A Greedy Grammar-based Compression of DNA Strings using Compressed Suffix Array

Thesis
submitted in partial fulfillment
of the requirement for the degree of
Master of Computer Science and Engineering
in
The Faculty of Engineering & Technology,
Jadavpur University
by

Name of the student	:	Subhabrata Dutta
Class Roll No.	:	001410502019
University Registration No.	:	107850 of **2009 - 10**
Examination roll number	:	M4CSE1615

under the guidance of
Professor Shovonlal Kundu
Department of Computer Science & Engineering
Jadavpur University, Kolkata - 700 032

May 28, 2016

Certificate of Submission

I hereby recommend that the thesis, entitled as **"A Greedy Grammar-based Compression of DNA Strings using Compressed Suffix Array"**, is prepared by:

Name of the student	:	Subhabrata Dutta
Class Roll No.	:	001410502019
University Registration Number	:	107850 of 2009-10
Examination Roll Number	:	M4CSE1615

under my guidance and supervision, be accepted in partial fulfillment of the requirements for the degree of Master of Technology in Computer Technology from the Department of Computer Science & Engineering under the Jadavpur University.

Signature of thesis supervisor	:	
Name	:	Professor Shovonlal Kundu
Department	:	Computer Science & Engineering Jadavpur University

Countersigned by,

Signature of head of the department	:	
Name	:	Professor Debesh Kumar Das
Department	:	Computer Science & Engineering Jadavpur University

Signature of Dean	:	
Name	:	Professor Sivaji Bandyopadhyay
	:	Faculty Council of Engineering & Technology Jadavpur University

Faculty of Engineering & Technology

Jadavpur University

Kolkata - 700 032

Certificate of approval

(Only in case the thesis is approved)

The thesis at instance is hereby approved as a creditable study of an Engineering subject carried out and presented in a manner satisfactory to warrant its acceptance as a prerequisite to the degree for which it has been submitted. It is understood that by this approval the undersigned does not necessarily endorse or approve any statement made, opinion expressed or conclusion drawn therein, but approve this thesis for the purpose for which it is submitted.

Signature of the Examiner

Signature of the Supervisor

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Data Compression Techniques

1.1.1 Why compress data

Data compression refers to reducing the amount of space needed to store a piece of data. Although computer storage is relatively cheep, as the amount of data keeps increasing rapidly the cost of storage is a factor. However, the most important reason for compressing data is that more and more we share data. The Web and its underlying networks have limitations on bandwidth that define the maximum number of bits or bytes that can be transmitted from one place to another in a fixed amount of time.

1.1.2 Two methods of compression: Lossless and Lossy

Compression can be either *lossy* or *lossless*. *Lossless compression* reduces bits by identifying and eliminating statistical redundancy. No information is lost in *lossless* compression. *Lossy compression* reduces bits by identifying unnecessary information and removing it.

Lossless compression techniques, as their name implies, involve no loss of information. If data have been losslessly compressed, the original data can be recovered exactly from the compressed data after a compress/expand cycle. Lossless compression is generally used for so-called "discrete" data, such as database records, spreadsheets, word-processing files, and even some kinds of image and video information. Text compression is a significant area for lossless compression. It is very important that the reconstruction is identical to the text original, as very small differences can result in statements with very different meanings. Furthermore, if data of any kind are to be processed or "enhanced" later to yield more information, it is important that the integrity be preserved. For example, suppose we compressed a radiological image in a lossy fashion, and the difference between the reconstruction Y and the original X was visually undetectable. If this image was later enhanced, the previously undetectable differences may cause the appearance of anomalies that could potentially mislead the radiologist.

Lossy compression works very differently. These algorithms simply eliminate "unnecessary" bits of information, tailoring the file so that it is smaller. This type of compression is used a lot for reducing the file size of bitmap pictures, which tend to be fairly bulky.

1.1.3 Online vs. Offline models of Data Compression

A data compression method is called *online* or *off-line* depending on whether the whole input data is fed to the algorithm for once or the input comes as a symbol or block stream and the compressor have to decide actin depending on the next input. Offline methods usually have the access to a wholistic knowledge about the dta, and that accounts for the huge space complexity they have to deal with. In case of online compressors, working space becomes less a issue than the complex decision making process for each input block/symbol on the fly, without having prior knowledge of the context.

1.1.4 Entropy Encoding

Consider a discrete, finite alphabet random variable X with alphabet $\Sigma = \{\alpha_0, \alpha_1, \cdots, \alpha_{n-1}\}$.
Let $p(x)$ denote the probability of $x \in X$. Then the information associated with event $X = x$ is given by,
$h(x) = -\log p(x)$
and the information content associated with X is
$H(X) = -\sum_{x\in\Sigma} p(x) \log p(x)$
This definition of information and entropy associated with logical entities was given by Shannon[27]. When data is compressed, the goal is to reduce redundancy, leaving only the informational content. Since the length of a codeword for message $a(i)$ must be sufficient

to carry the information content of $a(i)$, entropy imposes a lower bound on the number of bits required for the coded message. The total number of bits must be at least as large as the product of H and the length of the source ensemble. Since the value of H is generally not an integer, variable length codewords must be used if the lower bound is to be achieved. A code is asymptotically optimal if it has the property that for a given probability distribution, the ratio of average codeword length to entropy approaches 1 as entropy tends to infinity. That is, asymptotic optimality guarantees that average codeword length approaches the theoretical minimum (entropy represents information content, which imposes a lower bound on codeword length).

1.1.5 Burrows-Wheeler Transform

A very promising development in the field of lossless data compression is the Burrows-Wheeler Compression Algorithm (BWCA), introduced in 1994 by Michael Burrows and David Wheeler[4]. The algorithm received considerable attention since of its Lempel-Ziv like execution speed and its compression performance close to state-of-the-art PPM algorithms. It is based on a permutation of the input sequence - the Burrows-Wheeler Transformation (BWT), also called Block Sorting -, which groups symbols with a similar context close together. In the original version, this permutation was followed by a move to front (MTF) transformation and a final entropy coding (EC) stage. Later versions used different algorithms which come after the Burrows-Wheeler transform, since the stages after the Burrows-Wheeler transform have a significant influence on the compression rate too.
BWT works as follows:
Input: $String[n]$;
Output: bwt($String$)

1. Create a $n \times n$ array $\mathcal{M}$;
2. $\mathcal{M}[0][] \to String[n]$;
3. For $i \in [0 \cdots n-2]$, rotate $\mathcal{M}[i][]$ and store it in $\mathcal{M}[i+1][]$
4. Sort all $\mathcal{M}[i][]$ lexicographically
5. $bwt(String) \to \mathcal{M}[][n-1]$

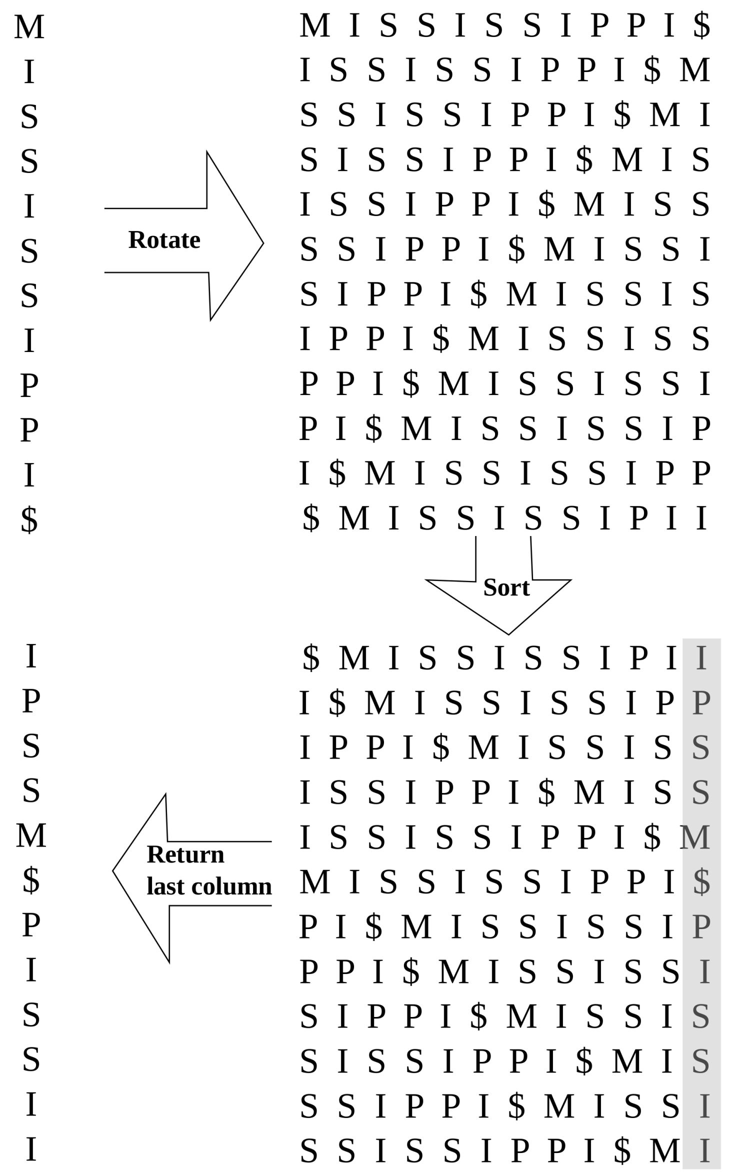


Figure 1.1: Burrows-Wheeler transformation of string *MISSISSIPPI$*

1.2 Context Free Grammar

A context-free grammar $\mathcal{G}$ is defined by the 4-tuple:[28]
$\mathcal{G} = \{V, \Sigma, R, S\}$ where

1. V is a finite set; each element $v \in V$ is called a non-terminal character or a variable. Each variable represents a different type of phrase or clause in the sentence. Variables are also sometimes called syntactic categories. Each variable defines a sub-language of the language defined by $\mathcal{G}$.

2. Σ is a finite set of terminals, disjoint from V, which make up the actual content of the sentence. The set of terminals is the alphabet of the language defined by the grammar $\mathcal{G}$.

3. $R : V \Rightarrow (V \cup \Sigma)*$ is a finite relation where the asterisk represents the Kleene star operation. The members of R are called the rules or productions of the grammar.

4. S is the start variable (or start symbol), used to represent the whole sentence (or program). It must be an element of V.

A context-free grammar provides a simple and mathematically precise mechanism for describing the methods by which phrases in some natural language are built from smaller blocks, capturing the ”block structure” of sentences in a natural way. Its simplicity makes the formalism amenable to rigorous mathematical study. Important features of natural language syntax such as agreement and reference are not part of the context-free grammar, but the basic recursive structure of sentences, the way in which clauses nest inside other clauses, and the way in which lists of adjectives and adverbs are swallowed by nouns and verbs, is described exactly.

The formalism of context-free grammars was developed in the mid-1950s by Noam Chomsky [7], and also their classification as a special type of formal grammar (which he called phrase-structure grammars).

1.3 DNA sequence and Protenes

DNA sequences are, as believed till date, the functional units of any living organism. From the most primitive Virus to the most complex Human beings, permutations of these four Nitrogen bases on the anti-parallel polynucleotide strands control every single protene synthesis for growth, reproduction and other organic functionalities. DNA usually occurs as linear chromosomes in eukaryotes, and circular chromosomes in prokaryotes. The set of chromosomes in a cell makes up its genome; the human genome has approximately 3 billion base pairs of DNA arranged into 46 chromosomes. This encouraged biologists from all over the world to start enumerating every possible DNA sequences of living humans. The Human Genome Project (HGP) is an international scientific research project with the goal of determining the sequence of chemical base pairs which make up human DNA, and of identifying and mapping all of the genes of the human genome from both a physical and functional standpoint. Though motivated by a sheer deterministic approach to define every functionality of a complex system by summing its basic units, this endeavour is enormously helpful to reveal the long hidden mystries of natural life-forms.

A DNA sequence is composed of nucleotides of four types: adenine (abbreviated A), cytosine (C), guanine (G) and thymine (T). For complete genomes, these texts can be very elongated. The human genome for example contains three billions characters over 23 pairs of chromosomes. It contains all the genetic substance of the human beings. With escalating number of genome sequences being made available, the difficulty of storing and using databases has to be addressed. Powell et al show that compressibility is a fine dimension of relatedness among sequences and can be effectively used in sequence alignment and evolutionary tree construction[25]. Conventional text compression schemes are not competent when DNA sequences are concerned. Since DNA sequence contain only 4 bases A, G, T, C, each base can be represented by 2 bits. Though standard compression tools like compress, gzip and bzip2 have more than 2 bits per base when compressing DNA data. Consequently, DNA compression has become a challenge[5].

1.4 Suffix Tree and Suffix Array

Suffix trees [22][30] and suffix arrays [21] are probably the most important data structures in stringology. A suffix tree is a trie storing all the suffixes of a string in lexicographic order.

T is a string of length n, over an alphabet Σ. Let $T = \alpha\beta\gamma$, for some strings α, β and γ (α and γ could be empty). The string β is called a substring of T , α is called a prefix of T, while γ is called a suffix of T . The prefix α is called a proper prefix of T if $\alpha = T$. Similarly, the suffix γ is called a proper suffix of T if $\gamma = T$.

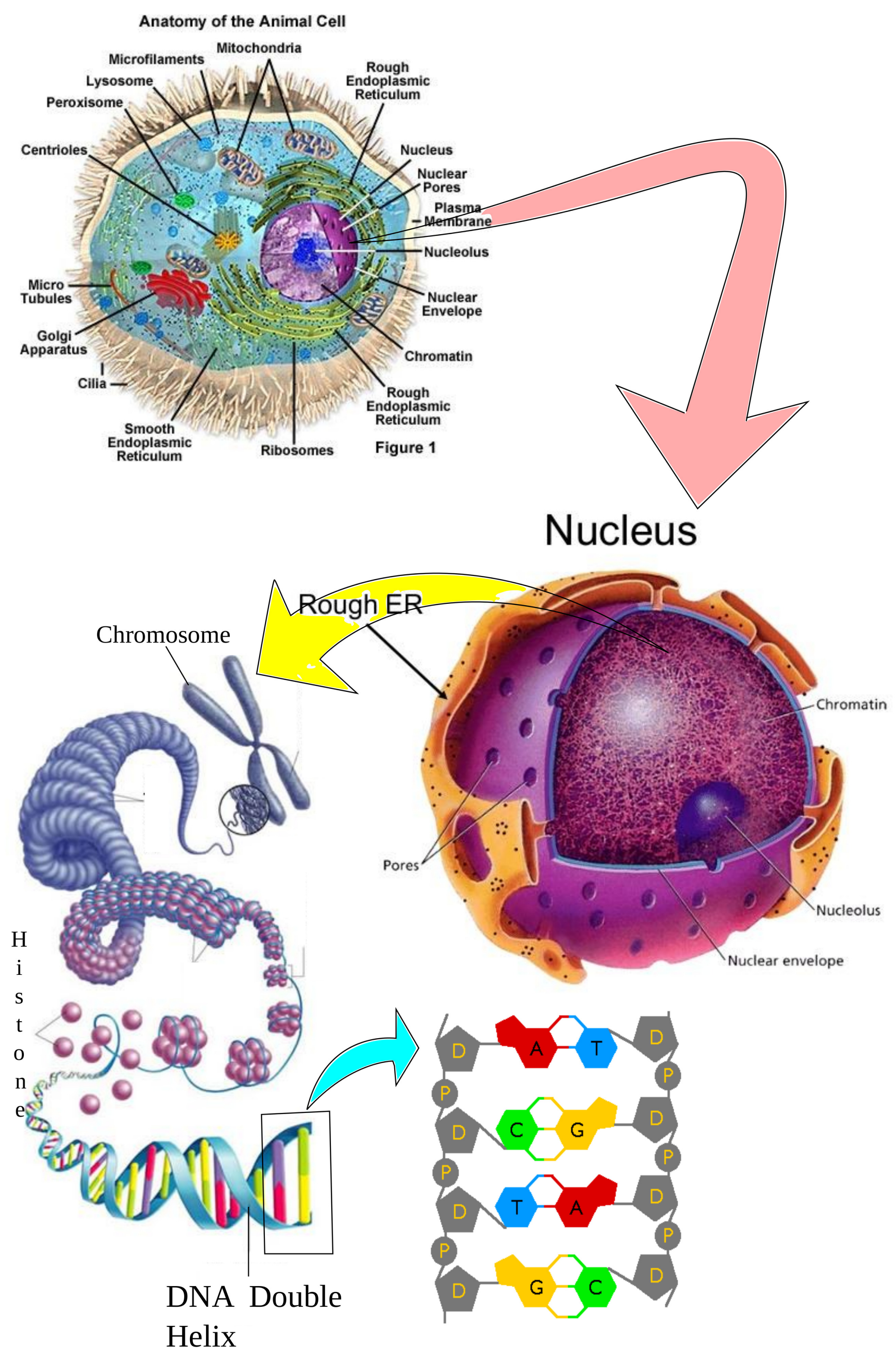


Figure 1.2: Image showing an Eucaryotic cell, its neuclear structure, chromatenes and the DNA base pair structure(image courtsey: http://ww2.valdosta.edu/;https://pmgbiology.files.wordpress.com/)

For simplicity in constructing suffix trees, we usually ensure that no suffix of the string is a proper prefix of another suffix. This can be done by placing a sentinel symbol at the end of T such that the sentinel does not appear anywhere else in T. In practice this is often achieved by simply appending a \$ to T, such that $\$ \notin \Sigma$. This constraint implies that each suffix of T will have its own unique leaf node in the suffix tree of T, since any two suffixes of T will eventually follow separate branches in the tree.

Given a string T of length n, its suffix tree $\mathcal{T}$ is a rooted tree with n leaves, where the i-th leaf node corresponds to the i-th suffix T_i of T. Except for the root node and the leaf nodes, every node must have at least two descendant child nodes. Each edge in the suffix tree represents a substring of T, and no two edges out of a node start with the same character. For a given edge, the edge label is simply the substring in T corresponding to the edge. We use l_i to denote the i-th leaf node. Then, l_i corresponds to T_i , the i-th suffix of T. For example, the suffix tree for the string *"acatgtgaca\$"* is shown in Figure 1.3.

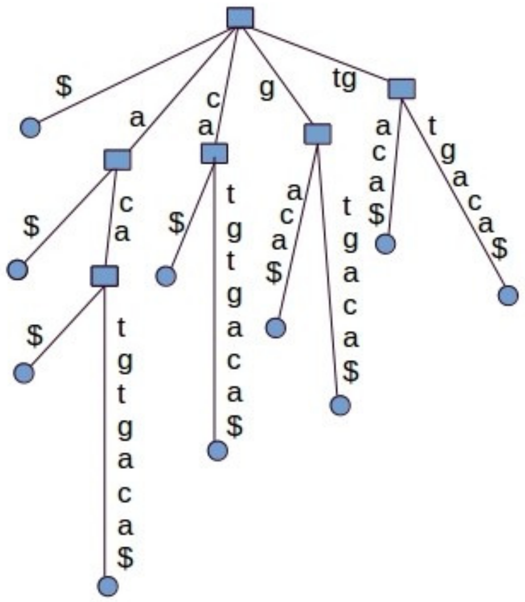


Figure 1.3: Suffix tree of the string *"acatgtgaca\$"*

Given the string T = $T[1...n]\$$, of length n, but with the end of string symbol appended to give a sequence with a total length $n+1$, the suffix tree of the resulting string $T\$$ will have the following properties:

1. Exactly $n + 1$ leaf nodes;

2. At most n internal (or branching) nodes (the root node is considered an internal node);

3. Every distinct substring of T is encoded exactly once in the suffix tree. Each distinct substring is spelled out exactly once by traveling from the root node to some node u, such that L(u) is the required substring.

4. No two edges out of a given node in the suffix tree start with the same symbol.

5. Every internal node has at least two outgoing edges.

Time complexity to build a suffix tree from an input size n is $\mathcal{O}(n)$ and $\mathcal{O}(P)$ time to report a single occurence of a single pattern of length P. Major disadnantage of suffix trees is their high space usage, which can be O(n|Σ|) for the fastest application, which is very much high for large files. To deal with this problem, the suffix array was invented. A suffix array SA is a lexicogaphically sorted array of al the suffixes of the input string. Suffix array for input string $T = acatgtgaca\$$ is $SA = [10, 9, 7, 0, 2, 8, 1, 6, 4, 5, 3]$. Entries of suffix array can be found by a simple pre-order traversal of the suffix tree. Given a suffix tree, thus building of suffix array requires $\mathcal{O}(n)$ worst case time. Although the suffix tree can be constructed in $\mathcal{O}(n)$ time, it was a long unsolved open problem whether a suffix array can be built from scratch in linear time untill 2003 [17][14].

i	T[i..n]	Sorted Suffix	SA[i]	lcp[i]
0	*acatgtgaca$*	*$*	10	0
1	*catgtgaca$*	*a$*	9	0
2	*atgtgaca$*	*aca$*	7	1
3	*tgtgaca$*	*acatgtgaca$*	0	3
4	*gtgaca$*	*atgtgaca$*	2	1
5	*tgaca$*	*ca$*	8	0
6	*gaca$*	*catgtgaca$*	1	2
7	*aca$*	*gaca$*	6	0
8	*ca$*	*gtgaca$*	4	1
9	*a$*	*tgaca$*	5	0
10	*$*	*tgtgaca$*	3	2

Table 1.1: Suffix and LCP array of *"acatgtgaca$"*

1.5 Least Common Prefix array

To report the occurence of a certain substring using suffix array, one need to compute the "least common prefix" of all the suffixes. The LCP array L = $[L_0, L_1, \cdots, L_i, \cdots, L_n]$, where L_i is the lcp-value of the i-th suffix which is the length of the longest common prefix of $SA[i]$ and $SA[i-1]$. The lcp-value of $SA[0]$ is set to zero.

In Table 1.1, lcp value of $SA[3] = 3$, because $SA[2] = aca\$$ and $SA[3] = acatgtgaca\$$ both have *"aca"* as their longest common prefix. Given the lcp array, one can easly find the lcp between any two non-consecutive suffixes. The lcp between suffixes $SA[i]$ and $SA[j]$ where $j > i$ and $j - i > 1$ is $min\{lcp[i+1], lcp[i+2], .., lcp[j]\}$.

1.5.1 LCP-interval

The concept of lcp-interval was given by Abouelhoda et. al.[1]. This is simply an interval of indices with same lcp. Figure 1.4 shows the lcp interval tree of the string *"acatgtgaca$"*. Each node of the tree contains an interval $[sp, ep]$ along with lcp-value l where the following conditions are met,

1. $lcp[sp] < l$,
2. $lcp[k] \leq l$ for all k with $sp + 1 \leq k \leq ep$,
3. $lcp[k] = l$ for at least one k with $ep + 1 \leq k \leq ep$,

4. $lcp[ep + 1] < l$

One can see the resemblence between the suffix and the lcp interval tree. In fact each node in an lcp interval tree corresponds to an internal node in a suffix tree.

i	0	1	2	3	4	5	6	7	8	9	10
SA[i]	10	9	7	0	2	8	1	6	4	5	3
LCP[i]	0	0	1	3	1	0	2	0	1	0	2

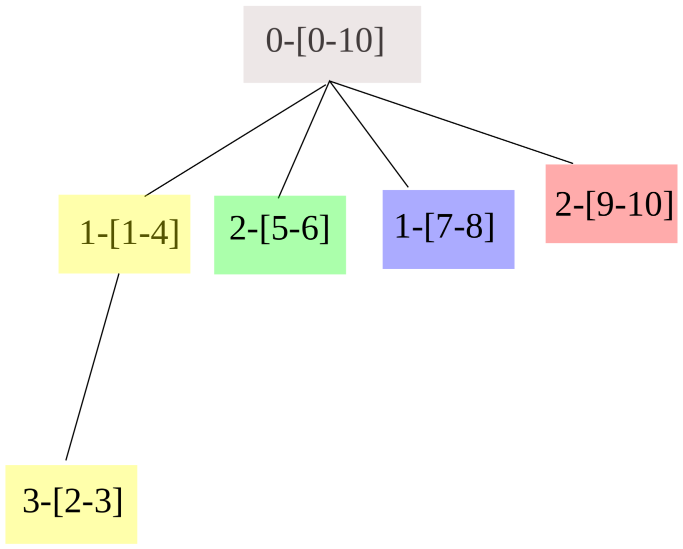


Figure 1.4: lcp-interval tree of the string *acatgtgaca$*

1.6 Compressed Suffix Array

Although suffix arrays were designed to get rid of the complex structure and functionalities of suffix trees, they still need space much bigger. To handle this problem, two succint representation of suffix array were proposed, one by Grossi and Vitter [13], and other one by Sadakane[18]. Both of these algorithms use a same feature, compressibility of Ψ-function, which is defined as
$\Psi(i) = SA^{-1}[SA[i] + 1]$, where $SA^{-1}[x] = y$ when $SA[y] = x$;

To say simply, Ψ(i) returns the index of the suffix that is next to SA[i] in input text T.

i	SA[i]	Ψ(i)
0	10	
1	9	0
2	7	5
3	0	6
4	2	10
5	8	1
6	1	4
7	6	2
8	4	9
9	5	7
10	3	8

Table 1.2: Suffix array and Compressed Suffix Array of *"acatgtgaca$"*

In the given example in Table 1.2., the $\Psi(2) = SA^{-1}[SA[2]+1] = SA^{-1}[8] = 6$. Clearly we can see that $SA[\Psi(2)]$ is *"gaca$"*, suffix next to $SA[2] =$ *"agaca$"*.

Compressibility of this Ψ-function emerges from a certain property of this function, which can be observed from the example in hand. The array representing Ψ-values of a string is piecewise sorted. To be more specific, suffixes with same leading character have Ψ-values in sorted. According to the definition of suffix arrays, $SA[i]$ is lexicographically lower than $SA[i+1]$. If $SA[i]$ and $SA[i+1]$ has same leading character, say $SA_i[0]$, then the suffix next to $SA[i]$, which is $\{SA_i[1]SA_i[2]SA_i[3]\cdots\}$ must be lexicographically lower than $\{SA_{i+1}[1]SA_{i+1}[2]SA_{i+1}[3]\cdots\}$, the suffix next to $SA[i+1]$, and thus have index smaller than that. This piecewise sortedness of Ψ-values helps to store it using far lesser than the $\mathcal{O}(n \log n)$ bits, otherwise necessary to store an array of n-integers. One can use Elias-Fano encoding to store this array into $\mathcal{O}(n)$ bits.

Since the year 2000, two families of compressed suffix arrays (CSAs) emerged [23]. One family, simply called CSAs used the defined compressibility property of Ψ-functions; and simulated the basic SA procedure for pattern searching, achieving the same $\mathcal{O}(m \log n)$ counting time of basic SAs. A second family, called FM-indexes [9][11][10] built on the Burrows-Wheeler transform of T and on a new concept called backward-search, which allowed $\mathcal{O}(|P| \log \Sigma)$ and even $\mathcal{O}(|P|)$ time to count for occurences of a pattern P.

1.7 Thesis Organization

In Chapter 1, we discussed the different aspects of data compression, self-indexing data structures and related notions upon which this thesis is based on. Next in Chapter 2, the concept of Grammar-based compression is elaborated along with the explanation of some well known algorithms. The algorithm DC3 for a linear-time construction of suffix array is discussed in Chapter 3. From this constructed suffix array, the procedure of constructing a succint representation is elaborated in Chapter 4. Next, we proceed to compute the lcp information of suffices from the compressed suffix array generated previously, in Chapter 5. In Chapter 6, a greedy algorithm to report non-overlapping substrings of a DNA sequence is proposed, using which a Context Free Grammar is generated. Chapter 7 includes the findings of running this algorithm over different data sets and results are compared with using different parameters. Chapter 8 discusses the variations that can be used for varying user needs and further improvements that can be done.

Chapter 2

Grammar-based Compression

2.1 Definition

Grammar-based compression methods are a class of lossless universal data compression algorithm which generate a context free grammar $\mathcal{G}$ for a given input string T such that, when production rules of $\mathcal{G}$ are applied in order, T and only T is constructed. Grammar-based codes are universal in the sense that they can achieve asymptotically the entropy rate of any stationary, ergodic source with a finite alphabet [16].

Example: the following grammar $\mathcal{G}$ represents the string $T = acatgtgaca\$$

$$
\begin{aligned}
R_0 &\Rightarrow R_1 R_2 R_2 R_1 \\
R_1 &\Rightarrow aca \\
R_2 &\Rightarrow tg
\end{aligned}
$$

2.2 Development

Grammar-based data compression was first proposed explicitly by Kieffer and Yang [15] and Nevill- Manning [24], but is closely related to some earlier ”macro-based” schemes proposed by Storer [29]. Several grammar-based compression algorithms have been proposed. A large detailed can be found in [20]. Nevill-Manning [24] devised the SEQUITUR algorithm which incrementally builds a grammar in a single pass through the input string. This procedure was subsequently improved by Kiefer and Yang [15] to what we refer to here as the SEQUENTIAL algorithm. The same authors employed a completely different approach to generating a compact grammar for a given string in their BISECTION algorithm. This procedure partitions the input into halves, then quarters, then eighths, etc. and creates a nonterminal in the grammar for each distinct substring generated in this way. Bisection was subsequently generalized to MPM [16] in order to exploit multi-way and incomplete partitioning. De Marcken [8] presented a complex multipass algorithm that emphasizes avoiding local minima. Apostolico and Lonardi [2] proposed a greedy algorithm n which rules are added in a steepest-descent fashion. Finally, even though it predates the idea of grammar-based compression, the output of the well-known LZ-78 algorithm [31]. In 2003, Rytter proposed a new class of grammar called AVL-grammar [26], a straightforward extension of the classical AVL-tree. This algorithm achieves $mathcalO(n \log \Sigma)$ time complexity and $\mathcal{O}(\log n)$ ratio approximation of minimal grammar-based compression of a given string of length n over an alphabet Σ. Charikkar et al proposed α-balanced grammar, an $\mathcal{O}(\log(n/g*))$ approximation algorithm, where $g*$ is the size of the smallest grammar [6].

2.3 Some Issues related to Grammar-based Compression

2.3.1 Smallest Grammar Problem

In data compression and the theory of formal languages, the smallest grammar problem is the problem of finding the smallest context-free grammar that generates a given string of characters. The size of a grammar is defined by some authors as the number of symbols on the right side of the production rules [20].

Lehman [20] showed that the smallest grammar problem is NP-Hard, using a reduction from a restricted form of vertex cover based closely on an argument by Storer [29]. They also posed that it is impossible to achieve an approximation ratio less than $\frac{8569}{8568}$ in polynomial time unless P=NP.

2.3.2 Straight-line grammar

A straight-line grammar is a formal grammar that generates exactly one string. Consequently, it does not branch (every non-terminal has only one associated production rule) nor loop (if non-terminal A appears in a derivation of B, then B does not appear in a derivation of A).

A context-free grammar $\mathcal{G}$ is an SLG if:

1. for every non-terminal N, there is at most one production rule that has N as its left-hand side, and
2. the graph $G =< V, E >$, defined by V being the set of non-terminals and $(A, B) \in E$ whenever B appears at the right-hand side of a production rule for A, is acyclic.

An SLG in Chomsky normal form is equivalent to a straight-line program.

2.4 Some well known Grammar-based Compression methods

2.4.1 The Sequential Algorithm

Nevill-Manning and Witten introduced the SEQUITUR grammar compression algorithm in [24]. Kieffer and Yang [15] subsequently offered an improved algorithm, which is reffered as Sequential. This works as follows:

Begin with an empty grammar, and make a single left-to-right pass through the input string. At each step, find the longest prefix of the unprocessed portion of the input that matches the expansion of a secondary nonterminal, and append that nontermnal to the start rule. Otherwise, if no prefix matches the expansion of a secondary nonterminal, append the first terminal in the unprocessed portion to the start rule. In either case, if the last pair of symbols in the start rule already occurs at some non-overlapping position in the grammar, then replace both occurrences by a new nonterminal whose definition is that pair. Finally, if some nonterminal is used only once after this substitution, then replace it by its de nition, and delete the corresponding rule.

For example, consider the input string $T = agaacaaaataaaaaaaa$. We start with an empty start rule and go on adding characters as terminals till a non-overlapping repeatition is found. For the first six passes, no rules are added. At $S \to agaaca$, the next character is a. Then a rule substitution is made with $R_1 \to aa$. So the grammar becomes,

$$\begin{aligned} S &\to agR_1cR_1aataaaaaaaa \\ R_1 &\to aa \end{aligned}$$

For the next passes, every aa is consumed as R_1 till the start rule becomes $S \to agR_1cR_1R_1tR_1R_1aaaa$. Algorithm finds a repeat of R_1R_1 and adds a new rule $R_2 \to R_1R_1$. Eventually it generates the grammar

$$\begin{aligned} S &\to agR_1cR_2tR_2R_2 \\ R_2 &\to R_1R_1 \\ R_1 &\to aa \end{aligned}$$

2.4.2 The Bisection Algorithm

Kiefer and Yang introduced the BISECTION algorithm [15]. this procedure works on an input string T as follows.

Select the largest integer k such that $|T| > 2^k$. Partition T into two substrings with lengths 2^k and $|T| - 2^k$. Repeat this partitioning process recursively on each substring of length greater than one. Create a nonterminal for every distinct string of length greater than one generated during this process. Each such nonterminal can then be de ned by a rule with exactly two symbols on the right.

2.4.3 LZ78

The well-known LZ78 [14] algorithm can be regarded as a grammar-based compressor. The procedure works as follows. Begin with an empty grammar. Make a single left-to-right pass through the input string. At each step, nd the shortest pre x of the unprocessed portion that is not the expansion of a secondary nonterminal. This prefix is either a single terminal a or else expressible as Xa where X is an existing nonterminal and a is a terminal. Define a new nonterminal, either $Y \to a$ or $Y \to Xa$, and append this new nonterminal to the end of the start rule.

For example, on input $aababababbababababbabbbbb$, the grammar generated by LZ78 is

$$\begin{aligned}
S &\to X_1X_2X_3X_4X_5X_6X_7X_8X_9 \\
X_1 &\to a \\
X_2 &\to X_1b \\
X_3 &\to X_2a \\
X_4 &\to b \\
X_5 &\to X_4a \\
X_6 &\to X_5b \\
X_7 &\to X_2b \\
X_8 &\to X_7b \\
X_9 &\to X_4b
\end{aligned}$$

Chapter 3

Algorithm DC3

3.1 Basic Idea

As stated before, a linear time construction of suffix arrays without using suffix trees was a long unsolved open problem, though both the construction of a suffix tree from a given input and construction of suffix array by traversing a suffix tree were O(n)-time algorithms. The basic problem was that no known comparison-based sorting can sort in a linear time. This shortcoming of comparison-based sorting was excellently overcome in the Difference Cover Modulo-3[14]. DC3 uses Divide-and-Conquire approach. It partitions the suffix array into two sets, one with suffix entry divisible by zero, and the other one with the triplets constructed by suffix values $1 mod 3$ and $2 mod 3$. Then these two sets are sorted in linear time using radix sort. Sorted partitions are merged in linear time to construct the final suffix array.

Along with suffix construction, an array *charCount*[] is also constructed which holds the number of occurence of each character in the string. As this is a DNA string, *charCount*[] is has 4 cells.

3.2 Algorithm

Input string $T[0, n]$= *"acatgtgaca\$"* is used to ilustrate DC3. The desired output is $SA[0, n] = \{10, 9, 7, 0, 2, 8, 1, 6, 4, 5, 3\}$.Input string is converted into integer array using lexicographic ordering.

For example, the given string T becomes $[1, 2, 1, 4, 3, 4, 3, 1, 2, 1, 0, 0, 0]$. Ending two 0's are added make the total length divisible by 3.

step 1: Construct a sample
For k = 0, 1, 2, define

$$B_k = \{i \in [0, n] | i mod 3 = k\}$$

Let C = $B_1 \cup B_2$ be the set of sample positions and S_C the set of sample suffixes.

step 2: Sort sample suffixes
For k = 1, 2, construct the strings

$$R_k = [t_k t_{k+1} t_{k+2}][t_{k+3} t_{k+4} t_{k+5}] \cdots [t_{maxB_k} t_{maxB_k+1} t_{maxB_k+2}]$$

whose characters are triplets $[t_k t_{k+1} t_{k+2}]$. In this example,

$R_1 = \{[214][343][121][000]\}$
$R_2 = \{[143][431][210]\}$

Let $R = R_1 \odot R_2$ be the concatenation of R_1 and R_2. For this example,
$R = \{[214][343][121][000][143][431][210]\}$

To sort sample suffixes, first radix sort R and then rename each entry of R by their rank to generate new array R'. For our example,

$R' = \{5, 6, 2, 1, 3, 7, 4\}$

If all the entries of R' are not distinct, then DC3 is used recursively to sort R'; else, the order in R' gives the order of suffixes. In our example, entries of R' are all distinct, therefore we can construct the suffix array with suffixes at 1-mod3 and 2-mod3 positions,

$SA_{12} = \{10, 7, 2, 8, 1, 4, 5\}$

Once the sample suffixes are sorted, assign a rank to each suffix. For $i \in C$, let $rank(S_i)$ denote the rank of S_i in the sample set S_C . Additionally, define $rank(S_{n+1}) = rank(S_{n+2}) = 0$. For $i \in B_0$, $rank(S_i)$ is undefined.

$rank = \{\perp, 5, 3, \perp, 6, 7, \perp, 2, 4, \perp, 1, 0, 0, 0\}$

step 3: Sort non-sample suffixes
Represent each nonsample suffix $S_i \in S_{B_0}$ with the pair $(t_i, rank(S_{i+1}))$. Note that $rank(S_{i+1})$ is always defined for $i \in B_0$. Clearly we have, for all $i, j \in B_0$,

$$S_i \leq S_j \Leftrightarrow (t_i, rank(S_{i+1})) \leq (t_j, rank(S_{j+1}))$$

The pairs $(t_i, rank(S_{i+1}))$ are then radix sorted.In our example, order of the non-sample suffixes become

$$S_9 < S_0 < S_6 < S_3$$

step 4: Merge
The two sorted sets of suffixes are merged using a standard comparison-based merging. To compare suffix $S_i \in S_C$ with $S_j \in S_{B_0}$, two cases are distinguished,

$$i \in B_1 : S_i \leq S_j \leftrightarrow (t_i, rank(S_{i+1})) \leq (t_j, rank(S_{j+1}))$$
$$i \in B_2 : S_i \leq S_j \leftrightarrow (t_i, t_{i+1}, rank(S_{i+2})) \leq (t_j, t_{j+1}, rank(S_{j+2}))$$

3.3 Time Complexity

Whole DC3 algorithm can be simplified as recursively dividing input into lengths one-third and two-third, sorting them and then merge two sorted strings.

Sorting the sampled saffixes require time $\mathcal{O}(n)$, as radix sort is being used here on integer array each with three digits. Merging two sorted sub-arrays require linear time too. Thus the time complexity of this phase, i.e., construction of suffix array using DC3 algorithm uses $\mathcal{O}(n)$ time.

Chapter 4

Compressed Suffix Array Construction

4.1 Basic Idea

After DC3 constructing the suffix array from the input string, we need to construct a succint representation of this suffix array; for which $\Psi(i)$ for each suffix $SA[i]$ need to be evaluated. Given the definition of $\Psi(i)$, what we have to do is to find the position of the suffix $SA[i]+1$ for each suffix $SA[i]$. But searching the whole suffix array for each suffix to compute $\Psi(i)$, using $\mathcal{O}(n^2)$ does not look like a good approach. So we have to device an algorithm that will search the next suffix for each suffix in sub-linear time.

Also, we have to device a way to compute any $SA[i]$ back from $\Psi(i)$ in sublinear cost. For that, a sampled version of the original SA is stored. The basic idea behind such sampling was discussed by Navarro and Gog [12]; a simpler modification has been used here.

4.2 Algorithm to compute Ψ

From the partial sortedness property of Ψ-function, we must note that for any suffix $SA[i]$, $SA[i]+1$ can be found within the position range specified by the 2nd character in the suffix, thereby shortening the area to be searched by an order equal to alphabet size.

Further, to search for a key in an array in sublinear time, binary search can be used. But binary search can be done only when keys are in sorted order. So a radix sort is used.

Combining these two features, we get the final algorithm to compute $\Psi(i)$ for each $SA[i]$ as follows,

step 1:

Partition the SA into character-groups using $charCount[]$ constructed in the previous phase. Each group now contains the suffixes starting with same character.

step 2:

Radix Sort each of the groups, with key to sort being the SA entry.

step 3:

For every $i \in [1, 2, .., n]$, search for $SA[i] + 1$ in the group marked by character T[i+1] using binary search. Store the index of the searched key as the i-th entry to the compressed suffix array.

4.3 Algorithm to construct a sampled SA

From the definition of $\Psi(i)$,

$$\begin{aligned} SA[\Psi(i)] &= SA[i] + 1 \\ \Rightarrow SA[i] &= SA[\Psi(i)] - 1 \\ &= SA[\Psi^2(i)] - 2 \\ &= SA[\Psi^3(i)] - 3 \\ &= SA[\Psi^j(i)] - j \end{aligned}$$

That is, given the suffix array entry of position i, we need to make j hops to find the suffix at position $\Psi_{-j}(i)$. Using these phenomenon, we can store the SA entries after equal hops, say h, so that it would take a maximum h hops to calculate each $SA[i]$ back from the compressed suffix array. To know which positions of the orginal suffix array are sampled, we store a bit vector of length equal to suffix array, and set each bit for which the corresponding $SA[i]$ is stored in the sampled suffix array. For later usage, the position corresponding to suffix entry 0, that is, first character of the string, is sampled and stored as *inZero*. The maximum hop is allowed to be $\log n$.

```
Input: compressedSuffixArray[n], SuffixArray[n]
Output sampledBitVector[n]
hop ←0;
while(!sampledBitVector[i]){
i ←compresssedSuffixArray[i];
hop++;
} if(hop>=maxHop)
set sampledBitVector[i];
```

Algorithm to compute sampling bit vector

An array *sampleSA* of length equal to total no. of bits set in BitVector is created to store the sampled suffix values. Suffix $SA[i]$ is stored in $sampleSA[j]$ where j is the rank of i in BitVector. This definition of *rank* of an entry in a bit vector is one used conventionally, i.e.,

$$rank(i, B) = \text{total no. of bits set in vector } B \text{ before } i$$

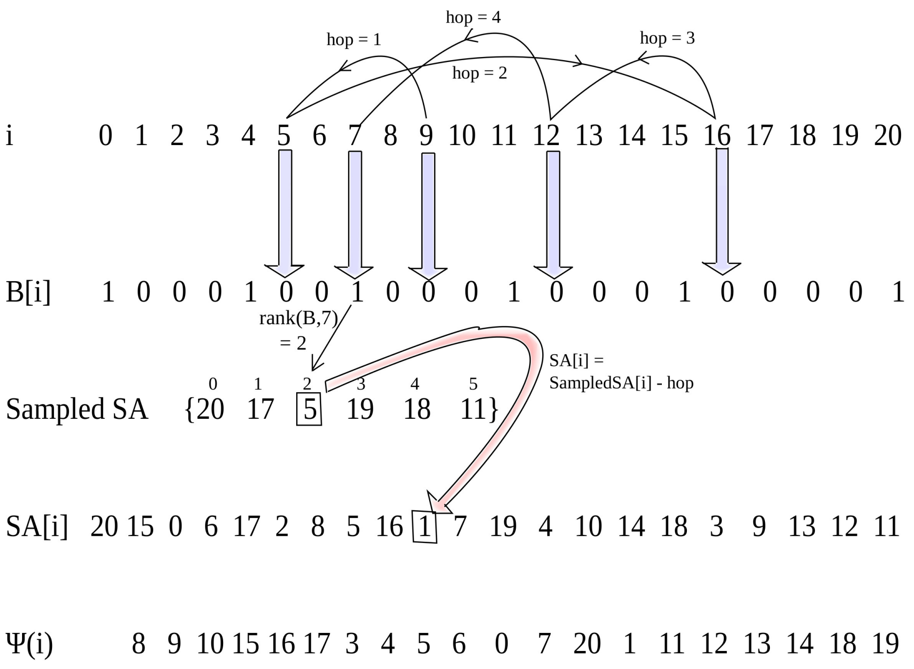

hop = 4
hop = 1
hop = 3
hop = 2
i 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
B[i] 1 0 0 0 1 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0 1
rank(B,7) = 2
SA[i] = SampledSA[i] - hop
0 1 2 3 4 5
Sampled SA {20 17 5 19 18 11}
SA[i] 20 15 0 6 17 2 8 5 16 1 7 19 4 10 14 18 3 9 13 12 11
Ψ(i) 8 9 10 15 16 17 3 4 5 6 0 7 20 1 11 12 13 14 18 19

Figure 4.1: Computation of $SA[9]$ of the string *"acagtcacagtttttacagt$"* using compressed suffix array along with sampled suffix entries; each *hop* is done from i to $\Psi(i)$

4.4 Time Complexity

In algorithm discussed in **Section 3.2**, each radix sort uses a total no. of passes equal to the no. of digits of maximum entry in a character group, which is $\mathcal{O}(\log n)$. So the time taken to compute the compressed suffix array is $\mathcal{O}(n \log n)$.

To compute *sampleSA*, each iteration of the **for**-loop executes a maximum of *maxHop* iterations of the **while**-loop. As maxHop = O(*log*n), the time complexity of algorithm given in **3.3** becomes O(n*log*n).

To return any suffix SA[i] using these data structures, a constant time access to *sampleSA* is preceeded by a O(*log*n) access to the compressed suffix array.

Chapter 5

LCP computation from Ψ-function

5.1 Algorithm

With the compressed suffix array at hand, we now proceed to compute the lcp values of each suffix $SA[i]$ using $\Psi(i)$. Remembering the definition of $\Psi(i)$, the Ψ-value of suffix $SA[i]$ points to the index of $SA[i]+1$ in SA. Also, all these $\Psi(i)$ values are in a sorted order for each character c of the alphabet. This means, given the $\Psi(i)$ for $SA[i]$ starting with character $T[i]$, we can tell exactly what is $T[i+1]$.

From the example shown in Figure 5.1 , $SA[2]$ and $SA[3]$ both beongs to the set of suffices starting with character *"a"*, which can be checked from the array *charCount*[]. We do every hop by setting the next indices to the previous one's Ψ-value. Here, after the first hop, we land at indices $\Psi(2) = 5$ and $\Psi(3) = 6$, both being in the sorted suffix-set starting with character *"c"*. This means both these suffixes have same 2nd character as well as 1st, so we continue with our next hop. This procedure goes on repeatitively untill the two indices found belongs to different character-group.

The algorithm to find lcp-value of the i-th suffix is given below:

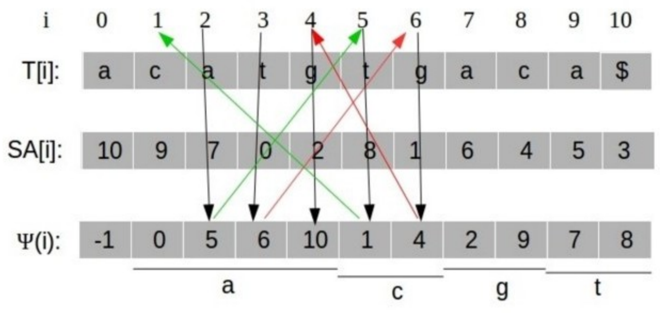


Figure 5.1: Showing the hops made from each suffix to its next one, using compressed suffix array; green arrows showing the path of hops for $SA[2]$, while the red one showing the same for $SA[3]$; black arrows points from index to the respective compressed suffix array entry

```
*Input*:compressedSuffixArray[n], characterCount[4];
*Output*:Lcp[i];
hop ← 0;
index ← i;
nextIndex ←i + 1;
**while**(index and nextIndex belong to same character group){
index ← compressedSuffixArray[index];
nextIndex ←compressedSuffixArray[nextIndex];
hop ←hop + 1;
}
return hop;
```

5.2 Time complexity

For each $i \in [1, 2, .., n]$,the given algorithm repeates upto hop times, which is, the lcp value of suffix $SA[i]$. Therefore, the time complexity for input size n becomes $\mathcal{O}(nl)$, where l is the maximum lcp value possible. Worst case scenario occurs when the string is a continuous repeat of the same character, i.e., *"aaaaaa..."*.In that case,

$$lcp[i] = n - i;$$

which means, total hops performed $= n+(n-1)+(n-2)+....+2+1 = n^2$,
and the complexity becomes $\mathcal{O}(n^2)$.
But in practical DNA sequences, this is impossible that a whole string is just a continuous repeat of the same nitrogen base. Moreover, it is rare to find a repeated sequence of a single base as a substring of length > 50. Thus in most cases we can expect this algorithm to work in linear time.

Chapter 6

Construction of a Greedy Grammar

6.1 Basic Idea

Having computed the compressed suffix array and the lcp array, we now proceed to generate a context free grammar that will uniquely expand to the input string. Our basic approach is the same as [3]: start with the whole string as start rule and substitute each non-overlapping repeating substring as a new rule to the grammar. But the choice of which non-overlapping substring to substitute and how this substring is reported is the basic problem.

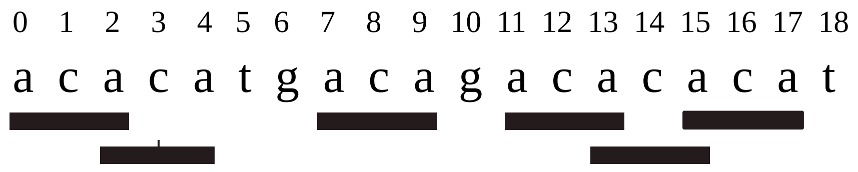


Figure 6.1: All the overlapping and non-overlapping repeatitions of *"aca"* in *"acacatgacagacacacat"*

6.1.1 Repeat: Overlapping and Non-overlapping

Let $T[0..n]$ be a string with suffixes $SA[i]$ and $SA[j]$ having $lcp(i,j) = k$. Clearly, the two substrings $\{T[i, i+1, .., i+k-1]\}$ and $\{T[j, j+1, .., j+k-1]\}$ are same. Here we find a repeat. If there are r suffixes $SA[i_0], SA[i_1], \cdots, SA[i_{r-1}]$ such that $lcp(i_s, i_t) \geq k$ for all

$i_s, i_t \in [i_0, i_{r-1}]$ then the substring $T[i_0 \cdots i_{k-1}]$ has r repeated occurences in T.

But not all repeats are non-overlapping. As we can see in Figure 6.1, *"aca"* has a total six occurences, but only four of them are non-overlapping. To build a grammar, we are interested in non-overlapping repeats only.

6.1.2 Longest repeated vs. Maximum repeated

In a given string, the substring which has maximum no. of repeats may not be the same as the longest substring repeated. In most of the case, the substring with maximum repeats is the shortest one. Moreover, if we think in terms of total area covered by repeatition, that can vary for different strings.

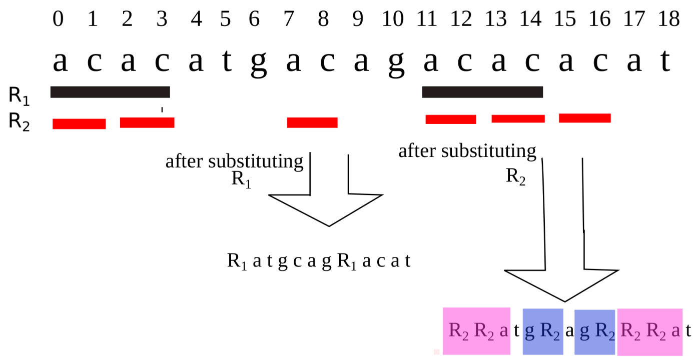

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
a c a c a t g a c a g a c a c a c a t
R_1
R_2
after substituting R_1
after substituting R_2
R_1 a t g c a g R_1 a c a t
R_2 R_2 a t g R_2 a g R_2 R_2 R_2 a t

Figure 6.2: Black strips showing the longest repeated *"acac"* and red strips showing maximum repeated *"ac"* in *"acacatgacagacacacat"* along with the rules after substitution. Colored region shows repeatition in substituted text

In Figure 6.2, area covered by *"ac"* is larger than that of *"acac"*. Then the choice of which substring to substitute as a rule at this stage becomes complex. At the first look it may seem the rule that substitutes *"ac"* should precede over *"acac"* to achieve a smaller grammar. But if we look closely, a different scenario appears. A rule substitution for *"ac"* leads to a start rule having repeatition, as can be seen in Figure 6.2, which can be removed by only a hieararchial grammar and that would increase the grammar size after all. Also, the second approach faciliates faster pattern matching in grammar compressed texts as patterns are more likely to be a substring of a rule substituting longer substrings.

6.2 Algorithm

6.2.1 Data Structures

Three data structures are used in this algorithm,

substringList: This array of structures stores the lcp values of suffices in order of their appearence in the input string, along with their positions. Refer to Table 1.2 and Table 1.1, the array *substringList* for input string ”acatgtgaca” is
{[3,3,0],[6,2,0],[4,1,0],[10,2,0],[8,1,0],[9,0,0],[7,0,0],[2,1,0],[5,0,0],[1,0,0]}
where entries are [*pos, lcp, tag*].

ruleList: This array of structures stores each rule generated along with the substring it replaces. Also the length of that substring is stored.

rIndex: This array of integers marks the positions of the string which each rule substitutes. It has same length as input string. If *rule i* substitutes substring *T*[*j, j+k-1*], then *rIndex*[*j*] = *i* and all other positions within that substring are set to -1. Positions not replaced by any rule are marked with -2.

Apart from these three data structures, two bit vectors are used to count for the positions replaced by the rules and to test overlapping of two substrings.

6.2.2 Generating Rules

Rules are generated in longest first order. So we need to start from the maximum lcp value *maxLcp*. *substringList* is traversed from the start for each lcp value. This traversal is done only for lcp less than half of total length, because any repeating substring longer than half of input string is bound to be overlapping.
Set j = length of repeating substring to search;

Step 1: Identify non-overlapping repeat

As *substringList* stores the suffixes in order to their position in text, given *substringList*[i], we search for the next suffix (in suffix array order) in the j-neighbourhood of *substringList*[i]. A variable *offset* is set to 0; If not found, we can say that the common prefixes of *substringList*[i].pos and *substringList*[i].pos-1+*offset* are non-overlapping; else we continue searching with incrementing *offset* till lcp is less than j;

Example is shown using string *"acatgtgaca"*. Here $maxLcp = 3$; we start with $j = 3$ and $i = 0$;

SA[*substringList*[0].pos-1] = SA[3-1] = 7; as no entry in *substringList* in the 3-neighbourhood of *substringList*[0] has *substringList*[i].pos = 2, suffices SA[0] and SA[7] have non-overlapping common prefix of length 3, which is *"aca"*. Now we check for the bit vector B_2 marking substituted positions, which is currently all set to 0. So we have two substitutions to do, at positions $T[0]$ and $T[7]$ of length 3. Bit vector B_2 is marked accordingly.

With the above step, substrings which appear at any two consecutive suffices in the suffix array are reported. But this is done when we start at a position having suffix that is lexicographically higher than some other existing suffix with lcp $\geq$j. That is, suffixes are reported from right to left of the suffix array. If any suffix with lcp $\geq$j resides to the right of the start point, it will not be reported in this pass; and subsequently may not be reported ever. That is why after checking all the suffixes on the left side of the start point (i.e., by decreasing *substringList*[i].pos) we need to check those on the right. This is simply done by incrementing *pos* and checking the overlap as before.

Step 2: Insert rule

When a repeating substring is found, it can be a new substring found or it can be another repeatition of an already substituted substring. Rule substituting these substrings must be placed according to these two conditions. We keep a variable *rMax* to store existing no. of rules. Whenever a new rule is added, *rMax* is incremented. Now, after finding a pair of non-overlapping repeating substrings with start position *pos1* and *pos2*, we check if either of *rIndex*[*pos1*] or *rIndex*[*pos1*] have an existing rule (in that case, either of the entries will be positive). If one of them have an existing rule marked, *rIndex*[*pos1*](or, *rIndex*[*pos1*]) is marked with that rule. If no existing rule is found, we increment *rMax* by 1 and mark the positions with this new rule.

These two steps are repeated while j is decremented, untill B_2 is full (which means all of the text has been substituted by substrings) or j=1 (because we need substrings with atleast two characters).

6.2.3 Generating grammar from rules

After the previous phase is over, array *rIndex* is marked with all the rules. Now we proceed to generate a grammar from *rIndex*.

Step 1: Initialize *ruleList* with rules
rMax gives the total no. of non-start rules; so first *ruleList* is created with size *rMax*. Next, *rIndex* is scanned;

```
for each positive entry rIndex[i],
    set ruleList[i].rule = rIndex[i];
    set ruleList[i].ruleIndex = i;
    set ruleList[i].strLen = no. of trailing -1's in rIndex + 1;
```

Step 2: Sort *ruleList* and eliminate repeats
As stated in Step 1. of 5.2.2, Our algorithm reports suffixes with common substring in a right-to-left fashion. All the occurences of a substring is reported untill an already marked section is found. But if there lies any occurence of that substring within the marked area(inside some previously reported longer substring) which lies left to the working position in suffix array, and some other occurence of the same substring within some suffix residing left to the already marked suffix, they are not reported. So these substrings appears again in *rIndex*, but with different rule. To eliminate such repeating rules, we need to sort the rules and identify repeats.

To sort *ruleList*, rules are groupped with same substring-length, *strLen*. Each group is radix sorted. Next, a repeatings are eliminated and a rank array *indexRule* is created to hold the ruleIndices of the unsorted *ruleList* as follows:

```
for each i do {
    set indexRule[ruleList[i].ruleIndex] = i;
    increment rI;
    if ruleList[i] and ruleList[i-1] contains same substring {
    set ruleList[i].ruleIndex = ruleList[i-1].ruleIndex;
    decrement rI;
    }
    ruleList[i].ruleIndex = rI;
}
```

Array *indexRule* at position i stores the position of *ruleList* which needs to be accessed for a rule i in *rIndex*; that is, *indexRule* works as a indirect index of *ruleList*.

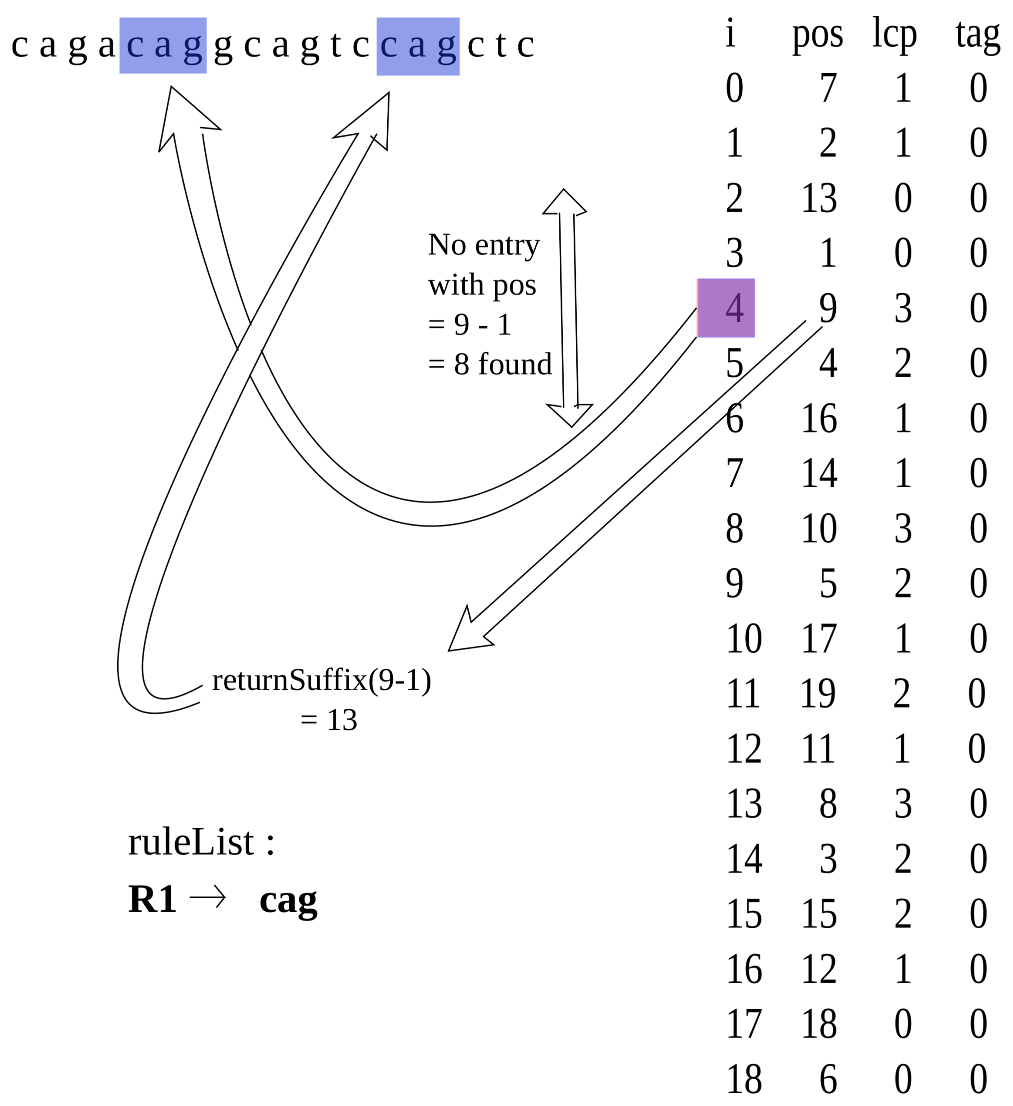
Step 3: Generate final grammar

After having all the unique rules in *ruleList*, we proceed to generate the start rule. This is a straight forward single pass over *rIndex*; for each i ∈[0,..,n]

if *rIndex*$[i]$ is positive, set $T[i]$ to the corresponding rule in *ruleList*, using *indexRule*
if *rIndex*$[i]$ is -1, discard $T[i]$
Now the string is converted to the start rule, with *ruleList* storing the remaining rules.
The grammar generated for the string *"tatcgaaaggttgtccacattgggaagtaacttgg"* by the given algorithm is,

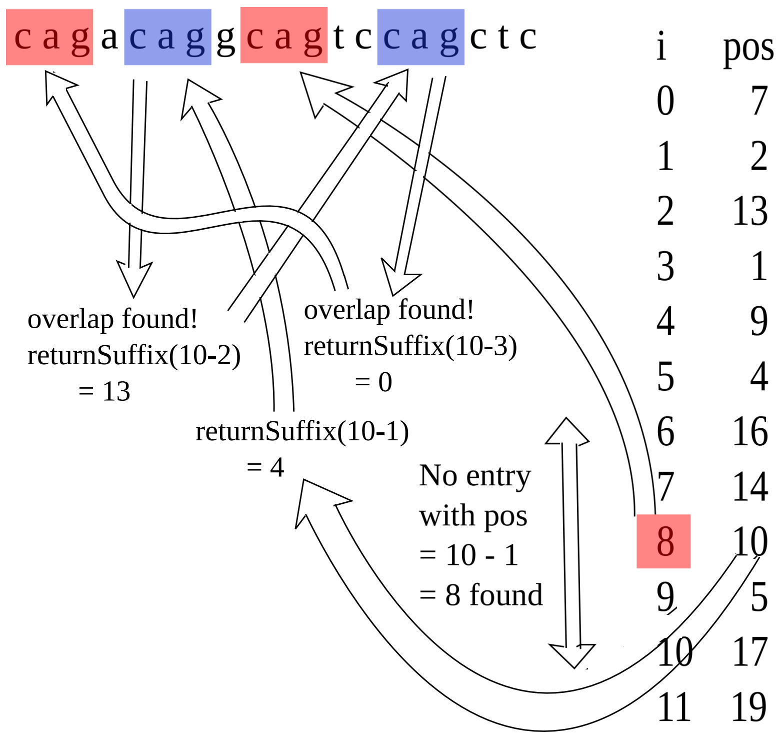
$$\begin{aligned}
R_0 &\Rightarrow R_4R_5R_2aggttgR_5R_3R_3R_1R_2gR_4acR_1\\
R_1 &\Rightarrow ttgg\\
R_2 &\Rightarrow gaa\\
R_3 &\Rightarrow ta\\
R_4 &\Rightarrow tc
\end{aligned}$$

In Figures 6.3, 6.4 and 6.5, rule generation for a string *"cagacaggcagtccagctc"* is shown. Clearly we can see in Figure 6.4, after checking if occurences of the substring *"cag"* in $SA[10]$ and $SA[9]$ are overlapping or not, our algorithm proceeds to substitute these two occurences with rule R2; but $SA[9]$ has already been used for the previous rule. So algorithm proceeds to check $SA[8]$ and $SA[7]$ and finally substitutes *"cag"* in $SA[7]$ along with $SA[10]$. We can see that both $SA[10]$ and $SA[9]$ is being used in a rule for the first time; so there is no way to check whether this substring has been used before to generate rules. Thus we get two rules in the grammar $R_1 \rightarrow cag$ and $R_1 \rightarrow cag$ both having same substring to substitute. Thats why we need to sort and eliminate duplicate rules after generating them.


c a g a c a g g c a g t c c a g c t c
No entry with pos = 9 - 1 = 8 found
returnSuffix(9-1) = 13
ruleList :
R1 → cag

i	pos	lcp	tag
0	7	1	0
1	2	1	0
2	13	0	0
3	1	0	0
4	9	3	0
5	4	2	0
6	16	1	0
7	14	1	0
8	10	3	0
9	5	2	0
10	17	1	0
11	19	2	0
12	11	1	0
13	8	3	0
14	3	2	0
15	15	2	0
16	12	1	0
17	18	0	0
18	6	0	0

Figure 6.3: 1st rule generation by the algorithm with $j = 3$ and $i = 4$ on string ***cagacaggcagtccagctc***



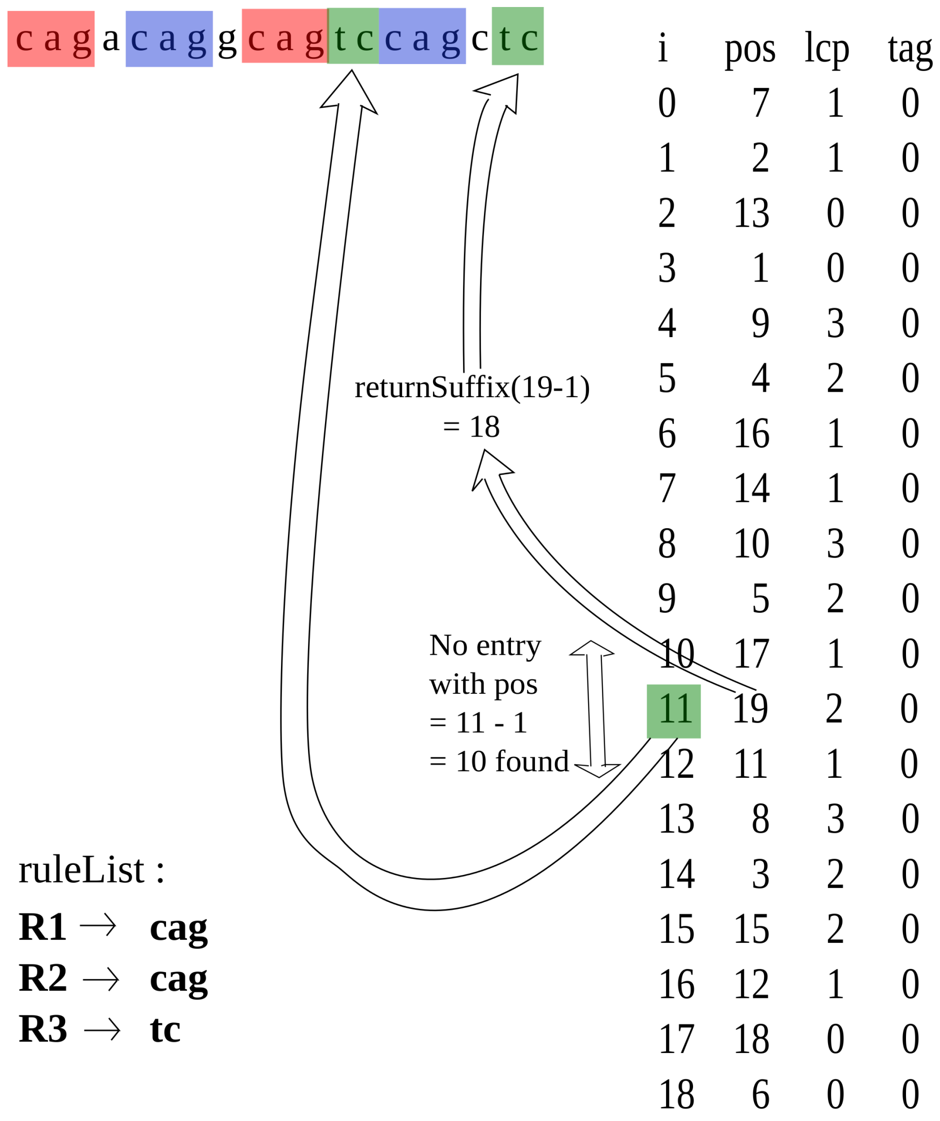
i	pos	lcp	tag
0	7	1	0
1	2	1	0
2	13	0	0
3	1	0	0
4	9	3	0
5	4	2	0
6	16	1	0
7	14	1	0
8	10	3	0
9	5	2	0
10	17	1	0
11	19	2	0
12	11	1	0
13	8	3	0
14	3	2	0
15	15	2	0
16	12	1	0
17	18	0	0
18	6	0	0

ruleList :

R1 $\rightarrow$ **cag**

R2 $\rightarrow$ **cag**

Figure 6.4: 2nd rule generation by the algorithm with $j = 3$ and $i = 8$ on string ***cagacaggcagtccagctc***



i	pos	lcp	tag
0	7	1	0
1	2	1	0
2	13	0	0
3	1	0	0
4	9	3	0
5	4	2	0
6	16	1	0
7	14	1	0
8	10	3	0
9	5	2	0
10	17	1	0
11	19	2	0
12	11	1	0
13	8	3	0
14	3	2	0
15	15	2	0
16	12	1	0
17	18	0	0
18	6	0	0

ruleList :

R1 $\rightarrow$ **cag**
R2 $\rightarrow$ **cag**
R3 $\rightarrow$ **tc**

Figure 6.5: 3rd rule generation by the algorithm with $j = 2$ and $i = 11$ on string *cagacaggcagtccagctc*

6.3 Time Complexity

Grammar generation is the costliest part of this algorithm in terms of time. First of all, we have check for substrings of length from maximum lcp to 2. For each of these iterations, data structure $subList[n]$ is traveresed fully. To substitute a certain substring as rule, we need to call the $returnSuffix()$ function which takes $\mathcal{O}(\log n)$ time to compute $SA[i]$ from $\Psi(i)$ and $sampledSA$. All of this makes our algorithm to use $\mathcal{O}(maxLcp * n \log n)$. Worst case appears when the string is of type *"aaaaaa.."*, i.e., a repeated sequence of a single base. Then maximum lcp becomes n, making the worst case complexity $\mathcal{O}(n^2 \log n)$.

But as stated in Section **4.2**, this case is almost impossible to happen in practical DNA sequences. So in most of the cases, $\mathcal{O}(maxLcp * n \log n)$ is the running time.

Chapter 7

Experimental Results

The proposed algorithm is implemented in C and run in an Intel^{R} Core^{TM} i3-2100 3.10 GHz CPU with 1333MHz memory speed using different DNA samples of yeast, rat and human from the Manzini DNA corpus. The overall performance is shown below: Through

Input Size(KB)	Execution time(s.)	Grammar size(KB)	No. of Rules
10	0.14	6.5	700
20	0.38	12.7	1288
30	0.83	19	1833
40	1.54	25	2367
50	2.52	31	2902
60	3.89	36	3422
120	23.54	71	5567
240	153.72	141	11443
302	155.89	178	14218
620	618.48	360	28166
730	1030.26	410	32776

Table 7.1: Input size, execution time, generated grammar size and no. of rules generated for different input data sets

out the results shown in this chapter, size of grammar has been set as total no. of symbols appearing at the right hand side of the rules; though an estimate of total bits needed to encoded will give the exact result, this coarse estimation used by Lehman[20] gives nonethless a good approximation.

From table 7.1, a graph is plotted showing compression ratio vs. input size in Figure 7.1.

In Figure 7.1, we can observe a steep rise in compression ratio

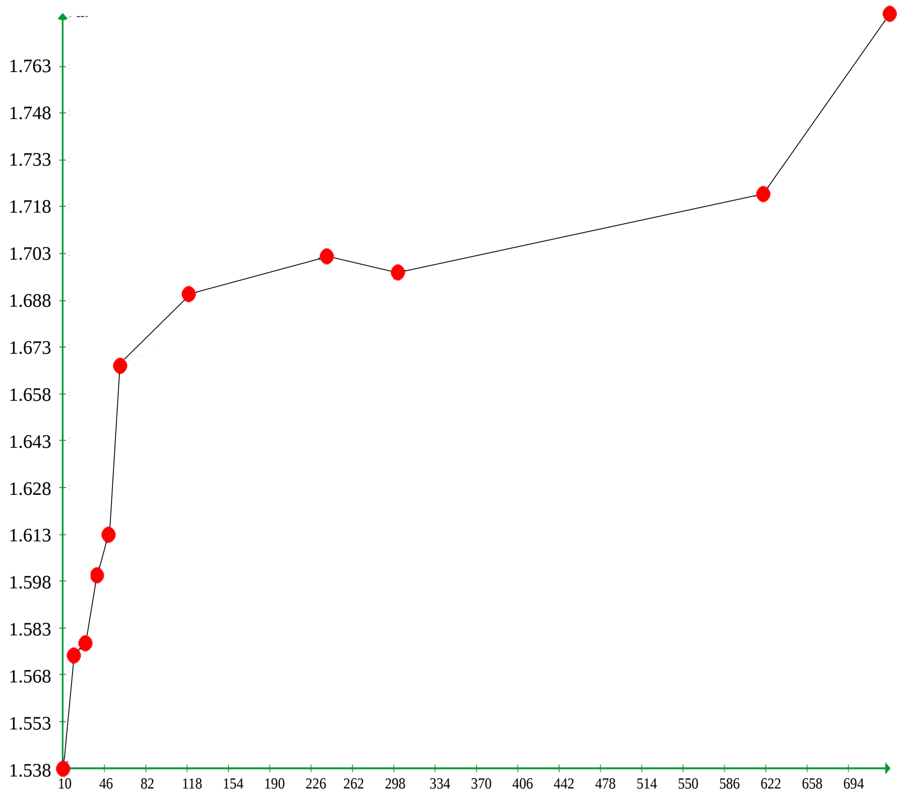


Figure 7.1: Graph showing compression ratio achieved vs. input size(in KB)

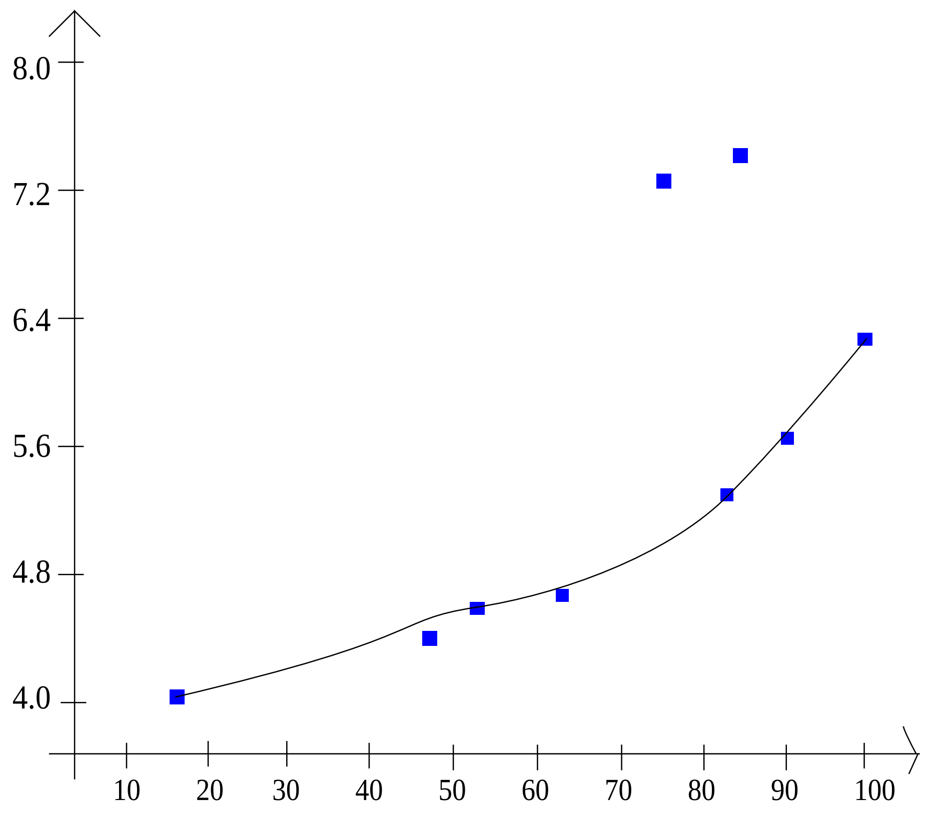


Figure 7.2: Graph showing execution time vs length of longest repeating substring; Two outliers are also shown

with input size 10KB to 60KB, and then a steady increment.

Next, in Figures 7.2, execution times for 9 different input files, each of size 60KB is shown, plotted against lengths of respective longest substrings.

Next two graphs are plotted using a single 302KB file. In Figure 7.3, different execution times for varying lengths of longest substring to substitute bounded by the algorithm are shown. In Figure 7.4 size of the grammar generated for the respective cases are shown.

In Figures 7.3 and 7.4, we can observe the drammatic effects on output if we bound our algorithm to choose a predefined longest sub-

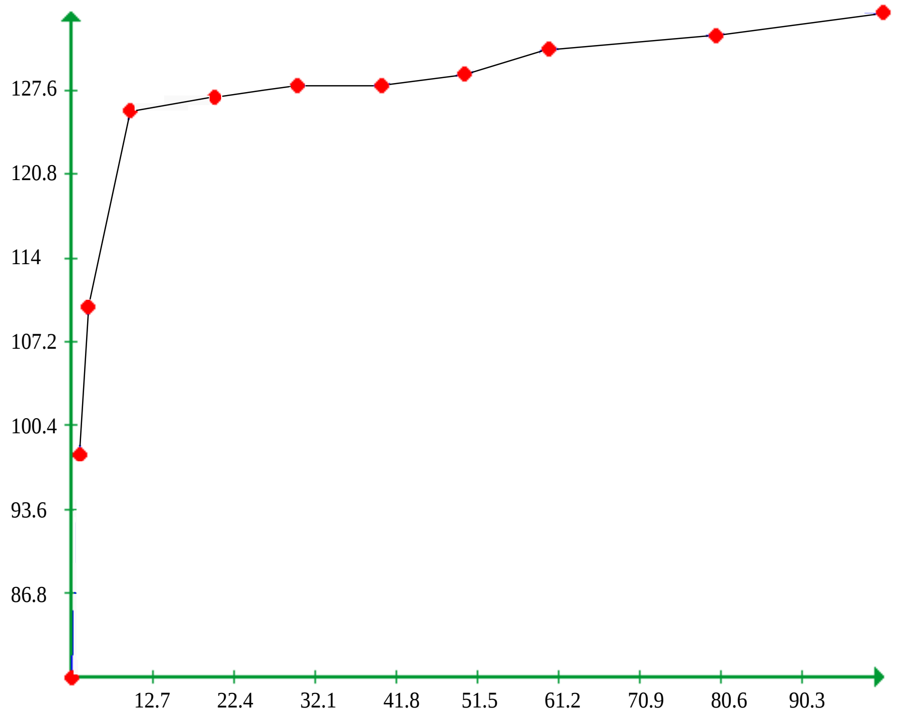


Figure 7.3: Graph showing different execution times for a single input file of size 302KB for different values of j

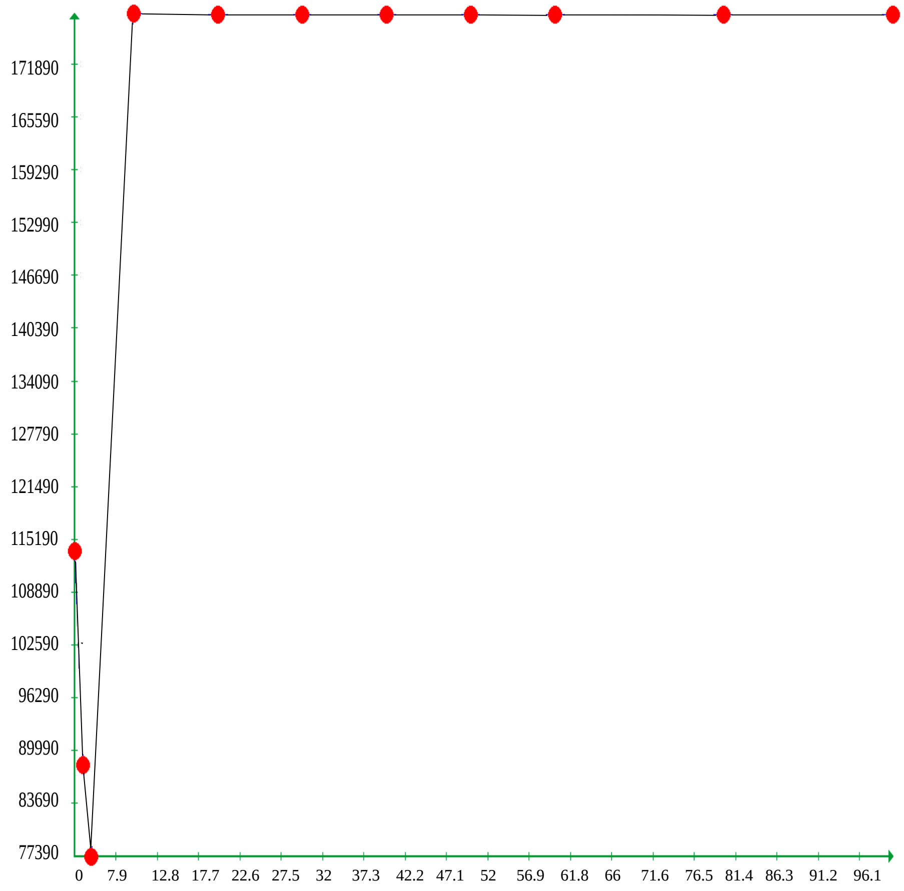


Figure 7.4: Graph showing generated grammar size for a single file of size 302KB for different values of j

string shorter than the actual longest repeating substring in input. The variable which defines the length of the substring the algorithm will look for was j (Section **5.2.2**). In Figure 7.3 We can see execution time varying linearly with $j = 3$ to $j = 10$, which is obvious from the algorithm. But then after $j = 10$ it settles to rise with a very slow rate, though linear. This can be inferred from the fact that most non-overlapping repeats are of length less than 10. This effect is shown in the graph of Figure 7.4 also. Here we can see a sharp fall in grammar size as we increase j, followed by a sharp rise and then a steady, almost constant grammar size can be seen.

Chapter 8

Disscussion and Scope of Improvement

From the algorithm analysis and experiment results observed, it can be seen that the proposed algorithm achieves compression ratio within 1.5 to 1.9, in the range of input data size used for experiments. But both in terms of compression and time complexity, SEQUITUR [24], RE-PAIR [19] or Bisection [16] supercedes. SEQUITUR generates a grammar of size 9KB with an input size 60KB in linear time, which is way too smaller than this algorithm. Only in terms of pattern searching in compressed data, our algorithm works better than all of three, at least theoretically. But there are ofcourse some fields of modification and variations that can be used in this algorithm, some of them discussed here.

8.1 Achieving smaller space complexity

In our algorithm, we generated a suffix array using DC3 from input string and then generated the compressed suffix array. Although compressed suffix arrays were designed for a space usage $\mathcal{O}(\frac{n}{\log n})$, this approach uses $\mathcal{O}(n)$ working space due to the construction of the uncompressed suffix array. This drawback can be overcome if we construct a compressed suffix array directly from the input text. In [18], an algorithm was proposed which constructs a compressed suffix array by reading characters from the end of string, without generating an intermediate suffix array. A certain variation of that algorithm does so in $\mathcal{O}(n \log n)$ time, using a Red-Black search tree

[18]. This, along with a succint representation of lcp-information generated can be helpfull to achieve a lower space complexity.

8.2 Choice of substring to substitute

From the graphs in Figures 7.3 and 7.4, we can see that our greedy approach of choosing the longest substring fast to substitute may be right with substring length <10, but after that it have a worsening effect. As we can see, most of the frequent substrings are of length below 10. Substrings longer than 10 repeat very rarely. So with our approach to start substituting the longest one first, we diminish the possibility of a better compression as shorter substrings would substitute more occurences, without adding a longer rule and increasing size of the grammar.

We can locate the range of the longest substrings with a sufficient number of repeat from the lcp-interval table(Section 1.3) with a rough approximation. Then rather than using the overall longest substring, we can rank and select substrings those are longer and also repeatitive. A metric for such a ranking cam be total area covered by all the repeatitions of each substring along with a weightage upon length of substring.

8.3 Segment-wise grammar generation

DNA strings are highly repeatitive and this repeatitive nature is expressed all over the string. So, for a string of size 500KB, if we choose substring of length, say 12, to have 100 occurences in a segment of 30KB, it is likely to have more or less same no. of occurences in any other segment of 30KB. Then we can use that substring for the whole file without explicitly running the algorithm over it. We may have to just check for non-overlapping occurences and then substitute them by the rule already generated.

Originally our algorithm runs on the whole string and generates a compressed suffix array, computes the lcp-information and then proceeds to construct the grammar, last phase being the slowest. We can skip that work by simply generating rules for a small segment and use it all over the string. Ofcourse, this ‘small’ segment must be large enough to reflect the string statistics of the large DNA string in a miniature level.

8.4 Going towards hieararchy

Although we devised this algorithm to generate a two-level grammar only, i.e., where each rule other than the start rule have no non-terminal in right hand side, this algorithm can be modified to generate multi-level grammars also. In that case, we may not choose the longest substring first; rather we should go for substrings covering largest area with all their occurences. This will result in repeatitions in the start rule, which will be substituted by further rules. With this modification, we will have to construct suffix array and compressed suffix array using integer alphabet.

Although the motivation behind this two-level grammar generation algorithm was to faciliate faster search in grammar-compressed text and if we modify it to be multi-level, this advantage will surely decrease, stil there are chances of better performance as in this case, we are not abandoning our approach of searching for long repeatitions; it is just being restricted to the number of occurence as a parameter.

Chapter 9

Conclusion

After this study of grammar-based compressions and construction of a new algorithm, we can conclude that this field has a vivid work to do. In fact the question of smallest grammar is unclear in itself and givn the bound of approximation, there are lots of scope of development. Also, use of suffix tree, suffix array as well as compressed suffix array in grammar based compression is a totally new approach. Further developments in efficient and succint representation of suffix data can prove to be helpful for this field. The basic differences lying inside this approach is that a suffix tree gives a wholistic view of the string statistics of a data better than any data structure till date; therefore data compression with efficient search of relevant data patterns in a compressed data is far suitable with suffix tree (or its representatives such as suffix array, compressed suffix array, enhanced suffix array etc.), at least theoreticaly.

Bibliography

[1] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

[2] Alberto Apostolico and Stefano Lonardi. Some theory and practice of greedy off-line textual substitution. In *Data Compression Conference, 1998. DCC'98. Proceedings*, pages 119–128. IEEE, 1998.

[3] Alberto Apostolico and Stefano Lonardi. Off-line compression by greedy textual substitution. *Proceedings of the IEEE*, 88(11):1733–1744, 2000.

[4] Michael Burrows and David Wheeler. A block-sorting lossless data compression algorithm. In *DIGITAL SRC RESEARCH REPORT*. Citeseer, 1994.

[5] Minh Duc Cao, Trevor I Dix, Lloyd Allison, and Chris Mears. A simple statistical algorithm for biological sequence compression. In *Data Compression Conference, 2007. DCC'07*, pages 43–52. IEEE, 2007.

[6] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, April Rasala, Amit Sahai, et al. Approximating the smallest grammar: Kolmogorov complexity in natural models. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 792–801. ACM, 2002.

[7] Noam Chomsky. On certain formal properties of grammars. *Information and control*, 2(2):137–167, 1959.

[8] Carl De Marcken. The unsupervised acquisition of a lexicon from continuous speech. *arXiv preprint cmp-lg/9512002*, 1995.

[9] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 390–398. IEEE, 2000.

[10] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM (JACM)*, 52(4):552–581, 2005.

[11] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 3(2):20, 2007.

[12] Simon Gog and Gonzalo Navarro. Improved and extended locating functionality on compressed suffix arrays. In *Experimental Algorithms*, pages 436–447. Springer, 2014.

[13] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.

[14] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *Automata, Languages and Programming*, pages 943–955. Springer, 2003.

[15] John C Kieffer and En-Hui Yang. Grammar-based codes: a new class of universal lossless source codes. *Information Theory, IEEE Transactions on*, 46(3):737–754, 2000.

[16] John C Kieffer, En-Hui Yang, Gregory J Nelson, and Pamela Cosman. Universal lossless compression via multilevel pattern matching. *Information Theory, IEEE Transactions on*, 46(4):1227–1245, 2000.

[17] Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Linear-time construction of suffix arrays. In *Combinatorial pattern matching*, pages 186–199. Springer, 2003.

[18] Tak-Wah Lam, Kunihiko Sadakane, Wing-Kin Sung, and Siu-Ming Yiu. A space and time efficient algorithm for constructing compressed suffix arrays. In *Computing and Combinatorics*, pages 401–410. Springer, 2002.

[19] N Jesper Larsson and Alistair Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11):1722–1732, 2000.

[20] Eric Lehman and Abhi Shelat. Approximation algorithms for grammar-based compression. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 205–212. Society for Industrial and Applied Mathematics, 2002.

[21] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.

[22] Edward M McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM (JACM)*, 23(2):262–272, 1976.

[23] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys (CSUR)*, 39(1):2, 2007.

[24] Craig G. Nevill-Manning and Ian H. Witten. Identifying hierarchical strcture in sequences: A linear-time algorithm. *J. Artif. Intell. Res.(JAIR)*, 7:67–82, 1997.

[25] David R Powell, Lloyd Allison, and Trevor I Dix. Modelling-alignment for non-random sequences. In *AI 2004: Advances in Artificial Intelligence*, pages 203–214. Springer, 2004.

[26] Wojciech Rytter. Application of lempel–ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1):211–222, 2003.

[27] Claude Elwood Shannon. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):3–55, 2001.

[28] Michael Sipser. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston, 2006.

[29] James Andrew Storer. Data compression: methods and complexity issues. 1979.

[30] Peter Weiner. Linear pattern matching algorithms. In *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*, pages 1–11. IEEE, 1973.

[31] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *Information Theory, IEEE Transactions on*, 24(5):530–536, 1978.