Creating Cross-References Using a Suffix Array:

A Unique Way to Analize Koiné Greek and Music

Nathan Banks

B.Sc., Trinity Western University, 2004

Thesis Submitted In Partial Fulfillment Of
The Requirements For The Degree Of
Master of Science
in
Mathematical, Computer, And Physical Sciences
(Computer Science)

The University Of Northern British Columbia
August 2009

Abstract

A suffix array is a method of reorganizing data into a form which facilitates searching and exposes sections of a document that are repeated. Finding parallels is important to both biblical scholars and music theorists. After an extensive overview of suffix arrays and the creation algorithm employed, this thesis exploits the data structure's properties to find parallel passages and to create cross-references. This was done by analyzing two suffix arrays: one generated from the Koiné Greek New Testament and another created from a fugue written by J. S. Bach. This thesis proves that the data structure itself is extremely useful for analyzing both ancient writings and western music by demonstrating one way to produce cross-references in an extraordinarily efficient manner. In addition to analyzing data, this research could serve as a foundation for computer assisted machine translation and music information retrieval.

Contents

Introduction

Studying a suffix array is a little bit like studying a brick. One doesn't normally study a brick because it is not very interesting in and of itself. Everyone knows that a brick is made of baked clay, that it is relatively heavy, and that it can serve as a good paperweight as long as one does not care about scratching the table. A structural engineer would also know that a brick has a compression strength but a low tensile strength. Unfortunately, these properties are generally far less exciting than the houses which can be constructed with bricks, but a better brick will produce a better house. Therefore, it is very useful to study the properties of a brick because the properties of these will influence the properties of a houses.

A suffix array is basically a document which is transformed into something entirely different. This metamorphosis is the basis of many searching, translating, and compression algorithms. It is very useful to study the raw information created by a suffix array for many of the same reasons it is useful to study the structural properties of a brick. It can also be about as interesting as studying a brick (except that computer scientists are usually intrigued by the one whereas structural engineers are typically interested in the other). It is possible to create faster cross-referencing systems, better translation software and faster search software by analyzing the simplest forms of a suffix array. This thesis provides such an analysis.

This thesis is divided into four chapters. In Chapter 1 on page 6, I look at a broad overview of suffix arrays. This is a little like looking at the history of a brick. Although the language is not extremely technical, the intended audience is people who work with computers. In this chapter, I first look at various uses of the suffix array in Section 1.3. I also look at some alternatives to suffix arrays such as the inverted file structure in Section 1.4. Finally, in Section 1.5, I discuss various common topics in the literature surrounding suffix arrays such as their construction, compression, and use in search systems.

In Chapter 2 on page 32, I study two particularly interesting algorithms related to suffix arrays. This is a little like looking at how a brick is formed and baked. The most intricate part is where I describe Ko and Alru's suffix array construction algo-

rithm in Section 2.1. I have tried to write this section in a manner which is simpler than Ko and Alru's original paper [21]. Nonetheless, I suspect that it will probably be extremely difficult to understand without a substantial background in mathematics or computer science.[1] There is one extremely important thing about this algorithm which is the one thing I hope any biblical scholars or musicians who are interested in my thesis will understand (after which they may skip the chapter entirely). This important point is that this construction algorithm works in $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ space. This means that it will take twice as much time and twice as much space to process twice as much data. Although this may seem logical enough, computer scientists know that some algorithms can work at a much slower speeds such as $\mathcal{O}(n^2)$, which means that it takes four times as long to process twice the data. An example of this type of algorithm is given in Section 3.3.2. Having an algorithm that works in $\mathcal{O}(n)$ time and space makes processing extremely large documents fairly easily, but having one that works in $\mathcal{O}(n^2)$ time could make processing very large documents prohibitively difficult. This suffix array construction algorithm which I have described in Chapter 2 and implemented in Chapters 3 and 4 is a relatively fast $\mathcal{O}(n)$ algorithm. I also explain how the suffix array structure could be augmented to facilitate a fast approximate string matching routine in Section 2.2. The technique suggested in this section would be extremely expensive with current technology for very large sets of data, but it would work very well for smaller documents such as the Koiné Greek New Testament.

In Chapter 3 on page 51, I study a suffix array generated from one relatively popular Koiné Greek document: the New Testament. This is a little like studying the characteristics of a brick when it is used to build houses. I have tried to write chapter 3 of the thesis in a way that both computer scientists and biblical scholars can understand because my research is relevant to either field. But because there is a significant amount of jargon in both disciplines, this has been moderately difficult. I begin by showing many of the things a suffix array will bring to the surface in its raw

[1] I had a fair amount of difficulty understanding it myself when I translated the algorithm from English into C.

form in Section 3.2.5. Later, I show how a suffix array can be used in combination with a simple fuzzy-logic algorithm to find cross-references in Section 3.3. Processing the entire Greek New Testament took just 3.5 seconds to generate the suffix array and 10 seconds to create cross-references on my relatively slow 1.2GHz computer. And because the construction algorithm has a $\mathcal{O}(n)$ construction time, it is also quite fast to process much larger documents. Although a suffix array is somewhat useful and cross-references are especially useful for their own sake, I also explain how this research could be expanded to cross-reference other Koiné Greek documents and create a computer assisted translation system in Section 3.4.

In Chapter 4 on page 90, I study a suffix array generated from a particularly interesting type of music: a Bach fugue. This is a little like studying how a brick can be used to build bridges. A brick is not normally the type of material one uses to build bridges just as a suffix array is not normally the way one would search through or analyze music, but just as arch bridges made of bricks can serve as functional structures, so also suffix arrays can serve as a wonderful way to index music. I have tried to write chapter 4 in a way which is interesting to both computer scientists and musicians because the information presented is relevant to both fields. Unfortunately, it is prohibitively difficult to write it in such a way that it is fully understandable for people who cannot read music at all, however I have explained or avoided the use of technical jargon wherever possible.

A suffix array can be used as a building block to accomplish many things. In this thesis, I not only look at the suffix array, but I look at its application to Koiné Greek and music. Chapters 1 and 2 are designed to teach concepts, whereas Chapters 3 and 4 were written to prove them. Although the data presented here is useful in and of itself, it is my hope that the information will be used to create more intricate programs in both disciplines. The simplest way to continue my research into using suffix arrays with Koiné Greek would be using the same mechanisms to create a cross-references for a larger corpus than just the New Testament (pages 73–84). The most impressive way to expand this research would be to create a computer as-

sisted translation system designed specifically to translate the Greek New Testament into any arbitrary language as discussed on page 88. The research I have done by applying a suffix array to music is even more basic than the Koiné Greek research. As discussed on page 105, the simplest way to expand this research would be to use a different format for sheet music that would allow the use of a larger corpus. If such a corpus were analyzed using a suffix array, this database could easily be used to search for excerpts or count precisely how many times a particular excerpt is used in a relatively large body of music. Neither of these operations would be computationally expensive, so a single web-server with this type of database could serve hundreds or perhaps thousands of users.

This seems to be as good a time as any to remind the reader that both links and an index have been created in the pdf version of this thesis. If you're reading on paper, feel free to request an electronic version by e-mailing nathan at thru dot st. With this file it is possible to click on any reference in the contents or elsewhere, and browse through the contents in a side pane of a pdf viewer. The author is also quite happy to answer other inquiries.

Chapter 1

An Overview of Suffix Arrays

1.1 Background

This chapter is a survey of several issues related to suffix arrays. My particular interest is in using suffix arrays to search for n-grams (phrases) in statistical machine translation and cross-referencing systems, but this bias is fairly subtle until the introduction of my proposed Approximate Searching Algorithm in Section 2.2. The reader may also notice a greater emphasis on large alphabets than normal, and this is because I have used an alphabet of several thousand characters in Chapters 3 and 4. This first chapter is intended to be a reference for suffix arrays, an overview of all the major issues, and an illustration of the most interesting algorithms. The illustrations border on being too verbose rather than too terse because they are designed to allow the reader to understood how an algorithm works intuitively rather than formally.

This thesis uses several conventions related to variable names and definitions because there is actually a remarkable similarity between the choice of names and symbols in all the literature. The most important are as follows. The set of all the characters in the alphabet is defined by Σ. In the traditional case, $\Sigma = \{ a, b, c, \ldots \}$ but there are also several common variations. The alphabet may be case and punctuation insensitive, or something else entirely, such as $\Sigma = \{ A, C, G, T \}$ for a DNA sequence. Many algorithms only work if the size of the alphabet, represented by $|\Sigma|$, is relatively small. A special character, **$**, is usually defined as something that is lexicographically smaller than any character in Σ. The **$** is similar to the null character in a null terminating string. A suffix array T is an array of alphabet characters, and for every index i, $T[i] \subseteq \Sigma$. The string ends with a **$** and its length $n = |T|$. Thus, a $\mathcal{O}(n)$ construction algorithm varies asymptotically with the length of the input string. P is a string that is a pattern to search for in T where $P[i] \subseteq \Sigma$ for each index i in P and $m = |P|$. Therefore a $\mathcal{O}(m)$ search algorithm varies asymptotically with the length of the pattern searched regardless of the size of n. Finally, the suffix array itself is denoted by S. These variable names seemed to be used even in older suffix tree literature, so I suspect it has become a *de facto* standard.

The chapter proceeds as follows. Section 1.2 serves as a definition: it describes what a suffix array is and what it looks like. Section 1.3 reviews common uses for suffix arrays. Section 1.4 describes other algorithms that perform these same functions. Section 1.5 is the highlight of the chapter. It studies the three most common computational problems concerning Suffix Arrays: creating new arrays 1.5.1, searching through arrays 1.5.2, compressing arrays 1.5.3, and cache aware programing (though this has been prodominately ignored in the literature) 1.5.4. Finally, Section 1.6 critiques several papers which are also surveys of other papers. Each of the papers briefly serveyed in this section is an excellent reference.

1.2 Structure of the Suffix Array & Suffix Tree

Edward Fredkin introduced the idea of a "trie" in 1960 [13]. This structure is a type of tree, but he coined the word "trie" because it was designed primarily for

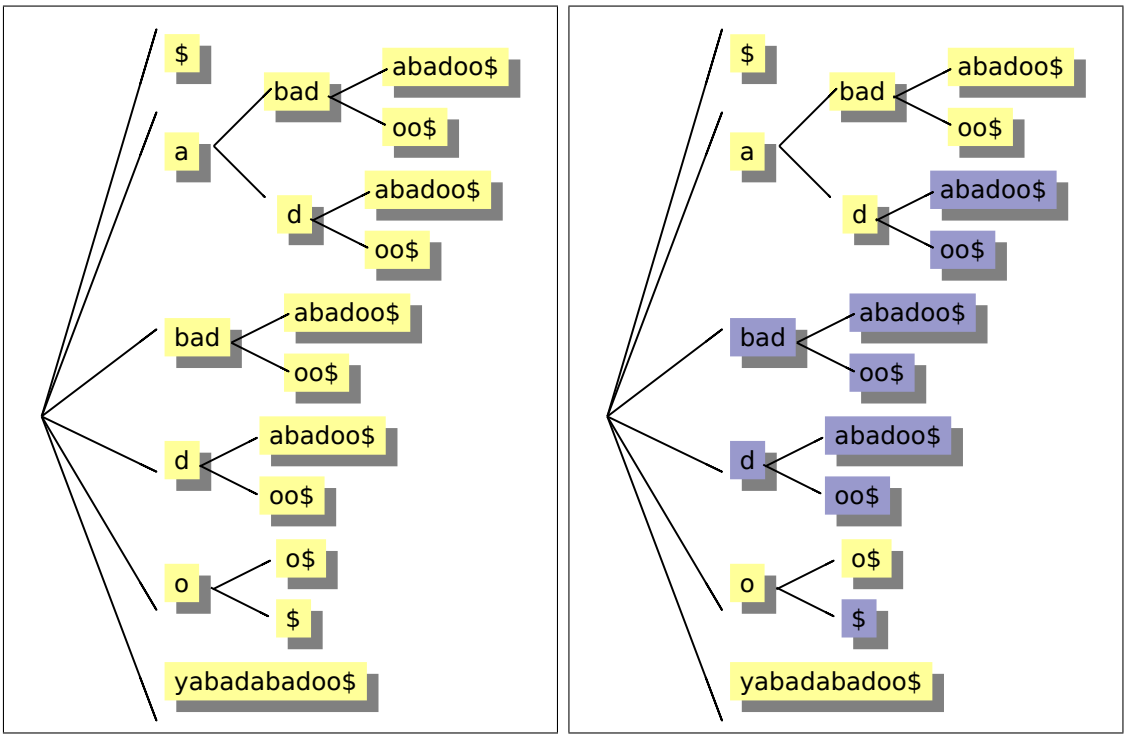


Figure 1.1: The suffix tree for "yabadabadoo$". The right image shows repeated subtrees in a darker colour.

re***trie***val.[1] Later, the idea of a suffix tree (often called a suffix trie) was developed. As the name implies, a suffix tree is a trie designed to retrieve all of the suffixes in a string. A suffix tree for $T =$"yabadabadoo\$" is represented in Figure 1.1. The root node is on the left side, and the nodes that are greyed are repeated—therefore the parent nodes may use a pointer to the previously defined sub-tree. This allows the Suffix Tree to be stored in $\mathcal{O}(n \log n)$ space.

Let's try to retrieve the suffix $P =$"badabadoo\$" from the suffix tree represented in Figure 1.1. From the root node, find the node that begins with "b".[2] Because this node contains "bad", we verify the first three characters P and continue searching for "abadoo\$", starting with the first character. This search can be performed within $\mathcal{O}(m)$ time or technically $\mathcal{O}(m\,|\Sigma|)$ time (depending on the implementation). It is easy to see that the search phrase, P, doesn't have to be a suffix. If P="badab" then the same procedure would return a find after all the characters in P are gone. It is also possible to detect a non-existent string by returning null after the first character is not found because every substring in T is stored in the suffix array.

Suffix Arrays

The basic structure of a suffix array is shown in Figure 1.2. They serve a similar function to suffix trees because both structures allow fast searching. In fact, it is possible to construct a suffix array from a suffix tree, and it is possible to use a suffix array as a suffix tree by creating additional data structures [1]. The primary advantage of the suffix arrays over suffix trees is that they can be represented in $\mathcal{O}(n)$ space instead of $\mathcal{O}(n \log n)$, and almost all the data structures which augment a suffix array for various algorithms—including those which provide suffix tree emulation—may also be represented in $\mathcal{O}(n)$ space. The reason for the difference is that a suffix

[1]The pronunciation of "trie" is traditionally the same as the pronunciation of "tree" due to its etymology, however some people pronounce it "try" to create a distinction

[2]The parent node may connected to its children in several ways. If the children are stored as a linked list, a search for a child can be done in $\mathcal{O}(|\Sigma|)$ time. If, however, a balanced binary tree or radix sort is used, it would take $\mathcal{O}(log\,|\Sigma|)$ or $\mathcal{O}(1)$ time respectively. The size of the alphabet is usually the primary factor to consider.

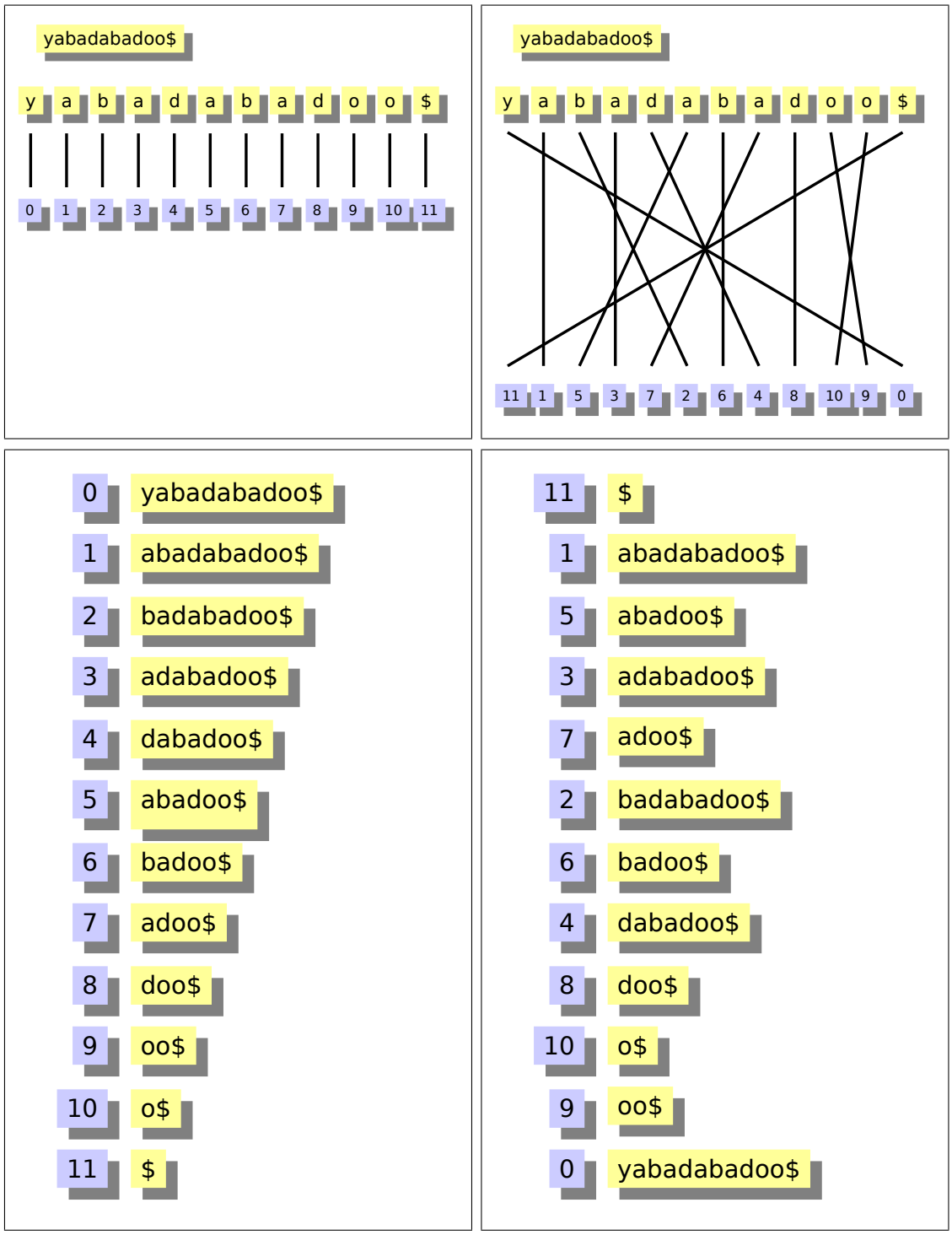


Figure 1.2: *Left:* An unsorted array of suffixes for "yabadabadoo$". *Right:* The properly sorted suffix array. *Top:* The pointers for each corresponding data structure. *Bottom:* The list of pointers in a data structure next to the string it dereferences.

array never repeats any characters in the text which can be seen by comparing Figures 1.1 and 1.2.

A diagram of a suffix array for "yabadabadoo$" is shown in Figure 1.2. The top two pictures show the suffix array data structure, and the bottom two show the set of strings represented by this data structure. Common knowledge of the C programming language will help illustrate the concept (see Figure 1.3).

```
#include<stdio.h>
int main()
{
  const char string1[13]="yabadabadoo$";
  char *string2;

  string2=&string1[4];
  printf("%s\n",&string1);
  printf("%s\n",string2);
}

Output:
yabadabadoo$
dabadoo$
```

Figure 1.3: A program to illustrate the concept of a suffix string.

In C, a string is typically defined as `const char string1[13]="yabadabadoo$"` or simply `char *string2`. String functions such as printf, expect to receive a pointer to a string, so it's quite possible to have many strings point to the same array of characters. For example, if I deference the location `string1[4]` and set string2 to this location, `string2=&string1[4]`, then `string2="dabadoo$"` even though no new memory is allocated. Thus, it is possible to have an array full of all the suffixes to a string without the need to repeat any of the actual suffixes.

Now presume that there is an array of suffixes S, each pointing a different location in a string. For each index i, define the array of suffixes as $S[i] = i$ as shown on the top left of Figure 1.2. Then the array will contain every possible suffix in order of appearance as shown in the diagram. But it is only becomes a suffix array after it

is sorted lexicographically in the right side of that image.[3] A binary search takes $\mathcal{O}(log\ n)$ time and a string comparison takes $\mathcal{O}(min(m, n))$ time where n and m are the length of the two strings. Therefore, after the array is successfully sorted, it is possible to search for a substring P of length m using in $\mathcal{O}(m \log n)$ time. For other examples of suffix arrays, the reader is also directed to Sections 3.2.1 and 4.2.1.

1.3 Basic Uses for Suffix Arrays

This section describes what suffix arrays are used for. Although suffix arrays are frequently used for searches, there are many common uses. The Burrows and Wheeler Transform is explained in some detail (Section 1.3.2), but the algorithms associated with constructing and searching through arrays are examined in Section 1.5. There are also several new potential uses described in Chapters 3 and 4 of this thesis.

1.3.1 General Searching

A suffix array is usually used for searching for a pattern P in a Suffix Array A to see if the pattern occurs, how many times it occurs, and where it occurs. Most people who use computers are familiar with searching for things; they look for words in file or on the web. It would be possible, for example, to search through an encyclopedia on a CD using a suffix array. Although suffix arrays can be used to search for text, the costs associated with a suffix array are often too high in the general case—especially with dynamic content. One of the reasons it is effective to search through the Koiné Greek New Testament with a suffix array is that the content never changes (see Chapter 3). However suffix arrays are particularly useful for searching for long patterns.

The suffix array shines whenever many queries have to made on the same data. This type of query is often used in Statistical Machine Translation (SMT), where

[3]Note that **$** is the smallest character, therefore the string "o$" < "oo$" even though all the other characters are the same. Because every suffix string has a different length ending in this unique character, no two suffix strings can be equal. This different from the rotating model proposed by Burrows and Wheeler in [4]. (See Section 1.3.2)

large texts of Parallel Corpora are search to find possible translations. A Parallel Corpus is a corpus of documents which have been translated from one language into another language. For example the Hansard (parliamentary transcripts) in Ottawa are translated into both English and French. As with any Parallel Corpus, this database can be mined to find the way words and phrases have been translated in the past. Research has shown that searching for longer phrases instead of shorter words generally produces better translations [22], and a suffix array is an extremely good structure for this type of query, because many searches for long phrases can be made quickly [5].

Natural Language is by no means the only data which can be searched. In fact, suffix arrays are particularly useful for searching through DNA sequences because the content of a DNA strand is long and static. One of the advantages of suffix arrays is that search queries don't have to be broken up into words to be indexed. DNA is a complicated language. It has an alphabet of only four characters, but not one of these characters is a space. The suffix array seems to be particularly good at searching for long DNA sequences which occur infrequently [42]. It is also possible to search for a tune in music, and this will be explored in Chapter 4.

Sometimes only the number of times a pattern occurs is needed, and this question can be answered extremely quickly by using a suffix array. For more information on implementing a search and count algorithm, see Section 1.5.2.

1.3.2 The Burrows and Wheeler Transform

One of the most interesting uses for the suffix array is the Burrows and Wheeler Transform (BWT); they published the transform in 1994 as "A Block-sorting Lossless Data Compression Algorithm" [4]. The most common program that uses this algorithm is bzip2. The ironic thing is that a BWT really doesn't compress anything at all, it simply converts data into something that's the same size, but the resulting data is far more compressible. Performing the first steps of a BWT is basically identical to sorting an array of suffixes to create a suffix array.

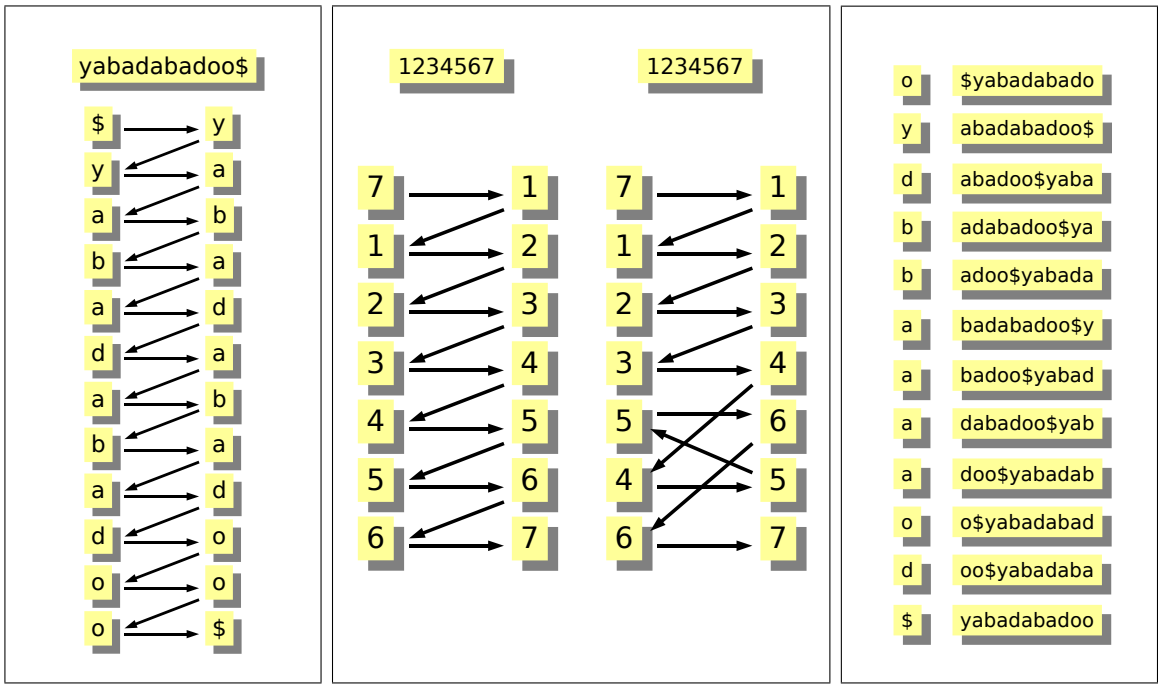


Figure 1.4: *Left:* A string can be regenerated if every set of two successive characters is known. *Centre:* This property holds even if the order of the sets of the sets of characters is scrambled. *Right:* The data from a BWT—each character previous to an entry in a suffix array is extracted.

Text from a BWT is very similar to a linked list. For example, consider the two columns of text shown in Figure 1.4. In each example in the figure, the left column represents a character, and the right column represents the next character in the string. The two columns can act as both data and pointers, and this is easiest to see if there are no repeated characters, such as the string "1234567". Starting at the top of the left column of this string (see the centre of Figure 1.4), we find a 7. The corresponding character at the top of the right column is the next character in the string, a 1. Although this is the character we wish to store to reconstruct a string, it also serves as a pointer. The 1 in the left column is the next digit in the linked list. Another way of thinking about it is that by linking every pair of characters in the string, (starting with 71, then 12, then 23, etc.) we can reconstruct the entire string. Thes is because the second character of each pair is always the first character of the next pair. Note that each character must be unique (for now—

eventually we'll convert a character into sets of characters), and a pair of characters takes twice as much space as the string. In this simplified case, it's easy to see that it's possible to reconstruct "1234567" even if the order of the pairs of characters is scrambled (see Figure 1.4). This is very similar to a linked list; regardless of where a particular node is stored in memory, it's possible to traverse the entire list as long as the pointer to the next node is valid.

Now consider the special case of a suffix array, such as the suffix array for "yabadabadoo$" shown in Figure 1.4. In this picture, the sorted order of each string is shown in the right column, and the character immediately preceding each entry is shown in the left column. The string is circular both for historical reasons and clarity.[4] The most amazing and innovative thing about the BWT is that given the string "oydbbaaaaod$" found in the left column, the entire string "yabadabadoo$" can be reconstructed!

This is understood best by working through an example, such as inverse BWT depicted in Figure 1.5. For each frame, the same encoded string is shown in the left column, and the next character is shown in the right column. The data in the left column was generated in the forward BWT described in the previous paragraph. This is the data we need to decode. The right column can be reconstructed using only the data in the left column by a simple counting sort. This works because the left column was generated by finding the previous character in a suffix array, and the first character for each string in a suffix array will always be equal to or greater than the first character in the previous suffix because a suffix array is sorted by definition. As the second column is being generated, the **$** terminating character is encountered (it is not necessarily the last character in the compressed string), and this is used as a starting point for the rest of the process. (Alternatively, the position of the **$** could be stored explicitly so that this special character wouldn't

[4]When the BWT was introduced in [4], there was no special terminating character that never occurred anywhere else in the array. Instead, the data was treated in a manner similar to circular buffer. With a terminating character that only occurs once in the data, a cyclical buffer will be sorted in the same order as an acyclical buffer. This is extremely helpful in special cases, where a string will repeat indefinitely. If one sorted "abcabcabc" cyclically with no terminating character, special care must be taken to make sure string comparisons don't loop forever!

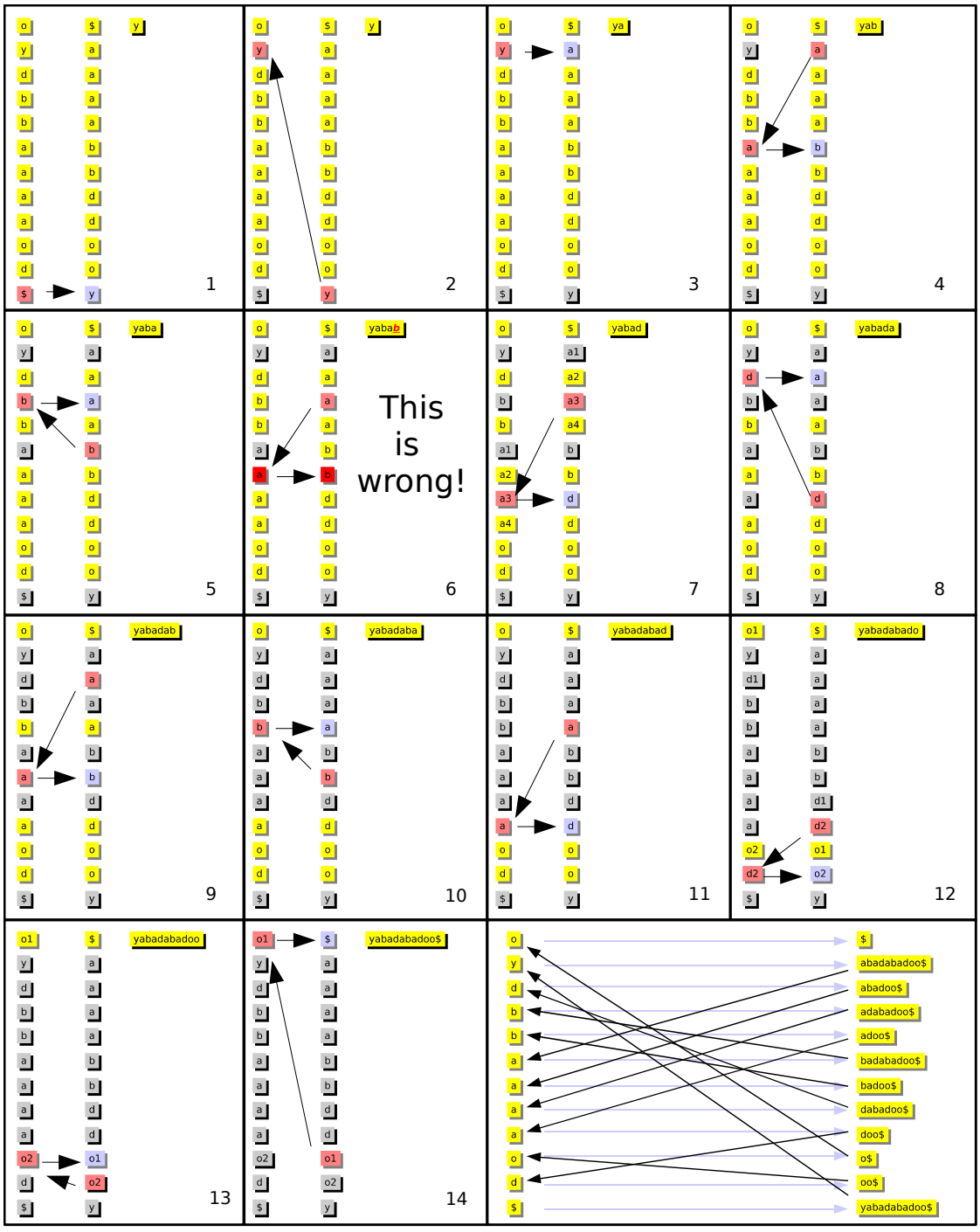


Figure 1.5: An example of regenerating “yabadababadoo$” from its scrambled form.

need to be part of the alphabet.)

After **$** is found, the next character will be the first character in the right column which also will be the first character in the string (see Frame 1 in Figure 1.5). Note that this structure is very much like the linked-list described previously. By constructing the right column, we have generated a set of character pairs from a set of characters. The first character in the string is "y". Now we use this as a pointer to find the "y" in the left column (Frame 2). This corresponds to the next character, "a" giving us a string of "ya" (Frame 3). This process repeats effortlessly until frame 6, where "yabab" is generated instead of the correct "yabad". This error is caused by the fact that the right column is a single character instead of a true pointer, and almost every compressible string contains repeated characters. (See Section 1.5.3 for an intriguing exception.)

The solution is simple, as seen in Frame 7, each character must be counted, and the first "a" in the left column corresponds to the first "a" in the right column, the second to the second, etc. However the reason this works is not particularly simple. Looking back at the suffix array for "yabadabadoo$" in Figure 1.4, one can see that the order of characters in the encoded text is based on sorting the entire string rather than simply sorting the left-most character we use for decoding the string. For example, the string "abadoo$" appears before "adoo$" in the list even though the first characters in both strings are both "a". This is because the rest of the string, "badoo$"<"doo$". There are several "a"s in the left column, but the one corresponding to "badoo$" occurs before the one corresponding to "doo$" because these suffixes are also sorted. One can also look at this backwards. There are two suffix strings in the right column preceded by an "d" in the left column. It would be possible to remove all the rows of strings that are not preceded by a "d" leaving only "abadoo$" and "oo$". It would be possible to add the first character "d" to the two strings (perhaps by reading across both columns), and the suffixes "dabadoo$" and "doo$" are created. Now, by looking in the right column of the entire suffix array, you will find that both of these suffixes already exist. In fact,

each “d” in the left column has been encoded in the same order in the right column, which is why this principle works. This is why it’s possible to extract the data, and use the single characters as pointers even when they are not unique. Using this principle, it is possible to follow from Frame 7 in Figure 1.5 until the entire string is reconstructed when the last **$** is reached in Frame 14. Yabadabadoo!

Compression using the BWT

The most unusual thing about the BWT compression mechanism is that the transform doesn’t actually compress the text! Instead, a stream of the same size is generated, and this stream is more compressible. By looking at Burrows and Wheeler’s original example in Figure 1.6, you can see that many of the sentences using the

final char (L)	sorted rotations
a	n to decompress. It achieves compression
o	n to perform only comparisons to a depth
o	n transformation} This section describes
o	n transformation} We use the example and
o	n treats the right-hand side as the most
a	n tree for each 16 kbyte input block, enc
a	n tree in the output stream, then encodes
i	n turn, set $L[i]$ to be the
i	n turn, set $R[i]$ to the
o	n unusual data. Like the algorithm of Man
a	n use a single set of probabilities table
e	n using the positions of the suffixes in
i	n value at a given point in the vector $R
e	n we present modifications that improve t
e	n when the block size is quite large. Ho
i	n which codes that have not been seen in
i	n with ch appear in the {\em same order
i	n with ch. In our exam
o	n with Huffman or arithmetic coding. Bri
o	n with figures given by Bell~\cite{bell}.

Character	Alphabet	Encoded
a	aeiou	1
o	aeiou	4
o	oaeiu	1
o	oaeiu	1
o	oaeiu	1
a	oaeiu	2
a	aoeiu	1
i	aoeiu	4
i	iaoeu	1
o	iaoeu	3
a	oiaeu	3
e	aoieu	4
i	eaoiu	4
e	ieaou	2
e	eiaou	1
i	eiaou	2
i	ieaou	1
i	ieaou	1
o	ieaou	4
o	oieau	1
In:	aooooaaiioaeieeiiioo	
Out:	14111214133442121141	
Frequencies:	a: 20% e: 15% i: 30% o: 35% u: 0%	1: 50% 2: 15% 3: 10% 4: 25% 5: 0%

Figure 1.6: *Left:*This example is quoted directly from Burrows and Wheeler’s paper paper [4]. It depicts the result of performing a BWT on their own paper. It is also the only internal evidence that their paper was produced using LaTeX. *Right:* This chart shows how the text on the left would be encoded using a move-to-front buffer with an alphabet of “aeiou”.

word “on” and “in” occur close to each other in the text. It’s also possible to see

that any word ending in an "n" has a vowel as the second last character. After the transform, similar phrases occur together, and so the probability of one character being identical to the previous character increases greatly. It would be very simple to compress this is using Run Length Encoding, but usually a different mechanism is employed.

In order to exploit the repetition in data, a move-to-front buffer is often used. For example, let's say we have an alphabet of only vowels i.e. $\Sigma = \{a, e, i, o, u\}$ and each letter is represented by it's position in Σ, so $a = 1$, $e = 2$, etc. Let us write the sequence "aooooaaiioaeieeiiioo" from Figure 1.6 using a move-to-front buffer. The first character, an "a", would be written as a 1, and because it's already in the beginning of the alphabet it would remain in position 1. The second character is an "o". It is stored as a 4 because at the time it's written, it's in the fourth position of Σ. Before it's written, $\Sigma = \{a, e, i, o, u\}$, but after it's written, $\Sigma = \{o, a, e, i, u\}$ because the letter "o" is moved to the front of the buffer, and every other letter is moved back one position. The string can easily be reconstructed using the same move to front principal because the alphabet is never changed until after a character is written. In fact, in Figure 1.6, the entire string may be encoded or decoded using the table. The frequency table reveals that half of the time the output string contains a 1 because half of the time the input repeats. Once again, the output of the transformation is the same size of the input. However this resulting string can be encoded using Huffman or arithmetic encoding extremely effectively because the real input alphabet has 256 characters, not just the vowels, and yet the process would still result in about half the characters in this sequence being a "1", meaning these characters could be encoded using only one bit! The implementation of arithmetic encoding is beyond the scope of this thesis.

1.3.3 The Longest Common Prefix

An auxiliary array for the Longest Common Prefix (LCP) is often used to search for patterns in an array of text. An example of how this data can be used to interpret

a suffix array constructed from Koiné Greek is given in Section 3.2.2. The LCP structure is also be used as one part of a system to emulate a suffix tree using a Suffix Array [1, 9]. It is possible to create the LCP array in $\mathcal{O}(n)$ time [18]. This array is also a byproduct of some suffix array creation algorithms [29]. The LCP has a simple structure, shown in Figure 1.7. Each index of the LCP contains the number of characters which are identical to the previous suffix in the suffix array. For example, referring to Figure 1.7, the prefix "bad" is contained in both "badabadadoo$" and "badadoo", so the LCP for "badadoo$" is 3. The LCP for the suffix "badabadadoo$" is 0 because its prefix shares no characters in common with "adoo$".

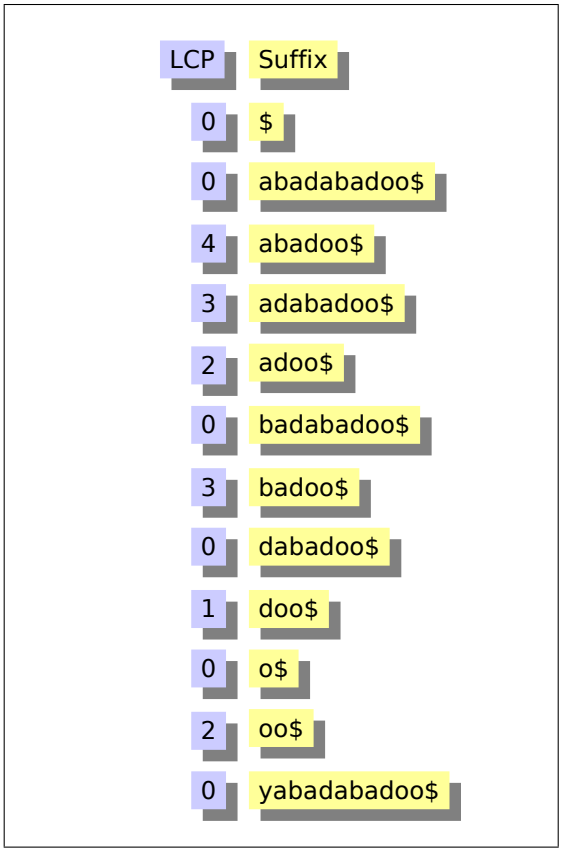


Figure 1.7: The Longest Common Prefix for each line in the suffix array of "yabadabadoo$".

It is more difficult create a suffix array of a text with high LCP's because many of the suffixes require many compare operations to resolve potential conflicts. Some

creation algorithms tolerate this better than others. The fastest creation mechanism has safeguards to detect these special cases [29, 24]. The mean LCP of some texts is very high: human chromosome 22 has a mean LCP of almost 2000 characters [29]. Data with high LCP's are often the most interesting part of the array. Yamamoto and Church used the LCP structure to find all of the "interesting" substrings in the Wall Street Journal and a Japanese corpus [38]. The suffix array and LCP are used to find such phrases, and then each phrase is tested based on other statistical measures (mutual information and residual inverse document frequency). They found interesting results. For phrases containing "the" they found "the up side", "the will of", "the saying goes". These principles could be applied to finding patterns in musical compositions, and this will be explored in Section 4.1.1.

1.4 Alternatives to Suffix Arrays

Although suffix arrays have many advantages, they also have disadvantages. They are able to ignore spaces and search starting in the middle of words, which is terrific for analyzing DNA, but not necessarily the best thing for text. It is possible to convert every character to lower case, and every punctuation mark between words to a space in order to create a more practical, case insensitive search or perform even more preprosessing for approximate string searches (See Section 2.2), but when one searches for a "red rose" they usually don't want the references to "bored roseanne" that a suffix array will return. However this limitation can be overcome by using a large alphabet where each letter represents a word as explained in Section 3.1.1. Another restriction, the random access pattern which makes suffix arrays unsuitable for today's secondary storage technology, may disappear in time if computers begin to run on flash memory instead of traditional hard disks. Professional photographers no longer use large compact flash micro-drives because Flash technology has improved, so one can imagine a day when the hard drive becomes obsolete.

1.4.1 Basic String Matching

One cannot completely ignore the basic algorithms for exact string matching, similar to that used by grep. The amount of time it takes to construct a suffix array makes it inefficient to construct the array to do a small number of queries. The simplest string matching algorithm runs in $\mathcal{O}(n\,m)$ time, and simply compares every character in the search pattern to every character in the text. However the Knuth Morris Pratt (KMP) algorithm uses an array to ensure that every substring of a pattern can be reconsidered if a match fails, and this algorithm runs in $\mathcal{O}(n+m)$ time [20].[5] With some of the advanced suffix array structures presented in Section 1.5, it would also be possible to create a suffix array and perform a search all in $\mathcal{O}(n+m)$ time, but it would take several dozen different searches in the same text before this actually proved more efficient. Perhaps it would be appropriate for a pdf viewer to create a suffix array in the background as a new file is loaded in anticipation of a user's search, but it would be inefficient for a file editor to do this. Because the KMP algorithm works on a stream of characters, it can easily be used to search efficiently for relatively small patterns in compressed documents or documents which are too large to fit in RAM. This is the simplest type of searching algorithm and it's often appropriate to use it.

1.4.2 Suffix Trees

Edward Fredkin introduced the idea of a suffix tree as mentioned in Section 1.2 [13]. They occupy $\mathcal{O}(n \log m)$ space, but can perform $\mathcal{O}(m\,|\Sigma|)$ or $\mathcal{O}(m \log |\Sigma|)$ searches depending on their implementation. The first $\mathcal{O}(n)$ suffix tree construction algorithm was found six years before the first suffix array construction algorithm—in the intervening years one could construct a suffix array most efficiently by first constructing a suffix tree [35, 29]. It's also possible to create dynamic suffix trees, which makes them more attractive than suffix arrays for some applications [25].

[5] This is one of two documents which are found in all the literature, but I have not yet found a copy myself.

However, it's possible to merge two suffix arrays together in constant time [16]. By using this technique, it may be possible to have a small suffix tree-based work corpus to handle data that changes frequently beside a more permanent suffix array corpus. The suffix tree could then be appended to the suffix array either every night or whenever the processor is idle, and then a new suffix tree could be created for the next set of changes.

Because a suffix tree is a tree structure, it is sometimes necessary for some of the structure to reside in secondary storage. By using Patricia tree's (which are closely related to suffix trees) in combination with suffix arrays, Ferragina and Grossi were able to create an SB-tree data structure which works very well in practise [11]. Many of the best algorithms for searching suffix arrays described in Section 1.5.2 are actually emulating a suffix tree [9, 1]. Presuming the data is static, it is usually more efficient to use a suffix array to emulate a suffix tree. Another possible exception would be imprecise string queries. In 1993, Ukkonen proposed three methods for augmenting suffix arrays to allow searching for strings that are close to a particular pattern [36]. Although Ukkonen suggested changing his algorithm to use suffix arrays, I don't think this has been done. However, I propose an entirely different approach in Section 2.2. Although the suffix array is Superior to a suffix tree in many ways—especially since the former can emulate the latter—the suffix tree remains more useful in some circumstances.

1.4.3 Inverted Files

No overview of information retrieval would be complete without mentioning the inverted file (sometimes called an inverted index). It is a broad term for a database of search terms (usually words), and each search term is linked to a specific file. Often, the database may also store information about the precise location of each word instead of simply the file location, and this is the type of index which is closest to the suffix array. This is the type of system that almost all web search engines use. Signature files are very similar, but instead of storing a particular search term, they

store the hash of a search term. If two words have the same hash, they share the same database entry. False positives are filtered out later. This structure has proved to be both larger and slower than the inverted file [42]. Perhaps the most recent and comprehensive overview of Inverted files was created two years ago by Zobel and Moffat [41]. In it they point out many of the virtues of Inverted Files and a few of the deficiencies of suffix arrays.[6] Inverted files may become several times larger than the text they index, but it is possible to created a compressed inverted file. One such datastructure uses wavelet tree's to store the inverted file along with enough auxiliary data to store the data itself [3]. The paper describing this data structure has one of the best comparisons between a Huffman tree and a wavelet tree, and it shows why the latter is more efficient; the compression techniques presented could apply to suffix arrays as well.

Inverted files use many different mechanisms to actually store the database of terms. These terms may be individual words or phrases, and they may also correct spelling errors or wild-card searches. Phrases are often restricted to two words, and then a three word phrase is found by searching for the intersection of words 1 & 2 with words 2 & 3. Hash tables or tries may be used to store the indexes. A bloom filter may be used to see if a word exists in the database or not to avoid checking a disk database for a term which doesn't occur. A bloom filter will occasionally "lie" and say that a term is in the database when it really is not, but when it says the term is not in the database it is always truthful, so it will usually avoid a costly disk read. The best set of compromises for a large inverted file which is stored primarily on a disk is probably the burstsort [32]. This uses several structures together, and it is also conscious of the processor cache (see Section 1.5.4).

The inverted file is not restricted only to applications which have definite word boundaries. In order to search for other terms, an arbitrary word length is chosen and strings of characters are generated from this. Then to search for a smaller term, the index of all the artificial terms with a common substring are searched. For

[6] Although one of the deficiencies they site, the inability to search compressed data, was solved several years earlier [12, 10]. See Section 1.5.3

example, to search for "red", the six-character terms "paired", "spared", "shreds", and "redact" would all be searched. For terms longer than six characters, the intersection of common substrings would be searched in the same manner as phrase searches.

Most languages and language searches do have distinct word boundaries. But in the domain of searching DNA which is often ascribed to the suffix array, inverted files augmented using these techniques still perform extremely well [31]. This experiment compared the compressed versions of both the inverted file and the suffix array, and using uncompressed versions of these algorithms would probably lead to a different outcome. Nonetheless, it shows that suffix arrays seem to be better at finding a few long string matches than inverted files, and inverted files seem to be better and finding a large number of small matches.[7] In any case, the compressed suffix array is far, far faster at finding the precise number of times a pattern occurs than the inverted file.

1.5 Hot Topics about Suffix Arrays

A number of topics relating to suffix arrays are ubiquitous in the literature, and many other topics are almost entirely ignored. The problem of constructing suffix arrays seems to be particularly common, perhaps because it is the most difficult step in compression schemes using the BWT. These few hot topics are certainly interesting problems, but I suspect that the colder topics I have addressed in Chapters 3 and 4 of this thesis will also prove very interesting to the reader.

[7] I suspect this is due to the compression. An uncompressed suffix array should reveal a simple list of pointers, and each one will correspond to a location in a similar manner to inverted files.

1.5.1 Constructing a Suffix Array

An Overview of Algorithms

There are many different algorithms which are used to sort suffix arrays. They are described quite thoroughly in Pugilist, Smyth, and Turpins "Taxonomy of Suffix Array Construction Algorithms" [29]. It is an extension of their earlier work which only compared $\mathcal{O}(n)$ algorithms [30]. This thesis not only reviews most of the algorithms, but also explains the chronology, relationship and origin of these techniques. By classifying all these algorithms, it is a true taxonomy.

The two dominate strains of algorithms strains of construction algorithms, are analogous to quicksort and heapsort. The algorithms similar to quick sort are extremely fast in practise, but have a very poor worst case. (If previously sorted data is passed to quick sort, it's normal $\mathcal{O}(n \log n)$ complexity drops to $\mathcal{O}(n^2)$.) The fastest array construction algorithm in the experiments takes up to $\mathcal{O}(n^2 \log n)$ in the worst case, though perhaps it's real asymptotic complexity has yet to be proved [24]. The algorithm is fast because it combines the best part of many other sorting techniques with cache conscious programming and safeguards against the worst-case running times. The first step is taken from an improved version of the $\mathcal{O}(n)$ algorithm we will look at in Section 2.1. Subsequent steps use a version of quicksort that detects a problem after 48 recursive calls and diverts to heapsort.

All of the $\mathcal{O}(n)$ algorithms work on the same principal. They sort a subset of the array recursively, usually $\frac{1}{2}$ to $\frac{2}{3}$ of the elements, and find the position of the remaining elements using data from the sorted section. This recursion means that the algorithms run in $\mathcal{O}\left(n\left[1 + (\frac{2}{3})^1 + (\frac{2}{3})^2 \ldots\right]\right)$ which is a geometric series that resolves to $\mathcal{O}(3n) = \mathcal{O}(n)$. Although this seems very attractive, the fastest $\mathcal{O}(n)$ algorithm takes 2.5 times longer to run on average than the fastest algorithm $\mathcal{O}(n^2 \log n)$ algorithm [29]. Nonetheless, this particular algorithm is interesting enough to merit its description in Section 2.1.

It's important to note that many of these algorithms use a bucket-sort mechanism which is inappropriate for a large alphabet size. The fastest algorithm according

to experiments [29] is particularly troublesome. In their paper introducing the algorithm, Maniscalco and Puglisi say, "Finally, it is important to note that this sampling method is only suitable for applications in which $|\Sigma|^2$ is a manageable size—fortunately this is most often the case" [24]. There are exceptions where the alphabet is unmanageable, such as the approximate string matching algorithm described in Section 2.2 which works most efficiently if each word in a text is given a number similar to a Chinese character set. A linear time algorithm introduced by Na overcomes this limitation [26]. Although he requires $o(n \log |\Sigma|)$ bits of working space to do this, the original text requires $n \log |\Sigma|$ bits of space to store—so this restriction is no restriction at all.

One final algorithm which is particularly interesting runs in $\mathcal{O}(n \log \log n)$ time, but it uses a distributed system to generate the array [19]. To create parallelisation, it splits the array into multiple pieces and works out the middle. Tests show that it speeds up the process almost optimally. Although the text itself isn't distributed through the network, it would be interesting to see if this algorithm could be adapted from a distributed system to a multiprocessor system.

1.5.2 Searching a Suffix Array

When one searches for a pattern, they are generally looking for one of three things: whether the pattern exists, how many times the pattern exists, or where the location of each instance is. Suffix arrays can perform all of these searches very effectively. In particular, suffix arrays are good at the counting problem because of the way the indexes are stored. For example, referring back to Figure 1.2, one could search for the string "bad" in "yabadabadoo$" by locating the 6th & 7th terms of the suffix array which both begin with "bad". All of the suffixes below the 6th index are lexicographically less than "bad", and all those after the 7th are greater. The fact that there's two items can be deduced from the fact that they occupy the spaces $6 \ldots 7$. This makes it possible to count every instance of "bad" or any other substring in the same time it takes to make two searches.

The simplest method to search a suffix array is the same as the simplest method to search any other array structure: the binary search. This allows a string to be found in $\mathcal{O}(m \log n)$ time where m represents the length of the pattern being searched. Each subsequent string can be found in $\mathcal{O}(m \log n + f)$ time where f represents the number of matches. However there are several auxiliary structures which may be used in order to reduce the search time to $\mathcal{O}(m)$ where time where the alphabet size is constant.

The first of these algorithms was published by Abouelhoda, Ohlebusch and Kurtz in 2002 [1]. They used various data structures, including the Longest Common Prefix (see Figure 1.7), in order to emulate a suffix tree using a suffix array. With this structure, a search could be made efficiently with a small alphabet in the $\mathcal{O}(m\,|\Sigma|)$ time. This was extended by two papers which appeared in the same issue of the Korean "Journal of KISS: computer systems and theory" [17, 39]. Both of these algorithms use a rather novel datastructure that allows searches in just $\mathcal{O}(m \log |\Sigma|)$ time which drastically increases the search speed for large alphabets. The same group of people produced an algorithm which augments the suffix array to emulate a suffix tree, but using a more advanced structure that still allows searches in $\mathcal{O}(m \log |\Sigma|)$ time [9]. For large alphabets, this is probably the most efficient structure for searching so far.

1.5.3 Compressing a Suffix Array

Suffix tree compression was first introduced by Ferrangina and Manzini in 2000 [12, 10]. The remarkable thing about their compression scheme is this: not only is the compressed index structure smaller than the original text, but the original text can be decompressed using only the compressed index! In other words, the compressed suffix tree can replace both the index *and* the original data.

Explaining these compression structures is beyond the scope of this thesis, but an excellent overview of many self-indexing datastructures was recently created by Navarro and Mäkinen [27]. An intuitive example of why the index is compressible

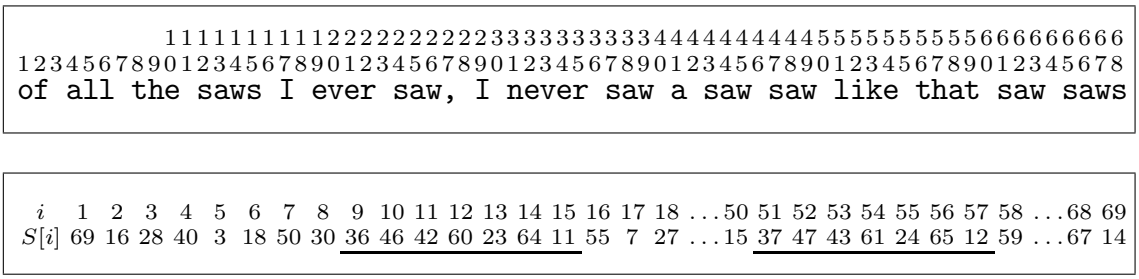


Figure 1.8: An example of why a suffix array is compressible, quoted from Puglisi, Smyth, and Turpin [31].

is presented in Figure 1.8. This contains the text on the top, which results in the suffix array on the bottom. The underlined portion is a sequence of numbers which is identical to the next sequence of numbers, except that the second time each number is incremented by one value. Following the pointer $S[i]$ back to the text, one can see that each of these suffixes begin with "␣␣␣s" (where ␣␣␣ is used to define a space). It is this type of repetition which these compression algorithms take advantage of.

1.5.4 Cache Aware Programming

This is actually not a topic which is frequently addressed in the literature, but it is something I have tried to consider as I wrote my own suffix array construction and processing algorithms. One of the reasons that the fastest suffix array construction algorithm is so fast, is that Maniscalco and Puglisi used cache conscious programming techniques [24]. In a similar manner, Shinha and Zobel created a dynamic trie structure that was also aware of the cache [32]. The parameters of their (now almost archaic) machines are shown in Figure 1.9. Using `lmbench`, I performed some experiments on my own computer (a 2GHz Athlon 64 with DDR400 RAM), and discovered that an L2 cache miss used at least 60ns of time, which works out to 120 instructions. Processor manufactures don't often disclose the cost of a cache miss, but there are various techniques used for detecting this latency along with other parameters. One automatic technique performs significantly better than lmbench at detecting a variety of parameters, and it is described in [40]. It is telling that the only cache conscious construction algorithm is also the fastest.

Workstation	Pentium	Pentium	UltraSPARC	Power Mac G5
Processor type	P III Xeon	P IV	USPARC III	PowerPC 970
Clock rate (MHz)	700	2000	750	1600
L1 cache (KB)	16	8	64	32
L1 block size (Bytes)	32	64	32	64
L1 associativity	4	4	4	2
L1 miss latency (cycles)	6	7	12	8
L2 cache (KB)	1,024	512	8,192	512
L2 block size (Bytes)	32	64	512	128
L2 associativity	8	8	1	8
L2 miss latency (cycles)	109	285	174	324
Memory Size (MB)	2,048	2,048	4096	256

Figure 1.9: A description of the architectural parameters of the machines used by Shinha and Zobel, quoted directly from their paper [32].

Finally, with the advent of multi-core processors, it's important to remember that the bottleneck of the suffix array is the memory latency rather than the memory bandwidth or processing speed. However, because a dual-channel memory system uses a 128-bit (16-Byte) wide data cache, if all of the pointers which are associated with each other are stored in the same part of memory, this could greatly increase efficiency. In fact, the cache line will usually be 64 bytes wide on modern machines.[8] There are also cache-prefetch instructions for both AMD and Intel processors, however these work better on sequential data—with a suffix array one doesn't often know what memory they will need until the time comes.

1.6 A Survey of Surveys

There are many surveys of datastructures which can be used for information retrieval. Each of these documents is in this section because it does not primarily propose an algorithm but instead compares algorithms which were made previously. Although I'm not sure if it's been published formally, Pickens has an excellent paper which gives an overview of different music representation schemes [28]. Zobel and

[8] I realize that I run the risk of calling something modern that the reader will now consider antiquated.

Moffat created an excellent overview of the Inverted File structure [41]. It is to inverted files as this chapter is to suffix arrays—except that their paper is longer, more articulate, and has better research. With the help of Ramamahonarao, they also confirmed that the Signature File is almost always inferior to an Inverted File [42].

There are a few papers which focus on suffix arrays. Navarro and Mäkinen give a recent and extensive overview of compressed suffix arrays [27]. Puglisi, Smyth, and Turpin recently compared the speed of inverted files to the speed of suffix arrays for searching DNA strands [31]. The trio also analyzed linear-time suffix array construction algorithms [30]. The following year, in 1007, the three created the best overview available: "A taxonomy of suffix array construction algorithms" [29]. This document looks at the history of all of the various techniques.

1.7 An End of an Overview

Suffix arrays have a number of uses which have been described in this chapter. The chapter is primarily a reference and a broad overview of all the literature, and any missing information will probably be addressed by other papers listed in Section 1.6. Much of the literature addresses how suffix arrays are used, but most of it seems to focus instead on the problems of creating and searching suffix arrays. One of the best, and most interesting algorithms is described in Section 2.1, and a new way to use a suffix array to perform approximate string matching on a small query is highlighted in Section 2.2. Although this technique is extremely fast, it gains its speed primarily at the cost of memory usage. Nonetheless, the system would be useful for searching through small documents today, and larger documents tomorrow. The suffix array has received much greater attention in the past decade due to DNA research, and the Burrows and Wheeler Transform described in Section 1.3.2. Although most small searches are still done by scanning the entire text, and most large search systems still operate using inverted files, the suffix array has enough value to genetics and Statistical Machine Translation that it is here to stay.

Chapter 2

Two Suffix Array Algorithms

2.1 Ko and Alru's $\mathcal{O}(n)$ Construction Algorithm

Arguably, the most interesting suffix array creation algorithm was introduced by Ko and Aluru in 2003 [21]. This is the fastest $\mathcal{O}(n)$ algorithm for generating a suffix array [29]. This section will walk through the process of sorting the array. As with all suffix array sorting algorithms, we wish to sort the suffix array A that indexes the data T. There is also a hidden reverse list R which allows a procedure to find the current index into the suffix array A by knowing the index in T. This array must change as A is sorted, so that at any time $R[i] = k$ iff $A[k] = i$. The paper distinguishes between a character at an index by t_i and an entire string starting at an index by T_i with the different capitalization, and I will follow this convention.

Of all the sections in this thesis, this one may be the most difficult to understand. It is labourious to decipher the original paper, and I have endevoured to write something more comprehencible. The algorithm contains several unique data structures which were cumbersome to implement, and I will gloss over these details, focussing instead on a description of the algorithm which is easier to understand intuitively.

The fascinating thing about this algorithm is that it works. I implemented it to perform the tests presented in Chapters 3 and 4 of this thesis. The piece of music I processed was too short to generate reasonable timing statistics, but my program created a suffix array from a Koiné Greek document, The New Testament, in just 3.5 seconds on a Pentium 4M 1.2GHz computer. By concatenating five copies of this document together, my implementation also proved the $\mathcal{O}(n)$ properties of this algorithm. Repeated tests showed that the program took four times as long to process five times the data.

2.1.1 Step 1: Finding L-Type and S-Type Suffixes

Let us define a L-Type suffix as any suffix where $T_i > T_{i+1}$, and a S-Type suffix as any suffix where $T_i < T_{i+1}$. Thus a S-Type suffix is *Smaller* than the following suffix in the string, and a L-Type suffix is *Larger*. It is possible to go through an

entire suffix array from left to right and find every L-Type suffix and every S-Type suffix in one pass. It is rather obvious that if first characters of two suffix strings $t_i < t_{i+1}$ then the entire strings $T_i < T_{i+1}$. What is less obvious is what to do if the characters $t_i = t_{i+1}$. Because of the terminating "$", no two suffixes can be equal, so how does one figure out whether one suffix is bigger or smaller?

This is how. If one starts counting from left to right using a new variable k starting at $k = i + 1$ until there's a new character which means $t_k \neq t_i$. At this point, the string $T_{i...k}$ has one more character than $T_{i+1...k}$, so we need to compare the substring $T_{i...k-1}$ with $T_{i+1...k}$. We already know that every character in $T_{i...k-1}$ is identical because this string only goes to the position $k - 1$, and only the very last character t_k is unique. This is the key. If $t_i > t_k$, then $T_i > T_{i+1}$ making this a L-Type suffix. Not only this, but every other suffix from $i \ldots k - 1$ is also a L-Type suffix.

Let's use the example in Figure 2.1. The index i will start at position $i = 0$, and $t_i > t_{i+1}$ because "y">"a", so this is a "larger" L-Type suffix. Increment i so that $i = 1$, and $t_1 < t_2$ because "a"<"b" making this a "smaller" S-Type suffix. And so on. But things get interesting again once $i = 9$ because the two suffix strings T_9 ="oo$" and T_{10} ="o$" share the same first character: $T_9 = T_{10}$ ="o". Starting k at $k = 11$, one finds that $t_{11} < t_9$ because "$" is smaller than anything, and we haven't advanced yet, so $k = 11$. This makes position $i = 9$ an L-Type suffix, because "oo$" is larger than "o$". After advancing to $i = 10$ which is still $i < k$,

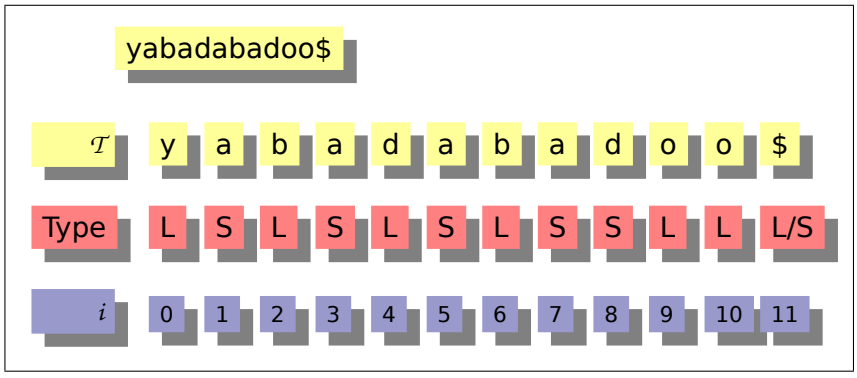


Figure 2.1: All of the "Larger" L-Type suffixes and "Smaller" S-Type suffixes.

the next suffix strings T_{10} and T_{11} are "o$" and "$". They are T_9 and T_{10} with the first character removed: "oo$"→"o$" and "o$"→"$". We know that these missing characters are equal, because this is why we needed to define k and start looking for a different character. This means that removing the first character from each string will not change which one is greater. In other words, if $t_i = t_{i+1}$ and $T_i > T_{i+1}$ then $T_{i+1} > T_{i+2}$. This is why every character between i and $k-1$ is changed to be L-Type or S-Type based only on tk being different (see Figure 2.1). This step only takes one pass, so it is $\mathcal{O}(n)$.

2.1.2 Step 2: Sorting all the S-Type Suffixes

It is possible to sort the entire array if either the S-Type suffixes or the L-Type suffixes are sorted.[1] After counting the two types of suffixes in Figure 2.1, we find 7 L-Type suffixes but only 6 S-Type suffixes, therefore it is preferable to sort the S-Type suffixes (it is possible to sort either type).

This process is shown in Figure 2.2, starting on the left side. Let us define a new array, the S-Dist, which measures the distance to the most recent S-Type Suffix. The S-Dist starts at zero to indicate values which are not part of any S-Type suffix, and thereby do not need to be sorted. The first important position is at $i = 2$. This is an L-Type suffix which is 1 suffix away from an S-Type suffix. The index $i = 3$ is an S-Type suffix itself, but it is still deemed two positions away from the nearest S-Type suffix.[2] These are used to define the positions in S-Type substrings, also shown in Figure 2.2. We wish to sort these using a bucket sort strings character by character, starting at the first character in each substring and working to the last character of the longest substring.

The first step after finding the S-distances (which is not shown until Figure 2.4)

[1] In fact, Mori discovered a technique that requires only the right-most suffix in each series of repeated S or L type suffixes needs to be sorted. He did not formally publish his work, but it is described in [24].

[2] This follows the paper [21], though I orinally suspected it would be possible to use a zero value for these suffixes. I changed my mind after trying to generate a suffix array from the string "assassin$"..

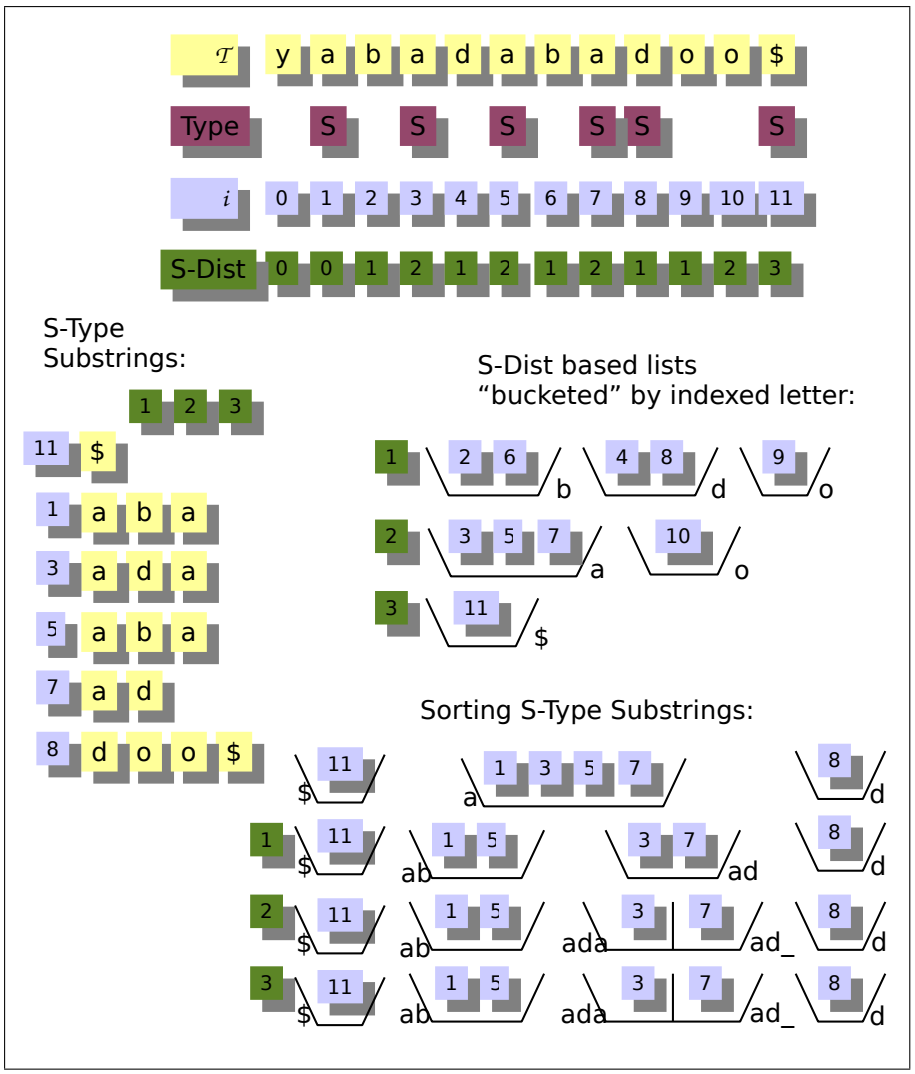


Figure 2.2: The process of all the sorting S-Type substrings.

is performing a bucket sort of the entire array. At this point, we only care about the suffixes starting with "a" being before those starting with "b" (later we will care about S and L-type suffixes). Next, we will extract all the S-Type indexes and put them in a special set of buckets which can be subdivided into smaller buckets (without leaking). This is shown in the top row of the "Sorting S-Type Substrings" section of Figure 2.2 (this is the only part of the figure which is for reference only—all of the information here is stored somewhere else). Each entry in this row matches the index for the first characters of the "S-Type Substrings".

Then the "S-Dist based list" (see Figure 2.2) is created. An array array[3] from 1 to the largest S-Dist value is created. Each element of this "vertical" array contains another "horizontal" array[4] which is large enough to contain all the data (See Figure 2.2). The buckets in this list are there for pedagogical reasons—they do not need to be defined. Because the entire array has already been bucket sorted by the first letter at this point, these larger buckets will ensure that the items in the smaller buckets of the S-Dist structure are already in the right order. The S-Dist structure is allocated, and then S-Distances are read from the array we created from left to right (see the top S-Dist list which is dark/green in the top of Figure 2.2). Every character with an S-Distance that's greater than zero is stored in the appropriate row of the S-Dist array, in the next available column in the sub-array. It is important that the character t_i of each of these indexes i appears in the correct order in the "horizontal" S-Dist array. It is *not* important that the entire strings T_i be in order, this is accomplished later. The S-Dist array is created in its entirety.

Now, referring again to Figure 2.2, we wish to sort the S-Type Substrings in the flexible "Sorting S-Type Substrings" structure using the "S-Dist based lists" we just created. The top line (row 0) of the S-Type Substrings has been created, and

[3]Technically any data structure could be used for this step, but an array is the easiest structure to envision.

[4]This is an array of an array which is awkward, but a two-dimensional array will not work because this would take $\mathcal{O}(n^2)$ space. It's theoretically possible to run through all the indexes once to create the S-dist, again to count the occurrences of each number, and once more to actually process the data in $\mathcal{O}(3n) = \mathcal{O}(n)$ time, and this terrible strategy (see Caching 1.5.4) is what is described.

everything else *above* this line has also been made.

We start with the special subdividable buckets that are sorted according to the first letter—this is row 0. Unlike the rest of the diagram, the section "Sorting S-Type Substrings" is a progression of time, though the dark/green numbers for each row do correspond to S-Dist values. We wish to take row 0 and transform it into row 1. To do this, we will go through all of the entire S-Dist data structure from left to right, top to bottom. Here, we find a 2, and looking in the data we find t_2="b". However, we don't have an index 2 anywhere in our structure on the bottom because we are trying to sort S-Type substrings and 2 is an L-Type. Therefore, we need to find the S-Type index that 2 is associated with. This is done by remembering the dark/green S-Dist value. After all, we are bucketing the 2nd characters in all the S-Type substrings. The current S-Dist is 1, and the index we want is $2-1$, which is the first element of the "a" bucket under "Sorting S-Type Substrings" in Figure 2.2. It is already in the correct bucket according to the first character, so we need to move it to the front of its current bucket. There is also a pointer to the front of the "a" bucket which needs to be incremented to create a new "front" for the next step.

We have processed the index 2, and it's time to move to the next index in the "S-Dist based lists". This index is 6. Again, this is an L-Type index, and we use the current S-Dist number (1) to adjust the index to the previous S-Type index ($6-1=5$). Looking in the top row, we find the 5 and move it to the new so-called "front" of the "a" bucket. (The 1 in the a-bucket was sorted in the previous step, but the 3 is unsorted, which is why we incremented the "front".) We exchange the position of the 3 and the 5, moving the 5 to the correct position, and increase the "front" to make sure the item we just sorted is not moving again.

This next step is an important step. We reach the index 4 (the first element in the pedagogical "d" bucket in the S-Dist array). This converts to the S-Type index 3 which has just been swapped with the 5. Perhaps it would be efficient to make these buckets more than pedagogical after all, because t_5 ="d" which is different than any other element in the substantive "a" bucket at the bottom. This is why

the buckets must be subdividable. In the next row, labeled 1 in "Sorting S-Type Substrings" from Figure 2.2, you will notice that this bucket has been split into two: one "ab" half and another "ad" half.[5]

Creating new buckets is the key to this process. We want to create one bucket for every unique "S-Type Substring" (see the list of these substrings in Figure 2.2). Before, $\{1, 3, 5, 7\}$ were all grouped together because they started with an "a", but now $\{1, 5\}$ are in the "ab" bucket and $\{3, 7\}$ are in the "ad" bucket. Eventually, every unique substring will have its own bucket, and all the identical substrings will share a common bucket.

Things progress along rather smoothly, as one goes through the "S-Dist based lists" until after we finish the S-Dist= 2 values. At this point, there is an implicit bucket, because the 3 has been moved to the front of the "ad" bucket, but the 7 has remained at the end without being called. This creates a new implicit bucket represented by "ad___" to emphasize that it's just "ad" rather than its "ada" counterpart it shares its half-bucket with in Figure 2.2.[6] Creating this extra string is the only hiccup in the rest of his process.

The complexity of this section is $\mathcal{O}(n)$, presuming a relatively small alphabet of less than n, because the entire S-Dist lists every element precisely once–except for those indexes where the S-Distance is 0 which are omitted entirely. The array can be created with a linear scan, and each index is processed in $\mathcal{O}(1)$ time, so the current complexity is sound. In brief, this is only an extension of the bucket sort (also called radix sort).

2.1.3 The Recursive Step

Now that the S-Type substrings have been sorted, the recursive step is introduced. This is the fun step. In order for this to remain $\mathcal{O}(n)$ complexity instead of $\mathcal{O}(n \log n)$, less than $\frac{1}{2}$ of elements are passed on to each recursive step, as explained in Section 1.5.1.

[5]Reading this sentence quickly may prove as a good test for dyslexia.
[6]These half-bucket remains at the end of the bucket by design [21].

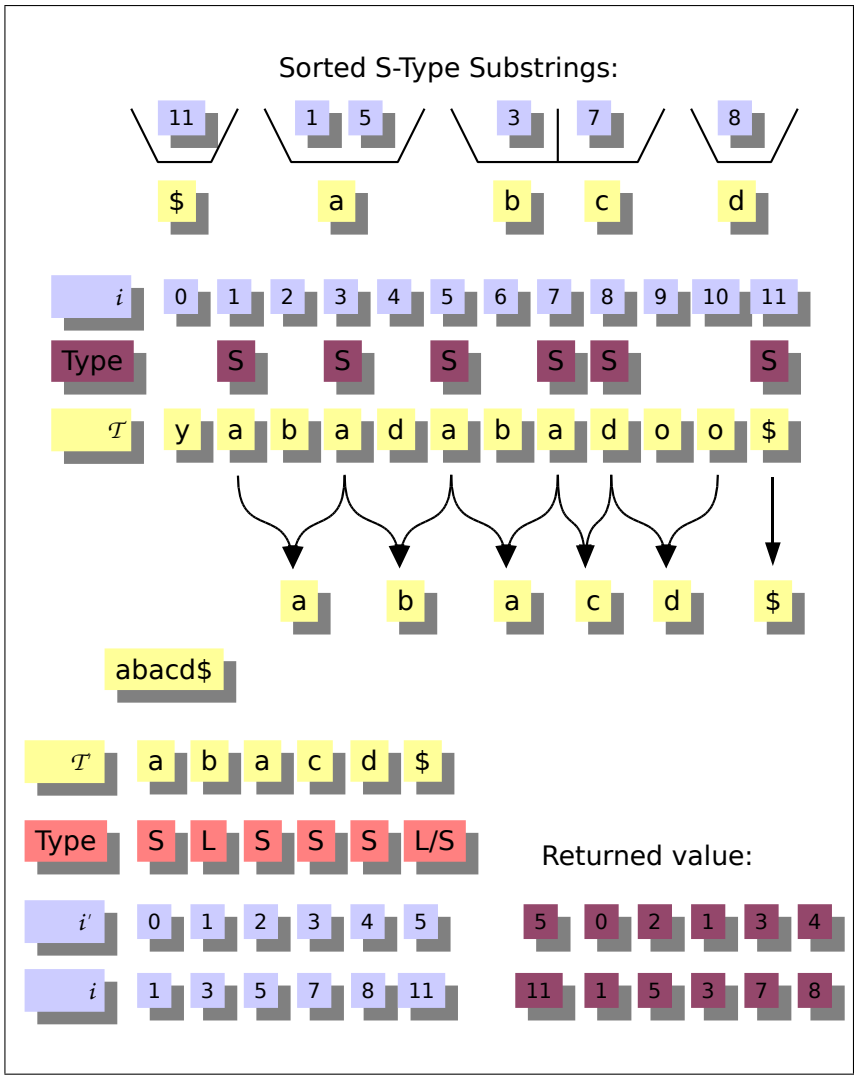


Figure 2.3: The recursive process.

To do this, a new alphabet must be created. In order for the bucket sort to continue to be effective, the size of the alphabet is restricted to a maximum of one letter for each S-Type substring $|\Sigma| < n$. This is shown in Figure 2.3. Normally, the alphabet would have a numeric representation, but the diagram continues to use letters to differentiate between the alphabet and the other numbers in the image.

To review, we now have one bucket for every unique S-Type substring in the array (See Figure 2.2), and these buckets are already in sorted order. The indexes and their buckets are shown at the top of Figure 2.3. Going from left to right, and from least to greatest, a new letter is assigned to every bucket so. Because every S-Type Substring in a given bucket is less than the substring in the next bucket, the new alphabet characters are assigned in lexicographic order.[7]

Next, a new string is created, and every S-Type substring is reduced into the new alphabet as shown in Figure 2.3. This could be done relatively inefficiently, but in $\mathcal{O}(n)$ time by going through the sorted S-Type Substring list and the corresponding alphabet to create a new string with the correct number of each character. For the diagram, this string would become "aabcd$". A forward and reverse array of links between each character and the appropriate suffix, and each suffix and the appropriate character would also be maintained. Then, by scanning the original data for all of the S-Type suffixes and moving the corresponding character to the front, one could *unsort* the new string. It would change from "aabcd$" to "abacd$". Some form of links between the characters and their corresponding indexes (such as the list of i and i' in Figure 2.3) must also be maintained. It would undoubtedly be possible to create a more efficient data structure for the process. The procedure is called recursively with the new string (T'), and the new S-Type and L-Type suffixes are found, and the entire process is repeated.

After the called function returns, the newly created array A' is itself sorted, and the final order of all the S-Type indexes can be taken from the contents of this array. The bottom right corner of Figure 2.3 shows the returned values, and how a new

[7]Ko and Aluru prove this formally in their paper [21], but they desperately needed the diagram shown in Figure 2.3 in this thesis to efficiently explain the concept.

array for i can be created from the i' that was returned with the recursive call. We now have all of the S-Type suffixes sorted into their final order in a single array.

2.1.4 Step 3: Sorting the L-Type Suffixes from the S-Type Data

Before S-Type substrings (which are shorter than the full suffix strings), and before the S-Type suffixes were sorted recursively, the entire suffix array A was organized into buckets. We had also evaluated whether each position was a L-Type or an S-Type index. Now, the sorted order of the S-Type indexes must be incorporated into the array.

There is an extremely important attribute of an S-Type suffix that has not yet been discussed. Any "Smaller" S-Type suffix will appear *after* a "Larger" L-Type suffix which starts with the same letter (ie. one that is in the same bucket.). This is counter-intuitive, because the a "smaller" S-Type was labeled S-Type because it was smaller than the suffix which immediately succeeds it in the original data, but it makes perfect sense after giving it some thought.[8] Take any simple suffix which does not begin with a repeated character. By definition, at the index i an S-Type suffix's first character $t_i < t_{i+1}$ whereas an L-Type suffix begins with $t_i > t_{i+1}$. This is the context in which the S-Type suffix is "smaller", and the L-Type "larger". Now let's presume that an S-Type suffix and an L-Type suffix both begin with the letter "d" (such as L-Type "da..." and S-Type "do..." in the diagrams). In this case, they cannot be sorted based only based on the first "d" in the string, but must be sorted by the rest of the string instead. Now an S-Type suffix is "smaller" than the suffix starting with it's second character, because this is where the second suffix came from. Let S be the S-Type suffix string, L be the L-Type suffix string, and s_i and l_i represent the individual characters in the i position. We already know that the first characters of both strings are the same, $s_0 = l_0$ because they're in the same bucket. And we know that, in the simple case, the second character $s_0 < s_1$ and

[8] Once again, Ko and Aluru give a formal proof of this in their paper [21].

$l_0 > l_1$. Now because $s_0 = l_0$, $l_1 < s_0 < s_1$ which means $L < S$. (The example confirms that the L-Type "da..." is indeed less than the S-Type "do...".) This also extends beyond the simple case to one where the first characters $s_0 = l_0 =$"d" and the remainder of the strings $L_{1...} <$"ddddd..."$< S_{1...}$.

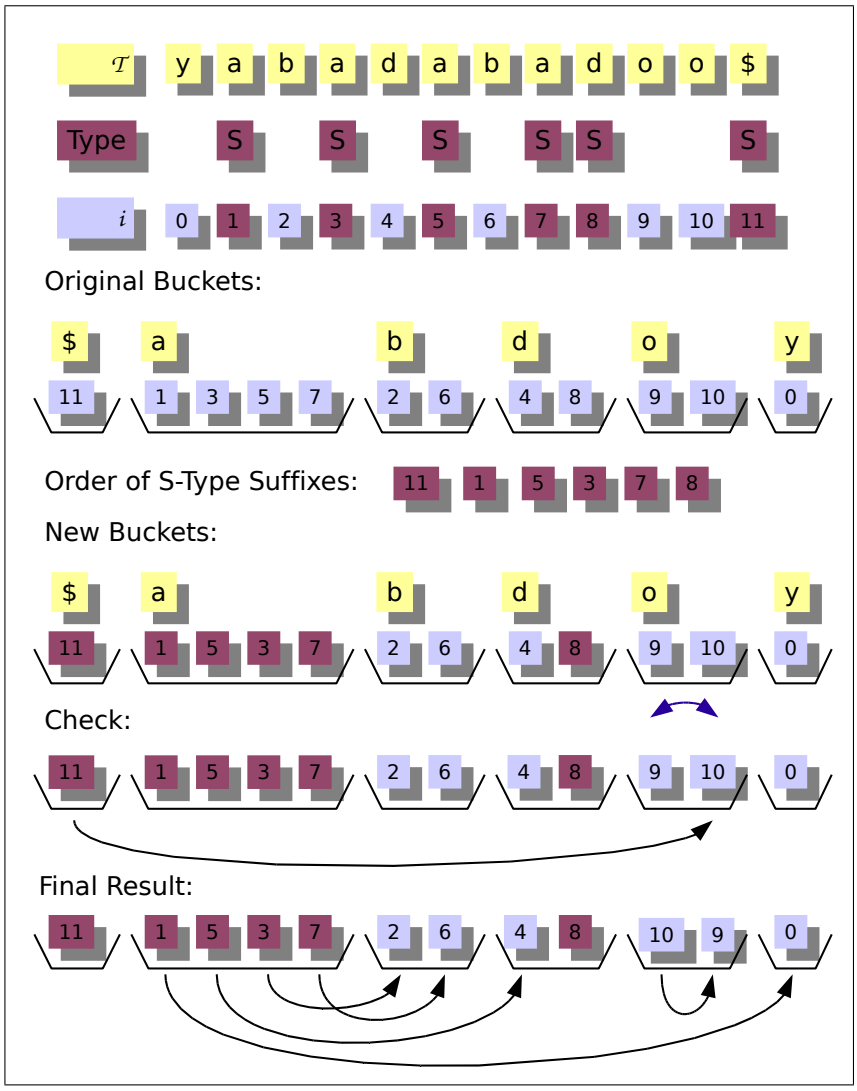


Figure 2.4: Using the correct order of the S-Type suffixes to determine the final suffix array.

Referring to Figure 2.4, we have the "Original Buckets" ready made, and the "Order of S-Types Suffixes" still needs to be integrated into the "New Buckets". Let's say that each bucket in the array A has both a "start" pointer and an "end"

pointer which are not truly the start and end of their respective buckets. Instead, this "start" and "end" correspond to the first and last *unsorted* location in its respective bucket.

Moving backwards from right to left through the "Order of S-Type Suffixes" which have been returned and are presumed correct, let's move each index encountered to the "end" of its bucket, and move the "end" of the bucket forward so that this index cannot be moved again. For example, referring to Figure 2.4, the index 8 is already at the "end" of its bucket so only the "end" would be changed, and likewise with the 7. However the 5 is out of place. By this time (due to the 7) the "end" of the "a" bucket would be one position to the left, and this position needs to become a 5. This is done by swapping the 5 (the next largest value) with the 3 in the "end" position (the next largest location). After this process is complete, every S-Type suffix will already be in the final location in the array. This is because it's already in the correct order returned from the recursive call, it's already in its correct bucket, and it's greater than every L-Type suffix in its bucket.

Now comes a fascinating process of moving each L-Type suffix to its proper location. This depends on the reverse suffix array R which allows us to link an index in the data T back to its position in the suffix array A. First, presume that everything in the array to the left of the index i is already correctly sorted. (In our case, the "$" character is smaller than anything else.) Now, find the position A_i in the data T, and go back one character. Use R to find the current position of that suffix in A. In Figure 2.4, during the "check", when $i = 0$, $A_i = 11$ (the last suffix of the data "$" is already in the correct position in the array). Now instead of looking at that suffix, look at the suffix one before this value ($T_{11-1} = T_{10} =$"o$"). Notice that the second character in this string is the character corresponding to our current bucket. Presuming the lexicographic value of our current bucket is smaller than the value of the bucket this index is in, this string must be smaller than any other string in its entire bucket. In our case, the string "o$" is smaller than the string "oo$" because the second character is "$"<"o". Therefore move it to the

front of its current bucket (remember that the bucket represents its first character, so it cannot move out of that). In this case, the 10 is moved to the front of the "o" bucket by swapping the 10 with the pointer to the "front", and then increasing the "front" because this index is now in its final location.

By moving forward from left to right through the entire suffix array A, the final sorted suffix array is obtained. In Figure 2.4, the "Check" represents the only change in the example, however the "Final Result" shows the arrows representing the rest of the checks which need to be made, even though they don't change the data. When each position in the suffix array is reached, the index pointed to by the end of the arrow is moved to the front of the bucket as long as both of these conditions apply: firstly, the index must be ahead of the current position in the suffix array (otherwise it will already have been sorted), and secondly, the index must be an L-Type suffix (otherwise it was already sorted by recursion). For example, there's no arrow going from the 4 in the "d" bucket both because it would be going backwards (the 3 is stored to the left of the 4) and because it would point to an S-Type suffix.

Now let's say we tried to perform this step without first sorting the S-Type suffixes. In our example, it would work by coincidence, but if the S-Type suffixes were scrambled it would still fail even if every index were sorted by bucket. This is because the second character of an S-Type suffix can be larger than its first character. At the time the position of an S-Type suffix is reached, the second character would not have been tested. In order for it to be sorted into the correct position, the scan would have to proceed backwards from right to left. With an L-Type suffix, even if the first few characters were the same, eventually there would be a tie-breaker between it and the next-closest L-Type Suffix, and this tie-breaker character would be from a bucket that's already been tested. This would move the Suffix ending the tie-breaker to the beginning of its bucket, and therefore it's previous character would be tested before any of the other previous characters—even if the "front" of the bucket were updated only just in time for the next test. (The formal proof for this looks a little more eloquent [21].)

Missing Link

The precise opposite set of procedures ought to be created in case the Type-S suffixes are more prevalent than the Type-L (such as is the case with the new string T' in Figure2.3). Because it's impossible for an index to be both a Type-L and a Type-S suffix (with the exception of the terminating charcter "$"), the number of the substrings which need to be sorted is less than $\frac{n}{2}$. By being able to handle either case, sort a suffix array in both $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ space.

2.2 Approximate String Matching

In music, and also in many other forms of Information Retrieval, an approximate string matching algorithm can be employed. This was recognized by Dias and Gil who used suffix arrays to compute "Positional N-grams" [14]. A Positional N-gram allows a Statistical Machine Translation system to associate out-of order words. The paper focuses not on searching for the words specifically, but on building a database of the probability that words could be associated together. It uses a suffix array to count the number of times a pivot word appears in a text, and then creates its statistics based on this.

There are two papers which address approximate string matching without using an inverted index. The one created by Yamamoto et. al. uses a suffix array to generate many small queries extremely quickly, and then calculates a separate similarity index [37]. The other is a technique created by Ukkonen in 1993 that augments a suffix *tree* with enough information to allow an efficient approximate search using dynamic programing. Although he suggests that perhaps his process could also be implemented using a suffix array, the technique is restricted because it depends on a small alphabet size.

The goal of most approximate string matches is to find two strings with a small edit distance k between them. This idea was introduced in Russia by Levenshtein

in 1966 [23].[9] He defines the edit distance between String A and String B as the minimum number of insertions, deletions, or replacement it would take to transform String A into B. The technique presented in this chapter generally follows this convention.

As with many algorithms, I have a specific goal in mind, and it serves as a wonderful example. I would like to find the edit distances between different *sentences* rather than individual words to try to improve SMT so that it can generate more effective results with a smaller corpus. As an example of a small corpus, consider the Greek New Testament used in Chapter 3. It contains about 5500 unique words, and contains 138000 words total. Greek is an intensely morphological language, so many of these words have many different forms in the text. But the text has been hand-sorted, and each word has been stemmed to an appropriate value.[10] It is possible to build a suffix array based on these stemmed values, which is like an array of inverted-index values, or like a computer representation of a Chinese text with one character (represented by a number) for every word. After creating this version of the text to allow searching these words and phrases, the searches could be used to create cross-references between passages or, in the case of a new translation, could be used to reference previously translated sections. (Grammatical attributes could be stored separately, perhaps in a separate suffix array, but this is not used in the examples.)

However this is a simple conversion process to facilitate searching for stemmed words, but it does not account for an edit distance. That is what is defined below in Figure 2.5. This is a structure which can handle both insertions and deletions, but not replacement. Because the text is small, it may be stored many times (the New Testament can fit comfortably into 32MB more than a dozen times, even with a large amount of data to allow quickly searching a suffix array). The insertion and deletion operations may both be performed by performing deletions in the Search

[9]This is the other document found in all the literature which I have not been able to obtain a copy of.

[10]Perhaps the most famous is *Strong's Exhaustive Concordance of the Bible* published in 1890.

String and in the text.

Pattern	Data stored in the suffix array											Search String			
XXXX	we	can	see	there	***is***	***the***	text	which	is	stored	repeatedly	this	is	the	search
XXX_	we	can	see		***is***	***the***	text		is	stored	repeatedly	this	is	the	
XX_X	we	can		there	***is***	***the***		which	is	stored		this	is		search
X_XX	we		see	there	is		text	which	is		repeatedly	this		the	search
_XXX		can	see	there		the	text	which		stored	repeatedly		is	the	search
XX__	we	can			***is***	***the***			is	stored		this	is		
X_X_	we		see		is		text		is		repeatedly	this		the	
X__X	we			there	is			which	is			this			search
_XX_		can	see			the	text			stored	repeatedly		***is***	***the***	
_X_X		can		there		the		which		stored			is		search
__XX			see	there			text	which			repeatedly			the	search

Figure 2.5: Searching a suffix array allowing for two insertions and two deletions

In the example in Figure 2.5, we have a maximum search length of four characters, and 11 copies of the text are necessary. (For a more comfortable edit distance of 9, 46 copies are needed.) There is a pattern, shown in the left column, that is used to delete words in both the search string and the original text. This pattern can be generated by counting in binary, and then removing all of the entries that require more than two deletions. The search string is edited in the same way.

Now, using up to 2 deletions and up to 2 insertions, it is possible to find "this is the search" in the text. Only the phrase "is the" actually produces results, and these four results are highlighted in the diagram. The best part about this algorithm is that the first word of the phrase does not have to line-up with the original text. This allows the first position of the Search String to match the fourth position of the text. Thus every instance of the String within this edit distance is found using only 11 searches—presuming the length of the string is equal or less than 4 "characters" ($|P| < 4$). It would take $\mathcal{O}(4)$ time to index in a single suffix array. To get this efficiency, it is necessary to store all of the data in just one suffix array rather than splitting the suffix array into sections. The precise nature of the match can be detected after it's found based on the index or supplimentary data.

The primary problem with this basic search, is that it only accounts for insertions and deletions. Thus many results are returned, when one is significantly better than the rest because the edit distance is better. This can be solved by creating a special

Pattern	Data stored in the suffix array											Search String			
XXXX	we	can	see	there	is	the	text	which	is	stored	repeatedly	this	is	the	search
XXX_	we	can	see	—	is	the	text	—	is	stored	repeatedly	this	is	the	—
XX_X	we	can	—	there	is	the	—	which	is	stored	—	this	is	—	search
X_XX	we	—	see	there	is	—	text	which	is	—	repeatedly	this	—	the	search
_XXX	—	can	see	there	—	the	text	which	—	stored	repeatedly	—	is	the	search
XX__	we	can	—	—	***is***	***the***	—	—	is	stored	—	this	is	—	—
X_X_	we	—	see	—	is	—	text	—	is	—	repeatedly	this	—	the	—
X__X	we	—		there	is	—		which	is	—	—	this	—		search
_XX_	—	can	see	—		the	text	—	—	stored	repeatedly	—	***is***	***the***	—
_X_X	—	can	—	there	—	the	—	which	—	stored	—	—	is	—	search
__XX	—	—	see	there	—	—	text	which	—	—	repeatedly	—	—	the	search

Figure 2.6: Searching a suffix array allowing for an edit distance with up to two replacements

word, "—" in Figure 2.6, that is unique in the entire datastructure. This word is used in order to represent any word that is being replaced, thus extending the algorithm from one that only allows insertions and deletions to one that can detect replacements as well. Notice that the amount of time needed to search in this structure is virtually identical to the previous structure, but the amount of space needed has actually increased because of the need to store the extra "—" words. However the result is significantly better. An extension to allow words to be swapped could be made as well. It would probably be most efficient to do this in the queries, except, perhaps, for the special case where two words are swapped so that these could be searched at the same time as the other strings. (Using the New Testament as an example, "Jesus Christ" and "Christ Jesus" are both extremely frequent terms which refer to the same person.)

Finally, there is a special case of searching for two words which are both close to each other. This is shown in Figure 2.7. The search string could be reversed to allow either word in the search pattern to appear first. The thing that makes it a special case, is that the memory required for the suffix array varies directly with the maximum distance between the words required in the search. As you can see in the diagram, all of the blank spaces do not effect the search, and do not take up space, thus each set of rows will compress into the size of one row. It's also important to note the false-positive for the phrase "repeatedly there" in Figure 2.7 which wraps around from the end of one document to the beginning of the next iteration. This

Pattern	Data stored in the suffix array											Two-Word Search Strings	
X	we	can	see	there	is	the	text	***which***	***is***	stored	repeatedly	text see	***see text***
X_	we		see		is		text		is		repeatedly	***is which***	***which is***
_X		can		there		the		which		stored		there repeatedly	*repeatedly there*
X__	we			there			text			stored		***there is***	is there
_X_		can			***is***			***which***			repeatedly		
__X			see			the			is				
X___	we				is				is				
_X__		can				the				stored			
__X_			***see***				***text***				*repeatedly*		
___X				*there*				which					
X____	we					the					repeatedly		
_X___		can					text						
__X__			see					which					
___X_				***there***					***is***				
____X					is					stored			

Figure 2.7: Searching for a pair of words which occur within 5 words from each other.

problem can be solved by using a special end-of-document character in addition to the blank-word character.

Chapter 3

Suffix Arrays and Koiné Greek

3.1 Background

By the time of his death in 323 BC, Alexander the Great had conquered most of the known world. His conquests ushered in the Hellenistic Period and this was the primary reason Greek was the language of trade four hundred years later, much the way English is today. The form of Greek spoken in the 1st Century A.D. is known as Koiné Greek, from the Greek word "κοινή" which means common. One rather notable document written in Koiné Greek is the New Testament. Many people, myself included, learned this ancient language expressly to understand this particular text better. It would be surprising to find any text (either sacred or secular) which has been analyzed more than the New Testament from the Bible. Because of this, it serves as an excellent testing ground for new ways of analyzing text. In this thesis I have been able to start with pre-analyzed text, and show how the suffix array can be used to analyze it further. The original Koiné Greek text is extremely well analyzed—several parsed versions are available in electronic form.

One fascinating thing about analyzing Koiné Greek is that there is such a small corpus of text. Documents constantly need to be translated between French and English due to the political system in Canada and the European union. Legislation, other legal documents and Parliamentary proceedings need to be translated whenever they are written. This is not the case with Koiné Greek because the language is dead; it's sufficiently different from modern Greek that a different system would need to be employed for recent Greek documents. People who translate from Koiné Greek usually focus on translating into new languages with a relatively small number of native speakers and on understanding precisely what the author meant by what he said. This leads to a more painstaking approach to translation than one usually sees in other areas.

Another fascinating thing about the Koiné Greek New Testament is that because its corpus is so small, the text has been repeatedly analyzed before. In 1890, after years of research with the help of more than a hundred colleagues, Dr. James Strong published his concordance of the Bible. This amazing work assigned a unique num-

ber to every Koiné Greek and Hebrew word. Today this type of research would be impressive, but the work seems even more astounding because it was occomplished without the assistance of a computer! Because Greek is a highly morphological language, automatically parsing a word—especially a verb—is not a simple task. To combat this problem, I have used a numbering system as a starting point for my analysis of the text.

3.1.1 Description of the Format

For the purpose of this project, I used a slightly different version of the New Testament which was formatted for computer processing by James Tauber and Ulrik Peterson [33]. (Although the text itself is currently available from the original website, the information about the text can only be accessed through an Internet archive such as the Wayback Machine [34].) The Greek text was stored with all its accents using the UTF-8 form of Unicode, and the formatting was very easy to work with. Each word had its own line, and line has a separate column for five items: the chapter and verse, part of speech, morphology, word as it appears in the text, and word as it appears in a dictionary. This is somewhat similar to the way I reduce music to a series of intervals in Section 4.1.2. For the purpose of this thesis, I found every unique word in its dictionary form based on the cast column from Tauber and Peterson's text. After this, I converted each unique word into a number. Finally, I went back through the original text and converted the last column in each line from the dictionary form of the word into a number. It is this number that I used to create the suffix array.

For example, the line: "040316 V- 3AAI-S– ἠγάπησεν ἀγαπάω" represents the first verb from John 3:16.[1] The number (040316) can be split into three columns (04 03 16). The "04" represents the book, John, which is the 4th book in the New

[1] The full text of John 3:16 is "οὕτως γὰρ ἠγάπησεν ὁ θεὸς τὸν κόσμον ὥστε τὸν υἱὸν τὸν μονογενῆ ἔδωκεν ἵνα πᾶς ὁ πιστεύων εἰς αὐτὸν μὴ ἀπόληται ἀλλ᾽ ἔχῃ ζωὴν αἰώνιον" which means "For God loved the world this way: he gave his one-of-a-kind son so that all those believing in him should not die but have eternal life." This verse is extremely popular in North America and its reference can often be seen on signs in the crowd at sporting events.

Testament. The 03 and 16 represent the chapter and verse respectively. The part of speech is a verb (V-) and it's parsed as a 3rd person singular, aorist (past tense) active indicative (3AAI-S–). The verb, "ἠγάπησεν" (he loved), is then represented in its dictionary form, "ἀγαπάω".[2]

For the purpose of this thesis, almost all the information for each word is set aside and ignored. The only piece of information I use for searching and sorting the suffix array is the dictionary form of the word, except that I converted each unique word into a number. For the example above, I changed the line to "040316 V-3AAI-S– ἠγάπησεν 3514" before loading it into the main program. The number 3514 is similar to a Strong's number because it represents the Greek word "ἀγαπάω" however it was easier for me to write a script to generate my own numbering scheme than to borrow Strong's. The precise numbering scheme doesn't matter as long as there is a unique number for each word. The additional information for each word is stored in the data structure as it is loaded so that the text can be printed in the original form and the verse references can be used for creating the cross-reference (see Section 3.3).

3.1.2 Textual Limitations

One of the more fascinating things about studying a 2000 year old document is the textual differences of various manuscripts. The New Testament is compiled from hundreds—even thousands—of different manuscripts many of which contain only small fragments of the whole text. The oldest manuscript is the Rylands Library Papyrus (P52) which is a small fragment from the Gospel of John dated circa 125 A.D. As a comparison, Homer's *Odyssey* and *Iliad* are compiled from far fewer fragments which were transcribed a greater number of years from when the original autograph was written.

Although I realize that a translator should take note of these subtle textual

[2] "ἀγαπάω" is the traditional lexical form of the Greek word. It similar to the 1st person present active indicative form (i.e. ἀγαπω, I love), but the slightly different "άω" ending is used to indicate how the verb is conjugated. This is why the word was not translated in the text.

differences, I have totally ignored them in my research. This is mostly because I am trying to prove computational concepts rather than create a full translator's aid. It is also because most of these differences are very subtle and it's usually very easy to see which text is accurate by looking for patterns in the way various supporting manuscripts change over time. However if this project were expanded into a full translator's aid, it would be relatively important to allow the translator to see the text in its various forms.

3.2 Finding Phrases in the Greek New Testament

The simplest application of my research is finding common phrases in the New Testament. This is almost like looking at the raw data created through the suffix array construction (Section 2.1) and finding all of the Longest Common Prefixes (Section 1.7). Getting to this point, however, was the most difficult part of the thesis.

A biblical scholar who reads this chapter will probably find it interesting, but they will also realize that I haven't found any phrases which haven't been studied before. The methods that I use here reveal new insights into the New Testament to the same extent that a flashlight helps a person see on a sunny day. If I've written any insights, they are insights which have been seen before. I am not trying to prove that the suffix array achieves better results than a trained scholar or other computerized tools. Instead, I want to show that the suffix array is capable of quickly and automatically revealing the same things that people have known about for centuries. In other words, if a suffix array contains known results with the New Testament, it should be useful to reveal new results with other documents. Although it may be able to provide search results for the New Testament more quickly than other tools, the difference in time would typically only be a matter of milliseconds—though even a number of milliseconds may matter in a computer

assisted translation environment. I also try to speculate on how finding repetition within a small document could help create a computer assisted translation system described more fully in Section 3.4.

3.2.1 Methodology

As mentioned in Chapter 1, a suffix array is a table of all of the suffixes in a text sorted alphabetically. Each suffix starts from some word in the New Testament and goes to the end of the document. Sorting the table will place every identical word or phrase right next to all of its other occurrences in the text. The concept is best illustrated by example, and the simplest example was already given in Section 1.2. The result is basically the entire New Testament sorted by the rest of the New Testament.

Line	LCP	Verse	Koiné Greek Text
1	4	Mat 19:28	ὁ υἱὸς τοῦ ἀνθρώπου ἐπὶ θρόνου δόξης αὐτοῦ καθήσεσθε καὶ ὑμεῖς ἐπὶ δώδεκα θρόνους
2	5	Mat 9:6	ὁ υἱὸς τοῦ ἀνθρώπου ἐπὶ τῆς γῆς ἀφιέναι ἁμαρτίας τότε λέγει τῷ παραλυτικῷ ἐγερθεὶς
3	4	Mat 11:19	ὁ υἱὸς τοῦ ἀνθρώπου ἐσθίων καὶ πίνων καὶ λέγουσιν ἰδοὺ ἄνθρωπος φάγος καὶ οἰνοπότης
4	14	Luke 7:34	ὁ υἱὸς τοῦ ἀνθρώπου ἐσθίων καὶ πίνων καὶ λέγετε ἰδοὺ ἄνθρωπος φάγος καὶ οἰνοπότης
5	5	Luke 17:26	τοῦ υἱοῦ τοῦ ἀνθρώπου ἦσθιον ἔπινον ἐγάμουν ἐγαμίζοντο ἄχρι ἧς ἡμέρας εἰσῆλθεν Νῶε
6	4	Luke 12:40	ὁ υἱὸς τοῦ ἀνθρώπου ἔρχεται εἶπεν δὲ ὁ Πέτρος κύριε πρὸς ἡμᾶς τὴν παραβολὴν ταύτην
7	5	Mat 24:44	ὁ υἱὸς τοῦ ἀνθρώπου ἔρχεται τίς ἄρα ἐστὶν ὁ πιστὸς δοῦλος καὶ φρόνιμος ὃν κατέστησεν
8	5	Luke 18:8	ὁ υἱὸς τοῦ ἀνθρώπου ἐλθὼν ἆρα εὑρήσει τὴν πίστιν ἐπὶ τῆς γῆς εἶπεν δὲ καὶ πρός τινας
9	5	Luke 21:27	τὸν υἱὸν τοῦ ἀνθρώπου ἐρχόμενον ἐν νεφέλῃ μετὰ δυνάμεως καὶ δόξης πολλῆς ἀρχομένων
10	9	Mark 13:26	τὸν υἱὸν τοῦ ἀνθρώπου ἐρχόμενον ἐν νεφέλαις μετὰ δυνάμεως πολλῆς καὶ δόξης καὶ τότε

Figure 3.1: An excerpt from the suffix array generated using the New Testament

The example in Figure 3.1 is intended to show how a suffix array is useful for analyzing this document. Each line in this excerpt starts with the phrase "ὁ υἱὸς τοῦ ἀνθρώπου" which means the son of man. This is the most common way Jesus used to refer to himself in all four gospels. With some knowledge of Greek, it's easy to see that the morphology has been ignored as explained in Section 3.1.1 above. For example, line 5 start with "τοῦ υἱοῦ τοῦ ἀνθρώπου" (of the son of man) which is included with the other references despite being in the genitive form. This particular

reference actually starts in the middle of a sentence and the middle of a concept—the verse actually refers to "ταῖς ἡμέραις τοῦ υἱοῦ τοῦ ἀνθρώπου" (the days of the son of man). Although this phrase is only two lines earlier in the input document, it occurs over four thousand lines later in the output of the suffix array because of the way all the words are sorted alphabetically.

3.2.2 About the Longest Common Prefix (LCP)

Another fascinating part of this excerpt is the LCP (Longest Common Prefix) column. As mentioned in Section 1.7, the longest common prefix is the greatest number of words one line has in common with the line immediately previous to it. All of the words in Figure 3.1 share the first 4 words in common, and this is why the LCP is always at least 4. The longest LCP is between lines 3 and 4 with 14 words in common—long enough to exceed the size of the table![3] There are a couple interesting things about these two lines. One is that Matthew wrote this accusation in the third person plural "λέγουσιν" (they say) whereas Luke used the second person "λέγετε" (you say). Because the root word is the same, the algorithm ignores this subtle difference. The next interesting thing is that this phrase spans from verse 34 to verse 35 in Luke, but the same 14 words from both verses are contained in the same verse in Matthew. The suffix array ignores verse and sentence boundaries. This is a good thing partially because these boundaries were not marked in the original manuscripts; the verses were added later.[4] Sometimes sentences were marked but sometimes even the spaces between words were omitted. The verse boundaries are used, however, with the way I process cross-references (see Section 3.3).

The LCP index for each verse is an extremely useful tool, but finding the LCP between two different lines is not always immediately obvious. For example, how

[3]The full text from Mat 11:19 is "ὁ υἱὸς τοῦ ἀνθρώπου ἐσθίων καὶ πίνων καὶ λέγουσιν ἰδοὺ ἄνθρωπος φάγος καὶ οἰνοπότης τελωνῶν φίλος καὶ φίλος καὶ ἁμαρτωλῶν καὶ ἐδικαιώθη ἡ σοφία ἀπὸ τῶν ἔργων αὐτῆς" which means "The son of man [came] eating and drinking and they say, 'Behold a gluttonous and drunk man, a friend of tax collectors and of sinners.' And wisdom is justified by her works."

[4]This did cause some trouble described on page 83.

can one calculate the LCP of lines 5 and 7? The LCP column of both lines indicates a 5, but they only have four words in common. The verb "ἐσθίω" (I eat) in line 5 becomes the verb "ἔρχομαι" (I come) in line 7. This means that the pre-calculated LCP values can only be used with adjacent lines directly, though other lines can be calculated using the intermediate lines as a guide.

For example, let's find the Longest Common Prefix (LCP) between line 5 and every other line with at least one word in common, starting in the upward direction. The LCP at line 5 is 5 indicating that lines 4 and 5 have 5 words in common. Note that "ἐσθίων" (eating) and "ἤσθιον" (they ate) are both forms of the same Greek word "ἐσθίω" (I eat). The LCP index at line 4 is 14, however because the current LCP with the intermediate line 5 is only 5, the LCP between line 3 and line 5 is still only 5. At line 3 the LCP drops down to 4 indicating that lines 2 and 5 have an LCP of 4. As the process continues, with the current LCP sometimes decreasing but never increasing, an LCP of zero is reached somewhere above the top of the table. When an LCP of zero eventually comes, this indicates a phrase which doesn't start with "ὁ" (the).

After processing all of the lines which match line 5 in the upward direction, it's also important to find all common lines in the downward direction too. Note how the LCP between lines 5 and 6 is stored beside line 6 instead of line 5. This is the difference between pre-incrementing the line and post-incrementing it whenever a new LCP value is retrieved from the table. Apart from this subtle difference, the LCP is calculated in the same way. Lines 5 and 6 have an LCP of 4. Lines 5 and 7 also have an LCP of 4 because the extra word in common is shared between lines 6 and 7 but not between lines 5 and 6. As the process continues, it would find that line 5 shares these same four words with every other line in this excerpt.

3.2.3 Program Performance

As mentioned in Section 1.5.4, one of the greatest problems associated with suffix arrays is the problem of cache misses. This is because a large amount of data is

stored in a seemingly random fashion, and it's difficult to predict where the next memory reference will be. The fastest computing hardware for suffix arrays has low memory latency. At the time of this writing, AMD is usually superior to Intel in this regard because of the way AMD processors have integrated memory controllers. Also, multi-core processors will not speed up a program that requires low memory latency even if the code is rewritten to take advantage of the greater processing power.

Although I didn't fully optimize my program to be cache conscious, I did ensure that most of the data was stored in arrays of structures where each element contained several fields rather than separate arrays with related data at the same index of both arrays. This ensured that when I was dealing with one piece of information associated with some data, it would be more likely that all of the other information associated with that data would also find its way into the cache at the same time. The program could be improved by tweaking the structures to occur on common cache boundaries such as (every 16, 32, or 64 bytes) and using a memory allocation routine which also respects cache boundaries. The program could also easily be modified to use less memory which would be particularly useful for larger documents.

I was pleasantly surprised to find that my implementation of the suffix array construction algorithm described in Section 2.1 ran quickly. It took just 3.5 seconds to process all 138000 words in the new testament on a modest 1.2 GHz Pentium 4M. This includes the time taken to load and save various files. I also verified that this program took about four times as long when I gave it five times as much data indicating that the construction algorithm does behave in a linear fashion (at least until it runs out of RAM).

3.2.4 Common Phrases in the Greek New Testament

The simplest way to create something useful from the raw data generated from the suffix array and Longest Common Prefix (LCP) calculations, is to find the longest and most common phrases. Although it is somewhat useful in its own right, this is

particularly important in understanding the type of information that a suffix array may be able to uncover. If one is to use a Suffix Array for much of anything, it would be good to know what sort of information such a data structure may contain. A small excerpt from the middle of this output is shown in Figure 3.1; the full output contains about 138000 lines of text with one line for every word. I processed all of the LCP values as explained in Section 3.2.2 and then sorted them according to the most common phrases with the same number of words. This section examines the results.

The most common phrase (if it can be called a phrase) is the one-word phrase "ὁ" which is the Greek word for "the". This word occurs almost 20000 times in the New Testament which represents 14% of the entire document! In fact, this word is so ubiquitous that more useful results may be obtained by eliminating it altogether. If the article were removed, perhaps the suffix array could find better string matches. One example which shows how trivial the article can be to the meaning of a phrase, and how this makes it more difficult to search for a phrase, is given on page 70. The word could either be re-inserted into the text or tied directly to the previous word. One of the interesting things about Greek is it has only one article. English uses both the indefinite article "a" and the definite article "the", likewise French uses both "*la*" and "*une*", but in Koiné Greek there is no such distinction. Another interesting (and common) Greek construction uses the article followed by a participle. An example can be found in John 3:16 which is quoted in Section 3.1.1 above. In this text "ὁ πιστεύων" is generally rendered "the [one] believing" and the article is used to allow the verb to be used as a noun.[5] Similarly, "ὁ μὴ πιστεύων" (the [one] not believing) is also a very common construction—in fact these two phrases are contrasted in 1 John 5:10. The two words "ὁ μὴ" (the not) occurs 83 times in the New Testament and this is almost always followed by a participle. This is the type of information which would be sacrificed if the article were dropped when the suffix array is sorted, but perhaps this information

[5]I translated the text "πᾶς ὁ πιστεύων" as "all those believing" because I wanted to keep the participle in English but "all the [ones] believing" is a little more awkward.

is already sacrificed because the sorted information only makes use of the lexical form. The text above is basically converted to "ὁ πιστεύων" (the [one] believing) into "ὁ πιστεύω" (the I believe). The precision of the phrase is sacrificed in order to create more matches.

The most common two word phrase is almost as useless. It's "καί ὁ" (and the). It has 1582 occurrences and perhaps the most important thing it shows is the importance of manually discerning which phrases are important, as we will in Section 3.2.5.

The most common three word phrase in the New Testament is somewhat interesting because it's the way the authors of the gospels typically introduce Jesus' response to various questions. The phrase is "καί εἶπεν αὐτῷ" which means "and he said to him". This phrase occurs 167 times mostly in the gospels with the occasional reference in Acts and Revelation. Once again, the results are changed because of the way the morphology of the words is ignored. The phrase "καὶ λέγει αὐτοῖς" (and he says to them) counts towards the 167 references despite being in the present tense and having a plural object. Likewise, the phrase "καὶ λέγουσιν αὐτῇ" (and they say to her) is included despite having a plural form of the verb "λέγω" (I say) and despite having a feminine object. It's interesting to note that the object is always in the dative form (i.e. "αὐτῷ" which means "to him") rather than the accusative or nominative form (i.e. "αὐτός" which means "he"). This has nothing to do with the search algorithm—to which both phrases appear to be identical—and everything to do with Koiné Greek grammar. The verb "λέγω" always takes its object in the dative case in the same way that the verb "to say" in English requires the auxiliary particle "to" in order for it to have an object who hears what's said. In English, we say something *to* somebody because if we say somebody it sounds as if somebody is something which is spoken rather than someone who listens. The same thing is accomplished in Koiné Greek by putting the object into the dative case.

The most common four word phrase is probably the first phrase with literary significance. It is "ὁ υἱὸς τοῦ ἀνθρώπου" (the son of man) which is the most common way Jesus referred to himself. Jesus was not the first person to use the designation,

it occurs a few times in Psalms and extremely frequently in the writings of the prophet Ezekiel. In the New testament, the phrase is used 77 times, and all but two of these occurrences is from one of the Gospels which is no surprise because these are the books where one would expect Jesus to refer to himself. The exceptions are also interesting. The first is in Acts 7:56 while Stephen was being stoned to death: "καὶ εἶπεν ἰδοὺ θεωρῶ τοὺς οὐρανοὺς διηνοιγμένους καὶ τὸν υἱὸν τοῦ ἀνθρώπου ἐκ δεξιῶν ἑστῶτα τοῦ θεοῦ" (and he said, "Look! I see heaven opened and the son of man standing at the right hand of God"). The other exception is in Eph 3:5 where Paul refers to mysteries that "τοῖς υἱοῖς τῶν ἀνθρώπων" (the sons of men) from other generations did not know. This is the only time in the New Testament that this phrase doesn't refer to Jesus, and also the only time it's used in the plural.

The most common five word phrase occurs only in the Epistles, especially in the writings of Paul. The previous phrase is the way Jesus referred to him self, and this is one way in which other people referred to him: "ὁ κύριος ἡμῶν Ἰησοῦς Χριστὸς" (our lord Jesus Christ). The phrase occurs 36 times in the New Testament.

There are a couple six word phrases which occur eight times in the New Testament. One is the phrase, "εἰς τοὺς αἰῶνας τῶν αἰώνων ἀμήν" ("truly forever and ever," literally "into the ages of ages amen"). The Greek word "αἰών" is the etymology of the English word, "eon". It was written primarily in non-Pauline epistles. The shorter phrase, "εἰς τὸν αἰῶνα" (literally "into the age") occurs 35 times including the eight references mentioned above, and it was used throughout the Gospels and Epistles. This is the first term which is a Koiné Greek idiom—this is the way they said forever. The longer term simply emphasized the fact that it was really forever and ever rather than specifying a longer period of time in the same way that "forever and ever" is not a longer period of time than "forever". This is also the first time we've stumbled across an idiom, and it shows that the Suffix Array is useful for bringing such phrases together even though some human intervention is required to bring them to the forefront.

The most common seven, eight, nine, ten, eleven, and twelve word phrases all

occur in the greetings at the beginning of Paul's Epistles. The suffix array does not distinguish between a "full" match and a "partial" match which means that any three word phrase also contains two two word phrases: one with the first and second word and one with the second and third. This is why this one phrase occurs so many times. The greeting is this: "χάρις ὑμῖν καὶ εἰρήνη ἀπὸ θεοῦ πατρὸς ἡμῶν καὶ κυρίου Ἰησοῦ Χριστοῦ" (grace to you and peace from God our father and lord Jesus Christ). This same greeting occurs eight times in the New Testament. In his letter to the Colossians, Paul interjected the word "εὐ῾ριστιῦμεν" (we give thanks) which is enough of a change to ensure that the matches with nine or more words in common only occur seven times in the Suffix Array. It's easy for a person to see the similarity by browsing through the Suffix Array, but it's more difficult for a computer to calculate the difference. One possible solution was proposed in Section 2.2 and another will be explored in Section 3.3. Also, the Greetings in 2 Corinthians and in Ephesians actually have 24 words in common because they continue in the same way after this phrase.

The longest phrase which is duplicated word-for-word is a 41 word phrase that occurs in both Acts 28:26 and Mat 13:14. Both Jesus and the apostle Paul apply Isaiah 6:9-10 to the people who are unable or unwilling to receive their message. They say "ἀκοῇ ἀκούσετε καὶ οὐ μὴ συνῆτε καὶ βλέποντες βλέψετε καὶ οὐ μὴ ἴδητε ἐπαχύνθη γὰρ ἡ καρδία τοῦ λαοῦ τούτου καὶ τοῖς ὠσὶν βαρέως ἤκουσαν καὶ τοὺς ὀφθαλμοὺς αὐτῶν ἐκάμμυσαν μήποτε ἴδωσιν τοῖς ὀφθαλμοῖς καὶ τοῖς ὠσὶν ἀκούσωσιν καὶ τῇ καρδίᾳ συνῶσιν καὶ ἐπιστρέψωσιν καὶ ἰάσομαι αὐτού" (By hearing you will hear, but you will not understand; by seeing you will see, but you will not recognize. For the heart of this people has become dull, and their ears barely hear and they have closed their eyes. Otherwise they would have seen with [their] eyes and heard with [their] ears and understood with [their] hearts and repented and I would heal them.) It is fascinating to notice that this, the longest repeated phrase, consists of only 41 words. One would expect that a longer Old Testament quotation would have been chosen by multiple New Testament authors. This is because the New

Testament authors usually quote fairly short passages and because there are often subtle differences between the passage which is quoted and the Septuagint (i.e. the Ancient Greek translation of the Hebrew Old Testament). Perhaps this is because New Testament authors often quote from memory or because they remember the original Hebrew form.

3.2.5 Interesting Phrases in the Greek New Testament

As we have seen in Section 3.2.4, looking at the most used phrases in a document is an interesting exercise, but it often does not uncover extremely useful information especially with short phrases. It would be better to find all the common idioms or find passages which are related to each other (as we will in Section 3.3). Nonetheless, it is important to look at this type of information because it serves as a building block. In this section, instead of looking only at the very most common phrases which are a given length, I look at some of the most interesting phrases chosen from the top contenders. This will help show what type of information a suffix array will bring to the forefront.

Interesting Two Word Phrases

One interesting two word phrase which is most common (among interesting phrases) is "λέγει αὐτῷ" (he says to him). Often this occurs in a different tense or number, such as "εἶπεν αὐτοῖς" (he said to them), but these both have the same root word and therefore they appear in the suffix array together. The construction occurs 483 times. The phrase occurs almost entirely in the gospels and Acts because these are the narrative books where one would expect people to be saying things to people. What makes this example particularly interesting is the concept of how much time could be saved if a human translator only had to translate the phrase a few of the 483 times it occurs. This is very similar to the phrase "λέγω ὑμῖν" (I say to you) which occurs 185 times. One interesting thing about reading through the suffix array at this point is how often the same words follow this phrase. There are 53

examples of the phrase "λέγω ὑμῖν ὅτι" (I say to you that), 14 examples of "λέγω ὑμῖν οὐκ" (I say to you not), and 10 examples of "λέγω σοι ἐὰν" (I say to you if). The fact that the two word phrase is followed by one of these words over 40% of the time helps explains why a suffix array can be used as the basis of a compression algorithm (See section 1.5.3). The phrase "λέγοντες ὅτι" (saying that) also occurs 105 and the similar phrase "λέγω ὑμῖν ὅτι" (I say to you that) occurs 52 times. (The fact that these two phrases need to be counted separately shows one limitation with suffix arrays which it may be possible to combat using the techniques described in Section 2.2.)[6] A couple of these examples show some of the subtleties of Koiné Greek grammar. In English, when we say that something happens the word "that" implies that we are summarizing something in our own words whereas in Koiné Greek it is also proper to use "ὅτι" to introduce a direct quotation and this helps explain why the construction is so common. Also, in English it would be difficult to construct a sentence with the phrase "λέγω ὑμῖν οὐκ" (I say to you not), but Koiné Greek has a far more fluid word order which is similar to Russian. In Greek it is quite proper to begin a sentence with a verb and it is also quite proper to negate this verb—this construction puts emphasis on the verb and the fact that it's negated. A suffix array is a useful tool partially because it brings these type of grammatical subtleties together.

Another two word phrase which is especially interesting to those who don't know Koiné Greek is "ὁ Ἰησοῦς" (literally the Jesus). It's interesting because in Koiné Greek proper nouns usually use the article. This may be partially because the morphology of a proper noun is often less extensive than other words, but it's easy to figure out the case of a proper noun (or any noun) using the article. For example, the dative case "τῷ Ἰησοῦ" (to Jesus) differs from the genitive case "τοῦ Ἰησοῦ" (of Jesus) only because of the article. The pair of words occurs 406 times in the New Testament. On a similar note, the two word phrase "Ἰησοῦς Χριστός" occurs 134 times whereas "Χριστὸς Ἰησοῦς" only occurs 95 times. It is important to realize

[6]The two other related phrases, "οἴδατε ὅτι" and "γινώσκετε ὅτι" which both mean "you know that" occur 87 and 42 times respectively.

that though these two phrases are almost identical, they appear in entirely different places in the suffix array because of the subtle difference in word order. It may be possible to use the approximate matching concepts introduced in Section 2.2 in order to overcome this limitation by ensuring that various mutations of the text also enter the suffix array, and the cross-referencing mechanisms in Section 3.3 overcome the problem with a more brute-force approach: if every word in every sentence is analyzed, then the order they appear in becomes less important.

Some languages use a pair of words which are usually translated as one word in a different language. One example of this is the two word phrase "εἰ μὴ" which literally means "if not" but is almost always translated "except". The pair of words occur 85 times in the New Testament and these all occur in the same place in the suffix array. It is quite common to use groups of words together in order to create better translations, and this type of phrase is precisely why the technique works. A suffix array would help even further because it ensures that all of these groups of words and their corresponding translations could be seen together.

Different languages have different ways of interpreting a double negative. The English phrase "you don't know nothing" means technically something different than what is generally meant with colloquial usage, and this is the type of fallacy that it's easy for programmers and logicians to recognize. The Koiné Greek usage is probably best illustrated in John 11:49: "Καϊάφας, ἀρχιερεὺς ὢν τοῦ ἐνιαυτοῦ ἐκείνου, εἶπεν αὐτοῖς, Ὑμεῖς οὐκ οἴδατε οὐδέν" (literally, Caiaphas, who was the high priest that year, said to them, "You don't know nothing!"). This shows that in Greek instead of one negative canceling another, they accumulate to form an even stronger negative. The most common example is a two word phrase "οὐ μὴ" built from two different forms of the word "not". In English, this is translated "absolutely not". English requires a positive to reinforce the negative because two negatives would cancel each other. By using a suffix array to search for the first and last occurrence of "οὐ μὴ", one can quickly see that it is used 93 times in the New Testament.

Common Objects for Possessive Pronouns

There are several two word phrases which occur very frequently that all use a possessive pronoun (Koiné Greek actually uses a pronoun in the genitive case for this function). These phrases indicate ownership of something, and several somethings are extremely common which is why these come up as common two-word phrases in the suffix array. Two interesting phrases are "πατρός μου" (my father) and "πατέρα σου" (your father) which occur 102 times and 43 times respectively. The number and case of each pronoun is removed with the parsing information, but the 1^{st}, 2^{nd}, and 3^{rd} person is preserved. For example, "ἡμῖν" (us) becomes "ἐγώ" (I) for the purpose of the sorting routine whereas "σου" (your sg.) becomes "σύ" (you). This is why there are two separate entries for "πατρός μου" (my father) and "πατέρα σου" (your father). The interesting thing is that both phrases usually refer to God. For example, Jesus repeatedly used the phrase "πατὴρ ὑμῶν τῷ ἐν τοῖς οὐρανοῖς" (your father who is in heaven) in the sermon on the mount (Mat 5-6) and these instances are all grouped together in the suffix array. The usage of "πατρός μου" (my father) is very similar, and even to the point that the phrase "πάτερ ἡμῶν ὁ ἐν τοῖς οὐρανοῖς" (our father in heaven) is quite common.[7]

The next most common phrase of this type is "μαθηταὶ αὐτοῦ" (his disciples) which is used 115 times almost entirely in the Gospels. All but six of these times, the phrase begins with an article which also makes it a very common three word phrase (see page 70). The phrase almost always refers to the disciples of Christ, though a few times it refers to the disciples of John the baptist (e.g. Luke 11:1; Mat 11:2), and in Acts it refers to the disciples of Paul (Acts 9:25). In the other instance in Acts 14:20, the case of the words is different "κυκλωσάντων δὲ τῶν μαθητῶν αὐτὸν" (but the disciples gathering around him). In this case "αὐτὸν" (him) still refers to Paul, but because its in the accusative case (him) rather than the genitive (of him/his) these are not necessarily disciples of Paul. This is one example where totally ignoring the case of the nouns and conjugation of the verbs

[7] In fact, this is God is addressed in the Lord's prayer, which may be the most famous prayer ever made (Mat 6:9).

created a match which may be slightly detrimental. Perhaps it's possible to create a system which prefers precise matches but allows any other as well.

Another common phrase is "κύριος ἡμῶν" (our lord) which occurs 88 times in the New Testament. In case you're curious, the Koiné Greek word "κύριος" can be translated as sir, lord, or master. In the cases where the word is typically translated "sir", "κύριος" is used as a formal greeting such as when the Samaritan woman (who had a relatively social status as a Samaritan woman) first spoke to Jesus (who had a higher status as a Jewish man) before he said who he was (John 4:11). But the full phrase "κύριός μου" (my master) typically refers to a master/slave relationship, which is how Jesus used the phrase in the parable of the shrewd steward (Luke 16:5). This is one case where consistently translating this two word phrase "κύριός μου" as "my master" will help distinguish between when the word should be translated "master" and "sir". One particularly common use of this phrase in the New Testament is "ὁ κύριος ἡμῶν Ἰησοῦς Χριστὸς" (our lord Jesus Christ). The full phrase occurs 38 times in the New Testament and an additional 11 times without the word "Χριστὸς" (Christ)—more than half of all of the occurrences of the two word phrase. Once again, the suffix array brought all of these references together. Every instance of the full phrase is from the Epistles or Acts which shows that the apostles looked at their relationship to Jesus at least partially as a master/slave relationship.[8]

One of the most interesting phrases that falls into this category is "χειρὸς αὐτοῦ" (his hand). Although this commonly refers to someone holding something in his hand or stretching out his hand, it also refers to two Koiné Greek idioms (which are occasionally found in other languages too). The first sense is expressed well by a phrase in Luke 21:12: "ἐπιβαλοῦσιν ἐφ' ὑμᾶς τὰς χεῖρας αὐτῶν" (they will lay their hands on you). Here, the concept of "χεῖρες αὐτῶν" (their hands) refers to someone

[8]This is confirmed in Rom 1:1 where Paul identifies himself as, "Παῦλος δοῦλος Χριστοῦ Ἰησοῦ" (Paul, a slave of Christ Jesus). Many English translations translate "δοῦλος" as "servant" instead of "slave"; possibly because slaves in Roman times were usually treated much better than slaves in the Southern U. S. in the 19th century. Some translations including New King James and New American Standard use the term "bond-servant", and Holman Christian Standard does use the literal translation "slave".

having physical control over someone else. Another common use of this phrase is "ἐξέφυγον τὰς χεῖρας αὐτοῦ" (I escaped his hands) such as when Paul was lowered in a basket to escape the Governor Damascus (2 Cor 11:13). In both cases, the idiom is used to express the idea of a person with power seizing someone else. The phrase is also used to express someone's actions. In particular, sometimes something is done "διὰ τῶν χειρῶν αὐτοῦ" (through his hands), though the precise construction beginning with "διὰ" (through) is found eight times in the New Testament, three times with the article and five times without (e.g. Mark 6:2; Acts 7:25). Because of the subtle difference with the article these occur in two slightly different positions in the suffix array, perhaps this shows that it is difficult to find Koiné Greek idioms without searching for groups of words at an extremely fine level. It may also show good reason for omitting the article entirely (as suggested on page 60). Fortunately, during the work of translation, the suffix array could still be used to refer a translator to the previous times he's translated the same idiom. It would even be possible to do this automatically.

The Prepositional Phrase

Prepositions are often the most difficult words to master in a new language because the same word is used in many different situations and every language seems to have its own rules regarding which word is used in which situation. However a suffix array can be used to bring prepositional phrases together. Koiné Greek is particularly difficult because the meaning of the preposition varies depending on the case of word it modifies. For example, "κατὰ τὸν Παῦλον" means according to Paul because Paul is in the accusative case whereas "κατὰ τοῦ Παύλου" means "against Paul" because Paul is in the genitive case.[9]

The wonderful thing about a suffix array is it brings all phrases beginning with the same word together, and this can be particularly useful for observing all of the

[9]An example of the latter can be found in Acts 25:2 where charges are brought against Paul, and the former can be found in Acts 25:14 where Paul's case is described as "τὰ κατὰ τὸν Παῦλον" (literally that [which is] according to Paul).

different ways in which a particular preposition is used. For example, the Koiné Greek term "ἐν σοὶ" (in you) is used 108 times in the New Testament. The phrase is used primarily to describe various intangible things which can be found in a person. The word "εἰς" (into) is particularly interesting. It is possible to go "εἰς τὰ ὅρια" (into a house) which is used 50 times, "εἰς τὸν οὐρανὸν" (into heaven) which is used 23 times, or "εἰς τὸ ὄρος" (into a mountain) which is used 20 times. If this suffix array were of an English text, one would be more likely to go "into a house", go "to heaven", and go "up a mountain". In the New Testament, one often finds "ἐπίστευσαν εἰς αὐτόν" (they believe into him) and here "εἰς" (into) functions as a particle which is typically translated as "in" in English. The two word phrase "πιστεύεω εἰς" (I believe in) occurs 42 times in the New Testament. The fact that there are so many different ways to translate one preposition illustrates the difficulty of translating this type of phrase, and it also shows how a suffix array can be used as a building block for translation. The structure brings all of the nouns which are commonly associated with a given preposition to the same place and this allows an automatic translation system to search for an appropriate preposition very quickly.

Longer Phrases and Idioms

Many of the most common three word phrases are not interesting at all, except as a warning of what pitfalls to avoid. These phrases include "ὁ θεὸς ὁ" (the God the) and "ὁ θεὸς καὶ" (the God and) which are both occur in the New Testament roughly 130 times. As mentioned on page 67, the phrase "οἱ μαθηταὶ αὐτοῦ" (his disciples) is used 108 times in the New Testament. Many of the most interesting three word phrases are interesting four word phrases without the article.

One such four word phrase is "ὁ υἱὸς τοῦ θεοῦ" (the son of God). This phrase occurs 27 times as a four word phrase with the article and an additional four times without. It's also used an additional ten times as a two word phrase, without any article at all (e.g. Rom 1:4). Its usage is spread throughout the gospels and epistles, and the phrase almost always refers to Jesus Christ (e.g. John 20:31 ex-

ception Rom 8:19). The fact that this phrase occurs in three different forms which vary only based on the article is good evidence it may be able to obtain more useful results in Koiné Greek if the article were dropped altogether. This concept was discussed previously on page 60.

There is one four word phrase which occurs only 24 times in the New Testament. The most fascinating thing is that every one of these occurrences is found in the gospel of John! It is the phrase "ἀμὴν ἀμὴν λέγω ὑμῖν" (truly truly I say to you). In fact when I first learned Koiné Greek and started reading the gospel of John because it's written using simple language, I quickly noticed how often this phrase repeated partially by how I could read the phrase so much faster than the rest of the text. It would be nice if a computer assisted translation system could perform a similar function if the text were translated into a new language as discussed on page 88. In this gospel, and only in this gospel, Jesus uses this phrase to introduce something important that he's about to say. A computer assisted translation system could quickly learn this phrase and speed up the translation of this one book. A similar phrase "εἶπεν αὐτῷ ὁ Ἰησοῦς" (Jesus said to him) is found in all four gospels. The interesting thing about this phrase is that the subject "ὁ Ἰησοῦς" (Jesus) never changes, but the tense of the verb and the object change substantially. This is one time when finding the root of each word really makes a difference. Sometimes the aorist tense "εἶπεν" (he said) is used, and other times the present tense "λέγει" (he says) is used. Although the object is always in the dative case, sometimes it's masculine "αὐτῷ" (to him), sometimes it's feminine "αὐτῇ" (to her), and quite often it's plural "αὐτοῖς" (to them). In the case of a word study or cross-referencing system, making these phrases identical is a really good idea. In the case of computer assisted translation, each of these phrases would usually be translated somewhat differently, but the way the different phrases are translated could have an impact on each other. Perhaps the best mechanism would be to use the suffix array for the lexical form of each word as I have done, but find similar phrases by using the parsing data which is loaded with each word. This is discussed further on page 85.

There is one four word phrase which is often considered the key of what the synoptic gospels are about. In the book of Matthew, this is almost always referred to as "ἡ βασιλεία τῶν οὐρανῶν" (the kingdom of heaven), whereas in the other books of the bible it is called "ἡ βασιλεία τοῦ θεοῦ". The first phrase is used 31 times—exclusively in the book of Matthew—and the second phrase is used 64 times in the New Testament. The phrase "εἰς τὴν βασιλείαν τοῦ θεοῦ" (into the kingdom of God) is used 15 times along with its counterpart "εἰς τὴν βασιλείαν τῶν οὐρανῶν" (into the kingdom of heaven). In one way, this is nothing new because biblical scholars have known about this for a great deal of time, but it is interesting to see how the suffix array brings this phrase to the forefront. Once again, automatically translating such a long phrase after the first instance is translated manually could speed up the translation process and improve the consistency of the final product.

There are very few long phrases which occur more than a couple times. The most interesting of these is Paul's greeting in his letters which was discussed previously on page 62. One additional nine word phrase: "ἐκεῖ ἔσται ὁ κλαυθμὸς καὶ ὁ βρυγμὸς τῶν ὀδόντων" (where there will be weeping and gnashing of teeth). This is how Jesus often refers to "γέεννα" (Gehenna, hell).[10] The phrase occurs six times in the book of Matthew and once in the book of Luke. Elsewhere, the term "κλαυθμός" (weeping) is used when people will never see each other again, such as when Paul left Ephesus or children are killed (Acts 20:37; Mat 2:18) whereas the term "βρυγμὸς" (gnashing) seems to be restricted to this one phrase. Once again, this shows the usefulness of using relatively long phrases to assist in translating a single document because the translator would only need to look up these words the first time he translated the phrase. The next time the entire phrase would be available to him.

The final long phrase is a 10 word phrase which is repeated nine times in the first three chapters of the Revelation of John but occurs nowhere else in the New Testament. It's the statement: "ρο ἔχων οὖς ἀκουσάτω τί τὸ πνεῦμα λέγει ταῖς

[10]Gehenna, also known as the Valley of Hinnom, is a place outside of Jerusalem that was used as a garbage dump. It is also the place where some of the people of Judah, including some kings, would sacrifice their children to the god Molech (2 Chr 28:3, 33:6; Jer 7:31, 19:2-6).

ἐκκλησίαις" (the [one] having ears, let him hear what the spirit says to the churches). This is another phrase that John repeats so often in a small text that it's obvious to the reader without the use of a suffix array! In the beginning of Revelation, John is writing to seven churches, and he uses this phrase partially as a way to introduce each church and partially to separate them from each other. The fact that such a long phrase can occur in just a few chapters shows that sometimes a computer assisted translation could be useful without a previous corpus even in some of the shortest documents.

3.3 Cross-Referencing the Greek New Testament

If a thesis can have a climax, then this is probably it. This is the place where a suffix array is applied rather than studied. (Unlike most good literature, however, Section 4.2 is almost like a second climax in the middle of the dénouement.) By creating a meaningful cross-reference for the New Testament in just 10 seconds on a 1.2GHz computer, I believe I have done something which has seldom (if ever) been done before. Although this type of speed is not necessary for creating cross-references in a small corpus such as the Koiné Greek New Testament. As mentioned on page 86, this technique could be applied to a much wider corpus than just the New Testament. It could be used on all Koiné Greek works, modern legal documents, or even E-mail spam.

3.3.1 What the Algorithm Does

Often when a person is reading something, they find that it reminds them of something else. This is why this thesis contains many cross-references within itself. Pastors and biblical scholars have often used this type of tool to study the New Testament, and many Bibles are printed with a column devoted to cross-references of the text. Perhaps the simplest example of how useful this can be comes from

studying the four gospels which each give a slightly different account of Jesus' life. Many of the stories from one gospel are also found in another, but they usually do not occur in the same sequence and they can contain subtle differences or emphases. This is likely due to the fact that most of these stories condense the events from an entire day into a few short paragraphs, and different people will have slightly different memories from the same day. Because of this, it is very useful to find where the events from one gospel is repeated in another, and we find in Section 3.3.3 that this algorithm does just that.

3.3.2 How the Algorithm Works

The best way to create a cross-reference of the New Testament would be to start with a single verse which I will call the reference point. Although the reference point can change, it doesn't change extremely often. This verse would serve as a home base or an anchor to which all other verses in the New Testament would be tested. One could then read through the rest of the New Testament reflecting on the impact and similarities between the verse serving as the reference point and the verse currently being read. I will call the verse currently being read the test point. The test point is a fleeting thing which changes to the next test point as soon as the previous one is considered. If any similarity is found between the reference point and a particular test point, then a cross-reference can be created which will link these two verses. By treating every verse as a reference point once, and scanning through the rest of the verses as test points each time, every verse can be compared to every other verse. Eventually one would create a very good cross-reference of the entire New Testament. Each time a new verse is chosen as a reference point is chosen, the process of finding the best test point would begin anew.

Unfortunately, this would take a prohibitively long time for a person to do manually because it is an $\mathcal{O}(n^2)$ algorithm which takes four times as much effort for twice as much data. For example, if there were just 10 verses to cross-reference, one could generate a cross-reference using this technique by reading through each

of these verses 10 times, so 100 verses would need to be read. In reality, only half this many verses would need to be read because in the first pass the reference point would be verse 1 and the nine other verses would serve as test points. In the second pass verse 2 would not have to be compared to verse 1 because the comparison was already made when verse 1 was the reference point. However it may still be a good idea to use verse 2 as a reference point because this could achieve different results. What if the first part of verse 2 is somewhat close to the first part of verse 1, but the second part of verse 1 is almost identical to verse 4? In this case Verse 1 would refer to verse 4 and Verse 2 would refer to verse 1. Because there are 7942 verses in the New Testament, this technique would require reading 63 million verses to create a full cross-reference.

Using a computer to compare 63 million verses is not prohibitively complex, but it wouldn't be particularly fast either. In order to create my cross-referencing system I use a suffix array to speed up the process. Although technically I cannot prove that the mechanism is any faster than $\mathcal{O}(n^2)$, practically it is more than fast enough to do the job. The speed at which it operates depends on how many common words are in the vocabulary. For the New Testament, it took just 10 seconds to reference the Koiné Greek New Testament on a relatively slow 1.2GHz Pentium 4M computer. Because 3.5 seconds was spent generating the suffix array, it took 6.5 seconds to run through this algorithm.

The only way for a computer to figure out if two things are similar (without using a thesaurus or creating a terribly complex semantic model) is to look for common words and common phrases. Fortunately, as we have just seen is Section 3.2.5, the suffix array brings all common words and phrases together to the same spot. I exploited this property of the suffix array to compare a reference point with every possible test point while ignoring the impossible ones. The biggest thing the suffix array allowed me to do is avoid searching through test points which in fact had no words in common at all.[11] This is the key to this method of creating cross-references.

[11]On a technical side, this required creating a reverse index. By definition, a suffix array would index the position in a text where a word or phrase is found, but I also needed a reverse index to

I created a procedure that searched the suffix array to find the verse (i.e. the best test point) with the most words and phrases in common with the current verse (i.e. the reference point).

I will now consider the problem of comparing a single verse (the reference point) with all other verses in the New Testament (each test point). This is done by maintaining a score between the reference point and every possible test point. This could be done using various data structures to store the current list of verses that have been tested, but I chose to use a slightly different approach. I created a data structure with an entry for every verse in the New Testament and each one of these entries had a number which represented the current score, sort of a running tally, for the similarity between the reference point and the test point. Whenever a new word or phrase was found between the two verses, the corresponding score was increased by the appropriate value. I also kept three variables which held the verse number and the score of the current top three contenders. These variables were update whenever a the score between the reference point and a particular test point became greater than one of the other top three contenders. When all of the comparisons were done, these three variables held the value of the three best cross-references between the verse being compared (the reference point) and every other verse in the New Testament (the top three Test Points).

Someone with a background in computer science will quickly realize that maintaining a record of the score corresponding to every verse in the New Testament can be computationally expensive. The simplest way I overcame this problem is by updating the reference and value of the top three verses (i.e. test points) on the fly. This meant that I didn't have to look a second time at a test point that only had one fairly poor match with the reference point. But because I wouldn't look at this reference point again, the data structure storing all of the potential scores become corrupted with irrelevant information. Instead of clearing the data structure

find where a word is found in the suffix array. Fortunately, I had already generated this reverse index when I implement the Longest Common Prefix (LCP) algorithm created by Toru Kasai et al. [18]. The LCP is described in Section 1.7.

after each reference point was tested, I created a second field associated with each verse that kept track of whether a particular score was outdated.[12] Whenever a new reference point was chosen, a number would be incremented. Whenever information about a test point had an outdated number, it would be ignored. This simple mechanism allowed me to avoid the cost associated with a more complicated data structure and avoid the cost associated with clearing such a large data structure. The data structure which contained the current scores only had to be reset once at the beginning of the program and again whenever the "outdated" field was about to overflow (after a few billion reference points had been considered).

Comparing Individual Words and Phrases

The reader may now wish to refresh his or her memory about what the suffix array looks like by referring back to Figure 3.1 on page 56 because this will be helpful in understanding how the suffix array finds common words and phrases between a reference point and a particular test point. The first word of the reference point is considered first, and therefore the position of this word is found in the suffix array. At this point, there is also a value for the Longest Common Prefix (LCP) which represents the longest phrase this has in common with the verse at the previous position in the suffix array. If the LCP is zero, then there are no words in common, if it's one there's one word in common, etc. Let's say that the LCP is 2, so there is a two word phrase at the beginning of this verse (the reference point) that is shared with another verse (the test point). Because of this common phrase, the current score for the test point is increased by the appropriate amount for this pair of shared words. Because this is the only verse that has any score at all, it will also become one of the top three closest verses because these top scores are zeroed whenever a new reference point is considered.

The next step is a little interesting. Let's say that the LCP of the current test point is 1. This means that though the reference point and the current test point

[12]This was done as a separate field in the same array so that only one cache miss would occur whenever both pieces of data were referenced (see Section 1.5.4).

have two words in common, the reference point and the next test point has only one word in common. The test point is then updated, and the score for the verse at the new test point is increased by the appropriate amount for one word in common.

Now for the tricky case. Let's say that the LCP at the new test point is 10. This means that this verse shares a ten word phrase in common with the previous verse in the suffix array. However it does *not* mean that either of these test verses have a ten word phrase in common with the reference point. The reason for this and an example was described in Section 3.2.2. Instead, the newest test point still has just one word in common with the reference point, and so the verse associated with this place in the suffix array is increased by the appropriate amount. Let's say the LCP at this final test point is 0, which indicates that there are no more words in common in this direction.

This same process must be repeated going down the suffix array because usually there will be shared words in both directions. Finally, the entire process must be repeated for every word in verse at the reference point, first searching up, then searching down the suffix array. This will ensure that any verse that has any significant word in common will be checked, even though the process will consider far less data than the entire New Testament.

About the Fuzzy-Logic

I came up with a very simple fuzzy-logic system to score various matches. Although I believe the basic system is sound because I have proved this in Section 3.3.3, I also think that it could be improved substantially by tweaking the numbers. A normal logic system deals with true and false values. All or nothing. Zero or one. A traditional fuzzy-logic system allows for some ambiguity by allowing any fraction somewhere between 0.0 and 1.0 so 0.5 could be used to represent a 50%–50% chance. I opted to use integer arithmetic instead because the numbers use less space and it is substantially faster on many platforms. It may be better to convert this into a floating-point system (one that allows decimal places to be used) before trying to

tweak the performance.

I first started by coming up with a score for every word. This was based on how unique it was in the Koiné Greek New Testament. The words were counted as the data was being loaded. Any word used many thousands of times such as "ϱο" (the) and "καί" (and) were given a score of 0 because they weren't interesting at all. This also allowed the process of finding new test points for a particular reference point to be aborted when two verses shared only uninteresting words, and this saved a great deal of time. The more unusual a word was, the more its score was. For example, if a word which is only used a few times in the New Testament, it would have a score of 100, but if it was used several hundred times it would have a score of 6. The precise curve I created could probably be tweaked for optimal performance, but the results show that the general idea is quite sound.

In addition to this, I put a very substantial weight on having phrases in common. If a two word phrase were being considered, each word in the phrase was counted at twice the normal value. For example, if the phrase were "ὁ κλαυθμὸς" (the weeping), then the first word "ὁ" (the) would normally carry no value, but the second word "κλαυθμός" (weeping) would normally have a value of 100. However the phrase would count as 200 because a two word phrase is counted at twice the value. In addition to this, the next word in the verse to be considered would be "κλαυθμός" (weeping) which would match the same set of verses with an additional weight of 100. This double-counting is much easier to try to compensate for rather than remove, especially since counting the same pair of verses multiple times is precisely what this cross-referencing system is all about. However I rather naïvely gave three word phrases three times the weight, four word phrases four times and so on which is almost definitely adding too much weight to longer phrases. These longer phrases are also counted when some of the words occur after the end of the verses being compared. Nonetheless, by using a suffix array, it is possible to allow a phrase of several words to carry a much greater weight than individual words, and this is an extremely powerful tool.

3.3.3 Whether the Algorithm Performed

As mentioned previously, the total time taken to calculate the cross-reference for every verse in the Greek New Testament is 10 seconds on a Pentium 4M 1.2GHz computer. It takes 3.5 seconds on the same computer just to calculate the suffix array, therefore it takes about 6.5 seconds to create these cross references. It is easy to see that the technique I've employed is fast with this type of data, but in this section I will endeavor to prove that the results are also good. To do this, I will look at some of the references which have a particularly high score.

There are a number of verses which this system found cross-references for across all four gospels. This is a particularly interesting type of reference to check because it is the most logical type of cross-reference one would expect. One such example is Luke 3:4 which was cross-referenced to Mat 3:3, Mark 1:3, John 1:23. Perhaps it is little surprise that this verse is actually a quotation from the prophet Isaiah which all four gospels share: "ὡς γέγραπται ἐν βίβλῳ λόγων Ἡσαΐου τοῦ προφ ητου, Φωνὴ βοῶντος ἐν τῇ ἐρήμῳ, Ἑτοιμάσατε τ᾽ην ὁδὸν κυρίου, εὐθείας ποιεῖτε τὰς τρίβους α υτοῦ. . . " (as written in the book of words of Isaiah the prophet: "The voice [of one] shouting in the desert, 'Prepare the way of the lord, make his path straight. . . ' "). Although the precise introduction to this passage changes from gospel to gospel, the quotation always refers to John the Baptist in all the gospels. The score for cross-reference linking this verse from Luke to the gospel of John is 2811, whereas the score linking it to the other two gospels is over 34000.

Another set of verses which scores relatively evenly across all four gospels is Luke 22:39, John 8:1, Mat 26:30, and Mark 14:26. This scores about 2085 points across all four gospels, and John's gospel is the most succinct: "Ἰησοῦς δὲ ἐπορεύθη εἰς τὸ Ὄρος τῶν Ἐλαιῶν." (And Jesus went up the Mount of Olives).[13] All four of these verses involve a group of people going to the Mount of Olives, but the context of the verses change. One interesting thing about this reference is that John 8:1 matches both Mat 21:1 and Mat 26:30 but doesn't match Mark 14:26. This is an

[13]The astute reader will notice that this is a place where someone goes "εἰς τὸ ὄρος" (into a mountain) as mentioned on page 3.2.5.

example of where a group of verses is not always referenced the same way in both directions due to the short length of one of the verses in question (see page 75). Another fascinating thing is that the verse referenced in John is at an entirely different place in the narrative of Jesus as compared to the other three gospels. In John 8:1, Jesus goes to the Mount of Olives to give his judgment to the woman caught in adultery (though the most reliable manuscripts actually omit this story), but in the other three gospels Jesus goes to the mount of olives to pray the night before his trial (i.e. Good Friday). The fifth reference, Matthew 21:1, is the time when Jesus first enters into Jerusalem from the Mount of Olives (i.e. Palm Sunday).

Reference	**Score**	**New Testament Text**
Mat 11:1	N/A	καὶ ἐγένετο ὅτε ἐτέλεσεν ὁ Ἰησοῦς διατάσσων τοῖς δώδεκα μαθηταῖς αὐτοῦ, μετέβη ἐκεῖθεν τοῦδιδάσκειν καὶ κηρύσσειν ἐν ταῖς πόλεσιν αὐτῶν. And when Jesus finished teaching his twelve disciples, he lead them from there to teach and preach in their cities.
Mat 13:53	2538	καὶ ἐγένετο ὅτε ἐτέλεσεν ὁ Ἰησοῦς τὰς παραβολὰς ταύτας, μετῆρεν ἐκεῖθεν. And when Jesus finished saying these parables, he left from there.
Mat 26:1	2514	καὶ ἐγένετο ὅτε ἐτέλεσεν ὁ Ἰησοῦς πάντας τοὺς λόγους τούτους, εἶπεν τοῖς μαθηταῖς αὐτοῦ... And when Jesus finished all these words, he said to his disciples...
Mat 7:28	2460	καὶ ἐγένετο ὅτε ἐτέλεσεν ὁ Ἰησοῦς τοὺς λόγους τούτους, ἐξεπλήσσοντο οἱ ὄχλοι ἐπὶ τῇ διδαχῇ αὐτοῦ. And when Jesus finished all these words, the crowds were astounded by his teaching.

Figure 3.2: Three verses in Matthew which were cross-referenced to the first verse.

It is sometimes interesting the way references are created within the same book. For example, Mat 11:1 is cross-referenced to Mat 13:53, Mat 26:1, and Mat 7:28. By looking at Figure 3.2 it is fairly easy to see why. Each of these verses talks about Jesus finishing in precisely the same way, and each time Jesus goes on to do something else. (As an aside, I found it easier to translate all of these verses in the same way because I could cut-and-paste the translation of the first part of each sentence, so it is reasonable to assume that automatically bringing these similarities

to the attention of another translator would also make it easier for them.) This shows how a single author often has a tendency to use the same phrase in the same way over and over. Perhaps it would also be good to give priority to previous translations of the same authors work both for computer assisted translation of the New Testament and also for translating other documents. The scores also show an abnormally heavy weight given to the six word phrase which is identical in all four passages.

There is a set of verses which, according to the references, don't seem to be related at all at first glance. However once again, these verses all quote the same Old Testament law. The verses are Mat 19:19, Gal 5:14, Mark 12:31, and James 2:8 which quote Lev 19:18. The common thread in all of them is the phrase "ἀγαπήσεις τὸν πλησίον σου ὡς σεαυτόν" (love your neighbor as yourself). A similar Old Testament quotation can be found in Rom 4:3, James 2:23, Gal 3:6, and Rom 4:22 which all cite the Old Testament passage Gen 15:6. The phrase is: " Ἐπίστευσεν δὲ Ἀβραὰμ τῷ θεῷ καὶ ἐλογίσθη αὐτῷ εἰς δικαιοσύνην." (Abraham believed God and it was credited to him as righteousness.) As I was looking through various translations of the word "λογίζομαι" (I reckon, consider, look upon as, credit) I noticed that the New American Standard Bible has a cross-reference between precisely the same passages this program came up with. I also noticed that they translated the phrase "credited to" in two instances and "reckoned to" the other two times. The New American Standard is an incredibly good and incredibly accurate translation I've used many times, but I find it interesting that they lacked consistency in this case.

There's a common doxology in several passages of the New Testament which were written by three different authors. The program cross-referenced 1 Pet 4:11 with Rev 1:6, Rev 5:13, and Heb 13:21 because all of these verses contain the phrase: "διὰ Ἰησοῦ Χριστοῦ, ᾧ ἡ δόξα εἰς τοὺς αἰῶνας τῶν αἰώνων ἀμήν." (through Jesus Christ, to whom be glory forever and ever, amen.) It's interesting that the two verses from Revelation scored higher than Hebrews because Revelation is missing the first four words from this phrase—the subject of both passages is given without using the

word Jesus Christ. Perhaps it's simply because the verses in Revelation are longer, and therefore have more opportunity to match other words.

Reference	Score	New Testament Text
Col 4:18	N/A	Ὁ ἀσπασμὸς τῇ ἐμῇ χειρὶ Παύλου. μνημονεύετέ μου τῶν δεσμῶν. ἡ χάρις μεθ' ὑμῶν The greeting of Paul in my hand. Remember my imprisonment. Grace be with you.
1 Thes 5:28	9888	Ἡ χάρις τοῦ κυρίου ἡμῶν Ἰησοῦ Χριστοῦ μεθ' ὑμῶν. The grace of our lord, Jesus Christ, be with you.
2 Thes 3:17	3516	Ὁ ἀσπασμὸς τῇ ἐμῇ χειρὶ Παύλου, ὅ ἐστιν σημεῖον ἐν πάσῃ ἐπιστολῇ· οὕτως γράφω. The greeting of Paul in my hand, which is the sign in all letters; this is how I write.
1 Cor 16:21	3516	Ὁ ἀσπασμὸς τῇ ἐμῇ χειρὶ Παύλου. The greeting of Paul in my hand.

Figure 3.3: Three of Paul's greeting which was cross-referenced to the first example.

Earlier, on page 62 we saw that Paul often used the same greeting in each of his letters. The cross-referencing system also referenced one of his common closing remarks when Col 4:18 was paired with 1 Thes 5:28, 2 Thes 3:17, and 1 Cor 16:21. Different parts of the verse were matched together, as can be seen by looking at Figure 3.3. It is fairly easy to see why each of these verses is similar; each one is a greeting given by Paul at the end of his letter. The first two share one part of Paul's typical greeting, and the last two share another part. What is extremely difficult to figure out, is why the references to Col 4:18 and 1 Thes 5:28 have a score which is three times as high as the other verses when they only share two phrases in common. The reason is that the second phrase is not, in fact, a two word phrase as it looks, but it is actually a 21 word phrase. What is extremely odd about this 21 word phrase is not that it spans past the end of one verse into the next one, we've seen this happen with the longest phrase in the suffix array on page 63. What is uncommon about this phrase is that it spans past the end of the verse and into the next book! Both of Paul's letters to the Thessalonians start with the same sentence: "Παῦλος καὶ Σιλουανὸς καὶ Τιμόθεος τῇ ἐκκλησίᾳ Θεσσαλονικέων ἐν θεῷ πατρὶ καὶ κυρίῳ Ἰησοῦ Χριστῷ, χάρις ὑμῖν καὶ εἰρήνη." (Paul and Silvanus and Timothy to the church of Thessalonians in God our father and lord, Jesus Christ, Grace and peace

to you.) Because Colossians comes immediately before 1 Thessalonians, and the first book before the second, this phrase was enough to make the difference between a short match and a much much longer one. This shows that the system may need a little tweaking, but it also proves that it works.

3.4 Potential for Further Research

Perhaps the most wonderful and frustrating thing about research is that it always opens new doors. Although I wish I could pursue every one of these possibilities, there comes a time when one must make an end. Nonetheless, I feel it's important to mention new topics because they are good applications for everything I have already done. This section is a little like looking at the name on the outside of each of these doors without actually knocking on any of them.

Fine-Tuning the System

As explained in section 3.3.2, I created a fuzzy-logic system that finds verses which appear similar to each other. The system was made in a reasonable fashion, and therefore I achieved reasonable results, but I did not have time to tune it to come up with the best results possible with this technique. Because the system works so quickly, it should also be possible to tweak some of the numbers in the fuzzy-logic system to achieve even better results than these. This could be done by taking a set of handmade cross-references of the Greek New Testament and running my cross-referencing algorithm repeatedly with slightly different weights. If the results became more like the handmade cross-references by putting more weight on one of the parameters, then even more weight could be put on the same parameter. If the results were inferior, then less weight could be added. The parameters which could be altered include the increased importance of finding multi-word phrases and the decreased importance of finding similar uncommon words.

It would also be interesting to try to use the suffix array to analyze the Koiné

Greek New Testament in slightly different ways. Perhaps it would be possible to disambiguate between different meanings of a word or phrase by to using the other words in each verse. This type of a mechanism may even be used to find Koiné Greek idioms other than the one mentioned on page 68. Although most of these have probably been found before, if the system could notice unusual features where a word doesn't function in the same way it does in other contexts, these peculiarities would be of great interest to linguists.

It would also be interesting to use a version of the Greek New Testament which would allow the user to see the textual variants of the original manuscripts as described in Section 3.1.2. Perhaps finding out which textual variants obtain substantially different cross-references would also aid in tracing the original text. In any case, if a computer assisted translation system were employed as described on page 88.

The way the current system uses the article but totally ignores all of the parsing information could be changed. The data structure that I used in these experiments does load the parsing information, but it just stores it without doing anything with it. As we saw on page 71, there are some cases where the same word in the same area of the suffix array can be parsed many different ways in the original text. This could be particularly important in a translation system where a previous translation which matches the tense and mood of the original word precisely could have a much higher weight than one where the tense is different. Nonetheless, a poor match would probably serve better than no match at all, therefore it would probably be better to maintain the suffix array in its current form and sort through the parsing information afterwords.

As mentioned on page 60, there are some situations where using the article just gets in the way. Perhaps the article could be included with the data-structure without being included in the suffix array much the way the parsing information is. Theoretically, if one were to use multiple suffix arrays to create approximate string matching as described in Section 2.2, it may not be necessary to omit the article in

the normal case. However I think the functionality of approximate searching could be extended even further if there were no article to get in the way. It is important to note that just as the parsing information is extremely important to a translator, so also the article can change the meaning of a text very substantially. However it is usually easier use a system that generates too many matches and then weed out the useless ones than it is to find new matches in a system that has a very restrictive suffix array.

Using the Cross-Reference System With Other Documents

The simplest way to expand this cross-referencing system would be to apply it to other Koiné Greek documents. Because the system works quickly, it should be fairly easy to process as much information as one can fit in memory at any given time. The Septuagint (i.e. the Greek translation of the Old Testament) would be the simplest place to start, and it would probably create the most meaningful results because of the extent to which New Testament authors referred to this text. However the meanings of various words have also been studied by looking at how they're used in Roman documents, letters by the early church fathers, and other ancient writings. It would be relatively simple and very interesting to apply this algorithm to a larger scope of writings to find common phrases as well as common words.

One of the most difficult things that lawyers and politicians must deal with is the sheer volume of laws, regulations, precedents, and court proceedings which have been recorded. It could be very useful for them to find documents which are similar to one that they have on hand and this cross-referencing system could be used to do it. In fact, finding similarities between research papers of all disciplines could be useful.

Using Suffix Arrays to Fight Spam

Almost everyone has to deal with at least some unsolicited e-mail. Although having tons of junk e-mail in your inbox is unpleasant, the most distressing thing is using

a filter that misclassifies some legitimate messages; as Paul Graham said, "A filter that yields false positives is like an acne cure that carries a risk of death to the patient" [15]. Ever since Graham put a note on his website which described how to use Bayesian filtering to stop spam in 2002, or perhaps ever since it was publicized through slashdot, anti-spam software and research has focused on this technique [15]. He was not the first person to achieve very good results with the technique, very promising results were obtained by using the e-mail body and header together as early in 2000 [2]. Still, his website's popularity certainly changed the way people filter spam.

In brief, Bayesian filtering works by using a corpus of wanted e-mails (ham) and unwanted e-mails (spam) to find the probability of a particular word being from a spam e-mail. The e-mail is sorted based on the combined probability of the 15 most interesting words (those that make the e-mail most spam-like or most ham-like).

The primary problem with Bayesian filtering is that the filtering is only as effective as the database it uses, and for new accounts there's often no database at all! A good database can be made by using an interactive system and white list to classify the couple hundred e-mails after which the system becomes very accurate [8]. Unfortunately, most users find the initial process rather painful. A combined database for multiple users makes initialization a little more palatable because they share the responsibility, but it also allows one user's mistake to corrupt the database for everyone and it doesn't achieve the same level of accuracy as custom databases for each user. One good approach is using multiple databases that are weighted differently depending on their size. Even better results can be obtained by grouping five words together [6].

In fact, this represents a totally different type of document which would could be incredibly useful to create cross-references with. Because current anti-spam technology usually uses individual words rather than phrases to perform its searches the technology is not as powerful as it could be. If a relatively large set of spam e-mails were put into a suffix array, a new message could be used as a reference point to be

tested against various test points in the spam corpus in much the same way that Greek was cross-referenced in Section 3.3.2. Although this wouldn't represent an incredible break through in the way that Paul Graham's Bayesian techniques did, it would allow a smaller database to be used to create a more effective barrier against spam.

Using a Suffix Array to Translate Koiné Greek into Arbitrary Languages

One final interesting thing about the Koiné Greek New Testament is the manner in which it is translated today. Although initially the New Testament was translated into several other languages, including Latin, this became uncommon until Martin Luther translated it into German in 1522 five years after starting the Protestant Reformation. Today, organizations such as SIL are translating the New Testament into hundreds of languages which do not already have a translation. SIL developed a program called "adapt it" to help people translate between two closely related languages. An interesting feature of this program is that it works without any prior knowledge of the target language. The program literally learns the target language as it goes. It is a translation memory designed to remember words and phrases, but these phrases must occur in the same order in the source and target language. It would be nice to use the data structures presented in this thesis to assist in this translation process. Most of the efforts and literature surrounding computer translation is focused on fully automatic translations of common languages whereas this is an opportunity to create a computer assisted translation system for obscure languages.

Finding the number of times a phrase is used in a target language is one of the most important steps in Statistical Machine Translation (SMT). The suffix array can retrieve this information extremely quickly by searching for the first and last occurrence of the phrase—this technique is extremely common in genetic research and could also be applied to Natural Language Processing (NLP). The fact that there are so many repeated phrases in the New Testament proves that translation

even with a single, moderately sized text can be sped up through an automated system. The problem with a small document is that suggestions would need to be generated from a relatively small but ever-growing corpus of previously translated text. However the fact that some phrases are repeated 24 times in a single book shows that even a small corpus of text could greatly speed up translation (see page 71). The wonderful thing about translations based on a small corpus is that because the current translator probably wrote a translation, he will probably like his translation. This could also help avoid inconsistencies in the way a particular phrase is translated which would improve the quality of the final result.

Chapter 4

Suffix Arrays and Music

4.1 Background

There have been no extensive musical texts preserved which are quite as old as the Koiné Greek New Testament, though a few ancient examples of written music have been found which date back to this time. The dawn of western music notation, however, can be dated back to the 9th century when a system of writing down music for a Gregorian chants developed. Two centuries later, something like a four lined staff was developed, and by the 16th century the modern system of a five line staff was commonplace. It is this modern system of sheet music which this thesis uses for its analysis.

4.1.1 Western Music Searches

There has been a significant amount of interest in using computers to search western music. One good (though brief) overview of the topic was written by Jeremy Pickens [28]. The basic problem is that unless a person has perfect pitch, a tune is necessarily recognized not as a sequence of notes but as a sequence of intervals. (And even those with perfect pitch can easily identify the same melody in a different key.) For example, a simple melody may be played at a variety of different speeds in a number of different keys. To add to the confusion, it's possible to have variants of the same melody, such as a piece that modulates from a major key into a minor key.

There has been little effort to use a suffix array to search or analyze music, though the possibility has been mentioned previously [37]. This is unfortunate because the structure seems to be perfect for the task. By using a suffix array, sequences of any length may be searched efficiently, and using data-relative data structures, even melodies in different keys could be retrieved.

4.1.2 Storing and Typesetting Sheet Music

As I was looking through various ways to generate sheet music, I was particularly impressed by the quality of the typesetting generated through MusiXTEX and the PMX preprocessor.[1] The system was developed primarily by Daniel Taupin before he died, tragically, in a climbing accident in 2003 [7]. I rejected MIDI which is the most common way to store music and transfer it between various programs because it does not allow the user to input typesetting commands. In retrospect, I probably should have developed my software around MusicXML because it is easier to find music in this format and because it is supported by a wide variety of music programs.

Preparing the Music for Analysis

Regardless of how music is typeset, the suffix array itself should be built out of music that's converted into a slightly different form. This is much the same as the way every word in Koiné Greek was reduced to its dictionary form as described in Section 3.1.1. As mentioned is Section 4.1.1, western music is written as a series of pitches whereas people generally hear music as a series of intervals. It is also impossible for the ear to distinguish between a melody written in 4:2 time from the same melody written in 4:4 time, especially if the former is played at twice the speed of the latter. In order to address these problems, in a searchable index each pitch is not usually given an absolute value, but a value relative to the previous note. The timing for each note is stored in a similar fashion. Instead of storing a "C quarter note" followed by an "E half note", it would start with a "C quarter note" (because any progressive series must start somewhere), and then move to "up a major third, twice as long".

The precise format I used was designed very loosely around PMX, but it was also

[1] After dealing with these tools, I am still very impressed by the quality of output created with MusiXTEX, but I'm rather dissatisfied with the capabilities of the PMX processor. Due to its strict meter checking, it was extremely difficult to typeset the output of the suffix array which had many partial bars that don't add up to the correct meter. I also found the system crashed when asked to produce more than 15 pages of information at a time.

designed to be similar to the Koiné Greek format I used in Chapter 3. Each input line represented a single note in the same way a line of text represented a Koiné Greek word. The first item in the line is the part number, the second is the bar number, several characters were devoted both to bar lines and the note using PMX definitions, and finally the timing and interval values were given as two hexadecimal numbers. These are the numbers which were used to generate the suffix array, and they were very similar to the word numbers described in Section 3.1.1.

The two numerical values may be of particular interest to someone interested in computing music both because they are based on relative pitch and because the PMX format specifications can be found elsewhere [7]. Each note is expressed as a 12 bit value, which is enough room for about four thousand numbers. This is split into a 4 bit number (-8 to 7) to represent the length of a note and an 8 bit number (-128 to 127) which represents the tone. Normally, this second number represents the relative pitch between the current note and the previous note as a number of semitones, but there are two special cases. The number -128 is a special case which represents a tied note. When the note is tied to the previous, the interval will always be 0 because by definition a tie will always sound at the same pitch. The other exception is the number -127 which is used to represent a rest. Once again, although the timing value is important, the pitch value is irrelevant because a rest indicates no sound at all.

The first number is a signed nibble (4 bits forming a number between -8 and 7). This represents the difference in time between the current note and the previous note. First, let's presume that the note is not dotted. Without any dots, it's possible to represent any note length as a function of 2^x times the previous length. This first number represents x in this equation. So a half note followed by a quarter is represented by $x = -1$ because $(\frac{1}{2})(2^{-1}) = (\frac{1}{2})(\frac{1}{2}) = (\frac{1}{4})$. Another way of thinking of it is that the value of -1 will always indicate that the current note is half as long as the previous note. Likewise a quarter note followed by a whole note will be represented by $x = 2$ because $(\frac{1}{4})(2^2) = (\frac{1}{4})(4) = 1$. This means that a value of 2

will always represent a note that's four times as long as the previous note.

The problem with this mechanism is that it still does not represent dots properly and dots do appear frequently in western music—in fact even the piece that I analyze for this thesis contains several dotted notes. They way I overcame this problem is by representing a dotted note as a note without a dot which is tied to another note that's half the value. Using this technique two dotted notes will still occur together in the suffix array because every other dotted note will also be represented in the same way. The only problem is that the LCP (Longest Common Prefix) may be skewed slightly because of this technique. Whenever I represented a dot in this manner I left all of the PMX information blank to indicate that nothing was to be typeset with this virtual second note.

Although I didn't implement this for this thesis, it would also be possible to verify that the same set of tied notes is always represented in the same way. For example, three whole notes which are tied together could either be represented internally as a breve followed by a whole note or as three whole notes. It would be good to develop a system to ensure that the same sounds would always be represented internally in a consistent way. Perhaps if two quarter notes which are tied across a bar line, these could be represented best internally as one half note. That way they could be found if the melody occurred in a different spot which didn't need the tie. It would be very important to ensure that the external representation remains the same as it was written. For the example of the tied quarters, it would drive a musician nuts to see a measure with an extra beat (because it now ends with a half note instead of a quarter) followed by a measure that was missing a beat (because its first quarter is missing). By storing the displayed information separately from the values that are used to create the suffix array, I have achieved this result in both Koiné Greek and music.

This method of representing music will allow every note to be based on the previous note based on relative data. It is much like a choir singing a capella. Often by the end of the piece the choir will have dropped by a semitone or more, much

to the exasperation of any members who have perfect pitch! In the same way, each note is based more on the previous note than on the where the note should be in an absolute sense. Also, just as a choir could speed up in the middle of a performance if the became too nervous, the tempo of each note is also based on the previous note rather than on the number of beats per minute. Despite the fact that the choir may be a little slow or a little high, it's still possible for the director to find the place where they're singing in a musical score.

4.1.3 Introducing the Fugue

It would be wonderful to test this algorithm against a relatively large corpus of music the same way I used a relatively large corpus of text to test Koiné Greek. But what corpus could I use? According to most music theorists, the most fascinating form of music to look for patterns in would be the fugue. These contrapuntal compositions begin with a melody which is referred to as the subject. This subject is typically answered by another voice which repeats the subject in a different key (almost always the dominant key). One group of fugues and preludes which would be particularly fascinating for any music theorist to analyze is volumes I & II of *Das Wohltemperierte Klavier* (The Well-Tempered Clavier) by J. S. Bach. He composed these pieces, "for the profit and use of musical youth desirous of learning, and especially for the pastime of those already skilled in this study."

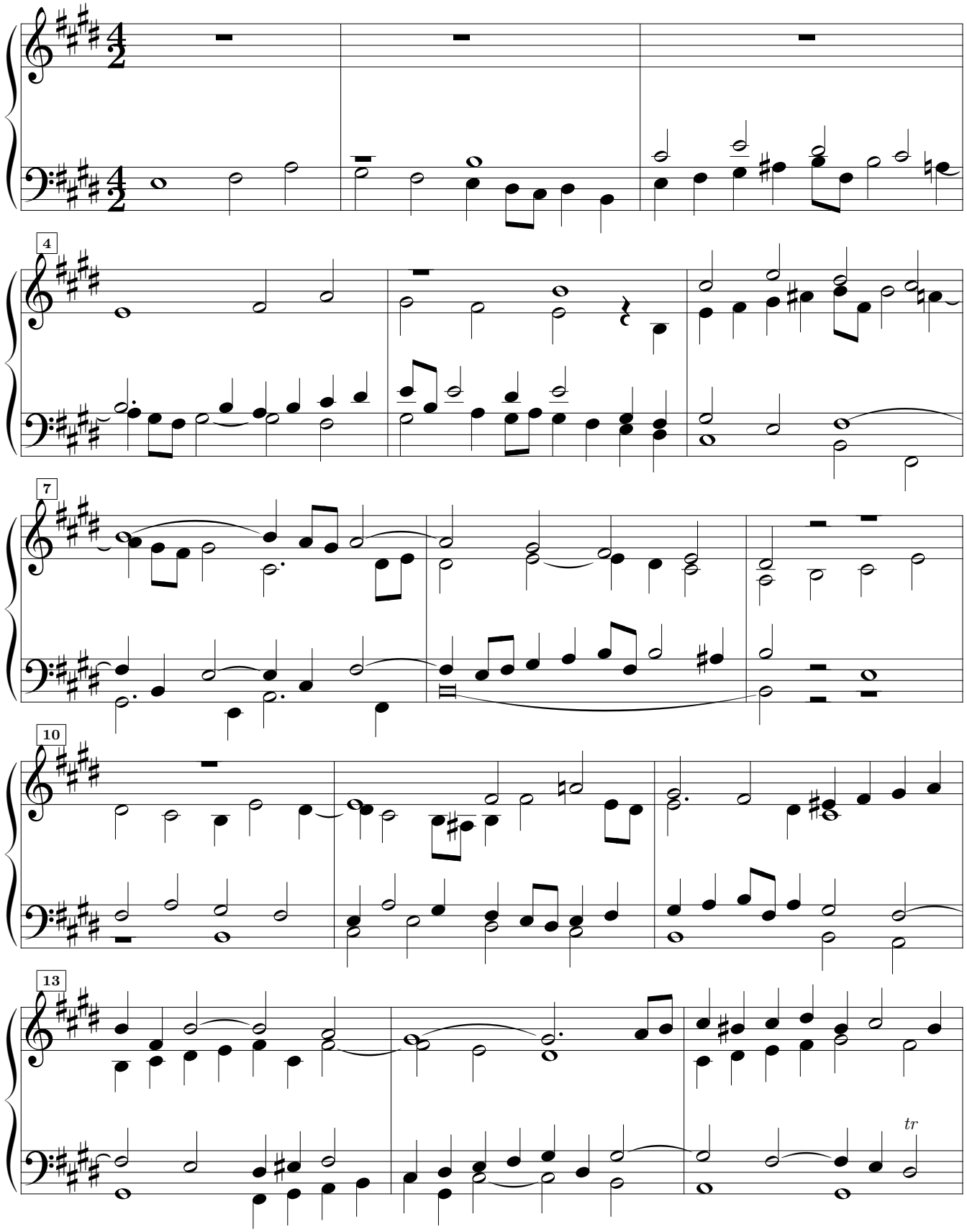
Each book is a set of 24 preludes and fugues written in every major and minor key for a musical keyboard—at the time, this would be typically be a harpsichord or clavichord.[2] Unfortunately, partially because I chose to use the PMX format, I could not obtain a copy of the entire *Well-Tempered Clavier*. I had to re-typeset the one piece of music which I did use in this thesis.

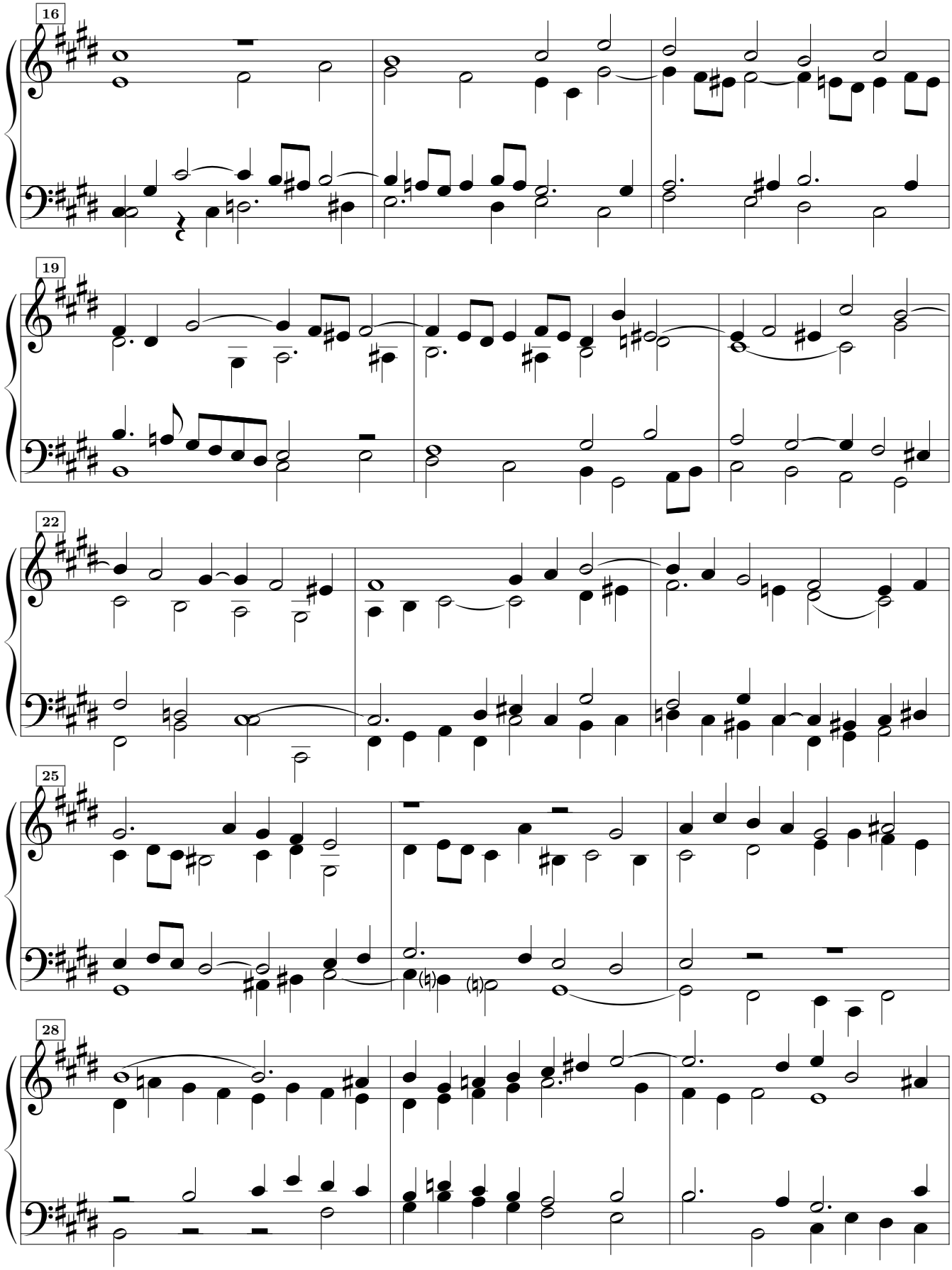
[2] Although the pieces could also be played on organs, these were usually still being tuned using the quarter-comma meantone temperament. Because they were not well-tempered, they could not play in every key. Some of the intervals in this type of tuning sound particularly beautiful, but others are "wolf-intervals" which sound terribly out of tune even to the untrained ear. Nonetheless, the quarter-comma meantone temperament was often used for tuning organs until the middle of the 19th century. Pianos had only just been invented and Bach probably didn't consider them at all for Volume I, though he could theoretically have had them in mind as he wrote Volume II.

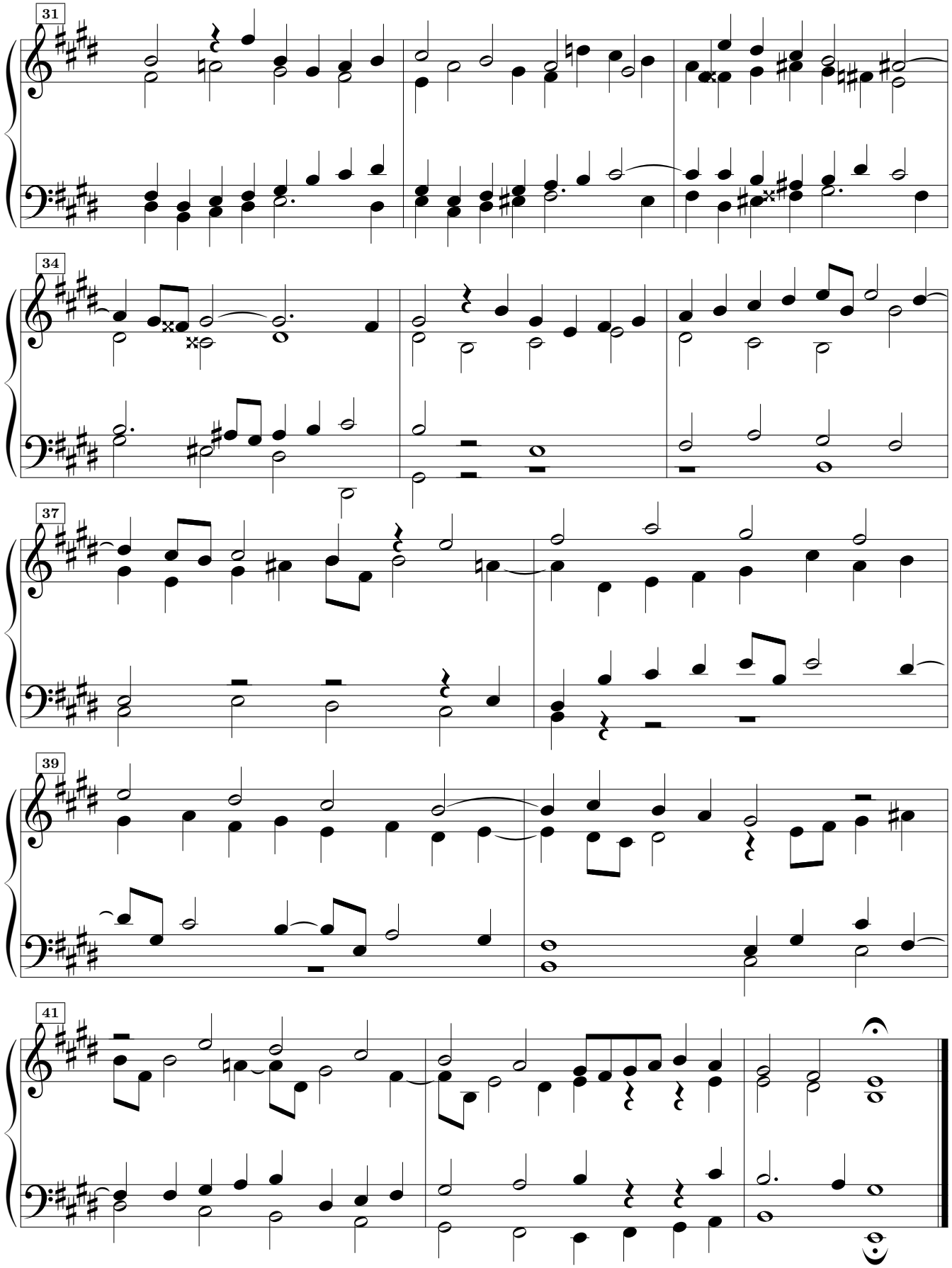
A little over a decade ago, I was watching a documentary that had an interview with Glenn Gould. As they were discussing *The Well-Tempered Clavier*, the interviewer asked if there were a fugue which was an archetype of the fugue. Gould responded by playing the E major fugue from book two of the *Well-Tempered Clavier* (BWV 878). Those familiar with Gould would not be at all surprised that he played it from memory just like the other pieces and excerpts he played in the interview. I loved the piece and learned it as soon as I had finished the documentary. It is found on the following page.

Fugue 9 from the Well Tempered Clavier II by J. S. Bach (BWV 878)

Grave




16
19
22
25
28


31
34
37
39
41

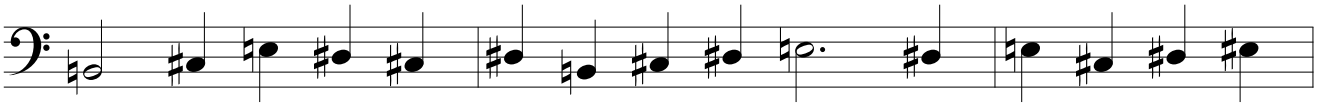
4.2 Analyzing the Fugue

There are several fairly simple things that someone can notice about this fugue. It was written in the key of E major. At just three pages, it is relatively short, but long enough to obtain useful results. It also has four parts. In fact, I used the names soprano, alto, tenor, and bass to represent the four parts even though this is not a choral piece (though it would be fun to try). But perhaps the most interesting thing about the fugue is the fact that the subject is very short and that it repeated without any alterations (i.e. it has a real answer not a tonal answer). This last point is important because otherwise it would be far more difficult to analyze using a suffix array.

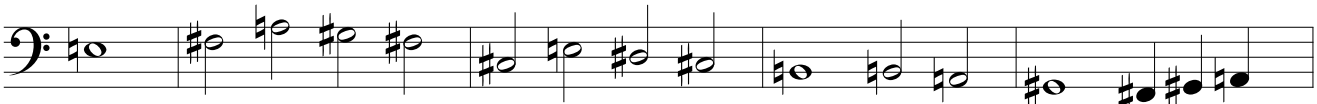
4.2.1 Finding the Subject in the Suffix Array

On the following page, I have included a part of the raw output from suffix array which contains the subject. This subject occurs unaltered in the fugue 16 different times! One can see several differences between the normal sheet music on the previous pages and the suffix array in the following pages. One difference is that the piece has four parts whereas the suffix array only considers one part at a time. Another difference is that many of the bars at the beginning and end of each line of the suffix array don't match the time signature of the piece. They are merely excerpts starting at arbitrary positions. There is also no key signature in the suffix array, instead every note is expressed using accidentals. This is somewhat awkward, but it avoids the problem of not seeing accidentals that begin before the current position in the bar. For example, in the soprano part (the top line) in bar 15, there is a b-sharp as the second note. At some point in the suffix array, the following b-sharp will be displayed at the beginning of the bar. If a person reads this line in the suffix array, they would be unable to see the previous b-sharp with its accidental, and therefore they would probably play the note as a b-natural instead. It may not be the most elegant looking solution, but I have avoided this problem by ignoring the key signature entirely and expressing every note using an accidental.

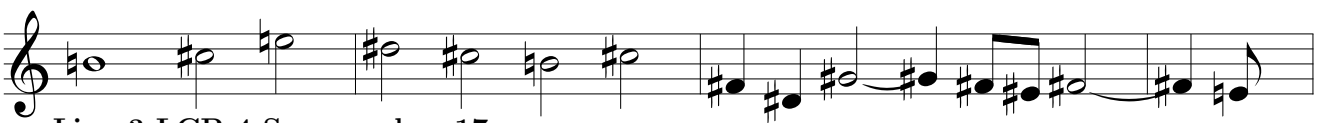
Subject from the Suffix Array of the Bach Fugue



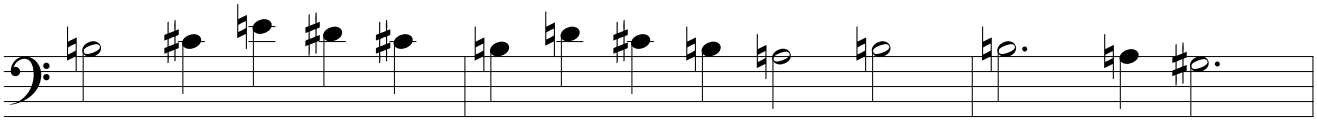
Line 1 LCP 2 Bass bar 30



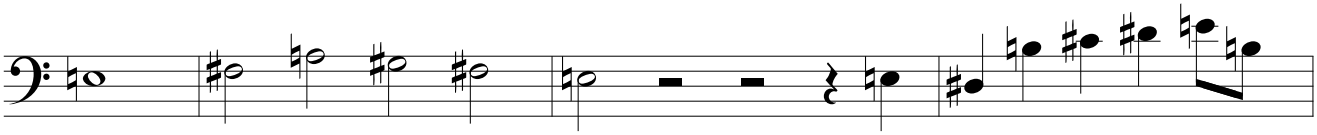
Line 2 LCP 4 Tenor bar 9



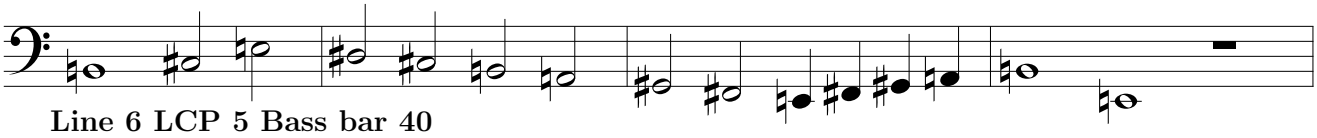
Line 3 LCP 4 Soprano bar 17



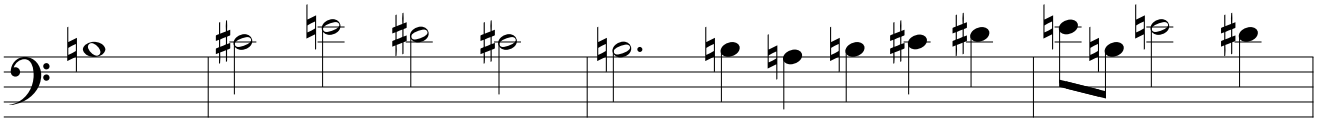
Line 4 LCP 5 Tenor bar 28



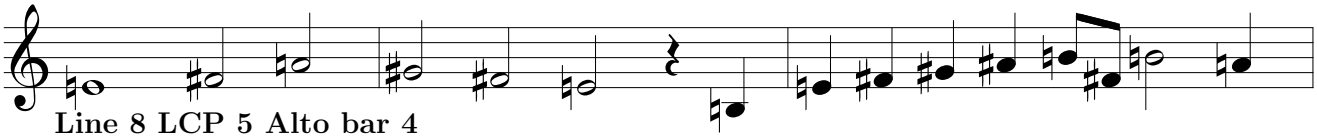
Line 5 LCP 5 Tenor bar 35



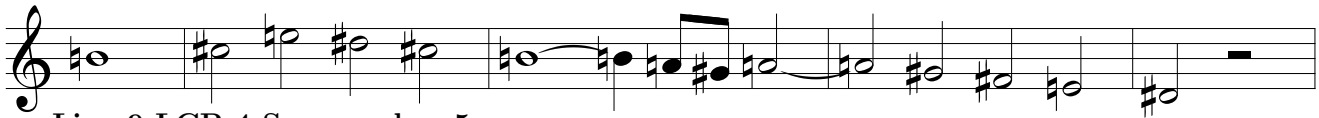
Line 6 LCP 5 Bass bar 40



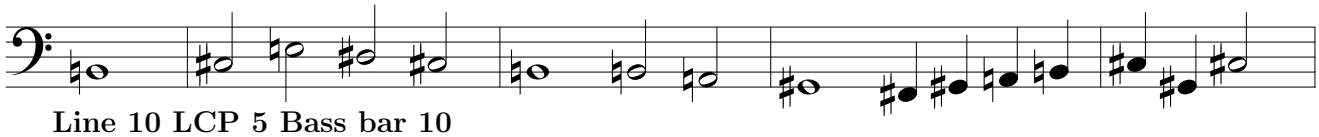
Line 7 LCP 5 Tenor bar 2



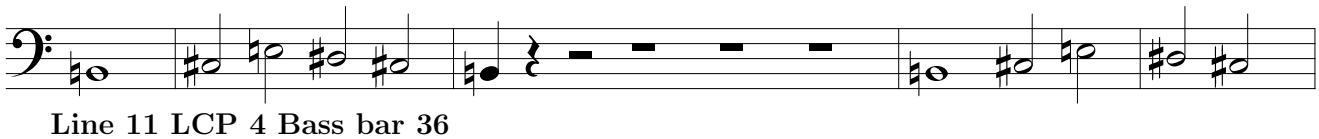
Line 8 LCP 5 Alto bar 4



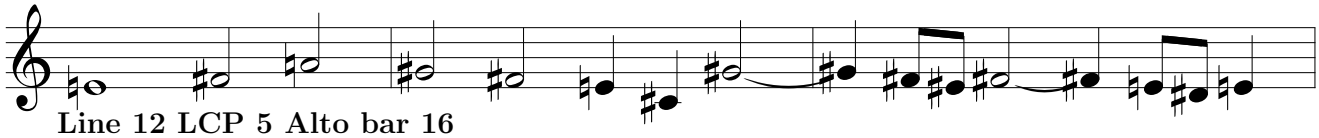
Line 9 LCP 4 Soprano bar 5



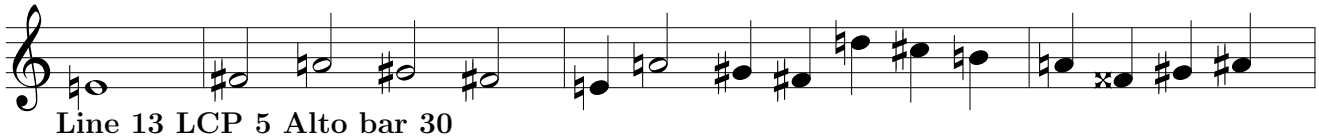
Line 10 LCP 5 Bass bar 10



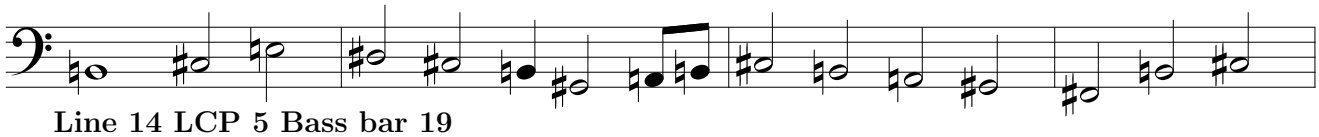
Line 11 LCP 4 Bass bar 36



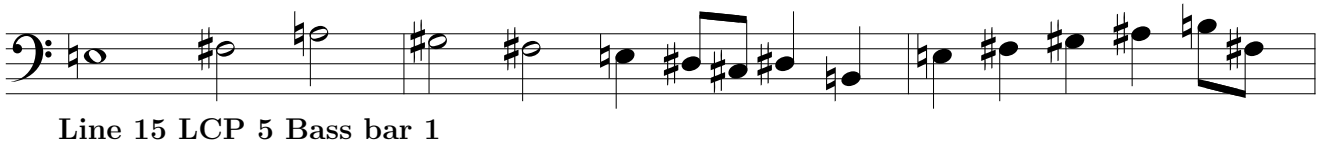
Line 12 LCP 5 Alto bar 16



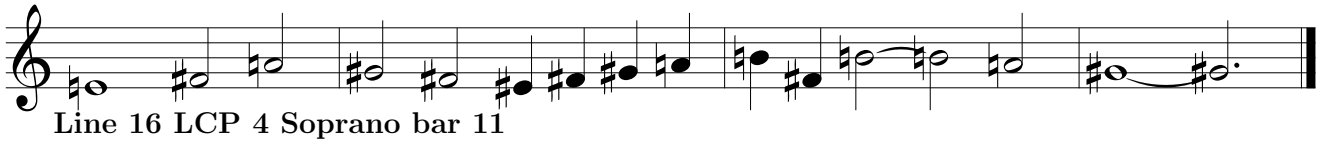
Line 13 LCP 5 Alto bar 30



Line 14 LCP 5 Bass bar 19



Line 15 LCP 5 Bass bar 1



Line 16 LCP 4 Soprano bar 11

The first occurrence of the theme in the piece is actually the 15th occurrence in the suffix array. The text below the line 15 reads, "Line 15 LCP 5 Bass bar 1". The text "Line 15" is simply a line number which makes it easier for me to talk about the suffix array using text. The "LCP" is the Longest Common Prefix which represents the number of *intervals* which this line has in common with the previous line. If you compare line 15 to line 14, you will see that they actually share six notes in common whereas line 15 and 16 share only five (the second pair is a little more obvious because it's not transposed). The reason the LCP is one less is because internally the suffix array starts at the second note which it sees as, "a note a full tone higher than the previous one, and half the length". This mechanism was described in detail in Section 4.1.2. The next piece of information is the voice which "sings" this particular part, in this case it's the bass line (the bottom line in the piece). The final piece of information is the bar number which is extremely useful for finding the part in the piece. It is useful for this chapter in much the same way that the verse reference is useful in Chapter 3.

One final interesting thing that this suffix array shows is how the relative timing and pitch information allows different kinds of matches. For example, Line 3 and Line 4 of the suffix array share the same pitch information (though on a different octave), but line 4 is twice as fast. This is one of two times Bach doubled the speed of the subject in his fugue. Line 5 and Line 6 of the suffix array show a place where the bass part is very similar to the tenor part a few bars earlier, but the tenor part is out by half a bar. This is because the bar markers are ignored in the suffix array. The place the bass actually first answers the subject presented by the tenor in Line 5 is in Line 11, and here the bass part is also offset by half a bar. The suffix array also shows how relative pitch can be used to obtain more matches. In seven cases the subject appears in the tonic key (starting with an E), and in nine cases the subject is written in the dominant key (starting with a B). If no relative pitches were used, or worse, if absolute octaves were used, then the subject would not appear at least half the time.

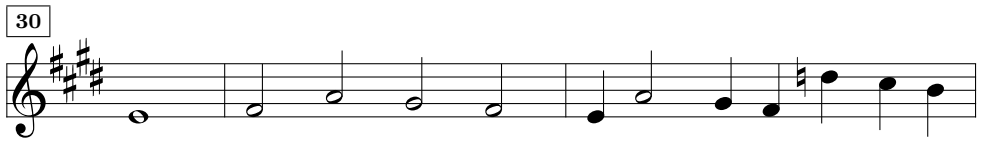


Figure 4.1: A seven note sequence taken from the Alto part.

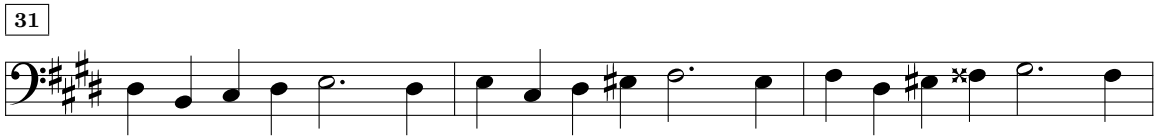


Figure 4.2: Bar 31 is repeated for 12 notes starting at bar 32 in the same line.

4.2.2 Three Other Interesting Passages

There are a few other interesting passages which are repeated several times in the suffix array. One such passage is shown in Figure 4.1. These seven notes are found in the alto part in Bar 16, this is echoed in the bass part in Bar 19, and it's repeated again in the alto part at the top of the last page in Bar 30. Although it's not too difficult to hear the way the alto part in Bar 16 is repeated in the bass a few bars later, it would be difficult to find the last occurrence of such a theme without the use of a suffix array.

There is a 12 note repetition shown in Figure 4.2 which shows Bach's mastery of counterpoint. What's most amazing about this is that the bass at Bar 31 is repeated a semitone higher in Bar 32 and this is repeated again in Bar 33 a semitone higher yet. This means that Bar 32 is functioning as the beginning of the sequence and the end of the same sequence—12 of notes are repeated despite the fact that there are only 18 notes overall! Bach switches the key repeatedly after a climax in Bar 29 gradually to build the listener up to anticipate another resolution in Bar 35. The most amazing part about this is the fact that this sequence not only sounds the same, but every tone is precisely the same interval as the previous one (presuming equal temperament). This is the type of data that suffix array algorithms such as the one presented in Section 2.1 must use inductive reasoning to resolve.

No discussion of this fugue would be complete without mentioning its counter-

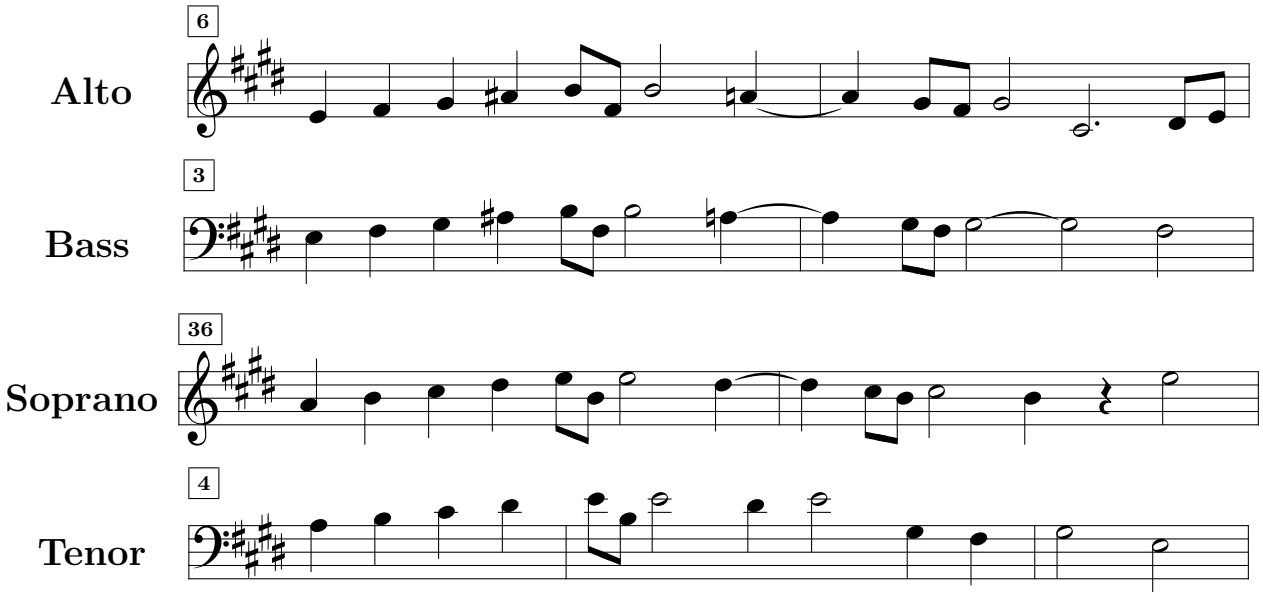


Figure 4.3: The countersubject is in different positions in all four voices.

subject. This is found once in each of the four voices, and the order shown in Figure 4.3 is the order they occur in the suffix array. By definition the countersubject is written to harmonize with the subject, and by refering to the original piece using the bar numbers, one can see that this happens each of the times the countersubject occurs. There are many notes in common in this countersubject, especially between the alto, bass, and soprano parts.

4.3 Potential for Further Research

The simplest way to extend the research of suffix arrays and music is to create a larger corpus of music. Although a fugue serves as a particularly good example—good enough to prove that the data structure is useful for analyzing music—the techniques presented in this thesis would be more useful if a larger body of music were analyzed. It would be good to search for common passages among a composers complete works, and it could be even better to do so for an even larger body of music.

One of the largest examples of compiling a large corpus of music is that the

suffix array could be used for searching through the entire corpus very efficiently. Most such tools search only for common themes which are indexed separately from the piece, but with a suffix array, thousands of scores could be searched easily and simultaneously. Even better than this, accurate frequency information for an excerpt could be retrieved passage can be retrieved in the time it takes to make two searches using a suffix array—one to find the first entry, and one to find the last. The difference between these two references represents the number of times the passage occured. This information could be extremely useful to music theorists who wish to determine how often a set of intervals are found in musical compositions.

The greatest advantage of PMX is that it creates really good looking sheet music in a manner that's flexible enough to allow changes in a score, but its greatest problem is that very few programs export to the PMX format. One way to make this research more applicable and more useful would be to allow it to use different formats. Although MIDI would be useful because it is extremely common, it could be better to use a format that is designed for typesetting music rather than listening to music. The most common format of this type seems to be MusicXML. This format can be read and written by over one hundred programs. Perhaps different formats could be employed simultaneously because the internal format used by the suffix array would have to be different than the display information anyway, as described in Section 4.1.2.

This thesis does not do anything to track chord progressions because only one note is considered at any given time. One of the problems with tracking chord progressions is that the algorithms used to find the current chord could easily become more complicated than the algorithm for the suffix array itself! Finding a chord for a given set of notes would be relatively easy except that western music allows various non-chord tones to "pollute" the data. Just considering neighbor tones, passing tones, and pedal points would be enough to make a programmer's head bleed. If this problem could be solved even partially, however, the resulting chord information would be extremely interesting to a music theorist, and there's no reason that this

data could not be stored in a suffix array. This is a large part of the way people perceive music, so it would be a useful project to pursue.

Approximate Matching for Music

Beyond transforming music to record intervals and times for each note rather than an absolute times and pitch, it would be possible to set up an approximate string matching algorithm for music which is similar to the one described for text in Section 2.2. Although skipping individual notes could be somewhat useful (as suggested for approximate string matching for text), music is a different type of language that would be better served through a different kind of changes. It's not uncommon for a composer to write a theme and then changed it between a major and minor key. It would not be extremely difficult to generate two sets of data from a piece of music written in a minor key: one for the music as written and another which is transformed into the parallel major key. This concept could be extended to include music modes other than major and minor. Given a small enough corpus or a large enough memory, it is possible for this type of system to match pieces of data that the ear would consider related but a computer would have difficulty working with.

The final and perhaps most difficult change would be to adjust the rhythm to match the same melody expressed in different ways. One moderately simple example of this is changing between straight rhythm and swing, but even this would be difficult. It could be easiest to remove the rhythm information altogether for these types of searches. A fugue is designed to be so intricate that the repetition is moderately easy to see in the music but difficult to hear among all the other sounds. Perhaps the greatest challenge to an algorithm such as this would be to find parallels which are designed to be easy for the listener to hear but difficult to see on the page. One such piece is an organ work created by J. S. Bach: Passacaglia and Fugue in c minor (BWV 582). A passicaglia repeats the same theme many times with a different harmonization each time. The theme is usually found in the bass line and often it is very easy to spot. However, other times various rests and

ornamental notes are added which make the melody harder to see. By listening to these sections, one can usually perceive the melody fairly easily because the most important notes still come on the beat. There are some times when it is difficult even for the listener to make out the melody, but the chord progressions are the same. Finding this theme in each of its versions would be an excellent test for a system that is designed to overlook subtle rhythmic variations.

Conclusion

The suffix array data structure was developed for computational linguistics, utilized in data compression, and wholeheartedly adopted by genetic research. It is a relatively new data structure which has attracted a great deal of research over the past decade. The problem of constructing a suffix array has been of particular interest and this topic has attracted most of the attention in recent papers. Several other concepts have also been explored, such as using a suffix array to search data which is compressed. In general, a suffix array is a mechanism for creating a searchable index. It is particularly useful for finding the number of occurrences of a query and searching through data with no word boundaries.

This thesis has presented two novel uses for the suffix array. Although it's fairly common to use a suffix array to process language, this is usually done character by character. Instead, this thesis uses a numeric index of stemmed words as its alphabet in order to find textual similarities at a sentence level rather than a word level. Such a large alphabet requires choosing a construction algorithm rather carefully. Although using a suffix array to process words is not unheard of, using it to process Koiné Greek is even more novel. I know that a searchable index of an English translation New Testament has been created using a suffix array as a proof of concept, but this is could be the first time it's used as a mechanism for analyzing the Greek text. Although creating a searchable index for western music is also relatively rare (probably due to the numerous musical formats and the difficulties associated with finding textual similarities in similar sounding passages), I believe it is very uncommon to use a suffix array as part of this process. The suffix array may be a novel tool for this task, but it seems extremely well suited. This is because, like a genetic sequence, a musical passage cannot easily be separated into words or sentences.

I have used a Suffix Array primarily to find the Longest Common Prefix in a Greek text and a piece of music. The longest prefixes correspond to similar passages in both music and language, and I have shown how the tool is particularly useful for analyzing parsed Koiné Greek and a musical fugue. The techniques used here also represent exciting opportunities for further research. They could be used to create a

large searchable index of western music or they could be used to translate the New Testament into new languages. I have uncovered new applications for the Suffix Array (or at least old applications utilizing new techniques). I have used this data structure in a way that no one else has, so by definition this thesis is groundbreaking. But I feel as if I have scratched through some topsoil only to reveal the rocks and clay which lie below.

Bibliography

[1] Mohamed Ibrahim Abouelhoda, Enno Ohlebusch, and Stefan Kurtz, *Optimal exact string matching based on suffix arrays*, SPIRE 2002: Proceedings of the 9th International Symposium on String Processing and Information Retrieval (London, UK), Springer-Verlag, 2002, pp. 31–43.

[2] Ion Androutsopoulos, John Koutsias, Konstantinos V. Chandrinos, and Constantine D. Spyropoulos, *An experimental comparison of naive bayesian and keyword-based anti-spam filtering with personal e-mail messages*, SIGIR '00: Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval (New York, NY, USA), ACM, 2000, pp. 160–167.

[3] Nieves R. Brisaboa, Yolanda Cillero, Antonio Farina, Susana Ladra, and Oscar Pedreira, *A new approach for document indexing usingwavelet trees*, DEXA '07: Proceedings of the 18th International Conference on Database and Expert Systems Applications (Washington, DC, USA), IEEE Computer Society, 2007, pp. 69–73.

[4] M. Burrows and D. J. Wheeler, *A block-sorting lossless data compression algorithm.*, Tech. Report 124, 1994.

[5] Chris Callison-Burch, Colin Bannard, and Josh Schroeder, *Scaling phrase-based statistical machine translation to larger corpora and longer phrases*, ACL '05: Proceedings of the 43rd Annual Meeting on Association for Computational

Linguistics (Morristown, NJ, USA), Association for Computational Linguistics, 2005, pp. 255–262.

[6] Shalendra Chhabra, William S. Yerazunis, and Christian Siefkes, *Spam filtering using a markov random field model with variable weighting schemas*, ICDM '04: Proceedings of the Fourth IEEE International Conference on Data Mining (Washington, DC, USA), IEEE Computer Society, 2004, pp. 347–350.

[7] Ross Mitchell Daniel Taupin and Andreas Egler, *Verner icking music archive: Musixtex files*, 2009, `http://icking-music-archive.org/software/indexmt6.html` [Online; accessed 31-August-2009].

[8] M. D. del Castillo and J. I. Serrano, *An interactive hybrid system for identifying and filtering unsolicited email*, WI '05: Proceedings of the 2005 IEEE/WIC/ACM International Conference on Web Intelligence (Washington, DC, USA), IEEE Computer Society, 2005, pp. 814–815.

[9] Minhawn Kim Dong Kyue Kim and Heejin Park, *Linearized linearized suffix tree: an efficient index data structure with the capabilities of suffix trees and suffix arrays*, Algorithmica (2007).

[10] P. Ferragina and G. Manzini, *Opportunistic data structures with applications*, FOCS '00: Proceedings of the 41st Annual Symposium on Foundations of Computer Science (Washington, DC, USA), IEEE Computer Society, 2000, p. 390.

[11] Paolo Ferragina and Roberto Grossi, *Fast string searching in secondary storage: theoretical developments and experimental results*, SODA '96: Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms (Philadelphia, PA, USA), Society for Industrial and Applied Mathematics, 1996, pp. 373–382.

[12] Paolo Ferragina and Giovanni Manzini, *An experimental study of a compressed index*, Information Sciences **135** (2001), no. 1-2, 13–28.

[13] Edward Fredkin, *Trie memory*, Commun. ACM **3** (1960), no. 9, 490–499.

[14] Alexandre Gil and Gaël Dias, *Using masks, suffix array-based data structures and multidimensional arrays to compute positional ngram statistics from corpora*, Proceedings of the ACL 2003 workshop on Multiword expressions (Morristown, NJ, USA), Association for Computational Linguistics, 2003, pp. 25–32.

[15] Paul Graham., *A plan for spam*, 2002, `http://paulgraham.com/spam.html` [Online; accessed 31-August-2009].

[16] Dong Kyue Kim Jeong-Eun Jeon, Heejin Park, *Efficient construction of generalized suffix arrays by merging suffix arrays*, Journal of KISS: computer systems and theory **32** (2005), 268–278.

[17] Heejin Park Jeong-Seop Sim, Dong Kyue Kim and Kun-Soo Park, *Linear-time search in suffix arrays*, Journal of KISS: computer systems and theory **32** (2005), no. 5, 255–259.

[18] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park, *Linear-time longest-common-prefix computation in suffix arrays and its applications*, CPM '01: Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching (London, UK), Springer-Verlag, 2001, pp. 181–192.

[19] Joao Paulo Kitajima and Gonzalo Navarro, *A fast distributed suffix array generation algorithm.*, SPIRE/CRIWG, 1999, pp. 97–105.

[20] Donald E. Knuth, Jr. James H. Morris, and Vaughan R. Pratt, *Fast pattern*

matching in strings, SIAM Journal on Computing **6** (1977), no. 2, 323–350.

[21] Pang Ko and Srinivas Aluru, *Space efficient linear time construction of suffix arrays.*, CPM (Ricardo A. Baeza-Yates, Edgar Chvez, and Maxime Crochemore, eds.), Lecture Notes in Computer Science, vol. 2676, Springer, 2003, pp. 200–210.

[22] Philipp Koehn, Franz Josef Och, and Daniel Marcu, *Statistical phrase-based translation*, NAACL '03: Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology (Morristown, NJ, USA), Association for Computational Linguistics, 2003, pp. 48–54.

[23] V. I. Levenshtein, *Binary codes capable of correcting deletions, insertions and reversals*, Soviet Physics Doklady **10** (1966), 707+.

[24] M. A. Maniscalco and S.J. Puglisi, *Faster lightweight suffix array construction*, In Proceedings of 17th Australasian Workshop on Combinatorial Algorithms, 2006, pp. 16–29.

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

[26] Joong Chae Na and Kunsoo Park, *Alphabet-independent linear-time construction of compressed suffix arrays using o(nlogn)-bit working space*, Theor. Comput. Sci. **385** (2007), no. 1-3, 127–136.

[27] Gonzalo Navarro and Veli Mäkinen, *Compressed full-text indexes*, ACM Comput. Surv. **39** (2007), no. 1, 2.

[28] J. Pickens, *A survey of feature selection techniques for music information retrieval*, 2001.

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

[30] Simon J. Puglisi, William F. Smyth, and Andrew Turpin, *The performance of linear time suffix sorting algorithms.*, DCC, IEEE Computer Society, 2005, pp. 358–367.

[31] ———, *Inverted files versus suffix arrays for locating patterns in primary memory.*, SPIRE (Fabio Crestani, Paolo Ferragina, and Mark Sanderson, eds.), Lecture Notes in Computer Science, vol. 4209, Springer, 2006, pp. 122–133.

[32] Ranjan Sinha and Justin Zobel, *Cache-conscious sorting of large sets of strings with dynamic tries*, J. Exp. Algorithmics **9** (2004), 1.5.

[33] James Tauber and Ulrik Petersen, *Ccat morphgnt*, 2006, `http://files.morphgnt.org/ccat-morphgnt/` [Online; accessed 31-August-2009].

[34] ———, *Ccat morphgnt on archive.org*, 2007, `http://web.archive.org/web/20071216141051/morphgnt.org/projects/ccat-morphgnt` [Online; accessed 31-August-2009].

[35] Yuanyuan Tian, Sandeep Tata, Richard A. Hankins, Jignesh M. Patel, and Jignesh M. Patel, *Practical methods for constructing suffix trees*, The VLDB Journal **14** (2005), no. 3, 281–299.

[36] Esko Ukkonen, *Approximate string-matching over suffix trees*, CPM '93: Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching

(London, UK), Springer-Verlag, 1993, pp. 228–242.

[37] Eiko Yamamoto, Masahiro Kishida, Yoshinori Takenami, Yoshiyuki Takeda, and Kyoji Umemura, *Dynamic programming matching for large scale information retrieval*, Proceedings of the sixth international workshop on Information retrieval with Asian languages (Morristown, NJ, USA), Association for Computational Linguistics, 2003, pp. 100–108.

[38] Mikio Yamamoto and Kenneth W. Church, *Using suffix arrays to compute term frequency and document frequency for all substrings in a corpus*, Comput. Linguist. **27** (2001), no. 1, 1–30.

[39] Jeong-Seop Sim Yong-Wook Choi and Kun-Soo Park, *Time and space efficient search with suffix arrays*, Journal of KISS: computer systems and theory **32** (2005), no. 5, 260–267.

[40] Kamen Yotov, Keshav Pingali, and Paul Stodghill, *Automatic measurement of memory hierarchy parameters*, SIGMETRICS Perform. Eval. Rev. **33** (2005), no. 1, 181–192.

[41] Justin Zobel and Alistair Moffat, *Inverted files for text search engines*, ACM Comput. Surv. **38** (2006), no. 2, 6.

[42] Justin Zobel, Alistair Moffat, and Kotagiri Ramamohanarao, *Inverted files versus signature files for text indexing*, ACM Trans. Database Syst. **23** (1998), no. 4, 453–490.