol CDAWG
book1	768772	7642891	9.35	9.83	15.75
book2	610857	5561505	8.61	9.67	12.71
paper1	53162	489518	8.82	9.82	12.72
paper2	82200	778492	9.10	9.82	13.68
paper3	46527	449325	9.34	9.80	14.40
paper4	13287	130946	9.36	9.91	14.76
paper5	11955	116458	9.35	9.80	14.04
paper6	38106	352260	8.77	9.89	12.80
alice29	152090	1450977	9.15	9.84	14.14
lcet10	426755	3886062	8.81	9.66	12.70
plrabn12	481862	4746891	9.63	9.74	15.13
bible	4047393	34952321	8.53	7.27	10.87
world192	2473401	19293646	7.68	9.22	7.87
bib	111262	938647	8.12	9.46	9.94
news	377110	3422077	8.77	9.54	12.10
progc	39612	356890	8.63	9.59	11.87
progl	71647	603927	8.06	10.22	8.71
progp	49380	416347	8.16	10.31	8.28
trans	93696	755344	7.80	10.49	6.69
fieldsc	11151	95032	8.30	9.78	9.40
cp	24604	211060	8.50	9.34	10.44
grammar	3722	33252	8.72	10.14	10.60
xargs	4228	39897	9.23	9.63	13.10
asyoulik	125180	1226042	9.51	9.77	14.93
ecoli	4638691	58709494	12.51	12.56	23.55
j03071	66496	680746	10.12	12.36	13.44
k02402	38096	484229	12.56	12.59	23.90
m13438	2658	33917	12.62	12.50	23.96
m26434	56738	699190	12.16	12.52	22.52
m64239	94648	1206867	12.59	12.62	23.94
v00636	48503	619250	12.62	12.57	24.04
v00662	16570	212306	12.65	12.69	24.10
x14112	152262	191084	12.39	12.58	23.43

Table 5.1. Comparison of the Total Space Requirement

On average, the suffix vector representation is the most space-efficient for all three types of text. The difference is most significant in case of formal texts, reasonable in case of English texts, and very slight in case of DNA sequences. Here we note again that the suffix vector structure's advantage over Kurtz's representation does not solely lie in its space requirement. In Chapter 6 we describe why it is faster to run algorithms on the suffix vector structure than on Kurtz's representation. The combination of space- and time-efficiency makes the suffix vector superior to Kurtz's representation.

Table 5.2 contains statistical information on the internal structure of the suffix vector. We use these statistical data in proving why our structure requires less space in case of some files and why it requires more in other cases. Data in columns are described below:

File name Name of the file analysed.

File size Size of the file analysed.

Nodes The total number of nodes in the original suffix tree.

Large nodes The number of large nodes in the suffix vector. These are the nodes that need to be fully represented.

Reduced boxes The total number of reduced boxes in the suffix vector. Reduced boxes store one node that represents many nodes at the same position.

Boxes The total number of boxes in the suffix vector.

Edges in tree The total number of edges in the original suffix tree.

Short edges The total number of edges that could be represented in one byte in the suffix vector.

Long edges The total number of edges that needed 4 bytes in the suffix vector, that is edges longer than 256 bytes.

Short depth The number of boxes where the value of the deepest node could be stored in 7 bits.

Long depth The number of boxes where the value of the deepest node could not be stored in 7 bits.

Reduced saved The number of nodes that do not need to be represented because they are represented by a single node in a reduced box.

File name	File size	Nodes	Large nodes	Reduced boxes	Boxes	Edges in Tree	Short edges	Long Edges	Short depth	Long depth	Reduced saved
book1	768772	385296	148527	38304	147381	1154068	439025	0	147381	0	88585
book2	610857	324525	91156	27424	93337	935382	325571	0	93331	6	94815
paper1	53162	29037	7936	2325	8349	82199	29355	0	8349	0	9231
paper2	82200	43210	13430	3983	13961	125410	45559	0	13961	0	12190
paper3	46527	23919	8023	2228	8333	70446	26215	0	8333	0	6448
paper4	13287	6874	2450	607	2514	20161	7719	0	2514	0	1962
paper5	11955	6221	1991	580	2137	18176	6864	0	2137	0	1811
paper6	38106	21088	5718	1743	6057	59194	21143	0	6053	4	7155
alice29	152090	80857	25184	7673	25843	232947	85160	0	25840	3	21920
lcet10	426755	226484	61268	20488	64468	653239	225806	0	64440	28	65301
plrabn12	481862	237072	89905	24788	91717	718934	266915	0	91716	1	56796
bible	4047393	2239780	506475	181205	535114	6287173	2034464	56	535006	108	709244
world192	2473401	1337299	217046	74368	233084	3810700	1122336	162	232301	783	462961
bib	111262	59842	12487	4225	13513	171104	54896	0	13511	2	19931
news	377110	196334	53746	13566	55340	573444	203549	1754	55265	75	59437
progc	39612	21171	5495	1613	5815	60783	21409	0	5814	1	7022
progl	71647	46504	7647	2385	7890	118151	40158	352	7860	30	16466
progp	49380	33065	4697	1594	4956	82445	28067	853	4929	27	11715
trans	93696	66568	7186	2565	7914	160264	49651	2685	7750	164	26446
fieldsc	11151	6585	1184	388	1250	17736	6220	0	1248	2	2119
cp	24604	12810	3055	705	3167	37414	12333	0	3165	2	4197
grammar	3722	2280	460	185	538	6002	1967	0	538	0	910
xargs	4228	2146	665	161	707	6374	2358	0	707	0	611
asyoulik	125180	62743	22547	6396	23394	187923	69692	0	23393	1	16005
ecoli	4638691	2978795	1985866	170433	1528121	7617486	4207235	16717	1527999	122	255040
j03071	66496	53654	15312	1873	12024	120150	56645	138	11998	26	13518
k02402	38096	24364	16573	1422	12705	62460	34701	0	12705	0	2032
m13438	2658	1680	1169	92	889	4338	2443	0	889	0	119
m26434	56738	37957	22641	2257	17515	94695	50723	0	17515	0	4099
m64239	94648	60900	41197	3741	31929	155548	86284	0	31929	0	5184
v00636	48503	30842	21330	1752	16379	79345	44400	0	16379	0	2405
v00662	16570	10689	7296	671	5642	27259	15195	0	5642	0	898
x14112	152262	99847	64136	6260	49369	252109	136350	292	49321	48	10802

Table 5.2. Statistical Data on Suffix Vectors

Before we discuss the reasons for better performance in some cases and poorer in other cases let us analyse that part of data that supports our observations from Section 5.4. Observation 5.1 says that the node depth of the deepest node can usually be stored in one byte. If you consider the 'long depth' column of the table you can

see that in most cases the number of boxes where we need more than one byte to store the depth value is 0. The highest number is recorded for world192. It is 783 boxes out of 233,084 boxes, which constitutes 0.34% of all boxes. Observation 5.2 follows from Observation 5.1.

The results also support Observation 5.3. The number of long edges is 0 for most files. The largest number of long edges can be found in ecoli, which means that this file contains long overlaps but it still only constitutes 0.4% of all edges represented in the vector.

The main advantage of our representation is that redundant information is eliminated. There are two ways of eliminating redundant information. In reduced boxes one node represents all nodes stored at that position. The 'reduced saved' column of the table shows us how many nodes need not be stored because of reduced boxes. Large nodes and nodes in reduced boxes are the only nodes that have to be represented with all their edges. For the rest of the nodes only some edges or even no edges need to be represented in the tree. *Table 5.3* shows the ratio of large nodes, boxes, and reduced nodes saved to the total number of nodes for the three types of files we have analysed.

Type	Large nodes/ Total nodes	Boxes/ Total Nodes	Reduced nodes saved/ Total nodes
English text	23.85%	24.92%	31.05%
Formal text	17.78%	18.65%	34.88%
DNA	65.95%	50.76%	8.92%

Table 5.3. Redundant Information in the Suffix Tree

Firstly, let us explain why we get much better results for formal texts. As one can see from the data in *Table 5.3*, for formal texts we have many nodes represented at each position (low boxes/total nodes value). These nodes share some information and give a chance for longer sequence of small nodes, which is exposed in the relatively low number of large nodes. Also the number of nodes saved in reduced boxes is slightly higher than in case of English texts and significantly higher than in case of DNA.

In case of DNA sequences, we can see that we hardly save any edges because the number of large nodes is close to the total number of nodes. Also we have an average of two nodes represented at a box, which is much lower than in the case of the other two types. It means that we have to represent most of the nodes and most of the edges. DNA sequences have more complicated suffix tree structures than natural English texts or formal texts. This complex structure means that less space can be saved by eliminating redundant information.

English text results are similar to formal text results but the slight difference in the percentage values supports the difference in the average storage space requirements.

We can also see that the higher the average space requirement per input symbol, the closer Kurtz's and the suffix vector representations are. *Table 5.3* shows that higher space requirement stems from less redundant information to eliminate, which supports this correlation.

5.5 Functionally Reduced Suffix Vector

As mentioned earlier there are string-matching problems that can be solved without suffix link information and next node information. Applications that require suffix link information usually also require next node information but there are applications that only require next node information, not suffix link information. An example of the latter application would be finding all occurrences of a pattern in a string. If we traverse down the tree and we find an occurrence, we have to check the subtree to find the exact occurrences. In case the length of the pattern is m and there are k occurrences it takes $O(m+k)$ time using a suffix tree. This can also be achieved by using a suffix vector even if we do not have suffix links. We still need to store next node information on edges to find that information. If we are only interested in one occurrence then we do not even need next node information because next node information is only used when we have to traverse the subtree of the node where the occurrence was found. This is obvious from the description of the exact-matching problem on suffix vectors earlier in this chapter.

In case of the general suffix tree we need suffix link and next node information because we defined this physical representation to be constructible in linear time. In case of the compact representation we can simply leave out either the suffix link information or the next node information. Leaving out only the suffix link information would not save much space but leaving out suffix link and next node information can significantly reduce the space requirement. We call the suffix vector representation, that does not store suffix link and next node information, the *functionally reduced suffix vector*.

Figure 5.9 depicts the physical representation of a functionally reduced suffix vector. In this representation we do not need next node pointers, which are eliminated at both the natural edges and normal edges. Suffix links are also eliminated. There were two special bits in the suffix link of the compact representation. The first bit denoted whether the given box was a reduced box or a normal box. This information is now shifted to the first bit of the number of nodes value. If the deepest node value is stored in 1 byte it can store values up to 127 as discussed earlier in this chapter. We also know that the number of nodes value cannot be greater than the deepest node value, thus the first bit of the number of nodes value can be used as a flag to indicate whether this is a reduced box or not. We do not need the flag indicating the length of the next node values for the natural edge because this information is not present in this representation.

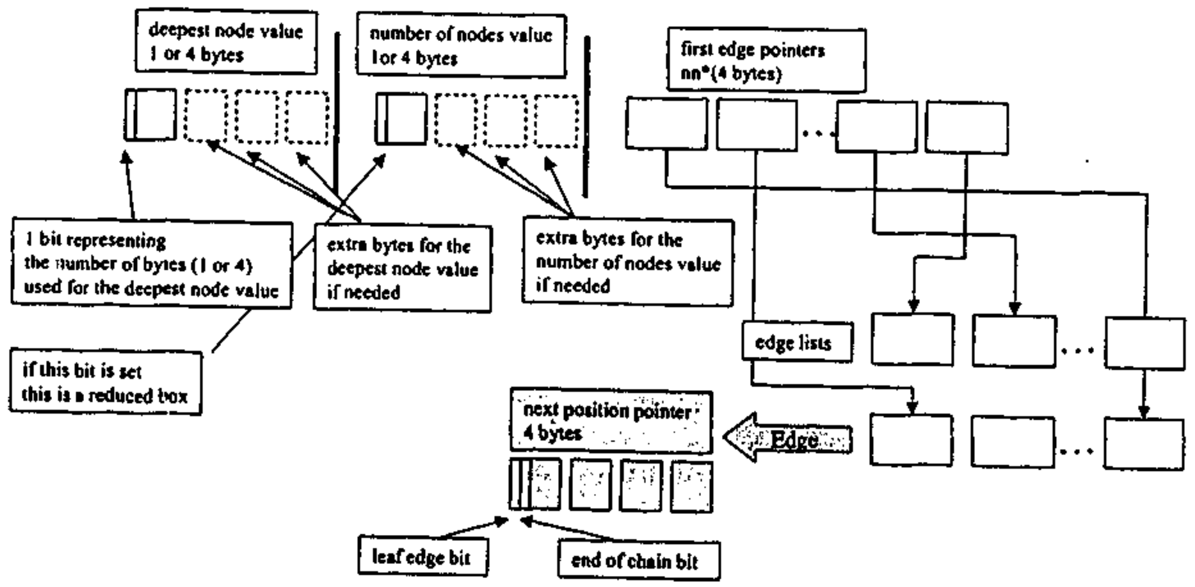


Figure 5.9. The Physical Representation of a Functionally Reduced Suffix Vector

Edges are now stored in 4 bytes and no additional space is required to store the next node information in this representation. Also the bit flagging the length of the next node pointer is eliminated.

This representation is not discussed further because in our document comparison algorithms we only use suffix vectors with suffix link and next node information.

5.6 Summary

We have proposed and analysed the suffix vector data structure in this chapter. This data structure is an efficient replacement for suffix trees. We have presented three different physical representations and compared them to other representations. The general representation does not make use of reduced boxes. Thus it is not as space-efficient as the compact representation. The most space-efficient representation is the functionally reduced suffix vector, but as its name suggests it, it is not as versatile as the other two. A linear time construction algorithm is described for the general suffix vector representation in the next chapter. Not only is the suffix vector representation better in terms of space requirement than any other representation, but it also eliminates some redundant information which is present in other representations. The fact that redundant information need not be analysed multiple times, along with the simplicity of the structure, which allows faster retrieval of information from the structure, makes algorithms using the vector run faster in practice.

We have compared our representation to the most space-efficient representation known to date: Kurtz's representation [Kur99]. We have shown that on average our compact representation performs better. Most of the space reduction in our representation comes from eliminating redundant information. It has formally been proven in this chapter that the information we eliminate is more than that eliminated in directed acyclic graphs.

We have also analysed the internal structure of this representation and explained the differences between results on different types of text.

CHAPTER SIX

Suffix Vector Algorithms

6.

6.1 Introduction

This chapter describes the algorithms that run on the suffix vector structure. In Chapter 5 we have shown that the structure is space-efficient and here we also show that algorithms can run sufficiently fast on the structure. We propose and analyse the construction algorithm that builds the general suffix vector. We also analyse how the matching statistics algorithm can run on the structure.

Section 6.2 describes how the general suffix vector can be constructed from a string in linear time [MZS02]. The algorithm is based on Ukkonen's linear-time suffix tree construction algorithm [Ukk95] but there are a few considerations that are specific to the suffix vector. We prove that the proposed algorithm runs in linear time. After the theoretical presentation of the algorithm the construction steps are demonstrated on an example string. The running time of the algorithm as well as the space requirement of the general suffix vector is presented in this section. We also present a conversion algorithm that converts a suffix tree or a general suffix vector into the compact representation. The pseudo code of the algorithm is presented as well as the proof that the conversion algorithm runs in linear time.

The speed of algorithms run on different representations is mostly determined by the time required to retrieve information from the tree. These issues are analysed in Section 6.3. Suffix vector also has the advantage that it eliminates some redundancy that is present in other representations. These features can be utilised in many algorithms run on the vector including the matching statistics algorithm.

Certain steps that must be carried out on a suffix tree representation are not necessary on a suffix vector representation. As an example we compare two strings using the matching statistics algorithm and we point out the steps that are not necessary in the suffix vector representation.

Section 6.4 summarises the results of this chapter and revisits the key issues presented in this chapter.

6.2 Building a Suffix Vector in Linear Time

As mentioned in Chapter 5 we are able to build a general suffix vector in linear time, which can then be converted to a compact representation in linear time. In Section 6.2.1 we describe a linear-time algorithm that builds a general suffix vector from scratch and Section 6.2.2 will describe how to convert a general suffix vector into a compact suffix vector.

6.2.1 Linear-Time Construction of a Suffix Vector

Since a suffix vector is an alternative representation of a suffix tree we have developed an algorithm that is based on Ukkonen's [Ukk95] linear-time suffix tree construction algorithm. Every step in Ukkonen's algorithm has a corresponding step in our construction algorithm. Building a general suffix vector is more complicated because we have to deal with reduced boxes. The fact that we store multiple nodes in one box makes things more complicated.

Ukkonen's algorithm builds the tree from left to right, which means that each edge label has the least possible indices. In phase *i* the algorithm makes sure that the implicit suffix tree T_i is complete by inserting suffixes *S[0..i]* through to *S[i..i]*. Ukkonen's algorithm in this form would not run in linear time. In each phase we have to make only a limited number of insertions. If we find that *S[j..i]* is already present in T_{i-1} we can conclude that all suffixes from *S[j+1..i]* through to *S[i..i]* are already present. With the automatic extensions of leaves we only have to start inserting the suffixes in phase *i+1* at *j*. The steps spent on reaching the required positions between extensions add up to not more than *3n* steps (*n* is the number of characters in the string) by using suffix links. The above is only a rough sketch of Ukkonen's algorithm. For detailed descriptions see [Ukk95 or Gus97] as well as Chapter 2 of this thesis.

We have already proven in Chapter 5 that suffix links point to nodes in the same box except for the suffix link of the node with the smallest node depth. Suffix links are updated during the construction algorithm. Once a node is created its suffix link must be directed to the next node created during the algorithm. These updates can also be performed in our representation.

In order for the construction algorithm to run in linear time we have to make sure that the size of a box is known when it is created. If a box needs to be extended it means that the information that is stored in the box must be copied to a new location in memory. The amount of data to be copied is proportional to the number of nodes stored in those boxes (in the case of a finite alphabet). In case we need to extend a box at some stage of the algorithm, we have to make sure that the overall amount of data copied in the entire running of the algorithm is proportional to the number of characters in the string.

A box may be extended in two directions. Let us assume that currently we have *x* nodes at position *i* and the depth of the deepest node is *y*. Of course, $x \le y$. Let us assume that we have to add a node with depth *z*. Either $z>y$ or $z \le y-x$. We will show that the latter case is only possible within one phase, and that in the first case, if $z-y>1$ then all other nodes between *z* and *y* will also be added to the box in the same phase. We call the former case *upward extension* and the latter case *downward extension* based on the way they are represented. In the proof of the following theorem we show that only upward extension is possible once a phase is over.

Theorem 6.1. Once a phase is over and a box has been created, it is impossible that a box must be extended downwards, that is adding a node with less depth than the existing node with the smallest depth.

Let us suppose that in phase *i* in extension *j* we have to create a node at position *x* with depth *d*. It means that *S[i-d..i]* was not found in the vector but *S[i-d..i-1]* could be found at *S[x-(d-1)..x]*. We have to create a node at *x* with one edge labelled (*i,end*) where '*end*' means that this is a leaf edge. The phase is not over because the actual substring was not found.

In the next extension we have to make sure that *S[i-d+1..i]* is in the vector. We know that *S[i-d+1..i-1]* is in the vector. There are two possibilities: it is at either *S[x-(d-2)..x]* or some other place, say *S[z-(d-2)..z]*. If it is found at the same position we have to create a node with depth *d-1* at *x* with an edge labelled (*i, end*). If it is found at some different position *z*, it is certain that $z<x$ because each node is stored at the

smallest possible position. If we do not have a node with depth d-1 at z we have to create one. The case when there is a box there but not a node with the given depth is discussed in Theorem 6.2. By the end of this extension we will surely have a node with depth d-1 at z, and we know that $S[x-(d-2)..x]=S[z-(d-2)..z]$, thus any subsequent extension in any subsequent phase would create a node at z rather than x.

It is possible that we keep inserting nodes at x until the end of the phase. A phase can finish in two ways. It is finished because either $S[j_{last}..i]$ is found in the vector or $S[i..i]$ is inserted at the root. In the latter case we have a box at x, which stores nodes with depth d through to 1, so it is obvious that no more nodes can be added below the node with the smallest depth. In the former case the string is definitely not found at $x+1$. If it had been found at $x+1$, $S[j..i]$ would have been found at $x+1$, too. If it has not been found at $x+1$ it means that at some extension we jumped to another *node z*, which leads us back to the previous case □.

Theorem 6.2. Let us denote the depth of the deepest node in a box by d and the number of nodes stored in the box by nn. If we extend a box upwards, that is adding a node with greater depth than the deepest node at the given position, by the end of the phase we will have a continuous range of nodes from the deepest node through to d–$nn+1$.

Let us suppose that at position x we have a box where the deepest node is d and in phase i at extension j we have to add a node with depth $d+e$. It means that the first occurrence of $S[i-(d+e)..i-1]$ is at $S[x-(d+e)+1..x]$. We also know that the first occurrence of $S[x-d+1..x]$ finishes at x otherwise we would not have a node with depth d at position x. In order to have a continuous range of nodes within the box we have to prove that in this phase extensions $j+1$ through to $j+e-1$ will also create a node at position x. There are two reasons for not creating a node at x for these extensions: either the phase finishes before $j+e-1$ or the first occurrence of $S[i-(d+k)..i-1]$ is at position $z<x$ for some $0<k<e$. Since in the former case each extension tries to find an edge with label $S[i]$ running out from x and there are no nodes with the given depth in that box, the only possibility is the natural edge, that is $S[x+1]=S[i]$. If this were true we would not have started extension j because $S[j..i]=S[i-(d+e)..i]=S[x-(d+e)+1..x+1]$ would have held. It means that the phase cannot finish between extensions j and $j+e$. The latter case is also impossible because we know that the first occurrence of $S[x-d+1..x]$ finishes at x and the first occurrence of $S[x-(d+e)+1..x]$ finishes at x, which means that all substrings between the two will finish at x □.

We have shown that a box can only be extended upwards, so we have to prove that the overall number of steps involved in copying those nodes is proportional to the total number of characters. We may also need to copy nodes when a reduced box becomes a regular box. This involves creating nodes that have not been created yet only because we could store them in one place. Thus these copying actions create new nodes that are included in the linear time-bound of Ukkonen's algorithm.

Before proving that node extensions do not violate the linear running time of the algorithm we prove the following corollary that is the result of the previous theorems.

Corollary 6.1. Every box starts off as a reduced box. The only possible difference between the nodes in the box is next node positions.

As we have already shown in Theorem 6.2, by the end of a phase we always get a continuous range of nodes. If a new node has to be created it means that a new edge running out of that node has to be created, too. This edge will be the same for each node at the given position: (i,end). The next node pointer belonging to a given depth is determined by the next node pointer of the edge that was split in this extension. This may vary from node to node within a box □.

In our algorithm, whenever a new box needs to be created, we start with a reduced box and we split that reduced box only if it is necessary. It may become necessary because a new edge is added to some intermediate node or the next node pointer must be changed for some intermediate node. If any changes need to be made to the deepest node, we can apply those changes to the only node in the reduced box because we are certain that those changes will need to be propagated to the rest of the nodes later in that same phase. The only thing we have to be careful with is that, when we add some extra information to the node in the reduced box, in the next extension we will try to add that same information to some intermediate node. We have to indicate that this new information is only there because of the nature of the reduced box. It means that, though the actual extension does not need to be completed, it does not stop the phase, as it would immediately cease when an extension was not necessary.

Our rule is that whenever some extra information needs to be added to an intermediate node of a reduced box, we split up that box and create a regular box with the given number of nodes. It may happen that later on, that extra information will also appear at all nodes, in which case it could become a reduced box again. We make sure that these reduced boxes are identified in the conversion phase when we convert our general representation to the compact representation. The conversion algorithm is described in Section 6.2.4.

There is only one bit missing of proving that our algorithm is linear in time. We still have not proven that the copying steps involved in box extensions are proportional to the length of the string. We know from Theorem 6.1 that a node cannot be extended downwards. It means that the only possible extension is when we have a box with a deepest node of depth d, and a new node needs to be created with depth $d+z$ where $z>0$. The following theorem formalises these ideas.

Theorem 6.3. During the general suffix vector construction algorithm the overall number of steps l involved in copying nodes is proportional to the length n of string S the suffix vector is built on, when a box with deepest node of depth d at position i must be extended upwards.

We assume that we have a finite alphabet. Then copying one node from one memory location to another can be done in constant time and the constant is determined by the size of the alphabet. Given this fact we have to prove that the number of nodes to be copied during the entire algorithm run is proportional to n.

Here we prove that each action of copying a node can be associated with a unique leaf. Since the number of leaves is equal to the number of characters in the string, this guarantees that no more than n copying actions will be carried out during the construction. In the proof we use both the tree and the vector[4] representation for demonstration, whichever is more appropriate.

Let us assume that we are in phase i at extension j, that is $S[j..i]$ must be added to the tree and we are currently in a position in the vector that has a label of $S[j..i-1]$. In *Figure 6.1* the new node is depicted with a lighter colour and the details of the paths running to nodes are depicted by a single line. In the worst case, we already have $i-j-1$ nodes at our current position f in the vector. It means that $i-j-1$ nodes must be copied in this extension. We associate the copying of the node of depth $i-(j+1)$ with leaf $j+1$ ($j+1$ is the first character in the path to the node), node of depth $i-(j+2)$ with leaf $j+2$, etc. *Figure 6.1* depicts that in the tree it will mean that on the path

[4] The expressions "*suffix vector*" and "*suffix tree*" are often used intcichangeably in this thesis because of their one-to-one correspondence.

running out of node labelled $S[j+1..i]$ we have to create a leaf with label $j+1$. It is created either in the next extension of the same phase or it is found in the vector. During some later phase it still has to be created because each position in the string must have an associated leaf in the tree.

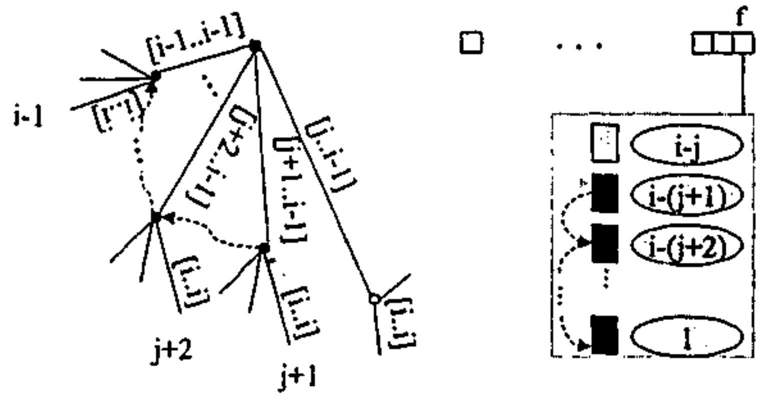


Figure 6.1. Associating Copying Activities with Leaves

It may be clear from this association that every leaf has only one copying action associated with it. We give a formal proof below. The idea that different copying actions have different leaves associated with them is illustrated in *Figure 6.2.*

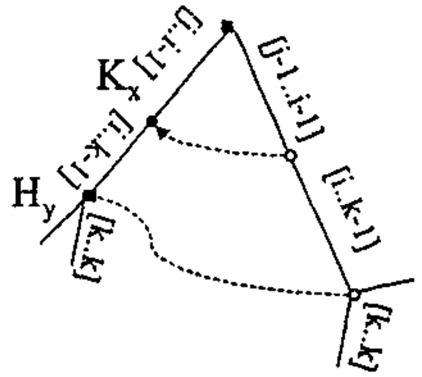


Figure 6.2. Different Copying Actions Have Different Leaves Associated with Them

Let us assume that the copying of two different nodes denoted by K_x and H_y, respectively, have the same leaf j associated with them. The leaf associated with copying a node is always in the subtree of the node, so both K_x and H_y must be on the path to leaf j. This means that at some stage we had to include $S[j-1..i]$ and $S[j-1..k]$ in the tree. Without loss of generality we may assume that $i<k$, that is node H_y is deeper than node K_x. From Ukkonen's construction algorithm [Ukk95], we know that it is only possible in two different phases. However, if they are inserted in two

different phases, that means that *S[j-1..i]* was found in the tree at phase *i*, extension *j-1*. Otherwise it would have meant that the next extension is *j..i*. Thus there would not be an extension *j-1..k*. This means that it could not have possibly generated the copying action of K_x, which contradicts with our initial assumption that it is associated with leaf *j*. Note that this conclusion also proves the situation when $K_x \equiv H_y$ because the same node cannot be copied multiple times in the same extension □.

6.2.2 Performance Results of the General Suffix Vector

In Section 6.2.1 we have formally proven that the construction algorithm that builds the general suffix vector is linear in time. In this section we present the practical results. We analyse the running time of the construction algorithm as well as the space requirements of the general suffix vector representation.

Table 6.1 contains the running time of the construction algorithm on our test machine (Intel Pentium II 433MHz, 128M RAM, Windows 2000 Professional). Following is a description of what is shown in each column:

File name The name of the file the suffix vector was generated on.

File size The size of the original file.

General Suffix Vector Size The size of the general suffix vector created from the file.

General Suffix Vector (bytes/input symbol) The average space requirement of the general suffix vector (bytes per input symbol).

Construction Running Time The running time of the construction algorithm in milliseconds.

Input characters / ms The number of input characters processed on average in a millisecond.

Bytes generated / ms The number of bytes generated on average in a millisecond. (These are the bytes occupied by the general suffix vector).

File name	File size	General Suffix Vector Size	General Suffix Vector (bytes/input symbol)	Construction Running Time	Input characters / ms	Bytes generated / ms
book1	768772	11102805	14.44	5789	132.80	1917.91
book2	610857	8409961	13.77	4734	129.05	1776.63
paper1	53162	727482	13.68	407	130.62	1787.43
paper2	82200	1170267	14.24	607	135.35	1926.89
paper3	46527	671955	14.44	377	123.41	1782.37
paper4	13287	194397	14.63	107	124.57	1822.47
paper5	11955	170968	14.30	100	119.55	1709.68
paper6	38106	523341	13.73	311	122.53	1682.77
alice29	152090	2185507	14.37	1188	127.99	1839.14
lcet10	426755	5862134	13.74	3576	119.34	1639.30
plrabn12	481862	7170978	14.88	4323	111.46	1658.80
bible	4047393	51453499	12.71	33562	120.59	1533.09
world192	2473401	28723455	11.61	20685	119.57	1388.61
bib	111262	1411371	12.69	831	133.89	1698.40
news	377110	5065889	13.43	3323	113.48	1524.49
progc	39612	522901	13.20	327	121.14	1599.09
progl	71647	898341	12.54	464	154.41	1936.08
progp	49380	616040	12.48	320	154.15	1923.12
trans	93696	1204981	12.86	785	119.36	1535.01
fieldsc	11151	139420	12.50	77	145.45	1818.52
cp	24604	332458	13.51	198	124.26	1679.08
grammar	3722	46894	12.60	23	159.51	2009.74
xargs	4228	59260	14.02	30	140.93	1975.33
asyoulik	125180	1839918	14.70	1031	121.38	1784.02
ecoli	4638691	88993178	19.18	31562	146.97	2819.63
j03071	66496	1073381	16.14	462	143.93	2323.34
k02402	38096	786963	20.66	277	137.53	2841.02
m13438	2658	55148	20.75	17	159.48	3308.88
m26434	56738	1136057	20.02	404	140.44	2812.02
m64239	94648	1964668	20.76	718	131.88	2737.58
v00636	48503	1008246	20.79	357	135.74	2821.58
v00662	16570	345741	20.87	117	142.03	2963.49
x14112	152262	3088828	20.29	1142	133.37	2705.54

Table 6.1. Running Time of the Construction Algorithm

The values in the column 'input characters per ms' show that our algorithm runs in linear time. The lowest value is 111.46 characters per millisecond and the highest value is 159.51 characters per millisecond. The latter belongs to file 'grammar' and it is small enough to fit into the cache. That is why we experience significantly better performance. The chart in *Figure 6.3* shows the running time of the algorithm for different file sizes.

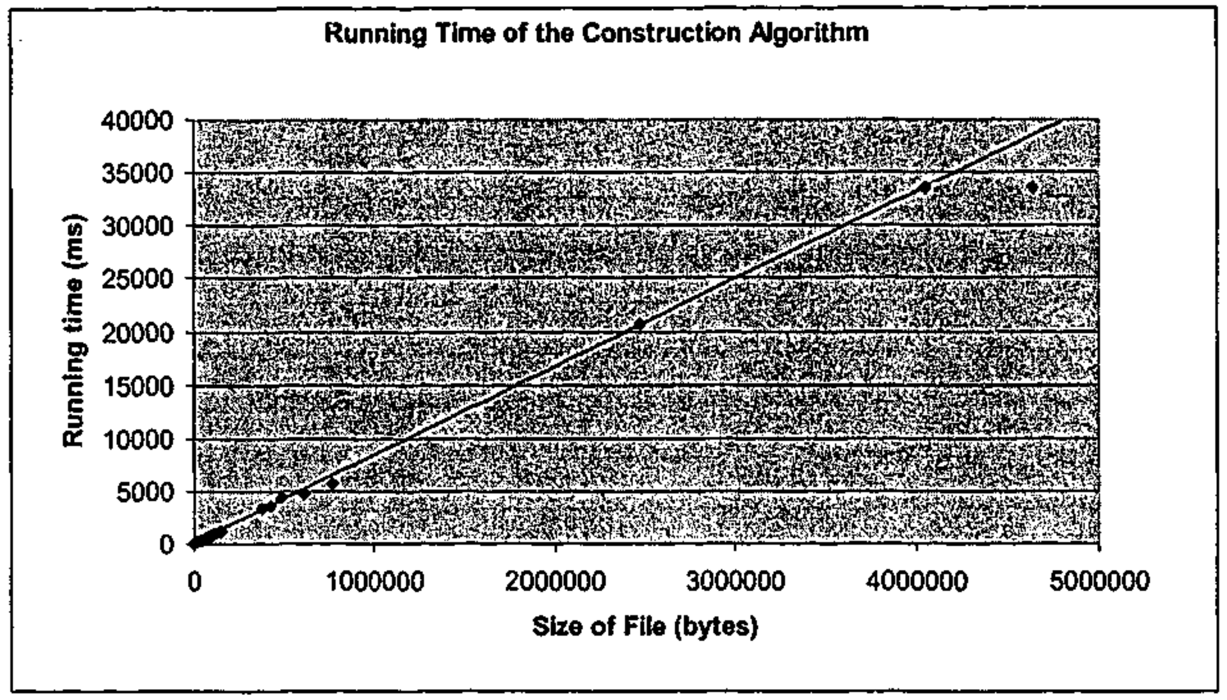


Figure 6.3. Running Time of the Construction Algorithm

6.2.3 Construction of the Suffix Vector on an Example String

In this section we show how the construction algorithm works on our example string. We build the general suffix vector for string *S='aabbbabbbabba$'*. We will refer to the theorems and observations of Section 6.2.1 whenever it is relevant. For each phase we show the current state of the suffix vector after the phase has finished. *Figures 6.4.(a)* through to *6.4.(j)* show these transformations.

Phase 1. (*Figure 6.4.(a)*) This is the beginning of the construction. We have to insert the edge *S[0..0]* into the vector. Since this is the first edge it will run out from the root. *Figure 6.4.(a)* depicts the suffix vector data structure after *S[0..0]* is inserted. Our current position in the vector is depicted by the arrow under the string. The shaded box with the solid border depicts the first substring that we had to insert into the tree (the first extension of this phase). The shaded box with the dashed border depicts the last substring that we had to explicitly insert in this phase (the last extension of the phase). The new edge from the root was created with the indices *(0,x)*, which means that the edge starts at *position 0* and it is a leaf.

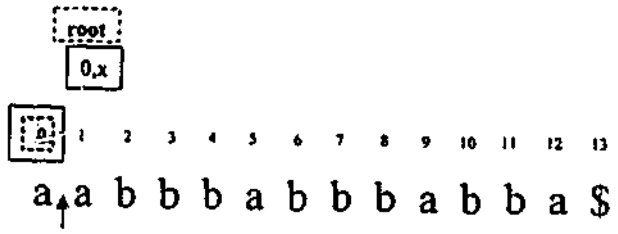


Figure 6.4.(a). Phase 1.

Phase 2. (*Figure 6.4.(b)*) Since Phase 1 finished with inserting a 1-character edge, in Phase 2 we have to insert *S[1..1]*. We go back to the last visited node, which is the root node in this case, and we have to follow its suffix link. Since this is the root, there is no suffix link, thus we have to find *S[1..1]='a'* from the root. We check the edges running out of the root and we find that edge. Since this edge has been found in the vector explicit extensions are unnecessary. There was no last explicit extension in this case. That is why the dashed box does not contain any indices. Our current position in the vector is still *position 0* (see the arrow below the string).

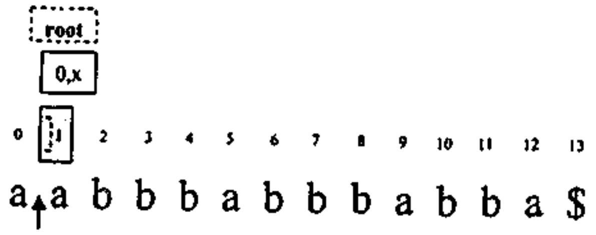


Figure 6.4.(b). Phase 2.

Phase 3. (*Figure 6.4.(c)*) We have found *S[1..1]* in the string, so in this phase the first extension will involve inserting *S[1..2]*. Our current position is 0, thus we have to check whether there is a continuation here that starts with *S[2]='b'*. First, we check the natural edge *S[1]*, however this does not match. A new box with a new node must be created here. The depth of the node is 1 because we arrived here when traversing the path of *S[1..1]*. The new node will have a new edge labelled *(2,x)*. The new box and the new edge are depicted by dashed borders. The first number in bold (1) says that the depth of this node is 1. The 'x' in bold means that following the natural edge there is no next node, which means that this is a leaf. When we created this node, we broke up the original edge *(0,x)* running out of the root, thus the next node on the natural edge inherits the next node information of the original edge *(x)*. The new node will become the next node of the original edge, thus the next node of edge *(0,x)* must be changed to the current position (the new edge is *(0,0)*). The suffix link of this node does not need to be set because this is a node with depth 1. Since the substring *S[1..2]* was not yet found in the vector we have to explicitly do the next extension of this phase, which is finding the string *S[2..2]* in the vector. The last visited node is the root since the newly created node need not be considered. We have to find an edge starting with *S[2]='b'* from the root. There is no such edge, so we have to create one. The new edge will be *(2,x)* and it is depicted by a dashed box. There are no more extensions in this phase. The first and last extensions are depicted with solid and dashed boxes, respectively.

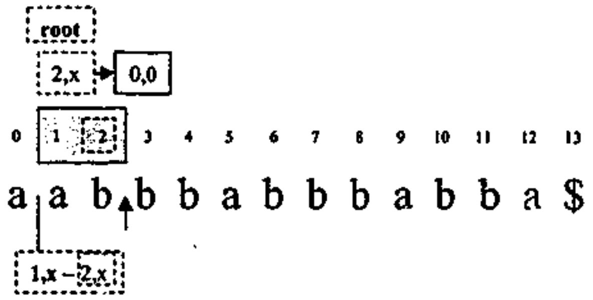


Figure 6.4.(c). Phase 3.

Phase 4. (*Figure 6.4.(d)*) Since Phase 3 finished with inserting a 1-character edge, the first extension of Phase 4 is inserting *S[3..3]*. Our current position in the vector is 2 and the last visited node is the root. Thus we must go back to the root. We have to find an edge that starts with *S[3]='b'*. This edge is *(2,x)*, which means that this extension is done implicitly and it marks the end of this phase.

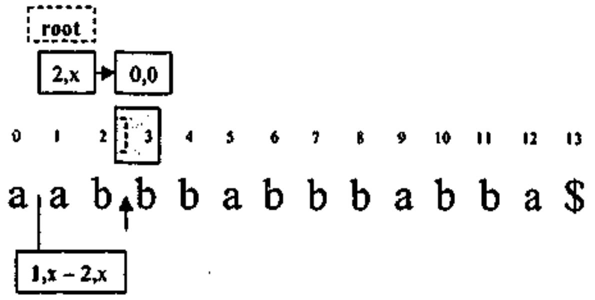


Figure 6.4.(d). Phase 4.

Phase 5. (*Figure 6.4.(e)*) In Phase 4 we have found *S[3..3]*. Therefore the first extension of this phase involves finding *S[3..4]*, which means that from our current *position 2*, we have to check if there is a continuation with *S[4]='b'*. F'rst, we check the natural edge and find the next character *S[3]='b'*, thus this extension is done implicitly. The current position pointer is incremented. We are required to examine whether the last visited node needs to be updated. Updating is unnecessary, as the edge we are currently on is *(2,x)*, thus there is no next node.

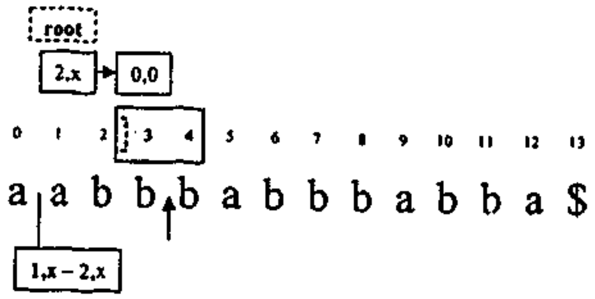


Figure 6.4.(e). Phase 5.

Phase 6. (*Figure 6.4.(f)*) In Phase 5 we have found *S[3..4]*, thus the first extension in this phase involves finding *S[3..5]* in the vector. Our current position is *3* and we have matched two characters *(3,4)*. We have to check whether there is a continuation with *S[5]='a'*. The natural edge does not match and there is no box at this position, thus we have to create a new box with a new node, which has an edge *(5,x)* running out of it. The new box and the new edge are depicted by dashed boxes. The next node pointer of the natural edge is inherited from the original edge *(2,x)*. The next node pointer of the original edge becomes 3. This box is a reduced box because during the construction algorithm each box starts off as a reduced box (see Corollary 6.1). The issue of a reduced box was not relevant in the previous case, as the deepest node there had a depth of 1. If the next node had to be created at this position, we would only need to update the number of nodes value and check whether the next node values match.

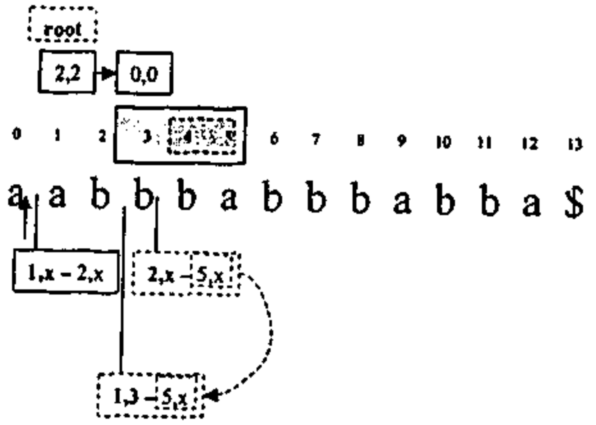


Figure 6.4.(f). Phase 6.

In the next extension we have to find *S[4..5]* in the vector. We know that *S[4..4]* is in the vector, hence we must traverse down to its position. The last visited node is still the root. We find *S[4..4]* on edge *(2,3)*, our current position is 2. From there we must check whether there is a continuation with *S[5]*. The natural edge does not match and there is no box at this position. A new box with a new node has to be created with an edge *(5,x)*. This new box is also depicted by a dashed box in *Figure 6.4.(f)*. The new box will inherit the next node pointer from the original edge *(2,3)* and the next pointer of the original edge will be our current position *(2)*. We created a new node, which means that the suffix link of the node created in the previous step must be directed to this new node. The suffix link is depicted by the dashed arrow between the two new boxes.

In this extension we had to create a new box at a different position and not a new node at the same position as the last extension. It means that our expectation that a new node will be created at the same position was not realised. Nothing was lost by creating the box at position 3 as a reduced box. Later in the construction algorithm we will find situations where this basic assumption holds true.

Since *S[4..5]* has not been found we have to do the next extension, that is finding *S[5..5]* in the vector. We have to go back to our last visited node, which is still the root. It means that we have to find *S[5]='a'* running out of the root. There is an edge *(0,0)* which starts with *'a'*. Thus this extension is implicitly done, which marks the end of this phase.

Phase 7. (*Figure 6.4.(g)*) In Phase 6 we have found *S[5..5]*, thus the first extension in this phase involves finding *S[5..6]* in the vector. Our current position is 0. We have to check whether there is a continuation with *S[6]='b'* from this position. The natural edge does not match but there is an edge running out of the node in this box *(2,x)*. We update our last visited node to the node at position 0 and set our current position to 2. The last visited node is depicted by a dashed box.

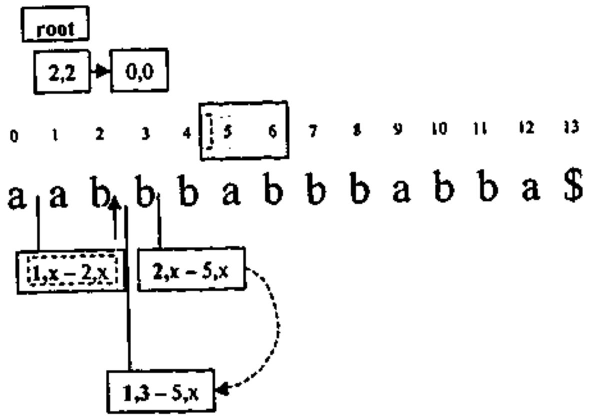


Figure 6.4.(g). Phase 7.

Phases 8.-12. (*Figure 6.4.(h)*) These phases are similar to Phase 7. The next character is always found on the natural edge, thus no explicit extensions need be done.

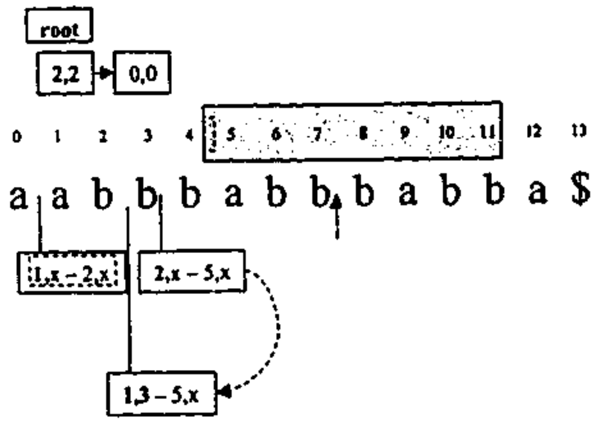


Figure 6.4.(h). Phases 8-12.

Phase 13. (*Figure 6.4.(i)*) In Phase 12 we have found *S[5..11]*, thus the first extension in this phase involves finding *S[5..12]* in the vector. Our current position is 7. We have to check whether there is a continuation from here with *S[12]='a'*. The natural edge (*S[8]='b'*) does not match and there is no box at position 7, which means that a new box and a new node with a new edge need to be created at position 7. We create a reduced box, which represents only one node at the moment. The new node inherits the next node pointer from the original edge *(2,x)* and the new next node pointer of the original edge becomes 7. This is the single node at position 0.

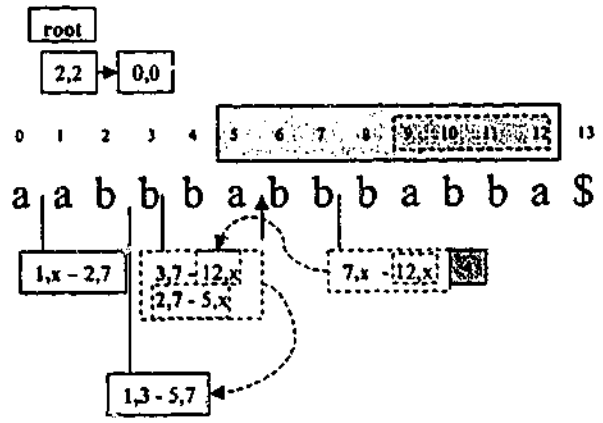


Figure 6.4.(i). Phase 13.

In the next extension the algorithm has to insert *S[6..12]*. The last visited node was at position 1 with depth 1. Since this is a node with depth 1 its suffix link points to the root. We know that *S[6..11]* is already in the vector, thus we first find an edge that starts with *S[6]='b'*. This is edge *(2,2)*. This is a one character edge, which means that we have to find *S[7..11]* starting after position 2. First, we check the natural edge (character 3) this matches. Thus we follow the natural edge. We have to look up the next node information from the box when we follow the natural edge. The next node is *x*, which means we are on a leaf and we can immediately jump *11-7+1=5* positions, which places us at position 7. We have to check whether there is continuation with *S[12]='a'* from here. We find that there is none. We have created a reduced box at position 7 in the previous extension. Now we must create a node at the same position with a depth value *d-1=3*. This would have the same edge running out *(12,x)* and the next node pointers of the natural edge also match because the next node pointer of this node is inherited from the original edge (*x*). Since this is a reduced box, we simply increment the number of nodes value to 2. Our last visited node was at position 2, the natural edge was followed, and subsequently, we updated its natural edge next node pointer to 7. This is in the box at position 2 but this value is changed later in this phase. That is why 3 is shown in *Figure 6.4.(i)*. The suffix link is not updated as a node was created at the same position and we know from Theorem 5.1 that only the suffix link of the node with the smallest depth needs to be explicitly stored.

The following two extensions are similar to the previous one. The first extension updates the next node pointer at position 3, while the second updates the next node information of the normal edge stored in box 2. The number of nodes value of the reduced box at position 7 becomes 4, and our last visited node is the node at position 2.

Since our last visited node is one-character deep we must match the characters from the root. In this extension *S[9..12]* must be inserted into the vector and we know that *S[9..11]* is already there. We start traversing from the root. *S[9]='a'* matches edge *(0,0)* at the root. We jump to the node at position 0 finding that we must follow the edge *(2,x)* from there, which places us at position 3. This means that the first occurrence of *S[9..11]* is at *S[1..3]*. We have to check whether there is a continuation with character *S[12]='a'* from here. The natural edge does not match, however there is a box at position 3.

The deepest node of this box has a depth of 2 and we have matched 3 characters. It means that there is no possible continuation from this node. In this situation we already have a box at a given position but this box needs to be extended upwards. This is the case that has been discussed in Theorems 6.1, 6.2, and 6.3. We have to create a new box, which has a deepest node value of 3, and the number of nodes stored in this box will become 2. The new box and the new edge are depicted by dashed boxes in *Figure 6.4.(i)* at position 3. The last created node was at position 7, which means that the suffix link of the box at position 7 must be directed to this box.

We show how the leaf numbers are assigned to copying a node as it is described in the proof of Theorem 6.3. Note that assigning this leaf is not part of the algorithm. This merely demonstrates the idea utilised in the proof of Theorem 6.3. The copying of this node is associated with the leaf with an index *j+1* where *j* is the index of the currently inserted leaf. We are currently inserting leaf *9*, thus we assign *10* to the copying of the currently existing node at position 3.

In the next extension we have to insert *S[10..12]* into the vector. Our last visited node is at position 0 with depth 1, thus we have to find *S[10..11]* starting from the root. Edge *(2,2)* matches *S[10]='b'*. From there we can follow the natural edge, which places us at position 3. We must determine whether there is a continuation from here that starts with *S[12]='a'*. The natural edge does not match but there is an edge *(5,x)* that matches the next character. This means we have found *S[10..12]* in the vector, therefore this phase is over. The last visited node is updated, which is the node at position 2 with depth 2.

Phase 14. (*Figure 6.4.(j)*) In Phase 13 we have found *S[10..12]*, thus the first extension in this phase involves finding *S[10..13]* in the vector. We are at position 5, the natural edge does not match *S[13]='$'* and there is not yet a box here, meaning that we must create one. A reduced box is created, which is depicted by the dashed box at position 5. We must update the next node pointer of the original edge. The original edge is the edge at position 3 with depth 2.

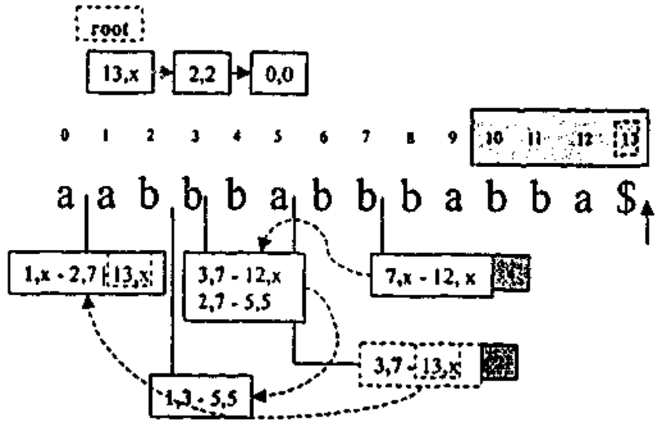


Figure 6.4.(j). Phase 14.

In the next extension the original edge is the edge of the node at position 2. Again we must create a new node at position 5, meaning that we increment the number of nodes value of the reduced box because the next node pointers match. The next node pointer of the original edge *(5,7)* has to be updated to 5.

In the next extension *S[12..13]* must be inserted. *S[12..12]* is found at position 0 and there is already a node with depth 1, although there is no edge starting with *S[13]='$'*, consequently this new edge is added. The new edge is depicted in *Figure 6.4.(j)* by the dashed box. The last created node was at position 5, which means that its suffix link has to be directed to this node.

In the next extension, *S[13..13]* must be found starting from the root. This is not the case. As a result, a new edge needs to be created with label *(13,x)*.

We completed the last phase and the suffix vector has been built.

6.2.4 Converting a General Suffix Vector into a Compact Suffix Vector

There are a few differences between the two proposed suffix vector representations. These differences are discussed in this section. A conversion method, that converts the general representation to the compact representation, is described for each difference. In order to make sure that the conversion algorithm works in linear time we prove that none of the nodes or edges are examined more than a constant number of times and that none of the edges or nodes are copied more than once.

Edges are stored as linked lists in the general suffix vector, and stored as fixed length byte arrays in the compact representation. The conversion between the two is straightforward. We must examine the linked list in one sweep to calculate the overall size of the byte array that stores the edges. A memory area must be allocated for the array and then the edge information must be filled in.

The depth of the deepest node in the box and the number of nodes stored in the box can occupy 1 or 4 bytes each. The first bit of the first byte of the box object is reserved to flag whether these values are stored in 1 byte (the remaining 7 bits of the first byte) or in 4 bytes. This information can be retrieved directly from the actual deepest node value in the general representation.

The next piece of information to be stored is the suffix link. The first two bits have special meaning. If the first bit is set this is a reduced box, which means that it represents more than one node. However, all the nodes contain identical information.

Reduced boxes are flagged by a bit in the general representation, so this information can be retrieved directly. The second bit indicates whether all next node values can be stored in 1 byte rather than 4 bytes. Proceeding through all next node values stored in the given box we can set this flag accordingly. We set the actual next node values by examining the next node values in the general representation. In the case of a reduced box there is only one next node value.

The most computationally expensive part of the conversion is to identify large nodes and set first edge pointers accordingly. The first node, which is the deepest, is always a large node. For a definition of large nodes see Section 5.3. The way the general suffix vector is constructed ensures that if two consecutive nodes have common edges (edges with the same start and end pointer) they appear in the same order. If the difference between two nodes is only extra edges they can be represented in the same list of edges by setting the pointers accordingly (Case B in Section 5.3). Of course, if all their edges are identical, they can use the same list of edges (Case A in Section 5.3). Comparing two edge-lists can be performed in constant time as identical edges appear in the same order. We begin comparing the edges of the deepest node *d* with the node with depth *d-1*. If they can be represented in the same edge-list (Case A or Case B) we then compare the node with depth *d-1* with the node with depth *d-2*. We continue until Case C applies or we reach the node with the smallest depth in the box. If Case C applies at a node with depth *x* we can create one edge-list for nodes down to depth *x+1*. The lower the node is in the list, that is, the less deep the node is, the more edges it will have. This means that first we must store those edges that appear in node with depth *x+1* but do not appear in node with depth *x+2*. Again this can be done in constant time because of the order of the edges. These steps are repeated until we reach the node with the largest depth value.

From here on, the node with depth *x* will act as the deepest node and the steps described above are repeated. By the time we reach the node with the smallest depth all first edge pointers will be set properly. *Figure 6.5* shows what the edge-lists look like by using large nodes.

It is possible that by the end of this step we find a reduced box that was not originally flagged as a reduced box. Two conditions must be met. Firstly, all next node pointers must be equal. Secondly, the deepest node must be the only large node, and Case A must apply between all consecutive nodes. If these criteria are met we can declare this box as a reduced box and change data accordingly.

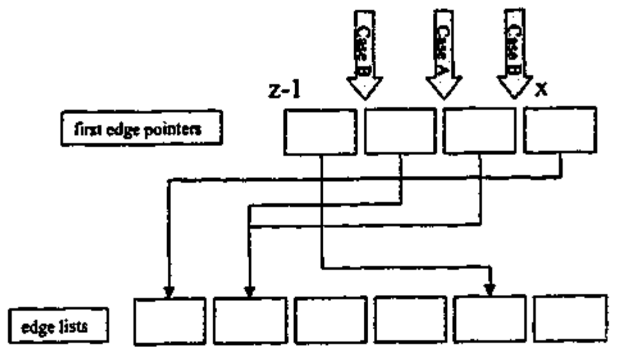


Figure 6.5. Utilizing the Concept of Large Nodes

The space-overhead of the conversion algorithm is very low. When general box information is copied, the old data-space can be freed immediately. When next node pointers are copied, the old space of next node pointers can immediately be freed. When copying edge-lists we may need to store one common edge-list between the two representations, other edge-lists are stored in either of the representations but not both of them.

The high-level pseudo code in *Figure 6.6* summarises the conversion algorithm.

```
For each box that is not empty
  Fill in box values
  x = deepest node
  While x is in Box
    d = x
    While d in Box and d is not large
      Decrement d
    End While
    Set Edges for x through d+1
    x = d
  End While
End For
```

Figure 6.6. The Pseudo Code of the Conversion Algorithm

6.3 Running Algorithms on a Suffix Vector

The efficiency of storing the suffix tree/vector is only one of the issues needing consideration. Retrieving information from the tree is at least as important as the

space requirement of the representation. The time-advantage of the proposed suffix vector over Kurtz's representation [Kur99] is two-fold. Firstly, this eliminates redundant edges and nodes; these do not have to be revisited in an algorithm that runs on the suffix vector. Secondly, the suffix vector structure itself is a simpler structure allowing faster retrieval of edge and node information. In the following section we analyse the number of steps potentially saved by not visiting redundant nodes and edges while in Section 6.3.2 we compare the number of operations needed to retrieve the information on nodes and edges in both representations.

6.3.1 Avoiding Revisiting Nodes in the Suffix Vector

As already reported in Chapter 2, the suffix tree is a versatile data structure used to solve many string-matching problems. Some problems, such as finding the first occurrence of a pattern in a string, do not require suffix link information. Other algorithms, such as the matching statistics algorithm, which we use in our document-comparison application, utilise suffix link information. In this group of algorithms, nodes may be visited multiple times and the suffix vector representation is capable of eliminating the examination of the same information multiple times. We illustrate how this is achieved in the case of the matching statistics algorithm. Other algorithms that utilise suffix links can also benefit from the suffix vector representation. The matching statistics algorithm aims at finding the matching statistics value for each position in string *P* compared to string *S* and is described in detail in Chapter 2.

Let us assume that we have found that the matching statistics value for position *i* in *P* is $ms[i]=l$ and we have finished matching at position *k* in *S*. In the suffix vector representation, this means that $S[k+l]\neq P[i+l]$ and either there is no box at position *k* or there is a box but no continuations match $P[i+l]$. We must return to the last visited node, follow its suffix link and go down to find the occurrence of $P[i+1..i+l-1]$, which we know is in the string. Consider the following two cases.

Case 1. We find $P[i+1..i+l-1]$ at some position $j\neq k$. In this case the part of the vector is different from the part of the previous phase; therefore edges must be examined. If this is the case it will require as many examinations as did the suffix tree representation.

Case 2. We find *P[i+1..i+l-1]* at position *j=k+l-1*. If there is no box at the given position, we recognise that *ms[i+1]=l-1*. Where there is a reduced box, we recognise we need not examine the edges running out of that node as we have examined the edges of node with depth *l* at position *k*. Now we must examine the edges of the node with depth *l-1*. These are the same as this is a reduced box. If there is a regular box at position *k* we may still save comparison steps in the compact representation. In phase *i* we can store the actual address of the first edge examined. In phase *i+1* we only have to examine edges up to this address as edges after this have been examined in phase *i* and no match has been found. If Case A applies (see Chapter 5) it means that this node has exactly the same edges as the node examined in phase *i*, so no further examination is required and *ms[i+1]* can be set to *l-1*. If Case B applies we may find a continuation before reaching the edges of the previous node. In that case we follow matching using that edge. The other case is when we reach the first edge of the previous node, so no further examination is required and *ms[i+1]* can be set to *l-1*. If Case C applies, it means that there are no shared edges between the two nodes making this case similar to Case 1.

Below is an example of the running of the matching statistics algorithm on a suffix tree and a suffix vector. We note the steps that are not necessary in the case of the suffix vector but that are still necessary in case of the suffix tree. The string that we compare with our example string is *T='ababcdabcab$'*. We build the compact suffix vector and the suffix tree for our original string *S='abcdabdbcdabb$'* and calculate the matching statistics of T. The suffix tree and the suffix vector for S are the same as in *Figures 5.1* and *5.3*. The path followed in each step is depicted by the dotted arrows in *Figure 6.7.(a)* through *6.7.(k)*.

Position 0. (*Figure 6.7.(a)*) We must traverse down the tree and the vector starting from the root. We must determine whether there is an edge running out of the root starting with *T[0]='a'*. There is *(0,1)*. Now we need to compare the next character of the string to the next character on the edge. In this particular case they are equal: *T[1]=S[1]*. We have reached the end of the current edge, so we have to update the current node value. *Node 1* will be our current node. We have to check whether there is an edge starting with *T[2]='a'* running out of this node. There is no such edge, so we set the matching statistics value for position *0* to *2*. So far, we have made the same number of comparisons. In the worst case we had to make 5 comparisons at the root, then 1 comparison to check the next character on the edge, and then 4 comparisons to check the edges running out of *node 1*. That is 10 comparisons up to this point in both the tree and the vector.

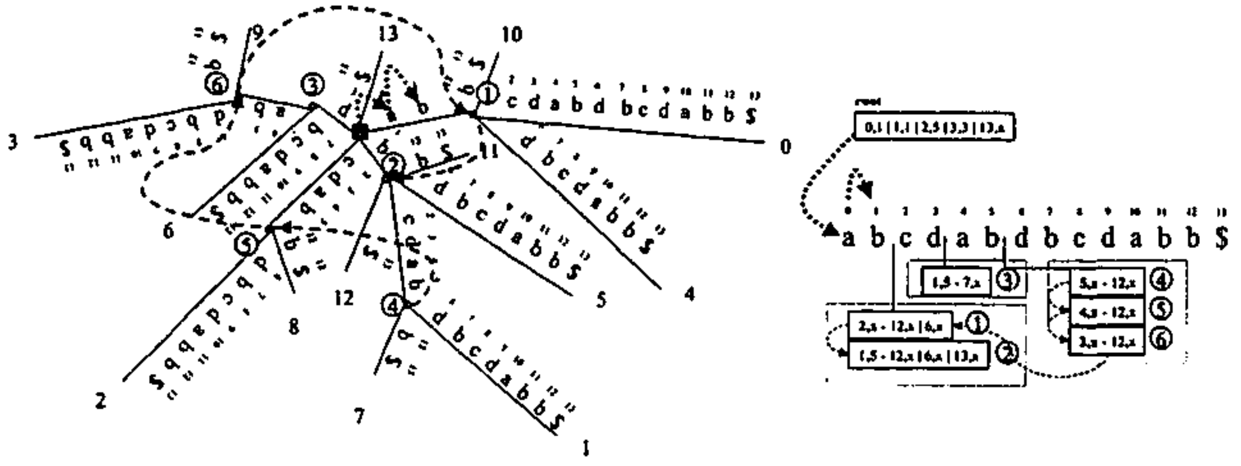


Figure 6.7.(a). Position 0.

Position 1. (*Figure 6.7.(b)*) Instead of trying to find the matching statistics value of position *1* by starting from the root we can follow the suffix link of *node 1*, which points to *node 2*. Note that these nodes are represented at the same position in the vector making the suffix link implicit. In the tree we must check the four edges running out of this node again, while in the vector we only need to check one edge not present in *node 1* as the rest of the edges have already been checked. This can be achieved by storing the pointer to the first edge every time we start analysing a node. When this edge is reached in the next phase we can stop. In this case we cannot find a matching edge, so we set the matching statistics value of this position to 1. The number of comparisons in the case of the tree and the vector up to this point are $C_{tree}=14$ and $C_{vector}=11$, respectively.

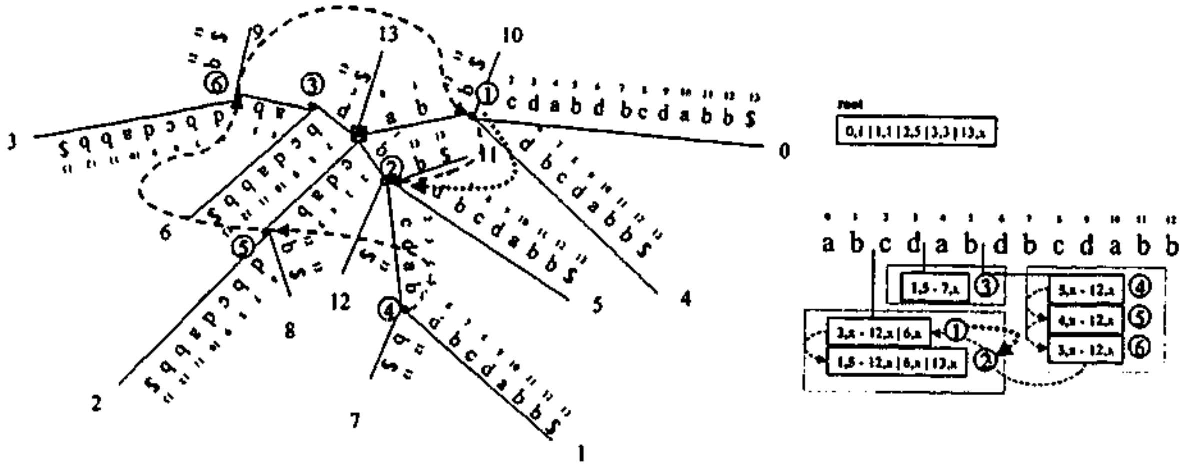


Figure 6.7.(b). Position 1.

Position 2. (*Figure 6.7.(c)*) The last match was 1-character long, thus we have to start traversing from the root again. We must check whether there is an edge running out of the root starting with *T[2]='a'* – there is *(0,1)*. In the worst case we have to do 5 comparisons in both cases. From there we can traverse down to *T[7]='b'*, meaning that *T[2..7]* can be found in *S*. Thus, the matching statistics value must be set to 6 at position *2*. One node encountered on our way (*node 1*) requires 3 comparisons in the worst case. In the case of the other characters we need one comparison per character for both the tree and the vector. By the end of this phase we have done C_{tree}*=26* and C_{vector}*=23* comparisons, respectively.

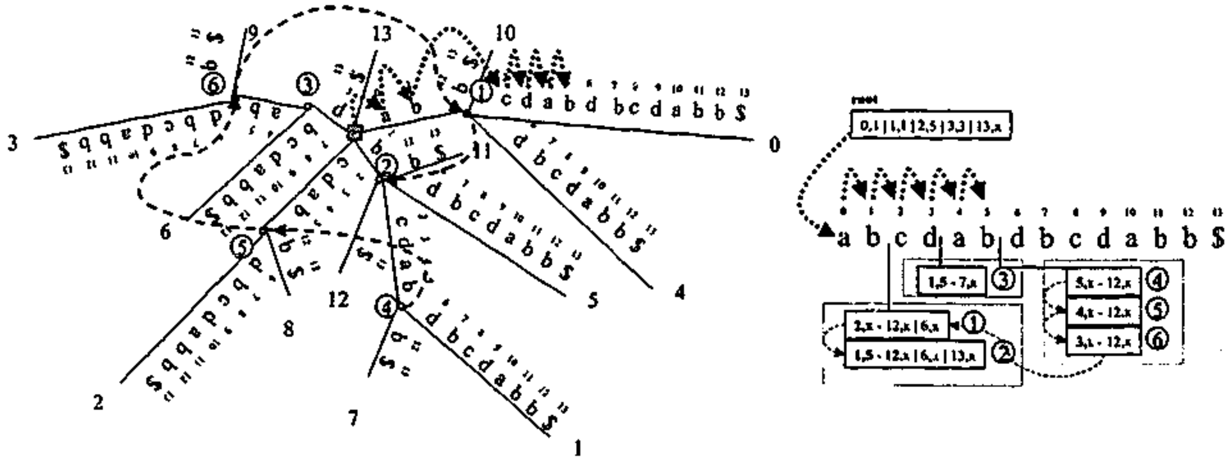


Figure 6.7.(c). Position 2.

Position 3. (*Figure 6.7.(d)*) The last visited node was *node 1*, so we must follow the suffix link of that node and then traverse down 4 characters that definitely match. We arrive at position *5* in the vector and *node 4* in the tree. We must check whether there is an edge starting with *T[8]='c'* running out of the node. This requires the analysis of the two edges in the case of both the tree and the vector. No match is found, thus the matching statistics value is set to 5 at this position.

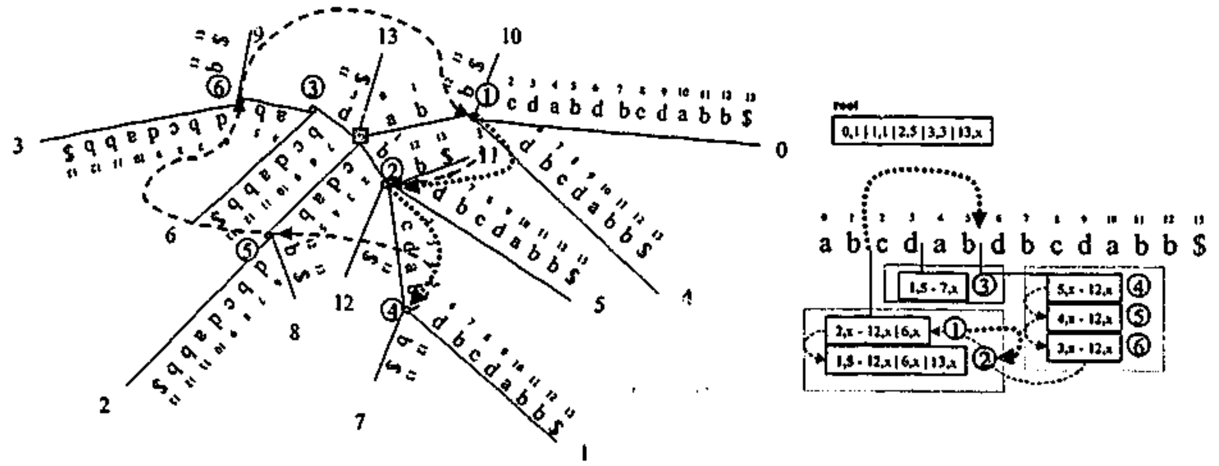

root
0,1 | 1,1 | 2,5 | 3,3 | 13,x
a b c d a b d b c d a b b $
1,5 - 7,x ③
5,x - 12,x ④
4,x - 12,x ⑤
3,x - 12,x ⑥
2,x - 12,x | 6,x ①
1,5 - 12,x | 6,x | 13,x ②

Figure 6.7.(d). Position 3.

Two comparisons are needed in both cases at *node 4* and a maximum of 4 comparisons when following the suffix link, making the total C_{tree}*=32* and C_{vector}*=29*.

Position 4. (*Figure 6.7.(e)*) Our last visited node was *node 4*, so we must follow its suffix link and check the possible continuations from that node. There is no edge running out of *node 5* that starts with *T[8]='c'*. In the tree we must explicitly check the edges running out of the node (2 comparisons). In the vector we can simply skip this step because we have a reduced box, which means we immediately conclude that this node has the same edges running out as the previous one. Therefore we save these two comparisons in the case of the vector. In *Figure 6.7.(e)* this box is shown as a regular box in order to illustrate the steps performed. The total number of comparison steps at the end of phase 4 is C_{tree}*=34* and C_{vector}*=29*.

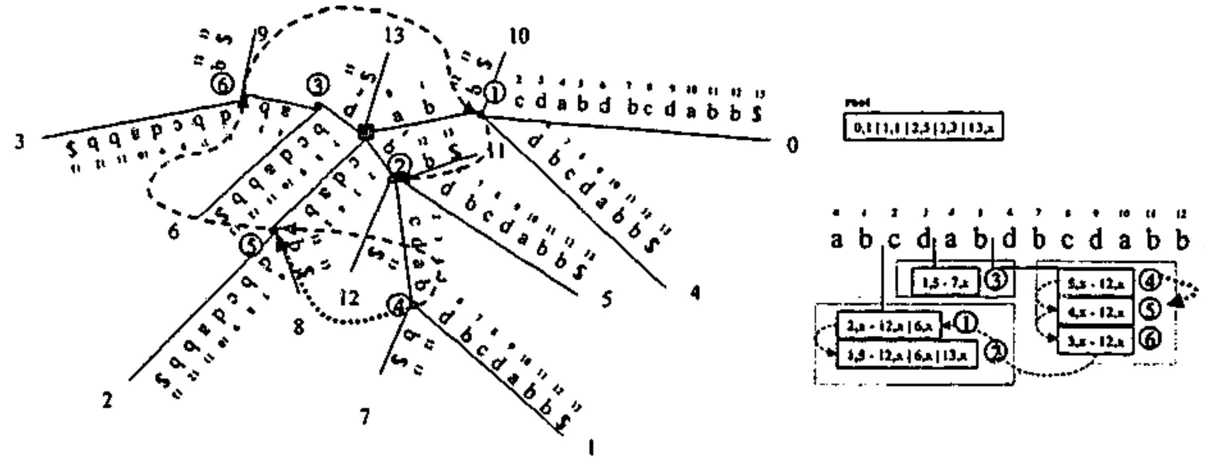

root
0,1 | 1,1 | 2,5 | 3,3 | 13,x
a b c d a b d b c d a b b $
1,5 - 7,x ③
5,x - 12,x ④
4,x - 12,x ⑤
3,x - 12,x ⑥
2,x - 12,x | 6,x ①
1,5 - 12,x | 6,x | 13,x ②

Figure 6.7.(e). Position 4.

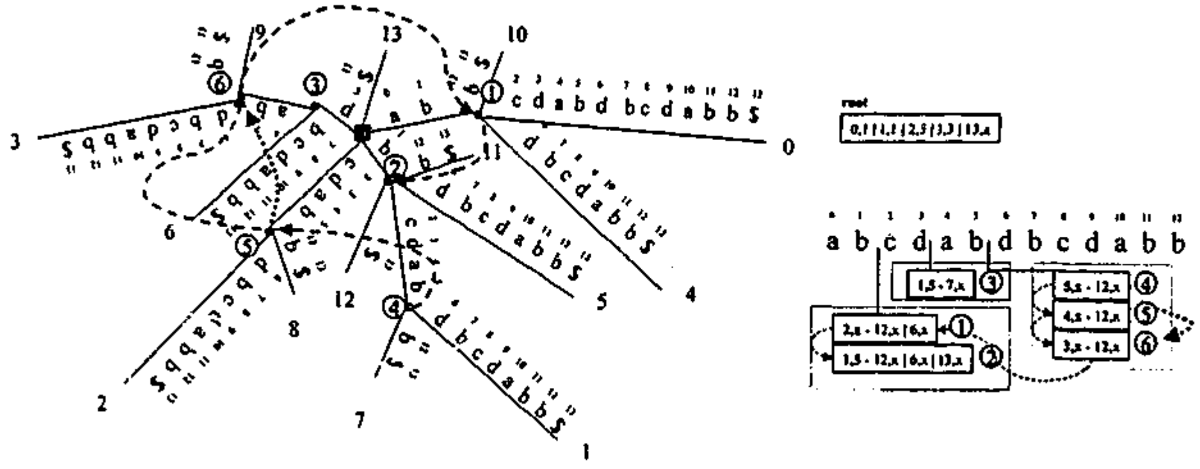

root
0,1 | 1,1 | 2,5 | 3,3 | 13,x
a b c d a b d b c d a b b $
1,5 - 7,x ③
5,x - 12,x ④
4,x - 12,x ⑤
3,x - 12,x ⑥
2,x - 12,x | 6,x ①
1,5 - 12,x | 6,x | 13,x ②

Figure 6.7.(f). Position 5.

Position 5. (*Figure 6.7.(f)*) In this phase we jump from *node 5* to *node 6*, which is still represented by the same reduced box in the vector. This means no further matches can be found and two further steps are saved in the vector. C_{tree}*=36* and C_{vector}*=29*.

Position 6. (*Figure 6.7.(g)*) In this phase we explicitly have to jump from *node 6* to *node 1* in both the tree and the vector. Note that in the vector these two nodes are stored at different positions. We must check the edges running out of this node, requiring a maximum of 3 comparisons in the worst case for both representations.

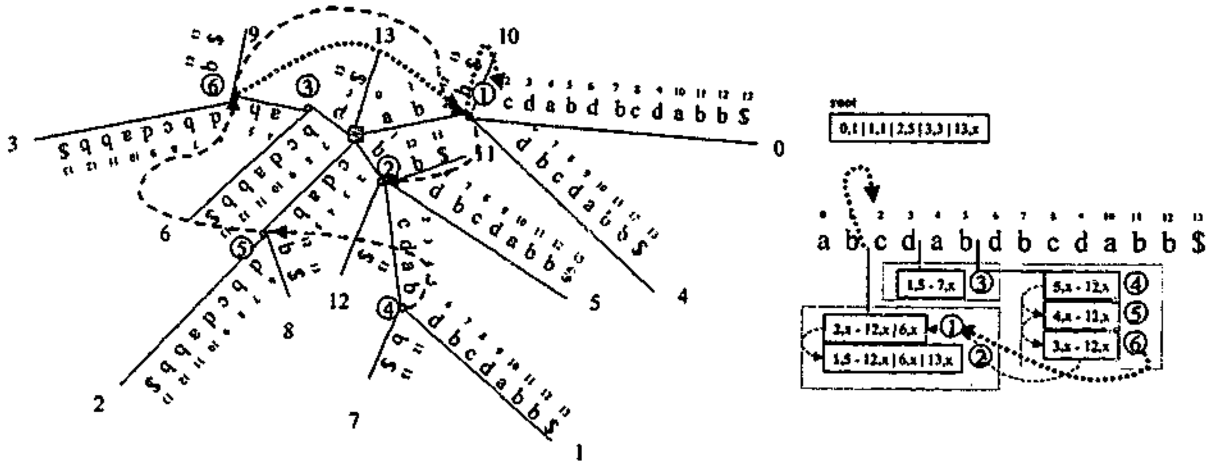


Figure 6.7.(g). Position 6.

In the suffix vector we can only save comparisons if the nodes to be examined in consecutive steps are represented at the same position. This holds true as no data is shared between nodes stored at different positions. There is an edge *(2,x)* running out of this node with *T[8]='c'* but no further matches are possible, so we set the matching statistics value for this position to 3. The total number of comparisons by the end of this phase is C_{tree}*=40* and C_{vector}*=33*.

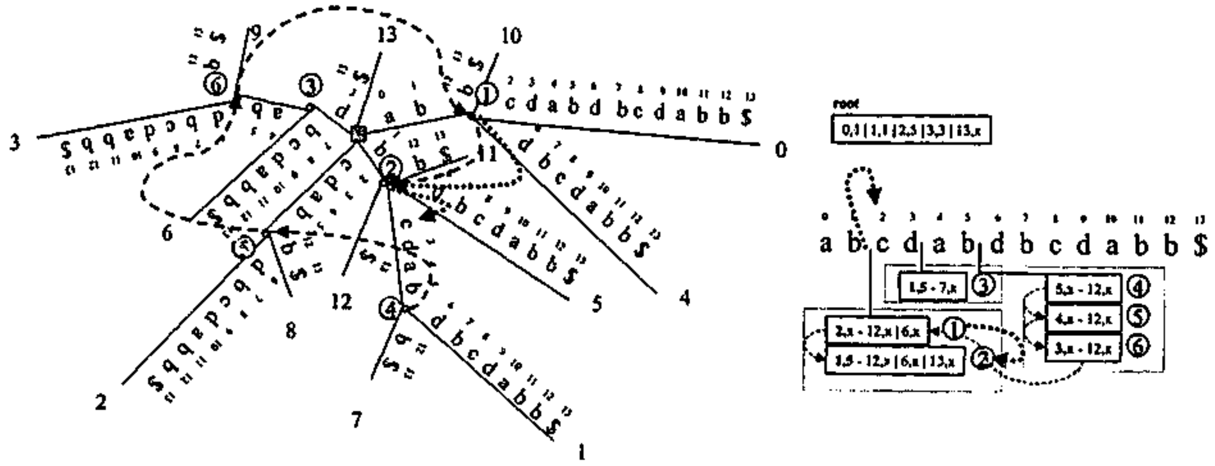


Figure 6.7.(h). Position 7.

Position 7. (*Figure 6.7.(h)*) We follow the suffix link from *node 1* to *node 2* and find the outgoing edge that starts with *T[8]='c'* by using at most 4 comparisons

in both cases and one more comparison is needed to test that no more matches are possible. The matching statistics value for this position is 2. C_{tree}*=45* and C_{vector}*=38*.

Position 8. (*Figure 6.7.(i)*) We have to begin from the root and find the edge that starts with *T[8]='c'*. In both cases this requires a maximum of 5 comparisons, with one more comparison required to ensure that no more matches are possible. The matching statistics value for this position is 1. C_{tree}*=51* and C_{vector}*=44*.

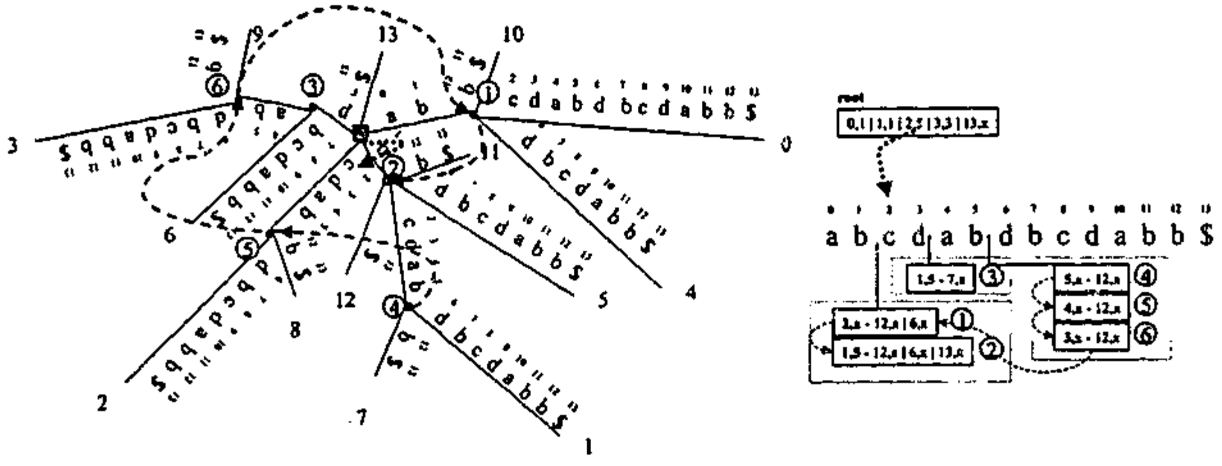


Figure 6.7.(i). Position 8.

Position 9. (*Figure 6.7.(j)*) As only one character could be matched in the previous phase we must start matching from the root. We find the edge starting with *T[9]='a'* in at most 5 comparisons in both representations. One comparison is required to match the next character. Three comparisons are necessary to find out that no more matches are possible from *node 1*. The matching statistics value is set to 2 for this position. The total number of comparisons done in this step is 9 in both cases, making the total values C_{tree}*=60* and C_{vector}*=53*.

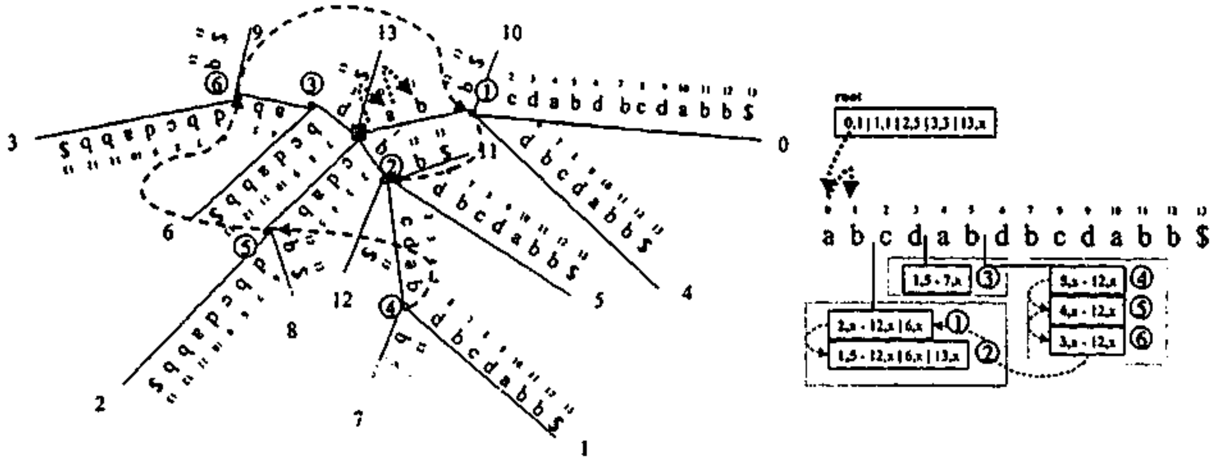


Figure 6.7.(j). Position 9.

Position 10. (*Figure 6.7.(k)*) This phase is similar to position *1*, with the only difference being that we definitely find *T[10]='b'* at the first comparison in the vector. In the case of the tree this may take a maximum of 4 comparisons. The matching statistics value is 2 for this position. The total number of comparisons up to this point is C_{tree}*=65* and C_{vector}*=55*.

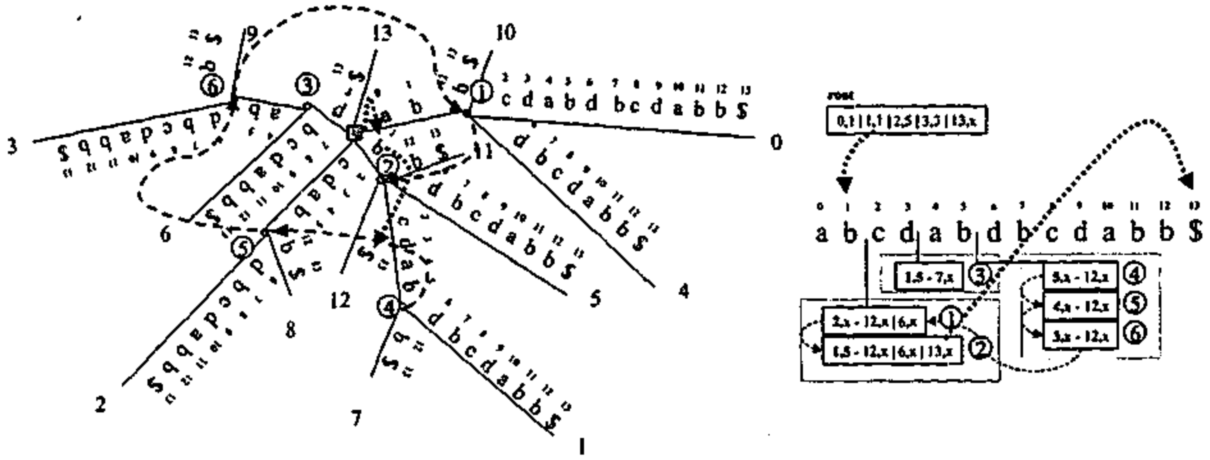


Figure 6.7.(k). Position 10.

Upon reaching position *10* we can stop the algorithm as we have found a match for the remaining part of the string. Therefore we know that the remaining matching statistics values represent matches up to the remaining part of the document.

In this example, we have saved one sixth of the steps. However we have not considered some other factors that may also save steps in the case of the vector. In the case we are on the natural edge, right after the node (one character down on the edge), and the node pointed by the suffix link is at the same position, we immediately conclude that we will follow the natural edge. This would place us at exactly the same position, so examining the outgoing edges is not necessary. However, it is still needed in case of the suffix tree.

6.3.2 Retrieving Information from the Suffix Vector

Not only does the suffix vector save time because of the redundant information that is not repeatedly examined but it is also a simpler physical structure than Kurtz's representation [Kur99]. Therefore, retrieving infoımation becomes simpler [MZV01]. In this section we analyse the number of basic operations needed to get certain information from the suffix tree and compare our proposed representation with Kurtz's one [Kur99]. A theoretical comparison here is more appropriate than an experimental one since the actual implementation may favour one representation over the other. The notation in this section is specific to the following comparison of the two representations.

There are three basic operations on a suffix tree:

1. getting the first edge running out of a node
2. getting the next edge from the current edge in the list of edges
3. following a suffix link

We divide primitive operations into three categories:

1. **Masking.** It is when we have a value (an integer or one byte) that stores multiple pieces of information and we have to mask some bits to retrieve the information we need. We denote the masking operation by **M**.
2. **Comparison.** It is when we have to compare two values (two integers or two bytes) and based on the result, we choose different execution paths. We denote the comparison operation by **C**.
3. **Addition (Subtraction).** It is when we have to add (or subtract) two values. We denote the addition (subtraction) operation by **A**.

In the following sections we analyse how many primitive operations are needed to execute the three basic operations on both representations. We analyse both best-case and worst-case scenarios.

6.3.2.1 Getting the first edge running out of a node

Let us assume that we have a pointer that points to the current node and we want to get the information of the first edge running out of this node. The information we need is the start position of the text represented on the edge, the end position of the text represented on the edge, and the position of the next node if we follow this edge. Firstly, we analyse worst-case scenarios.

The following list describes the steps needed in Kurtz's representation. To get the edge labels in Kurtz's representation, we need the depth value of nodes both at the end and the beginning of the edge and the headposition of the node at the end of the edge. We also need the first child information of the node at the beginning of the edge. The worst case is when both nodes are small nodes. Let us denote the node at the beginning of the edge by *B* (in subscripts lowercase letters are used). The large node belonging to this small node is *S*. The node at the end of the edge is denoted by *E* and the corresponding large node is denoted by *F*. Kurtz's representation is depicted in *Figure 3.3*.

1. **M.** We have to mask the 5 most significant bits of the first integer of *B* to learn the distance value. The distance value may take values between 0 and 31. Values 1 through 31 are valid distance values while 0 represents a large node and we do not need to store a distance value for a large node. Let us denote the distance value by D_b.
2. **C.** We have to decide whether the value retrieved in the previous step is 0 or not because the bit structure of the node is different for large and small nodes.
3. **A.** Since this is a small node we have to calculate the position of the large node *S* from the distance value.
4. **M.** We need the depth value of *node B*. Since this node is a small node we have to find out the depth value of the large node *S* to learn the depth value of this node. The position of the large node has been calculated in the previous step. We have to mask the most significant bit of the third integer in *node S* to find out whether the depth of this node can be represented in 10 bits or not.
5. **C.** We have to find out whether the bit retrieved in the previous step is 1 or 0.
6. **M.** Based on the result of the previous comparison we have to mask either 10 or 27 depth bits of the third integer of *S*. Let the depth of this large node be P_s.
7. **A.** The depth of the small node we are at is $P_b=P_s-D_b$.
8. **M.** We have to know whether the first child is a leaf or not because leaves and nodes are stored in two separate arrays. We have to mask the first bit of the first child bits of the current node. Let *l* denote the bit retrieved in this step.
9. **M.** We have to mask the 26 least significant bits from the first integer in *node B*.
10. **M.** We have to mask the 2 most significant bits of the second integer in *node B*.
11. **A.** We have to create a new integer from the bits retrieved in the previous two steps in order to create the first child pointer.
12. **C.** We have to know whether the bit retrieved in Step 8 (*l*) is 0 or 1. 1 denotes a leaf.
13. **M.** Based on the comparison of the previous step we have to find the headposition of the next node *E*. Let us denote this value by H_e. Since the first child is a node, we have to mask the 5 most significant bits of the first integer to find out whether this is a large or a small node. We assume that this is small node. The data is stored in the distance value D_e.

14. **C.** We have to decide whether the bit retrieved in the previous step is equal to zero or not.
15. **A.** Since this is a small node we have to calculate the position of the corresponding large node from the distance value D_e.
16. **M.** In order to learn the headposition of the large node *F* we have to mask the 27 least significant bits of the fourth integer. Let us denote this headposition by H_f.
17. **A.** The headposition of the small node is $H_e=H_f+D_e$.
18. **M.** We also have to find the depth of the end node *E*. Since this is a small node, first we have to identify the depth of the corresponding large node *F*. To decide whether the depth of the large node *F* is stored in 10 or 27 bits we have to mask the most significant bit of the third integer.
19. **C.** We have to decide whether the result of the previous step is 1 or 0.
20. **M.** Based on the result of the previous step we have to mask either 10 or 27 depth bits. Let us denote the depth of this node by P_f.
21. **A.** To obtain the depth of the small node: $P_e=P_f-D_e$.
22. **A.** The beginning position of the edge is the headposition of the end node (H_e) plus the depth of the originating node (P_b): $edge_{beg}=H_e+P_b$.
23. **A.** The length of the edge is a good substitute for the end position and it can be directly retrieved: $L=P_e-P_b$.

The total number of steps needed in the worst case is 23 (10 maskings, 5 comparisons, and 8 additions).

Now let us consider the number of steps needed in case of the suffix vector representation. We denote the originating node by *B* and the destination node by *E*. We leave natural edges out of this discussion and by getting the first edge we mean getting the first edge that is not a natural edge. The worst case is when this is a regular box (as opposed to a reduced box) and next node pointers are stored in 4 bytes. Our proposed representation is depicted in *Figure 5.8*.

1. **M.** We have to mask the first bit of a box to find out whether the deepest node value and the number of nodes are 1-byte or 4-byte values.
2. **C.** We have to decide whether the bit retrieved in the previous step is 1 or 0.

3. **M.** We have to mask the first byte or the first 4 bytes of the array to get the deepest node value D_b.
4. **M.** We have to mask 1 byte or 4 bytes to retrieve the number of nodes value N_b.
5. **M.** We have to mask the first bit of the next byte to find out whether this is a reduced box or not.
6. **C.** We have to decide whether the bit is 0 or 1.
7. **M.** We have to mask the second bit of the same byte to find out whether the next node pointers are stored in one byte or four bytes.
8. **C.** We have to decide whether the bit selected in the previous step was 1 or 0.
9. **A.** We have to find the position of the pointer that points to the first edge. We are currently at the position where the suffix link starts. According to our assumption, the next node pointers are stored in four bytes. Thus we have to multiply the number of nodes value by 4. Let us store this value in *h*.
10. **A.** We have to add 4 to *h*. (4 bytes are used to store the suffix link). With this offset we can locate the first of the first edge pointers, which points to the deepest node.
11. **A.** We have to get the position of the first edge pointer belonging to the current node in the list, which is an offset of *deepest_node-current_node_depth* from the first of the first edge pointers.
12. **M.** We have to mask the first bit of the edge to find out whether this is a leaf or not.
13. **C.** We have to find out whether the bit selected in the previous step is 1 or 0.
14. **M.** We have to mask the third bit of the edge to find out whether the next node pointer is stored in 1 byte or 4 bytes.
15. **C.** We have to find out whether the bit selected in the previous step is 1 or 0.
16. **M.** We mask the 29 least significant bits of the first edge to find the beginning position of the edge. The length of the edge can directly be retrieved from the next node pointer based on the result of Step 15. The length of the edge is a good substitute for the end position. Also note that the length of the edge is also a good substitute for the next node position because the nodes are stored along with the string.

In the worst case we need 16 steps (8 maskings, 5 comparisons, and 3 additions).

Let us now analyse best-case scenarios. The best case in Kurtz's representation is when we are on a large node and the first edge is a leaf. In this case the following steps are required.

1. **M.** We have to mask the 5 most significant bits of the first integer to learn the distance value. The distance value may take values between 0 and 31. Values 1 through 31 are valid distance values while 0 represents a large node and we do not need to store a distance value for a large node. We assume that we are on a large node, thus this value is 0.
2. **C.** We have to decide whether this value is 0 or not because the bit structure of the node is different for large and small nodes. In this step we learn that we are on a large node.
3. **M.** We need the depth value of the node. We have to mask the most significant bit of the third integer to find out whether the depth of a node can be represented in 10 bits or not.
4. **C.** We have to find out whether the bit retrieved in the previous step is 1 or 0.
5. **M.** Based on the result of the previous comparison we have to mask either 10 or 27 depth bits. Let the depth of this large node be P_b.
6. **M.** We have to know whether the first child is a leaf or not because leaves and nodes are stored in two separate arrays. We have to mask the first bit of the first child bits.
7. **C.** We have to know whether the bit retrieved in the previous step is 0 or 1. We suppose that we are on a leaf.
8. **M.** We have to mask the 26 least significant bits from the first integer.
9. **M.** We have to mask the 2 most significant bits of the second integer.
10. **A.** We have to create a new integer from the bits retrieved in the previous two steps. This will give us the position of the leaf, which is the same as the index. Let us denote this value by *l*.
11. **A.** The beginning position of the edge is the index of the leaf plus the depth of the original node (P_b). The beginning position of the edge is $edge_{beg}=l+P_b$.

In the best case of Kurtz's representation 11 steps are required (6 maskings, 3 comparisons, 2 additions).

In the suffix vector representation, in the best-case scenario we have small next nodes, thus the number of nodes need not be multiplied by 4 (Step 9). If the first edge is a leaf we do not need to retrieve information on the length of the edge (Step 14 and

Step 15). It will save us 3 steps. Thus in the best case we need 13 steps (7 maskings, 4 comparisons, 2 additions).

6.3.2.2 Getting the Next Edge from the Current Edge in the List of Edges

Now let us assume that we are examining the edges running out of a node to find out which one we have to follow. In case of Kurtz's representation, it means that we follow the branchbrothers of a node until we either find an outgoing edge that matches a certain character or find the last edge in the list. Firstly, let us analyse the number of steps needed in Kurtz's representation to get the information on the next edge in the list. Again, we need the start position and the end position of the edge. The worst case is when the next edge runs into a small node and we are currently on a node whose branchbrother must be retrieved. The steps required in this case are:

1. **M.** Since this edge runs into a node we have to decide whether the 29 least significant bits of the second integer store a branchbrother reference or a suffix link. They only store a suffix link if this is the last edge in the list. Of course, we always have to check whether this is the last edge. This information is encoded in the third most significant bit of the second integer, so we mask this bit.
2. **C.** We have to decide whether the bit obtained in the previous step is 1 or 0. If this was the last edge in the list we would be done but we assume that there are more edges.
3. **M.** We have to mask the 29 branchbrother bits in the node.
4. **M.** We have to mask the first bit of the branchbrother reference to learn whether this edge is pointing to a node or it is a leaf.
5. **C.** We have to find out whether the value obtained in the previous step is 0 or 1. We assume that this is a small node. Thus from here on we have to complete the same steps as we did for the destination node when we examined the steps to retrieve the first edge. This destination node is the descendant of the same node as the ancestor of the previously examined node. Thus we assume that the information (the depth value: P_b) on that node has already been retrieved. It means that only information on the destination node is needed.
6. **M.** We have to find the headposition of the destination node E. Let us denote this value by H_e. Since this is a node we have to mask the 5 most significant bits of the first integer to find out whether this is a large or a small node. We assume that this is a small node. The data is stored in the distance value D_e.

7. **C.** We have to decide whether the bits retrieved in the previous step are equal to 0 or not.
8. **A.** Since this is a small node we have to calculate the position of the corresponding large node from the distance value D_e.
9. **M.** In order to learn the headposition of the large node F we have to mask the 27 least significant bits of the fourth integer. Let us denote this headposition by H_f.
10. **A.** The headposition of the small node is $H_e=H_f+D_e$.
11. **M.** We also have to find the depth of the end node E. Since this is a small node, first we have to identify the depth of the corresponding large node F. To decide whether the depth of the large node F is stored in 10 or 27 bits we have to mask the most significant bit of the third integer.
12. **C.** We have to decide whether the result of the previous step is 1 or 0.
13. **M.** Based on the result of the previous step we have to mask either 10 or 27 depth bits. Let us denote the depth of this node by P_f.
14. **A.** To obtain the depth of the small node: $P_e=P_f-D_e$.
15. **A.** The beginning position of the edge is the headposition of the end node (H_e) plus the depth of the originating node (P_b). The beginning position $edge_{beg}=H_e+P_b$.
16. **A.** The length of the edge is a good substitute for the end position and it can be directly retrieved: $L=P_e-P_b$.

The worst case is 16 steps (7 maskings, 4 comparisons, 5 additions).

Now let us check the steps needed in case of the suffix vector representation. Here we assume that we have examined an edge and the next edge in the list needs to be analysed. We assume that in the previous step the number of bytes required to store next node information has been identified. The worst case here is when the next edge is not a leaf. The following steps are required:

1. **M.** We have to mask the second most significant bit of the leaf to decide whether we have reached the last edge or not.
2. **C.** We have to decide whether the bit retrieved in the previous step is 0 or 1. We assume that there is a next edge to be analysed and its position is known from the previously analysed edge.

3. **M.** We have to mask the first bit of this next edge to find out whether this is a leaf or not.
4. **C.** We have to find out whether the bit selected in the previous step is 1 or 0.
5. **M.** We have to mask the third bit of the edge to find out whether the next node pointer is stored in 1 byte or 4 bytes.
6. **C.** We have to find out whether the bit selected in the previous step is 1 or 0.
7. **M.** The length of the edge can directly be retrieved from the next node pointer based on the result of Step 5. The length of the edge is a good substitute for the end position. Also note that the length of the edge is also the next node position because the nodes are stored along with the string.

It gives us 7 steps in the worst case (4 maskings and 3 comparisons). The best case in both representations is when we have reached the last edge. It can be found out by masking the flag bit and decide whether it is 0 or 1 (1 masking and 1 comparison). It is true for both representations.

6.3.2.3 Following a Suffix Link

Suffix links are not integral parts of a suffix tree but many algorithms including the matching statistics algorithm make use of them. In this section we compare the number of steps required to get the position of the node pointed to by the suffix link in both representations.

In Kurtz's representation [Kur99] the worst case is when the current node is a large node and its depth is greater than 1023. Most of the files do not even have a single node like this because it means that it contains an overlap of at least 1024 characters. If we have such a node the suffix link information is stored with the last edge running out of that node, which means that we have to scan through all edges running out of that node to find the suffix link. The number of steps required here depends on the size of the alphabet. The following steps must be repeated as many times as the number of outgoing edges:

1. **M.** Masking the third most significant bit of the second integer to find out whether the 29 least significant bits store a branchbrother or a suffix link.
2. **C.** We have to decide whether the result of the previous comparison is 1 or 0.
3. **M.** Masking the 29 least significant bits of the second integer. If the previous comparison showed that this is a suffix link then we stop here. If it is a branchbrother reference we have to repeat these steps until the suffix link is found.

The number of steps in the worst case is *3*ALPHABET_SIZE*. If we do not consider this very rare situation, the worst-case scenario is when we are on a large node and we explicitly have to read the suffix link information. 26 bits of that information is stored in the third and fourth integer of the large node and 2 bits are stored in the leaf list at the position identified by the headposition of this node. We assume that the headposition has already been retrieved when we have analysed the edges running out of a node. The following steps are required:

1. **M.** Mask the 21 suffix link bits of the third integer in the large node.
2. **M.** Mask the 5 suffix link bits of the fourth integer in the large node.
3. **A.** Add the values obtained in the previous 2 steps.
4. **M.** Mask the 2 bits in the leaf list at the position identified by the headposition of this node.
5. **A.** Add the value obtained in step 4 to the value obtained in Step 3. It will give you the position of the node pointed by the suffix link.

In the worst case we need 5 steps (3 maskings and 2 additions) in Kurtz's representation.

In the suffix vector representation the worst case is when the current node is the one with the smallest node depth represented at the given box. In this case we have to read the suffix link value from the box. We assume that the position where the suffix link is stored in the box has already been determined during examining the edges. The following steps are required:

1. **C.** We have to decide whether this is the node with the smallest depth value.
2. **M.** The worst-case approach assumes that this is the node with the smallest depth value. Then we have to mask the 30 least significant bits of the suffix link integer. It gives us the suffix link value.

The suffix vector representation requires 2 steps in the worst case (1 comparison and 1 masking). The best case in Kurtz's representation is when we are on a small node. In this case if the distance of this node is 1 from the large node then the suffix link points to the large node. The position of this large node is 2 bytes more than our current position, thus a single comparison is enough. The same is true for the suffix vector representation. If the node is not the one with the smallest depth, the node pointed to by the suffix link is stored at the same position. This can also be decided by a single comparison step. Here we note that if the node pointed to by the suffix link is in the same box at the same position (the node is not the one with the smallest depth), then most of the steps that are concerned with retrieving information common to all nodes in the box (deepest node, number of nodes, etc.) are not necessary in the next phase of the algorithm.

The following table summarises our results. We can conclude that in all cases except for one the suffix vector representation is either faster or equal to Kurtz's representation, considering both best and worst case scenarios.

	Kurtz's representation		Suffix Vector	
	Worst case	Best case	Worst case	Best case
first edge	23 (10M 5C 8A)	11 (6M 3C 2A)	16 (8M 5C 3A)	13 (7M 4C 2A)
next edge	16 (7M 4C 5A)	2 (2C)	7 (4M 3C)	2 (2C)
suffix link	3 (2M 1C)*alphabet size	1 (1C)	2 (1C 1M)	1 (1C)
	5 (3M 2A)			

Table 6.2. Retrieving Information from the Suffix Vector

6.4 Summary

In this chapter, a linear time construction algorithm has been proposed for the general suffix vector representation. This algorithm builds and expands on Ukkonen's suffix tree construction algorithm [Ukk95]. We have also analysed the space requirement of the general suffix vector representation.

We have discussed a conversion algorithm that converts a general suffix vector into the more space-efficient compact representation. We have shown that this conversion can be done in linear time and it is also true that this algorithm with minor modifications could convert a general suffix tree representation into the compact suffix vector representation.

Not only is the proposed suffix vector representation better in terms of space requirement than any other representation, but it also eliminates some redundant information, which is present in other representations. The fact that redundant information need not be analysed multiple times, along with the simplicity of the structure, which allows faster retrieval of information from the structure, makes algorithms using the suffix vector run faster in practice [MZS02]. In this chapter we have analysed how fast the algorithms can run on the proposed structure.

First, we showed that algorithms using the suffix link information can benefit from the fact that redundant information is not replicated in the vector. An example has been given to demonstrate how these steps can be saved in case of the matching statistics algorithm.

Then we have also shown that our structure has a simpler physical representation than Kurtz's representation by analysing the number of operations needed to extract certain information from the structure.

CHAPTER SEVEN 7

MatchDetectReveal Prototype and Parallel Applications

7.1 Introduction

In this chapter we describe our practical experiences with the data structures discussed in previous chapters. We have built a prototype system that validates the proposed algorithms [MZS00b]. We have named our prototype system MatchDetectReveal (MDR). MDR is a copy-detection system that uses suffix trees, suffix vectors, and the matching statistics algorithm in the core component. The architecture of the system is described in Section 7.2. The functions of different components are discussed in detail in other sections.

We have run tests on the MDR prototype in different application areas. We have used RFC (Request for Comments) [RFC01] documents as a test base because there are many document streams in that set where documents are different revisions of the same original document. The MDR system has also been successfully applied to compare Spanish language literary works [ZBM01]. This project has been a joint project with the Cervantes Digital Library at the University of Alicante [BP01]. For testing the search engine component, both RFC documents and different Hungarian translations of the same text have been used. This latter document set has been used in a joint project with the Budapest University of Technology and Economics [MFZ$^+$02]. All these applications are discussed in Section 7.3.

Section 7.4 discusses the converter component, which is responsible for the conversion of documents before they are processed. Not only does the converter component convert documents from different file formats (e.g. HTML, MS Word, PDF, PS, etc.) into pure ASCII but it also creates a canonical form of documents, which is insensitive to slight alterations, e.g. capitalization, multiple whitespaces, and different punctuations.

The search engine component is based on the algorithms discussed in Chapter 2. We have run a number of tests that compared different chunking strategies, as well as defined an optimal hash size that is manageable in terms of space. At the same time it does not produce too many false positive cases. The results of these tests are presented in Section 7.5.

When we started testing our system we needed a document set where the overlap between documents was known in advance. We have built a document generator component that is able to generate documents with a predefined amount of overlap from a base document set. This component is also capable of substituting certain words with their synonyms using the thesaurus feature of Microsoft Word. Details of the functionality of this component are discussed in Section 7.6.

One of the challenges we have faced in our prototype system was how to present the results to the end-user in a user-friendly interface with as much information as possible. We have developed a visualiser component with different possible visualizations of the results [MZB01]. The different possible presentations are discussed in Section 7.7.

The testbed also includes a high-performance cluster of workstations, which allowed us to test our document comparison algorithm in a parallel environment. Tests have been run using different tools and messaging libraries [MZS99, MZS00a]. The results of parallel comparison tests are discussed in Section 7.8.

The purpose of this chapter is to demonstrate that the ideas presented in previous chapters are used in real-life applications. The wide range of applications discussed here show that suffix trees are important data structures.

7.2 MatchDetectReveal System Architecture

In this section we outline the architecture of the MatchDetectReveal (MDR) system. The general architecture is depicted in *Figure 7.1*. The components that have already been developed are discussed in detail in later sections while the functionality of each component is briefly discussed here.

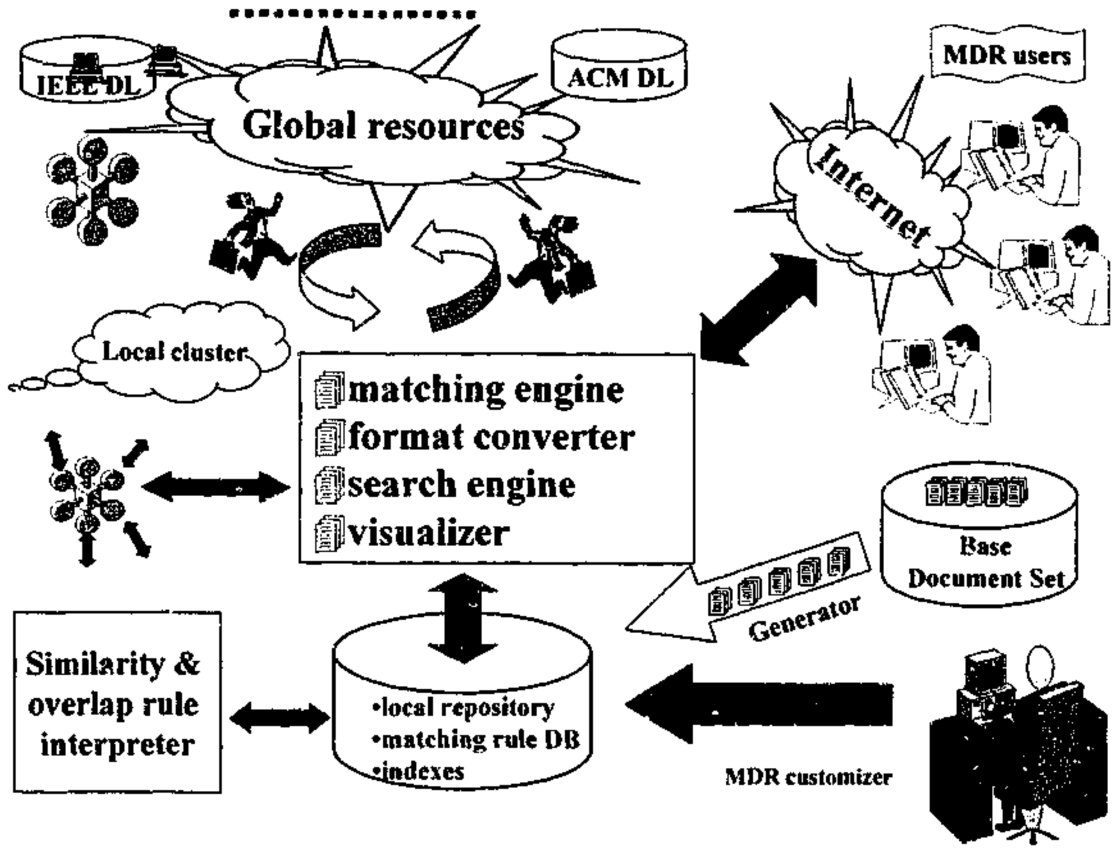


Figure 7.1. MDR Architecture

Users of the MDR system must be able to submit documents that either need to be stored in the document database or need to be compared to registered documents. We have developed an Internet-based interface for document submission, which accepts either stand-alone files or a batch of files compressed into a single zip file. Submitted documents can be compared to the documents that are registered in the local repository. Individual documents may be manually registered into the repository, while a Web-robot may be sent around the Internet to automatically register documents on the Internet into the database. Documents from different digital libraries, e.g. ACM DL or IEEE DL may also be registered into the database.

When a document is registered into a database it is converted and chunked into smaller parts using the chunking strategies discussed in Chapter 2. The hash values calculated on those chunks are registered in the database along with a unique file identifier. The conversion and chunking processes are further discussed in Sections 7.4 and 7.5.

If a document is submitted for comparison with registered documents it goes through the same pre-processing as documents before registration. The search engine component then compares the chunks of the submitted document to the chunks registered in the database. If the document contains overlapping chunks above a certain threshold, it is sent for further analysis to the matching engine. These thresholds are different depending on the application. The files that overlap with the submitted document are downloaded from the Web or retrieved from the local repository and the matching engine identifies overlapping chunks of text.

The matching engine, as we have already mentioned, is responsible for comparing the submitted document to candidate documents identified by the search engine. The matching engine uses the algorithms discussed in previous chapters. As shown in those chapters, our compact suffix vector representation is the most suitable for document overlap detection.

The 'Similarity and Overlap Rule Interpreter' component is outside the scope of this thesis and could be addressed in the future. This component will define how to handle rephrasing, changing the names of localities, substituting synonyms for certain words, etc.

With the constant increase of the processing power of commodity workstations, clusters of workstations are widely used for parallel processing [BB99]. Parallel processing can be performed in many different phases of the comparison but since this thesis focuses on the exact comparison phase in the matching engine, parallelising this phase is analysed in detail in Section 7.8.

The visualiser component of the system is responsible for presenting the end user with the parts of the document which are plagiarised or overlap. The output of the matching engine is a list of overlapping chunks defined as positions in the text and the length of the chunks. This information is then converted into a visual representation. The visualiser component is also responsible for restoring the original positions because the file "shrinks" in the conversion process by the elimination of multiple whitespaces. When we present the results, we retain the formatting present in the pure ASCII version of the documents. The visualiser component is further discussed in Section 7.7.

Document Generator is a supplementary component of the system, which generates sufficient number of documents with known overlap to test our algorithms. One can define the size and overall plagiarism content of the document to be generated as well as the number of files to be used for plagiarism and the size of the chunks. It is also possible to define the number of words to be substituted by their synonyms to test more sophisticated ways of plagiarism. A detailed description of the document generator can be found in Section 7.6.

7.3 MDR Applications

In this section we discuss different possible applications of our MDR prototype system. Of course, some applications have already been mentioned in previous chapters when we discussed the algorithms used by the system. Here we revisit those applications and present some other applications.

7.3.1 Plagiarism Detection

We have already discussed this straightforward application in previous chapters. We have to register potential documents in the database. As pointed out in the previous section, one option is to register documents from the Internet. These documents can be registered in the database by a Web-robot that goes around the Internet and sends back the files to the MDR system.

Plagiarism can be defined at a local level, too. A department may maintain a database of previous assignments and every newly submitted assignment can be compared to the database. Also a many-to-many comparison can be performed among assignments submitted on the same topic to discover co-operation between students.

7.3.2 Copy-Detection in a File System

This application is fundamentally the same as the one discussed in Section 7.3.1 but we might have different requirements with regards to the output. If we are interested in the similarity among the files, we can use the same method as in plagiarism detection. Another application may be to identify differences between documents. These are very important applications, for example in RFC documents [RFC01], because some files are revised versions of previous ones. In this case we can use the original method to identify overlapping chunks. Non-overlapping chunks are the new information in the text. With a few simple changes in the visualiser component it can directly output the differences rather than the similarity among files. We have run our tests on the RFC document set on the local cluster at the School of Computer Science and Software Engineering, Monash University. The results of these test runs are presented in Section 7.8 where parallel applications are discussed.

7.3.3 Cross-Referencing Multiple Editions of Literary Works

In this subsection we discuss the applications that have come out of our joint research project with the Cervantes Digital Library at the University of Alicante [BP01]. These applications are similar to the applications discussed above. However, they are put into a different context. The document set used in these tests was taken from the collection held in the Cervantes Digital Library. The library holds different collections of poems as well as different editions of the famous Spanish masterpiece El Ingenioso Hidalgo Don Quijote de la Mancha by Miguel de Cervantes. Different applications, which have been identified as useful by librarians, are discussed below and examples are given in Section 7.3.3.5.

7.3.3.1 Detecting Cross-References

The first application we thought of was to automatically detect quotations or cross-references between different texts. In this case the quote could be hyper-linked to the original. We can imagine many research situations where automatic detection of similar sections of text can be useful. It could be a valuable aid in history research, for instance.

7.3.3.2 Organizing Collections of Small Pieces of Text

Another use of MDR is to detect repeated poems in different poem collections (may also be tales, letters, etc.), where some of these textual units are repeated in different editions. Most often, when collections of poems are developed, it is difficult to keep track of which poems have been included and where, and which not. MDR proved to be a helpful tool for detecting and locating the matching pieces of text for verification purposes.

7.3.3.3 Comparative Analysis of Texts

The objective of this application is to compare different editions of the same literary work. Philologists are usually interested in such comparisons for research purposes. In ancient literature, there are usually different editions/translations of the same work, with modifications performed sometimes by the author, sometimes by the editor. This comparative analysis is in itself an interesting but tedious field of study where any kind of automation is welcome. To cite an example, there are various ancient editions of the *Quijote de la Mancha*, all originals from Cervantes, but still slightly different. This application differs from the previous ones, in the sense that here we have to find differences rather than matches. Although in this case

the matching strings are usually in the same order in both sources, synchronisation is not always possible since the sources may have long additions inserted in different places, apart from small differences in the matching zones. Conventional sequential comparison may also fail in this case.

7.3.3.4 Detecting variations and mistypings

There are variations in spelling owed to ancient editor changes, which are not strictly errors since the Spanish language was not normalized until 1713, when the Real Academia de la Lengua Española was created and language spelling started to settle [ZBM01]. Those variations, though legal, produce differences between editions.

On the other hand, no matter how much care correctors put on freeing texts from digitisation errors, there are always some errors that remain. So an unforeseen result of the application of MDR to comparative analysis was the detection of spelling errors or variations in digital library (DL) texts by the way of comparison of different editions of the same work.

7.3.3.5 Results of Comparison in the Miguel de Cervantes Digital Library

One of our first experiments was to compare a couple of collections of poems from Ramón de Campoamor y Campoosorio. We could easily keep track of the poems that appeared in both collections. *Figure 7.2* shows the cross-reference of one isolated poem that appears in two different poem selections.

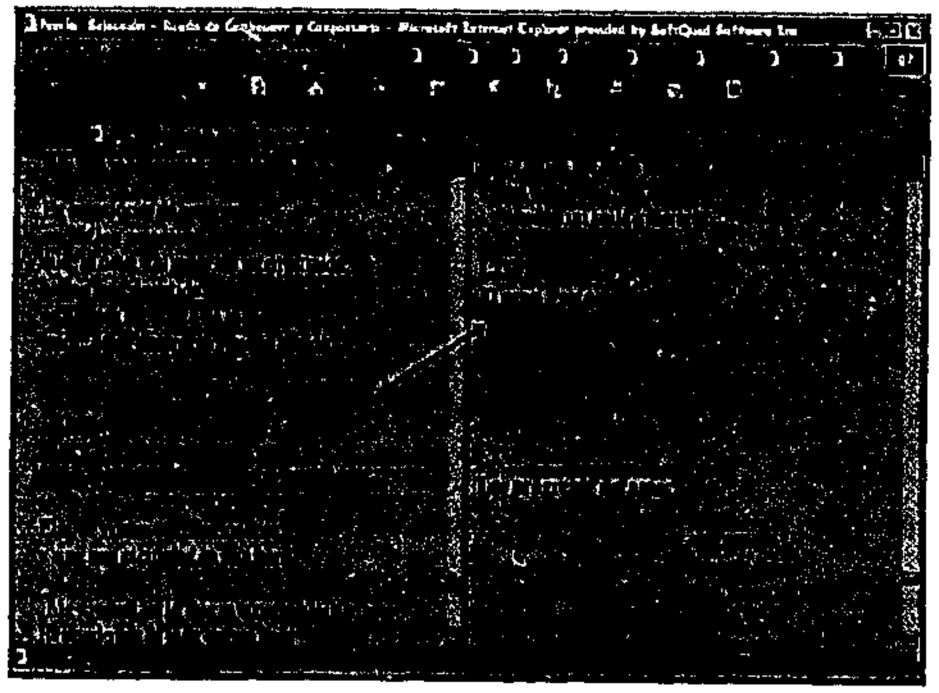


Figure 7.2. Comparison of Different Poem Collections

As an example of the problem discussed in Subsection 7.3.3.3, there are various ancient editions of the Quijote de la Mancha, all originals from Cervantes, but there are rare differences. In the example of *Figure 7.3* in line 17, we can see clear differences in the two versions of Don Quijote: in the left frame we can read "*Eran cuatro, y venian con sus quitasoles, con otros cuatro criados a caballo y dos mozos de mulas a pie*", while on the right one we see "*Eran seis, y venian con sus quitasoles, con otros cuatro criados a caballo y tres mozos de mulas a pie*". These sentences refer to the merchants from Toledo. The former case refers to six merchants, and the latter one refers to four merchants. Also these sentences refer to two servants on foot in the former case, while referring to three servants on foot in the latter case. In line 10 in *Figure 7.3* there is another difference in meaning. On the left it says "muy pensado" (thoughtfully) and on the right "muy bien pensado" (very well thought). These are examples of changes introduced by editors through the centuries.

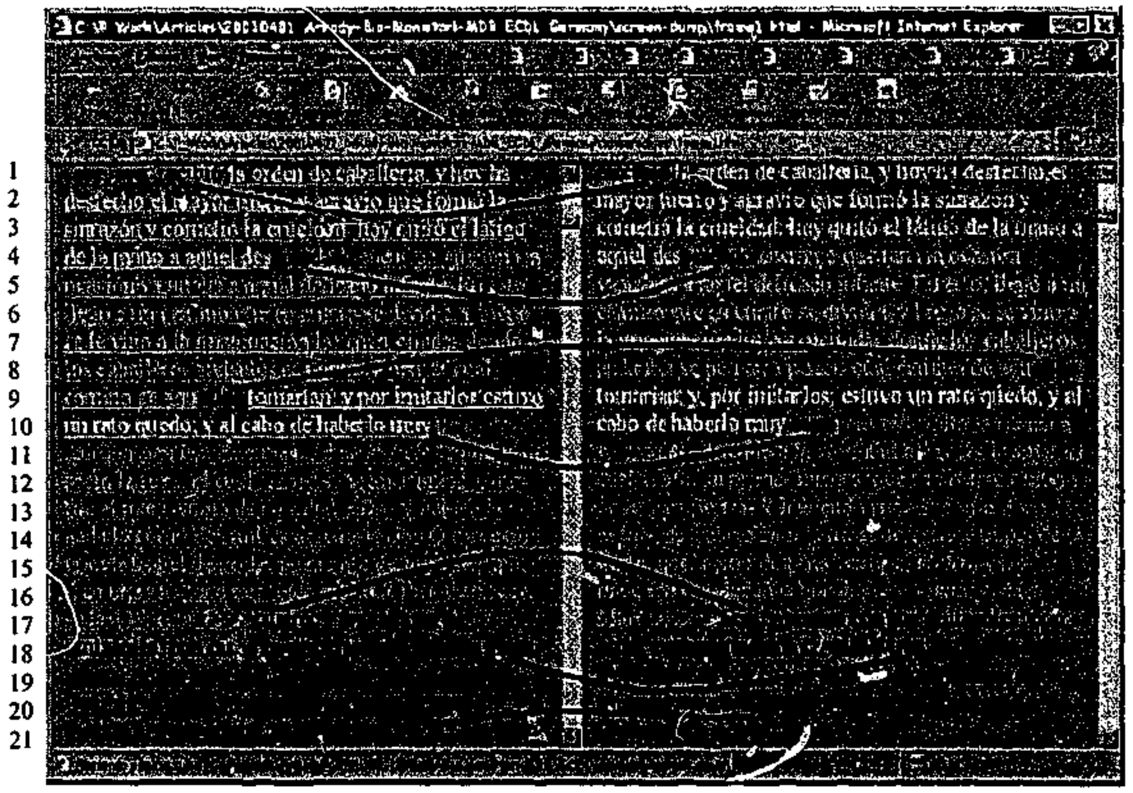


Figure 7.3. Comparing Two Editions of Don Quijote

7.4 The Converter Component

The converter component is responsible for converting documents into a common format that makes sure that slight alterations do not hinder copy detection. Firstly, the converter component has to convert documents in different file formats

into plain ASCII text. Common converters are available on the Internet that convert PDF [PDF01], PS [PS01], HTML [HTML01], MS Word Doc [Doc01], and MS Word RTF (Rich Text Format) [Doc01] files into plain ASCII files. The results of converting the same document from different file formats into plain ASCII will almost surely be different files. The most well known problem is the end of line characters: '0d' and '0a0d' in hexadecimal representation.

In order to overcome this problem and to detect overlap in case of slight alterations – i.e. capitalization of letters, multiple whitespaces, different punctuation signs – further conversion of the text files is required. We convert each letter into lowercase character and multiple whitespaces are converted into a single character. This single character is not necessarily a space character; later in this section we will explain why. We also consider every non-alphanumerical character as a white space character, which will enable us to detect overlap even if different punctuation signs are used.

The result of these restrictions is that our converted ASCII files will use an alphabet containing only alphanumerical characters and a single whitespace character. This is an alphabet of 37 characters. Also for the matching engine where suffix trees are used for document comparison, we need a unique termination symbol. Thus we only need 38 characters of the 256 ASCII characters available.

As described in Chapter 4, in a suffix tree we can use an array to represent the edges running out of a node, which is best served by an alphabet of a contiguous range of codes. We took the lowercase letters as a base starting from code 97 ('a'). All numbers are converted into the code range of 87 through to 96. The unique termination symbol can be chosen at either end of the range: 86 or 123.

Whenever we have to represent a white space character we use a coding scheme where we write the actual number of whitespaces represented by that character, rather than a single whitespace character. *Figure 7.4* shows an example of this representation. On the left the original string is shown, on the right the converted string as it looks like in a text editor, and at the bottom the ASCII codes are shown.

This representation helps us keep the formatting which is present in the original text file. When the matching engine compares files, it converts each ASCII code value smaller than the smallest ASCII code in our alphabet (87 in our system) into a single whitespace character (86 in our system). Then we have a contiguous range of

ASCII codes that are efficiently processed by the matching engine. Different punctuation signs and multiple whitespaces are ignored.

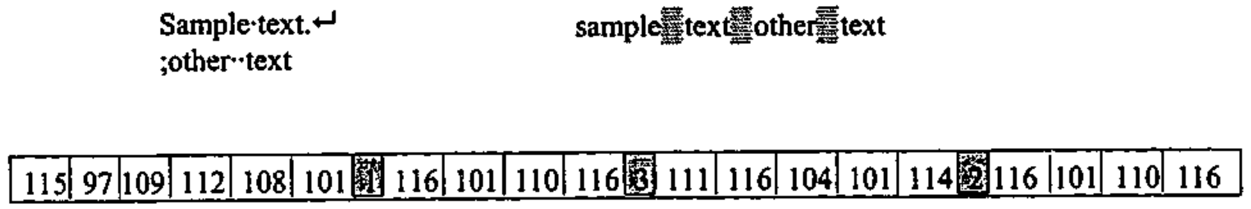


Figure 7.4. Converted ASCII File

When the visualiser component processes the positions of overlapping chunks found in the converted file, it translates the positions by taking into account the numbers representing the number of characters skipped. In the unlikely case where the number of characters skipped exceeds the maximum value (85), multiple skip positions are inserted.

Without lack of generality we would like to comment that the MDR system is capable of comparing binary files as well, though with less efficiency because of the alphabet size. However, comparing binary files is outside the scope of this thesis, since the focus of this thesis is on suffix trees and their variants. Needless to say that suffix trees are most efficient for natural language texts and DNA sequences.

7.5 The Search-Engine Component

This component of the system is responsible for chunking and registering documents in the database as well as identifying candidate documents when a suspicious document is submitted. It uses techniques similar to those discussed in Chapter 2. In this section we analyse different chunking strategies and we also look at the problem of false positives created by these techniques. The search engine component has been developed in a joint project with the Budapest University of Technology and Economics [MFZ$^+$02]. The test sets used here include the RFC document set [RFC01] and different Hungarian translations of the Bible [Kár01, BD01].

In the following subsection we discuss the hashing algorithm we have used to store chunks. In Section 7.5.2 we present the results of the tests we have run to experiment with different chunking strategies. Section 7.5.3 proposes some strategies to select representative fingerprints.

7.5.1 Hashing Chunks

Chunks are not stored as they are in the database for two reasons. First, chunks may be quite large in size, and we wish to limit the amount of storage required. Second, chunks contain intellectual property of the author of the file that we are processing. Instead of storing the chunks, we reduce them by applying a digesting tool. We use the MD5 algorithm [Riv92], which converts arbitrary byte streams to 128-bit numbers. Storing 128 bits would waste too much storage without offering any significant advantage over shorter representations. Of course, the more hex bits we store, the more accurate our system will be. Storing fewer bytes means that two chunks may produce the same digest value even when they are different. These undesired cases are called false positives and are discussed later in this subsection. Here we also note that in our system we use an extra stage when we compare documents. This extra stage is the matching engine component that uses exact string matching techniques to find the actual overlapping chunks. Thus some false positives are not of great concern, because they are eliminated in this stage.

False positives are a problem, which is common to every approach no matter what chunking or hashing scheme it uses. False positives are those cases where we have identical hash values despite the original chunks not matching. *Table 7.1* contains the number of false positives for different chunking methods and bit depths. These hash values have been generated by the MD5 algorithm by keeping only the left-most *k* bits.

Method	bit-depth	false positives	false positive (%)
hashed breakpoint (k=6)	24	8434	1.6868
hashed breakpoint (k=9)	24	6790	1.3580
overlapping (k=6)	24	7115	1.4230
sentence	24	13954	2.7908
hashed breakpoint (k=6)	32	23	0.0046
hashed breakpoint (k=9)	32	21	0.0042
overlapping (k=6)	32	26	0.0052
sentence	32	15	0.0030

Table 7.1. False Positives

The tests were carried out on 500,000 chunks. The results show a dramatic decrease in the number of false positives when we move from 24 bits to 32 bits. We suggest using 32 bits, not only because the number of false positives is less than 0.01%, but it is also an easier data width to handle in today's computers.

Of course, other systems use other hashing schemes to create the digest representation of a given chunk, and it would be interesting to compare the effect of different hashing schemes on the number of false positives. However, that comparison is beyond the scope of this thesis.

Hash values generated using the methods described above need to be stored in a database. We may choose to store the hash values in a general-purpose database management system, or alternatively, we can develop a special-purpose system tailored to store the hash values and their postings efficiently. In our system we have used Perl [WS92] hash data structures to store hash values.

7.5.2 Chunking Strategy Tests

The most promising chunking strategy among those presented in Chapter 2 is hashed-breakpoint chunking. It avoids the shifting problem without the need to store overlapping chunks. In this subsection we analyse the hashed-breakpoint strategy in detail.

We pointed out in Chapter 2 that the expected chunk length is *k* in case of *k*-hashed breakpoint chunking. If a common word happens to be a chunk boundary, the average chunk length may be much smaller than the expected average.

We have used two document sets for the comparison of chunking strategies. The first set is the set of RFC (Request for Comment) documents [RFC01]; the second set comprises different Hungarian translations of the Bible [Kár01, BD01]. We have chosen these two sets because we expect some overlap within these document sets for obvious reasons.

Figure 7.5 shows the average chunk length in the function of the *k* value. We can see that the higher the *k* value, the higher the chance is of greater deviation from the expected result. This behaviour can be explained by the uncertainty of this chunking strategy.

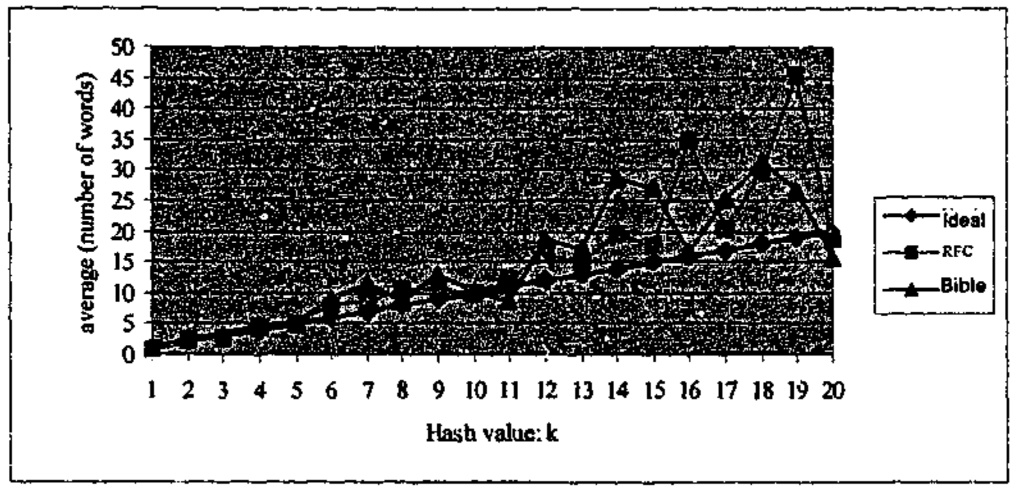


Figure 7.5. Average Chunk Length

We have picked nine pairs of documents from the Bible translations and compared them with different hash values. The results shown in *Figure 7.6* reflect the uncertainty of this chunking method.

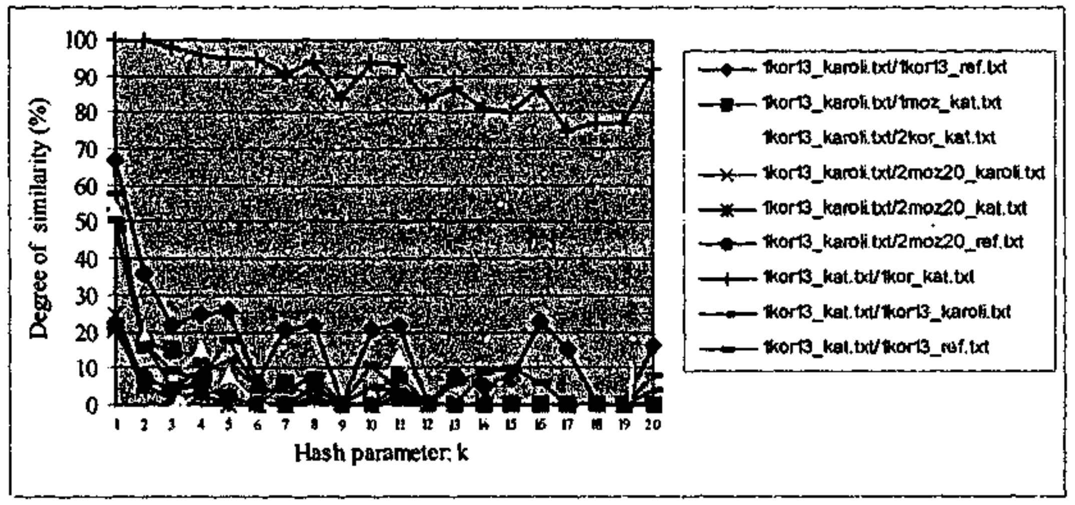


Figure 7.6. Overlap Percentage

The chosen pairs are different translations of the same part of the Bible. The correlation between *Figures 7.5* and *7.6* is obvious for the following reasons. For example, the first pair has a peak at *k*=16 following a low at *k*=14 and *k*=15. In *Figure 7.5*, we can see that at *k*=14 and *k*=15 the average chunk length is higher than expected. If we have longer chunks, the chance for overlap is smaller. On the contrary, at *k*=16 we have a low in *Figure 7.5*, which means that we have shorter chunks, so the chance for overlap is higher.

In *Figure 7.5* we can also see that the chosen *k* value has a significant effect on the amount of overlap detected. We propose to use more than one *k* value for comparison, and we can either choose the average of the reported overlaps or the maximum/minimum values depending on the actual application. This strategy eliminates the sensitivity of the algorithm to different *k* values. *Figure 7.7* shows the results of this approach. We aggregated the number of chunks reported by different *k* values and calculated the result based on the total number of chunks.

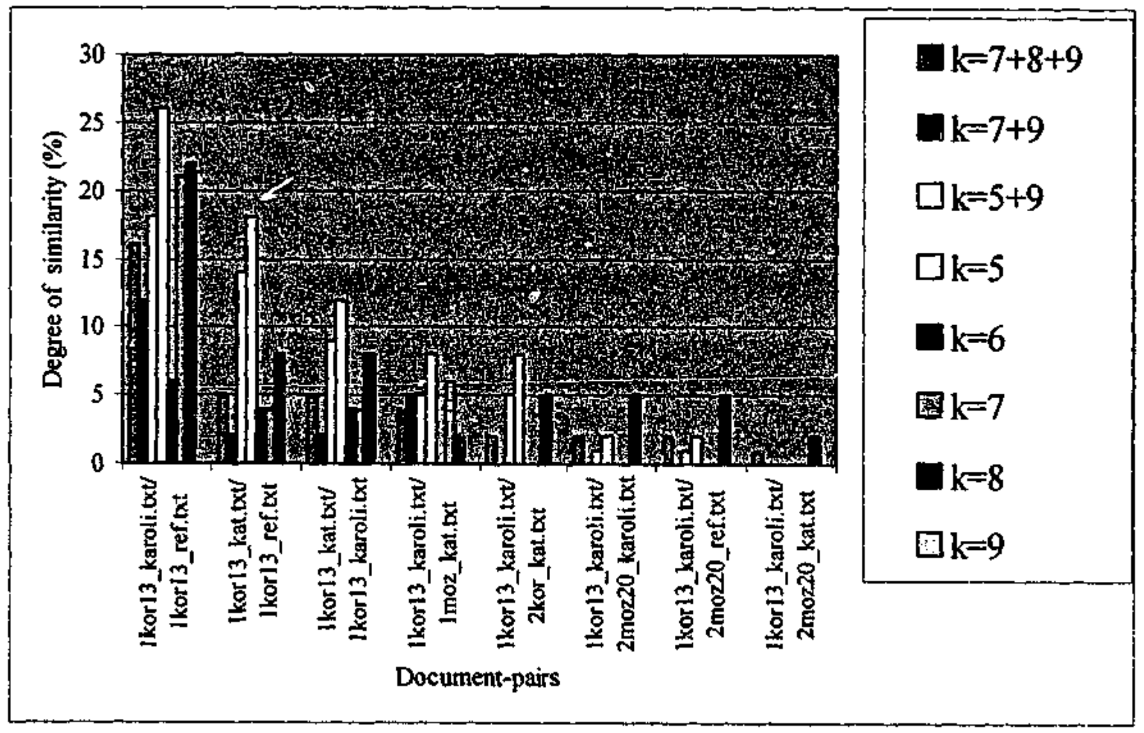


Figure 7.7. The Effect of Aggregate k Values

Of course, in our applications false positives are better than false negatives, because false positives can be eliminated in our extra stage of exact comparison. On the contrary, we will never identify documents missed by the first stage.

We have also conducted a test on different chunk sizes. In this test we have used overlapping chunks of different sizes because overlapping chunks are more predictable. The results are shown in *Figure 7.8*. As we expected longer chunks result in less detected overlap. Based on these experiences, we can conclude that for finding documents similar in style, we can choose a *k* value of 2 or 3, which would find similar phrases. For plagiarism detection, *k*=8 or *k*=9 seems to be a good parameter value, while for detecting longer overlaps, we can choose values greater than 10.

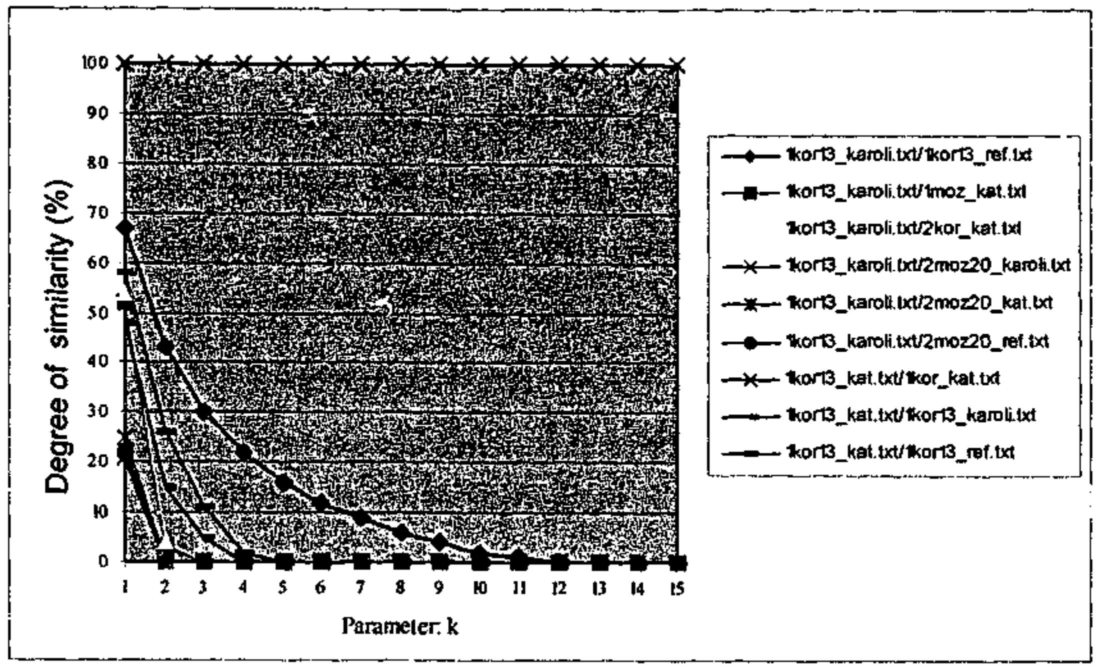


Figure 7.8. Overlapping Chunks

7.5.3 Fingerprint Selection

As discussed in Chapter 2, existing methods use some strategy to store only a certain subset of chunks, called the fingerprint. Of course, in an ideal case we would store all chunks, but long files lead to many chunks. Dealing with them all requires space for storing them and time for comparing them against other stored chunks.

However, it may not be necessary to store all chunks. The strategies used in systems discussed in Chapter 2 are mainly random. In this subsection, we propose a more strategic approach and we support the applicability of our strategy by test results.

A short chunk is not very representative of a text. The fact that two files share a short chunk does not lead us to suspect that they share ancestry. In contrast, very long chunks are very representative, but unless a plagiariser is quite lazy, it is unlikely that a copy will retain a long section of text.

We therefore discard the longest and the shortest chunks. We wish to retain similar chunks for any file. We have experimented with two culling methods. Let n be the number of chunks, m the median chunk size (measured in tokens), s the standard deviation of chunk size, b a constant, and L the length of an arbitrary chunk.

Sqrt. Retain $\lceil \sqrt{n} \rceil$ chunks whose lengths L are closest to m.

Variance. Retain those chunks such that $|L\text{-}m| \leq bs$. Increase b, if necessary, until at least $\sqrt{n}$ chunks are selected. We start with b=0.1.

We have tested our fingerprinting method on the RFC data set [RFC01]. We have found that the Sqrt method does not store enough chunks, thus the Variance method is preferable. In *Table 7.2* we show the results of these tests based on some RFC documents with known overlap. The table shows asymmetric similarity, that is RFC 1 compared to RFC 2 does not necessarily provide the same result as RFC 2 compared to RFC 1. We consider the results of the matching engine as accurate because it is based on exact matching. SE is our signature extraction method while OV is the overlapping chunk method.

Table 7.2 shows that our SE method tends to underestimate large overlaps while overestimating small overlaps (overlap is given as percentage). The overlapping-chunks method seems to provide more accurate results but at a much higher storage cost. Overestimation is not a problem in our system because we use the matching engine as a final filter, which correctly identifies overlapping chunks.

RFC 1	RFC 2	Matching Engine 1	Matching Engine 2	SE 1	SE 2	OV 1	OV 2
1596	1604	99	99	91	92	94	94
2264	2274	99	99	96	95	94	94
1138	1148	96	95	93	92	91	89
1065	1155	96	91	71	68	84	79
1084	1395	86	84	58	64	79	75
1600	1410	72	77	52	48	58	61
2497	2394	19	17	33	27	16	15
2422	2276	18	3	23	6	15	2
2392	2541	16	12	27	17	13	10

Table 7.2. Asymmetric Similarities

7.6 Generating Test Documents

The document generator component is a supplementary component that can generate documents of different size and different amounts of plagiarism. One can use different numbers of files and different sizes of chunks. One can also define how many words one wants to be substituted by synonyms. The aim of developing a document generator component was to generate test files with predefined overlap value that could be used to test the algorithms of the MDR system. The following subsection describes the algorithm used by the document generator, and performance results are given in Section 7.6.2.

7.6.1 Document Generation Algorithm

As our copy-detection algorithm works on documents converted into a unified format and defines overlap content based on the number of characters in the converted file we use converted files as our base document set. In order to have the specified overlap content we have to work on converted files because original chunks might decrease in size as a result of conversion.

The Generator has 8 input parameters:

- *min_file_size* - defines the minimum size of the document to be generated in kilobytes
- *max_file_size* - defines the maximum size of the document to be generated in kilobytes
- *min_overlap* - defines the minimum overlap content in percentage of the total size of the document
- *max_overlap* - defines the maximum overlap content in percentage of the total size of the document
- *num_files* - defines the number of files to be used for generation. It must be less than or equal to the number of base documents
- *min_char* - the minimum number of characters in a single chunk
- *max_char* - the maximum number of characters in a single chunk
- *num_words* – the number of words to be substituted in each chunk

In our algorithm we use as much randomisation as possible. Note that the parameters given above may be contradictory. For example we can define a document of size between 10K and 11K, which should contain overlap between 50% and 55% with 5 chunks from 5 different files and the size of each chunk between 2000 and 3000 characters. If we take 5 chunks with the minimum size of 2000 characters from 5 different documents we will end up with a 10000-character overlap, which should make up at most 55% of a maximum 11K document.

We have defined file size and overlap content as our priority parameters, so our algorithm will surely create a file of the size in the given range and the overlap content will also be in the given range. The algorithm will use as many files as possible for the generation up to the number of files given in the *num_files* parameters. All the chunks will be in the given range except for the last one, which might be smaller than the minimum value in case the overlap content criterion cannot be met in another way. In the following we will describe in detail how our algorithm works.

In the first step of our algorithm we generate the "genuine" part of the document, which is a random sequence of characters. These characters are from the alphabet of the converted file. We try to simulate a real document by varying the word length in a uniform distribution with a mean value of 6. The size of the genuine part is determined by the following formula:

$$random\_text\_size = \left(\min\_file\_size + \frac{\max\_file\_size - \min\_file\_size}{2}\right) \cdot \left(100 - \left(\min\_overlap + \frac{\max\_overlap - \min\_overlap}{2}\right)\right) / 100 \quad (7.1)$$

This formula defines the size of the genuine part as the mean percentage of the mean file size. The two boundary values for the genuine part would be

$$\max\_file\_size * (100 - \min\_overlap) / 100 \quad (7.2)$$

$$\min\_file\_size * (100 - \max\_overlap) / 100 \quad (7.3)$$

We chose formula (7.1) because it gives us more freedom when we add the random chunks.

The next step is to choose those documents whose chunks will be used for creating the overlap content of the document. Documents in our base document set use a naming convention of 'doc#.txt', where '#' denotes an integer value between 1 and the maximum number of files in our base document set. Now we only have to generate *num_files* random numbers between 1 and the maximum file number and file names can easily be generated from those numbers.

In the next step we extract chunks out of the document set generated in the preceding step. We take the files one by one and if we get to the last file we continue with the first one. We generate a random-size chunk within the given range, extract that chunk from the document and add it to the list of chunks. We also add a random number, which defines the position of that chunk in the document to be generated. The position value is in the range between 0 and *random_text_size*-1. *Figure 7.9* depicts how chunks are merged into the random text to form the document.

During the process of generating these chunks we keep a counter, which counts the number of overall overlap characters. After each phase, that is each chunk generation, we check if the overlap content is above *min_overlap*. If it is above *min_overlap*, we check if it is below *max_overlap*. If it is true, this step is finished.

Otherwise we do not add this last chunk to the list of chunks rather we define the minimum and maximum size of the chunk which meets the overlap criterion. If *max_char* is less than the maximum value defined above and greater than the minimum value we replace this maximum with *max_char*. If *min_char* is greater than the minimum value defined above and less than the maximum value we set *min_char* as the minimum value. We generate the last chunk with these parameters. There is a chance that the last chunk will not meet the criterion for chunks but the file size and overlap criteria will be met and we set priority on these parameters.

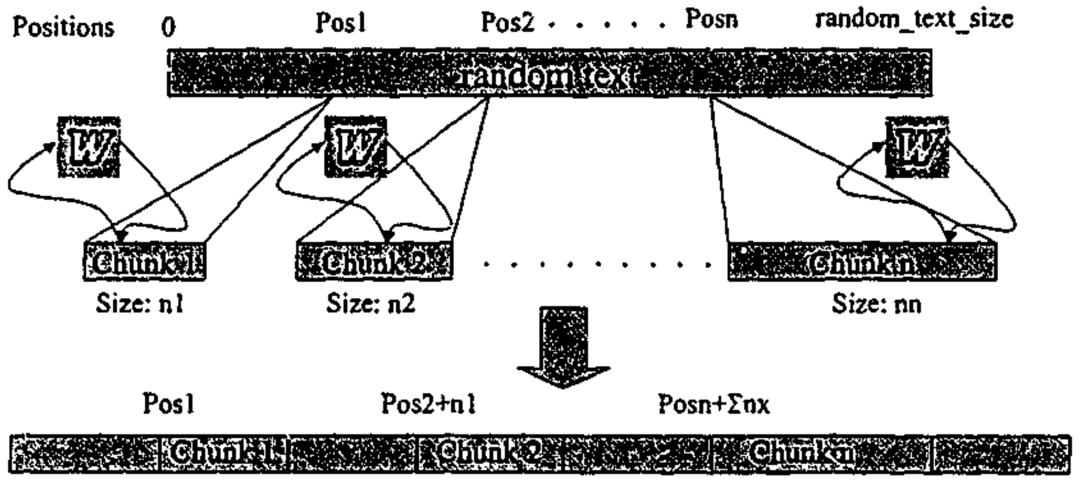


Figure 7.9. Merging Chunks into Random Text

When a chunk is extracted from a base document we substitute the required number of words with synonyms. Instead of creating a special-purpose database of synonyms for document generator we use OLE Automation to access the thesaurus of Microsoft Word. The required number of words is randomly picked from each chunk and Word is requested to provide a synonym for that word. It is possible that the randomly picked word does not have a synonym in the thesaurus. In this case we pick another word. In order not to create an infinite loop we have to put a limit on the number of attempts, which is an input parameter. Document Generator reports on the number of words that could not be replaced because of reaching the limit. With reasonable chunk sizes and reasonable limit on the number of attempts this value is likely to be 0.

The last step of our algorithm is to insert the chunks into the random text. When we create the chunks and add them to a list we can keep this list ordered, which helps us in the merging process. We take the first chunk of the list, which has the lowest position value and insert that chunk at the given position generated in the preceding step. We keep a counter, which stores the shift value, that is the number of character shifts due to previous insertions. Each insertion will increase this counter by the number of characters inserted, so the next insertion has to add the counter to the position value to insert that chunk to the proper position.

Before giving a summary of steps in our algorithm there is one more issue to be addressed. In real documents boundaries of chunks are also boundaries of words. To simulate this effect when a random size chunk is generated we adjust the boundary of the chunk to word boundaries and also during the merging process we insert chunks at word boundaries. The pseudo code of our algorithm is depicted in *Figure 7.10.*

```
Calculate the size of random text
Generate random text
Generate document set to be used
while min_overlap not reached do
  generate chunk from the next file
  if max_overlap is reached when chunk is added then
    define proper min and max values
    generate chunk with those values
  end if
  generate position for this chunk
  add chunk to ordered list
end while
shift <- 0
for i:=0 to 'size of list' - 1 do
  insert chunk at position+shift
  shift:=shift+'size of chunk'
end for
```

Figure 7.10. The Pseudo Code of the Document Generation Algorithm

7.6.2 Performance Analysis

In this subsection we analyse the performance of the above algorithm. The algorithm was implemented in Visual C++ and experiments were run on a PC with

Intel Pentium II 433MHz processor, 128MB RAM, and running Windows NT Workstation.

We have experimented with a fixed document size. Because of the random nature of the algorithm this varies between 290 and 310 kilobytes (min_file_size =290, max_file_size=310). We also fixed the chunk size between 800 and 900 bytes. The plagiarised portion of the generated file was experimented at 5 different values (50,000 bytes, 100,000 bytes, 150,000 bytes, 200,000 bytes, and 250,000 bytes). Again these values are approximate values because the file size and the overlap content are both given as a range. We also varied the number of files to be used for plagiarism (1,2,3,4,5). After each run we had the matching engine define the overlap content to check whether the plagiarised content is around the desired approximate value. Running time of the algorithm in these 25 cases are shown in *Figure 7.11*.

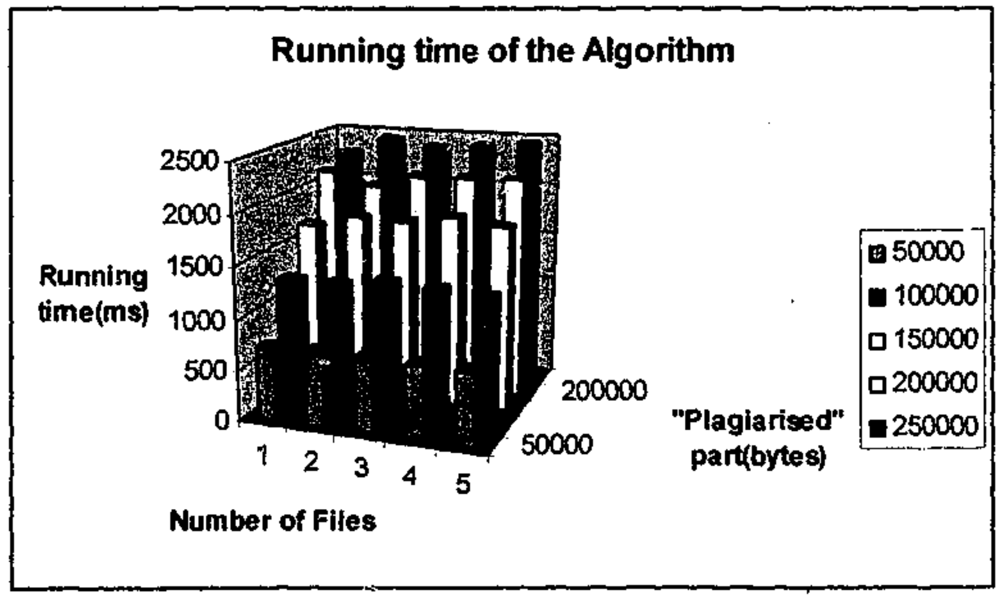


Figure 7.11. Running Time of the Algorithm

Figure 7.11 suggests that the number of files used have little effect on the running time of the algorithm. The only difference we would expect is the difference of the number of files to be opened but it is not significant. Running time increases if we increase the amount of "plagiarised" text because more chunks are read from files and processed. Generating the "genuine" part of the document is not so time-critical. According to a separate measurement it remained under 100ms in the above cases.

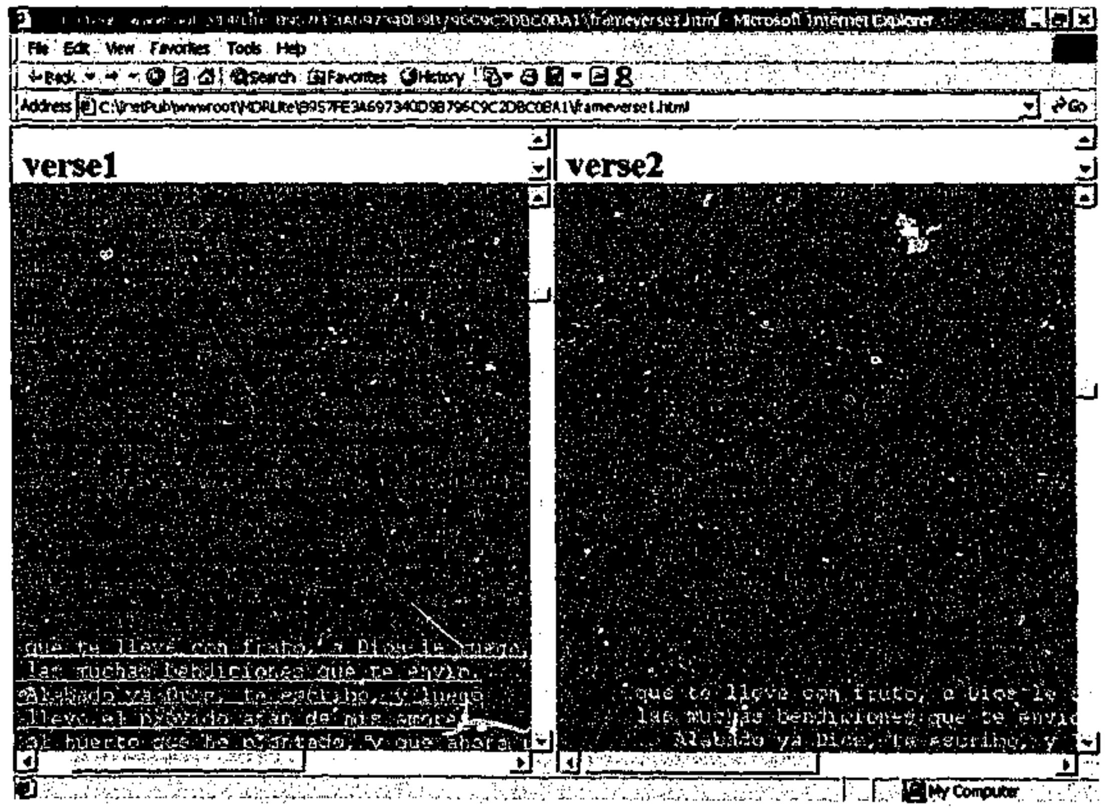
The running time significantly increases if we use word substitutions. If we set the number of words to be changed to 5 in each chunk, and set the attempt limit to 20 attempts in the simplest case when 50000 bytes are "plagiarised", we might end up calling MS Word more than 5000 times in the worst case. This test ran for 48 seconds. This is very time-consuming but remember that it is a case of creating a 200-page document w: . more than 50 half-page plagiarised chunks.

7.7 Visualising Results

The visualiser component takes the output of the matching engine and generates HTML files that show the overlap between documents. The idea is that we establish links between correlated documents and clicking on those links brings up the other document and jumps to the position where the overlappi.; chunk is located.

The interface is designed for one-to-many comparisons and the left pane always contains the suspicious document, while the right pane shows updated candidate documents while we traverse through the original document. Two different interfaces are proposed that could visualize the results of MDR in a user-friendly manner. These interfaces are discussed below.

The first interface uses different colours to denote the overlapping chunks. Ten different colours are identified and they are reused if more than 10 overlapping chunks are present in the document. Each chunk is also a link and by clicking on the link we jump to that section of the candidate document where the matching chunk can be found. *Figure 7.12* shows an example output.


frameverse1.html - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address C:\InetPub\wwwroot\MDRLite\B957FE3A697340D9B796C9C2DBC0BA1\frameverse1.html
Go
My Computer

verse1

que te lleve con fruto, a Dios le ruego,
las muchas bendiciones que te envío.
Alabado ya Dios, te escribo, y luego
llevo el pesado afán de mis amores
al huerto que he plantado, y que ahora

verse2

que te lleve con fruto, a Dios le
las muchas bendiciones que te envi
Alabado ya Dios, te escribo, y

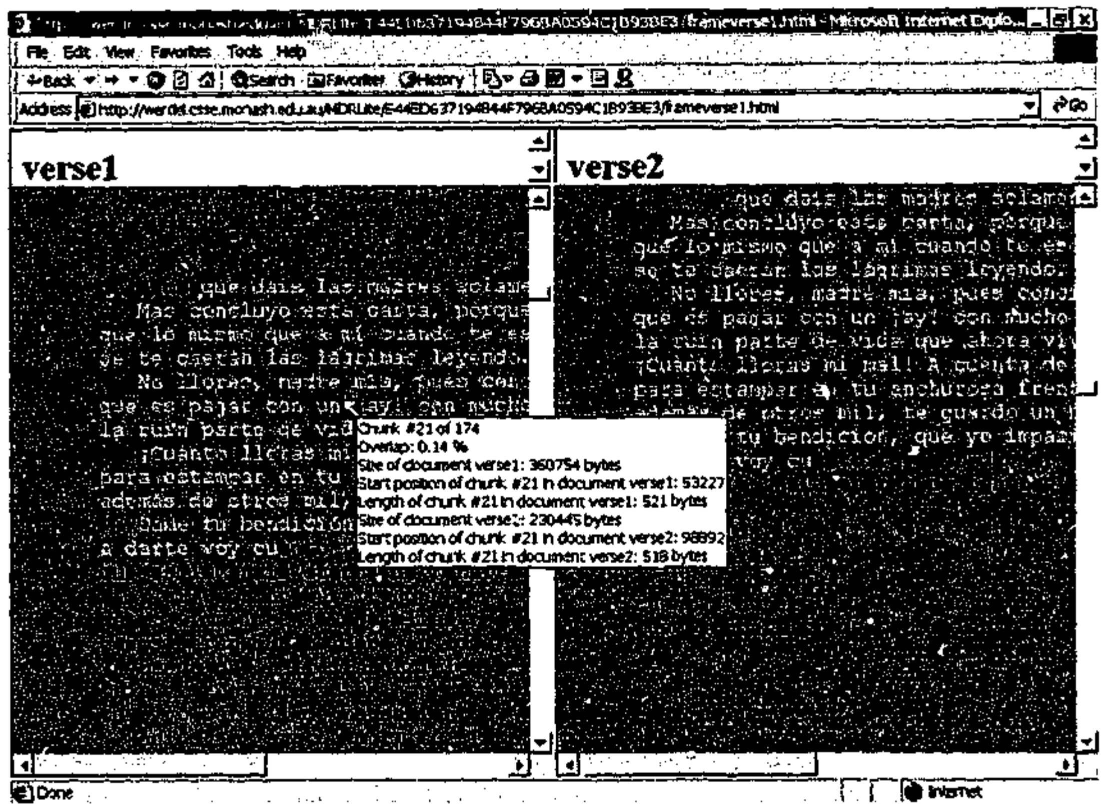
Figure 7.12. Output of the Visualiser Component

In this representation the interface contains four panes. The top panes contain the name of the documents shown in the bottom panes. The left panes show the original document while the right panes show the current candidate document. The current candidate document .:ay change because the original document may overlap with more than one document and the right panes always contain the corresponding candidate document and the name of that document.

The visualiser takes the position and length values, which are identified for the converted document, and calculates the positions in the original document, which may be different, since multiple whitespaces are eliminated in the converted document. Then it places HTML tags in the document to mark up overlapping chunks. It also generates the relevant colouring tags.

In the other interface the basic look-and-feel is the same as that of the previous one, though we make use of the features of JavaScript. In this interface each overlapping chunk is depicted by the same colour and once the mouse is over a given chunk it changes its colour, a tooltip shows information on that chunk and in the right pane the candidate document is automatically scrolled to the overlapping chunk and its colour also changes. The information shown in the tooltip includes the percentage of text the given chunk represents, the total number of overlapping chunks in the document, the sequence number of the given chunk, the position in both the original and the candidate document, the length of the chunk in both documents, and the overall size of both documents.

The process of generating these documents is very similar to the previous method. The only difference is that the markups that need to be inserted are more complex because they contain JavaScript code beside standard HTML tags. An example of this second interface is shown in *Figure 7.13*.


File Edit View Favorites Tools Help
Back
Search
Favorites
History
Address
http://werdst.csse.monash.edu.au/MDRLite/E44ED637194844F7968A0594C1B93BE3/frameverse1.html
Go
verse1
verse2
Done
Internet

```
 ,que dais las madres solame
   Mas concluyo esta carta, porque
que lo mismo que a mi cuando te es
se te caerán las lágrimas leyendo.
   No llores, madre mía, pues con
que es pagar con un ¡ay! con much
la ruin parte de vid
   ¡Cuanto lloras mi
para estampar en tu
además de otros mil,
   Dame tu bendición
a darte voy cu
```

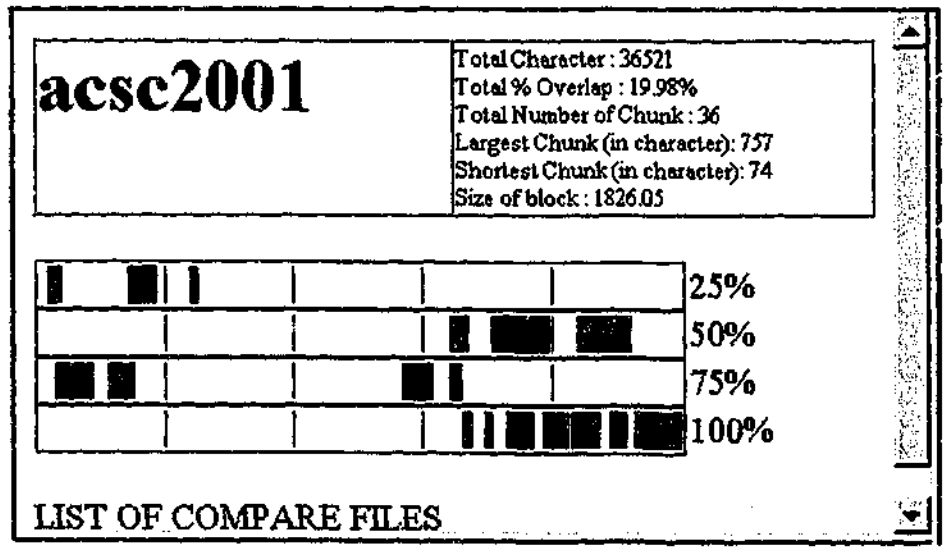
```
Chunk #21 of 174
Overlap: 0.14 %
Size of document verse1: 360754 bytes
Start position of chunk #21 in document verse1: 53227
Length of chunk #21 in document verse1: 521 bytes
Size of document verse2: 230445 bytes
Start position of chunk #21 in document verse2: 98892
Length of chunk #21 in document verse2: 518 bytes
```

```
        que dais las madres solame
   Mas concluyo esta carta, porque
que lo mismo que a mi cuando te es
se te caerán las lágrimas leyendo.
   No llores, madre mía, pues cono
que es pagar con un ¡ay! con mucho
la ruin parte de vida que ahora vi
¡Cuánto lloras mi mal! A cuenta de
para estampar en tu anchurosa fren
además de otros mil, te guardo un
tu bendición, que yo impa
voy cu
```

Figure 7.13. Output of the Visualiser Component

We have also experimented with expanding the upper information windows with information on the entire file. An example of that is depicted in *Figure 7.14*.

Besides the statistical information in the table at the top it contains a map of overlapping chunks in the document. Moving the mouse over any of those chunks brings up information specific to the given chunk in a pop-up window. Clicking on chunks in the document map scrolls the original document to the position of the given chunk and the right pane is also updated with the corresponding candidate document.



acsc2001	Total Character : 36521 Total % Overlap : 19.98% Total Number of Chunk : 36 Largest Chunk (in character): 757 Shortest Chunk (in character): 74 Size of block : 1826.05

LIST OF COMPARE FILES

Figure 7.14. Information Window

We believe that the visualiser component is a very important component that could be further developed to suit the needs of different end users as well as different applications. The prototype implemented in the MDR system has proved to be very successful despite the fact that the visualiser component fell outside the scope of this thesis.

7.8 Parallel Applications

Since we are considering the comparison of a huge document set, i.e. the Internet, it is possible that the comparison jobs cannot be handled by a single computer or process. We have analysed parallel approaches to compare documents [MZS99, MZS00a]. Before going into the details of how the algorithm works on a parallel platform we discuss different available platforms for the comparison.

One option for reducing comparison time is to use a local cluster. Clusters of workstations have emerged in recent years because the computational power of workstations has dramatically improved and their cost has remained low. They are more cost-effective solutions than high-performance parallel computers.

There are numerous cluster systems of commodity workstations built for different applications. The Berkeley NOW project [NOW01] uses Solaris Workstations and applies an OS layer called GLunix (Global Layer Unix) above Unix to provide a single system image. This layer provides transparent remote execution, support for interactive parallel and sequential jobs, load balancing, and a cluster-wide namespace. It also provides network RAM, which enables heavily loaded workstations to use the memory of idle workstations. The NOW project has also investigated in network interface hardware and fast communication protocols.

The Beowulf project [Beo94] emphasizes the use of mass-market commodity components. They use PCs running a modified Linux, which is able to handle multiple parallel Ethernet networks. Research has shown that up to three networks can be bundled together to obtain significant improvement in throughput. The project includes several programming environments, e.g. PVM, MPI, BSP etc.

Solaris-MC [Sol96] is a distributed operating system for a cluster of Solaris-based PCs and workstations. It is built on the top of the Solaris kernel to provide a global single system image. The most interesting feature of Solaris-MC is that it uses an object-oriented framework (CORBA) for interprocess communication. It also features a distributed file system called ProXy File System (PXFS).

In our project we have used the Monash Parallel Parametric Modelling Engine (PPME) at the School of Computer Science and Software Engineering, Monash University. The PPME is a cluster of high-end PCs. The system specifications included 22 x 330 MHz Pentium II processors; 32 x 350 MHz Pentium II processors; 8 x 500 MHz Pentium III processors; 5.8 Gbyte RAM; 180 Gbyte disk space at the time of our tests. Since then the machines have been upgraded.

All of the 32 machines are dual-processor Pentiums. 26 machines can be booted in either Linux or Windows NT while the remaining six machines are running Linux. One of the Linux-only machines serves as a cluster server. Two parts of the cluster are located on two different campuses and they are connected by an ATM network.

Parametric experiments can be easily executed using the Active Tools Clustor program [Clu99]. The capabilities of the Clustor tool are briefly described below. Clustor enables the easy execution of jobs over a predefined parameter range. If there are, for example, 5 parameters and we want to test our algorithm with 5 different values for each parameter it means that we have to execute $5^5 = 3125$ jobs. These jobs are dispatched and scheduled by Clustor according to a plan file, which can be generated by a graphical tool. Basically Clustor executes an executable file 3125 times with different parameters using as many processors as available and balances the load among them.

In the following subsection we present the results of the tests run on our local cluster using the Clustor tool. In Section 7.8.2 we describe the capabilities of the MPI library and Section 7.8.3 contains the results of the test runs using the MPI Library.

7.8.1 Comparing Documents Using a Local Cluster with the Clustor Tool

As discussed in Section 7.8, the Clustor tool provides an easy-to-use interface and a high-level service to submit jobs, which are then distributed among nodes. Our main problem is to compare a single document (suspicious document) to many documents (candidate documents). One job is to compare the suspicious document to one candidate document. When these comparisons have finished the results of separate comparisons must be merged. In Chapter 3 we have discussed how we can calculate the overlap from the matching statistics values. If multiple comparisons are done simultaneously we simply have to store the maximum value of the matching statistics for all comparisons and then the overlap value can be calculated using the same algorithm as described in Chapter 3 though we have to collect the results from different nodes.

To use Clustor we have to provide a plan file that describes the parameter range for the task, which executables must be executed on each node before any of the jobs start, the parameterised execution of the main task, and the post-processing phase.

In our case the parameter range is the range of document identifiers to be compared to the suspicious document. In the pre-processing phase we copy the suspicious document to each node. The main task reads the suspicious document, builds a suffix tree on it, and compares a candidate document to the suffix tree. The results of the comparison are saved in a file, which is copied back to the root node either right after the execution or in a batch after all jobs have finished.

This is the simplest and most straightforward application of the Clustor tool but there are a few problems to it. We have to read the suspicious document multiple times and build the tree multiple times because Clustor is unable to retain anything in memory between two executions. We could build the tree once in the pre-processing phase and then have each process read the tree rather than the document but the tree consumes more space than the document, and in practice it is slower to read the tree than to read the document and build the tree. Another enhancement could be to batch documents together and compare a batch of documents to the suspicious document. It reduces the overhead of rebuilding the tree for every new candidate document, since we only have to rebuild it for every batch.

When we performed our tests using Clustor we did not receive the speed-up that we had expected, so we compared the actual running time of a job to the running time given by Clustor. Results are shown in *Figure 7.15*.

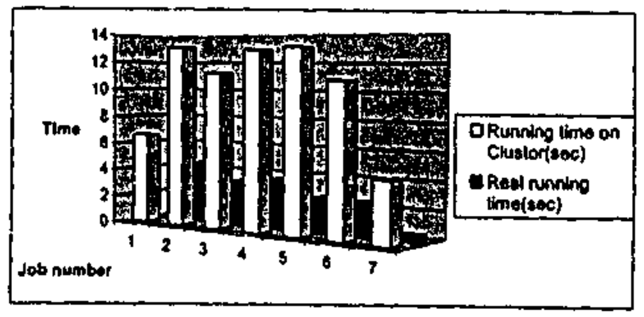


Figure 7.15. Running Time on Clustor

The difference between the two values is the overhead of copying files between nodes and initiating processes. The results show that the Clustor tool is not suited for short-running jobs. We have also experimented with different number of nodes for comparing documents. The results are shown in *Figure 7.16.*

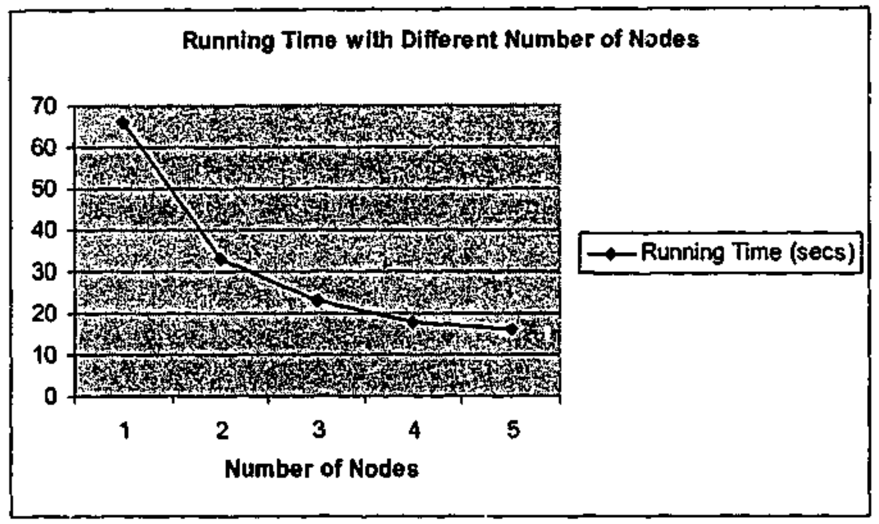


Figure 7.16. Different Number of Nodes

The figure shows that the speed-up is not increasing linearly with the number of nodes and applying more than 5 nodes does not reduce the running time much further. These jobs are very data-intensive, because they include copying of several candidate documents from one node to another. The Clustor tool is an excellent tool for parameterised execution and we can treat our problem as a parametric execution problem but the overhead is too much to use it in case of data-intensive jobs. The following subsections introduce approaches that use the MPI library and bypasses Clustor, so job distribution and execution is all controlled by the processes themselves.

7.8.2 MPI Library

In this subsection we introduce the message-passing model and describe why we have chosen the MPI Library to implement our parallel algorithms. The most popular parallel computational models include the data parallel model, the shared memory model, message passing, and remote memory operations [GLS99].

Data parallelism appears at many levels in computing but it was first introduced in vector processors. Data parallelism is now more a programming style than an architecture.

In the shared memory model different processes have access to the same memory space and they communicate by means of load and store operations. Today's PCs are becoming small-scale shared-memory machines and they are often referred to as symmetric multiprocessors (SMP).

In the remote memory operations model, processes are able to directly read and write memory spaces of other processes without the involvement of the process the data space belongs to. A similar model is the active message model [ECGS92] where the operations are performed by a small subroutine in the address space of the other process. Here we note that the MPI-2 standard [GLT99] also supports remote memory operations.

The message-passing model assumes individual processes with their own memory space and they communicate by means of messages, which require operations from both processes. The message-passing model has many advantages over other models. It is universal in a way that it fits many hardware and software configurations. The message-passing model has been found to be a useful and complete model, which also eliminates the most common problem in parallel programs, the overwriting of memory. With a proper debugger the developer may even see the status of message queues. The message-passing model gives a lot of control to the developer, who then is capable of optimising the code for caching and communication.

Because of the above-mentioned advantages message passing libraries had been developed independently in the past: PICL [GHPW90], PVM [BGMS91], PARMACS [BRH90], p4 [BL94], Chameleon [GS93], Zip-code [SSD$^+$94], and TCGMSG [Har91]. The MPI Forum [MPI01] was formed in 1992 to create a standard that combines all ideas of the above systems. The first MPI standard was released in 1994, while the latest is the MPI-2 standard, which was completed in the summer of 1997.

In our project we have used the Windows MPI implementation [WMPI01] of the MPI Library from Critical Software. The current version at the time of writing is 1.54. It did not pose any limitations, because we have not found the need for any of the extra features that are present in the MPI-2 standard. In the following subsection we describe how we used the MPI Library to bypass the overhead of the Clustor Tool.

7.8.3 Using the MPI Library to Compare Documents on a Local Cluster

If we use the MPI library for communication between processes comparing documents we do not have to pay the overhead of the Clustor tool. The most important improvement is that we can keep the suspicious document permanently in memory. The basic concept is that we have a master process that knows the names of candidate documents. There are several slave (or worker) processes that process documents. Slave processes read the suspicious document (it can be sent by the master process) and build the suffix tree for that document. Then this tree can be permanently kept in memory. Slave processes contact the master process when they need a new candidate document to be analysed. Slave processes can also keep the matching statistics array in memory and they only need to send that array back to the master process when the entire comparison task has finished.

Different architectures are possible for placing the master process and the slave processes. It is also a question whether the slave processes should use the shared file system to read the files, or have the master process, or another process, send the content of the file to the slave process for processing. The third problem is how to place the repository of files. We can store those files in a central database or we can have it distributed among processing nodes. *Figure 7.17* shows the architecture that resembles much of the only possible architecture of the approach using Clustor.

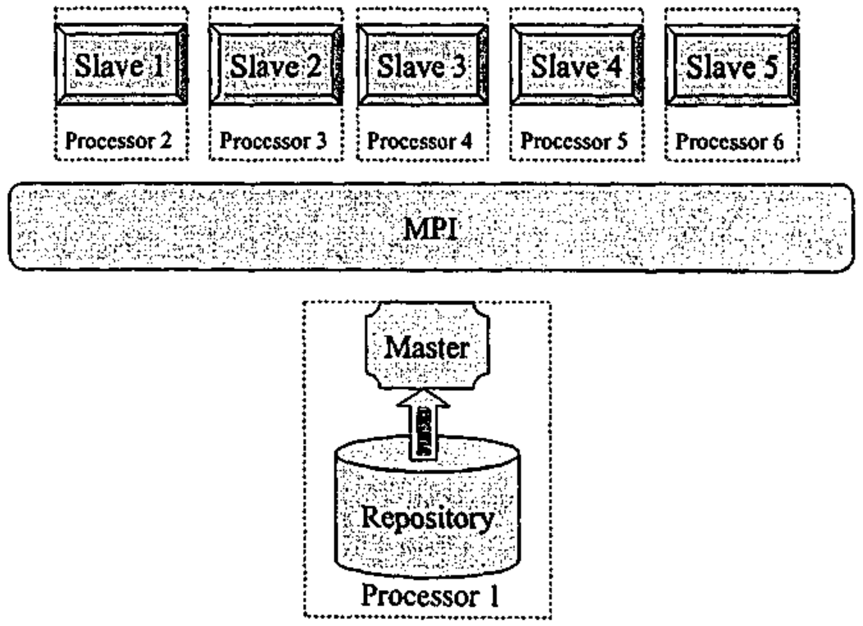


Figure 7.17. Clustor-like Architecture

First we compared the performance of using this approach and found that the overhead was reduced significantly. Then we compared the same batch of documents in two different architectures. The first architecture was the one depicted in *Figure 7.17* and the second one used the shared file system. We used 3 slave nodes and comparing 70 documents took 60 seconds using the shared file system, while it used only 40 seconds when the master process distributed the files.

To run a heavy test of the parallel algorithm we have compared 2590 RFC documents pairwise. The total size of the documents is 114MB (pure ASCII text). We have to emphasize that the matching engine algorithm used in these tests is suited for comparing a single (or a few) suspicious document to many candidate documents and it is not suited for pairwise comparison. Our pairwise comparison is equivalent to comparing 2590 suspicious documents to a 114MB repository or one file to 295GB of documents. The reason for choosing this pairwise comparison is that it generates enough jobs to run a heavy test of the algorithm. We used a fully replicated repository, that is all RFC files were stored on each node. We used one master process running on one node and one slave process each, running on the three other nodes. The architecture is depicted in *Figure 7.18*.

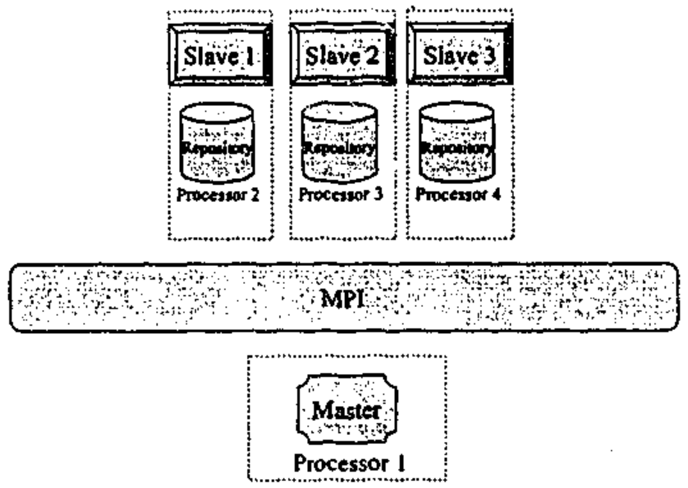


Figure 7.18. Distributed Repository Architecture

The running time of the algorithm was 164,817secs, which translates into a throughput of approximately 2Mbyte of documents per second. This result is reasonable because 2MByte is about 1200 pages of text. This experiment shows that parallel comparison of documents is feasible. Replication techniques should also be analysed to have a feasible distribution of documents. We have to stress again that our matching engine algorithm is only intended to be used in a second phase of a comparison – only on documents that have previously been identified as candidate documents by the search-engine.

7.9 Summary

In this chapter we have described the MatchDetectReveal prototype system that we built to test the feasibility of the algorithms proposed in previous chapters. The architecture of the MDR system was presented and different components were discussed.

The core component of the system is the matching engine that uses suffix trees and suffix vectors along with the matching statistics algorithm to identify overlap between documents. This component can only analyse a limited number of documents. Thus the search-engine operates as a pre-filter, which only feeds those documents to the matching engine that potentially overlap with the submitted document. The search-engine uses algorithms similar to those used in prototype systems discussed in Chapter 2. We have run different tests to find appropriate chunking and hashing methods, and we have also devised a fingerprinting method that is more strategic than the random methods of other prototype systems.

Our experiments were run on three different document sets that contain overlap among themselves. The RFC documents [RFC01] contain revisions of and responses to other documents in the set, different Hungarian translations of the Bible [Kár01, BD01] also overlap for obvious reason, and the documents of the Miguel de Cervantes Digital Library [BP01] revealed a few more applications of the system. In the context of the Cervantes Digital Library we discussed how the MDR system could aid librarians in many research areas.

The theoretical algorithms that we have discussed in previous chapters present the results as positions and chunk lengths. We introduced a visualiser component into the system that transforms these data into a presentation, which is easily perceptible by human. The results are shown in a Web-browser, thus no extra training is required for users to use the system.

The system was also supplemented with a document generator component that generated test documents with predefined overlap content. As a preparation for future extensions of the system the document generator is capable of substituting some words with their synonyms, which could later test the similarity and overlap rule interpreter component that will define how to handle synonyms, and changing of data and e.g. localities.

With the ever-increasing computational power of workstations, parallel computing is more readily available even for applications such as our document comparison. In the last section we have analysed how standard tools and tailored solutions can be beneficial to our document overlap problem area.

The aim of building the MDR system was to prove in practice that the algorithms presented in previous chapters are feasible and practical. The results show that these algorithms are very effective and the system can be used in many application areas, which is demonstrated by the different document sets we have used in our tests.

CHAPTER EIGHT

Conclusion

8.

8.1 Summary

This thesis has presented efficient algorithms and data structures for document overlap detection. The main focus was on the suffix tree structure and the matching statistics algorithm. The suffix tree structure is a versatile data structure that can be used not only in document comparison applications but also in many other areas of computing.

The main contribution of the thesis is the new space-efficient representation of a suffix tree: the suffix vector [MZV01, MZS02]. The suffix vector representation is more space-efficient than any other representation known to date with the same versatility, meaning the suffix vector representation can be used in any application where suffix trees are used. While there are other representations that may be more space-efficient, they are restricted for use only in certain algorithms.

The suffix vector representation is a conceptually new representation, although it can be homomorphically mapped to the suffix tree structure. Two alternative physical representations have been proposed: the general representation and the compact representation. The former can be directly built from a text file and is easier to extract data. The compact representation is the more space-efficient representation; this is better in terms of space-requirements than any other representation [MZV01]. The compact representation can be derived from either a suffix tree or a general suffix vector. The construction algorithm and its time-complexity have also been presented for the general representation [MZS02].

The suffix vector representation eliminates redundancies present in other suffix tree representations, hence some algorithms can run faster on this representation as they do not need to examine redundant parts of the tree multiple times. One of these algorithms is the matching statistics algorithm, which is heavily used in our document comparison system. Examples have been presented throughout this thesis to reinforce these concepts.

We have implemented the suffix vector representation and compared its space requirement to the most space-efficient representation to date. Kurtz [Kur99] uses a document set of 42 files to analyse the space requirement of his representation. We used the same set of documents and demonstrated that our representation is superior. Differences between the space requirements of Kurtz's representation and the proposed suffix vector vary between document types. We have also analysed the internal structure of the suffix vector representation and pointed out the causes of these differences.

We have also analysed the time-efficiency of the proposed suffix vector structure. Our comparison is based on primitive operations we have defined. We then analysed how data can be extracted from the representation by using these primitive operations. We have defined three crucial pieces of information that need to be extracted from the suffix tree. These operations have been analysed, and we have demonstrated that our representation requires fewer primitive operations than that of Kurtz [Kur99]. The other advantage of our structure, which we have already mentioned, is that redundant information is eliminated. Thus certain algorithms, such as the matching statistics algorithm, need to extract less information from the structure because redundant data is not extracted multiple times.

As previously mentioned, the suffix vector data structure is a general structure that can be used wherever suffix trees are used. We have also investigated special suffix tree representations that aid document comparison. Tailored suffix tree representations may have extra benefits for some applications.

One modification we investigated was the sparse suffix tree; this represents only those suffixes that start at the beginning of words [MZS99]. This representation has been proposed by other researchers in the past, however it has not been analysed for document comparison purposes. We have shown that the matching statistics algorithm can utilize this new representation and the running time of the algorithm is radically reduced.

The other modification that we have analysed is the Directed Acyclic Graph (DAG) representation of a suffix tree. Similarly to the suffix vector representation, the DAG representation merges some isomorphic parts of the suffix tree. In this thesis we have shown that the suffix vector representation is capable of eliminating even more redundancies. The DAG representation in its original form could not serve the matching statistics algorithm. Consequently extra modifications were proposed to prepare the structure for the matching statistics algorithm [MZS01].

The matching statistics algorithm has been used throughout this thesis to demonstrate the applicability of different structures. The reason for that is that we have analysed these structures in the context of document comparison. We have found that the matching statistics algorithm that operates on a suffix tree is an effective algorithm to define overlap between documents.

Having analysed existing copy-detection and copy-prevention methods we found copy-detection to be a more effective way of protecting intellectual property as it does not impede bona fide researchers and users [MFZ$^+$02]. Copy-detection methods proposed in the past share some common characteristics. All divide the text into smaller chunks. These chunks are then hashed to a more space-efficient representation and the comparison is based on the equality of the hash values. For space limitation reasons not all of these values can be stored in the repository. The process of selecting the chunks to be retained is called fingerprinting. The main problem with this approach is the uncertainty surrounding the chunking and fingerprinting methods. We have proposed a second phase, after using the aforementioned stage, to identify exact overlapping chunks of documents. This second phase is the matching statistics algorithm on those data structures discussed above.

We have found that the time-efficiency of the matching statistics algorithm can be improved in the event we need to compare one document to many documents. The original matching statistics algorithm would build a suffix tree for each candidate document to be compared to the suspicious document. We have illustrated that the suffix tree can be used in a reverse fashion where only one suffix tree needs to be built for the suspicious document and other documents can be compared to that single suffix tree [MZS99].

Throughout this thesis data structures and algorithms are discussed in the context of document comparison. Document comparison is a very broad term and we have found many specific applications where our prototype system could be used. The prototype system we have built is called the MatchDetectReveal (MDR) system; it is a complete document comparison environment, which can be used for numerous applications by setting different parameters [MZS00b].

In Chápter 7 the system architecture was presented and potential applications were analysed. The primary application of our system is plagiarism detection, which was our initial goal at the start of the project. As the project progressed we found other application areas including clustering a file system, cross-references in digital libraries, and search-engines. We have worked on a joint project with the Miguel de Cervantes Digital Library and their librarians have found the tool very useful in many areas, discussed in detail in Chapter 7 [ZBM01].

The MDR system has shown that the algorithms and data structures proposed in this thesis are effective in document comparison applications. One of the interesting components of the system is the parallel comparison engine, which uses a local cluster for comparing documents on a large scale. Both standard tools and message passing libraries have been tested and proved successful in our experiments on parallel environments [MZS99, MZS00a].

8.2 Future Work

Possible future extensions of the proposed algorithms and data structures have been identified in respective chapters in this thesis. This section revisits these problems and provides a summary of them.

The search-engine component of the system is a pre-filter that feeds a manageable-sized document set to our matching engine. This component has been developed in a joint project with the Budapest University of Technology and Economics. However some issues remain open, such as what is the best hashing scheme, how to tackle the uncertainty of hashed breakpoint chunking, and what is the effect of chunk sizes on false positives.

Another open research area is to determine how the advantages of data structures discussed here could be combined to create a more efficient representation. As an example, it is a future research direction to find out how the DAG representation could be combined with the sparse suffix tree representation.

There are certain other aspects of the system wrapped up in different components that need to be further analysed and developed. Perhaps the most important issue is how to handle inexact matches. Methods based on exact-comparison can be cheated by slight modifications to the text, which must be targeted in plagiarism-detection applications. Here we note that in case of substantial overlap between the modified texts, these methods are still useful. We have proposed a similarity and overlap rule interpreter component that would handle these problems in the future.

We have also explored some straightforward parallelization of the proposed algorithm, again many issues still remain open: what is the best distribution of the repository, what is the best topology of nodes, how many nodes can be used for significant speed-up?

We hold a firm belief that systems like MDR and research associated with them have many future applications and will contribute to advancement of knowledge.

Glossary

AVL Tree. A balanced binary tree where the height of the two subtrees (children) of a node differs by at most one. Named after Adelson-Velskii and Landis.

Candidate Document. A document identified by the search engine that overlaps with the original document.

Canonical Form of a Document. The form of the document after conversion, which tries to eliminate minor differences, such as different case, punctuation, spacing, etc.

Compact Suffix Vector. A physical suffix vector representation that consumes the smallest space but cannot directly be built in linear time.

C_{tree}. The number of comparison steps required in the suffix tree during the matching statistics algorithm.

C_{vector}. The number of comparison steps required in the suffix vector during the matching statistics algorithm.

DAG. Directed Acyclic Graph. A data structure that can be used to eliminate some redundancies of a suffix tree.

DAWG. Directed Acyclic Word Graph. The minimal deterministic automaton (not necessarily complete) that accepts all suffixes of *S*.

Downward Extension. An extension of a vector box where a node with a depth less than the current smallest node depth is added to the box.

False Negative. Identical chunks that are not reported overlapping.

False Positive. Chunks that are reported to be overlapping when they are different.

FAQ. Frequently Asked Questions.

Fingerprint. A representative selection of chunks of a document.

Functionally Reduced Suffix Vector. A suffix vector representation that does not include next node and suffix link information.

General Suffix Vector. A physical suffix vector representation that can be built in linear time.

Genuine Document. In plagiarism applications a document considered to be original work.

Hashed Breakpoint Chunking. A chunking strategy where chunk boundaries are determined with the help of a hash function.

IHTI. Improved Hash Table Implementation. A physical suffix tree representation proposed by Kurtz.

ILLI. Improved Linked List Implementation. A physical suffix tree representation proposed by Kurtz.

Implicit Suffix Tree. A suffix tree where a suffix does not necessarily finish at the end of a leaf.

Large Node. A node that needs to be represented in its entirety in the suffix vector representation.

Long Depth. The depth of a node is long if it is more than 127.

Long Edge. An edge, whose length is greater than or equal to 256 characters.

Longest Common Prefix. The longest common prefix of two strings is the prefix of both strings with the greatest length.

Longest Common Subsequence. A common subsequence of two or more strings that is of greatest length.

Longest Common Substring. The longest common substring of two or more strings is the substring of all strings which is of greatest length.

Matching Statistics Algorithm. An algorithm that defines the longest overlap present at each position of a given string.

Matching Statistics Position. The position of the chunk in the candidate document where the longest overlapping chunk from this position can be found.

Matching Statistics Value. The length of the longest overlapping chunk from a given position.

MDR. MatchDetectReveal. Our prototype system that has been developed to test the algorithms.

Natural Edge. An edge implicitly represented in the suffix vector representation.

Normal (Regular) Edge. An edge that is explicitly stored in the suffix vector representation.

Operational Test. Computerized tests that approximate violation tests. They are objective.

Reduced Box. A vector box that stores information on multiple nodes but those nodes are identical. Therefore only information on one node is stored.

RFC. Request for Comments. A series of notes, started in 1969, about the Internet (originally the ARPANET). The notes discuss many aspects of computer communication.

Regular Box. A vector box that is not a reduced box.

SBST. Suffix binary search tree. A suffix tree representation developed by Irving et al.

SCAM. Stanford Copy Analysis Mechanism. A copy-detection system developed by Garcia-Molina et al.

Shingle. A chunk of text.

Sif. A file-comparison tool developed by Manber.

Shift-rule. A shortcut used in exact string-matching algorithms.

Short Depth. The depth of a node is short if it is less than 128.

Short Edge. An edge whose length is less than 256 characters.

SHTI. Simple Hash Table Implementation. A physical suffix tree representation proposed by Kurtz.

SLLI. Simple Linked List Implementation. A physical suffix tree representation proposed by Kurtz.

Sparse Suffix Tree. A special suffix tree that contains only certain suffixes of a string.

Subsequence. A subsequence of a string is a sequence of characters obtained by deleting zero or more characters from it.

Substring. A consecutive range of characters within a string.

Suffix. Any substring in a string, whose last character is the last character of the string.

Suffix Link. A link between two nodes in a suffix tree where the label of the node pointed to by the suffix link can be obtained by eliminating the first character of the label of the originating node.

Suffix Tree. A data structure that contains each suffix of a word.

Suffix Vector. The proposed new data structure that has a one-to-one correspondence with suffix trees.

Suspicious Document. A document that is suspected to contain overlap with other documents.

Upward Extension. An extension of a vector box where a node with a depth greater than the current greatest node depth is added to the box.

Vector Box. The building block of the suffix vector data structure.

Violation Test. Decisions made by humans regarding document comparison tasks. These decisions are subjective.

References

[AB97] Arnold R. and Bell T. A Corpus for Evaluation of Lossless Compression Algorithms. *Proceedings of Data Compression Conference*, pp 201-210, 1997. URL http://corpus.canterbury.ac.nz

[AC75] Aho A, Corasick M. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM,* **18**:333-340, 1975.

[AG86] Apostolico A. and Giancarlo R. The Boyer-Moore-Galil String Searching Revisited. *SIAM Journal of Computing*, **15**:98-105, 1986.

[AGK00] Abramson, D., Giddy, J. and Kotler, L. High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid?. *International Parallel and Distributed Processing Symposium (IPDPS)*, pp 520- 528, Cancun, Mexico, May 2000.

[Aho90] Aho A. V. Algorithms for Finding Patterns in Strings. In Leeuwen J., editor, *Handbook of Theoretical Computer Science*. Chapter 5, pp. 257-300, Elsevier Science Publisher B.V., 1990.

[AHU74] Aho A. V., Hopcroft J. E., and Ullman J. D. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[ALS99] Andersson A., Larsson N. J., Swanson K. Suffix Trees on Words. *Algorithmica* **23**: 246-260, 1999.

[AN95] Andersson A., Nilsson S. Efficient Implementation of Suffix Trees. *Software - Practice and Experience* **25** (2): 129-141, 1995.

[Apo85] Apostolico A. The Myriad Virtues of Subword Trees, in A. Apostolico and Z. Galli. *Combinatorial Algorithms on Words.* (Springer-Verlag Heidelberg) pp. 85-96, 1985.

[BB99] Baker M. and Buyya R. Cluster Computing at a Glance in *Buyya R. High Performance Cluster Computing*. (Prentice Hall) pp. 3-47, 1999.

[BBE$^+$84] Blumer A., Blumer J., Ehrenfeucht A., Haussler D., and McConnell R. Building a Complete Inverted File for a Set of Text Files in Linear Time. *Proceedings of the 16th ACM Symposium on Theory of Computing*, 349-358, 1984.

[BBE$^+$85] Blumer A., Blumer J., Ehrenfeucht A., Haussler D., Chen M.T., and Seiferas J. The Smallest Automaton Recognizing the Subwords of a Text. *Theoretical Computer Science* **40** (1), 31-56, 1985.

[BCW90] Bell T.C., Cleary J.G., and Witten I.H. *Text Compression*. Prentice Hall, Englewood Cliffs, NJ, 1990.

[BD01] Békés G., Dalos P. Újszövetségi szentírás. *A külföldi katolikus magyar akció kiadása*, Róma, 1951. URL http://www.cadvision.com/mayl/bd-ind.html, 2001.

[BDG95] Brin S., Davis J., Garcia-Molina H. Copy Detection Mechanisms for Digital Documents. *Proceedings of the ACM SIGMOD Annual Conference*, pp. 398-409, San Francisco, CA, May 1995.

[Beo94] Beowulf Project (1994). URL http://beowulf.gsfc.nasa.gov/, 1994.

[BG92] Baeza-Yates R. and Gonnet G. A New Approach to String Searching. *Communications of the ACM*, **35**: 74-82, 1992.

[BGM97] Broder A.Z., Glassman S.C., Manasse M.S. Syntatic Clustering of the Web. *Proceedingsof the Sixth International World Wide Web Conference, World Wide Web Consortium, Cambridge*, pp. 391-404, 1997.

[BGMS91] Beguelin A., Geist G. A., Manchek R., and Sunderman V. A User's Guide to PVM: Parallel Virtual Machine. Technical Report TM-11826, Oak Ridge National Laboratory, Oak Ridge, TN, 1991.

[BL94] Butler R. and Lusk E. Monitors, Messages, and Clusters: The p4 Parallel Programming System. *Parallel Computing*, **20**: 547-564, April, 1994.

[BLMO94a] Brassil J., Low S., Maxemchuk N., and O'Gorman L. Technical report, AT&T Bell Labratories, 1994. URL ftp://ftp.research.att.com/dist/brassil/docmark2.ps, 1994.

[BLMO94b] Brassil J., Low S., Maxemchuk N., and O'Gorman L. Electronic marking and identication techniques to discourage document copying. Technical report, AT&T Bell Labratories, 1994.

[BM77] Boyer R. S., Moore J. S. A Fast String Searching Algorithm. *Communications of ACM*, **20**:762-772, 1977.

[BP01] Bia A. and Pedreno A. The Miguel de Cervantes Digital Library: The Hispanic Voice on the WEB. *LLC (Literary and Linguistic Computing) Journal*, Oxford University Press, **16**(2): 161-177, 2001.

[BR99a] Baeza-Yates R., Ribeiro-Neto B. Searching the Web *in Baeza-Yates R., Ribeiro-Neto B. Modern Information Retrieval*, ACM Press, 1999.

[BR99b] Baeza-Yates R., Ribeiro-Neto B. Modeling *in Baeza-Yates R., Ribeiro-Neto B. Modern Information Retrieval*, ACM Press, 1999.

[BRH90] Bomans L., Roose D., Hempel R. The Argonne/GMD Macros in FORTRAN for Portable Parallel Programming and Their Inplementation on the Intel iPSC/2. *Parallel Computing*, **15**: 119-132, 1990.

[BRN99] Baeza-Yates R., Ribeiro-Neto B., Navarro G. Indexing and Searching *in Baeza-Yates R., Ribeiro-Neto B. Modern Information Retrieval*, ACM Press, 1999.

[CMPS94] A.K. Choudhury, N.F. Maxemchuk, S. Paul, H.G. Schulzrinne. Copyright Protection for Electronic Publishing over Computer Networks, *IEEE Network Magazine*, May/June 1995. Vol. 9 No. 3 1994. pp. 12-20, 1994.

[CL94] Chang W.I. and Lawler E.L. Sublinear Approximate String Matching and Biological Applications. *Algorithmica* **12**:327-344, 1994.

[Clu99] Clustor Manual. URL http://hathor.cs.monash.edu.au/clustor/, 1999.

[Cop01] CopyCatch System. URL http://www.copycatch.freeserve.co.uk, 2001.

[CS85] Chen M.T., Seiferas J. Efficient and Elegant Subword Tree Construction, in Apostolico A. and Galli Z. *Combinatorial Algorithms on Words*. (Springer-Verlag Heidelberg) pp. 97-107, 1985.

[CV97] Crochemore M. and Vérin R. On Compact Directed Acyclic Word Graphs, in *Structures in Logic and Computer Science, a selection of essays in honor of A. Ehrenfeucht*, J. Mycielski, G. Rozenberg and A. Salomaa, eds., LNCS 1261, Springer-Verlag, 192-211, 1997.

[Den95] Denning P. J. Editorial: Plagiarism in the Web. *Communications of the ACM*, December 1995, Vol. 58, No. 12, pp.29, 1995.

[Doc01] Doc2txt converter. URL http://plaza27.mbn.or.jp/~satomii/soft/cui/doc2txt.html, 2001.

[ECGS92] Eicken T. von, Culler D. E., Goldstein S. C., and Schauser K. E. Active messages: A mechanism for integrated communication and computation. *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, pp. 256-266, 1992.

[EGGI92] Eppstein D., Galil Z., Giancarlo R., and Italiano G.F. Sparse Dynamic Programming I: Linear Cost Functions. *Journal of the ACM*, Vol 39, No 3. July 1992, pp 519-545, 1992.

[EVE00] EVE Plagiarism Detection System. URL http://www.canexus.com, 2000

[FK97] Foster I. and Kesselman C. Globus: A Metacomputing Infrastructure Toolkit. *Intl J Supercomputer Applications* **11**(2), pp. 115-128, 1997.

[FKS84] Fredman M. L., Komlos J., Szemeredi E. Storing a Sparse Table with O(1) Worst Case Access Time. *Journal of the ACM*, **31**: 61-68, 1984.

[FP74] Fischer M. and Patterson M. String-Matching and Other Products. In Karp R. M., editor, *Complexity of Computation*, pp. 113-125. SIAM-AMS Proc., 1974.

[FSK82] Felsenstein J, Sawyer S., and Kochin R. An Efficient Method for Matching Nucleic Acid Sequences. *Nucleic Acids Res.*, 10:133-139, 1982.

[FZMS02] Finkel R. A., Zaslavsky A., Monostori K., Schmidt H. Signature extraction for overlap detection in documents. *Australasian Computer Science Conference, Monash University, Melbourne, Victoria, 28 January - 1 February, 2002*, pp 59-64, 2002.

[GG88] Galil Z. and Giancarlo R. Data Structures and Algorithms for Approximate String Matching. *Journal of Complexity* **4**, 33-72, 1988.

[GGS96] Garcia-Molina H., Gravano L., Shivakumar N. dSCAM: Finding Document Copies Across Multiple Databases. *Proceedings of 4th International Conference on Parallel and Distributed Information Systems (PDIS'96)*, pp. 68-79, Miami Beach, Florida, 1996.

[GHPW90] Geist G. A., Heath M. T., Peyton B. W., and Worley P. H. PICL: A Portable Instrumented Communications Library, C Reference Manual. Technical Report TM-11130, Oak Ridge National Laboratory, Oak Ridge, TN, July 1990.

[GK97] Giegerich R. and Kurtz S. From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction. *Algorithmica* **19**:331-353, 1997.

[GKS99] Giegerich R., Kurtz S., Stoye J. Efficient Implementation of Lazy Suffix Trees. *Algorithm Engineering 1999*: 30-42, 1999.

[Gla99] Glatt Plagiarism Screening Program. URL: http://www.plagiarism.com/screen.id.htm, 1999.

[GLS99] Gropp W., Lusk E., Skjellum A. Using MPI. Portable Parallel Programming with the Message-Passing Interface. 2nd Edition. The MIT Press, 1999.

[GLT99] Gropp W., Lusk E., Thakur R. Using MPI-2: Advanced Features of the Message-Passing Interface. The MIT Press, 1999.

[Gri93] Griswold G. N. A method for protecting copyright on networks. *In Joint Harvard MIT Workshop on Technology Strategies for Protecting Intellectual Property in the Networked Multimedia Environment*, URL http://www.cni.org/docs/ima.ip-workshop/Griswold.html 1993.

[GS93] Gropp W. D. and Smith B. Chameleon Parallel Programming Tools Users Manual. *Technical Report ANL-93/23*, Argonne National Laboratory, Argonne, IL, March, 1993.

[GS95a] Garcia-Molina H., Shivakumar N. The SCAM Approach to Copy Detection in Digital Libraries. *D-lib Magazine*, November, 1995.

[GS95b] Garcia-Molina H., Shivakumar N. SCAM: A Copy Detection Mechanism for Digital Documents. *Proceedings of 2nd International Conference in Theory and Practice of Digital Libraries* (DL'95), pp. 155-163, June 11 - 13, Austin, Texas, 1995.

[GS96] Garcia-Molina H., Shivakumar N. Building a Scalable and Accurate Copy Detection Mechanism. *Proceedings of 1st ACM International Conference on Digital Libraries (DL'96)*, pp. 160-168, March, Bethesda Maryland, 1996.

[Gus97] Gusfield D. Algorithms on Strings, Trees, and Sequences. Computer Science and Computational Biology. Cambridge University Press, 1997.

[GV00] Grossi R., Vitter J. S. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing (STOC 2000)*, May 21-23, 2000, Portland, OR, USA. 397-406, 2000.

[HAI01] Hunt E., Atkinson M. P., Irving R. W. A Database Index to Large Biological Sequences. Proceedings of the 27th VLDB Conference, Rome, Italy, pp. 139-148, 2001.

[Har91] Harrison R. J. Portable Tools and Applications for Parallel Computers. *International J. Quantum Chem.*, **40** (847), 1991.

[Hei96] Heintze N. Scalable Document Fingerprinting. Proceedings of the Second USENIX Workshop on Electronic Commerce, Oakland, California, 18-21 November, 1996. URL http://www.cs.cmu.edu/afs/cs/user/nch/www/koala/main.html, 1996.

[Hir77] Hirschberg D. S. Algorithms for the Longest Common Subsequence Problem. *Journal of ACM*, 24:664-675, 1977.

[Hor80] Horspool N. Practical Fast Searching in Strings. *Software - Practice and Experience*, **10**:501-506, 1980.

[HT84] Harel D. and Tarjan R. E. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, **13**(2): 338-355, 1984.

[HTML01] HTML2txt Converter. URL http://www.infomedia.it/artic/Baccan/opensource/, 2001.

[HS91] Hume A. and Sunday D. M. Fast String Searching. *Software - Practice and Experience*, **21**:1221-1248, 1991.

[IL00] R.W. Irving and L. Love. The suffix binary search tree and suffix AVL tree, *University of Glasgow, Computing Science Department Research Report*, TR-2000-54, February 2000.

[Int01] InteriGuard System. URL http://www.integriguard.com, 2001.

[Kár01] Károli G. Szent Biblia, 1591. URL http://www.cab.u-szeged.hu/WWW/books/Biblia/, 2001.

[Kär95] Kärkkäinnen J. Suffix Cactus: A Cross between Suffix Tree and Suffix Array. *Proceedings of the Annual Symposium on combinatorial Pattern Matching (CPM'95), LNCS 937*, pp. 191-204, 1995.

[KD95] Kosaraju S. R. and Delcher A. L. Large Scale Assembly of DNA Strings and Space-Efficient Construction of Suffix Trees. *Proceedings of the 27th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 169-177, 1995.

[KMP77] Knuth D. E., Morris J. H., Pratt V. B. Fast pattern matching in strings. *SIAM Journal of Computing*, 6:323-350, 1977.

[Koc99] Kock. N. A case of academic plagiarism. *Communications of the ACM*, July 1999, Vol. 42, No.7, pp. 96-104, 1999.

[KU96] Kärkkäinnen J. and Ukkonen E. Sparse Suffix Trees. *Computing and Combinatorics, Second Annual International Conference, COCOON '96, Hong Kong, June 17-19, 1996, Proceedings*, 219-230, 1996.

[Kur99] Kurtz S. Reducing the Space Requirement of Suffix Trees. *Software-Practice and Experience*, **29** (13), 1149-1171, 1999.

[LI93] Lefevre C. and Ikeda J.-E. The Position End-Set Tree: A Small Automaton for Word Recognition in Biological Sequences. *Comp. Appl. Biosci.*, 9(3): 343-348, 1993.

[LV89] Landau G. M. and Viskin U. Fast Parallel and Serial Approximate String Matching. *Journal of Algorithms* **10**, 157-169, 1989.

[Man94] Manber U. Finding Similar Files in a Large File System. *Proceedings of the 1994 USENIX Conference*, pp.1-10, 1994.

[McC76] McCreight E. M. A Space-Economical Suffix Tree Construction Algorithm. *Journal of ACM*, **23** (2), 262-272, 1976.

[MFZ^{+}02] Monostori K., Finkel R., Zaslavsky A., Hodász G., Pataki M. Comparison of Overlap Detection Techniques. *The 2002 International Conference on Computational Science, Amsterdam, The Netherlands, 21 - 24 April, 2002*. (I) pp 51-60, 2002.

[MM93] Manber U., Myers E.W. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, **22** (5):935-948, 1993.

[Mor86] Morrison D. R. PATRICIA - Practical Algorithm to Retrieve Information Coded in Alphanumeric. *Journal of ACM* **15** (4), 514-534, 1986.

[MPI01] The MPI Forum. URL http://www.mpi-forum.org, 2001.

[MR75] Majster M. E. and Reisner A. Efficient On-Line Construction and Correction of Position Trees. *SIAM Journal of Computing* **9** (4), 346-351, 1975.

[Mye86] Myers E. W. An O(ND) Difference Algorithm and Its Variations. *Algorithmica* **1** 251-266, 1986.

[Mye88] Myers E. W. A Four Russian Algorithm for Regular Expression Pattern Matching. *Tech. Report 88-34*, Dept. of Computer Science, University of Arizona, Tucson, AZ, 1988.

[MZB01] Monostori K., Zaslavsky A., Bia A. Using the MatchDetectReveal System for Comparative Analysis of Texts. *Proceedings of the Sixth Australasian Document Computing Symposium (ADCS 2001), Pacific Bay Resort, Coffs Harbour, 7 December, 2001*, pp 51-58, 2001.

[MZS99] Monostori K., Zaslavsky A., Schmidt H. Parallel Overlap and Similarity Detection in Semi-Structured Document Collections. *Proceedings of the 6th Annual Australasian Conference on Parallel and Real-Time Systems (PART '99), Melbourne, Australia, 1999*. pp 92-103, 1999.

[MZS00a] Monostori K., Zaslavsky A., Schmidt H. Parallel and Distributed Document Overlap Detection on the Web. *Workshop on Applied Parallel Computing – PARA2000, 18-21 June 2000, Bergen, Norway*. pp 206-214, 2000.

[MZS00b] Monostori K., Zaslavsky A., Schmidt H. MatchDetectReveal: Finding Overlapping and Similar Digital Documents. *Information Resources Management Association International Conference (IRMA2000), 21-24 May, 2000 at Anchorage Hilton Hotel, Anchorage, Alaska, USA*. pp 955-957, 2000.

[MZS00c] Monostori K., Schmidt H., Zaslavsky A. Document Overlap Detection System for Distributed Digital Libraries. *ACM Digital Libraries 2000 (DL00), 2-7 June, 2000 in San Antonio, Texas, USA*. pp 226-227, 2000.

[MZS01] Monostori K., Zaslavsky A., Schmidt H. Efficiency of Data Structures for Detecting Overlaps in Digital Documents. *Australasian Computer Science Conference, Bond University, Gold Coast, Queensland, 29 January - 2 February, 2001*. pp 140-147, 2000.

[MZS02] Monostori K., Zaslavsky A., Schmidt H. Suffix Vector: Space- and Time-Efficient Alternative To Suffix Trees. *Proceedings of the 25th Australasian Computer Science Conference, Monash University, Melbourne, Victoria, 28 January - 1 February, 2002*, pp 157-166, 2002.

[MZV01] Monostori K., Zaslavsky A., Vajk I. Suffix Vector: A Space-Efficient Suffix Tree Representation. *Proceedings of the International Symposium on Algorithms and Computation, Christchurch, New Zealand, Dec 19-21, 2001*, pp 707-718, 2001.

[Net01] Evaluating the Size of the Internet. URL http://www.netsizer.com, 30/11/2001

[NOW01] NOW Project. URL http://now.cs.berkeley.edu/, 2001.

[Pap01] PaperBin system. URL http://www.paperbin.com, 2001.

[PDF01] PDF2txt Converter, URL http://alkaline.vestris.com/files/asearch-distrib/WinNT/, 2001.

[Pla99] Plagiarism.org, the Internet plagiarism detection service for authors & education. URL http://www.plagiarism.org , 1999.

[PS01] PS2txt Converter, http://www.research.digital.com/SRC/virtualpaper/pstotext.html, 2001.

[Rab81] Rabin M.O. Fingerprinting by random polynomials. Center for Research in Computing Technology, Harvard University, TR-15-81, 1981

[RFC01] RFC Overview. URL http://www.rfc-editor.org/overview.html, 2001.

[Riv92] Rivest R. L. RFC 1321: The MD5 Message-Digest Algorithm. *Internet Activities Board*, April 1992.

[SLL97] Si A., Leong H.V., Lau R. W. H. CHECK: A Document Plagiarism Detection System. Proceedings of ACM Symposium for Applied Computing, pp.70-77, Feb. 1997.

[Smi91] Smith P. D. Experiments With a Very Fast Substring Search Algorithm. *Software - Practice and Experience*, **21**:1065-74, 1991.

[Smi94] Smith P. D. On tuning the Boyer-Moore-Horspool String Searching Algorithm. *Software - Practice and Experience*, **24**:435-36, 1994.

[Sol96] Solaris-MC Project. URL http://www.sunlabs.com/research/solaris-mc/, 1996.

[SSD+94] Skjellum A., Smith S. G., Doss N. E., Still C. H., Leung A. P., and Morari M. The Zipcode Message Passing System. In Fox G. C., editor, *Parallel Computing Works!* Morgan Kaufman, 1994.

[Sun90] Sunday D. M. A Very Fast Substring Search Algorithm. *Communications of the ACM*, 33:132-142, 1990.

[Ukk85] Ukkonen E. Algorithms for Approximate String Matching. *Information and Control* 64, 100-118, 1985.

[Ukk95] Ukkonen E. On-Line Construction of Suffix Trees. *Algorithmica* 14, pp. 249-260, 1995.

[Wei73] Weiner P. Linear Pattern Matching Algorithms. *Proceedings of the 14th IEEE Symposium on Switching and Automata Theory*, pp. 1-11, 1973.

[WMPI01] Windows MPI. URL http://www.criticalsoftware.com/wmpi/, 2001.

[WS92] Wall L. and Schwartz R. L. Programming Perl. *O'Reilly & Associates*, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 1992.

[ZBM01] Zaslavsky A., Bia A., Monostori K. Using copy-detection and text comparison algorithms for cross-referencing multiple editions of literary works. *Proceedings of the 5th European Conference on Research and Advanced Technology for Digital Libraries September 4-9 2001, Darmstadt, Germany*, pp 103-114, 2001.