UNF UNIVERSITY of NORTH FLORIDA

UNF Digital Commons

UNF Graduate Theses and Dissertations | Student Scholarship

2006

Approximate String Matching With Dynamic Programming and Suffix Trees

Leng Hui Keng
University of North Florida

Suggested Citation

Keng, Leng Hui, "Approximate String Matching With Dynamic Programming and Suffix Trees" (2006). *UNF Graduate Theses and Dissertations*. 196.
https://digitalcommons.unf.edu/etd/196

This Master's Thesis is brought to you for free and open access by the Student Scholarship at UNF Digital Commons. It has been accepted for inclusion in UNF Graduate Theses and Dissertations by an authorized administrator of UNF Digital Commons. For more information, please contact Digital Projects.
© 2006 All Rights Reserved

UNF UNIVERSITY of NORTH FLORIDA

APPROXIMATE STRING MATCHING WITH DYNAMIC PROGRAMMING AND SUFFIX TREES

by

Leng Hui Keng

A thesis submitted to the
School of Computing
in partial fulfillment of the requirements for the degree of

Master of Science in Computer and Information Sciences

UNIVERSITY OF NORTH FLORIDA
SCHOOL OF COMPUTING

December, 2006

Copyright (©) 2006 by Leng Hui Keng

All rights reserved. Reproduction in whole or in part in any form requires the prior written permission of Leng Hui Keng or designated representative.

The thesis "Approximate String Matching with Dynamic Programming and Suffix Trees" submitted by Leng Keng in partial fulfillment of the requirements for the degree of Master of Science in Computer and Information Sciences has been

Approved by the thesis committee:

Signature deleted

Date

Dec 19, 2006

Yap S. Chua
Thesis Adviser and Committee Chairperson

Signature deleted

Dec. 19, 2006

Roger E. Eggen

Signature deleted

Dec 20, 2006

William Klostermeyer

Accepted for the School of Computing:

Signature deleted

12/21/06

Judith L. Solano
Director of the School

Accepted for the College of Computing, Engineering, and Construction:

Signature deleted

12/21/06

Neal S. Coulter
Dean of the College

Accepted for the University:

Signature deleted

2 JANUARY 2007

David E. W. Fenner
Dean of the Graduate School

ACKNOWLEDGMENT

After being in the workforce for over four years, it took me a great deal of courage to get enrolled in the School of Computing at the University of North Florida. Coming in with a Management Information Systems degree, I had to fulfill some undergraduate prerequisites in order to qualify for the master program in Computer Science. Along the way, I had the privilege to gain broader and deeper knowledge about computing through various learning channels. Like most sciences, computer science is not just about creating something new. Rather, it is about discovering new approaches and unlocking what we cannot comprehend easily. Most of the time, we have to act within the boundaries of our knowledge. Sometimes, we have to depend on our imaginations to find the answers.

After five years of humbling experiences, I continue to be amazed by the vast amount of intellect out there. This work is undoubtedly one of the most challenging and fulfilling endeavors I have ever waged academically. The subject of this thesis was not something I had in mind when I set out to pursue the thesis option more than a year ago. It was chosen mainly because we decided to explore an unfamiliar territory. For that, I am indebted to my thesis adviser Professor Yap Siong Chua for believing in me and for encouraging me to set challenging goals.

Throughout the journey, however, I discovered more than what I had anticipated. I learned to appreciate the virtue of so many selfless computer scientists around the world

who dedicate their lives to the field. The outcomes of their hard work are often taken for granted as a mere convenience in everyday life. I am grateful to these remarkable individuals who so generously share their knowledge over various publications and web sites. They have made the completion of this work possible.

I appreciate my thesis committee members, Professor Yap Siong Chua, Professor Roger Eggen and Professor William Klostermeyer, who reviewed my paper and provided feedback throughout this period of time. I am especially thankful for Professor Chua's tireless pursuit for perfection, his constructive criticism, and his unconditional guidance. I am grateful to the Director of the School, Professor Judith Solano, and the Advising Secretary, Pat Nelson, for editing this paper and ensuring that it conforms to the standard.

Indisputably, the completion of my thesis would not be possible without the full support from my managers at Merrill Lynch: Michelle Coffey and Ricky Bracken. They have my gratitude for giving me the flexibility to act on my dream amidst our overwhelming workload.

Most importantly, I can never repay my parents and my family for making this a possibility. I am also grateful for my elder brother Leng Shyang, who selflessly bought our very first computer with his hard-earned summer savings eleven years ago. Finally, this achievement would not be meaningful without the support of my loving wife Mandy. Her unwavering and unconditional support shows through in her caring, cooking, housekeeping, and not having cable or satellite TV for the past five years. This thesis is my dedication to her.

CONTENTS

FIGURES

ABSTRACT

The importance and the contribution of string matching algorithms to the modern society cannot be overstated. From basic search algorithms such as spell checking and data querying, to advanced algorithms such as DNA sequencing, trend analysis and signal processing, string matching algorithms form the foundation of many aspects in computing that have been pivotal in technological advancement.

In general, string matching algorithms can be divided into the categories of exact string matching and approximate string matching. We study each area and examine some of the well known algorithms. We probe into one of the most intriguing data structure in string algorithms, the suffix tree. The lowest common ancestor extension of the suffix tree is the key to many advanced string matching algorithms. With these tools, we are able to solve string problems that were, until recently, thought intractable by many. Another interesting and relatively new data structure in string algorithms is the suffix array, which has significant breakthroughs in its linear time construction in recent years.

Primarily, this thesis focuses on approximate string matching using dynamic programming and hybrid dynamic programming with suffix tree. We study both approaches in detail and see how the merger of exact string matching and approximate string matching algorithms can yield synergistic results in our experiments.

Chapter 1

INTRODUCTION

1.1 Background and Motivation

String comparison has an essential role in many areas of computing. Programs rarely complete tasks without performing some types of string manipulation or comparison. While simple programs can rely on basic *if* and *switch* statements to validate user input and incoming data for legitimacy, programs with higher complexity often need more advanced string matching techniques to get the jobs done.

1.1.1 String Comparison

Data are commonly presented in strings of alphanumeric characters in the form of human-readable characters, binary data, and encoded data. When character strings are mentioned, we often think of them as lines of English characters that we humans are most familiar with. However, these English characters we see on computer screens are merely a presentation of the underlying data. In fact, character data type is considered numeric in most programming languages such as C and JAVA. For example, in the ANSI character set, 'a', 'b', and 'c' are represented by the values 97, 98, and 99 respectively. While human languages can be presented with the ANSI character set and the international Unicode character set, binary data such as images and graphics use numbers to indicate the color or shade of a pixel. For instance, if an image format uses 8 bits to represent a pixel, each pixel can have up to $2^8 = 256$ colors or shades. Often, data need to be encoded for transmission or display purposes. For example, a JPG image can be

encoded into a base-64 text and be viewed as a string of printable ASCII characters. Whether strings of data are in numeric format or are encoded into human-readable format, the underlying data can be treated the same way for the purpose of comparison. We are only interested in the syntactic ordering of the string, not its semantic significance [Stephen94]. In molecular biology, complex structural information about DNA and protein are encoded as strings that consist of character G's, A's, T's and C's. The decoupling of semantic information from data allows computer scientists to focus on improving string manipulating algorithms and not be concerned with their biological meaning.

1.1.2 String Matching Categories

We can broadly divide string matching algorithms into two categories, exact string matching and approximate string matching. The need for exact string matching is apparent in our daily lives. For example, a university registrar's officer needs to find a student's records based on his student ID; a store supply manager needs to locate equipment whose part number is XWJ0001. Exact string matching algorithms have been researched and studied extensively in the past decades. They provide the foundation for the study of more advanced approximate string matching algorithms, as we will see in this paper.

The need for approximate string matching is not immediately obvious. In general, approximate string matching is about matching strings with an allotted margin of error. It enables us to do things that exact string matching cannot accomplish alone. Often, data becomes erroneous or corrupted due to human error or poor quality in the network

transmission. Sometimes, data simply change over time. For instance, a merger between two companies can render an old company name invalid. A language can evolve so much over the centuries that some words become obsolete or inappropriate. As a result of these errors and changes over time, important data are lost simply because we are unable to retrieve them.

In a world enriched by a wide variety of cultures, regional and local uniqueness can and often lead to undesired consequences. For instance, searching for the word "color" might not return any result in a piece of British literature, where the word is spelled "colour". Dates, currency symbols, and measurement units are also among examples of differences between countries and continents. Our lack of knowledge and understanding of other cultures could sometimes have serious repercussions. For example, misspelling or mispronouncing a foreign name could inadvertently allow a terrorist to elude a security check point. As the world becomes more connected, these differences have become more relevant than ever. Approximate string matching techniques can be used to resolve such problems by allowing for a margin of error in the matching process.

1.2 Application Areas

In recent decades, several phenomena have propelled the growth of data in all facets of our lives. They include the advent of the Internet, the ease and availability of personal computing, and the advancement of network communication. The use of string matching techniques is central in many areas of our daily lives and societies. We will examine a few of them.

1.2.1 Personal Computing

The first IBM PC was introduced in 1981. The PC ran on an Intel 8088 CPU at 4.77 MHz and had 16KB of memory. Currently (year 2006), a mid-range PC can easily be accompanied by a 2.4GHz CPU and 1 GB of super fast memory, running on an advanced operating system. What really contributes to the popularity of personal computing is, however, its availability and accessibility. The first IBM PC was sold for $1,565, which is tantamount to roughly $4,000 today, whereas today's mid-range PCs are priced as low as $400-$800. In today's society, it is not unusual for a home user to have hundreds of gigabytes of storage space to store and to backup a myriad of media contents, such as MP3 songs, movie clips, and family photo collections. At the time of this writing, various software giants, such as Microsoft, Google, and Linux, are competing to come out with the best desktop search engine to help home users organize and locate their information amidst a multitude of personal data. More often than we realize, personal computing depends on advanced string algorithms to perform tasks as common as spell-checking and correction, grammar usage checking, file comparison, virus detection, voice recognition, and web searches.

1.2.2 Corporate Electronic Records

In recent years, companies have raced to digitize paper records in hope of reducing litigation costs and penalties that amount to billions of dollars. Digitizing paper records also help companies achieve better recovery time in the wake of unequivocal terror threats and natural disasters. Everyday, millions of pages of documents are being converted into imaging data that measure in terabytes (one thousand gigabytes) and

petabytes (one million gigabytes). Consequently, we observe an unprecedented need for the capability to search for both the meta-data and the content in heterogeneous repositories spanning from email systems, imaging systems and file systems, to proprietary databases.

1.2.3 Signal Processing

Signal processing is also a broad topic that touches many aspects of our lives. For example, we have PDAs equipped with software that recognizes hand-writing, voice-recognition software providing much needed help to the physically challenged, and retinal scanning procedures capable of identifying a person accurately, to name a few. However, handwriting is inconsistent by nature, a voice might be altered by a cold, and the pupil in the eye could contract or dilate in response to surrounding lighting. In order for these recognition technologies to work, it is necessary that not only patterns are recognized but that there is an allowance for a certain degree of difference, as long as precision and security are not compromised.

1.2.4 Network Communication

In the past few years, spectacular leaps have also taken place in the areas of telecommunication, wireless technology, and computer networking. These advancements have promoted the growth of data transmission over a multitude of media at varying scale. With more and more data being transmitted across wires and air, the need for reliable communication is inevitably greater. Error correction algorithms, such as the Hamming distance, play an important role in identifying possible errors and correcting them to avoid costly retransmission. In the wake of a series of malware attacks, string

matching techniques are also pivotal in recognizing patterns of potential security breach and virus spread.

1.2.5 Computational Biology

Computational biology is one of the oldest areas which gave rise to some advanced string algorithms. Biologists encode DNA with a chain of nucleotides that contains the genetic information of the living being [Gusfield97]. There are four nucleotides, represented by the character A (Adenine), T (Thymine), C (Cytosine) and G (Guanine). Known genomic sequences are stored in specialized databases, such as BLAST and FASTA, so newly sequenced DNA can be compared or verified against these existing samples. Since the new sequence and the old sequence vary to a certain degree, approximate string matching is used to carry out such a comparison. While an English word is normally less than twenty characters long, the DNA sequence of a simple bacterium could easily contain millions of characters. Therefore, efficient search times and space utilization are keys to practicality in such applications. Recent breakthroughs in genome research and computational biology have rekindled the urge for faster and more efficient string algorithms.

1.3 On-line Searching versus Indexed Searching

Overall, string matching algorithms can be classified into on-line searching and index searching [Navarro98a]. On-line searching is useful for situations where neither time nor space is available. Moreover, the text and the pattern may not be known in advance. For instance, plagiarism detection could involve comparing large files at random. Similarly, identifying homology in DNA sequences requires searching for the longest common

subsequences in two DNA strings. On the other hand, it makes sense to preprocess a body of text to facilitate subsequent searches. Index searching is ideal for this situation. It is comprised of three steps. The first step is to preprocess the pattern or the text body. This includes persisting index information of the text for later use. The second step involves executing the search. The last step is the verification process where we locate the occurrences from our search result.

Naturally, index searching is concerned with index construction and index storage. Index construction is usually not an issue, if the index is built during off hours. Depending on the data structures used, the space required for indexes could grow rather quickly, ranging anywhere between 40% to many times the size of the text. However, the preprocessing overhead is often offset by its superior search time, although not necessarily. Both web page cataloging (e.g., search engine) and DNA sequence mapping are good examples of index searching applications.

In addition, index searching can be divided into two classes: word-retrieving index and sequence-retrieving index [Navarro00]. Word-retrieving index aims at applications that involve natural languages such as English. It is well researched due to the need for effective word processing and document information retrieval. On the other hand, sequence-retrieving index is the optimal choice when the body of text does not lend itself well to the concept of natural languages or words. Examples of such data include protein sequences, binary or media files, and encoded file content.

1.4 How This Paper is Organized

In chapter two and chapter three, we look at algorithms for exact string matching and approximate string matching. In addition, we introduce the concept of edit distance and dynamic programming in chapter three. We dedicate chapter four to the suffix tree data structure. We take an in depth look at suffix tree construction and the longest common ancestor extension. Chapter five focuses primarily on solving the k-difference problem with hybrid dynamic programming with suffix tree. In chapter six, we examine briefly another advanced data structure called suffix array. In chapter seven, we conduct some experiments to measure the performance of dynamic programming and hybrid dynamic programming with a suffix tree. In particular, we pay close attention to how the search time of each approach is influenced by changing text length, pattern length, and number of errors allowed. Finally, in chapter eight, we will present our conclusion.

Chapter 2

EXACT STRING MATCHING

Many efficient exact string matching algorithms have been devised in the last few decades. In this chapter, we examine four of them, from the intuitive naive approach to the advanced Knuth-Morris-Pratt and Boyer-Moore algorithms.

2.1 Problem Definition

We formalize the exact string matching problem as follows.

$\sum$ = Alphabet of a finite set of symbols. $|\sum| = \sigma$

T = A string of text derived from $\sum$. $|T| = n$

P = A string of pattern derived from $\sum$. $|P| = m$

The goal of our string algorithms is to search for occurrences of P in T. We assume that $m <= n$.

2.2 Exact String Matching Algorithms

2.2.1 The Naive Approach

The simplest of all exact string approaches is the naive approach, which scans the text $T[1..n]$ from left to right, comparing each character in the pattern $P[1..m]$ to the text during each iteration. If a mismatch is found, it moves to the next text position and starts to compare the pattern with $T[2..m + 1]$. This approach is intuitive but has the disadvantage of an $O((n - m + 1) m)$ run-time. As m approaches $n/2$, the worst case scenario of the naive approach reaches $O(n^2)$ quadratic time.

2.2.2 The Automaton Approach

An automaton is an abstract machine that maintains its state based on input derived from Σ. Given an input symbol, an automaton uses a *transition table* to determine its next state. The transition table consists of rows of states and columns of symbols in Σ. When applied in exact string matching algorithms, characters in P represent the states in the transition table while characters in Σ represent the input symbols. When the next input character read from T matches the next character in P, the automaton advances to the next state. Otherwise, it reverts to a previous state while preserving as many matching characters as possible. This avoids having to match the next input character from the beginning of P every time there is a mismatch. The automaton approach is easy to understand and its search time has a tight bound of $O(n)$. Unfortunately, while the transition table is the strength of an automaton, it has an $O(m\sigma)$ limitation on its construction time and space requirement.

2.2.3 The Knuth-Morris-Pratt Algorithm

In the year 1977, a clever observation was made by Knuth, Morris, and Pratt [Knuth77]. The Knuth-Morris-Pratt algorithm (KMP) is similar to the automaton approach, but it is able to mitigate the restriction imposed by the transition table. Instead of a transition table, it uses a prefix function, $p()$, which is constructed on-the-fly and is independent of n. The prefix function $p()$ contains knowledge about how well the pattern matches itself. Therefore, when a mismatch occurs, it knows to slide down the text as far as possible without missing any potential match. In short, $p(q)$ is the length of the longest prefix of P

that is a suffix of $P[1..q]$. The construction of $p()$ takes $O(m)$ time by matching the pattern to itself. KMP scans from left to right with a search time of $O(n)$.

2.2.4 The Boyer-Moore Algorithm

In the same year, Boyer and Moore introduced an algorithm with sub-linear time, $o(n)$ [Boyer77]. The Boyer-Moore (BM) approach utilizes a jump function that contains information about the pattern and allows the algorithm to jump ahead if a mismatch occurs. The strength of BM comes from the fact, contrary to common sense, it scans for the pattern from right to left. As a consequence, it does not scan every character unnecessarily, if the ends of the pattern $P[1..m]$ and text $T[i..i+m]$ do not match. This allows the algorithm to skip ahead faster and achieve greater efficiency as pattern length, m, grows. BM has a worst case scenario of $O(mn)$ when T consists of one repeating character and P consists of the same character except for the beginning character. For example, T=*aaaaaaaaaaaaaaaaaaaaaaaaaaaaa* and P=*baaaaaaaaaaaaaaaaaaaa*. The best case BM has a run-time of $O(n / m)$, when the last character of the pattern, $P[m]$, does not match any character in $T[m..n]$.

Chapter 3

APPROXIMATE STRING MATCHING

3.1 The Basic Concepts

Sometimes we need to locate data without exact information about the subject. This could be due to a variety of reasons such as changing personnel or corrupted data. These are situations where approximate string matching techniques can become valuable. Approximate string matching is also known as inexact string matching or string matching allowing k errors. By allowing a predetermined margin of error in our search result, we are able to retrieve more records of relevance. For instance, a search for the word *river* in a literature allowing for one error may return *diver, liver, rive, river, rivet,* or *rover*. However, an error margin that is too large could lead to a search result containing an overwhelming number of irrelevant data. The trade off between a relevant search result and an erroneous search result is described by Hall and Dowling [Hall80] as *precision* and *recall*. Precision refers to the proportion of retrieved data that are relevant; whereas recall refers to the proportion of relevant data actually retrieved. Precision goes up as recall comes down, and vice versa.

3.1.1 Edit Distance

The concept of distance is commonly used to measure the similarity (or differences) between two strings x and y. The more similar the two strings are, the smaller the distance is. Basically, the distance between x and y is a measure of the minimum number of operations needed to transform x into y [Navarro01]. Four types of string

manipulation are used to measure the distance of two strings, including insertion, deletion, substitution (replacement) and transposition. Depending on application requirements, the cost of each operation could vary. For instance, the transposition operation could carry a higher cost for a virus detection program than it would for a spell-checker.

There are several kinds of distance functions. The most common ones include Hamming distance, edit distance, and longest common subsequence distance [Navarro01]. Hamming distance allows for only substitution at one cost unit per operation. Edit distance, also known as Levenshtein distance, allows for insertions, deletions, and substitutions, at one cost unit each. The longest common subsequence (*lcs*) allows for only insertions and deletions, with the distance being the number of unpaired characters. Note that subsequence and substring are different. For example, *xyz* is considered to be a subsequence but not a substring of *xaaaaayaaaaaazaaaaa*.

Exact string matching techniques such as the naive matching algorithm can be easily adapted to perform simple approximate string matching with an average run-time of $O(mn)$ and a worse case of $O(n^2)$ when m approaches $n / 2$. However, more efficient algorithms have been devised. In the next section, we look at various approximate string matching approaches.

3.1.2 Problem Definition

The approximate string matching problem can be defined as:

> Given T, P, k, and $d()$, find the set of all substrings in T such that $d(P, T[i..j]) \leq k$, *where*
>
> k = the maximum number of errors allowed
>
> α = the error level = k / m
>
> $d()$ = the distance function

The distance $d(x, y)$ is the minimum cost of transforming string x into string y. Given $0 < k < m < n$, we can conclude that $0 < \alpha < 1$. An interesting measurement is the maximum error level, denoted as α^*. Without α^*, an error level that is too high would lead to almost all text positions matching the pattern. Therefore, we are interested in allowing errors only up to α^*. Sankoff and Mainville conjectured that $\alpha^* = 1 - 1 / \sqrt{\sigma}$ for optimal approximate string matching, according to [Navarro01].

3.2 Approximate String Matching Algorithms

3.2.1 Dynamic Programming

In 1980, Seller incorporated edit distance in dynamic programming for the first time [Seller80]. Dynamic programming is the oldest non-brute force approach of approximate string matching. In its purest form, dynamic programming is less competitive than some of the other approaches, but it has demonstrated tremendous flexibility and adaptability. In fact, many algorithms take advantage of this fact and combine dynamic programming with other advanced techniques to achieve better results. Dynamic programming can

attain a worst-case of O(kn) and an average case of O($kn / \sqrt{\sigma}$). We examine a hybrid dynamic programming approach in chapter 5.

The principal idea of dynamic programming is to break the problem down into its basic blocks, resolve the sub problems non-recursively, and record the results in a table [Weiss02]. In this case, we use a matrix $C_{0..m,0..n}$ to keep track of results obtained by solving sub problems, where $C_{i,j}$ represents the minimum edit distance to convert $P[i..m]$ into suffix $T[j..n]$. Therefore, at text position j where $C_{m,j} \leq k$, we can find the end of P in T with, at most, k errors [Navarro00]. We initialize $C_{i,0} = i$ and $C_{0,j} = 0$ where i represents the current position in P, and j represents the current position in T. We then construct the remainder of matrix C by filling the content with calculated values of k using

$$C_{i,j} = \begin{cases} \text{if } P_i = T_j \text{ then } C_{i-1,j-1} \\ \text{else } 1 + \min(C_{i-1,j}, C_{i-1,j-1}, C_{i,j-1}) \end{cases}$$

After the matrix is constructed, we can scan for $C_{m,n}$ to see if its value is less than k for a match. Figure 1 shows the dynamic programming matrix for pattern *mccain* in text *mccayne* using edit distance. Bold entries show matching text position j where $k <= 2$ for pattern length $i = 6$. Matches with 2 or fewer errors were found at position 4, 5, 6, and 7.

		n=0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
		""	m	c	c	a	y	n	e
m=0	""	00	00	00	00	00	00	00	00
1	m	01	00	01	01	01	01	01	01
2	c	02	01	00	01	02	02	02	02
3	c	03	02	01	00	01	02	03	03
4	a	04	03	02	01	00	01	02	03
5	i	05	04	03	02	01	01	02	03
6	n	06	05	04	03	**02**	**02**	**01**	**02**

Figure 1: The Dynamic Programming Matrix

Since the *min()* function requires just the last row to perform the comparisons, we only need to keep the previous row for all purposes. This drastically reduces the space requirement from O(mn) to O(n). The matrix, or part of the matrix, can be reconstructed by using a technique called trace back. It proceeds backward from $C_{m,n}$ to $C_{0,0}$.

In 1983, Ukkonen cleverly observed the diagonal-wise monotonicity attribute of the dynamic programming algorithm, which is that adjacent cells on any downward left-to-right diagonal of the matrix may increase by one, and the values never decrease in that direction [Ukkonen83]. He proposed a cut-off heuristic that stops the calculation as soon as the value of $k + 1$ is obtained. This significantly improved the run-time from O(mn) to O(kn). The diagonal-wise monotonicity implies that only the first $k + 1$ transitions on each diagonal of the matrix need to be found. This observation has provided an important framework for a variety of extensions.

3.2.2 Automata

Automata-based algorithms have also been studied at length throughout the years [Ukkonen93b]. Although the automata approach has a remarkable theoretical worst case

run-time of $O(n)$, its practicality is largely limited by its immense space and time requirements [Navarro01]. In practice, automata are usually implemented with a transition table, whose primary task is to keep track of state information. A transition table is two dimensional, represented by k rows and m columns. The rows represent the number of errors found and the columns represent matches of P found in T. To ease a transition table's space requirements, Ukkonen suggested adapting the concept of cut-off heuristic from dynamic programming. The heuristic essentially specifies that the states of columns larger than $k + 1$ can be replaced by a column of $k + 1$. He conjectured that only up to $3k / 2$ columns need to be computed.

3.2.3 Bit-parallelism

Baeza-Yates and Gonnet introduced a new concept called bit-parallelism [Baeza92]. In a new exact string matching algorithm, they took advantage of the intrinsic nature of bit-parallelism in computers by using the *shift-or* operation and storing state information in a computer *word*, which is typically 32 or 64 bits. In their paper, Baeza-Yates and Gonnet demonstrated a fast approximate string matching method using their *shift-add* algorithm. By taking advantage of the parallelism of bitwise operations, they were able to reduce the number of operations of a chosen algorithm by a factor of the size of a *word*. The *shift-add* algorithm forms the basis for the famous *agrep* program on various operating systems such as UNIX, Linux, OS/2 and Windows.

Bit-parallelism is not just an approach in itself. It can be implemented by extending existing approaches such as dynamic programming and automata. Since it is largely based on a computer *word* size, w, the approach works best when pattern length is less

than w. For patterns with length greater than w, m / w *words* can be combined to simulate a large word at the cost of some overhead.

3.2.4 Filtering

Filtering is a relatively new approach that emerged around early 1990s. The technique allows for a large chunk of text to be abandoned quickly, using exact string matching techniques such as Boyer-Moore. The remaining text represents areas with potential hits for the pattern, allowing up to k errors. The performance gain is possible because Boyer-Moore-like algorithms have sub-linear expected time. Although the filtering technique can quickly disqualify a large area of text, it is unable to pinpoint the exact positions of matching occurrences. Therefore, a verification process must be employed to determine the matching locations once the filtering phase is completed. This verification process can be coupled with other techniques we have seen thus far, such as dynamic programming or automaton-based techniques. Since the verification area is usually small and negligible, the exact choice of verification techniques is rarely of concern.

In filtering, there are two critical measurements: *filtering speed* and *filtration efficiency*. Filtering speed refers to the run-time of a filtering algorithm, while filtration efficiency refers to the accuracy of its filtration results.

a = the number of actual matches

p = the number of potential matches reported by the filtering algorithm

Filtration efficiency = a / b

Due to high overhead, the use of filtering techniques is justifiable only for a moderately or extremely long pattern. According to Navarro in [Navarro01], filtering techniques do not outperform pure dynamic programming and automata for $m < 100$. In practice, given its complexity and overhead, the value of m needs to be much higher in order to reap benefit from the filtering approach.

Another interesting aspect about filtering is it is highly sensitive to changes in error level, α. As α increases, filtration efficiency decreases and filtering speed reduces sharply. As previously mentioned, the rule of thumb for maximum filtering error level is:

$$\alpha = 1 - 1 / \sqrt{\sigma}$$

In practice however, α must be much lower than that to have meaningful filtering result.

3.2.4.1 Filtering History

In 1991, Jokinen, Tarhio, and Ukkonen observed a simple fact. In a block of text of length m, allowing k errors, there must be at least $m - k$ matches, regardless of the order of characters [Jokinen96]. They devised an algorithm that slides a window of length m over the text and keeps count of characters that match the pattern. If the count is greater than or equal to $m - k$, then the text area is verified with a conventional procedure such as dynamic programming. The algorithm only works well for a low error level [Navarro01].

In 1992, Wu and Manber presented a simple concept that states if a pattern is cut into $k+1$ pieces, at least one of them must be an exact match [Wu92]. The proof is if all $k+1$ pieces had an error, it would require $k+1$ operations to transform string x into string y.

In 1998, Navarro and Baeza-Yates devised a new technique called hierarchical verification [Navarro98b]. The pattern continues to split in halves until it is small enough to be implemented with a non-deterministic finite automaton (*NFA*). After the splits, the smallest piece is verified against P. If a match is found, or if the error count is less than k, the larger piece immediately above it is checked and the count of mismatches is recorded. As soon as the count of mismatches is greater than k, the surrounding text can be abandoned. This reduces unnecessary verification because smaller chunks of strings are used to determine the validity of a larger text area.

In 1990, Chang and Lawler introduced two new filtering techniques: LET and SET [Chang94]. LET stands for *Linear Expected Time* and SET stands for *Sub-linear Expected Time*. The LET technique traverses the text linearly and keeps track of all matching substrings. Then, it concatenates the k longest substrings to obtain the total length, l. If l is less than $m - k$, then the text area does not match the pattern and can be abandoned. Otherwise, the text area can be examined using dynamic programming. LET runs in $O(n)$ time, while the dynamic programming verification can take up to $O(kn)$ time.

SET is similar to LET except the text is split into fixed blocks of size $(m - k) / 2$. The check for k contiguous strokes starts only at block boundaries. Since the shortest match

in an area is of length $m - k$, at least one of these two blocks is always contained in a match. Otherwise, a block is discarded because no matching occurrence can contain it. The algorithm is sub-linear because a block is discarded after $O(k \log_\sigma m)$ comparisons on average [Navarro01].

In 1992, Ukkonen presented the q-gram approach, an idea similar to that of Chang and Lawler's LET approach for on-line searching [Ukkonen92]. Ukkonen divided the pattern P into blocks of substring called q-grams of fixed-length q. Thus, a pattern of length m has $m - q + 1$ overlapping q-grams. The technique keeps count on the matching q-grams. $m - q + 1$ grams must appear in any occurrence to have a potential match. Similarly, a verification procedure needs to be run once we filter out text areas in which it is impossible to have matches allowing up to k errors. It is often referred to as n-gram filtering.

Chapter 4

SUFFIX TREES

4.1 Background

The suffix tree of a string S is a tree-like data structure that represents all suffices of S. The suffix tree for $S[1..n]$ contains n leaves where the paths from the root to each leaf represent all substrings of $S[1..n]$, $S[2..n]$, $S[3..n]$, $S[4..n]$, ..., $S[n]$. This important data structure plays a pivotal role in many advanced string algorithms and offers solutions to many problems that were once thought intractable in linear time. As an example, a string *ababb$* has the suffices of *ababb$*, *babb$*, *abb$*, *bb$*, *b$*, and *$*. The suffix tree for *ababb$* is shown in Figure 2.

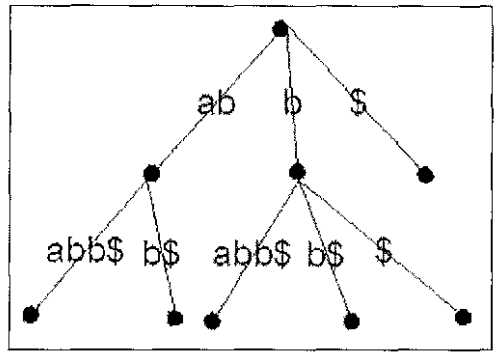

Figure 2: The Suffix Tree for *ababb$*

The suffix tree of string S can be constructed in $O(n)$ time, where n is the length of S. Thereafter, any pattern P of length m can be located in $O(m)$ time, at a space requirement of $O(n\sigma)$ where $\sigma = |\Sigma|$, by traversing the tree from the root node. When we reach the end of the pattern, a match is found. If we reach a leaf node before the end of the pattern,

then no match is found. The O(m) search time is a remarkable achievement not possible with Knuth-Morris-Pratt or Boyer-Moore algorithms. The Knuth-Morris-Pratt and Boyer-Moore algorithms preprocess the patterns in only O(m) time, but subsequent searches of the patterns necessitate a scanning of the text requiring O(n) time. The suffix tree provides a superior search time for large text. When equipped with the lowest common ancestor (*lca*), a suffix tree can be used to determine the longest common prefix shared by two suffices in constant time.

4.2 History

In 1973, Weiner [Weiner73] introduced a linear time suffix tree construction algorithm that was dubbed 'algorithm of the year 1973' by D. Knuth. The algorithm adds subsequently longer suffices to the tree. In 1976, E. McCreight devised a more efficient linear time construction by adding subsequently shorter suffices to the tree [McCreight76]. More importantly, McCreight introduced the concept of a suffix link that has become rudimentary in suffix tree algorithm development. Although McCreight's algorithm is efficient, it suffers from the same limitation faced by Weiner's algorithm, which is the entire string needs to be read before the suffix tree construction can begin. Thus, both algorithms are not suitable for on-line applications. In 1992, E. Ukkonen presented an algorithm that adds subsequently longer prefixes of the string [Ukkonen93a]. The algorithm appends new characters to the suffix tree as they are being read. This makes it the first on-line linear time suffix tree construction algorithm.

Perhaps the most exciting fact about suffix trees is they serve as the bridge between exact string matching and approximate string matching [Gusfield97, page 89]. We will study this in depth in section 5, when we probe into a hybrid dynamic programming algorithm.

4.3 A Suffix Trie and Suffix Tree

We now look at a more fundamental structure called *suffix trie*. Trie, from the word re*trie*ve, is a tree-like data structure that uses edges to represent characters of string suffices. Every path from the root to a leaf represents a suffix of the string. Every suffix of the string must be present in the trie. A trie can be and is often used as an automaton for string pattern matching [Stephen94]. The trie for a sample string *caccao$* is shown in Figure 3. The shortcoming of this simplistic data structure is it stores every character of the string suffices. A string of length n has n suffices with a total length of $1 + 2 + 3 + 4 + \ldots + n = n\,(n + 1) / 2 = O(n^2)$ characters.

Suffix trees are also known as compressed suffix tries or Patricia (Practical Algorithm to Retrieve Information Coded in Alphanumeric) trees, because they are more space-efficient than suffix tries. The most noticeable difference between suffix trie and suffix tree is the latter compresses all its unary paths (nodes with only one child) into *transitions*, which are also known as *edges*. The suffix tree for string *caccao$* is show in Figure 4. In this case, we see a reduction of more than fifty percent in the number of nodes, from 25 to 11.

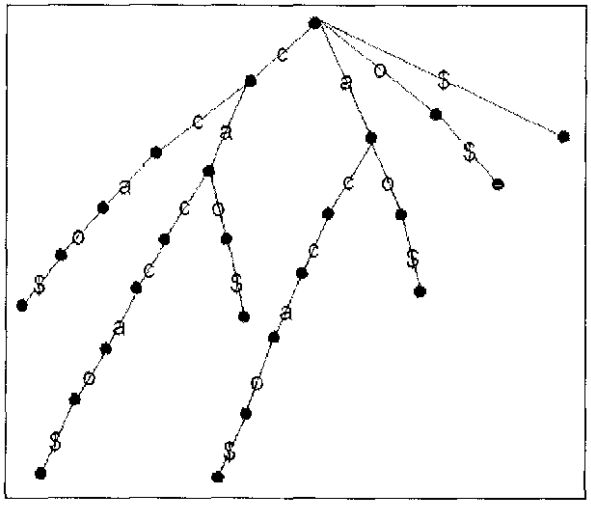

Figure 3: The Trie for *caccao$*

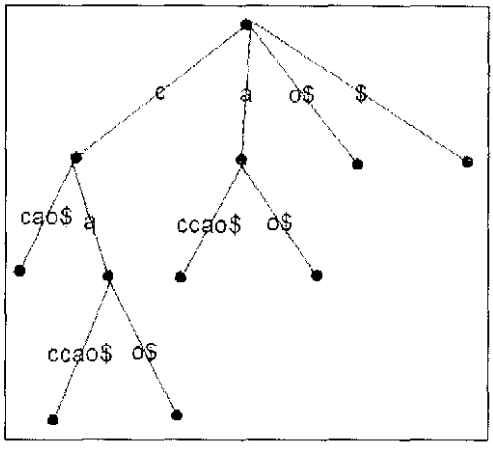

Figure 4: The Suffix Tree for *caccao$*

In practice, the space requirement for a suffix tree would grow quadratically, if each transition stores the substring it represents. A much more efficient approach is to have the transition referencing the starting and ending positions of the substring in S. Since each transition only requires two integers to store the starting and ending positions, this technique reduces the space requirement from $\Omega(n^2)$ to $O(n)$. When traversing down a suffix tree from its root, the search for a substring could end up in the middle of a transition. Take Figure 4, for instance. To search for the pattern *acc*, we slide down

transition *a* from the root and end in the middle of transition *ccao$*. We are said to have reached an implicit state. In other words, transitions that represent more than one character inherently contain implicit states. On the other hand, if our search ends at a node, we are said to have reached an explicit state. For example, searching for *a*, *c*, *accao$*, or *ao$* in the above suffix tree would end at explicit states.

There are two types of nodes in a suffix tree – internal nodes and leaf nodes. An internal node represents an explicit state, where two or more transitions branch out. Therefore, each internal node must have at least two child nodes. Leaf nodes do not have children. The path from the root to any leaf node represents a suffix of the string *S* of the suffix tree. Therefore, the suffix tree for a string of length *n* has *n* leaf nodes, if it is an explicit suffix tree. The tree also has a maximum of $2n$ - 1 nodes.

In 1976, McCreight [McCreight76] introduced the use of suffix links. Every internal node, with the exception of the root node, must have a suffix link. The suffix link serves as a shortcut to jump from one branch of the tree to another, during tree traversal. The concept is best illustrated with an example. In Figure 5, prefix *ca* points to its next longest prefix *a*. Prefixes *a* and *c* point to their next longest prefix, an empty string represented by the root node. The significance of the suffix links is they speed up the traversal of the suffix tree and make the linear time construction possible. We discuss suffix link implementation and its role in suffix tree construction in section 4.4.2.

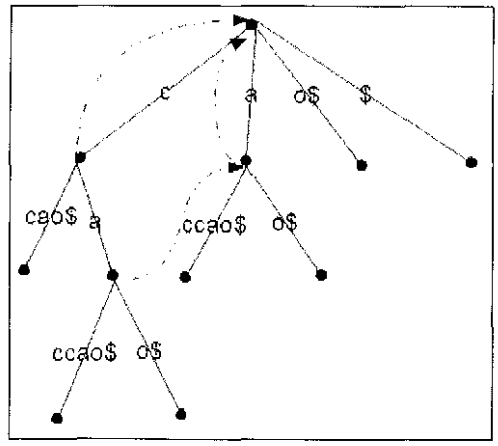

Figure 5: The Suffix Tree for *caccao$* with Suffix Links

In his paper [Ukkonen93a], Ukkonen presents the implicit or explicit states with *reference pairs* $\{s, (k, p)\}$. State s is some explicit state prior to state r, (k, p) represents the substring that spells out the transition from s to r, k is the index to the first character, and p is the index to the last character of the transitional substring. In other words, state s and substring (k, p) transition to state r. For example, if state s is *ca* and state r is *caccao*, the transitional substring *ccao* is presented as (2, 5).

4.4 Suffix Tree Construction

Here we present a suffix tree construction algorithm based on Ukkonen's algorithm [Ukkonen93a]. Our initial implementation in JAVA is based on the code sample by [Nelson96]. We chose to focus on Ukkonen's approach for its on-line attribute as well as its linear time construction efficiency. According to Gusfield [Gusfield97], Ukkonen's approach is also more concise and easier to understand than the other approaches.

4.4.1 Structures

Since JAVA is an object-oriented language, the main components of our suffix tree are defined in classes. The four primary classes are *Nodes*, *Transition*, *ReferencePair*, and *SuffixTree2*.

The Node class defines the nodes in the tree. Each node has a unique identifier and a reference to its parent node. Every internal node of a suffix tree has at least two transitions and the first character of each outgoing transition is guaranteed to be unique for the node. Therefore, each node could have up to σ branches, where σ is the size of the alphabet. It maintains a list of child nodes to which it is a parent. Every internal node has a suffix link that references to the next longest suffix of the string the node represents.

The Transition class defines the transitions in the tree. Each transition has a start node and an end node. It has the starting and ending positions for the substring it represents. Transitions are kept in a hash table at the suffix tree level for efficiency. Each transition in the hash table is uniquely identified by its hash key, which consists of its starting node identifier and its beginning character.

The ReferencePair class, as explained in section 4.3, defines the current state of the suffix tree. It has a state and a (k, p) pair that gives the beginning and ending indices of the substring after the explicit state. When the state is explicit, meaning we are in a transition

that represents only one character, k is set to be greater than p. When the state is implicit, meaning we are in a transition that represents two or more characters, p is greater than k.

The SuffixTree2 class defines the suffix tree data structure. It contains the main string S referenced by the Transition objects, a hash table to keep track of the Transition objects, and an array to keep track of the Node objects. Most importantly, The SuffixTree2 class implements Ukkonen's algorithm to construct a suffix tree on-line in O(n) linear time.

4.4.2 Building a Suffix Tree

Because of its on-line applicability, Ukkonen's suffix tree construction algorithm was a monumental achievement. A suffix tree of a string S is built incrementally as each character of S is read. For example, the suffix tree ST for string abc is built by adding newly read characters to ST one at a time, in the order of a, b, and c. Another way to look at it is that $ST[1..i]$ is built on top of $ST[1..i\text{-}1]$, which in turn is built on top of $ST[1..i\text{-}2]$, and so on.

4.4.2.1 Appending a New Character to the Suffix Tree

When appending the next character to a suffix tree, there are three possible scenarios. In the first scenario, we read in a character c and walk down the tree from the root, trying to update all suffices with character c. We end up at an internal node (explicit state) and find no transition that starts with c. As a result, we create a new transition to branch out of the node. Figure 6a shows the initial case of adding a character c to the root node.

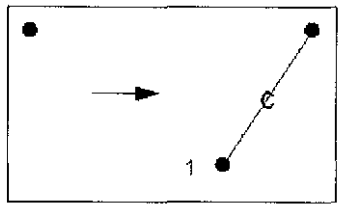

Figure 6a: Adding *c* to the Root Node

Whenever a new transition is added, a new node is created for that transition. Figure 6b shows another example of the first scenario. A new character *b* is read into a tree that ends with *c*, which causes the tree to grow from ...*c* to ...*cb*. We walk down the tree and update the suffices to ...*cb*. In order to present the suffix *cb* of ...*cb* string, we add a new transition for *b* out of the ending node of transition *c*. Similarly, to present the suffix *b* of ...*cb* string, we add another transition for *b* out of the root node.

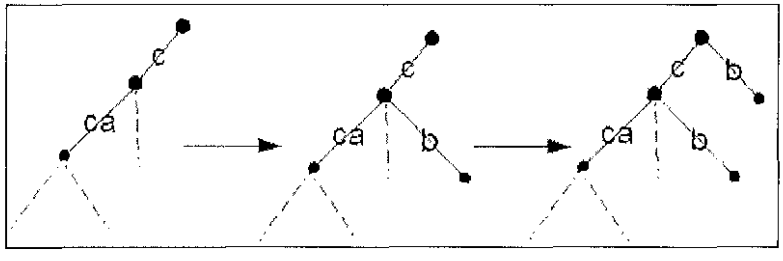

Figure 6b: Adding *b* to a Suffix Tree

In the second scenario, we read in a character *c* for the suffix tree of some string *S*. We traverse down the tree to append the new character to the suffices. At some state *R*, we find the new character c is already defined by an existing state (implicit or explicit). In this case, we do not do anything since a transition already exists for that state. For instance, Figure 7a shows a new character *c* is read into the suffix tree for string ...*ac*...*a*. The resulting string is ...*ac*...*ac*. There is no need for any action since the suffix

ac is already defined by an explicit state. Figure 7b shows the same scenario, except the suffix *ac* is represented by an existing implicit state.

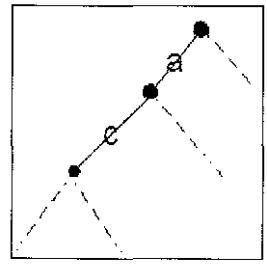

Figure 7a: An Explicit State Already Exists for Suffix *ac*

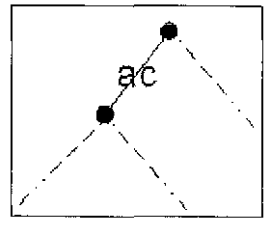

Figure 7b: An Implicit State Already Exists for Suffix *ac*

In the third and last scenario, the traversal reaches an implicit state (in the middle of a transition) and the next character is not found. In this case, we need to first split the transition by creating a new internal node (explicit state), then append a new transition to represent the new character. Figure 8 shows the scenario of adding *cao* to the sub tree.

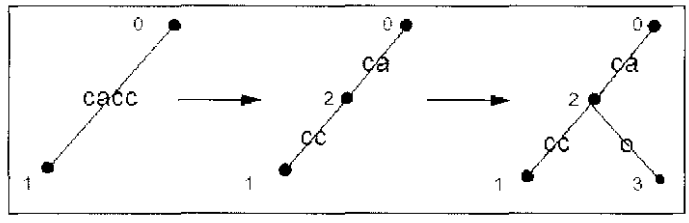

Figure 8: Adding *cao* to The Sub Tree by Splitting and Appending Transitions

4.4.2.2 Once a Leaf Node Always a Leaf Node

If we built the suffix tree for string *aaa* by adding the next character one at a time, it takes three iterations to complete the construction. Figure 9 demonstrates the point. Note that the ending index of the transition is updated each iteration.

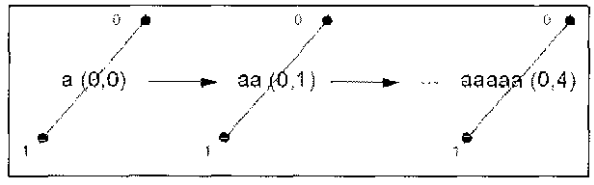

Figure 9: The Position of the Ending Character
is Updated in Each Iteration

Such inefficient updates would be impractical for large suffix trees. In fact, in every iteration, we could potentially update up to i transitions, where i is the position of the next character read. Hence, our construction algorithm would be limited to $O(n^2)$. We can do better by avoiding unnecessary updates to the open transitions (last transition prior to the leaf nodes). We mark the ending character of leaf nodes with ∞ to signify the transition may grow to cover the entire length of the suffix. At the end of the construction, we can easily update the transitions' ending positions to n in $O(n)$ time. Since a leaf node marks the end of the transition, once a node is created as a leaf node, it remains a leaf node. This supports the third scenario of appending new characters to the suffix tree discussed in section 4.4.2.1. Figure 10 illustrates this point.

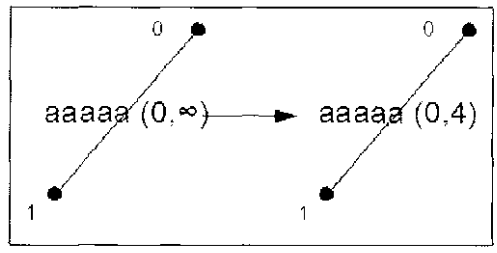

Figure 10: The Ending Index is Updated From ∞ to n afterward

4.4.2.3 Improving Construction Time with Suffix Links

Even with the improvement discussed in section 4.4.2.2, the construction time is still $O(n^2)$. This is because we have to traverse up to the root and down another branch in order to update the next suffix. Along the way, we examine all branching transitions. We could potentially search for up to i ending transitions, where i is the current position read. Dotted lines in Figure 11a illustrate this inefficient traversal. From point 1, we traverse up to the root and then down to point 2.

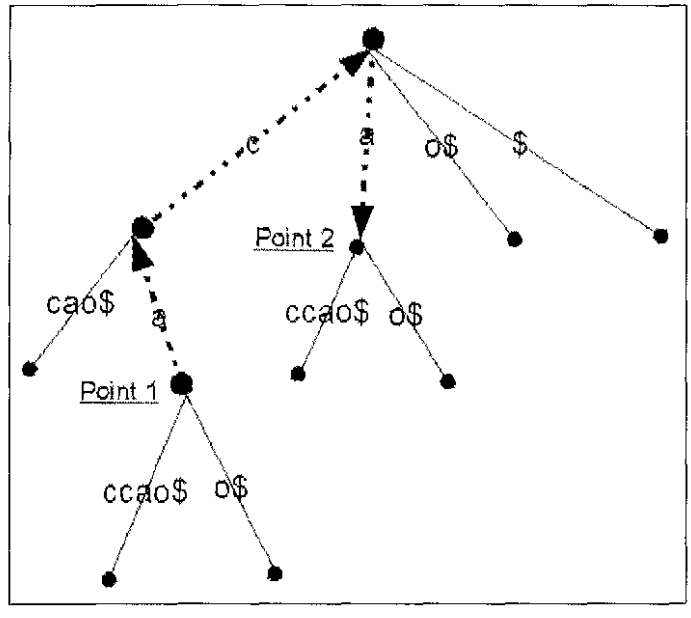

Figure 11a: Traversing Up the Root and Down another Branch

A suffix link allows for a shortcut from point 1 to point 2, as illustrated in Figure 11b.

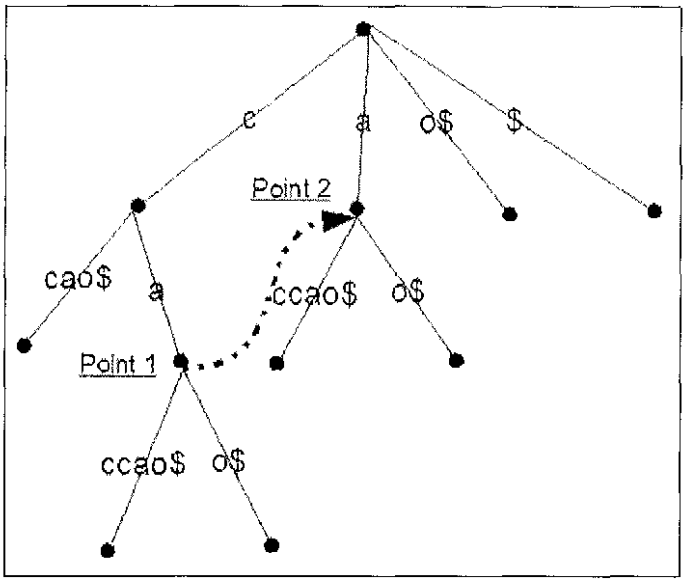

Figure 11b: Traversing to the Next Update Point with a Suffix Link

Each internal node of a suffix tree has a suffix link that references the next update point. Basically, a suffix link points to the next longest suffix of the string represented by the node. For example, if an internal node represents *ba*, its suffix link points to *a*. As internal nodes are created, we set the suffix link of the last update point to reference the current update point. This ensures the suffix links are kept up-to-date in scenario 3 described in section 4.4.2.1. Figure 12 demonstrates this process.

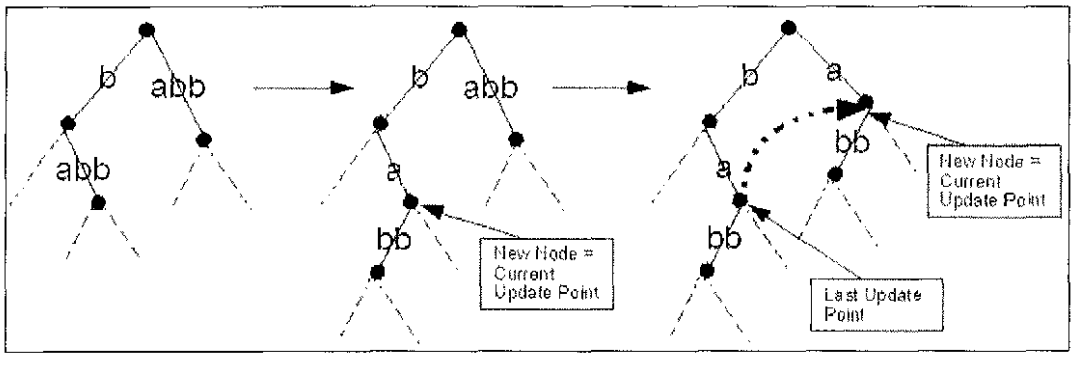

Figure 12: Suffix Link Update

4.4.2.4 The Main Suffix Tree Procedures

Ukkonen's algorithm consists of five main procedures: *update()*, *canonize()*, *testAndSplitI()*, *splitTransition()*, and *addStringToTree()*. We discuss each of them in detail.

4.4.2.4.1 The *splitTransition*() Procedure

The *splitTransition*() procedure splits a transition into two by inserting a new internal node at the position where a new transition will be attached. This handles the third scenario described in section 4.4.2.1. The concept of splitting a transition is simple, but the implementation could be a tricky one, since the old node must be updated and the old transition must be removed from the hash table and then added back to the hash table with a new hash key. The pseudo code for the *splitTransition*() procedure is shown below. From this pseudo code, it is clear that once a leaf node is created, it remains a leaf node.

```
procedure splitTransition(Transaction A, ReferencePair rp) {
      Node 0 (active node) = Internal node that represents rp.state
      Remove node 1 as the child node of node 0
      Remove the old transition A because it will have a new hash key
      Create a new transition B to branch out of node 0
      Add transition B to the hash table
      Create a new node 2 at the end of transition B
      Assign new node 2 as a child of node 0
      Update transition A's first char position
      Update transition A's start node id
      Update transition A's hash key
```

```
        Add transition A back to the hash table
        Update node 1's parent to be the new node 2
        Add node 1 as the new node 2's child
        Return newly created state (node 2)
}
```

4.4.2.4.2 The *testAndSplit*() Procedure

When adding a new character to a suffix, we use the *testAndSplit*() procedure to determine if scenario 1 or scenario 3 applies. The procedure splits a transition, if the conditions of scenario 3 are met.

```
procedure testAndSplit(ReferencePair  rp, char t) {
      // rp represent the substring in the suffix tree we are examining
      endpoint = null   // indicate whether we have reached an end point
      activenode = null // the current active node in
                        //   the suffix tree to work on
      if rp is an implicit state (we are in the middle of transition T)
            if t = rp's character
                  endpoint = true // scenario 1 in section 4.4.2.1
                  activenode = rp's state
            else
                  endpoint = false // scenario 3 in section 4.4.2.1
                  activenode = splitTransition(transition T, rp)
      else if rp is an explicit state (we are at an internal node)
            if transition T is not found
                  endpoint = false // scenario 2 in section 4.4.2.1
                  activenode = rp's state
```

```
            else
                  endpoint = true // scenario 1 in section 4.4.2.1
                  activenode = rp's state
      return (endpoint, activenode)
}
```

4.4.2.4.3 The *canonize*() Procedure

In Figure 13, the string *cacc* is presented by reference pair (2, (1,3)), where 2 is the state and (1,3) is the index pair for the substring *acc* with a zero-based positioning. Given the string *cacc*, the procedure to derive (2, (1,3)) is called *canonization*. Basically, we try to find the deepest internal node (closest to the leaf node) that represents the substring in the suffix tree. This can easily be done by traversing down from the root, character by character, through all the transitions in the path. However, such traversal would greatly impair our performance, because we would examine up to $1 + 2 + 3 + 4 + ... + n = n(n - 1) / 2 = O(n^2)$ characters in n iterations. Instead of examining one character at a time, we slide down the transition by the length of the substring (transition length = ending index - starting index + 1). For example, in Figure 14, to canonize substring *ababbaabc*, instead of examining all nine characters, we slide down the path by subtracting the transition length along the way. As a result, we perform only three comparisons to derive the canonical form of the same substring.

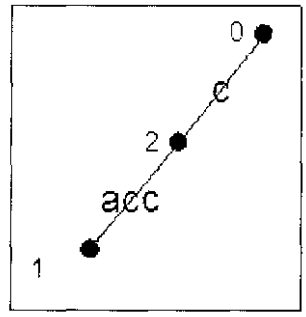

Figure 13: Reference Pair (2, (1,3))

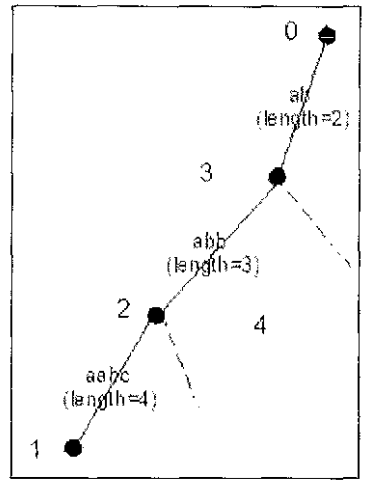

Figure 14: Sliding Down by Transition Length

The *canonize()* procedure follows:

```
procedure canonize(ReferencePair rp) {
      if (rp.k < rp.p) // if we are not at an explicit state
            look for the transition T that represents our state
            while (T's length > rp's span)
                  slide down the length of the transition by updating rp
                  => update rp's k with next transition's first index
                  => update rp's state to T's end node
                  T = the next transition (start from T's end node)
}
```

4.4.2.4.4 The *update*() Procedure

The *update*() procedure essentially wraps up the previous three procedures. The procedure represents what is done when a new character is read and appended to the suffix tree. It outlines what is done when ST(1..*n* - 1) is transformed into ST(1..*n*).

```
update (rp, last_char){
      old_active_node = -1
      (endPoint, activeNode) = testAndSplit(rp, last_char)
      while (not endPoint)
            add a new transition T to activeNode
            add T to the hash table
            add a new end node 3 to T
            add node 3 as a child of activeNode
            if  old_active_node > -1
                  set  old_active_node's suffix link to point to
                  activeNode
             old_active_node = activeNode
            if rp.state is already at root
                  we are done processing this iteration
                  advance rp's k by 1
            else
                  follow the suffix link of activeNode
                  continue processing
                  => rp.state = rp.state's suffix link
            canonize(rp) // canonize the new active point
            (endPoint, activeNode) = testAndSplit(rp, last_char)
}
```

4.4.2.4.5 The *addStringToTree*() Procedure

The *addStringToTree*() procedure shows the on-line property of Ukkonen's algorithm. Note that in this procedure, we invoke the *update*() procedure once for each character as we scan the text string left to right. At the end of an iteration, the end point becomes the next active point.

```
Procedure addStringToTree(String S){
      root = new Node
      activePoint = new ReferencePair
      activePoint's state = root
      activePoint's first char = 0
      activePoint's last char = -1
      // build STree(Ti-1)..STree(Ti) by adding the new char to STree
      i = 0
      while (i < |S|)
            endPoint = update(activePoint, i)
            activePoint = endPoint
            activePoint's p = i // advance the reference pair's p
            canonize(activePoint)
}
```

4.4.2.5 Explicit versus Implicit Suffix Trees

Most of the suffix trees presented thus far have a unique ending marker *$*. The unique ending marker results in an explicit suffix tree, where the paths from the root to all leaf nodes represent all the suffices of the string. Without the unique ending marker, a suffix which is also a prefix to other suffices does not terminate at a leaf node. Instead, it ends

in the middle of a transition. The resulting tree is called an implicit suffix tree. For instance, the suffix tree for *aba* and *aba$* are show below. In the suffix tree for *aba*, suffix *a* is implicitly represented in the tree. It does not terminate at a leaf node. In the suffix tree for *aba$*, suffix *a$* is explicitly shown (Figure 15).

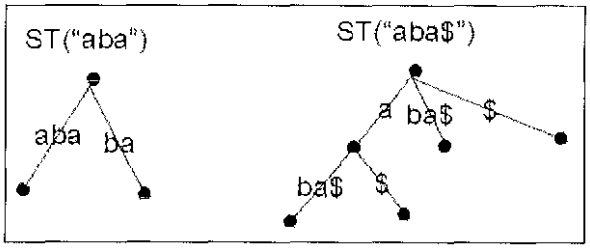

Figure 15: Implicit Suffix Tree versus Explicit Suffix Tree

Leaf nodes contain pertinent information about suffices such as their starting positions, node depths, and node identifiers. Since not all suffices of an implicit suffix tree terminate at leaf nodes, implicit suffix trees do not contain some of this information.

4.4.2.6 An Example of Suffix Tree Construction

Figure 16 shows the construction of a suffix tree for *caccao$*. We chose the string *caccao$*, because it covers all the scenarios we have mentioned. Dotted lines indicate suffix links.

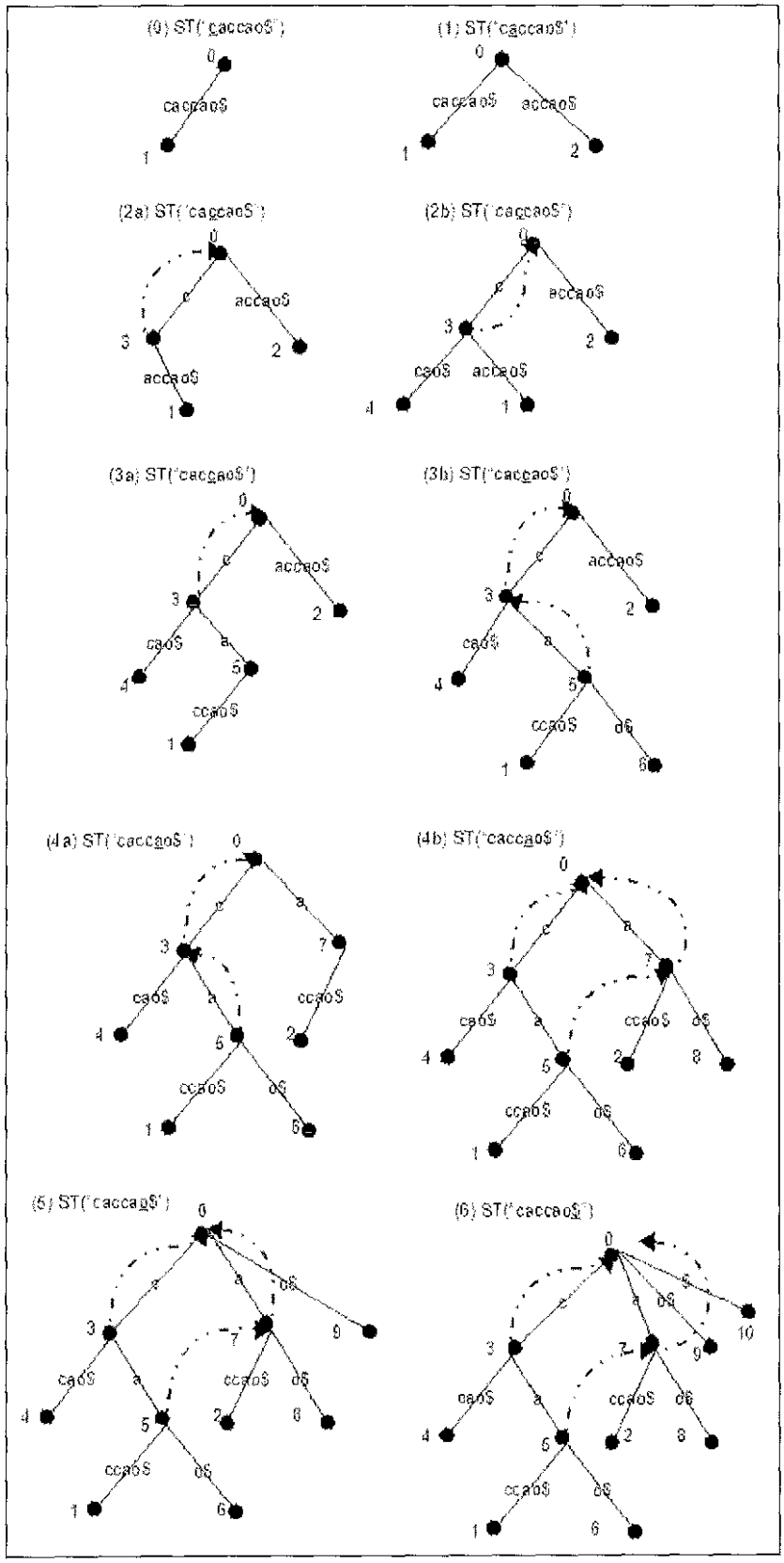

Figure 16: Suffix Tree Construction for *caccao$*

4.5 Lowest Common Ancestor

In this section, we are interested in finding the longest common prefix of two suffices in a suffix tree. This can be achieved using a very powerful tool called *lowest common ancestor* (*lca*), which was introduced by D. Harel and R. E. Targan in 1984 and improved by B. Schieber and U. Vishkin in 1988, according to Gusfield [Gusfield97, page 181]. In Figure 16 step 6, we have a suffix tree for string *caccao$*. With the root of a suffix tree being the highest node in the tree, the *lca* for node 2 and node 8 is node 7; the *lca* for node 1 and node 4 is node 3. A naive algorithm can be used to traverse up the tree from each of the two nodes in question until they both meet in O(n) time. In this section, we will learn how to locate the *lca* of two nodes in the suffix tree in constant time, O(1), independent of n.

The *lca* greatly extends the strength of many string algorithms. It is employed in conjunction with other techniques to resolve many advanced problems in linear time, some of which were, until recently, thought unattainable in linear time, including the search for the longest common substrings and the maximal palindrome problem.

Our *lca* algorithm presented here is based on the explanation in [Gusfield97]. Section 4.5 and 4.6 are in essence an excerpt of chapter eight of [Gusfield97], with our own samples and some additional notations to complement Gusfield's.

4.5.1 Binary Tree

To understand the suffix tree *lca* algorithm, we must first take a look at an interesting property of a complete binary tree. In a complete binary tree, each non-leaf node has two

children, and the total number of nodes, $n = 2p - 1$ where p is the number of leaf nodes in the tree. The path from the root to any leaf node is $d = \log_2 p$ deep. Figure 17 shows a complete binary tree where each node is labeled with its in-order number. The binary representation of each node's in-order number is shown in parentheses. What is unique about a complete binary tree is each node's in-order number in binary format actually describes the path from the root to the node. In fact, we will regard this in-order number as the node's path number and its binary representation as the node's bit path. We need $d + 1$ bits for each node to store its bit path. The root always has the left-most 1-bit set and padded with d zeros on the right. For example, a complete binary tree with $d = 5$ has a root node with 100000 as its bit path.

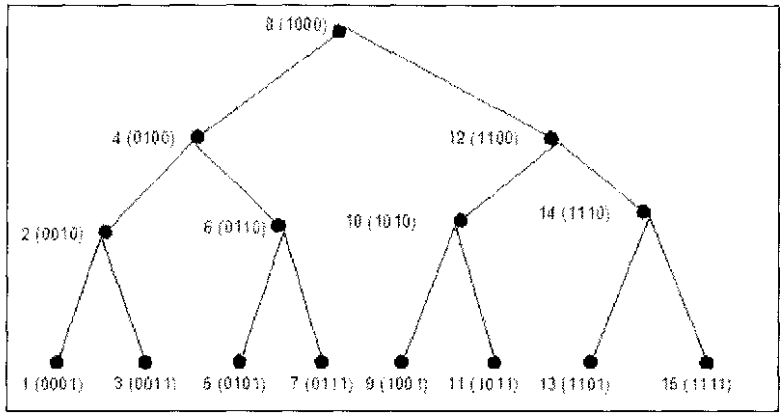

Figure 17: A Complete Binary Tree with the Nodes' In-Order Numbers Shown

The ith bit (from the left) of the bit path of some node v represents the ith edge from the root to v. If the bit is off (0), it means the edge branches left from its parent node; if the bit is on (1), the edge branches to the right. For example, node 10 in Figure 17 has a bit path of 1010. Reading the bit path from left to right, it translates to a right edge followed by a left edge. The position of the last 1-bit signifies the node's height in the binary tree. For node 10, the last 1-bit is in the second position from the right, which indicates that

node 10 has a height of two. Note in Figure 17 leaf nodes always have a height of one and their right-most bit is always 1. This inherent property of the bit path facilitates the search for the lowest common ancestor of two nodes in a complete binary tree. Given two nodes, we find the difference between their bit paths by performing a bitwise XOR. For example, the XOR for 0101 (node 5) and 0111 (node 7) is 0010. The left-most 1-bit position, *k,* is three, counting from the left. That indicates the two nodes start to diverge at depth three. Prior to the divergence (the first and second bit), they share the same path from the root to node 6, which has a bit path of 0110. The algorithm to locate the *lca* is as follows:

1. XOR the bit paths.
2. Shift the bit path of either one of the nodes to the right by $d - k$ position.
3. Set the right most bit to 1.
4. Shift the result to the left by d-k position.

In our example with node 5 and node 7, and with d=4, the steps are:

1. 0101 XOR 0111 = 0010, k=3
2. 0101 >> $d - k$ = 0010
3. 0010 => 0011
4. 0001 << $d - k$ = 0110 = node 6

Here is another example with node 9 and node 13, and d=4.

1. 1001 XOR 1101 = 0100, k=2
2. 1001 >> $d - k$ = 0010
3. 0010 => 0011
4. 0011 << $d - k$ = 1100 = node 12

4.5.2 Mapping a Suffix Tree to a Binary Tree

Before we can apply the binary tree *lca* technique, we have to map our suffix tree nodes to a binary tree, while retaining some of the nodes' ancestry information. We start by traversing through the suffix tree depth-first (pre-order) and assign a number to each node in O(n) time. Figure 18 shows the suffix tree for *caccao$* with depth-first numbering, as well as the binary representation of the numbers.

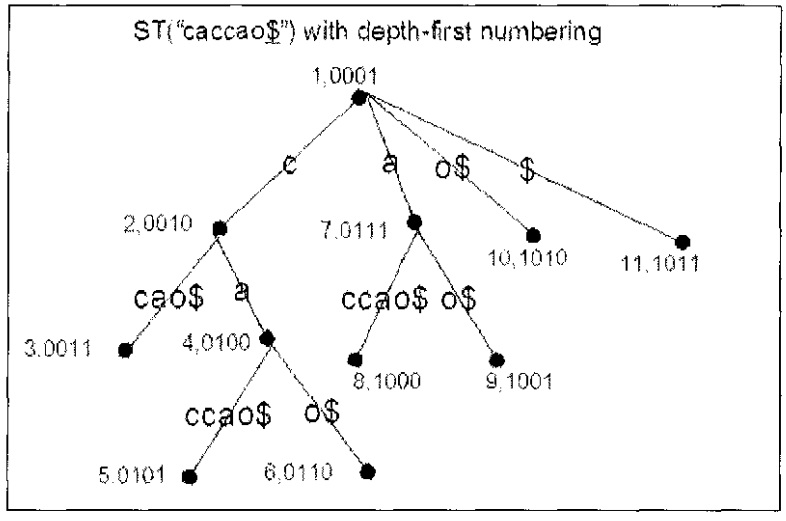

Figure 18: Suffix Tree for *caccao$* with Depth-first Numbering

Let k be the depth-first number of some node and let $h(k)$ denote the position of the right-most 1-bit of k, counting from the right. For example, $h(4)=3$, $h(8)=4$, and $h(3)=1$. We can calculate the h value of each node during the assignment of the depth-first id. Therefore, this can be accomplished in O(n) time, as well.

Next we define that for some node v, let $I(v)$ be a node w with the maximum $h(k)$ value of all nodes in the sub tree of v, inclusive of v. In other words, the k value of node v has the most zeros on its right end amongst its offspring and itself. Since $I(v)$ includes the entire

sub tree of v, we can deduce if v is an ancestor of node w, then $h(I(v)) >= h(I(w))$. Note there is always a w whose height, h, is uniquely the maximum in the sub tree of v.

Next, we group the suffix tree nodes into runs so each node in a run has the same $I(v)$. For example, Figure 19 shows how the suffix tree for *caccao$* can be organized into various runs of the same $I(v)$. Such organization ensures that $I(v)$ is always the deepest node in that run. This fact is crucial to the $I(v)$ computation using a bottom-up traversal on the suffix tree in linear time. We start by setting the leaf node's $I(v)$ value to the leaf node itself. As we move upward, if the node ID of the child's $I(v)$ is greater than the node ID of the parent's I(v), we set the parent's $I(v)$ to the child's $I(v)$.

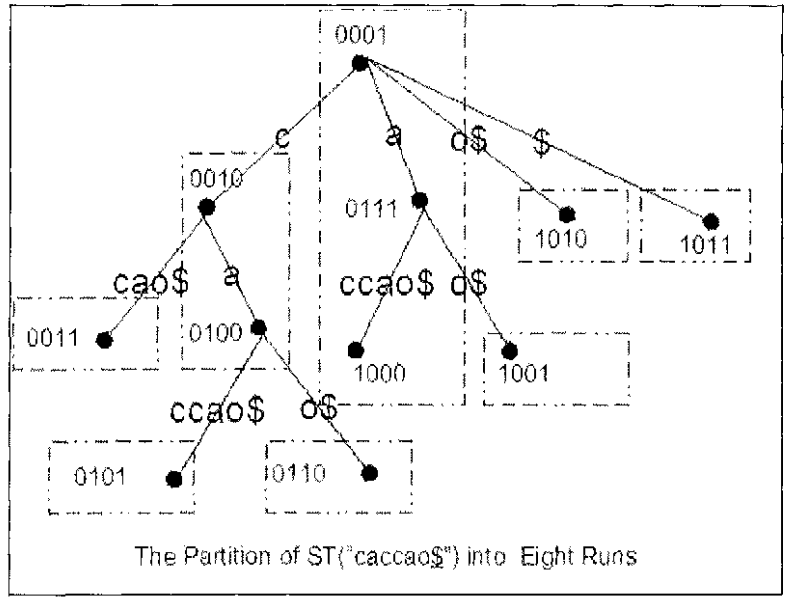

Figure 19: The Partition of the Suffix Tree for *caccao$* into Eight Runs

The fact that $I(v)$ has a unique maximal h value is important, because we need to map the $I(v)$ node of each run to a binary tree node, as illustrated in Figure 20.

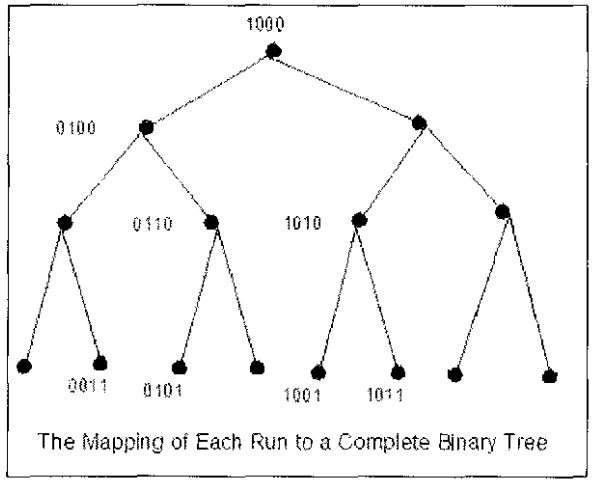

Figure 20: The Mapping of the I*(v)* Node of Each Run to a Complete Binary Tree

Next, we want to find the leader of each run. This is the node in a run closest to the root. In our example, the leader of the run containing node 1, 7, and 8 (depth-first id's) is node 1; the leader of the run containing node 2 and 4 is node 2; the leader of the run for the remaining singular runs are the individual nodes themselves. Being able to locate the leader enables us to find the next run above the current one. The parent of the leader of each run belongs to a separate run, or else the parent would have been the leader of the current run. Without the knowledge of the leader of each run, we would have to traverse up the tree and examine the *I* value of each parent node in order to locate the leader. Fortunately, we can find the leader and store them in a hash table during our bottom-up *I* value computation. We identify node *v* as the leader when node *v* and the parent of node *v* do not have the same *I* value. In our implementation, we store the leader of each run in a hash table, allowing us to retrieve the leader in O(1) average time.

For each node *v* in the suffix tree, we need to record the node in the binary tree to which the ancestors of *v* are mapped. This is a significant piece of information in facilitating the

search for the *lca* of two given nodes. To achieve this, each node is assigned an O(log n)-bit numeric variable denoted as Av. The ith bit in Av of v is set to 1 only if v has one or more ancestors mapped to height i in the binary tree. Recall we map v in the suffix tree to a binary tree node based on the bit path of $I(v)$. The ancestry information, Av, can easily be set after $I(v)$ of the nodes have been computed. We traverse down the suffix tree and copy the parent's Av information to the current node v, then set the ith bit of Av to 1, where $i = h(I(v))$. Note the same ith bit may be set more than once when v and its parent are on the same run, but this is not a problem. We can accomplish the ancestry information mapping in O(n) time, as well.

To summarize, here are the steps to map a suffix tree to a complete binary tree.

1. Traverse down the suffix tree depth-first. Assign depth-first numbers to the nodes and compute their h values.

2. Determine the $I(v)$ of each node and locate the leader of each run during the bottom-up traversal.

3. Map the suffix tree nodes to the binary tree nodes by associating each node with their respective positions in the binary tree. Implement the binary tree in the form of a binary heap. Store the nodes' depth-first numbers and in-order numbers in integer arrays [Weiss02, pages 715-717]. These arrays can be discarded to free up resources once the mapping has been completed.

4. Traverse down the suffix tree and preserve each node's ancestry information, Av.

Now that we have enhanced the suffix tree nodes with their respective information of *h*, *I*, *Av*, binary tree position, and depth-first number, we are ready to examine the retrieval of the *lca* of two suffix tree nodes in constant time.

4.5.3 Finding *lca* in Constant Time

Given two nodes *x* and *y* in the suffix tree, we want to find the lowest common ancestor (*lca*). The steps to locate the *lca* are as follows [Gusfield97, page 190].

Step 1. In the binary tree *B*, the node to which the *lca* of *x* and *y* is mapped tells us which run *z* falls under. Here are the details.

a) Find the *lca*, denoted as *h*, of *I*(*x*) and *I*(*y*) in the binary tree *B* as described in section 4.5.1. However, thus far we have only looked at how to locate *h* if *h* is neither *x* nor *y*. In the case where either *x* or *y* is *h*, *x* is the ancestor of *y*, if and only if, the following two conditions are present:

i. The depth-first id of *x* <= the depth-first id of *y*

ii. The depth-first id of *y* < the depth-first id of *x* + node count of sub tree *x*

Gusfield [Gusfield97, page193] describes a way to count the number of child nodes for each binary tree node by traversal. We have instead devised a formula that computes the number of child nodes based on the binary tree height and the position of the node in the binary heap. The formula is as follows:

c = the number of nodes in the binary sub tree of v (including v)

$c = (2^x) - 1$, where x = the height of B – floor($\log_2(n)$), and

n = the in-order number of v

b) Let $i = h(b)$ = the height of the *lca* b in the binary tree.

c) Use i to find j, where j represents $h(I(z))$ and $j >= i$ and Av[j]=1 for both x and y. Note i and j are counted from the right (the least significant bit).

Step 2. Locate node x', which denotes the closest node to x on the same run as z. In other words, x' is the node where we start entering the run that contains z. Note x' could potentially be x. For example, in Figure 19, if x=6 (0110) and y=3 (0011), the *lca* z would be node 2 (0010). In this case, x' would be node 4 (0100), and y' would be node 2 (0010) itself. To do so, the procedure is as follows:

a) If $h(I(x)) = j$, set $x' = x$ and go to step 3. This is because x and z are on the same run. This approach is simpler than the steps described in [Gusfield97, page 191].

b) Find k where k represents $h(I(w))$ and w is the node closest to the run of z (but not on the run). k = the left-most 1-bit to the right of j in the Av bits of x. Using k, we can derive the binary tree path number of node w using bitwise operations on k. Shift k to the right by k - 1 bits, set the right-most bit to 1, then shift k to the left by k - 1 bits. This identifies the run to which w belongs.

c) Obtain w by looking up the hash table for the leader of the run identified above.

d) Return x', the parent node of w. This is the entry point into the run of z.

Step 3. Repeat step 2 for node y to find y'.

Step 4. Compare x' and y'. The one with the higher depth-first id is the lowest common ancestor of both x and y. In our example in Figure 18, node 2 is the *lca*.

Each step above takes constant time to perform after preprocessing. Therefore, the algorithm to locate the *lca* of two nodes in a suffix tree can be done in constant time.

4.5.4 A Note on Our *lca* Implementation

To support the *lca* algorithm and computation, we have to enhance our Node class with variables to hold the depth-first id, h value, I node reference, binary tree position, and Av bits. We also enhanced our SuffixTree2 class with an auxiliary class Lca. The class Lca encapsulates all codes pertaining to the lowest common ancestor algorithm. It is designed to isolate the SuffixTree2 class from the *lca* piece for clarity and ease of maintenance. The full implementation of all our suffix tree and related classes can be found in the companion CD. Some variables and arrays may be discarded once the suffix tree is constructed or when the *lca* is computed. We have chosen to keep certain temporary processing storage for debugging and educational purposes.

4.6 The Longest Common Extension

The longest common extension (*lce*) problem is central in many string algorithms. The goal is to compute the length of the longest common prefixes between a suffix x of string *S1* and a suffix y of string *S2* in constant time. In Figure 21, substrings x and y of *S1* and *S2* have an *lce* of 5, where i and j are the starting positions of x and y respectively.

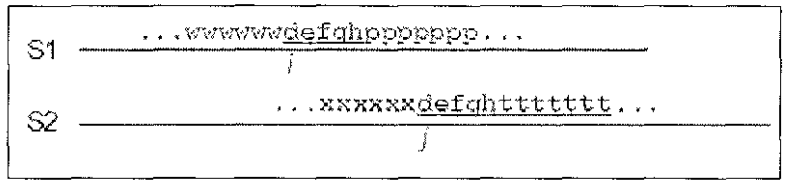

Figure 21: The *lcp* of Substrings x and y is 5

The concept is similar to the *lca* algorithm. While the *lca* deals with two suffices within the same string, the *lce* deals with two suffices of two distinct strings. In fact, our *lce* implementation is built on top of the *lca* algorithm.

4.6.1 Generalized Suffix Tree

It is possible to add the entire set of suffices of string *S2* to the suffix tree of string *S1* to take advantage of common prefixes. The resulting tree is called a *generalized suffix tree*. Each node in this generalized suffix tree will have bits identifying the string(s) to which it belongs. The node and the transition could be shared by multiple strings, and each string must have its own unique ending marker that does not appear anywhere else in the string content. For example, we use *$* and *#* for *S1* and *S2* respectively. This approach may be generalized further to accommodate more strings.

For implementation, the identifying bits of the nodes and the transitions need to be set:

1. When the node and transition objects are instantiated.

2. When the reference pair (active point) is being canonized. This is because we traverse down the tree on behalf of the string being added. Therefore, we need to mark the bit set to indicate they are a valid path for the string.

There is one more implementation detail we must look at to ensure *lce* retrieval takes constant time. At each leaf node representing suffix $S[i..n]$, we need to record the index i, which is the starting position of the suffix. After all suffices have been added to the suffix tree, we traverse down the tree and calculate the distance from the root for each node along the way. When we reach a leaf node, we record its suffix starting position. We also keep two arrays of Node references, *N1* and *N2*, which point to leaf nodes of the suffices for *S1* and *S2*. *N1* and *N2* allow us to locate a leaf node of a suffix based on its beginning position. For example, $N1[5]$ is a reference to the leaf node for suffix $S1[5..n]$. The pseudo code for computing the *lce* of two suffices x and y of *S1* and *S2* respectively is as follows:

```
procedure getLce(suffixPos1, suffixPos2) {  // returning the lce value
      node1 = N1(suffixPos1)
      node2 = N2(suffixPos2)
      lca = the lca of node 1 and node 2 (section 4.5.3)
      return lca's node depth
}
```

Chapter 5

HYBRID DYNAMIC PROGRAMMING WITH SUFFIX TREES

When performing exact string matching on very long strings, using the suffix tree gives us an advantage over the Boyer-Moore and Knuth-Morris-Pratt algorithms. Boyer-Moore and Knuth-Morris-Pratt algorithms preprocess the pattern in O(m) time. They then scan the text in O(n) time to search for the pattern. A suffix tree, on the other hand, preprocesses the text in O(n) time. Subsequent searches for any pattern thereafter require only O(m) time. In this chapter, we introduce the concept of hybrid dynamic programming with a suffix tree that can solve a k-difference problem in O(kn) time and space.

5.1 The Concept of Diagonals

In 1983, Ukkonen introduced a diagonal transition algorithm that has an O(k^2) run-time [Navarro01, page 48]. The concept is based on the observation values running on the downward, left-to-right diagonals of the dynamic programming table increase monotonically. Figure 22 illustrates the diagonal concept of a dynamic programming table. The main diagonal (diagonal 0) is the bold line. Diagonals below the main diagonal are marked with numbers from -m to -1 and diagonals above the main diagonal are from 1 through n.

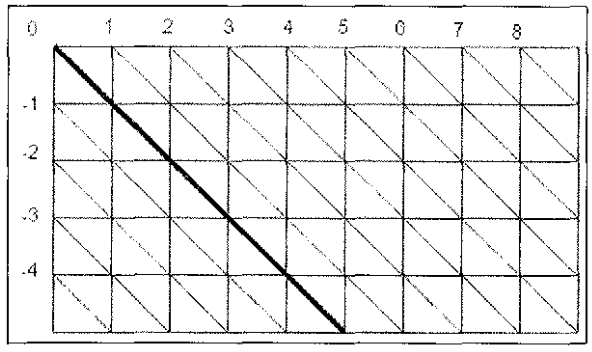

Figure 22: The Diagonal Concept of
a Dynamic Programming Table

Landau and Vishkin adopted this idea and introduced the first hybrid dynamic programming with a suffix tree approach that improves the run-time to $O(kn)$. The basic idea is we calculate the dynamic programming table diagonally and use the *lce* extension to solve the sub problem of the longest common prefix between the two strings, in constant time as we slide down the diagonals. We increment our error count by one and skip the mismatching character. We repeat the process until the error count exceeds k. If we reach k before we get to the end of the diagonal, we abandon the diagonal and move to the next one. If we reach the end of the diagonal, we have an occurrence of P in T with at most k differences.

5.2 The Concept of *d*-path

Gusfield defines a *d*-path as follows:

> A *d*-path in the dynamic programming table is a path that starts in row zero and specifies a total of exactly d mismatches and spaces.
>
> A *d*-path is the farthest reaching in diagonal i if it is a *d*-path that ends in diagonal i, and the index of its ending column c (along diagonal i) is greater than or equal to the ending column of any other *d*-path ending in diagonal i [Gusfield97, page 265].

In other words, the d-path of diagonal i is a path from row zero that ends in diagonal i with d differences. What we are interested in is the farthest-reaching d-path in diagonal i, which is a path with d differences that starts in row zero and ends in the deepest cell in diagonal i in the dynamic programming table. For the k-difference problem, we want to find the k-path for each diagonal in the dynamic programming table.

The d-path for diagonal i can be computed using the (d-1)-path for diagonal i+1, i-1, and i. We call these three paths R1, R2, and R3 respectively and define them as follows:

1. R1 represents the farthest-reaching (d - 1)-path on diagonal i + 1, accompanied by a vertical jump (equivalent to insertion of a space in the text) onto diagonal i. The jump essentially brings us from the d - 1 path to d path. Then we slide down on diagonal i until we find the next mismatch, using the suffix tree *lce* extension. At the end, R1 is a d-path.

2. R2 represents the farthest-reaching (d - 1)-path on diagonal i - 1, accompanied by a horizontal jump (equivalent to insertion of a space in the pattern) onto diagonal i. The jump essentially brings us from the d - 1 path to d path. Similarly, we slide down on diagonal i until we find the next mismatch, using the suffix tree *lce* extension. At the end, R2 is a d-path.

3. R3 represents the farthest-reaching (d - 1)-path on diagonal i itself. Since we knew this was where the last mismatch occurred, we skip one character. That essentially brings us from the d - 1 path to d path. Then we slide down on

diagonal i until the next mismatch, using the suffix tree *lce* extension. At the end, R3 is a d-path.

Since R1, R2, and R3 are all d-paths, the farthest-reaching d-path is the farthest among the three. Figure 23 demonstrates the concept of R1, R2, and R3.

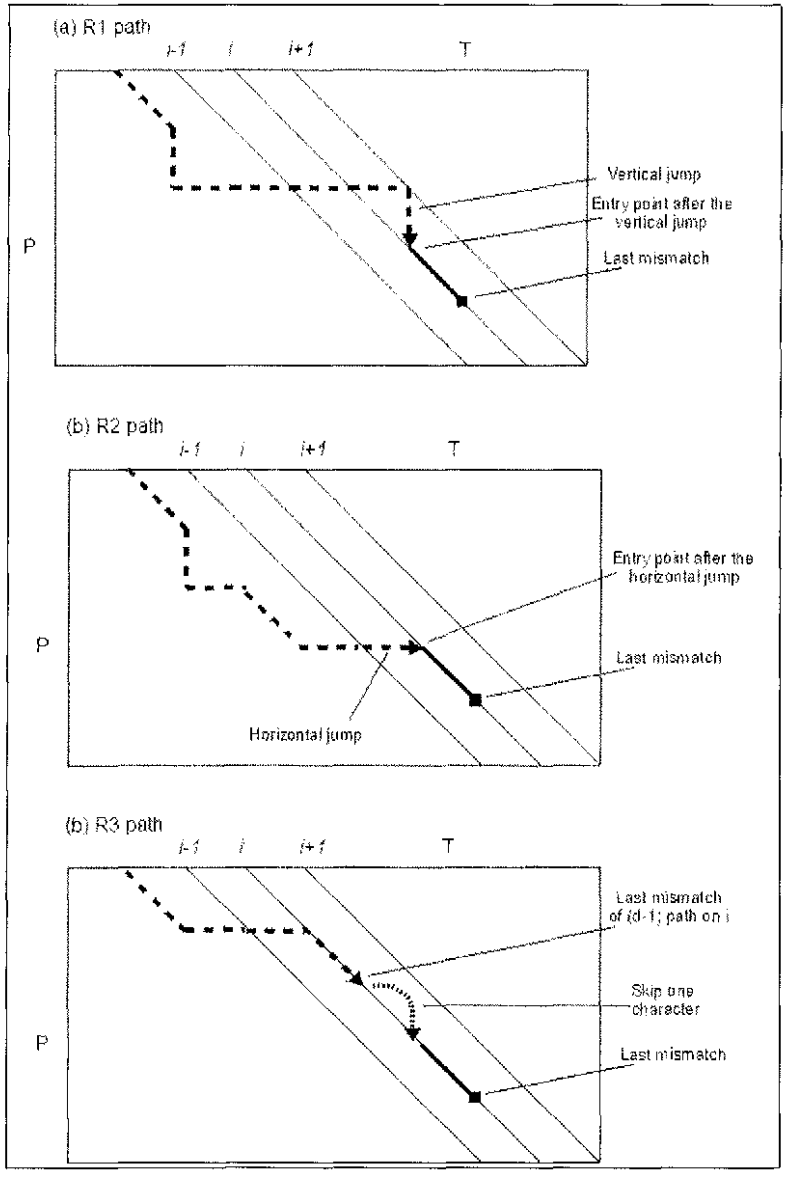

Figure 23: R1, R2, and R3 d-paths

5.3 Implementing the Hybrid Approach

Basically, the hybrid approach takes a pattern P and a body of text T, and build a d-path table of k rows and $m + n$ columns. For each error d, we iterate diagonals $-m$ through n. For each diagonal, we compute how far we can reach with d differences allowed. At the end of the algorithm, we examine the k row in the d-path table. Columns that reach farther than m indicate an approximate match of P in T with at most k differences. Pseudo code for this process follows:

```
procedure kdifferenceWithSuffixTree(String P, String T) {

      Obtain the text and the pattern

      Initialize the suffix tree and its lca extension

      Initialize the d-path table

      For d = 0 (the first row), for i=0 to n, find the lce between
      P[1..m] and T[i..n].  This is essentially exact string matching

      For d = 1 to k

            For each diagonal i (from -m to n)

                  use (d - 1)-path on the d-path table to find R1, R2,
                  and R3

                  update the d-path table (d row, i column) with the max
                  value among R1, R2, and R3

      On row k, any values that reach m indicate an approximate match of
      P in T with at most k differences.

}
```

Figure 24 shows a sample *d*-path table as well as a reconstructed dynamic programming table.

```
Pattern: GGATCGA$
Text: GRATTCAGTTA$
k: 4

The farthest-reaching d-path table for diagonal-i:

0)  0|0|0|0|0|0|0|0|1|0|0|0|0|0|1|0|0|0|0|0|
1)  1|1|1|2|1|1|1|2|1|1|1|1|1|2|2|1|1|1|1|0|
2)  1|2|2|3|3|2|2|4|3|3|2|2|3|4|4|3|3|2|2|2|0|
3)  1|2|3|4|4|4|4|5|7|6|6|4|5|5|5|4|4|3|3|2|0|
4)  1|2|3|4|5|5|5|7|8|7|7|6|7|6|6|5|5|4|3|2|0|

The reconstructed k-difference dynamic programming table for k=4

    - G R A T T C A G T T A
0) -|0|0|0|0|0|0|0|0|0|0|0|0|
1) G|0|0|1|1|1|1|1|1|0|1|1|1|
2) G|0|1|1|2|2|2|2|2|1|1|2|2|
3) A|0|1|1|1|2|3|3|2|2|2|2|2|
4) G|0|1|2|2|1|2|3|3|3|2|2|3|
5) C|0|1|2|3|2|2|2|3|4|3|3|3|
6) G|0|1|2|3|3|3|3|3|3|4|4|4|
7) A|0|1|1|2|3|4|4|3|4|4|3|4|
```

Figure 24: The *d*-path Table and the Reconstructed Dynamic Programming Table

In our *d*-path table above, we just need to examine the cells in the last row that fall between columns $-m + (-k)$ and $n - m$. Cells with values equal to the pattern length (less 1 for the ending marker) indicate an approximate match of the pattern P in text T. The size of the *d*-path table may be reduced from $O(kn)$ to $O(m + n)$, if we do not need to locate the starting position of the approximate match. Since we calculate $m + n$ diagonals in k iterations and the *lce* computation takes O(1) constant time, our implementation for the *k*-difference problem runs in $O(kn)$ time. The implementation of the hybrid algorithm does not require the dynamic programming table, which takes up $O(mn)$ space but is helpful during the debugging process. The primary *d*-path table requires k x $(m + n)$ space to record the *d*-path result. Since the size of the pattern is relatively insignificant in comparison to the text size, we can generalize the space requirement into $O(kn)$.

The k-difference solution using hybrid dynamic programming is not difficult to understand, but implementing it and verifying its correctness is time-consuming. First and foremost, a dynamic programming algorithm should already be implemented so we can verify the results of the hybrid approach. Before we can verify the result, the d-path table might need to be translated into a dynamic programming table so we can compare the results. The reconstructed dynamic programming table is also read slightly different than a dynamic programming table generated with a pure dynamic programming algorithm. This is because the dynamic programming table is concerned with the minimum edit distance, while the reconstructed dynamic programming table of the hybrid approach is concerned with the maximum matches of P and T, as shown in Figure 24.

Chapter 6

SUFFIX ARRAYS

As well studied as the suffix tree is, there are some intrinsic drawbacks in the data structure. The $O(n)$ space requirement of the suffix tree can measure twenty to fifty times the text size, which is detrimental in some application areas. Complex suffix tree construction algorithm is another reason the data structure is not commonly known among computer programmers. In order to achieve the $O(m)$ search time, a suffix tree requires $O(n\sigma)$ space. Alternatively, a search time of $O(n \log \sigma)$ time can be achieved with $O(n)$ space, assuming the alphabet size is fixed [Gusfield97, page 149].

The impact of the alphabet size σ is less of a concern for a language such as English where the entire alphabet can be represented with 128 symbols (ASCII characters). For other languages such as Chinese where each of the more than 30,000 characters must be assigned a unique code, this presents a space issue. Most non-English languages use a Unicode (16-bit) character set instead of the 7-bit ($2^7 = 128$) ANSI character set. Applications requiring an extremely high number of symbols are not related to human languages. In imaging, pictures are composed of long strings of characters, each of which represents a color component of a pixel. In molecular biology, long strings of integers represent locations in a DNA sequence where certain substrings are found [Gusfield97, page 155]. Each integer in this case represents a unique symbol in an alphabet. Therefore, the alphabet sizes could be in the range of millions or more.

6.1 The Concept

In 1989, Manber and Myers introduced the concept of a *suffix array*, which can be used to solve some of the most common suffix tree applications with three to five times less space [Manber93, page 1]. A suffix array of string S is the lexicographically sorted suffices of S. A suffix array is normally in the form of an integer array that represents the positions of the suffices in the string. For example, the suffix array for string *mississippi* is:

Order	Pos	Text
0	10	i
1	7	ippi
2	4	issippi
3	1	ississippi
4	0	mississippi
5	9	pi
6	8	ppi
7	6	sippi
8	3	sissippi
9	5	ssippi
10	2	ssissippi

Figure 25: The Suffix Array for *mississippi*

6.2 The Efficiency of a Suffix Array

6.2.1 Space Requirement

Unlike the suffix tree, since a suffix array is an integer array that stores the positions of suffices, it is not subject to the size of the alphabet and is optimal for large alphabets. Even with it auxiliary longest common prefix (*lcp*) extension (section 6.4), a suffix array requires only $2n$ space. Although in practice a suffix array could take up to $5n$ space, which is an order of magnitude less than the space requirement of a suffix tree ($20n$ to $50n$). This makes the suffix array an ideal candidate for many applications.

6.2.2 Search Time

In its simplest form, a suffix array *sa* can be used to locate a suffix P in a string S of length n in $O(m \log n)$ time using a basic binary search. However, complemented with *lcp* extension and advanced binary search techniques, we can improve our search for P in T to $O(m + \log n)$ time. More importantly, the efficiency is independent of the alphabet size, which is a major concern in many application areas.

6.3 Suffix Array Construction

Since its introduction, researchers have found several approaches to construct suffix arrays. We discuss three of them below.

6.3.1 The Naive Approach

The naive approach to build a suffix array involves looping through the string n times. During the iteration, the algorithm compares the current suffix to each suffix, which takes $O(n^2)$ time. That brings the total time of the naive approach to $O(n^3)$. Although this approach is not practical for real world applications, it is easy to understand and can be used to validate more advanced approaches on shorter strings. The pseudo code follows.

```
procedure suffixArrayNaive(char[] s)
{
      bitset = bit[n]   // to track if a suffix has been
                        //   assigned a position
      for (i=1 to n)
            minPos = -1   // to track the next min suffix position
            for (j=1 to n)
```

```
            if (bitset[j]) continue // this suffix has
                                    //  been assigned. Skip
            if (minPos = -1) minPos = j
            else minPos = min(minPos, j) // get the min of the two
        sa[i] = minPos
        bitSet[minPos] = true   // indicate suffix has
                                //  been assigned a position
}
```

6.3.2 The Suffix Tree Approach

Deriving a suffix array from a suffix tree is quite straightforward. We traverse down the suffix tree from the root in depth-first manner and visit the child nodes in the lexicographical order. Such traversal ensures that leaf nodes visited are always in lexicographical order. Figure 26 shows a suffix tree for the string *bananas$*. Each node is shown with its lexicographical order number.

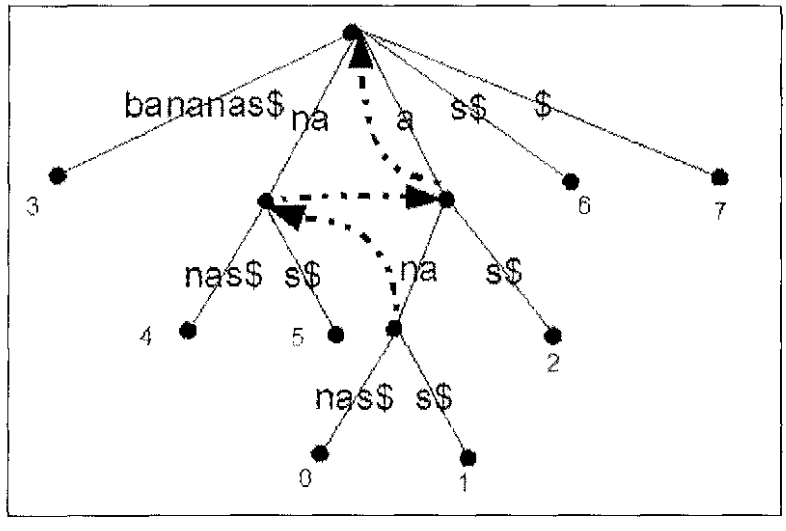

Figure 26: The Suffix Tree for bananas$ with
Leaf Nodes Lexicographically Marked

6.3.3 The Linear Time Approach

Manber and Myers introduced a suffix array construction algorithm without constructing a suffix tree in advance [Manber93]. This algorithm takes O(n) expected time and O(n log n) worst-case time. Constructing suffix arrays using this approach can take three to ten times longer than deriving them from suffix trees.

Finally in 2003, three different linear time approaches to construct suffix arrays without involving suffix trees were developed. We briefly describe the *skew algorithm* developed by Karkkainen and Sanders [Karkkainen03]. The approach is indeed very fast and consists of the following four main steps:

1. Given a string S with suffices 0..n, 1..n, 2..n, 3..n, ..., i..n, divide the suffices into three buckets of $k = i \bmod 3$. So $k = 0$, 1, and 2.

2. Recursively radix-sort the suffices for bucket $k = 1$ and 2 together. Then assign each suffix their ranking in the bucket.

3. Repeat step 2 for bucket $k = 0$ and rank each element as well.

4. Merge the resulting arrays from step 2 and step 3 using a regular sorted array merging technique. The result is a suffix array for string S.

6.4 The Longest Common Prefix

The longest common prefix (*lcp*) is an auxiliary integer array that can improve the search time of a suffix array to O(m + log n) time. The *lcp* array keeps track of the length of the

longest common prefix, *lcp*(*i*, *j*) of two adjacent suffices in the suffix array. The *lcp* information for the string *mississippi* is shown in Figure 27.

The *lcp* extension allows for the retrieval of the longest common prefix between two suffices in constant time. In 2001, Kasai, Lee, Arimura, Arikawa, and Park introduced a new *lcp* construction algorithm in O(*n*) time [Kasai01]. Through clever observation of the relations between each suffix and previously acquired *lcp* result, the algorithm ensures that each character is examined only once during the iteration, hence achieving a linear time *lcp* construction. In [Manber93], Manber and Myers offered an ingenious augmentation to the regular binary search algorithm with the *lcp* array. They achieved the O(*m* + log *n*) search time by making sure each character in *P* is compared only once in each search.

Order	Pos	Text	LCP
0	10	i	0
1	7	ippi	1
2	4	issippi	1
3	1	ississippi	4
4	0	mississippi	0
5	9	pi	0
6	8	ppi	1
7	6	sippi	0
8	3	sissippi	2
9	5	ssippi	1
10	2	ssissippi	3

Figure 27: The Suffix Array for *mississippi* with *lcp* Information

6.5 The Advantages of a Suffix Array

Unlike the suffix tree, a suffix array can be used to solve a range of practical problems with a modest memory requirement. Its competitive worse case search time of O(*m* + log *n*), which is independent of alphabet size, is another major advantage. In addition, a

suffix array is less complicated than a suffix tree, which contributes to its popularity among practitioners. As a suffix array and the *lcp* extension are both integer arrays, persisting them are considerably easier and a myriad of tools are readily available.

Chapter 7

EXPERIMENTS

In this chapter, we discuss a series of experiments conducted using two approximate string matching algorithms, to solve the *k*-difference problem on a very large text string and long pattern.

7.1 Overview

We conducted three sets of experiments using dynamic programming to solve the classic *k*-difference problem. We repeated the same experiments using hybrid dynamic programming with a suffix tree. We measured and compared the time required to locate the ending indices of the occurrences of pattern P in text T. Each algorithm was run against two types of data: text strings from English literature (200KB – 1MB) and a section of a DNA sequence (200KB – 1MB). The first set of experiments measured the impact of text size on search time. We varied the text length n, while keeping the pattern length m and the number of errors allowed k constant. The second set of experiments evaluated the impact of pattern length m on search time, with n and k unchanged. The last set of experiments examined how changes in the number of errors allowed k affect search time, keeping the values of n and m constant.

Several intrinsic differences between the dynamic programming and the hybrid dynamic programming algorithms are noteworthy. The dynamic programming algorithm is an on-line algorithm that requires no preprocessing of the text or the pattern. On the other hand,

the hybrid dynamic programming algorithm preprocesses the text and constructs a suffix tree in advance to gain performance in subsequent searches. Although the suffix tree construction takes $O(n)$ time, we were only concerned with the search time. In many applications, the text is known in advance. The results of our experiments are presented in tabular and graphical formats.

7.1 The Objectives

The dynamic programming algorithm can be broken down into three parts: initialization of the dynamic programming table, construction of the table, and locating the occurrences. The hybrid dynamic programming algorithm consists of three major parts as well: initialization of the suffix tree and *lca* data structures, construction of the d-path table, and locating the occurrences. The suffix tree construction time and the *lca* preprocessing time were excluded from our measurement for reasons previously stated. We measured the time to initialize and fill in the d-path table and the time to retrieve the ending indices of matches.

The experiments had two main objectives:

1. To measure the impact of text size n, pattern length m, and number of errors allowed k, on both the dynamic programming and the hybrid algorithms by varying one variable at a time.

2. To measure the impact of alphabet size σ on both the algorithms using two sets of data: ASCII-based literature and a DNA sequence.

7.2 Experiment Details

7.2.1 Hardware Platform

The experiments were run on a computer with a 32-bit x86 AMD Athlon-XP 1.7 GHz processor with 256KB L2 cache. The system runs at 266MHz front system bus speed with 768 MB of PC2100 (266 MHz) DDR RAM.

7.2.2 Software Platform

The experiments were conducted on a platform running SUN Java SDK version 1.4.2_10. The operating system was SuSE Linux 9.0 Professional with kernel version 2.4.2.

7.2.3 Experiment Data

The experiment input consisted of very large text strings in English and a sample DNA sequence. The text strings were collected from the Project Gutenberg archive, including Confucian Analects, Tao Te Ching, Moby Dick, The Notebooks of Leonardo Da Vinci, and The Art of War [GreatBooks06]. Figure 28a shows a snippet of the data. The DNA sample is part of the DNA sequence of a house mouse acquired from NCBI-GenBank [GenBank06]. A snippet is shown in Figure 28b. The text strings use the ASCII character set, which has an alphabet size of 128. The DNA sequence consists of nucleotides encoded with characters A, T, C, and G. It has an alphabet size of four.

Each experiment was carried out in five runs and the average search time was recorded. To generate a pattern for the experiment, we randomly selected a string of length m from the body of text. We randomized the sampled string with up to k actions of insertions,

deletions, replacements, or doing nothing. Therefore, when we specified a pattern of length m, our randomized pattern generator returned a pattern with a length between m - k, when k deletions were performed, and $m + k$, when k insertions were performed.

```
CHAP. XXI. Tsze-lu asked whether he should immediately
carry into practice what he heard. The Master said, 'There are your
father and elder brothers to be consulted:-- why should you act on
that principle of immediately carrying into practice what you hear?'
Zan Yu asked the same, whether he should immediately carry into
practice what he heard, and the Master answered, 'Immediately
carry into practice what you hear.' Kung-hsi Hwa said, 'Yu asked
whether he should carry immediately into practice what he heard,
and you said, "There are your father and elder brothers to be
consulted." Ch'iu asked whether he should immediately carry into
practice what he heard, and you said, "Carry it immediately into
practice." I, Ch'ih, am perplexed, and venture to ask you for an
explanation.' The Master said, 'Ch'iu is retiring and slow;
therefore,
```

Figure 28a: A Snippet of the Text Strings Used in the Experiments

```
Atccaccaggtttgaagatctggatattgtaatctgtgagatccatgggtttttaaaatgctt
atcataaaaaacatgcaatcatccatgcaagtggttggctttcctttgtgtctccctaagttctgtt
gctgactccatgattgtactggagcacacgttgaaaattgtgtatgttgggaggagatgtctg
ttggtgatgctttgcaaqggtcttacaagttgtggatgataagaatcacgattgtaqtatcat
acctgttcatgttgttcccatgttcattgttcagatattaaaatgtcatttttataatggacc
aggcccagatcgaggtggtgccttctgcctcagccctgatcatcaaagccctcaaggagccac
gaagcagaagaacattaaacacagtgggaacattacttttgatgagattttcaacattgcccg
cggtctttggctagagaactttctggaactatcaaggagatcctgggtgctgcacagtctgtg
atggccgccaccctcatgacatcatagatgacatccacagtggtgcagtggagtgcccagcta
aagaaaatattccaataaaagactatctgataaccagccgcagggatgtttatgtcatcacatg
ttatctttcattcctccaattgccagtagtaccgatcttttttccatgaggtgtgaagaggca
atcgaaagagcacttttcctcttcatttcactttgaaattcagcatcattaaaacttttcc
ctttgaaaggaatcagttaaaaatcacatcatagctgggcatggtggtgcatgcctttaatcc
gcagaggcaagtagatttctgagttcgaggccagcctggtctataaagtgagttccaggacag
```

Figure 28b: A Snippet of the DNA Sequence Used in the Experiments

7.3 Experiment Results

7.3.1 Experiment 1

Figure 29a shows the results for Experiment 1, which measured the impact of text size while keeping pattern length and number of errors allowed constant. Figure 29b and 29c show the graphs for the DNA and the literature results.

Experiment 1a & 1b – The Impact of Text Length, n (with m=1000, k=20)

	Time (sec)			
	DNA		Literature	
File Size (KB)	DP	Hybrid	DP	Hybrid
200	8.77	12.72	8.59	13.82
400	17.96	27.28	17.12	29.21
600	26.42	40.38	26.35	50.52
800	55.03	54.96	34.64	68.97
1000	74.91	89.92	44.45	76.1

Figure 29a: The Result of Experiment 1

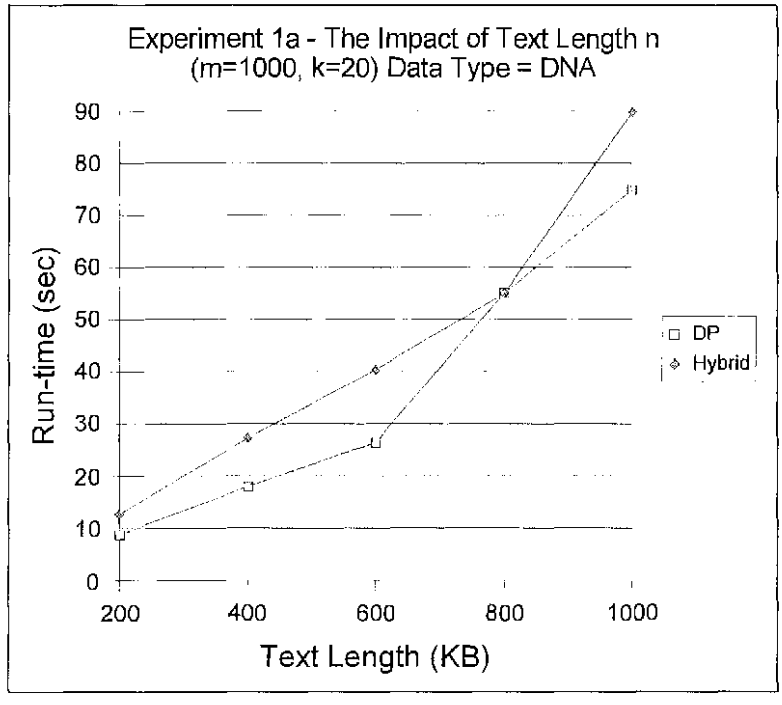

Figure 29b: The Graphs for the Results of Experiment 1a

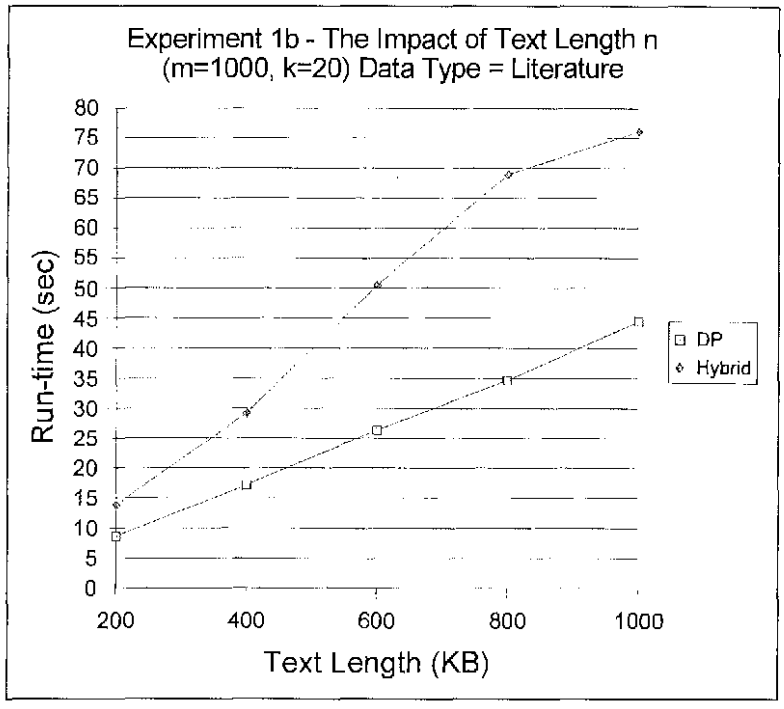

Figure 29c: The Graphs for the Results of Experiment 1b

In Experiment 1, the search times for both the dynamic programming approach and the hybrid approach increased linearly with text size. Note that dynamic programming outperformed hybrid dynamic programming in this experiment, but we were mainly interested in the effect of the text size on search times. This set of experiments clearly supports the O(mn) and O(kn) analysis on the search time, since m and k are constant.

7.3.2 Experiment 2

Keeping text size and number of errors allowed constant, Experiment 2 examined the impact of pattern length on search time. The results are shown in Figure 30a and graphed in Figure 30b and 30c.

Experiment 2a & 2b – The Impact of Pattern Length, m (with n=400KB, k=20)

	Time (sec)			
	DNA		Literature	
m (thousand)	DP	Hybrid	DP	Hybrid
0.5	9.27	27.98	13.8	28.51
1.0	18.55	31.56	17.22	28.71
1.5	30.64	26.58	25.63	28.82
2.0	40.86	26.7	35.08	28.46
2.5	51.08	26.87	44.94	31.14
3.0	61.69	31.98	53.54	33.71
3.5	71.86	32.16	59.89	33.18
4.0	82.36	31.61	68.82	32.81
4.5	92.67	30.99	86.4	34.81
5.0	103.02	32.51	100.99	33.55

Figure 30a: The Results of Experiment 2

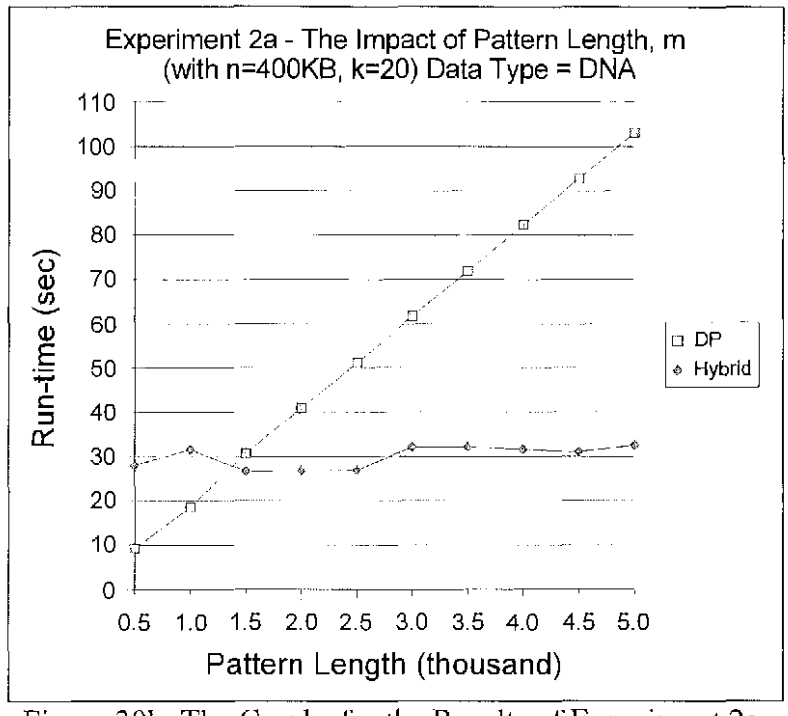

Figure 30b: The Graphs for the Results of Experiment 2a

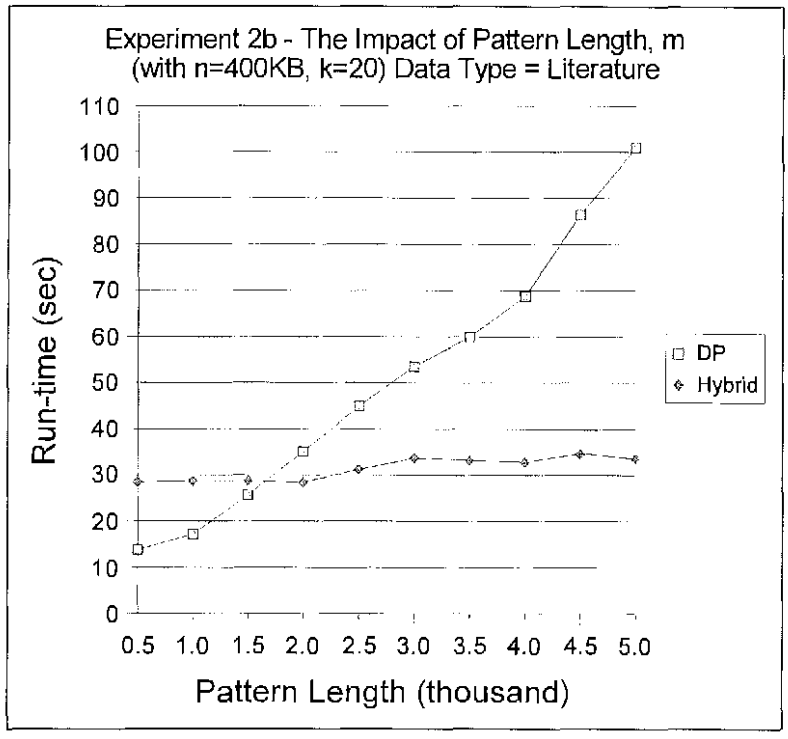

Figure 30c: The Graphs for the Results of Experiment 2b

In Experiment 2, we observed the advantages offered by a suffix tree, especially for a very long pattern. The hybrid approach easily outperformed the dynamic programming approach because it solves the k-difference using the suffix tree's *lce*, which can be performed in constant time. Although the increase in the pattern length m led to an increase in the d-path table processing time, which is $O(k(n + m))$, the increase is quite trivial. We simplified the processing time to $O(kn)$ (section 5.3).

Even at a modest text size of 200KB, we observed a markedly superior performance by the hybrid approach using a suffix tree. For text larger than 600KB and patterns longer than 2KB, the performance of the dynamic programming algorithm deteriorated so much that the hybrid programming approach is advantageous even if we include the time for suffix tree construction in the comparison. It is evident the hybrid dynamic programming

algorithm with a suffix tree has a significant edge over the pure dynamic programming algorithm, when the text is very large and the pattern is long.

7.3.3 Experiment 3

Experiment 3 examined how the number of errors allowed affects search time. Figure 31a shows the results. Figure 31b and 31c show the graphs for the results.

Experiment 3a & 3b – The Impact of errors allowed, k as a Percentage of Pattern Length, m (with n=400KB, m=2000)

	Time (sec)			
	DNA		Literature	
k (% of m)	DP	Hybrid	DP	Hybrid
40 (2%)	39.37	59.42	39.52	60.49
80 (4%)	38.65	120.88	35.5	116.79
120 (6%)	38.02	176.76	35.53	188.82
160 (8%)	38.77	225.04	37.28	268.08
200 (10%)	39.06	290.59	36.83	346.87

Figure 31a: The Results of Experiment 3

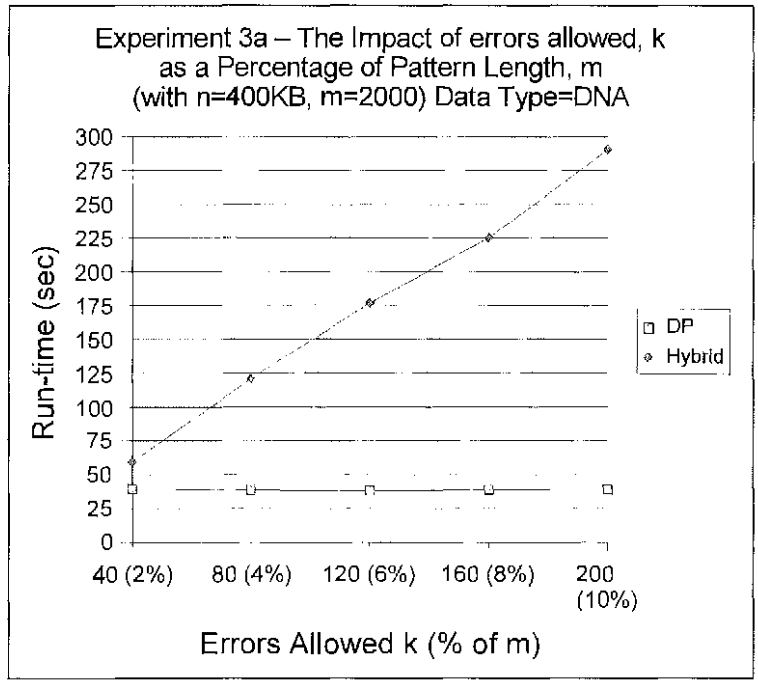

Figure 31b: The Graphs for the Results of Experiment 3a

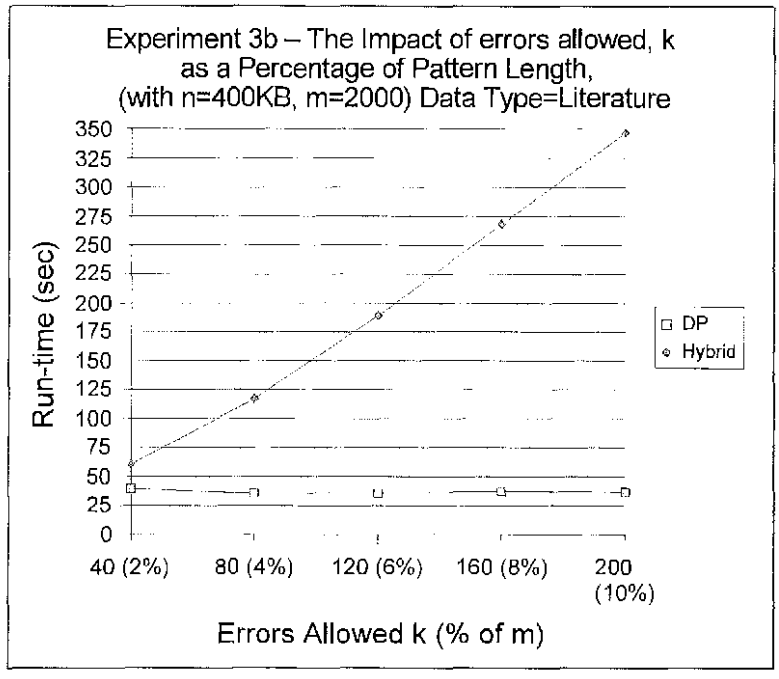

Figure 31c: The Graphs for the Results of Experiment 3b

In Experiment 3, we examined how k, the number of errors allowed, affects the search time. It showed a change in k has a significant impact on the hybrid approach, but not on the dynamic programming approach. The latter remains unaffected by changes in k, because we simply scan through the last row of the dynamic programming table and find entries smaller than k, which requires $O(n)$ time regardless of the value of k. This underscores our $O(kn)$ and $O(mn)$ analysis for each approach respectively.

7.4 Analysis of the Experiments

Overall, our experiments yielded outcome consistent with the theory. The results give us great insight into the role of the variables we measured and confirmed our understanding of the data structures.

7.4.1 The Impact of the Alphabet Size

The outcome of all three experiments were not affected by the alphabet size, σ. This can be explained by our implementation. For the hybrid approach with a suffix tree, we used the hash table class from the JAVA collection API, to keep track of transitions coming out of each node. The hash table has an O(1) average retrieval time and has a significantly smaller memory footprint compared to reference arrays. Although we could achieve O(1) worst case retrieval time using a reference array, the higher memory requirement of each node would eventually offset the benefit, especially if σ is large. On the other hand, the dynamic programming approach does not have dependency on the alphabet size, because an integer array is used to store the edit distances.

However, this does not imply that alphabet size did not play a role in our suffix tree implementation. When the input text is very long and the alphabet size is small, such as in the case of the DNA sequence, the resulting suffix tree is deeper. That translates to more tree nodes, longer construction time, as well as greater memory consumption. The negative impact of a smaller alphabet did not manifest itself in our results because our hardware platform was not pushed to the limit in these experiments.

7.4.2 Memory Management Issue

The $O(mn)$ space limitation of the dynamic programming approach presents a memory management problem. For short strings, the $O(mn)$ space requirement is negligible. However, the memory requirement increases rapidly as m and n increase. For example, for a small text of 1MB in size, if the pattern we are trying to match is 50K in length, the

dynamic programming table size equates to 1MB * 5K = 5GB. Since most computers today are equipped with one to two GB of memory, this leads to virtual memory management issues such as thrashing.

To avoid this problem, we enhanced our dynamic programming algorithm to use an integer array of only $2n$, and we improved the d-path table for the hybrid algorithm to use a character array of only $2(m + n)$. In either approach, we search the last row of the tables to locate the ending positions of the approximate matches. As a result, we are able to process longer strings while avoiding (delaying) the on set of thrashing. This allowed us to conduct experiments with a longer pattern length m and larger k.

7.4.3 Experiment Conclusion

The hybrid approach performs well when k is small. Fortunately, in practice, applications such as DNA sequence matching, voice recognition, and error corrections are limited to a low level of error. Nonetheless, it is important to recognize that given its strengths, the use of a suffix tree is only appropriate when the right conditions are present. These conditions include:

1. When k is small,
2. When m is large, and/or
3. When the combination of mn is not conducive for the pure dynamic programming approach.

Chapter 8

CONCLUSIONS

8.1 Research Results

The applications of approximate string matching algorithms are ubiquitous in our daily lives even though their presence is not always immediately obvious. Rapid growth of data volume in our lives and advancement in the sciences only exemplify the importance of string algorithms in the foreseeable future. Although string matching algorithms have been well studied and researched in the past few decades, there continues to be breakthroughs and improvements to achieve faster and better results. This thesis aimed to introduce the concept of an approximate string matching algorithm and explore some of the advanced algorithms. We covered a wide range of topics from exact string matching to approximate string matching. We examined Ukkonen's linear time suffix tree construction and implemented a hybrid dynamic programming algorithm using a suffix tree. We conducted a series of experiments using two of the algorithms discussed and analyzed the results.

8.2 Experiment Results

Our empirical data demonstrated the effectiveness of the *lca* extension of a suffix tree and how it can be used to augment regular dynamic programming in the k-difference problem. It also showed how hybrid dynamic programming is very much susceptible to changes in k. While dynamic programming works well for small to medium text size and pattern length, it is also ideal as k increases. On the other hand, the hybrid dynamic

programming approach has an advantage when the string and pattern are long. It is generally ideal for very large text strings that can be preprocessed for subsequent searches.

8.3 Future Work

Until recently, the construction of a suffix array was derived from its corresponding suffix tree in order to achieve $O(n)$ worst-case time. This requires a suffix tree be constructed before the suffix array and is a major restriction. With the newly developed $O(n)$ time suffix array construction and $O(n)$ time *lcp* computation, the use of suffix arrays is expected to become commonplace in many areas. Although we have successfully tackled the construction of a suffix array and the *lcp* extension in linear time, our focus was on suffix trees, the *lca* extension and the *lce* extension. We fell short of exploring suffix arrays in depth. Our future plans include implementing search algorithms using suffix arrays as described in [Gusfield97], investigating generalized suffix arrays for multiple strings, and computing the *lca* of two suffices. Other interesting topics include the persisting and compression of suffix trees and suffix arrays. Cache obliviousness has also been mentioned in several research papers. Finally, we would like to solve the k-difference problem with a suffix array and compare the results with experiment results in this work.

REFERENCES

Print Publications:

[Amir00]
Amir, A., M. Lewenstein, and E. Porat, "Faster Algorithms for String Matching with k Mismatches," Proceedings of the 11th ACM-SIAM Symposium on Discrete Algorithms (2000), pp. 794-803.

[Baeza92]
Baeza-Yates, R. A. and G. H. Gonnet., "A New Approach to Text Searching," Communications of the Association for Computing Machinery, Vol. 35, No. 10 (October 1992), pp. 74-82.

[Boyer77]
Boyer, R., and J. Moore, "A Fast String Searching Algorithm," Communications of the Association for Computing Machinery, Vol. 20, No. 10 (1977), pp. 762-772.

[Chang94]
Chang, W. and E. Lawler, "Sublinear Approximate String Matching and Biological Applications," Algorithmica, Vol. 12, No. 4-5 (October 1994), pp. 327-344.

[Chaudhuri03]
Chaudhuri, S., K. Ganjam, V. Ganti, and R. Motwani, "Robust and Efficient Fuzzy Match for Online Data Cleaning," Microsoft Research and Stanford University, 2003.

[Cole98]
Cole, R. and R. Hariharan, "Approximate string matching: A simpler faster algorithm," SODA: ACM-SIAM Symposium on Discrete Algorithms (1998), pp. 463-472.

[Cormen98]
Cormen, T. H., C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, Second Edition, The MIT Press, Cambridge, and McGraw-Hill Book Company, London, 2002.

[Gusfield97]
Gusfield, D., Algorithms on Strings Trees, and Sequences - Computer Science and Computational Biology, Cambridge, New York, 1997.

[Hall80]
Hall, P. A. V. and G. Dowling, "Approximate String Matching," Computing Surveys, Vol. 12, No. 4 (December 1980), pp. 381-402.

[Jokinen96]
Jokinen, P., J. Tarhio, and E. Ukkonen, "A Comparison of Approximate String Algorithms," Software - Practice and Experience, Vol. 26, Issue 12 (December 1996), pp. 1439-1458.

[Karkkainen02]
Karkkainen, J. and S. Burkhardt, "One-Gapped q-Gram Filters for Levenshtein Distance," Center for Bioinformatics – Saarland University, Saarbrucken, 2002.

[Karkkainen03]
Karkkainen, J. and P. Sanders, "Simple Linear Work Suffix Array Construction," Proceedings of the 13th International Conference on Automata, Languages and Programming, Vol. 2719 of LNCS (2003), Springer-Verlag, pp. 943-955.

[Kasai01]
Kasai T., G. Lee, H. Arimura, S. Arikawa, and K. Park, "Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications," Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching, Vol. 2089 (2001), A. Amir and G. M. Landau, Springer-Verlag, Berlin Heidelberg, pp. 181-192.

[Keng05]
Keng, L., "A Survey of String Matching Algorithms," Graduate Research Paper, Department of Computer and Information Sciences, University of North Florida, Florida, 2005.

[Knuth77]
Knuth, D. and J. Morris, and V. Pratt, "Fast pattern matching in strings," SIAM Journal on Computing, Vol. 6, No. 1 (1977), pp. 323-350.

[Manber93]
Manber, U. and G. Myers, "Suffix arrays: A new method for on-line string searches," SIAM Journal on Computing, Vol. 22, No. 5 (1993), pp. 935-948.

[McCreight76]
McCreight E., "A Space-Economical Suffix Tree Construction Algorithm," Journal of the Association for Computing Machinery, Vol. 23, No. 2 (April 1976), pp. 262-272.

[Navarro98a]
Navarro, G., "Approximate Text Search," PhD. dissertation, Department of Computer Science – University of Chile, Santiago, 1998.

[Navarro98b]
Navarro, G. and R. Baeza-Yates, "Improving an Algorithm for Approximate Pattern Matching," Algorithmica, Vol. 30, No. 4 (1998), pp. 473-502.

[Navarro00]
Navarro, G. E. Sutinen, and J. Tarhio, "Indexing Text with Approximate q-grams," Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching, (2000).

[Navarro01]
Navarro, G., "A Guided Tour to Approximate String Matching," Association for Computing Machinery Computing Surveys, Vol. 33, No. 1 (March 2001).

[Nelson96]
Nelson, M., "Fast String Searching With Suffix Trees," Dr. Dobb's Journal, August 1996.

[Seller80]
Seller P., "On The Theory and Computation of Evolutionary Distances: Pattern Recognition," Journal of Algorithms, Vol. 1 (1980), pp. 359-373.

[Stephen94]
Stephen, G., String Searching Algorithms, World Scientific, Gwynedd, 1994.

[Ukkonen83]
Ukkonen, E., "Algorithms for Approximate String Matching," Proceedings of the International Conference Foundations of Computation Theory, Vol. 158 (1983).

[Ukkonen92]
Ukkonen, E., "Approximate String Matching with q-grams and Maximal Matches," Theoretical Computer Science, Vol. 92, Issue 1 (1992), pp. 191-211 .

[Ukkonen93a]
Ukkonen, E., "Approximate String Matching over Suffix Trees," Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching, 1993.

[Ukkonen93b]
Ukkonen, E., "Approximate String Matching with Suffix Automata," Algorithmica, Vol. 10, No. 5 (1993), pp. 353-364.

[Weiner73]
Weiner P., "Linear pattern matching algorithms," IEEE 14th Annual Symposium on Switching and Automata Theory, Conference Record (1973), pp. 1-11.

[Weiss02]
Weiss, M. A., Data Structures & Problem Solving using JAVA, Second Edition, Addison Wesley, Boston, 2002 .

[Wu92]
Wu, S., U. Manber., "Fast Text Searching Allowing Errors," Communications of the Association for Computing Machinery, Vol. 35, No. 10 (1992), pp. 83-91.

Electronic Sources:

[Allison06]
Allison, L., "Suffix Trees,"
http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Tree/Suffix/, last accessed October 23, 2006.

[GenBank06]
GenBank FTP Site, ftp://ftp.ncbi.nih.gov/genbank/, last accessed November 14, 2006.

[GreatBooks06]
Great Books and Classics, http://www.grtbooks.com/, last accessed November 14, 2006.

[Gilleland06]
Gilleland, M., Merriam Park Software, "Levenshtein Distance, in Three Flavors,"
http://www.merriampark.com/ld.htm, last accessed October 23, 2006.

[Lewis06]
Lewis, C., "Approximate Matching with Suffix Trees,"
http://homepage.usask.ca/~ctl271/810/approximate_matching.shtml, last accessed November 12, 2006.

[Moore77]
Moore, J. S., "The Boyer-Moore Fast String Searching Algorithm,"
http://www.cs.utexas.edu/users/moore/best-ideas/string-searching/index.html, last accessed October 24, 2006.

[Muhammad06]
Muhammad, R. B., "Boyer-Moore Algorithm,"
http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/StringMatch/boyerMoore.htm, last accessed October 24, 2006.

[NCBI06]
National Center for Biotechnology information FTP Site,
http://www.ncbi.nlm.nih.gov/Ftp/, last accessed November 14, 2006.

[Rouchka06]
Rouchka, E., "Dynamic Programming,"
http://www.sbc.su.se/~pjk/molbioinfo2001/dynprog/dynamic.html, last accessed November 12, 2006.

[Tsadok06]
Tsadok, D. and S. Yono, "ANSI C implementation of a Suffix Tree,"
http://mila.cs.technion.ac.il/~yona/suffix_tree/, last accessed October 23, 2006.

[Ukkonen05]
Ukkonen, E., "Suffix tree and suffix array techniques for pattern analysis in strings," http://www.cs.helsinki.fi/u/ukkonen/Erice2005.ppt, October 2005, last accessed October 24, 2006.

APPENDIX A

Glossary

$\sum$ = alphabet of a finite set of symbols. $|\sum| = \sigma$

T = a string of text derived from $\sum$. $|T| = n$

P = a string of pattern derived from $\sum$. $|P| = m$ where $m <= n$.

k = the maximum number of errors allowed

α = the error level = k / m

$d()$ = the distance function

VITA

Leng Hui Keng graduated from Florida State University with a Bachelor of Science degree in Management Information Systems in 1997. Since 2001, Leng has been enrolled in the School of Computing at the University of North Florida. With Professor Yap Siong Chua as his graduate thesis adviser, Leng expects to receive a Master of Science in Computer and Information Sciences from the University of North Florida in December of 2006. Leng is currently employed at Merrill Lynch as an assistant vice president responsible for project management and application development. Leng has more than ten years of experience in Web and enterprise application development, and is a Microsoft Certified System Developer. Leng expects to earn his Project Management Professional certification in early 2007.

Leng continues his quest for knowledge outside of school and work. Leng is proficient in database design, JAVA, and various Microsoft programming languages. He is a fan of SuSE Linux and Fedora Linux operating systems. He holds a Stellent IBPM technical certification in imaging solutions.

Leng came to the United States at the age of 18. An ethnic Chinese Malaysian, Leng speaks three languages and two dialects. Leng enjoys an occasional hike in the wild with his wife and their dog. After taking a break to complete his master's thesis, he intends to return to teaching at a local martial arts school as a voluntary instructor.