DOKUZ EYLÜL UNIVERSITY
GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

SUFFIX TREE INDEXING FOR MUSIC INFORMATION RETRIEVAL

by
Gıyasettin ÖZCAN

March, 2008
İZMİR

SUFFIX TREE INDEXING FOR MUSIC INFORMATION RETRIEVAL

A Thesis Submitted to the
Graduate School of Natural and Applied Sciences of Dokuz Eylul University
In Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy in Computer Engineering, Computer Engineering Program

by
Gıyasettin ÖZCAN

March, 2008
İZMİR

Ph.D. THESIS EXAMINATION RESULT FORM

We have read the thesis entitled **"SUFFIX TREE INDEXING FOR MUSICAL INFORMATION RETRIEVAL"** completed by **GIYASETTİN ÖZCAN** under supervision of **ASSISTANT PROF. DR. ADİL ALPKOÇAK** and we certify that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Doctor of Philosophy.

Assistant Prof. Dr. Adil ALPKOÇAK

Supervisor

Assistant Prof. Dr. Damla KUNTALP

Committee Member

Instructor Dr. M. Kemal ŞİŞ

Committee Member

Professor Dr. Alp KUT

Jury Member

Professor Dr. Şaban EREN

Jury Member

Prof. Dr. Cahit HELVACI

Director

Graduate School of Natural and Applied Sciences

ACKNOWLEDGMENTS

First of all, I would like to express my sincere appreciation to my advisor, Assistant Prof. Dr. Adil ALPKOÇAK, for his strong support, encouragement, patience, valuable insights. In addition to bringing this research work to a successful completion, he also contributed in all aspects of my academic life. During this time, he devoted his time and energy to improve this thesis despite his busy schedule.

I extend my thanks to the members of my committee, Prof. Dr. Yetkin ÖZER, Instructor Dr. M. Kemal ŞİŞ, and Asst. Prof. Dr Damla KUNTALP for their useful comments and suggestions during my Ph.D. study.

During this study, I have obtained valuable suggestions and help from Musicology department. I would like to thank to Dr. Cihan Işıkhan for his knowledge and contributions.

I thank all my friends and professors during my study;. Especially Semih UTKU, Hulusi Baysal, Tolga Berber, Zeki Yetgin and Taner Danışman.

I need to remark valuable suggestions and proofreading's of Dr. Osman S. ÜNSAL. He is more than a friend for me.

I owe a special debt of gratitude to my parents, Naci and Necla ÖZCAN. I would not have been able to get this far without their constant support and encouragement. Finally I need to mention thanks to my wife Özlem for her patience, courage, recommendation, and love. We shared so much things during this time.

Gıyasettin ÖZCAN

SUFFIX TREE INDEXING FOR MUSIC INFORMATION RETRIEVAL

ABSTRACT

This thesis intended for fast and reliable data retrieval from music databases. It introduces new data reduction and indexing approaches for both polyphonic and monophonic music sequences.

The study contributes to the literature from three aspects. These are data reduction, suffix tree indexing and tree alignment on external memory. In terms of data reduction, we present a new melody extraction approach for polyphonic music sequences. The new melody extraction approach considers the pitch histogram, and entropy of music sequences. Consequently, accompany channels of the MIDI music sequences are determined for data reduction. In terms of indexing, we present a new suffix tree construction approach for streaming music sequences. Current suffix tree construction algorithms have leaks about indexing music sequences. Hence, we adapted the physical structure of suffix trees for music notes. At last, we consider balance and alignment of suffix trees. In music, alphabet size of music is large. Therefore, we present clustering of music sequence. Therefore each sequence cluster can be indexed by a separate suffix tree to balance the tree.

Both our melody extraction and suffix tree construction approaches are tested in detail and discussed. Our evaluation metrics are based on cognition, mathematical proofs and simulations. Experimental results showed that our approaches outperforms.

Keywords : Music Information Retrieval, MIDI, Melody Extraction, Clustering, Time Series Indexing, Online Suffix Tree Construction, Streaming Sequences.

.

MÜZİKSEL BİLGİ ERİŞİM SİSTEMLERİNDE SONEK AĞACI İLE DİZİNLEME

ÖZ

Bu tez çalışması, müziksel veri tabanlarından hızlı ve güvenli veri erişimi hedeflemiştir. Bu amaçla gerek tek sesli, gerekse çok sesli müzik dosyalarında veri indirgeme ve dizinleme yaklaşımları önermiştir.

Çalışmanın literatüre katkısı üç alt konudandır. Bunlar veri indirgeme, sonek ağacıyla dizinleme ve ağacın dışsal bellekte yerleşimi. Veri indirgeme sürecini temin etmek amacıyla yeni bir ezgi çıkarım algoritmaları önermekteyiz. Geliştirdiğimiz ezgi çıkarım algoritması nota perdelerinin histogram ve entropisini dikkate almaktadır. Süreç sonunda ezgiye katkıda bulunmadığı tespit edilen notalar veri setinden atılmaktadır. Dizinleme açısından ise akışkan müziksel nota serilerinin dizinlenmesini sağlayacak yeni bir sonek ağacı önermekteyiz. Gözlemlerimize göre mevcut sonek ağaçları müzik verilerini dizinlemek amacıyla tasarlanmamıştır. Bu eksikliği gidermek amacıyla sonek ağacının fiziksel yapısı, müziğe göre uyarlanmıştır. En son olarak sonek ağacının dengesiz yapısı ve belleğe yerleşimi irdelenmiştir. Daha açık bir ifade iler dışsal belleğe erişimi azaltmak için müziksel verilerin dizinlenmeden önce sınıflandırılması önerilmiştir. Böylece her bir sınıfa ait müzik verileri ayrı bir ağaçta dizinlenecektir.

Gerek melodi çıkarma, gerekse sonek ağacı inşasına ilişkin yaklaşımları detaylı şekilde test edilmiş ve tartışılmıştır. Deneylerin değerlendirilmesi için müzik kulağı, matematiksel ispatlar ve simulasyon kullanılmıştır.

Anahtar sözcükler Müziksel Bilgi Erişim, MIDI, Ezgi Ayıklama, Kümelendirme, Zaman Serilerinin Dizinlenmesi, Eşzamanlı Sonek Ağacı oluşturma, Akışkan Dokumanlar.

ABBREVIATIONS

MIR	Music Information Retrieval
MIDI	Musical Instrument Digital Interface
DNA	Deoxyribonucleic acid
BM	Boyer Moore String Matching Algorithm
KMP	Knuth Morris Pratt String Matching Algorithm
Shift-OR	Shift-OR String Matching Algorithm
GST	Generalized Suffix Tree
OGST	Online Generalized Suffix Tree
FIFO	First In First Out
LRU	Least Recently Used
PAAL	Parent Address Appended List

CONTENTS

Page

CHAPTER ONE

INTRODUCTION

1.1 Motivation

In the last two decades, storage capacity of the computers has increased drastically. Accordingly, computers can store multimedia applications such as music and video. Besides, network communication on the net has increased the importance of computers on multimedia. Especially commercial multimedia companies have interested in storage and retrieval abilities of computers.

Storing huge amount of music documents triggers new problems to be solved. In the last 20 years, disk access time keeps constant over this time. In fact, storage devices include mechanical devices and situation limits enhancements on disk access (Salzberg, 1998). In order to overtake the pitfall, software based solutions have been proposed. Concretely, information retrieval techniques remedy the drawbacks of disk access.

There are varieties of information retrieval techniques to handle large data sets. Most commonly used ones are effective buffering on memory, indexing, data reduction, data transformation, and fast string processing (Baeza-Yates and Neto, 1999). However, there is no unique retrieval strategy; instead each application may have specific properties and need a different indexing, or buffering methodology.

Currently, music databases are very common and people demand fast access from these databases. For instance, musicologist, students, art lovers, businessmen and even lawyers have tendency to access to music databases for different reasons. Here we briefly explain the fundamental reasons:

• Musicologists want to analyze current music pieces from a large music library. Such analysis may require complicated queries. For instance, a query may include C major music sequence using instruments violin, flute and harp. Later on, the

musicologist may want to do analysis, comparisons on the query results. Such process on a large database should ensure satisfactory retrieval speed.

• Students need to learn music cognition facts. As a result, computers and music databases are educational tools for students. Displaying music pieces for demanded characteristics is appreciated by students. Therefore complex queries from a large database are indispensable.

• Art lovers want to find their favorite music fragments for listening. In some cases, they may know what they are looking. For instance name and the artist of a music file. However, they may want to query music by whistling or humming. Moreover, they may search for similar music pieces to a certain file. In fact, searching similar music files from a database has a high computation cost.

• Commercial firms want to sell their new musical products to the customers. Therefore, they need to present the best service. For companies, customer satisfaction and easy access to the product are indispensable. Also they need to present effective and fast interfaces to increases their sales.

• Music similarity can be fairly used in ethics. Every year huge number of new musical product put into the music market, it is very hard to catch all cheating events. Moreover, music cognition is subjective; decision of cheating may depend on the listener. Hence, it is difficult to prove that a new music piece violate copyright law. However, computational musicology can solve those pitfalls satisfactorily, if search on music databases can be processed in a fair time. In this respect, faster query search algorithms are able to present results in acceptable time. Moreover, computer based similarity searches are objective since similarity rules are determined before similarity search starts. Therefore prejudice becomes impossible.

1.2 Music Terminology

In this section, we briefly explain music and its fundamental terms. George Sand said that goal of music was enthusiasm. He adds that none of the remaining arts could generate such exalted feeling in human sense (Feridunoğlu, 2004). Basic elements of the music are rhythm, melody and harmony. Rhythm is the division of time into equal or non-equal intervals. In music, consecutive rhythm events lead to regularity and organization. Meanwhile melody is the musical idea, which influences the listener by its own letters. Finally harmony means accordance of different sounds between time intervals.

Music alphabet is composed of notes. (Lemstrom, 2000) explains the note as follows: “When a musical instrument is played, it evokes a tone sensation in listener. Tone sensation is comprised of attributes salience, pitch, timbre, onset time and duration. The written instructions to play the tone are called a note”. One of the most effective properties of a note is pitch, and it is the perceived frequency of a note.

Music can be represented as sequence of notes. This fact can be illustrated by MIDI music format. MIDI is an acronym for Musical Instrument Digital Interface and it is the common protocol which enables communication between digital music devices and computers (General MIDI – Wikipedia, n.d.). In MIDI, music format is composed of note sequences and some meta data information. While notes keep pitch, duration and volume information, meta data handles general information of a sequences such as tempo, metronome ticks or instrument.

Music can be either in monophonic or polyphonic form. Polyphony is the occurrence of multiple notes at a time (Temperley, 2001). In other words, polyphony permits appearance of several notes simultaneously. In contrast, monophonic music has strict constraints, where there can be only one note at a time to be played.

The pitch distance between any two notes is named as interval and the smallest interval in western music is described as semitone. In western music there exist 12

semitones and it is called as octave. Interestingly, music can be represented by 12 notes; which is an octave. When difference between two pitches is 12, then the pitches share the same octave. Hence it is possible to represent music by an octave. Recall that MIDI music format has 128 pitch frequencies; so it can represent 10 different octaves

1.3 Music Information Studies

Common MIR Studies can be separated into five different sub fields: These are preprocessing, indexation, string matching, extraction, and interface design.

• Preprocessing: Natural music is based on signals. The instrument that is playing a note can be detected by its frequency. The signal processing applications tries to convert sound into music files. At the same time, compression algorithms can be necessary, since multimedia applications take high amount of space.

• Indexation: Indexing is an alternative way for fast data search on large databases. In general, a music database may contain thousands or even millions of music files. Search and retrieval processes on large music databases can be very slow without optimization. For instance, music files can be clustered for fast search. Concretely, music sequences with common properties can be can be grouped. In addition, tree and hashing index methods are possible.

• String Matching: Music similarity is a common search method in MIR applications. User submits a sequence of notes as a query, and query is matched to the all sequences in the databases. Since both query and database are represented by string sequences, string matching algorithms can be implemented. String matching problem can be divided into exact matching and approximate matching. Common algorithms are Boyer-Moore (Boyer and Moore, 1977), Knuth-Morris-Pratt (Knuth et. Al., 1977) and bit parallel string matching (GusfieldBook, 1997). String matching algorithms are very important in bioinformatics and search engines.

• Extraction: Music is a combination of melody and accompanies notes. In general, human perception focuses on melody of music and memorizes easily. Because of this fact, most of the search operations on music databases interests in melody. Based on this fact, melody extraction is a common study topic in MIR (Uitdenbogert and Zobel, 1998). For instance, cognitive studies denotes that memory is generally takes place in higher pitches. In addition, harmony, key, or rhythm information of a music file can be extracted using artificial intelligence techniques. In order to do this, musical rules and facts should be taught to the computers.

• Interface design: Improvements on computer hardware devices are fascinating. For instance hardware devices can handle large amount of music data. Similarly, file transfer rate on internet has increased drastically in the last ten years. In parallel to hardware evolution, new software components are highly demanded. Recently, music has been represented in different digital format. Processing on music databases become common. However, recent software's do not satisfy user demands since current interface application are not mature yet. Therefore interface design on musical database is a hot research topic.

1.4 Contributions

Contribution of this thesis is threefold. First, we present a new melody extraction approach which will be used for data reduction in music databases. Our second contribution is about sequence indexing with online suffix trees. In this respect, we modify the physical node representation of Ukkonen's online suffix tree. Finally, we present a sequence clustering approach to index sequences with multiple suffix trees. We present that multiple suffix trees can be efficiently used if the alphabet size of sequence database is large.

In terms of melody extraction, the thesis analyzes cognitive studies on the field and introduces a new approach based on pitch histogram and cognition. Our survey on early melody extraction algorithms showed that early studies mainly focused on cognitive studies. However, those studies did not consider pitch histogram of MIDI

channels deeply. We present that the performance of the melody extraction can be extended by considering channel clustering based on pitch histogram

In contrast to early studies, our approach is able to select multiple melody channels from a sequence. Depending on the pitch histogram of MIDI channels, we present a clustering approach and determine total melody channels of music.

In the second phase, we consider indexing music sequences. We present that fast sequence search on music databases can be ensured by suffix trees. We show that advantage of a suffix tree comes into prominence when fast subsequence query search on a large database is the fundamental requirement. Also we denote that online suffix trees are very important for streaming music sequences. As a result, we present an approach which enhances the performance of suffix trees for music sequences.

In contrast to suffix trees, classic string matching techniques fail when the database is very large. For a simple search, all elements of the database should be read and compared with query. Such cost cannot be paid by large databases. Hence, tree structures are preferred. In literature, there exist many different sequence indexing methods such as Suffix trees, suffix arrays, string-b trees, hashing, etc.. However, the fastest query response can be yielded by suffix trees (Ferragina et. Al, 1998), (Farach et. al, 1998), (Manber and Myers, 1993).

Suffix trees are composed of nodes and edges. While nodes represent a unique suffix in the sequence database, and edges connect the nodes. Here, physical alignment of the tree nodes can cause deep impact over tree construction cost. It is possible to align the sibling nodes of the tree inside an array or in a linked list. Even more, sibling nodes can be aligned by hash or tree.

We present that suffix trees have three common drawbacks of suffix trees. These are poor memory locality, high space consumption and unbalanced tree structure. Because of these three drawbacks, suffix tree construction is difficult.

Poor memory locality is the result of random ordered node generation of suffix tree. As a result of poor memory locality, nodes of a common path are aligned into different pages of the disk. When the path is traversed during construction or search, so many disk pages should be fetched coerciblely.

Our contribution on suffix trees is about its physical node representation. The new node representation ensures fast access to child and parent nodes. Also our node model is space efficient. In other words space and page requirement of online suffix trees is acceptable for our model.

Another contribution of the thesis is that, we determine the frequently accessed nodes of the suffix tree. As a result, we can align those nodes in a special way to accelerate their retrieval time. We present that node access to frequently accessed nodes of the tree can be estimated by alphabet properties.

In this study, we insert MIDI music sequences into the suffix tree. Since MIDI alphabet can return 128 different pitches, MIDI alphabet has 128 letters too. When compared with other alphabets, MIDI alphabet has medium size. For instance DNA has only four letters and Turkish alphabet contains 29. On the other hand some Time Series applications may contain thousands of letters (Keogh and Kasetty, 2002). It is not to see that different physical node representations yield different performance for each case.

Final contribution of the thesis is about sequence clustering be due to densely populated pitches of a sequence. Since MIDI alphabet is large, a MIDI sequence may contain few of possible pitches. For instance, pitch average of accompany channels are decently low and they rarely contain high-pitch nodes. Because of this fact, indexing all sequences by one suffix tree may not yield best performance. Hence we present multiple suffix tree construction approach.

In this study, each of our proposals is supported by experimentation. Both melody extraction and suffix tree construction approaches are tested on selected MIDI datasets. Musicology department has contributed on MIDI dataset generation.

1.5 Thesis Organization

The thesis consists of five chapters.

Chapter 2 presents our on Melody Extraction approach. Initially, we present the preliminaries and basic concepts of Melody Extraction. Then, we introduce the common pitfalls of early melody algorithms. Next, we explain the importance of the pitch histogram over melody extraction. Finally, we present a new melody extraction approach which not only considers cognitive aspects of music, but also pitch histogram and hierarchical clustering of music channels.

Chapter 3 ensures a bridge between the Music Information Retrieval and Time Series Similarity studies. Although MIR is a young research topic, Time Series Studies have long history and mature experience on data management and indexing. Hence, learning the early experiences from another research field should ease the problem in MIR. In this chapter, we also analyzed the pros and cons of indexing and similarity approaches. Hence we can route our study destination.

In Chapter 4, we introduce our indexing approach. That is to say, we index streaming music sequences by an online suffix tree. Suffix tree introduces fast sequence search on large databases. However, suffix tree construction on external memory has common pitfalls. These are (1) poor memory locality, (2) high space consumption and (3) unbalanced tree structures. Moreover music sequences have an important constraint: Every day, database should be updated with new music sequences. In order to solve the problems, we introduce a new online suffix tree construction approach for streaming music sequences.

In Chapter 5, we consider poor memory locality on suffix trees in detail. Here we analyze the factors causing poor memory locality. The chapter analyzes the cost of node swapping on the tree. While node swapping solves the poor memory locality problem, it is very expensive. Hence it is necessary to present a trade-off for using node swapping. In addition, we introduce the contribution of multiple suffix tree construction on a database. In this way, suffix trees can be more balanced.

Finally, the conclusions are introduced in Chapter 6. The chapter also presents key contributions and fundamental findings of this thesis. Also, the thesis looks at the future of MIR research, data reduction, indexing and external suffix tree construction.

CHAPTER TWO

TIME SERIES SIMILARITY AND INDEXING

2.1 Introduction

Time Series are a sequence of values, ordered in time. Almost all temporal events in the nature can be seemed as Time Series Implementation. Music sequences are not an exception. During a certain time and order, notes of music start playing and stops. Occurrence order of the notes between limited time interval leads to Time Series implementation. This fact is very important since Time Series Similarity is a deep and mature research field in Computer Science. As a result, we can make use of early experience from the field and introduce new enhancements.

Studies on Time Series Similarity subject try to understand the underlying theory of the successive data points(Agrawal, 1993), (Keogh et. al., 2005). They attempt to determine which dynamics generate the structure. However, most of the Time Series implementations cannot be formulated. For instance, there is no magic formula to estimate climate changes over a year. Instead, scientists estimate future by early experiences. Another example is daily stock market records. Daily values of stock market are reported in a 2-d graphics. In order to estimate the future, economist seeks for similar 2-d events occurred in the past. In Figure 2.1-a, stock behaviors' of Arcelik is observed in 2005. Interestingly, the stock plotted similar graphics in 2006 and shown in Figure 2.1-b (cnnturk.com, 2007).

2.2 Time Series Studies

Searching similar events on a Time Series database is one of the goals of computer science. This is a very difficult job since (1) all early experiences should be stored inside a database, (2) size of the most databases overtakes terabytes, (3) database should be eligible to extension for new experiences, and (4) database should handle approximate matching (Keogh and Kasetty, 2002).

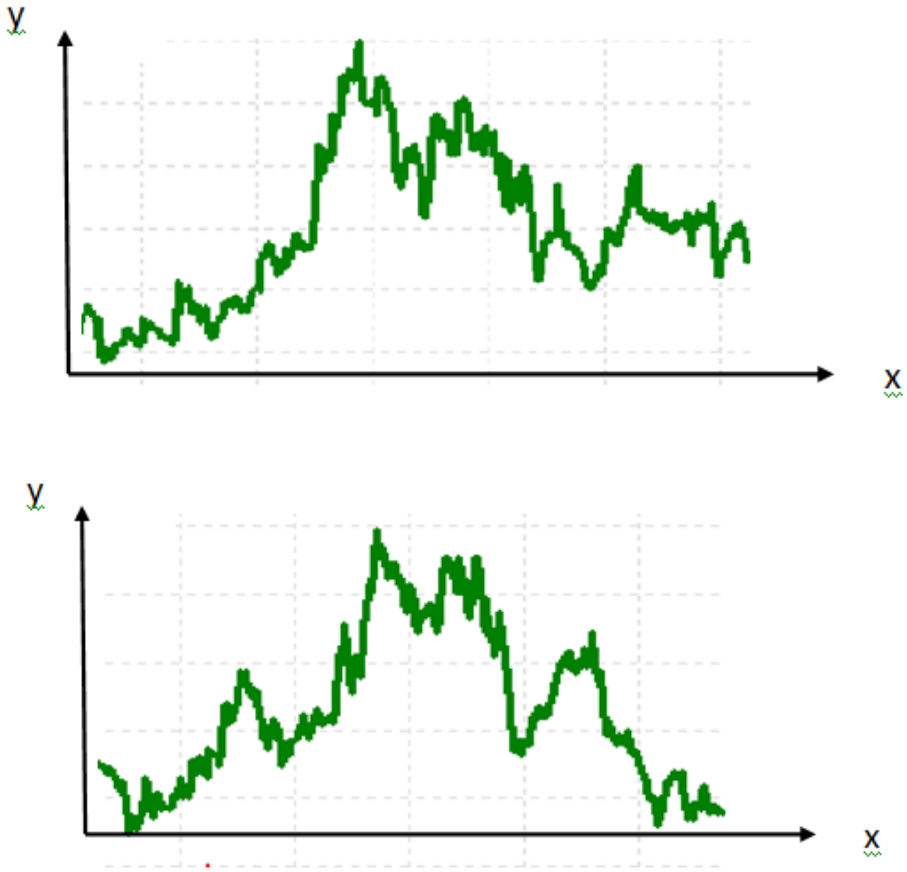


Figure 2.1 Time Series Analyis of two similar events; a-) Values changes of a stock after Jan 2005 b-) Values changes of a stock after Jan. 2006. (Arcelik-Cnnturk.com, n.d.)

String matching operations on very long strings are time consuming. In order to handle large datasets, Time Series studies consider data reduction, segmentation, and indexing (Keogh and Kasetty, 2002).

2.2.1 Data Reduction

Goal of the data reduction is to represent data with less number of symbols without loosing core information. Some of the most popular data reduction methods are Discrete Fourier Transform, Discrete Wavelet Transform, Piecewise Linear Approximation, and Singular Value Decomposition (Shatkay, 1995), (Stollnitz, 1994), (Keogh and Kasetty, 2002).

In terms of seismologic, economic and weather data, above data reduction techniques make sense. Nonetheless, music is an exception. Fourier or similar transformations interfere music cognition. On the other hand, music introduces its own data reduction techniques. By using melody extraction algorithms, music sequences can be represented by less number of notes. A musical data reduction is shown in Figure 2.2. As in the figure, some of the notes are eliminated since they do not contribute to the melody of the music sequence. Detailed analysis of melody extraction is presented in Chapter 4.

2.2.2 Segmentation

It divides music sequence into meaningful fragments. As a result, each fragment can be processed separately (Keogh, 2001). There are three types of segmentation techniques in the literature. These are Sliding Window, Top Down and Bottom up approaches. Sliding window approach starts with the atomic fragments or points. Iteratively segment will be expanded until an error bound is encountered. Later, next free point starts to issue a new segment and do the same process. Top Down approach recursively partitions the Time Series data until some predetermined criteria has been encountered. Bottom Up approach issue smallest possible segments initially. Later on short segments are merged until some error bound has been encountered.

2.2.3 Indexing

Storing data in a clever way speeds up access time on a database (Salzberg, 1998), (Folk et. al., 1997). That is the goal of indexing. In a clever indexing mechanism, a simple search query does not scan entire data set. Instead, it will minimize scanning.

There are various indexing techniques in computer science field. The simplest approach is ordering. Similar to dictionary, all data is alphabetically ordered. On the other hand, there exist complex indexing mechanisms as well. Inverted Files, Signature Files, B-trees, Multidimensional Indexing methods and Suffix Trees are

commonly used indexing mechanisms in the field (Beckmann et. al, 1990), (Bayer and McCreight, 1972), (Gaede and Gunther, 1998).

2.3 Indexing with Multidimensional Access methods

Given a sequence S as $[s_1,s_2, \ldots,s_n]$ multidimensional structures attempt to index s_i in the i^{th} dimension. As a consequence, S can be indexed in n-dimensional index structure. The multidimensional sequences can be indexed by R-tree, R*-tree, bsp-trees, and quad-trees (Beckmann et. al, 1990), (Gaede and Gunther, 1998). In literature, R*-tree is a common technique to index multidimensional index structures. R*-trees do not index time series directly; instead the trees index user defined envelops of the time series. As shown in the Figure 2.3, algorithms determine meaningful envelops to index sequences. In the tree envelops are indexed as multidimensional points.

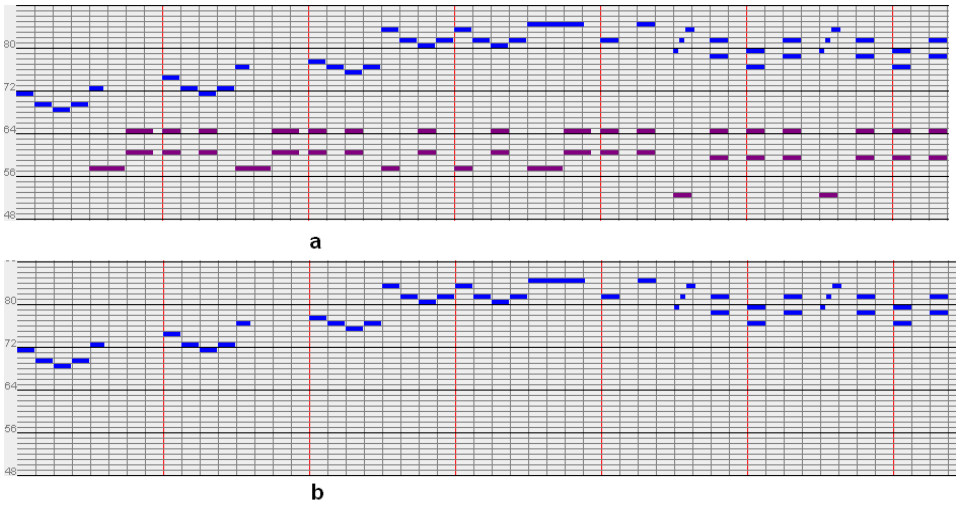


Figure 2.2 Data Reduction in a MIDI music sequences. In Figure a, all sequences are shown. In Figure b, Notes coming from the accompany channel are removed from the sequence set without reliability concerns.

The queries are assumed as a set of multidimensional points in the space. When a query hits an envelope in the tree, then there is a probability that query may take place in the database. For such cases, detailed comparisons can be made in the relevant time series sequence.

Multidimensional indexing has a common pitfall. They expose poor performance when dimensions are greater than 12. In such cases, dimensionality reduction techniques can be used for some data sets. For instance, features of the stock market data can be extracted by DFT (Shatkay, 1995). It is a fact that DFT can extract most effective coefficients of a time series. On the other hand, less effective coefficients can be dropped during indexing. Such approach really makes sense for stock data.

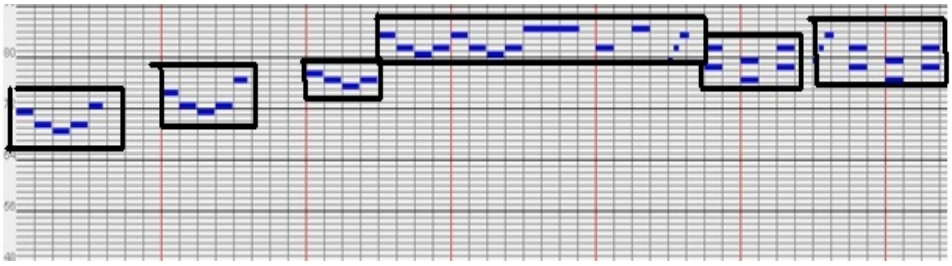


Figure 2.3 Generation of Minimum Bounding Rectangles on a sequence.

In contrast to stock data, music sequences cannot be reduced to 12 dimensions. Otherwise cognition of the music will be lost. Therefore multidimensional indexing methods fail when music sequences are considered. Moreover, inverted lists are not convenient since occurrence of each musical sequence is generally one.

2.4 Indexing with Signature files

In terms of music sequences, signature files can be tried (Jönsson, 1999). Signature files contain hashed terms from documents. The hashed terms are called signatures and used as probabilistic filters for initial text search. A signature file example is denoted in Figure 2.4. In contrast to R-tree, signature files can handle large dimensions. Nonetheless, signature files have a common pitfall. Size of a signature file is large as well. It is said that size of the signature files is around 10% of the size of the original file (Jönsson, 1999). Hence for each query, 10% of the document should be scanned. Such fact is against the definition of indexing. Recall that indexing aims to store data in a clever way to speed up access time. This cannot be achieved by scanning 10 % of a text.

Text	Simple	signature	file	example
Word Signature	0101	0011	1111	0110
Document Signature	0101 0011 110110110			

Figure 2.4 Representing sequences with signatures

Suffix trees are assumed as an alternative indexing technique for Time Series data (Huang and Yu, 1999), (Lin et. al., 2003). In fact, they can handle larger dimensions. At the same time, they do not cause a common pitfall as Signature Files yield. However, suffix trees are popular since they minimize the scan process. Because of these facts, suffix trees are well known string processing implementations. Certainly they have important drawbacks as well.

2.5 Similarity

Goal of Time Series indexing is fast data retrieval. The user inputs a query and the query is searched in the database (Chakrabarti, 2001). Here, search process is based on similarity. Therefore, it is necessary to determine what does similar means.

Definition of similar is fuzzy. Two objects are assumed similar if they have common characteristics. For instance, grass and leaves have green color. In terms of color, two objects are similar. In contrast, size of a grass and a leaf is different. Hence, two objects are not similar when we look at them from a different aspect.

It is also necessary to define the similarity ranking of objects. In some cases, we may not find absolutely similar objects. Hence, quantifying the similarity between two objects becomes necessity. For instance, in Figure 2.5, three Time Series data looks similar. However, last two series has more common properties and they are more similar to each other.

2.5.1 Edit Distance

In information theory, similarity is defined by a relevant edit distance. Edit distance computes the total process requirement to transform one sequence to other. In terms of sequence transformations, letter substitutions are the overall process costs. Let "abdcaa" and "abccaa" are two sequences. In order to transform first sequence to the second one, its third letter should be changed.

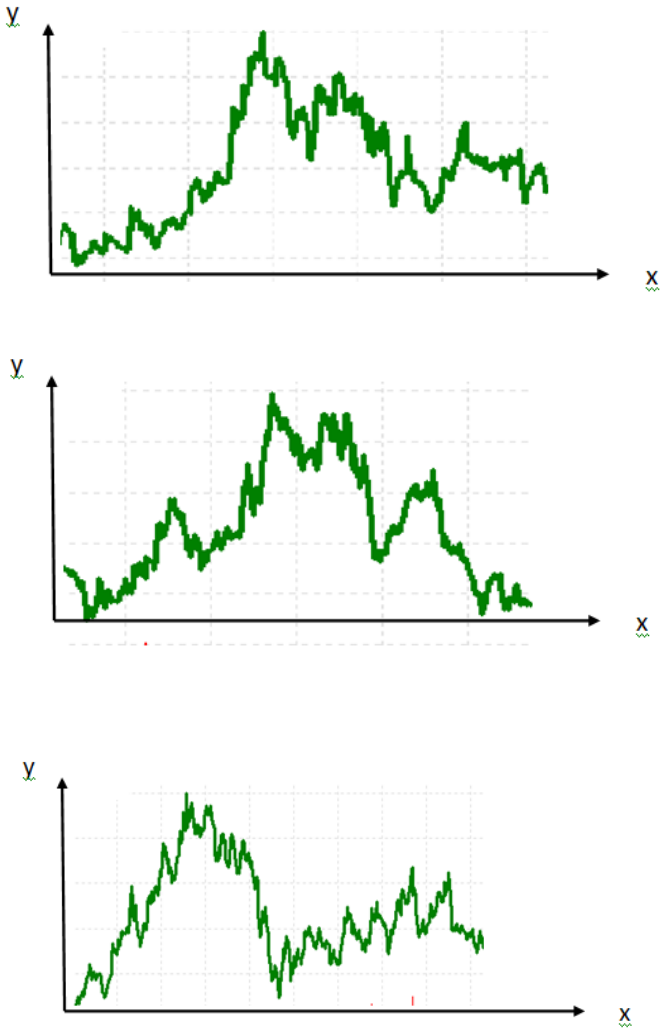


Figure 2.5 Time Series Similarity of three sequences (Arcelik-Cnnturk.com, n.d.).

There are several algorithms which define the edit distance metric. Most commonly used ones are Hamming Distance, Levenshtein distances (Navarro, 1998),

(Navarro, 2001) The Levenshtein distance between two string is determined by total letter insertion, deletion or substitution of one letter. For instance if we need to transform “money” to “core” we need to process to substitute and delete process.

2.5.2 String Matching

String matching is an alternative retrieval strategy to search similar documents from databases. While edit distance metrics introduce transformation rules, they do not present fast process times. The string matching studies ensure fast sequence search on large databases.

String matching algorithms are common in our life. For instance search engines make use of string matching technique to introduce query result to the user faster. Although, size of the searched database can be more than terabytes and the scan operations ends up in milliseconds. In addition, human genome project is solved by string matching algorithms as well. So that large human DNA can be understood for possible cures.

In order to process string matching on large sequences, classic string matching algorithms are not eligible. In fact, terabytes of net information or 3-gigabyte human DNA cannot be processed with a brute-force string matching algorithm. Using a brute force approach, similarity search on large DNA sequences will take no less than a minute. Because of this fact, new string matching algorithms are proposed.

String matching can be divided into two sections. These are exact string matching and approximate string matching. Exact string matching algorithms search exactly similar patterns of a query. In contrast, approximate string matching algorithms consider errors.

Let text, T, be a set of strings where

$$T \in \Sigma^* \text{ and } |T| = n, \tag{2.1}$$

and pattern, P, be a set of strings where

$$P \in \Sigma^* \text{ and } |P| = m. \tag{2.2}$$

To be more concrete,

$$T = T[1], T[2] \ldots T[n] \text{ and } P = P[1], P[2] \ldots P[m] \tag{2.3}$$

The exact string matching problem search the pattern in the text under the condition that

$$P[1] = T[i], P[2] = T[i+1], \ldots P[m] = T[i+m-1] \tag{2.4}$$

In contrast approximate string matching accepts limited number of errors. Instead, limited number of inequalities is accepted. If, for instance, error limit is 1, all below possibilities should be assumed as approximately similar.

$P[1] \neq T[i], P[2] = T[i+1], \ldots P[m] = T[i+m-1]$ or
$P[1] = T[i], P[2] \neq T[i+1], \ldots P[m] = T[i+m-1]$ or
.
.
.

$$P[1] = T[i], P[2] = T[i+1], \ldots P[m] \neq T[i+m-1] \tag{2.5}$$

2.6 Transforming Time Series Data into Discrete Form

During the nineties, Time Series Studies became mature for static data sets (Keogh and Kasetty, 2002). However, the situation was still new for streaming data

set. In the last decade, Time Series Studies focused on Suffix Trees and their applications.

2.6.1 Adaptive Query Processing for Time-Series Data

Huang and Yu claimed that it is less controllable by the end user when data is transformed from time domain to frequency domain (Huang and Yu, 1999). They proposed to convert time series data into discrete form, they founded equivalent strings. What they did was finding the difference between consecutive positions and defines a letter for each difference value. Their work is based on two sections. First section is the preprocess stage. Here, the time series data is transformed into strings. It is application dependent, how many strings will be used by application. They introduce "numsegment" parameter for this. In addition, "min" and "max" are bounds of changes.

Preprocess stage continues by index phase. Here authors tried a suffix tree construction method. Their suffix tree construction algorithm is based on Mc Creight suffix tree method. In addition, they used additional ID and position parameters.

2.6.2 SAX

A more popular time series, suffix tree indexing has been proposed by Li, Keogh (Keogh et. al 2005), (Lin et. al., 2003). They converted the time series into string form. Later they tried to find the surprising patterns of the strings without a prior experience. Their goal was creating an approximation of data which can fit in memory such that it can maintain essential features. .

SAX method considers surprising patterns without knowing what is surprising. In order to do this, they tried Markov models. They used a random projection method to find the most attractive motifs.. Randomly selected masks computes the similarities of substrings and assigns the similarities into collision matrices.

In their research they also presented colored bitmap of time sequences. As a result, before doing a comparison between two sequence, approximate representation of two sequences are compared. So that search time reduces.

In their study, Keogh et al believed that future of time series is beyond SAX. Also they believe that classical time series are mature at the moment; however there is still work to do for streaming time series applications.

2.7 String Matching Algorithms

Indexing data is the core part of the researches. However it does not compensate all difficulties. Recall that query results can be returned by comparison. Hence, it is necessary to mention about string matching algorithms.

2.7.1 Brute Force Text Matching

Discussions about string matching start with the brute force method. This method aligns the leftmost end of P with the leftmost T. Later, from left to right all characters are compared from left to right until mismatch character occurs or we encounter the end of pattern. This approach is very slow. Let m and n be the size of pattern and text respectively, the computation cost of the comparisons take O(mn) time in the worst case.

As an example assume that T = “abababababac” and P = “abac”. Initially brute force matching encounters that

$T[1] \neq P[1]$

As a result, matching for $T[1]$ fails. In the second step, algorithm restart comparing by $T[2]$ and $P[1]$. Since first letter of the pattern is matched, algorithm tries to match $T[3]$ and $P[2]$. Since a match occurs, third letter of the pattern is tested with $T[4]$. Nevertheless, fourth letter of the pattern does not match. While failure was

obvious in the second step, it took extra cycles to find the unmatched. Hence unnecessary comparisons would be made.

Although the problem can be solved in 10 comparisons, brute force algorithm ends up with 19 comparisons.

2.7.2 Boyer-Moore Algorithm

Boyer Moore has three clever ideas which do not take place in the brute force approach. These are bad character rule, scan from right to left and good suffix rule (Boyer and Moore, 1977). When a bad character is encountered in the text, next pattern comparison can be shifted until the bad character is avoided. In order to use bad character effectively, characters are searched from left to right.

Good suffix rule determines shifting position after a mismatch occurrence. In case a mismatch occurs, matched characters can be aligned as the suffix of the matched sequence. As a result, next available position computed by good suffix.

2.7.3 Knuth-Morris-Pratt (KMP) Algorithm

The KMP algorithm is based on preprocessing of the pattern before string matching starts up (Knuth et. al, 1977). Like the brute force method, KMP algorithm scans the text from left to right. Similar to Boyer Moore algorithm, KMP is based on preprocessing of the pattern. However, KMP matching procedure operated from left to the right. In case a mismatch occurs a preprocess table determines the number of characters to shift. It based on the common strings.

2.7.4 Shift-Or Bitwise matching algorithm

Bitwise shift-or algorithm makes use of the intrinsic parallelism of bitwise operators on the memory. In general, it yields satisfactory results when word size is less than memory word size of the machine. pattern length and alphabet size do not affect search time. An important point about shift or algorithm is that, it is adaptable to approximate string matching.

2.8 Suffix Tree Construction

Data search on large sequences is an important problem for two reasons. Firstly, it may be necessary to scan the whole database for a simple user query. Secondly, user queries on the database may occur very frequently. Hence scanning the database for each query should drown the information retrieval mechanism.

Suffix trees solve the query handling problem satisfactorily. If the sequence database is indexed by a suffix tree, query time depends on the length of the query. Hence the optimal query retrieval time will be ensured.

In order to reduce query search time, the tree indexes all possible suffixes of sequences. Given that the length of a sequence is n, then the sequence will have n different suffixes. Therefore possible n suffixes of the sequence will be indexed by the suffix tree.

Inside suffix trees, a common prefix is represented by a single node. Also hierarchical alignments of the nodes are interesting. A parent node always represents a sub sequence, which is a prefix of a sequence that is represented by its all child nodes. Such alignment strategy ensures fast subsequence search on sequence databases.

While suffix trees ensure fast access to the queries, their construction time and space consumption could be high. Because of its drawbacks, suffix trees could not become popular before seventies. Afterwards a new tree construction algorithm attracted researchers (Weiner, 1973). The algorithm ensured linear time construction of the suffix trees. In addition space requirement reduced to linear time.

Before the new century, suffix trees mainly indexed in random access memory. Nevertheless, they denoted poor performance on external memory applications. There were three common pitfalls, which reduce the performance of suffix trees.

These are poor memory locality, high space consumption, and non-balanced tree structure.

Until now, especially poor memory locality is the most important problem of suffix trees. In terms of external memory applications, disk access time is the bottleneck of the computers, since hard disks contain mechanical parts.

We believe music sequences can be indexed by suffix tree efficiently. When compared with biological data, music sequences are shorter. Hence depth of the suffix tree will be moderate for music.

Suffix trees introduce valuable options for music. For instance, dynamic sequence insertions are supported by suffix trees. So that new music albums can be updated in a suffix tree very fast.

CHAPTER THREE

MELODY EXTRACTION AS A DATA REDUCTION METHOD

Music files are mostly in polyphonic form, where multiple notes sound simultaneously. However, human have tendency to memorize only melody of the music, where melody is a linear, recognizable musical unit. In order to determine melodic lines from polyphonic music files, Melody Extraction Algorithms have been issued.

In this section, we present a new Melody Extraction approach and make experiments on MIDI file format. Depending on pitch histogram and cognitive features of music, we eliminate the MIDI channels which are potentially lack of melodic content. To do this, firstly, we determine the highest pitch line of each MIDI channels and compute pitch histogram. Next, we present an agglomerative hierarchical clustering technique, and gather the channels with similar histogram features. Depending on music cognition facts, we select best channel from each cluster as melody and discard the rest. Lastly, we implement early Melody Extraction algorithms in the reduced MIDI set.

For evaluation, we selected 31 MIDI music files. Selected files disclose different musical features such as pitch frequency, tremolo, arpeggio, glissando and rest.

3.1 Introduction

Recently music files have been converted into digital format, leading to digital music databases. Consequently, fast and reliable music retrieval algorithms have been demanded from industrial, educational and judicial communities. In order to design efficient music retrieval algorithms, interdisciplinary studies have been focused on polyphonic nature of music, where polyphony is the simultaneous sound of notes. Melody Extraction is a research field, which generates monophonic equivalent of polyphonic files, where monophony guarantees linear sequence of

notes (Meek, 2001). Hence, output of Melody Extraction takes less space in databases but contains genuine part of the music.

In 1995, (Ghias et. al., 1995) presented that percussion channel never contributes to melody. As a result, elimination of percussion channel not only enhances the relevancy of the search, but also speeds up the retrieval time. In 1998, a breakthrough paper proposed music manipulation approach. (Uitdenbogerd, 1998). Uitdenbogerd and Zobel pointed out that retrieval on monophonic music files is comparably easy, whereas dealing with polyphonic files require significant endeavor. They put forward four different techniques to generate monophonic equivalent of polyphonic files and made experiments in MIDI music files. Their first technique, Skyline Algorithm, collected the notes of a MIDI file into single MIDI channel. Thereupon, algorithm followed the highest pitch line of the note sequence as the melody. In order to keep more notes in the final output, Skyline Algorithm modified note durations. Their last three algorithms attempted to select the best MIDI channel which keeps melody (Uitdenbogerd, 1998), (Uitdenbogerd, 1999). In order to determine best channel, they presented cognitive criterions such as pitch average or entropy of a channel. Remainder of MIDI channels were entitled as accompaniment and discarded. However, all four algorithms led a main drawback; features of the music files were determining the performance of each algorithm.

In order to enhance Melody Extraction, Chai presented Revised Skyline Algorithm (Chai, 2000). Having sorted notes based on pitch level, she eliminated low pitch notes until monophony is obtained. Moreover, Chan claimed that average volume of the channel may disclose the location of melodic content (Chan, 2002). Nevertheless, none of the melody extraction algorithms overtook feature dependency. In contrast, each algorithm succeeded in a special data set. In fact, multi cultural progression of music was leading to complex cognitive rules, consequently causing obstacle against melody extraction.

In order to overtake feature dependency, we combine the Melody Extraction techniques from literature. Initially, we cluster MIDI channels, be a result of pitch

histogram. Consequently, each cluster contains MIDI channels maintaining specific histogram features. Next, we select the best channels of the clusters as melody and eliminate remaining channels from MIDI file. Finally, we implement early Melody Extraction algorithms in the reduced MIDI set. Experiments acknowledge our method; implementing Skyline over the reduced MIDI set outperforms. Moreover, a combined channel selection approach overtakes the previous channel selection algorithms.

The remainder of the paper is organized as follows; section 2 describes the definitions and related work. Section 3 presents Partial Skyline approach. Section 4 exposes experimental results and makes comparisons. Section 5 concludes the paper and gives a look to the further study on this subject.

3.2 Related work

MIDI is an acronym for Musical Instruments Digital Interface. For simplicity, musical notes are stored in 16 channels, where each MIDI channel represents an instrument. Let M be a MIDI file composed of channels. Formally,

$$M = \{ c_1 , c_2 , \ldots\ldots, c_{16} \} . \tag{3.1}$$

We assume that each channel, c_i, be a set, containing k notes. Mathematically,

$$c_i = \{ n_{i1}, n_{i2}, \ldots . n_{ik} \} \text{ where } 1 \leq i \leq 16 \tag{3.2}$$

Melodic content of the music might be distributed among channels. However, (Ghias et. al., 1995) showed that percussion channel never contained melodic information. As a result, elimination of percussion channel, c_{10}, from M did not ruin melodic robustness.

Interestingly, average frequency of percussion channel was low. Cognitive studies showed that it was not a coincidence. (Dowling, 1982), (Temperley, 2001). On the

contrary, human have tendency to memorize high frequency notes. Based on this fact, Uitdenbogerd and Zobel followed the highest pitch line of the *M*. If multiple notes occur simultaneously, they eliminated the notes exposing low frequency.

For our case, there are three important note properties: pitch, note onset time and note offset time; respectively p_{ij}, s_{ij}, and e_{ij}. Formally, we define a note as:

$$n_{ij} = \{ p_{ij}, s_{ij}, e_{ij} \} \quad 1 \leq p_{ij} \leq 128 \tag{3.3}$$

By nature, MIDI notes are sorted based on note onset time. Therefore,

$$\forall\, n_{ij}, n_{i(j+1)} \in c_i \; ; s_{ij} \leq s_{i(j+1)} \,. \tag{3.4}$$

However, there is no constraint for offset time. If subsequent note's onset is earlier than preceding note's offset, then polyphony will occur. Formal definition of polyphony is:

$$\exists\, n_{ij}, n_{i(j+1)} \in c_i \; ; e_{ij} > s_{i(j+1)} \tag{3.5}$$

3.2.1 Skyline Algorithms

(Uitdenbogerd, 1998), and (Uitdenbogerd, 1999) presented four techniques to generate monophonic equivalent of polyphonic files. Their first technique, Skyline Algorithm, collects all notes of *M* into one channel and follows the highest pitch line. In addition Skyline Algorithm manipulates note durations. Skyline Algorithm is explained in Algorithm 3.1. At the first line of the algorithm, first note is selected. Fourth line of the algorithm considers all notes that have the sane onset time. In case multiple notes have same onset time, note with maximum pitch frequency will be kept, whereas rest of notes will be eliminated (line 5-10). Therefore monophony can be ensured. On the other hand, monophony can be obtained by shortening the duration of notes. If a new note onsets before preceding note offsets, offset of

preceding note is rearranged in line 11-12. In order to illustrate the algorithm, Figure 3.1 denotes Skyline algorithm.

Although Skyline yielded impressive results, three critics have been mentioned. Firstly, manipulating the note durations can change music properties. Secondly, collecting all notes into one channel removes rest. In other words, silent intervals between notes may carry on hidden melody. To consolidate the problem, we introduce the Skyline algorithm in Figure 3.1. Lastly, we may encounter music samples where melody is maintained by low pitches. Depending on the note durations, some notes are eliminated from the set in figure 3.1-b.

Algorithm 1 Skyline

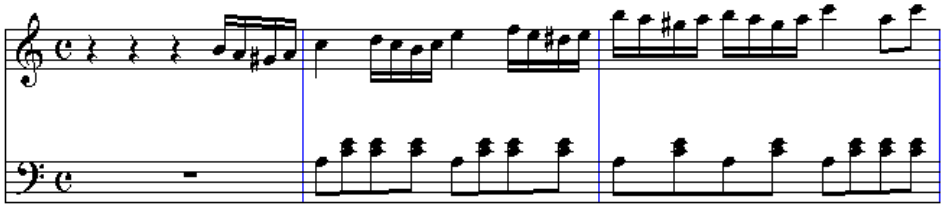
```
begin
1. j := 1; i =1;
2. for each n_ij ∈ M do
3.   k := j + 1
4.   while ( s_ij = s_ik )
5.     if ( p_ij < p_ik )
6.       eliminate p_ij
7.       j := k
8.     else
9.       eliminate p_ik
10.      k := k + 1
11.    if e_ij >  s_ik  then e_ij =  s_ik
12.    j := k
13. end for
end
```

It is a fact that, keeping the original durations of notes was a solvable issue. (Chai, 2000) presented the Revised Skyline Algorithm, where note elimination starts from the lowest pitch and continues until monophony is ensured. Nevertheless, solution to the last two critics required channel elimination. So, proper channel elimination techniques were needed to decompose channels which contain accompaniment information. Here, cognitive studies were expected for participation

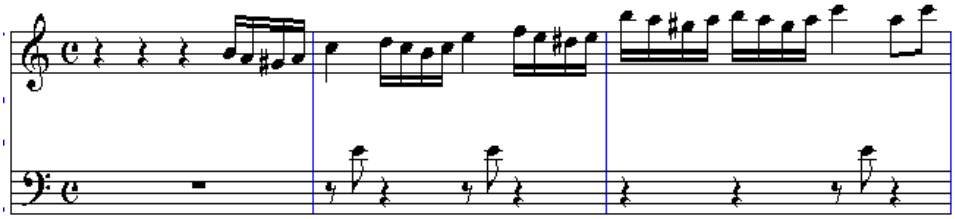
3.2.2 Channel Selection Algorithms

Uitdenbogerd and Zobel attempted to select best MIDI channel to represent melody of *M*. Their first algorithm, Top Channel, obtains Skyline output of each MIDI channel. Later on, algorithm computes average pitch frequency, a_i, of c_i. At the end, Top Channel algorithm eliminates all MIDI channels except c_i possessing maximum a_i. Their second algorithm, Entropy Channel, was quite similar to Top channel. But Entropy Channel Algorithm considers first order predictive entropy of c_i as channel selection criterion. In music sequences, predictive entropy can be defined as a measure of uncertainty between consecutive sequence letters. Here, we define b_i as the entropy of c_i. Lastly, they used heuristics to find the channel with maximum b_i, which was very similar to Entropy Channel Algorithm. Channel Selection algorithm is illustrated in Figure 3.1-c. Notes of the second channel are eliminated from the set.

a)



b)



c)

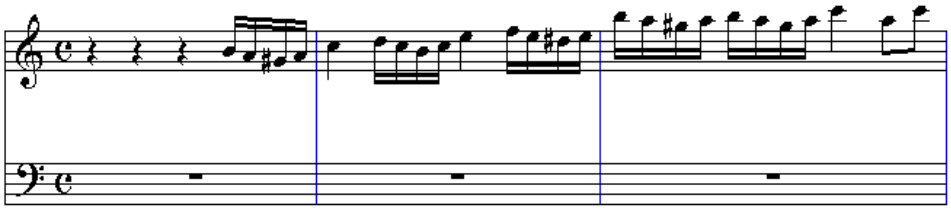


Figure 3.1 Notes of Alla Turka a)- Original Notes are decomposed in two MIDI channels, b)- Notes after Skyline Algorithm, c)- Having eliminated notes from secondary channel, both melody and rests will be maintained. Because we eliminated only accompany notes.

In addition, volume information could reveal some hints about melodic content. Chan presented that there was a relation between high volume and melody (Chan, 2002). Therefore he selected the channel which has maximum average volume.
Experiments show that performance of channel selection algorithms depends on the data set. Moreover, channel selection algorithms have a common pitfall. If perceived melody circulates around channels, selection of one melody channel will lead to loss of melodic information. We believe that there are still cognition facts which are not unraveled yet. If a music expert could select melodic line by premonition, then optimum melody extraction algorithm should be able to do so.

3.3 PARTIAL SKYLINE APPROACH

We believe that combining chief melody extraction algorithms in a new approach will outperform. Furthermore, pitch histogram of a channel reveals basic motifs of the melodic content. Thereupon, we can cluster channels which expose similar histograms. Here, we also consider histogram similarities between MIDI and its channels.

We prefer to implement three primary melody extraction operations before computing pitch histogram. Our first preliminary operation eliminates percussion data which is stored in c_{10}. Secondly, we apply Skyline Algorithm to all channels. So, only perceptively attractive notes make contribution to histogram. Thirdly we represent MIDI notes by 12 semitones. In order to achieve this, we computed modulo 12 equivalents of pitches(Lemstrom, 2000). As a result, pitch histogram of channels takes place in 12-dimensional space.

3.3.1 Analyzing Pitch Histogram of MIDI Channels

A histogram is the graphical version of a table that shows what proportion of cases fall into each of several or many specified categories (Histogram-wikipedia, n.d.). Let h_i is the pitch histogram of c_i, then we can define histogram set H as

$$H = \{ h_1 , h_2 , \ldots., h_{16} \} \tag{3.7}$$

Average histogram of all channels, h_A, can be computed as:

$$h_A = \frac{\sum_{i=1}^{16} h_i}{16} \tag{3.8}$$

Instead of searching for a standardized pitch histogram, our reference point is reached by h_A. Let d_i is the Euclidian distance between h_i and h_A.

$$d_i = \mathrm{d}\,(h_i, h_A) \tag{3.9}$$

Than we define distance set, D, such that :

$$D = \{ d_1 , d_2 , \ldots., d_{16} \} \tag{3.10}$$

It is a fact that d_i exposes histogram similarity between M and c_i. In other words, channels with similar d_i frequently reveal similar music features. Having clustered channels based on d_i, each cluster keeps a peculiar feature of music file.

In order to illustrate the significance of pitch histogram, we present Bon Jovi's favourite son "Always". Table 3.1 clarifies the histogram based distance features of channels. In terms of distances, there are two sudden increase occur between consecutive rows. Hence, channels can be decomposed into three basic clusters. Any well designed algorithm should be able to cluster the channels in proper manner.

Table 3.1 Histogram related distance features of "Always" from Bon Jovi.

Channel	Contain melody	Distance	consecutive difference	Group No
c_1	Y	0.0885	---	**1**
c_6	-	0.0935	0.005	**1**
c_3	-	0.0974	0.003	**1**
c_9	-	0.1065	0.009	**1**
c_4	-	0.1092	0.002	**1**
c_8	Y	0.1609	0.051	**2**
c_5	Y	0.2070	0.046	**3**
c_2	-	0.2149	0.007	**3**
c_{11}	**-**	**0.2174**	**0.002**	**3**

3.3.2 MIDI Channel Classification

In order to collect the channels with similar property at one cluster, we implement agglomerative clustering approach. In addition, we present a technique which computes a threshold. So that clustering approach iterates until the threshold value.
In general average, median or standard deviation features are used to stop agglomerative clustering. However, such techniques require satisfactory amount of samples. On the contrary, *M* contains at most 16 channels. Therefore, such threshold determination techniques do not make sense.

In order to present a better threshold for our purpose, we compute weighted average pitch histogram of *M*. Recall that c_i contains t_i notes. Than weighted average histogram, h_W, will be,

$$h_W = \frac{\sum_{i=1}^{16} h_i t_i}{16} \tag{3.11}$$

We determine the threshold, r, as

$$r = \frac{d(h_W, h_A)}{2} \tag{3.12}$$

Logic behind our threshold is as follows: If all channels have equal number of notes, than r will be zero. Therefore, clusters contain one channel. On the contrary, if a channel contains 99 % of the notes, it will lead a big threshold value and one cluster collects all the channels. Consequently, all but one channel from clusters will be eliminated.

For the sample song threshold, r, is 0.0214. Consequently, agglomerative clustering iterates until distance between merging clusters are smaller than threshold. Table 3.2 shows the decomposition of clusters after agglomerative clustering.

3.3.3 Combined Channel Selection Approach

Having clustered similar featured MIDI channels, we select best MIDI channel from each cluster and eliminate the rest of channels from MIDI. Here we present a combined Channel Selection algorithm which is a mixture of Top Channel and Entropy Channel algorithms.

Recall that predictive entropy can be defined as a measure of uncertainty between consecutive sequence letters (Uitdenbogert, 1999). Correspondingly, we let a_i and b_i be the pitch average and predictive entropy of c_i; combined channel selection criterion, x_i, is computed as:

$$x_i = a_i + b_i \times 128 \tag{3.13}$$

Pitch average of MIDI channel, a_i, can range between 1 and 128. On the contrary predictive entropy, b_i, ranges between 0 and 1. Therefore multiplying b_i by 128 balances the weight of a_i and b_i.

Table 3.3 and 3.4 show that Top Channel and Entropy Channels are supplementary to each other. There are examples where either Top Channel or Entropy Channel selects the convenient channel. Meanwhile, Combined Selection Algorithm chooses the best of Top Channel or Entropy Channel. Consequently, it overtakes all previous channel selection algorithms.

Having benefit from Combined Selection approach, we compute x_i values of all channels and determine the best channel from each cluster. In Table 3.2, c_1 has the maximum x_i value in the first cluster and selected as melody channel. In the same way, c_8 and c_5 are selected as melody and rest of the channels are eliminated from the M.

Table 3.2 Decomposition of MIDI channels consequently, channel selection in clusters.

Channel	Is Optimal Melody Channel	Cluster No	x_i	Combined approach selects
c_1	Y	1	154.8	**X**
c_6	-	1	100.1	
c_3	-	1	107.7	
c_9	-	1	98.6	
c_4	-	1	94.0	
c_8	Y	2	102.8	**X**
c_5	Y	3	78.6	**X**
c_2	-	3	49.5	
c_{11}	-	3	74.9	

Our example, approach yields three clusters. Rarely, files can yield five or more clusters so, do melody channels. In such cases, considering channels of the four clusters which show shorter d_i suffices. Because, melody clusters have tendency to expose shorter d_i values.

Finally, in our sample, Partial Skyline approach keeps important melodic contents, although some channels do not expose attractive pitch, entropy or volume properties. An example of this situation occurs in the sample song in Figure 3.2.

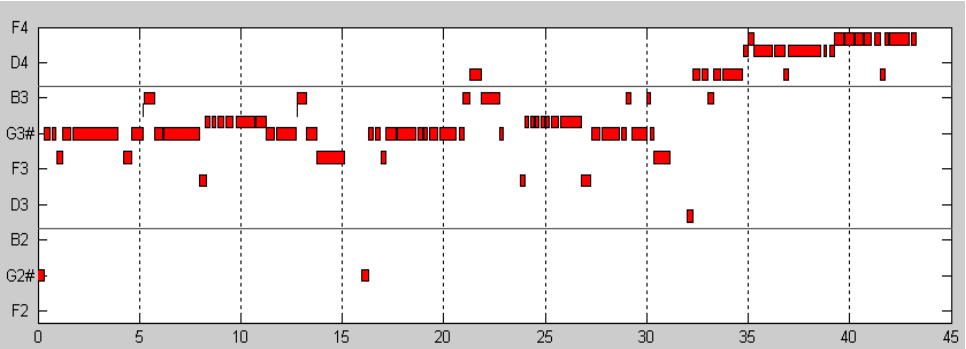


Figure 3.2 Piano roll of 5th channel from "Always". Although channel contains decent melodic information, it is eliminated by pitch frequency, entropy and volume techniques.

3.3.4 Summing up Partial Skyline Approach.

In order to summarize our study, we present the 8 basic steps Partial Skyline Approach

1. Apply Skyline Algorithm to all Channels
2. $\forall c_i \in M$, compute a_i and b_i
3. $\forall c_i \in M$, combined channel selection criterion is : $x_i = a_i + b_i \times 128$
4. Represent music with 12 semitones.
5. Compute clustering threshold.
6. Iterate Agglomerative Hierarchical Clustering on distance set, until threshold is encountered.
7. Apply Combined Selection approach in clusters. Eliminate remaining MIDI channels.
8. Apply Skyline Algorithm in the reduced MIDI set.

3.4 Test Results

Features of the music files have deep impact on the performance of Melody Extraction Algorithms. For instance, if a database consist of music files where accompany has high frequency, then worst performance will be generated by Skyline and Top Channel. However, such judgement cannot be generalized. In contrast, Skyline entitled as the most successful melody extraction algorithm.

3.4.1 MIDI Test bed

In order to represent music universe, a test bed should consider all aspects of music features and consequently collect files. In our test bed , considered features are high frequency, accompany has high frequency, Melody change instrument, Rest, Arpeggio, tremolo, volume, and glissando. In this respect, we selected samples where each selected feature is dominant at least in three music files. Selected files and their properties are revealed in Table 3.5.

3.4.2 Evaluation Methodology

Evaluation of a Melody Extraction Algorithms is collaborative study of both computational and musical science. Firstly, researchers from musicology department selected the music sequenced to be analyzed. The data set is illustrated in Table 3.5. Later, manually selected the channels where melodic contents take place. Additionally, musicologists determined weights, when multiple melody channels takes place in the same file. Because, finding the channel which has densely melodic content is a desirable property.

Later we compared the results between manual selections and the selection of the melody extraction algorithms. Computational evaluation was based on recall and precision. In addition, final outputs have been appraised by music experts.

3.4.3 Evaluation of Channel Selection Algorithms

In our test bed, Top Channel and Entropy Channel Algorithms expose yield similar performances. Depending on the data set features, each algorithm overtakes other. Meanwhile, in table 3.4, average velocity algorithm, which is based on volume average of notes, exposes unsatisfactory output in our test bed.
Test results in Table 3.4 shows that combining multi features of the channels can show impressive enhancement. Covering the whole scope of data sets can be possible when all cognitive features of music are considered. Consequently, we Combined obtained best performance from our Combined Channel Selection Approach.

3.4.4 Evaluation of Skyline Algorithms

Recall that Partial Skyline approach eliminates the channels which are potentially accompaniment; consequently implements Skyline Algorithm on selected MIDI channels. Thus, performance depends on the correctly decomposition of accompany channels. Table 3.4 shows that Partial Skyline algorithm exposes good performance in terms of Recall and Precision. In addition, we observed that partial skyline algorithm rarely miss weighted melody channels.

We believe that evaluation based on recall and precision is not enough for music files. Moreover, outputs should be analyzed perceptively. Table 3.3 explains the fundamental pros and cons of Skyline Algorithms.

Table 3.3 Evaluation of Skyline Algorithms

Algorithm		**Comments**
Skyline	Pros	Melody Changes instrument and melody has high frequency.
	Cons	Remove Rests, modify note durations, include accompany.
Revised Skyline	Pros	Melody Changes instrument and melody has high frequency. In terms of rests, overtakes Skyline.
	Cons	Not convenient for tremolo and arpeggio. Worst, when accompany has high frequency.
Partial Skyline	Pros	Lessens the interference from accompany channels. Compensate all deficiencies of Skyline Algorithms.
	Cons	Elimination of melodic content is possible.

3.4.5 Effect of the Feature over Partial Skyline

In some music samples, determining the borders of clusters are very sharp. Consequently Partial Skyline approach runs properly. For instance tremolo, which is the rapid succession of the same note, exposes sharp histogram distances. On the contrary, if the borders of clusters are fuzzy and accompany has high frequency, Partial Skyline will not guarantee generating satisfactory outputs. Furthermore, notes which have short durations are fatal to elimination if Revised Skyline Algorithm is considered.

Arpeggio, rapid succession of notes rather than concurrent notes, is successfully determined by Partial Skyline approach. In addition, channel elimination improves Skyline Algorithms, when note durations are very short. Because short durations of notes leads to densely population of concurrent pitches. Similarly, channel elimination is beneficial to keep rests as well.

Glissando, continues sliding of consequent notes, does not cause any effect on the histogram, neither do on partial Skyline approach. However, glissando is a hint for melodic content. We believe combining glissando into melody selection criterions enhance melody extraction.

Table 3.4 Comparison of Channel Selection Algorithms. Best Recall is obtained from Partial Skyline approach, whereas Combined Selection approach is the best in terms of Precision.

Recall		**Precision**		**Weighted Recall**	
Partial Skyline	0.769	Combined Selection	0.935	Partial Skyline	0.769
Top Rank	0.681	Entropy	0.774	Top Rank	0.687
Combined Selection	0.593	Top Chan	0.741	Combined Selection	0.603
Entropy C	0.477	Partial Skyline	0.653	Top Ch	0.487
Top Chan	0.465	Velocity	0.516	Entropy	0.470
Velocity	0.377	Top Rank	0.5	Velocity	0.374

Table 3.5 MIDI Testbed

ID	Name	Description	c_i	Optimals
1G1	Mozart Alla Turka	Melody has high frequency	2	**1**
1G2	Chopin - Etud Op. 12	"	1	**1**
1G3	Mozart -12 Variations	"	2	**1**
2G1	Beethoven 5. Symph	Meody change instrument	12	**8,3,5,1**
2G2	Mozart -12 Variations	"	2	**1,2**
2G3	Mozart-Concerto	"	9	**1,2,6,9**
3G1	Madonna Frozen	Accomp. has high frequency	11	**4**
3G2	Roxette- It must be love	"	8	**2**
3G3	R.Martin - Maria	"	8	**16,3**
4G1	Bach - Goldberg	Rest	1	**1**
4G2	Barry Manilov-C.Cabana	"	3	**7,2**
4G3	Yanni - Themes	"	4	**1,12**
5G1	Green Sleeves	Arpeggio	1	**1**
5G2	Rodgers - Romant	"	3	**1**
5G3	Albeniz - Asturias	"	2	**2**
6G1	Pulp Fiction	Tremolo	4	**6**
6G2	Vivaldi - Concerto	"	6	**1**
6G3	Empire Strikes Back	"	10	**3,9,4**
7G1	Tschaikovski - 1812	Volume	15	**1,6,13,4,7**
7G2	F. Lizst - Totantanze	"	4	**1,4**
7G3	M. Jackson - Thriller	"	7	**4**
8G1	C. Dion - All By Myself	Glissando	5	**7,8,1**
8G2	Tears in heaven	"	5	**5**
8G3	Congo	"	15	**2,1,5**
MG1	Mozart – 40 Symph.	Mixed	11	**12,3,1,4,2**
MG2	Tears in heaven	"	8	**4,5**
MG3	C.D. All By myself	"	9	**4,7,8,1**
MG4	Beethoven - Pastoral	"	11	**11,2,3,4,12**
MG5	Bye bye love	"	6	**4**
MG6	C.D. All Coming back	"	14	**13,6,1,15,7**
MG7	**Bon Jovi - Always**	"	**10**	**8,5,1**

Performance of Melody Extraction Algorithms depends on the data set. Due to multi cultural structure of the music, different criterions are essential for different

data sets. In order to design global Melody Extraction algorithm, we combine the criterions from literature. In addition, we analyzed the pitch histogram to decompose melody and accompaniment MIDI channels. Our test bed contains MIDI music files which cover common features of Western music.

Test results show that our combined approach overtakes when data set covers all important music features. However, results do not expose 100 % reliability. In order to present best melody extraction algorithm, cultural progressions and cognitive studies should be analyzed.

CHAPTER FOUR

ONLINE GENERALIZED SUFFIX TREE CONSTRUCTION ON DISK

In section study, we present an online generalized suffix tree construction algorithm on disk, where multiple sequences are indexed by a single suffix tree. Typically, performance of suffix tree construction on disk suffers from poor memory locality and high space consumption problems, especially when alphabet size is large. In order to overcome these problems, we propose a novel suffix tree construction algorithm which (1) takes letter frequencies into consideration and (2) involves an alternative physical node representation. We run a series of experimentation under various buffering strategies and page sizes. Experiment results showed that our algorithm outperforms existing approaches.

4.1 Introduction

Suffix trees, are versatile data structures which enable fast pattern search on large sequences. The large sequence, data set, can be a DNA sequence of a human, whose length is 3 billion; or it can be a collection of musical fragments, where number of fragments in the collection is large but average length of each fragment is moderate[. In both sequence cases, total size of the data set may be extremely large. For such large sequence sets, suffix trees introduce a fundamental advantage; sequence search time does not depend on the length of the data set.

Concept of suffix tree construction was initiated before the seventies by a brute force approach (Gusfield, 1997). For each suffix insertion, brute force approach preceded a common prefix search operation in the tree. Nevertheless, it was not practical since computational cost of the brute force suffix tree construction was at least exponential. In the seventies, linear time suffix tree construction algorithms were introduced using suffix links and tested on memory. (Wiener, 1973) and (McCreight, 1976). In these algorithms, suffix links functioned as shortcuts, which enable fast access to the suffix insertion positions of the tree. In other words, they

hold the address of a node, which contributes to the next suffix insertion position. As a result, traversing the tree for each suffix insertion was not necessary; instead suffix links from the previous step supplied the direct address. Due to this strong advantage, linear time suffix tree construction became possible. (Gusfield, 1997).

Although early suffix tree construction algorithms ensure linear time construction, they share a common pitfall: the offline property. For instance, in McCreight algorithm all letters of the sequence should be scanned before suffix tree construction procedure starts up. Such situation may cause an important constraint, if, for example, the occurrence of the rightmost letter is delayed. Twenty years after McCreight, Ukkonen has presented an online version (Ukkonen, 1995). In the online construction algorithm, the scanned part of the sequence can be projected to the suffix tree whereas; it is possible to extend the suffix tree by reading the next letter from the sequence.

Advancement on the suffix tree construction took a step by Generalized Suffix Tree (GST) (Bieganski et. al., 1994). Bieganski looked at the problem from a different aspect and pointed out the importance of indexing multiple sequences in a single suffix tree. In GST, it was necessary to identify the origin of each sequence. Hence extra node identifiers were added within leaf nodes. As a result of GST, most of the symbolic representations of Time Series could be indexed by a single suffix tree (Huang, 1999).

4.2 External Memory Suffix Tree Construction

We believe suffix tree construction algorithms on memory are almost mature. However, suffix tree construction applications on disk lead to important difficulties such as high space consumption and poor memory locality. Concretely, space consumption of a suffix tree node is high and less number of nodes can fit into a page. If a suffix tree contains large number of nodes, disk page requirement of a suffix tree will be large as well. On the other hand, poor memory locality is inevitable since suffix tree nodes are generated in random order and nodes of a

selected path are generally spread across different pages. Because of this, traversal on a path frequently leads to indispensable page misses.

Recently, some researches have pointed out disk based suffix tree algorithms. In 1997, Farach-Colton proposed a theoretical algorithm, which ensures linear time construction, (Farach et. al, 1998) but his algorithm has not supported by practical results. A popular platform, PJama suggests a reduction in space cost by removing suffix links from suffix tree (Hunt et. al., 2001). In addition suffixes were grouped according to their common prefixes. Finally, suffixes in the same group are inserted to the tree together. Hence both poor memory locality and space consumption drawbacks were improved. Recently, new studies have followed a similar path (Cheung et. al., 2005), (Phoophakdee and Zaki, 2007), (Tian et. al., 2005), (Wong et. al., 2007). Nevertheless, these algorithms did not consider online property of suffix trees and put constraints on dynamic sequence insertions. By the same token, GST could not consider online suffix tree construction since it was proposed before Ukkonen algorithm.

In 2004, Bedathur and Haritsa presented a fast online suffix tree construction on disk and made experiments on DNA and protein sequences(Bedathur, 2004). Based on Ukkonen's strategy, they presented a flexible algorithm, which enables dynamic modification on the suffix tree and considered physical node representations. In addition, they presented a buffering strategy, which considers probabilistic behaviors of path generation. However, the algorithm did not consider indexing multiple sequences in a suffix tree. In other words, they did not consider online GST.

In this study, we present an Online Generalized Suffix Tree (OGST) algorithm on disk. To the best of our knowledge, this is the first study dealing with OGST construction on secondary memory. Briefly, contribution of this study is threefold: The first, we modify the suffix tree nodes, so that the size of each tree node increases but direct access to parent becomes possible. Second, we introduce a frequency based sequence insertion strategy to enable fast access of frequent nodes of the tree. This requires knowing the occurrence frequency of alphabet letters before sequence

insertion. Third, we show that buffering performance will be increased, if the letter frequency distribution of available sequences is taken into consideration. In order to evaluate this approach, we make use of a popular Western folk musical database (Yet another Digital Tradition page, n.d.) which contains four thousand musical sequences, where alphabet size is 128 and average sequence length is sixty.

The remainder of the section is organized as follows: section 4.3 introduces some basic definitions and explains alignment of online suffix tree construction on memory. In Section 4.4, we introduce a new physical node representation. Also, we analyze the contribution of alphabet letter frequencies. Consequently, we present an improved sequence insertion order strategy. Test results are demonstrated in Section 4.5. Finally, Section 4.6 concludes the paper and gives a look to further studies on this subject.

4.3 Online Generalized Suffix Tree Construction

This section provides an overview of Online Generalized Suffix Trees (OGST) and analyzes the factors affecting the performance of disk based suffix trees.

4.3.1 Definitions

Suffix trees enable fast string processing on large data sequences. As shown in Figure 4.1, suffix tree is composed of edges, internal nodes, and leaf nodes. In particular, edges connect the nodes of tree and represents subsequence of a suffix. From root to an internal node, following the edges of a specific path may lead to a common prefix. In the tree, each common prefix is represented by a unique internal node and every internal node will branch to children nodes. Particularly, suffixes are represented by leaf nodes. That is to say, a leaf node addresses a unique suffix from data set.

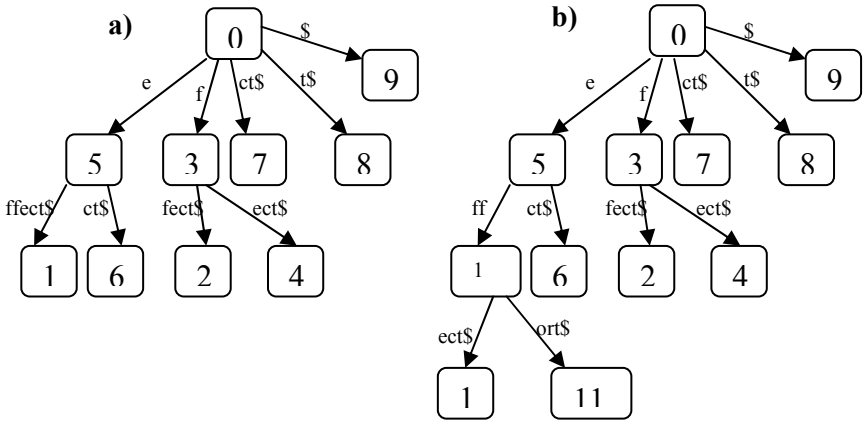


Figure 4.1 a) Suffix tree after inserting all suffixes of the sequence "effect$". Inside the rectangles, node generation orders are pointed out. The delimiter character, $, is used to maintain the end of a sequence**. b)** The suffix tree after insertion of the suffix: "effort$".

There are several ways to construct suffix trees. One of them is online construction, where scanned part of the sequence can be immediately projected to the suffix tree. Besides, generalized suffix trees index multiple sequences.

In this subsection, some of the preliminary definitions about OGST are given to clarify the notation used throughout the thesis. Let us assume that we have a collection of sequences, *S*, such that

$$S = \{ S^1, S^2, \ldots, S^k \} \tag{4.1}$$

Here, an arbitrary sequence, S^j, is defined as an ordered set containing all possible suffixes defined in alphabet, Σ. More formally, an arbitrary sequence S^j is defined as follows:

$$S^j = \{s_1^j, s_2^j, \ldots, s_n^j\} \tag{4.2}$$

where an arbitrary suffix, s_i^j , is a sequence containing the only last (n-i+1) letters of S^j in the same order. Meanwhile, the alphabet of the data set containing letters, Σ, is defined as follows:

$$\Sigma = \{ \sigma_1, \sigma_2 ,\ldots, \sigma_\gamma \} \tag{4.3}$$

where the alphabet size is equal to γ, (i.e., $|\Sigma|=\gamma$).

4.3.2 Online Generalized Suffix Tree Construction on Disk

While suffix trees introduced salient enhancements on string processing, most of the major research in this area were concentrated on memory aspects (McCreight, 1977), (Ukkonen, 1995). A decade ago, it was widely believed that disk based implementations were not feasible due to memory bottleneck (Bedathur, 2004). Here we first analyze the main reasons of memory bottleneck and later propose a new solution to enhance it in the next section.

The memory bottleneck is due to two factors. First one is high space consumption. Indeed, space cost of suffix trees range within 17n to 65n bytes (Wong et] al., 2007). For this reason, suffix trees can not fit into memory for large sequences. The second factor to cause memory bottleneck is the poor memory locality of nodes inside the tree. When a new common prefix occurs among suffixes, a representative internal node is generated and inserted into tree. Since generation order of the internal nodes are random and solely depends on the common prefix occurrences, random distribution of suffix tree nodes are indispensable and leads to poor memory locality.

Figure 4.1 illustrates a typical suffix tree. In Figure 4.1-a, suffixes of “effect” are indexed by a suffix tree. While dashed circles represent leaf nodes, flat circles denote internal nodes. The edges, which connect nodes, represent subsequences of a suffix. In order to illustrate poor memory locality, we illustrate node generation orders inside of circles. Therefore, we assume that initially root is generated and its

generation order becomes 0. In the tree, nodes of the paths have irregular node generation order. Figure 4.1-b shows the tree after inserting a new suffix, “effort”. Due to new common prefix between new suffix and the tree, a new internal will be generated. Since largest generation order in the tree was 9, generation order of the new node becomes 10. In the next sections, we will illustrate that node generation orders have decent impact over disk based suffix tree construction.

Tree travel is a technique for processing the connected nodes of a tree in some order (Tree Traversal – NIST, n.d.). For each suffix insertion, it is necessary to follow a top down tree traversal path which starts from root. Nevertheless, if the depth of the tree is big, following single path and inserting a single node takes O(n) time. Therefore, suffix tree construction becomes expensive. Instead, a shortcut from the previous node generation step can enable direct access to the demanded tree position. Here, corresponding shortcuts are named as suffix links. For a detailed description of linear time suffix tree construction and suffix links, reader is referred to read (Gusfield, 1997)

In an online suffix tree construction algorithm, physical node representations take more attention. An internal node may have so many children, or it may have only two children. Furthermore, total children of an internal node may increase during the suffix tree construction. Although it is possible to prepare a node for the worst case and align space to address maximum number children, it may not be space efficient.

4.3.3 Physical Representation of Suffix Tree Nodes

In a suffix tree, there are two fundamental choices for the physical representation of the suffix tree nodes: Linked-list Representation and Array Based Representation (Bedathur, 2004). We denote the physical node structures of both representations in Figure 4.2. The former strategy reduces space cost of an internal node; therefore it will be convenient for limited memory applications. However, from a disk based aspect, priorities change. Performance on disk depends solely on total pages misses (Salzberg, 1988). The latter strategy enforces nodes to contain more information

about children addresses; as a result, it ensures less number of hops to access to the next node in the path. The tradeoff between two strategies is an important factor for disk based application.

During the node generation, an internal node initially obtains two branches to handle child nodes. In other words, an internal node initially contains two child pointers and corresponding branches. While suffix tree construction proceeds, the node may obtain new children and consequent branches. For each new child, the node will need to keep an extra pointer. Henceforth, space requirement of an internal node increases. For instance, in Figure 4.1-b, the root node has five branches and consequent children nodes. Certainly, there is a limit on maximum branches from a node. If $|\Sigma|=5$, than an internal node will contain maximum five children.

In the suffix tree, alphabet size determines the maximum number of child pointers. It is the maximum possible sequences, which do not have common prefixes. The difficulty is holding possible $|\Sigma|$ branches in each internal node; at the same time, we have to optimize the ratio of used/unused child pointers.

4.3.4 Array Node Representation

Array Based Representation is a static data structure. By the time of an internal node generation, all possible $|\Sigma|$ child pointers will be arranged within an internal node; no matter how many of them are used. So, array-based representation simplifies the future modifications on the internal nodes. In Figure 4.2-a, alphabet has $|\Sigma|$ letters, rightmost $|\Sigma|$ pointers address possible branches. Start offset and edge length fields are aligned to represent subsequence of the sequence. In brief, each internal node of a generalized suffix tree requires $4+|\Sigma|$ pointers.

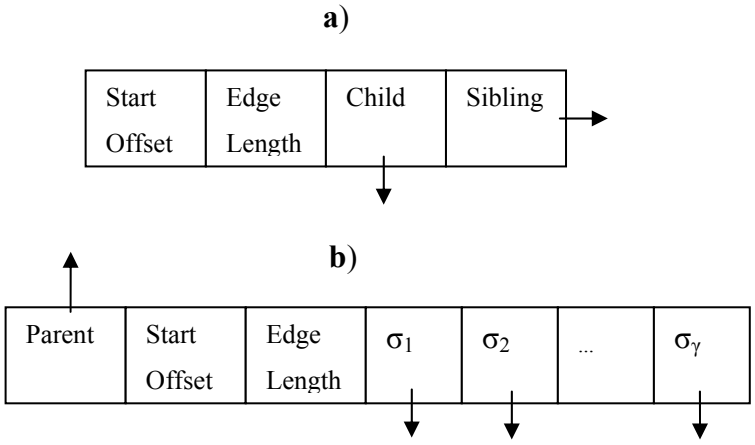


Figure 4.2 Two different way of physical representation of an internal node. a) Linked-list b) Array based.

Array based representation has both advantages and disadvantages. First, it is the simplest child access method and it enables direct access to the children nodes and consequent branches. In contrast to these advantages, it may lead to unproductive space consumption since some of the reserved child pointers for branching. In the worst case, $|\Sigma|$-1 letters of the alphabet may occur only once in the data set. Hence, almost all of the reserved child pointers waste space.

4.3.5 Linked List Based Node Representation

In contrast to array based representation, Linked-List representation does not hold the address of all possible children. Figure 4.2-b denotes the physical layout of an internal node where interconnection between nodes is maintained by child and sibling pointers. As it shown in Figure 4.3, children of a common parent are linked with sibling pointers and leads to a linked list. Recall that first letter of each child node is unique and represented by σ_i. We name the linked list in the figure as *sibling lists* throughout this study. In figure 4.3, only head of the sibling list is able to access to its sibling nodes. For instance, in order to access to v_{i2}, parent firstly need to access to v_{i1}. In brief, parent node need to hold only one child node, which is the head of *sibling list*.

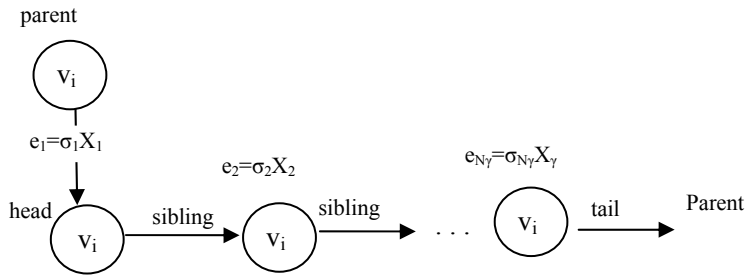


Figure 4.3 Parent to child and child to parent traversals on the suffix tree.

In terms of List Based Representation, handling parent address does not cause extra space consumption. Instead, we can handle the parent address with sibling pointer. Recall that tail of the sibling list does not address any sibling. Using such unused space for parent address supports a reduction in space consumption. Because of this fact, Linked List representation minimizes the space consumption.

Although Linked List representation minimizes space consumption, it increases total node fetches during child access. This fact may cause a trouble in a disk based suffix tree if nodes of a *sibling list* take place in the different pages. As a result, child access time on a *sibling list* may be expensive.

4.4 Fast and Space Efficient Suffix Tree Construction Algorithm on Disk

In a disk based OGST implementation, we believe that three issues are very important: (1) memory utilization, (2) fast access to a child node and (3) fast access to parent. Here, we present a technique which has two legs. In the first leg, we deal with direct access to a parent node. In the second leg, we sacrifice direct access to the child node for the sake of space utilization. However we can still present a strategy which enables fast access to a child node. To do this, we consider occurrence frequency of letters and probabilistic sequence occurrences.

4.4.1 Direct Access to Parent and Children Nodes

In order to optimize space utilization, we prefer a Linked List based node representation. However, we modify the nodes of Linked List by appending a parent address pointer and ensure direct access to parent node, named as Parent Address Appended List Representation (PAAL). As a result of this modification, size of an internal node will be increased by 25%. We assume that the extra space cost would be justified since it reduces total traversals on linked lists. In Figure 4.4, we denote PAAL as an alternative physical node representation.

In contrast to direct access to parent, direct access to children is problematic. Although reserving a pointer for each possible child is possible; it may not be feasible due to additional space overhead. As aforementioned, reserving a child pointer for each letter is not feasible. Instead, we prefer space efficient linked list node representation where sibling nodes are connected. However, we aim to place the most frequently accessed nodes to the head side of linked lists. Therefore, we maintain a speed up on access time of frequently access nodes. On the other hand, we try to align rarely referenced nodes at the tail of sibling list and venture their expensive access cost.

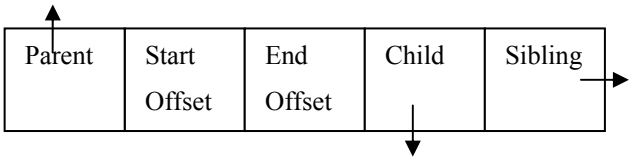


Figure 4.4 Physical Node Representation of PAAL

4.4.2 Impact of Alphabet Size on Tree Construction

Alphabet size may have deep impact over the suffix tree construction performance. In a static array node representation, large alphabets increase the size of internal nodes whereas; in a linked list representation they lead to an increase on the length of sibling lists. We need to mention that most of the studies from literature aim to index DNA sequences where alphabet size is only four. Therefore, space

utilization of array based representation leads to acceptable ratios. Nevertheless, conditions are different when alphabet size is large. For instance MIDI alphabet contains 128 letters [32]. Together with a delimiter character, alphabet size becomes 129.

Table 4.1 denotes space consumption of internal nodes for both array based and list based representations given that alphabet size is 129. As shown in the table, space consumption for the array based representation is fairly high; consequently less number of nodes can fit into a page. In contrast, Linked list representation is more efficient. As a result, a page contains optimal number of nodes.

Table 4.1 Space cost of suffix tree nodes for MIDI, where alphabet size is $|\Sigma|= 129$. MIDI alphabet contains 128 letters plus delimiter character, $.

Representation	Pointer Size (bytes)	Internal Node Size (bytes)	Leaf Node Size (bytes)
Array Based	4	532	16
Array Based	8	1064	32
Link List Based	4	20	16
Link List Based	8	40	32

If we assume that nodes of a common parent are stored in the same disk page, Linked List based representation outperforms. Nonetheless poor memory locality of nodes reduces performance on disk. As in Figure 4.3, access to $v_{i\gamma}$ can be very expensive when alphabet size is large, since each node access may cause a page miss to disk.

4.4.3 Impact of Letter Frequency Distribution on Tree Traversal

Probabilistic occurrences of letters in the alphabets are generally different. The English language is a good example. In English, ‘E’ is the most frequently used letter. In contrast, ‘Q’ is the letter whose occurrence frequency is the least (Morse

Code – Wikipedia, n.d.). In 1830's, Morse alphabet is inspired by such information. Similarly learning such information from a domain expert can enhance suffix tree construction as well.

In PAAL node representation, child access cost depends on traversals on sibling lists. We illustrate the situation in Figure 4.5, where children of a common parent are linked in sibling lists. In the Figure, the node which is the head of the sibling list, v_{i1}, can be accessed by one hop. However, accessing the node v_{iN}, which is the tail of sibling list causes extra traversal costs. It is preferred to see that v_{i1} is the most frequently referenced node, whereas access to the v_{iN} is very rare.

In the suffix tree, a less frequently used letter constitutes comparably simpler branches. As a result, depths of the relevant branches are comparably shallow and short. In contrast, branches those starts with dominant letters are complex and deep. Therefore, probability of reading such braches in the future is comparably higher

4.5 Probabilistic Occurrence of Longest Common Prefix

In this section, we explain which nodes of the suffix tree are accessed frequently. To do this we consider letter occurrence probabilities. Namely, we set about occurrence probability of all possible common prefixes. Computing such occurrence probability is very important since each common prefix is represented by an internal node. Therefore frequently accessed nodes can be estimated while suffix tree construction proceeds.

Lemma 1: Given the two sequences S^1 and S^2, $\frac{|\Sigma|-1}{|\Sigma|^{x+1}}$ is the probability that first x letters of both sequences are common but their $(x+1)^{th}$ letter is different.

Proof: The probability that first x letters of S^2 is equal to S^1 is $\frac{1}{|\Sigma|^x}$ and probability that S^2 has different letter on the $(x+1)^{th}$ position is: ($|\Sigma|$ - $\frac{1}{|\Sigma|}$). Hence probability occurrence of branching at the x^{th} element of S^1 is $\frac{|\Sigma|-1}{|\Sigma|^{x+1}}$

Lemma 4.1 implies that probability of branching after reading the first letter of a suffix path is quite high. In a large data set, we can find many suffixes whose common prefix length is one. On the other hand, probability of common prefix occurrence reduces when value of x becomes larger. In other words, edge lengths in deeper side of the tree have tendency to be long.

Lemma 4.2: For a given depth of a node is, p, the sequence between root and this node has at least p letters.

Proof: Since depth of a node is p, there exist exactly p edges to connect nodes on the same path. As it mentioned in suffix tree components section, edges contain at least one letter. Hence the sequence between root and this node has at least p letters.

Lemma 2 implies that depth of a node in the tree leads decent impact over the sibling list length. Nodes in the higher end side of the tree commonly have more siblings and length of the sibling list in the higher end side of the tree is long. Therefore access to the tail of a sibling list more number of node access. On the other hand, sibling list are shorter in deeper side no matter the length of the alphabet. For this reason, the probability of a common prefix comes into prominence in a tree level and length of the corresponding sibling list comes to the prominence.

4.5.1 Alignment of Sibling Nodes to Enhance Memory Locality

In order to reduce side effects of poor memory locality, we preprocess the available sequences before inserting them into tree. More concretely, we try to postpone constructing the specific branches of the tree by delaying the insertion of relevant sequences. In this way suffix tree nodes can have comparably better memory locality. In order to illustrate this, we present a primitive alphabet and respective data set and make the following assumptions to denote a primitive case. Let there exist two alphabets, Σ_1 and Σ_2 satisfying the following properties:

$\Sigma_1 \cup \Sigma_2 = \Sigma$

$\Sigma_1 \cap \Sigma_2 = \emptyset$

and $|\Sigma_1| = |\Sigma_2| = |\Sigma|/2$

We also assume that there exist three sequences, S^1, S^2, and S^3. All letters of S^1 and S^2 comes from Σ_1 and Σ_2, respectively. On the other hand, S^3 contains letters from Σ. In order to introduce the effect of data set size, we assume that length of S^1 is longer than the length of S^2. All three sets are planned be inserted to the same suffix tree.

Lemma 3: Maximum tree construction performance will be obtained, if we insert sequences of sets in the following order: First insert S^1, later S^2, and finally S^3.

Proof: Performance of tree construction depends on total node accesses in a sibling list. In general, cost of an unsuccessful node search depends on the length. The shorter the sibling list, the faster the search time. In the first phase, we insert S^1, where length of siblings list cannot exceed $|\Sigma|/2$. Hence inserting the S^1 can be done quickly.

In the second phase we insert the set S^2 into tree and maximum length of the sibling list will be extended to $|\Sigma|$. Still, cost of a successful search is $O(|\Sigma|/2)$, since nodes which are generated in the first phase takes place in tail side of sibling list and

successful searches never visit them. However, cost of an unsuccessful search time increases to O(|Σ|). Because of this fact large sequence set, S^1, should be inserted to the tree before S^2.

In the third set, S^3 will encounter longest sibling list. Prior, most frequently referenced sibling lists should have already contained |Σ| elements. Hence, both successful and unsuccessful search time will be proportional to |Σ| and searches lead more page misses. That is to say, the third phase does not introduce any performance gains.

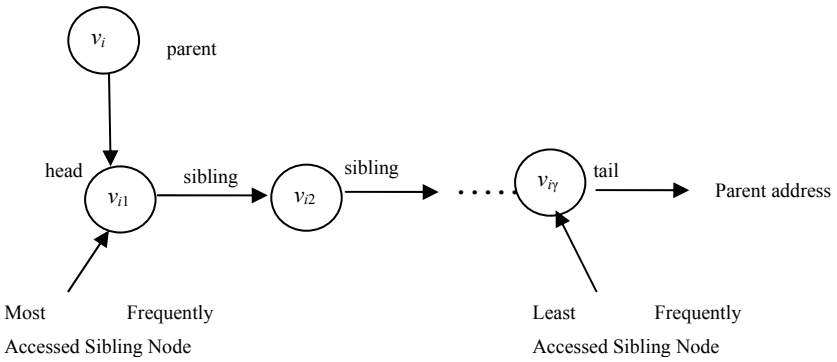


Figure 4.5 Sibling list for a common parent. While direct access to the first child is possible, access to requires traversal on sibling list.

4.5.2 Computing the Rank of a Sequence and Inserting to the Suffix Tree

As aforementioned, access probabilities of two sibling nodes are different; resulting from the alphabet letter frequencies. Therefore, aligning most frequently accessed node as the head of a sibling is preferred. When a new node is generated, it will be appended to the relevant sibling list as the new head. Correspondingly, rarely accessed nodes should be generated first to takes place in tail side of *sibling lists*. In this section, we ensure this by introducing a sequence insertion order strategy.

As in the English letters, we assume that average occurrence frequency of each letter and its corresponding histogram is known. We name such histogram as

centroid. Later on, we compute the letter occurrence frequencies of the available sequences and compute their Euclidian distance to centroid as their rank. In contrast to expectations, we insert sequences which are dissimilar to the centroid. Namely, sequences those yield higher rank are inserted to the tree earlier than those with a lower rank. Therefore nodes representing the rarely used letters will be inserted first. On the other hand, generation of the frequently accessed nodes of the tree will be delayed. Consequently frequently accessed nodes can stand as the head of the sibling lists of tree after inserting a group of sequences into tree.

We need to emphasize that each internal node becomes head of a sibling list just after its generation. We prefer that all accesses to the new node should be processed before another node is appended to the same sibling. In other words, we should try to construct all sections of a branch before locating to another branch. This can be achieved if sequences which have common characteristics are inserted to tree back to back. Therefore, computing the rank of a sequence seriously enhance performance

In the future, dynamic sequence insertions may continue and lead to generation of new nodes in random order. However, it will not cause a big problem since nodes in higher end side of the tree will have been already organized. Meanwhile future sequences mostly reference frequently accessed nodes in the tree where those nodes will have already taken head of sibling lists.

4.6 Experimental Results

In this section, we present the experimentations, which evaluate the physical node representation techniques on a disk based suffix tree. Besides, we consider the effect of buffering techniques and page size. Our evaluation criterion is based on page hits and misses. In our experiments, cost of a pointer is four bytes. Since space consumption of internal nodes and leaf nodes are different, we distinguish internal and leaf nodes in disk. In other words, a page is composed either all leaf nodes or internal nodes. Unless mentioned, we assume that size of a page is 4096 bytes. However, compare the performances of different page sizes.

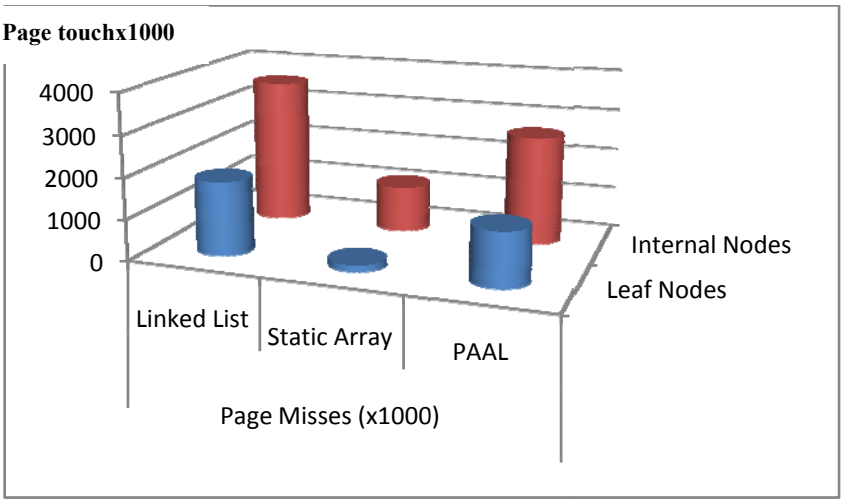
In order to test the algorithms, we make use of Digital Tradition Folk Music Database, containing nearly 4000 music files (Yet Another Digitial Tradition Page, n.d.). In the data set, MIDI music file format is used. The data set is composed of approximately 250.000 letters and standard MIDI alphabet contains 128 letters. After insertion of all sequences into tree, we observed that tree contained 126192 internal nodes and 229681 leaf nodes. We need to mention that sequence length is not necessarily equal to the total leaf nodes in the tree. This is because two sequences can have common suffixes and both will be indexed by a unique leaf node.

4.6.1 Comparison of Physical Node Representation Approaches

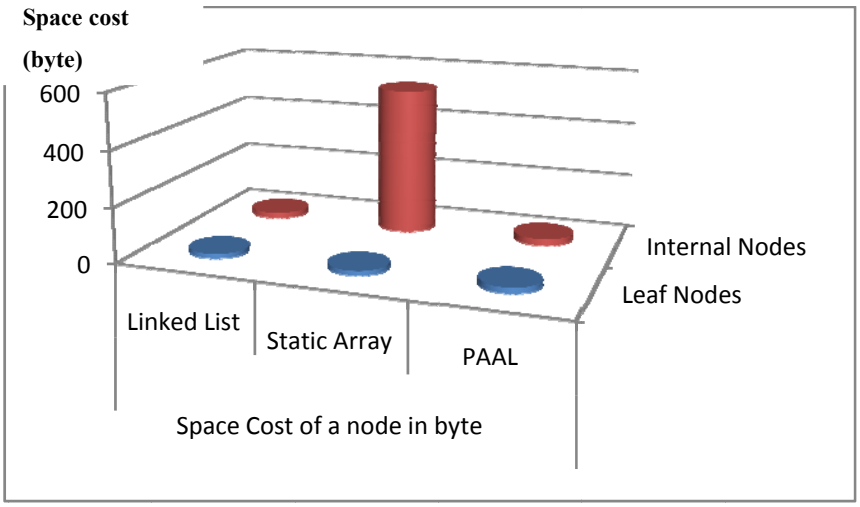
In Figure 4.6-a, we denote the space consumption of various node representation techniques. In the figure, Linked List representation ensures the best space utilization. Meanwhile, PAAL representation yields satisfactory space utilization as well, since it causes only 25% extra cost. The reason of the space overhead is the retention of parent node address in each node. However, space consumption of the Static Array Representation is quite high. Basic factor of the worst space utilization is the overheads in internal nodes. As aforementioned, an internal node needs to maintain $|\Sigma|+4$ pointers. In terms of MIDI sequences, high space consumption is indispensable since alphabet size is quite large (i.e., $|\Sigma| = 128$). Due to this fact, performance of Static Array is even worse if the alphabet size increases.

In terms of suffix tree construction speed, total page miss occurrence comes into prominence. Concretely, the fewer the page misses, the faster the algorithm is. In Figure 4.6-b, we denote the total page misses caused by physical node representation techniques when buffering is not considered. Although Linked List node representation is space efficient, it leads to high amount of page misses; hence it cannot be feasible on disk. Basic factor behind the performance loss is the node traversals on sibling lists. On the other hand, Static Array outperforms and ensures to least number of page misses. In fact, Static Array Representation enables direct

access to the child or parent node. Meanwhile our PAAL introduces compromise between Static Array and Linked List.



a-)



b-)

Figure 4.6 Comparison of phyiscal node representation algorithms without buffering a) Total page miss occurence. b) Total page requirement.

So far, we did not consider the impact of buffering on suffix tree construction. In the next section we will introduce page management strategies and show how PAAL outperforms when buffering is used.

4.6.2 Effect of Buffering

In terms of suffix tree indexing, page requirement of a large file is enormous. For instance; human DNA leads to 3 billion leaf nodes and at least 750 million internal nodes. Consequently, data will be aligned into millions of pages. Due to random distribution of the nodes; probability of finding two consecutive nodes in the same page is very small. The chance can be increased by buffering. In this section, we evaluate the effect of buffering on both vertical and horizontal traversals and discuss the contribution of letter frequency based sequence insertions on buffering.

Here, we compare the performance of three physical node representations using the following page replacement policies (Tanenbaum, 2006):

Least Recently Used (LRU): If a page fault encounters, least recently used page will be replaced.
First In First Out (FIFO): In case a page fault occurs, the page which stayed longest in buffer is replaced.
TOP_Q : Replaces the page if the average depth of the nodes in the page is the highest. In addition, replaced nodes will not be dropped immediately; instead they are processed in a FIFO fashioned buffer. We assume that 20 % of the buffer is reserved for FIFO buffer.

In Figure 4.7, we compare three physical node representations approaches and obtain results when the buffer contains 16, 64, 256 and 1024 pages. Test results imply that, no matter the page management strategy in all conditions, Linked List Representation yields the worst page miss ratio. In contrast, Static Array Representation outperforms and ensures least page misses when buffer contain 16 pages. However, increasing the total pages in buffer does not enhance its performance decently. Instead, increasing size of the buffer drastically rehabilitates the performance of PAAL. All in Figures 4.7-a,b,c we observe that PAAL ensures least number of page misses when buffer contains 1024 pages, hence outperforms.

From the results, we can conclude that buffering cannot expose its positive effect if the disk space is used extravagantly as in Static Array node representation.

4.6.3 Vertical and Horizontal Traversal on Buffering

We denote parent to child or child to parent access as vertical traversal. On the other hand, following the paths routed by suffix links are named as horizontal traversals. Online Suffix Tree Construction includes both horizontal and vertical moves. Thus, control of memory locality becomes even more complex and difficult. Here we can find an interesting advantage if consecutive sequences have long common prefixes. As a consequence, respective sequences follow the same path and nodes. Hence contribution of buffering over page hits will be more effective.

In the data set, inserting sequences with similar characteristics increase page hits. Figure 4.7 denotes that PAAL positively affects the paging performance. In all three buffering strategies, frequency based insertions outperforms. The reason of the performance improvement is as follows: Initially, sequences contain least frequently used letters and average lengths of the sibling lists are short. Moreover, consequent sequences should have similar letter characteristics and generally tries to access the same pages. The same condition occurs in the second phase, where we insert sequences those contain frequently used letters, so that frequently accessed nodes of the tree can take place at the head side of the sibling lists and average child access time reduces.

4.6.4 Effect of the Page size

Currently, disk based suffix trees are mostly tested on 4K pages. In this section, we want to analyze the contribution of page size over construction speed. For a constant buffer space, we compared the performance of variable page sizes. If page size is doubled, then less number of pages can fit into buffer space. In Figure 4.8, we evaluate performance of variable page size. In particular, we compare the

performance of 2K, 4K, 8K and 16K and reveal page miss occurrences under variable buffer management strategies. In this section, our performance criteria should be data transfer time (dtt) since page miss cost of 2K and 16 K is not same. We compare page sizes for each node representation.

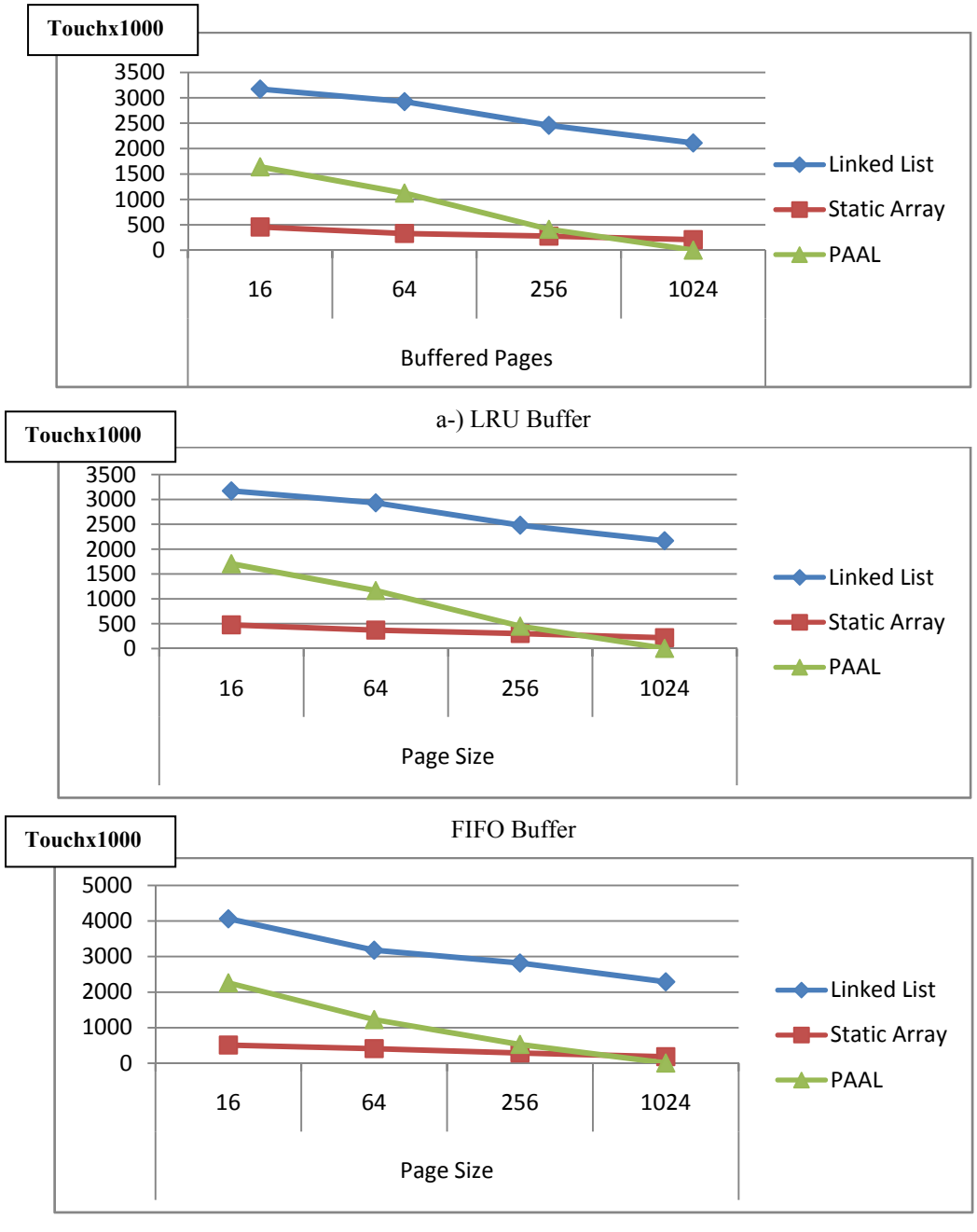


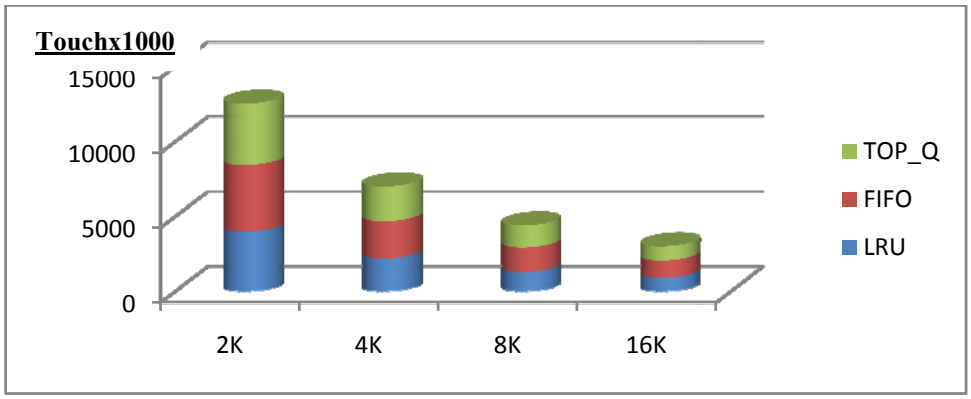
Figure 4.7 Comparison of the node representation algorithms in terms of page misses*1000. a) LRU buffering b) FIFO buffering c) TOP_Q buffering

Figure 4.8-a, illustrates the performance under the condition that physical node representation is based on PAAL. In the figure, we see that large page size reduce total page misses. However, data transfer time of a 16K page is eight times more than a 2K page. Hence small page size outperforms. Similarly, Figure 4.8-b and 4.8-c support the same result as well. Besides, Figure 4.8 illustrates that buffer management strategies yield different outcomes when underlying physical node representation changes. For instance, Figure 4.8-c denotes that Static Array Representation prefers TOP_Q buffering strategy. In contrast, TOP_Q is not convenient for Linked List Representation and illustrated in Figure 4.8-b.

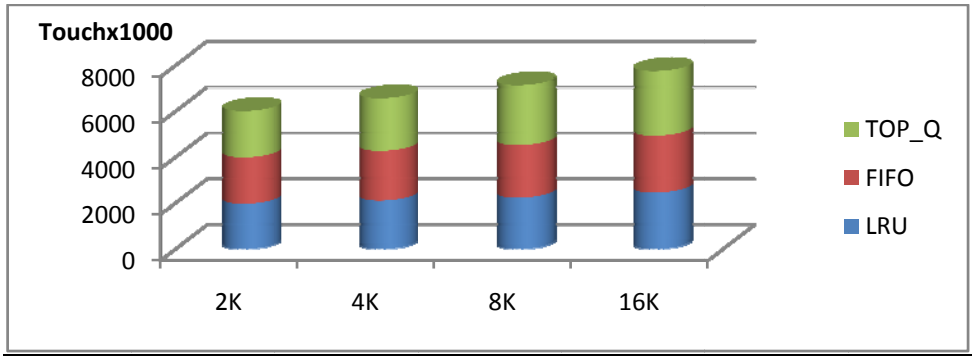
To sum up, we proposed a novel approach for Online Generalized Suffix Tree (OGST) construction on secondary memory. We showed that poor memory locality is indispensable when online generalized suffix tree construction is implemented. Moreover, we showed that large alphabet size drops the performance drastically on secondary memory. To solve this problem, we presented a space efficient physical node representation, named as PAAL, to enable direct access to the parent. However, it does not ensure direct access to the child node for the sake of space optimization. In order to speed up child access time, we estimated the most frequently accessed children of a parent node. Therefore, children of a common parent are aligned on a sibling list depending on their estimated access frequencies. In this study, we estimate the access probability of child nodes by letter occurrence frequencies of the alphabet. In contrast to expectations, we assign higher insertion priority to the sequences those contain least frequently used letters of the alphabet. Consequently, least frequently used children of a common parent are aligned at the tail side of sibling lists. In this way, new sequence insertions to the suffix tree yield better performance.

In this area there exist a number of research problems. The poor memory locality problem using suffix links is the most important one. Handling dynamic changes on data set makes the problem even more complicated. We believe bulk loading should make sense on a disk based suffix tree construction. Furthermore, constructing multiple suffix trees should be studied if data set contains multiple sequences and

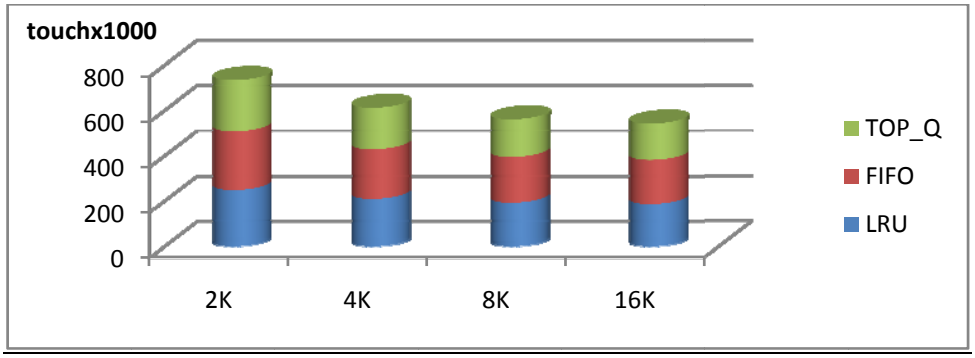
total letters of alphabet is large. We also believe that merging two suffix trees on disk is still an open problem.



PAAL



LRU



TOP_Q

Figure 4.8 Comparison page size over performance. a) LRU buffering b) FIFO buffering c) TOP_Q buffering

In order to find detailed statistical analysis of the tests, we denote all results inside Appendice.

CHAPTER FIVE

BALANCING SUFFIX TREE AND ALIGNMENT

The section explains random node generation inside a suffix tree and its three unwanted yields: unbalanced tree structure, poor memory locality and high space consumption. Later on, we explain possible recruitment techniques. We present that classification of sequences and indexing with multiple suffix trees should reduce total page touches and speed up construction time for large sequences. Also properties of the data set can present its own conveniences. Especially music has some properties which eases the indexing problem. At the end of the section, we illustrate our techniques with experiments and denote test results.

5.1 Introduction

Suffix trees not only introduce the fastest search on sequence databases, but also cause very complicated alignment strategy. These advantages and disadvantages are the yield of nodes that represent common prefixes. Indeed, in a data set, so many sequences and their suffixes have common prefixes. Hence representing all common prefixes by a unique node saves time and space. In this respect, common prefix representation is very important. Nevertheless such representation introduces its own difficulties into the field as well. Basically, a dataset cannot contain all possible common prefixes for a fixed depth since length of the suffixes inside a tree varies.

It is important to estimate node occurrence behaviors and their connections inside suffix tree. As a result, we can realize the importance of the problem and its difficulty. In this section, we explain probabilistic node occurrence behaviors inside suffix tree. So that we can understand factors causing unbalanced tree structure and poor memory locality.

Node generation behaviors have direct effect on poor memory locality as well (Schürmann and Stoye). Occurrence of a new common prefix between multiple sequences is random and consequent node generations are complex. Such event is the fundamental factor of poor memory locality. Hence analyzing the probabilistic behaviors of the tree is important to reduce poor memory locality problem.

In a file processing application, page touches are the fundamental performance concern (Folk et. al., 1997). Concretely, the more the page touches the slower the query speed is. As a result alignment strategy of suffix tree nodes is critical. We prefer that accessed nodes are stored mostly in a common page. While it is theoretically easy to implement such node alignment, practical suffix tree construction algorithms include leakages. Main leak is the common prefix occurrence and consecutive node generation inside the tree. In this the next subsections, we analyze the common prefix occurrence, corresponding node generation events, their probabilistic behaviors.

5.2 Definitions

In order to show the unbalanced structure of a suffix tree, use a data set where every sequence has the same length. So, we can show all possible paths and respective nodes of a tree. We assume that each sequence, S^J, has n letter. Let's assume that we insert enough number of sequences into tree, so that all possible common prefixes are represented by a unique node.

Given that alphabet, $\sum$, has d letters. Therefore, each internal node of the tree will have absolutely d children. This fact is shown in Figure 5.1.

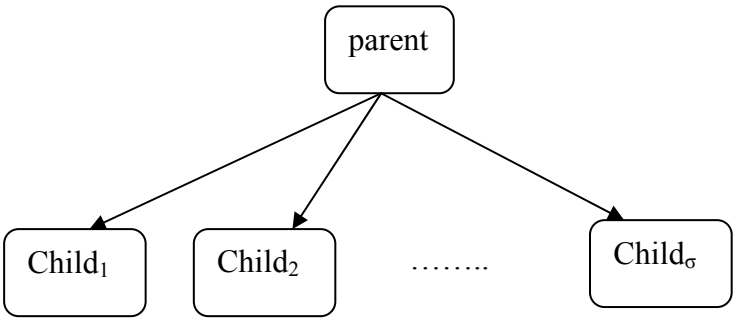


Figure 5.1 Node of a balanced suffix tree: Parent and its children of an internal node. Since alphabet size is , each node has exactly nodes inside the tree.

Since each internal node has σ children, total nodes in a depth level increase be a consequence of its depth. In other words, nodes in a depth level increase exponentially. If the first d level of the nodes have exactly σ nodes, there exist σ^d nodes in level d.

Lemma 5.1: Total nodes of the tree, T^d, until depth d is:

$$T^d = 1 + \sigma^2 + \ ... + \sigma^d \tag{5.1}$$

Proof: At depth d, there exist exactly σ^d nodes. Similarly depth d-1 has $\sigma^{d\text{-}1}$ nodes. Hence lemma is proved.

Lemma 5.2: Total leaf nodes of a tree are always more than total non-leaf nodes.

Depth of the leaf nodes is always more than non-leaf nodes. If the lemma is true, then,

$$\sigma^d > 1 + \sigma^2 + \ ... + \sigma^{d\text{-}1} \tag{5.2}$$

By induction we can prove this.
Basic step; Given that $2 \leq d$ and $2 \leq \gamma$

$\sigma^2 > 1 + \sigma^1$ Basic step is proved.

Induction step: we assume that $\gamma^d > 1 + \gamma^2 + \ldots + \gamma^{d-1}$

$$= \sigma\, \sigma^{d-1} > \sigma\,(\frac{1}{\sigma} 1 + \sigma^1 + \ldots + \sigma^{d-2})$$

$$= \sigma^{d-1} > (\frac{1}{\sigma} 1 + \sigma^1 + \ldots + \sigma^{d-2}) \quad (5.3)$$

Hence induction is proved.

Lemma 5.1 and 5.2 are important for two reasons: 1) A suffix tree may have enormous number of node at higher depths. 2) Alignment and balance of these nodes are extremely difficult. The suffix tree will have σ^d nodes at depth σ. This is a very serious problem; since σ depends on the length of the sequences. For instance, even shorter sequences contain 100 letters. This leads to at least 10^5 nodes at depth 100. This is a very serious problem; since σ depends on the length of the sequences. For instance, even shorter sequences contain 100 letters. This leads to at least 10^5 nodes at depth 100. As a result of aligning 10^5 nodes on disk, the small sequence length demands petabytes of memory. Because of this fact, reserving enough space for all possible nodes is not wise. Instead nodes should be generated, whenever they occur.

The alphabet size has decent effect on the total nodes in the tree as well (Gusfield, 1997). When alphabet size increases, the problem becomes even more extravagant. Recall that total nodes of a tree will be σ d nodes at depth σ. Increasing the σ value on the formula leads more nodes.

As it mentioned in section 4, node generation order of the suffix tree is random. Collecting interconnected nodes in a page is not possible during construction. While page addresses of nodes can be swapped afterwards; it causes high computation costs. In the next subsection we consider the node alignment of suffix tree nodes inside pages

5.3 Suffix example

For a given suffix tree, there exists only one possible suffix tree representation. Hence it is deterministic. On the other hand, alignment orders of the nodes may vary. For instance tree nodes can be randomly distributed among disk pages.

As an illustration of generalized suffix trees, we let there exists threee sequences to be indexed.

S^1 = “ABCACBBCCC$”
S^2 = “ACABBACCBAB$”
S^3 = “AAABAACCBAB$”

The alignment of the sequences on disk is illustrated in Figure 5.2. The sequences are distinguished by a delimiter character. The second row of the figure denoted the alignment orders. Therefore, a subsequence can be represented by it onset and offset. For instance [12, 23] represent S^2

A	B	C	A	B	B	B	C	C	C	$	A	C	A	B	B	A	C	C	B	A	B	$	A	A	A	B	A	A	C	C	B	A	B	$
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

Figure 5.2 Sequences on disk and their alignment.

5.3.1 Suffix List View

We illustrate the suffixes of all sequences in Figure 5.3. We expect that a suffix tree should index all possible suffixes of the sequences. Recall that there exists 35 different suffixes to be indexed. Nevertheless, some suffixes of the set are same. For instance suffixes with ID's 17 and 29 are same. Hence both suffixes should be represented by a single path. We also need to mention that if two sequences are same their corresponding suffixes will same as well. Hence corresponding suffixes will be represented by a single path once as well.

ID	Suffix	ID	Suffix	ID	Suffix
1	ABCACBBCCC$	12	ACABBACCBAB$	24	**AAABAACCBAB$**
2	BCACBBCCC$	13	CABBACCBAB$	25	**AABAACCBAB$**
3	CACBBCCC$	14	ABBACCBAB$	26	**ABAACCBAB$**
4	ACBBCCC$	15	BBACCBAB$	27	**BAACCBAB$**
5	CBBCCC$	16	BACCBAB$	28	**AACCBAB$**
6	BBCCC$	17	ACCBAB$	29	**ACCBAB$**
7	BCCC$	18	CCBAB$	30	**CCBAB$**
8	CCC$	19	CBAB$	31	**CBAB$**
9	CC$	20	BAB$	32	**BAB$**
10	C$	21	AB$	33	**AB$**
11	$	22	B$	34	**B$**
		23	$	35	**$**

Figure 5.3 Suffixes of the tree sequences.

5.3.2 Suffix Tree View

In a suffix tree, the suffixes are represented by leaf nodes. Meanwhile, internal nodes represent common prefixes. For each suffix insertion, a leaf node will be aligned into disk page. During this process, a common prefix search will be made. When new common prefixes occur by a new suffix insertion, corresponding internal nodes should be generated and inserted into the tree.

Root of the suffix tree contains only one child address. Other children of the root are accessed by sibling pointers. The situation is illustrated in Figure 5.4. As an example, if we want to access to the node, “2:3”, we need to access to “2:1” and then follow the sibling pointers. Since each node does not contain addresses of all its children, decent amount of space is preserved.

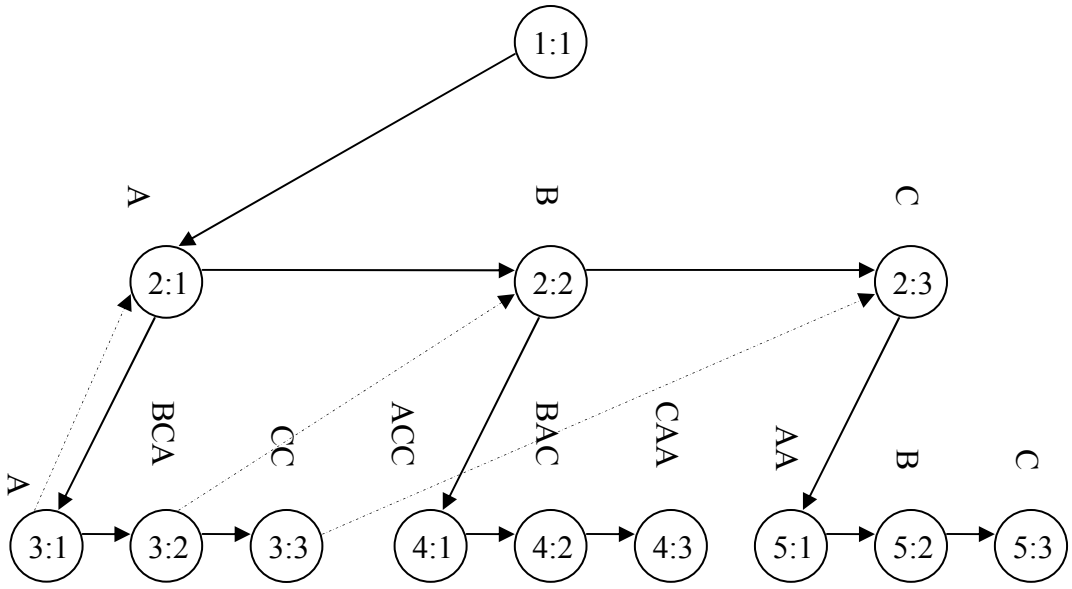


Figure 5.4 A suffix tree where alphabet is Σ = {A, B, C, D}. In the figure, insertion order of nodes are illustrated inside the nodes.

In the figure, dashed lines represent suffix links. The links depend on the leftmost letter of the edge. We need to mention that occurrence of node access using suffix links are comparably less. As a result, impact of a page miss due to a suffix link access may not be very hard. On the other hand sibling access is frequently occurs. Basic factor is parent to child accesses and for such accesses sibling lists are used. As in Figure 5.4, we would rather access single page during tree traversal. As an example, we need to fetch six different nodes to access “3:9”.

5.3.3 Planar View

Recall that above formulas are computed for the case where access probability of every node is same. Recall that access frequency of each node may not be same. This depends on the occurrence frequency of alphabet letters.

1:1	**null**	**null**	**null**	**72**	**null**	**null**
	-	-	-	-	-	**-**
	-	-	-	-	-	**-**
	---------	---------	---------	---------	---------	**---------**
2:1	**1**	1	0	144	96	**0**
2:2	**2**	2	0	216	120	**0**
2:3	**3**	3	0	288	null	**0**
3:1	**--------**	--------	--------	--------	--------	**--------**
3:2	**25**	25	72	0L	168	**72**
3:3	**2**	2	72	72L	192	**96**
	13	**13**	**72**	**144L**	**null**	**120**

Figure 5.5 In the best case, disk layout of file containing internal nodes.

5.4 Alignment of Suffix Tree on Disk

Suffix tree is a deterministic tree. In other words, node hierarchy only depends on the sequence. Nevertheless, alignment order of the nodes on disk is not deterministic. As shown in Figure 5.6, 5.7, and 5.8 same nodes of the tree can be aligned to disk in different ways. Hence, alignment order strategy of nodes may determine the total page misses.

0	**26**	**35**	**144**	**20**	**24**
20	**27**	35	144	40	**25**
40	**30**	35	144	null	**28**
	----------	----------	----------	----------	**----------**

Figure 5.6 In the disk case, disk layout of file containing leaf nodes.

Since node generation order of a suffix tree is random, memory locality needs to be enhanced. This can be achieved by two different ways. Either alignment order of nodes will be changed or node generation inside tree will not be made sequentially. Instead group of nodes will be inserted together using bulk loading or clustering.

1:1	**null**	**null**	**null**	**72**	**null**	**null**
	-	-	-	-	-	**-**
	-	-	-	-	-	**-**
	--------	--------	--------	--------	--------	--------
2:1	**1**	1	0	144	216	**0**
2:2	**2**	2	0	288	340	**0**
2:3	**3**	3	0	432	504	**0**
	--------	--------	--------	--------	--------	--------
3:1	**25**	25	72	0L	168	**72**
3:2	**2**	2	72	72L	192	**96**
3:3	**13**	**13**	**72**	**144L**	**null**	**120**

Figure 5.7 In the worst case, disk layout of file containing internal nodes.

Aligning all nodes inside a single page is impossible for large databases. Even more aligning a selected path into a single page does not make sense. Remaining paths cannot be aligned optimally inside tree. In the best case, we try to align a node and its children in the same page. Therefore each traversal cost causes $d/2$ page misses. We believe this is acceptable amount.

0	**26**	**35**	**144**	**216L**	**24**
20	**27**	35	144	340L	**25**
40	**30**	35	144	null	**28**
	----------	----------	----------	----------	----------

Figure 5.8 In the worst case, disk layout of file containing leaf nodes

A parent may have σ children and σ^2 grandchildren. Given that σ is large, aligning all children on the same page may not be feasible due to available page size.

In the worst case, each node access leads to a page misses. In a static array node representation, totally d page misses occur. However, in a linked list node representation, the situation is terrible. In average d. σ /2 page misses occur.

	Static Array	Linked List
Best case →	d / 2 misses.	$d / 2$
Worst case →	d misses	$d\sigma/2$

Without manual contribution, tree nodes are randomly distributed among pages. However, sequence insertion order and swapping the nodes of the tree is able to change address. Here we consider the cost of related action. In the first phase, we consider the sequence insertion order approach.

5.5 Swapping the nodes

A node addresses a parent, a child, a suffix link and a sibling. If address of any of each changes, than the node should be modified. In other words, if we change the address of a node, we should search all nodes those addressing it as parent, child, suffix link or sibling. Although finding parent or child links easily; finding the node which addresses it as suffix link may be very hard.

In Figure 5.9, physical node representation is denoted. As shown in Figure 5.9, a tree node links to four different addresses. Hence changing the address of a node leads to address changes in other nodes. For instance assume we have changed the address of the node at position 2:2 in Figure 3. Hence we need to access to all of its children and change their parent address. Similarly, we need to access to the node at position 2:1. Finally we need to find a node whose suffix link addresses node at 2:2.

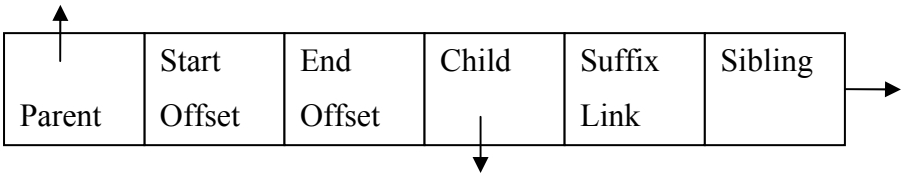


Figure 5.9 Physical Node Representation of PAAL

Access time to child nodes can be made in satisfactory time if all children are aligned to the same disk page. Iteratively each node's parent address can be modified and written to the relevant page. Hence modification of child nodes can be made in O(1) time.

A node will be addressed by an either a child pointer or sibling pointer. In both cases, we need to access to the parent. In Figure 5.4, 2:2 is accessed by a sibling pointer. Later on 2:1 is accessed. Without buffering the procedure cost two page touches. However, within a single buffer space page miss number can be reduced to one, since 2:1 and 2:2 can be aligned in the same page.

Modification the address of the node whose connection is due to suffix link is problematic. Actually brute force search is necessary. Hence its cost is O(n). Here we can present our solution to reduce this number to O(1).

Lemma : Path traversal with suffix links ends up at the root.

Proof: After each suffix link access, length of the suffix decrease at least once. This fact is illustrated in Figure

After a single step, the new node will represent a suffix of the previous node. Hence after each move, length of the represented prefix reduces. Finally, length of the common prefix becomes zero and it is represented by the root node.

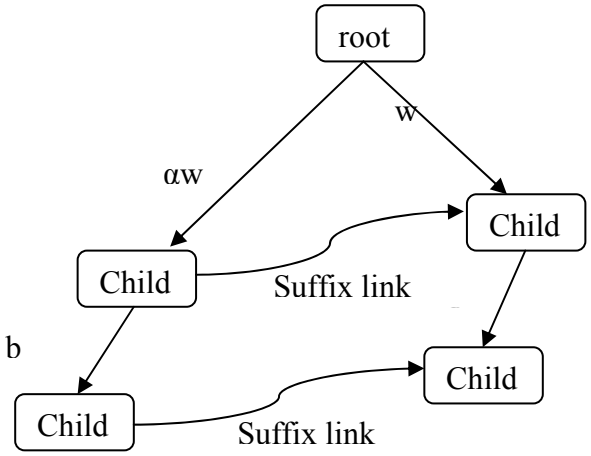


Figure 5.10 Objective of suffix links inside suffix tree. α, w, b are three sequences. If we insert the suffix αwb into tree, then we will need to insert

Changing the address of a node requires updates at other tree nodes. Let's assume that we changed the address of $Child_{21}$. After such change, we need to update nodes those addressing $Child_{21}$ as a parent, child, sibling or suffix link. Hence all necessary nodes should be accessed during address change procedure.

In the suffix tree, parent access, child access and sibling access time is limited by the alphabet size. Therefore, each node updates can be processed in constant time. However, searching the node which addresses $Child_{12}$ as suffix link is expensive. For such case, brute force node search is indispensable. As a consequence node search time depends on the depth of the tree. Concretely, its computational cost is $O(n)$.

Two approaches can be presented to solve this problem. First, we can drop suffix links from the suffix tree and sacrifice its advantages. Secondly, we can do address changes on the higher end side of the tree.

Dropping suffix links from the suffix tree may speed up suffix tree construction for static sequences such as DNA. Also dropping them reduce the space cost. However, streaming sequences require suffix links. Basically, suffix links enables

dynamic sequence insertions in linear time. Therefore first approach, which drop suffix links fails for streaming sequences.

Address changes only at higher end side of the tree should be very effective. Like all tree data structures, higher end side of the tree is frequently accessed. Hence rich memory locality on higher end side of the tree speeds up the performance decently. In contrast, access to nodes at lower end side of the tree is rare. It should be wise to sacrifice their cost.

Fast address change techniques on a suffix tree can have important contributions to the string processing researches. In some cases, multiple suffix trees construction can be necessary. Therefore, merging suffix trees become essential. In the next section we consider merging suffix tree approach and its importance for music sequences.

5.6 Multiple Suffix Tree Construction

In some special cases, multiple suffix tree construction should make sense. Depending on the data set, sequences can be classified into groups and each sequence set can be indexed by different suffix tree. Especially, multiple suffix tree construction should make sense when alphabet size is large. Music can be a good example.

Multiple suffix trees leads to efficiency when alphabet size is large. Although letter range is wide, a sequence may not include all possible letters. Music is a good example. For instance MIDI music alphabet has 128 letters (MIDI-Wikipedia, n.d.). In such case, static array fails since each internal node reserves 128 child pointers during its generation. As shown in Figure 5.11, size of an internal node will be terribly large due to child pointers. Similarly, linked list node representations cause failures since child access may cause 128 page access.

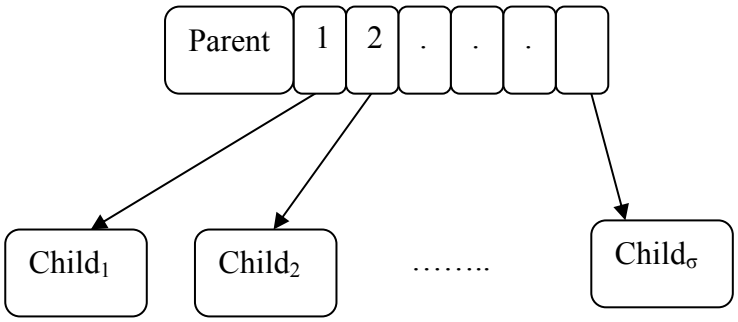


Figure 5.11 Space curse on static array physical node representation for large alphabets. All possible children of an internal node should be reserved during construction.

Multiple suffix trees can speed up the suffix tree construction and search time. Using with a wise classification technique, internal nodes may have less number of children to address. In this way, performance can be enhanced clearly. However, indexing with multiple suffix trees have different effect on linked list and array based node representation.

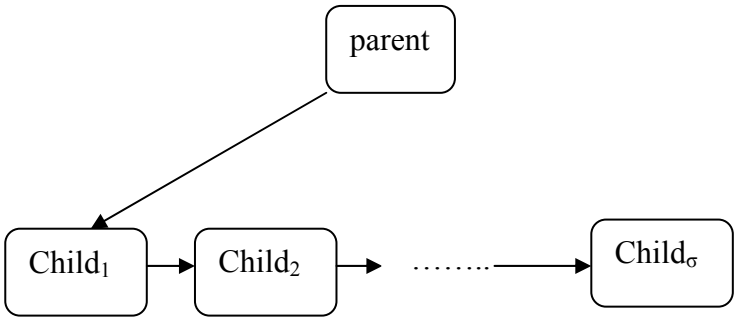


Figure 5.12 Page access curse on linked list node representation for large alphabets. Direct access to the child node may not be possible. Length of the sibling list depends on the alphabet size.

5.6.1 Multiple Suffix Trees and Array Based Node Representation

As it mentioned before, array based node representation reserve maximum space at the beginning. In order to see enhancements on array based representation, alphabet size should be reduced for a data set. It is the mission of sequence classification algorithm to collect sequences in a well formed.

In the dataset, sequences are generally large. In this respect, introducing constraints on letter usage is difficult. As a result, classification algorithms generally cannot reduce the alphabet sizes so much.

In music, octave equivalence makes the array based representation as an efficient implementation. The octave equivalence enables to represent music with 12 notes. Therefore, music sequences can be fit inside a alphabet which has 12 letters. Our analysis show that performance of suffix tree increases when alphabet size shrinks 10 times. Basic factor behind this enhancement is based on shorter alphabet size. Therefore, space consumption of suffix array can be reduced ten times.

5.6.2 Multiple Suffix Trees and List Based Node Representation

Multiple suffix tree construction can be convenient for List based node representations. In such representation, exceptional letters can take place in the data set. In contrast to array based representation, those exceptional letters do not cause any space cost. Instead these letters leads to few extra nodes which take place in the sibling list. Assuming that these nodes take place at the tail of the sibling list, their extra cost may be almost zero.

In terms of linked list node representation, data set should be clustered before insertion (Jain and Dubes, 1997). The clustering algorithm should discriminate sequences be a result of their alphabet usage. Later on each tree indexes the sequences with a specific cluster. Since each tree focus on a set of letters, alphabet size can be shrinking with few exceptions.

The clustering mechanism can be illustrated by a MIDI music sequences. MIDI pitches range between 0 and 127; in other words alphabet size 128. However, each sequence does not contain all letters of the alphabet. For instance, music sequences contain 12 different notes. As a result of this fact, sequences can be clustered by their letter histograms.

Multiple suffix tree has certain drawbacks as well. First of all, a search operation should be made in each suffix trees. Secondly, the structure leads to extra space cost. When a query is demanded, it should be searched in each suffix tree. Therefore, search time increases. Moreover, multiple suffix tree requires extra nodes. In each tree, common prefixes are generated independently. Hence, two nodes of distinguished trees may have similar function and represent same common prefix.

We believe that it is worth to pay to the drawbacks of multiple suffix trees if the alphabet size is large and sequences generally contain less number of letters. Since sequences generally contain few different letters, they can be clustered easily.

Let's assume that a unique suffix tree indexed a sequence data set where it consumes η amount of space and average search time on the tree become φ. If we index the same sequence set with g suffix trees, total space consumption of all trees will be less that $\eta\, g$. Similarly, average search time will be less that $g\, \varphi$. Since g is a small, extra costs on space cost and search time can be acceptable. Practically, space cost and search time of multiple suffix tree will be more comfortable than their maximum values.

5.7 Experimentation

In this section, we analyze the factors which cause unbalanced tree structure. In order to ensure balanced tree structure, we considered multiple suffix tree construction on MIDI music sequences.

In order to evaluate multiple suffix tree constructi
which contains national anthems of 190 countries. In a
with the most popular 219 film soundtracks and 208 p
almost all files are in polyphonic form. As a
multidimensional form. In this respect, Melody Ex
Here, each MIDI channel is transformed into a differe
action, we have obtained 9236 monophonic sequence
tree. The properties of the sequences are shown in Fig
5.17.

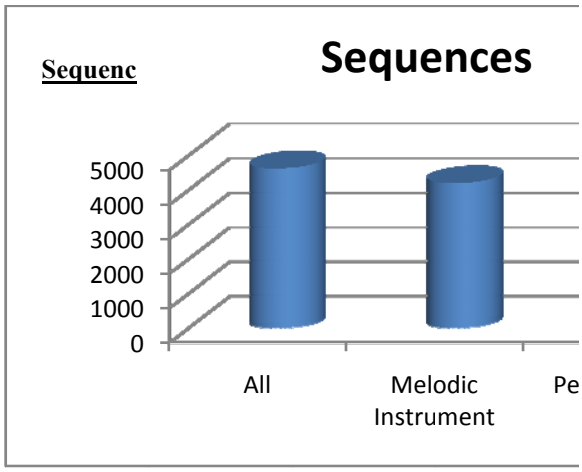


Figure 5.13 Number of Sequences. 10% of the sequen

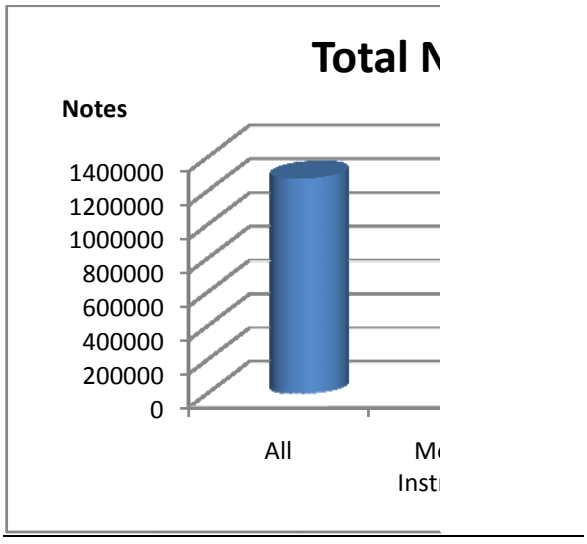


Figure 5.14 Total Notes in each set.