Recompression : An Approximate Algorithm For Grammar-Based Compression

Thesis submitted to
The Faculty of Engineering & Technology, Jadavpur University
In partial fulfillment of the requirements for the Degree Of

Master of Computer Science & Engineering

In the Department of Computer Science & Engineering
By

Sourav Mitra

Exam Roll No. – M4CSE1624R
Class Roll No. – 200910502008
Registration No. – 91224 of 2004-2005

Under the esteemed Guidance of
Professor Shovonlal Kundu

Department of Computer Science & Engineering
Jadavpur University, Kolkata-700032
May, 2016

Recompression : An Approximate Algorithm For Grammar-Based Compression

Thesis submitted to the
Faculty of Engineering & Technology, Jadavpur University

In partial fulfillment of the requirements for the Degree Of

Master of Computer Science & Engineering

In the Department of Computer Science & Engineering
By

Sourav Mitra

Exam Roll No. – M4CSE1624R
Class Roll No. – 200910502008
Registration No. – 91224 of 2004 - 2005

Under the esteemed Guidance of
Professor Shovonlal Kundu

Department of Computer Science &Engineering ,
Jadavpur University ,Kolkata-700032

May, 2016

FACULTY OF ENGINEERING AND TECHNOLOGY

JADAVPUR UNIVERSITY

Certificate of Recommendation

This is to certify that the dissertation entitled "**Recompression : An Approximate Algorithm For Grammar-Based Compression**" has been carried out by**Sourav Mitra (Examination Roll No.M4CSE1624R,Class Roll No. 200910502008 and University Registration No.91224 of 2004-2005)** under my guidance and supervision and be accepted in partial fulfillment of the requirement for the Degree of Master of Computer Science and Engineeringfrom the Department of Computer Science &Engineeringin the Faculty of Engineering and Technology, Jadavpur University. The research results presented in the thesis have not been included in any other paper submitted for the award of any degree in any other University or Institute.

………………………………
(**ProfessorShovonlalKundu**)
Thesis Supervisor,
Department of Computer Science and Engineering,
Jadavpur University, Kolkata-700032 .

Countersigned:

………………………………..
(ProfessorDebesh Kumar Das)
Headof the Department,
Department of Computer Science and Engineering,
Jadavpur University, Kolkata-700032.

…………………………………………………….
(ProfessorSivajiBandyopadhyay)
Dean,
Faculty of Engineering and Technology,
Jadavpur University, Kolkata-32.

FACULTY OF ENGINEERING AND TECHNOLOGY

JADAVPUR UNIVERSITY

<u>Certificate of Approval</u>

This is to certify that the thesis entitled "**Recompression : An Approximate Algorithm For Grammar-Based Compression**" is a bona-fide record of work carried out by **Sourav Mitra (Examination Roll No.M4CSE1624R,Class Roll No. 200910502008 and University Registration No.91224 of 2004-2005)** in partial fulfillment of the requirements for the award of the degree of Master of Computer Science and Engineering in the Department of Computer Science and Engineering, Jadavpur University, Kolkata. It is understood that by this approval the undersigned do not necessarily endorse or approve any statement made, opinion expressed or conclusion drawn therein but approve the thesis only for the purpose for which it has been submitted.

……………………………

(Signature of Examiner 1)

Date:

……………………………

(Signature of Examiner 2)

Date:

FACULTY OF ENGINEERING AND TECHNOLOGY

JADAVPUR UNIVERSITY

Declaration of Originality and Compliance of Academic Ethics

I hereby declare that this thesis entitled "**Recompression : An Approximate Algorithm For Grammar-Based Compression**" contains literature survey and original research work by the undersigned candidate, as part of his Degree of Master of Computer Science & Engineering.

All information have been obtained and presented in accordance with academic rules and ethical conduct.

I also declare that, as required by these rules and conduct, I have fully cited and referenced all materials and results that are not original to this work.

Name : Sourav Mitra

Exam Roll No : M4CSE1624R
Class Roll No : 200910502008
Registration No : 91224 of 2004-2005

Thesis Title:Recompression : An Approximate Algorithm

For Grammar-Based Compression

…..………………………………………

(Signature with Date)

Acknowledgement

I would like to start by thanking the holy trinity for helping me deploy all the right resources and for shaping me into a better human being.

I would like toexpress my deepest gratitude to my advisor, **Prof.ShovonlalKundu**, Department of Computer Science and Engineering, Jadavpur University for his admirable guidance, care, patience and for providing me with an excellent atmosphere for doing research.Our numerous scientific discussions and his many constructive comments have greatly improved this work.

I would like to thank **Prof. Debesh Kumar Das**, Head of the Department, Department of Computer Science and Engineering, Jadavpur University, for providing me with moral support at times of need.

Most importantly none of this would have been possible without the love and support of my family. I extend my thanks to my parents, especially to my mother whose forbearance and whole hearted support helped this endeavor succeed.

This thesis would not have been completed without the inspiration and support of a number of wonderful individuals — my thanks and appreciation to all of them for being part of this journey and making this thesis possible.

……………………………………………..

Name : Sourav Mitra

Exam Roll No : M4CSE1624R
Class Roll No : 200910502008
Registration No : 91224 of 2004-2005

Department of Computer Science & Engineering

Jadavpur University

Contents

ABSTRACT

In this Thesis work, a simple linear-time algorithm constructing a context-free grammar of size $O(g\log(N/g))$ for the input string, where N is the size of the input string and g the size of the optimal grammar generating this string is presented. The algorithm works for arbitrary size alphabets, but the running time is linear assuming that the alphabet Σ of the input string can be identified with numbers from $\{1, \ldots, N^c\}$ for some constant c. Otherwise, additional cost of $O(N\log|\Sigma|)$ is needed.

Algorithms with such an approximation guarantee and running time are the particular simplicity of the algorithm as well as the analysis of the algorithm, which uses a general technique of *recompression* recently introduced by Artur Jez, Institute of Computer Science, University of Wrocław, Poland. Furthermore, contrary to the previous results, this work does not use the L Z representation of the input string in the construction or in the analysis.

CHAPTER : 1 INTRODUCTION

Data compression is the art of reducing the number of bits needed to store or transmit data. Compression can be either lossless or lossy. Losslessly compressed data can be decompressed to exactly its original value.

All data compression algorithms consist of at least a model and a coder (with optional preprocessing transforms). A model estimates the probability distribution. The coder assigns shorter codes to the more likely symbols. There are efficient and optimal solutions to the coding problem. However, optimal modeling has been proven not computable. Modeling (or equivalently, prediction) is both an artificial intelligence (AI) problem and an art.

There are two major categories of compression algorithms: lossy and lossless. Lossy compression algorithms involve the reduction of a file's size usually by removing small details that require a large amount of data to store at full fidelity. In lossy compression, it is impossible to restore the original file due to the removal of essential data. Lossy compression is most commonly used to store image and audio data, and while it can achieve very high compression ratios through data removal, it is not covered in this article. Lossless data compression is the size reduction of a file, such that a decompression function can restore the original file exactly with no loss of data. Lossless data compression is used ubiquitously in computing, from saving space on your personal computer to sending data over the web, communicating over a secure shell, or viewing a PNG or GIF image.

In the grammar-based compression text is represented by a context-free grammar (CFG) generating exactly one string. The idea behind this approach is that a CFG can compactly represent the structure of the text, even if this structure is not apparent. Furthermore, the natural hierarchical definition of the CFGs makessuch arepresentation suitable for algorithms, in which case the string operations can be performed on the compressed representation, without the need of the explicit de-compression[2,8,10,17,3,1]. Lastly, there is a close connection between block-based compression methods and the grammar compression: it is fairly easy to rewrite the LZW definition as an$O(1)$larger CFG, LZ77 can also be presented in this way, introducing a polynomial blow-up in size (reducing the blow up to $\log(N/l)$, where l is the size of the LZ77 representation, is non-trivial).

1.1 Application of grammar-based compression:

The goal of algorithms on compressed strings is to check properties of compressed strings and thereby beat a straight forward “decompress-and-check”strategy. There are three main applications for algorithms of this kind [13].

- ❖ In many areas, large string data have to be not only stored in compressed form, but the initialdata has to be processed and analyzed as well. Here, it makes sense to design algorithms thatdirectly operate on the compressed string representation in order to save the time and spacefor (de)compression. Such a scenario can be found for instance in large genom databases orXML processing.

- ❖ Large and often highly compressible strings may appear as intermediate data structures inalgorithms. Here, one may try to store a compressed representation of these intermediatedata structures and to process this representation. This may lead to more efficient algorithms.Examples for this strategy can be found for instance in combinatorial group theory[57, 58, 95, 97, 125], computational topology [37, 122, 124], interprocedural analysis [52],and bisimulation checking [61, 79].

- ❖ In some situations it makes sense to compute in a first phase a compressed representationof an input string, which makes regularities in the string explicit. These regularities maybe exploited in a second phase for speeding up an algorithm. This principle is knownas acceleration by compression. It was recently applied in order to speed up the Viterbialgorithm for analyzing hidden Markov models [83] as well as speeding up edit distancecomputation [31, 59].

1.2 Proposed Approach :

While grammar-based compression was introduced with practical purposes in mind and the paradigm was used in several implementations [12,11,16], it also turned out to be very useful in more theoretical considerations. Intuitively, in many cases large data have relatively simple inductive definition, which results in a grammar representation of a small size. On the other hand, it was already mentioned that the hierarchical structure of the CFGs allows operations directly on the compressed representation. A recent survey by Lohrey[13]gives a comprehensive description of several areas of theoretical computer science in which grammar-based compression was successfully applied.

The main drawback of the grammar-based compression is that producing the smallest CFG for a text is intractable: given a string w and number k it is NP-hard to decide whether there exists a CFG of size k that generates w [20]. Furthermore, the size of the grammar cannot be approximated with an approximation factor better than 8569/ 8568 , unless P=NP [1].

1.3 Recompression :

In this Thesis an algorithm is proposed, it is constructed using the general approach of *recompression*, developed by the A. Jez. In essence, we iteratively apply two replacement schemes to the text T :

pair compression of *ab* For two different symbols (i.e. letters or nonterminals) a, b such that substring ab occurs in Treplace each of ab in T by a fresh nonterminal c.

*a***'s block compression** For each maximal block a^l, where a is a letter or a nonterminal and $l > 1$, that occurs in T , replace all a^ls in T by a fresh nonterminal a_l.

Then the returned grammar is obtained by backtracking the compression operations performed by the algorithm: observe that replacing $ab$with c corresponds to a grammar production

$$c \rightarrow ab \tag{1a}$$

and similarly replacing $a\_$ with $a_$corresponds to a grammar production

$$a_l \rightarrow a^l \tag{1b}$$

The algorithm is divided into *phases*: in the beginning of a phase, all pairs occurring in the current text are listed and stored in a list P , similarly, L contains all letters occurring in the current text. Then pair compression is applied to an appropriately chosen subset of P and all blocks of symbols from L are compressed, then the phase ends. If everything works perfectly, each symbol of T is replaced and so T 's length drops by half ; in reality the text length drops by some smaller, but constant, factor per phase. For the sake of simplicity, we treat all nonterminals introduced by the algorithm as letters.

CHAPTER : 2
SURVEY OF RELATED WORKS

2.1 Compression Algorithms :

There are two major categories of compression algorithms: **lossy** and **lossless**. Lossy compression algorithms involve the reduction of a file's size usually by removing small details that require a large amount of data to store at full fidelity. Lossless data compression is the size reduction of a file, such that a decompression function can restore the original file exactly with no loss of data.

The basic principle that lossless compression algorithms work on is that any non-random file will contain duplicated information that can be condensed using statistical modeling techniques that determine the probability of a character or phrase appearing. These statistical models can then be used to generate codes for specific characters or phrases based on their probability of occurring, and assigning the shortest codes to the most common data. Such techniques include entropy encoding, run-length encoding, and compression using a dictionary. Using these techniques and others, an 8-bit character or a string of such characters could be represented with just a few bits resulting in a large amount of redundant data being removed.

2.2 The Journey of Lossless Compression Algorithms :

Data compression has only played a significant role in computing since the 1970s, when the Internet was becoming more popular and the Lempel-Ziv algorithms were invented, but it has a much longer history outside of computing. Morse code, invented in 1838, is the earliest instance of data compression in that the most common letters in the English language such as “e” and “t” are given shorter Morse codes. Later, as mainframe computers were starting to take hold in 1949, Claude Shannon and Robert Fano invented Shannon-Fano coding. Their algorithm assigns codes to symbols in a given block of data based on the probability of the symbol occurring. The probability is of a symbol occurring is inversely proportional to the length of the code, resulting in a shorter way to represent the data.

Two years later, David Huffman was studying information theory at MIT and had a class with Robert Fano. Fano gave the class the choice of writing a term paper or taking a final exam. Huffman chose the term paper, which was to be on finding the most efficient method of binary coding. After working for months and failing to come up with anything, Huffman was about to throw away all his work and start studying for the final exam in lieu of the paper. It was at that point that he had an epiphany, figuring out a very similar yet more efficient technique to Shannon-Fano coding. The key difference between Shannon-Fano coding and Huffman coding is that in the former the probability tree is built bottom-up, creating a suboptimal result, and in the latter it is built top-down.

Early implementations of Shannon-Fano and Huffman coding were done using hardware and hardcoded codes. It was not until the 1970s and the advent of the Internet and online storage that software compression was implemented that Huffman codes were dynamically generated based on the input data. Later, in 1977, Abraham Lempel and Jacob Ziv published their groundbreaking LZ77 algorithm, the first algorithm to use a dictionary to compress data. More specifically, LZ77 used a dynamic dictionary oftentimes called a sliding window. In 1978, the same duo published their LZ78 algorithm which also uses a dictionary; unlike LZ77, this algorithm parses the input data and generates a static dictionary rather than generating it dynamically.

2.3 Compression Techniques :

Many different techniques are used to compress data. Most of the compression techniques cannot stand on their own, but must be combined together to form a compression algorithm. Those that can stand alone are often more effective when joined together with other compression techniques. Most of these techniques fall under the category of entropy coders, but there are others such as Run-Length Encoding and the Burrows-Wheeler Transform that are also commonly used.

2.3.1 Run-Length Encoding :

Run-Length Encoding is a very simple compression technique that replaces runs of two or more of the same character with a number which represents the length of the run, followed by the original character; single characters are coded as runs of 1. RLE is useful for highly-redundant data, indexed images with many pixels of the same color in a row, or in combination with other compression techniques like the Burrows-Wheeler Transform.

Here is a quick example of RLE:

Input: AAABBCCCCDEEEEEEAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAA

Output: 3A2B4C1D6E38A

2.3.2 Burrows-Wheeler Transform :

The Burrows-Wheeler Transform is a compression technique invented in 1994 that aims to reversibly transform a block of input data such that the amount of runs of identical characters is maximized. The BWT itself does not perform any compression operations, it simply transforms the input such that it can be more efficiently coded by a Run-Length Encoder or other secondary compression technique.

The algorithm for a BWT is simple:

1. Create a string array.
2. Generate all possible rotations of the input string, storing each in the array.
3. Sort the array alphabetically.
4. Return the last column of the array.

BWT usually works best on long inputs with many alternating identical characters. Here is an example of the algorithm being run on an ideal input. Note that & is an End of File character:

Input	Rotations	Alpha-Sorted Rotations	Output
HAHAHA&	HAHAHA&	AHAHA&**H**	**HHH&AAA**
	&HAHAHA	AHA&HA**H**	
	A&HAHAH	A&HAHA**H**	
	HA&HAHA	HAHAHA**&**	
	AHA&HAH	HAHA&H**A**	
	HAHA&HA	HA&HAH**A**	
	AHAHA&H	&HAHAH**A**	

Because of its alternating identical characters, performing the BWT on this input generates an optimal result that another algorithm could further compress, such as RLE which would yield "3H&3A". While this example generated an optimal result, it does not generate optimal results on most real-world data.

2.3.3 Entropy Encoding :

Entropy in data compression means the smallest number of bits needed, on average, to represent a symbol or literal. A basic entropy coder combines a statistical model and a coder. The input file is parsed and used to generate a statistical model that consists of the probabilities of a given symbol appearing. Then, the coder will use the statistical model to determine what bit-or-bytecodes to assign to each symbol such that the most common symbols have the shortest codes and the least common symbols have the longest codes. It is also known as ***Statistical Encoding.***

2.3.3.1 Shannon-Fano Coding :

This is one of the earliest compression techniques, invented in 1949 by Claude Shannon and Robert Fano. This technique involves generating a binary tree to represent the probabilities of each symbol occurring. The symbols are ordered such that the most frequent symbols appear at the top of the tree and the least likely symbols appear at the bottom.

The code for a given symbol is obtained by searching for it in the Shannon-Fano tree, and appending to the code a value of 0 or 1 for each left or right branch taken, respectively. For example, if "A" is two branches to the left and one to the right its code would be "001_2". Shannon-Fano coding does not always produce optimal codes due to the way it builds the binary tree from the bottom up. For this reason, Huffman coding is used instead as it generates an optimal code for any given input.

The algorithm to generate Shannon-Fano codes is fairly simple :

1. Parse the input, counting the occurrence of each symbol.
2. Determine the probability of each symbol using the symbol count.
3. Sort the symbols by probability, with the most probable first.
4. Generate leaf nodes for each symbol.
5. Divide the list in two while keeping the probability of the left branch roughly equal to those on the right branch.
6. Prepend 0 and 1 to the left and right nodes' codes, respectively.
7. Recursively apply steps 5 and 6 to the left and right sub-trees until each node is a leaf in the tree.

2.3.3.2 Huffman Coding :

Huffman Coding is another variant of entropy coding that works in a very similar manner to Shannon-Fano Coding, but the binary tree is built from the top down to generate an optimal result.

The algorithm to generate Huffman codes shares its first steps with Shannon-Fano:

1. Parse the input, counting the occurrence of each symbol.
2. Determine the probability of each symbol using the symbol count.
3. Sort the symbols by probability, with the most probable first.
4. Generate leaf nodes for each symbol, including P, and add them to a queue.
5. While (Nodes in Queue > 1)

    i. Remove the two lowest probability nodes from the queue.
    ii. Prepend 0 and 1 to the left and right nodes' codes, respectively.
    iii. Create a new node with value equal to the sum of the nodes' probability.
    iv. Assign the first node to the left branch and the second node to the right branch.
    v. Add the node to the queue

6. The last node remaining in the queue is the root of the Huffman tree

2.3.3.3 Arithmetic Coding :

This method was developed in 1979 at IBM, which was investigating data compression techniques for use in their mainframes. Arithmetic coding is arguably the most optimal entropy coding technique if the objective is the best compression ratio since it usually achieves better results than Huffman Coding. It is, however, quite complicated compared to the other coding techniques.

Rather than splitting the probabilities of symbols into a tree, arithmetic coding transforms the input data into a single rational number between 0 and 1 by changing the base and assigning a single value to each unique symbol from 0 up to the base. Then, it is further transformed into a fixed-point binary number which is the encoded result. The value can be decoded into the original output by changing the base from binary back to the original base and replacing the values with the symbols they correspond to.

A general algorithm to compute the arithmetic code is:

1. Calculate the number of unique symbols in the input. This number represents the base b (e.g. base 2 is binary) of the arithmetic code.
2. Assign values from 0 to b to each unique symbol in the order they appear.
3. Using the values from step 2, replace the symbols in the input with their codes
4. Convert the result from step 3 from base b to a sufficiently long fixed-point binary number to preserve precision.
5. Record the length of the input string somewhere in the result as it is needed for decoding.

Here is an example of an encode operation, given the input “ABCDAABD”:

1. Found 4 unique symbols in input, therefore base = 4. Length = 8
2. Assigned values to symbols: A=0, B=1, C=2, D=3
3. Replaced input with codes: “0.01230013_4” where the leading 0 is not a symbol.
4. Convert “0.01231123_4” from base 4 to base 2: “0.01101100000111_2”
5. Result found. Note in result that input length is 8.

Assuming 8-bit characters, the input is 64 bits long, while its arithmetic coding is just 15 bits long resulting in an excellent compression ratio of 24%. This example demonstrates how arithmetic coding compresses well when given a limited character set.

2.3.3.4 Adaptive vs. Static encodings :

One problem with the previous three encodings is that the decoder needs to know the probability distribution before it starts the decoding process. It's also a problem from the encoder's point of view: if it doesn't a priori know the probability distribution, it needs to scan through the entire input stream and compute the probabilities before rewinding to the beginning and encoding the stream. If you're encoding a real-time stream, and you don't know the probability distribution, this is less than ideal.

One way to solve this is to compress in chunks. You read in a whole chunk, calculate the statistics, dump the probability distribution, dump the compressed chunk, and repeat, each time resetting the statistics to zero.

Another solution is to use an adaptive scheme. In an adaptive scheme, you assume at the beginning that the probability distribution is uniform. That is, every character has an equal chance of occurring. Each time you encode a symbol, you update the probability of that symbol, and readjust whatever coding mechanism you're using. This sort of scheme has the advantage that you don't even have to know how much data you're going to compress: you just take it as it comes in. It also deals well with input streams that change characteristics over time (especially if you do something clever like only keep track of the last n bytes when calculating the probability distribution).

2.3.4 Compression Algorithms:

2.3.4.1 The Lempel-Ziv Algorithms:

The Lempel-Ziv algorithms compress by building a dictionary of previously seen strings. Unlike PPM which uses the dictionary to predict the probability of each character, and codes eachcharacter separately based on the context, the Lempel-Ziv algorithms code groups of characters ofvarying lengths. The original algorithms also did not use probabilities—strings were either in thedictionary or not and all strings in the dictionary were give equal probability. Some of the newervariants, such as gzip, do take some advantage of probabilities.

At the highest level the algorithms can be described as follows. Given a position in a file,look through the preceding part of the file to find the longest match to the string starting at thecurrent position, and output some code that refers to that match. Now move the finger past thematch.

The two main variants of the algorithm were described by Ziv and Lempel in two separatepapers in 1977 and 1978, and are often referred to as LZ77 and LZ78. The algorithms differ in howfar back they search and how they find matches. The LZ77 algorithm is based on the idea of asliding window. The algorithm only looks for matches in a window a fixed distance back from thecurrent position. Gzip, ZIP, and V.42bis (a standard modem protocol) are all based on LZ77. TheLZ78 algorithm is based on a more conservative approach to adding strings to the dictionary. Unixcompress, and the Gif format are both based on LZ78.In the following discussion of the algorithms we will use the term cursor to mean the positionan algorithm is currently trying to encode from.

2.3.4.2 Sliding Window Algorithms : LZ77

Published in 1977, LZ77 is the algorithm that started it all. It introduced the concept of a 'sliding window' for the first time which brought about significant improvements in compression ratio over more primitive algorithms. LZ77 maintains a dictionary using triples representing offset, run length, and a deviating character. The offset is how far from the start of the file a given phrase starts at, and the run length is how many characters past the offset are part of the phrase. The deviating character is just an indication that a new phrase was found, and that phrase is equal to the phrase from offset to ***offset+length*** plus the deviating character. The dictionary used changes dynamically based on the sliding window as the file is parsed. For example, the sliding window could be 64MB which means that the dictionary will contain entries for the past 64MB of the input data.

Given an input "abbadabba" the output would look something like "abb(0,1,'d')(0,3,'a')" as in the example below:

Position	Symbol	Output
0	a	a
1	b	b
2	b	b
3	a	(0, 1, 'd')
4	d	
5	a	(0, 3, 'a')
6	b	
7	b	
8	a	

While this substitution is slightly larger than the input, it generally achieves a considerably smaller result given longer input data.

2.3.4.3 Dictionary Algorithms : LZ78

LZ78 was created by Lempel and Ziv in 1978, hence the abbreviation. Rather than using a sliding window to generate the dictionary, the input data is either preprocessed to generate a dictionary with infinite scope of the input, or the dictionary is formed as the file is parsed. LZ78 employs the latter tactic. The dictionary size is usually limited to a few megabytes, or all codes up to a certain numbers of bytes such as 8; this is done to reduce memory requirements. How the algorithm handles the dictionary being full is what sets most LZ78 type algorithms apart.

While parsing the file, the LZ78 algorithm adds each newly encountered character or string of characters to the dictionary. For each symbol in the input, a dictionary entry in the form (dictionary index, unknown symbol) is generated; if a symbol is already in the dictionary then the dictionary will be searched for substrings of the current symbol and the symbols following it. The index of the longest substring match is used for the dictionary index. The data pointed to by the dictionary index is added to the last character of the unknown substring. If the current symbol is unknown, then the dictionary index is set to 0 to indicate that it is a single character entry. The entries form a linked-list type data structure.

An input such as "abbadabbaabaad" would generate the output "(0,a)(0,b)(2,a)(0,d)(1,b)(3,a)(6,d)". You can see how this was derived in the following example:

Input:	■	a	b	ba	d	ab	baa	baad
Dictionary Index	0	1	2	3	4	5	6	7
Output	NULL	(0,a)	(0,b)	(2,a)	(0,d)	(1,b)	(3,a)	(6,d)

2.3.4.4 LZW

LZW is the Lempel-Ziv-Welch algorithm created in 1984 by Terry Welch. It is the most commonly used derivative of the LZ78 family, despite being heavily patent-encumbered. LZW improves on LZ78 in a similar way to LZSS; it removes redundant characters in the output and makes the output entirely out of pointers. It also includes every character in the dictionary before starting compression, and employs other tricks to improve compression such as encoding the last character of every new phrase as the first character of the next phrase. LZW is commonly found in the Graphics Interchange Format, as well as in the early specifications of the ZIP format and other specialized applications. LZW is very fast, but achieves poor compression compared to most newer algorithms and some algorithms are both faster and achieve better compression.

2.4 Chomsky classification :

Most famous classification of grammars and languages introduced by Noam Chomsky is divided into four classes:

- Recursively enumerable grammars –recognizable by a Turing machine
- Context-sensitive grammars –recognizable by the linear bounded automaton
- Context-free grammars - recognizable by the pushdown automaton
- Regular grammars –recognizable by the finite state automaton

2.4.1 Type-0 : Recursively enumerable grammar

Type-0 grammars (unrestricted grammars) include all formal grammars. They generate exactly all languages that can be recognized by a Turing machine. These languages are also known as the recursively enumerable languages. Note that this is different from the recursive languages which can be decided by an always-halting Turing machine.
Class 0 grammars are too general to describe the syntax of programming languages and natural languages.

2.4.2 Type-1 : Context-sensitive grammars

Type-1 grammars generate the context-sensitive languages. These grammars have rules of the form $\alpha A \beta \rightarrow \alpha \gamma \beta$ with **A** a nonterminal and α,β and γ strings of terminals and nonterminals. The strings α and β may be empty,but γ must be nonempty. The languages described by these grammars are exactly all languages that can be recognized by a linear bounded automaton.

Example:
AB → CDB
AB → CdEB
ABcd → abCDBcd
B → b

2.4.3 Type- 2 : Context-free grammars

Type-2 grammars generate the context-free languages. These are defined by rules of the form $A \rightarrow \gamma$ with **A** a nonterminal and γ a string of terminals and nonterminals. These languages are exactly all languages that can be recognized by a non-deterministic pushdown automaton. Context-free languages are the theoretical basis for the syntax of most programming languages.

Example:
$A \rightarrow aBc$

2.4.4 Type-3 : Regular grammars

Type-3 grammars generate the regular languages. Such a grammar restricts its rules to a single nonterminal on the left-hand side and a right-hand side consisting of a single terminal,possibly followed (or preceded,but not both in the same grammar) by a single nonterminal. The rule $S \rightarrow \varepsilon$ is also allowed here if **S** does not appear on the right side of any rule. These languages are exactly all languages that can be decided by a finite state automaton. Additionally,this family of formal languages can be obtained by regular expressions. Regular languages are commonly used to define search patterns and the lexical structure of programming languages.

Example:
$A \rightarrow \varepsilon$
$A \rightarrow a$
$A \rightarrow abc$
$A \rightarrow B$
$A \rightarrow abcB$

2.5 Grammar-based compression:

Grammar-based compressions (or Grammar-based codes) are compression algorithms based on the idea of constructing a context-free grammar (CFG) for the string to be compressed. To compress a data sequence $x = x_1 \ldots x_n$, a grammar-based code transforms$x$into a context-free grammar G . The problem to find a smallest grammar for an input sequence is known to be *NP-hard* , so many grammar-transform algorithms are proposed from theoretical and practical viewpoints. The produced grammar G can be further compressed by statistical encoders.

2.5.1 The 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. Others also add the number of rules to that. The (decision version of the) problem is NP-complete.

“ *What is the smallest context-free grammar that generates exactly one given stringσ ?* ”

This is a natural question about a fundamental object connectedto many fields such as data compression, Kolmogorov complexity,pattern identification, and addition chains.

Due to the problem’s inherent complexity, our objective is to find an approximation algorithm which finds a small grammar for the input string. We focus attention on the approximation ratio of the algorithm (and implicitly, the worst case behavior).

The smallest grammar problem was articulated explicitly bytwo groups of authors at about the same time. Nevill-ManningandWitten stated the problem and proposed the SEQUITUR algorithm as a solution [31], [16]. Their main focus was on extractingpatterns from DNA sequences, musical scores, and even theChurch of Latter-Day Saints genealogical database, althoughthey evaluated SEQUITUR as a compression algorithm as well.

The other group, consisting of Kieffer, Yang, Nelson, andCosman, approached the smallest grammar problem from atraditional data compression perspective [33], [32], [27]. First,they presented same deep theoretical results on the impossibilityof having a "*best*" compressor under a certain type ofgrammar compression model for infinite length strings [11].Then, they presented a host of practical algorithms includingBISECTION, multilevel pattern matching (MPM), and LONGESTMATCH. Furthermore, they gave an algorithm, which we referto as SEQUENTIAL, in the same spirit as SEQUITUR, but withsignificant defects removed. All of these algorithms are described later on. Interestingly, on inputswith power-of-two lengths, the BISECTION algorithm of Nelson, Kieffer, and Cosman [34] gives essentially the same representation as a binary decision diagram [35]. Binary decisiondiagrams have been used widely in digital circuit analysissince the 1980s and also recently exploited for more general compression tasks [36], [37].

2.5.1.1 Theorem : 1

There is no polynomial-time algorithm for thesmallest grammar problem with approximation ratio less than*8569/8568*unless$P = NP$ [1].

2.5.2 Approximation Algorithms :

For many important optimization problems, there is no known polynomial-time algorithm tocompute the exact optimum. In fact, when we discuss the topic of NP-completeness later in thesemester, we'll see that a great many such problems are all equivalently hard to solve, in thesense that the existence of a polynomial-time algorithm for solving any one of them would implypolynomial-time algorithms for all the rest.

The study of approximation algorithms arose as a way to circumvent the apparent hardness ofthese problems by relaxing the algorithm designer's goal: instead of trying to compute an exactlyoptimal solution, we aim to compute a solution whose value is as close as possible to that of theoptimal solution. However, in some situations it is desirable to run an approximation algorithmeven when there exists a polynomial-time algorithm for computing an exactly optimal solution.

For example, the approximation algorithm may have the benefit of faster running time, a lowerspace requirement, or it may lend itself more easily to a parallel or distributed implementation.These considerations become especially important when computing on “big data," where theinput size is so astronomical that a running time which is a high-degree polynomial of the inputsize (or even quadratic, for that matter) cannot really be considered an efficient algorithm, atleast on present-day hardware.

In light of the apparent intractability of the problems we believe not to lie in P, it makes sense to pursue ideas other than complete solutions to these problems. Three standard approaches include:

- *Exploiting special problem structure*: perhaps we do not need to solve the general case of the problem but rather a tractable special version;
- *Heuristics*: procedures that tend to give reasonable estimates but for which no proven guarantees exist;
- *Approximation algorithms*: procedures which are proven to give solutions within a factor of optimum.

Of these approaches, approximation algorithms are arguably the most mathematically satisfying, and will be the subject of discussion for this section.
An algorithm is a factor α approximation (α -approximation algorithm) for a problem iff for every instance of the problem it can find a solution within a factor α of the optimum solution.

If the problem at hand is a minimization then $\alpha > 1$ and this definition implies that the solution found by the algorithm is at most α times the optimum solution. If the problem is maximization, $\alpha < 1$ and this definition guarantees that the approximate solution is at least α times the optimum.

It has been determined that a large class of common optimization problemsis classified as NP-hard. It is widely believed—though not yet proven (ClayMathematics Institute, 2003)—that NP-hard problems are intractable, whichmeans that there does not exist an efficient algorithm (i.e. one that scales polynomially)that is guaranteed to find an optimal solution for such problems.

Examples of NP-hard optimization tasks are the minimum traveling salesmanproblem, the minimum graph coloring problem, and the minimum bin packingproblem. As a result of the nature of NP-hard problems, progress that leads toa better understanding of the structure, computational properties, and ways ofsolving one of them, exactly or approximately, also leads to better algorithmsfor solving hundreds of other different but related NP-hard problems. Severalthousand computational problems, in areas as diverse as economics, biology,operations research, computer-aided design and finance, have been shown tobe NP-hard.

A natural question to ask is whether approximate (i.e. near-optimal) solutionscan possibly be found efficiently for such hard optimization problems.Heuristic local search methods, such as tabu search and simulated annealing, are often quite effective at finding near-optimal solutions.However, these methods do not come with rigorous guarantees concerningthe quality of the final solution or the required maximum runtime. The design of good approximation algorithms is a very active area of researchwhere one continues to find new methods and techniques. It is quitelikely that these techniques will become of increasing importance in tacklinglarge real-world optimization problems.

2.5.3 Approximation Algorithms for Grammar-Based Compression:

Grammar-based data compression was first proposed explicitly by Kieffer and Yang [11] and Nevill- Manning [16], but is closely related to some earlier **“macro-based" schemes** proposed by Storer [20].

Several grammar-based compression algorithms have been proposed. Nevill-Manning [16] devised the Sequitur algorithm which incrementally builds a grammar in a single pass through the input string. This procedure was subsequently improved by Kieffer and Yang [11] to what we refer to here as the Sequentialalgorithm. The same authors employed a completely different approach to generating a compact grammarfor a given string in their Bisection algorithm. Thisprocedure partitions the input into halves, then quarters,then eighths, etc. and creates a nonterminal inthe grammar for each distinct substring generated inthis way. Bisection was subsequently generalized to Multilevel Pattern Matching (MPM) [27] in order to exploit multi-way and incompletepartitioning. De Marcken [26] presented a complex multi-pass algorithm that emphasizes avoiding local minima. Apostolico and Lonardi [28] proposed a greedy algorithm(hereafter called Greedy) in 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 LZ78 algorithm [30] can also be interpreted as a grammar. (In contrast, the output ofLZ77 [29] has no natural interpretation as a grammar.)

Here six previously proposed algorithms for the smallest grammar problem: LZ78, BISECTION, SEQUENTIAL, LONGEST MATCH, GREEDY, and RE-PAIR are being discussed. In addition, I discuss some closely related algorithms: LZW, MPM, and SEQUITUR also.

Although most of the algorithms here were originally designed as compression algorithms, those are viewed as approximation algorithms for the smallest grammar problem. A good grammar-based compression algorithm should attempt to find the smallest possible grammar generating the input string. Nonetheless, there do exist disconnects between the theoretical study of the smallest grammar problemand practical data compression.

2.5.3.1 LZ78 :

The well-known LZ78 [30] algorithm can be regardedas a grammar-based compressor. The procedureworks as follows :

1. Begin with an empty grammar.
2. Makea single left-to-right pass through the input string.
3. At each step, find the shortest prefix of the unprocessedportion that is not the expansion of a secondary nonterminal.
4. This prefix is either a single terminal a or elseexpressible as$Xa$where$X$is an existing nonterminaland a is a terminal.
5. Define a new nonterminal, either$Y \rightarrow a$ or $Y \rightarrow Xa$, and append this new nonterminalto the end of the start rule.

2.5.3.2 LZW:

Some practical improvements on LZ78 are embodiedin a later algorithm, LZW [38]. The grammars implicitlygenerated by the two procedures are not substantively different,but LZW is more widely used in practice.

2.5.3.3 SEQUITUR Algorithm :

SEQUITUR forms a grammar from a sequence based on repeated phrases in that sequence. Each repetition gives rise to a rule in the grammar, and the repeated subsequence is replaced by a nonterminal symbol, producing a more concise representation of the overall sequence. It is this pursuit of brevity that drives the algorithm to form and maintain the grammar, and, as a by-product, provide a hierarchical structure for the sequence [16].

At the left of Figure-1a is a sequence that contains the repeating string *bc* . Note that the sequence is already a grammar—a trivial one with a single rule. To compress it, SEQUITUR forms a new rule $A \rightarrow bc$, and *A* replaces both occurrences of *bc* . The new grammar appears at the right of Figure-1a.

The sequence in Figure-1b shows how rules can be reused in longer rules. The longer sequence consists of two copies of the sequence in Figure-1a. Since it represents an exact repetition, compression can be achieved by forming the rule $A \rightarrow abcdbc$ to replace both halves of the sequence. Further gains can be made by forming rule $B \rightarrow bc$ to compress rule *A*. This demonstrates the advantage of treating the sequence, rule *S*, as part of the grammar—rules maybe formed from rule *A* in an analogous way to rules formed from rule *S*. These rules within rules constitute the grammar's hierarchical structure.

The grammars in Figures 1a and 1b share two properties:

p_1: no pair of adjacent symbols appears more than once in the grammar;
p_2: every rule is used more than once.

Property p_1 requires that every diagram in the grammar be unique, and will be referred to as ***diagram uniqueness***. Property p_2 ensures that each rule is useful, and will be called ***rule utility***. These two constraints exactly characterize the grammars that SEQUITUR generates.

Figure-1c shows what happens when these properties are violated. The first grammar contains two occurrences of *bc*, so p_1 does not hold. This introduces redundancy because *bc*appears twice. In the second grammar, rule*B* is used only once, so p_2 does not hold. If it were removed, the grammar would become more concise. The grammars in Figures 1a and 1b are the only ones for which both properties hold for each sequence. However, there is not always a unique grammar with these properties.

For example, the sequence in Figure-1d can be represented by both of the grammars on its right, and they both obey p_1 and p_2. We deem either grammar to be acceptable. Repetitions cannot overlap, so the string *aaa*does not give rise to any rule, despite containing two diagrams *aa*.

SEQUITUR's operation consists of ensuring that both properties hold. When describing the algorithm, the properties act as *constraints.* The algorithm operates by enforcing the constraints on a grammar: when the diagram uniqueness constraint is violated, a new rule is formed, and when the rule utility constraint is violated, the useless rule is deleted. The next two subsections describe how this occurs.

	Sequence	**Grammar**
a	S → abcdbc	S → aAdA A → bc
b	S → abcdbcabcdbc	S → AA A → aBdB B → bc
c	S → abcdbcabcdbc	S → AA A → abcdbc ——— S → CC A → bc B → aA C → BdA
d	S → aabaaab	S → AaA A → aab ——— S → AbAab A → aa

Figure 1 : Example sequences and grammars that reproduce them: (a) a sequence with one repetition; (b) a sequence with a nested repetition; (c) two grammars that violate the two constraints; (d) two different grammars for the same sequence that obey the constraints.

2.5.3.4 Bisection Algorithm:

Kieffer and Yang introduced the Bisection algorithmin [27]. This procedure works on an input stringσ as follows :

1. Select the largest integer$k$ such that$2^k < |\sigma|$.
2. Partition σ into two substrings with lengths$2^k$and $|\sigma| - 2^k$.
3. Repeat this partitioning process recursivelyon each substring of length greater than one.
4. Create a nonterminal for every distinct string of lengthgreater than one generated during this process.
5. Each such nonterminal can then be defined by a rule withexactly two symbols on the right.

2.5.3.5 Sequential Algorithm :

Nevill-Manning and Witten introduced the Sequiturgrammar compression algorithm in [31]. Kieffer and Yang [27] subsequently offered an improved algorithm,which we refer to here as Sequential. Sequentialworks as follows :

1. Begin with an empty grammar,and make a single left-to-right pass through the inputstring.
2. At each step, find the longest prefix of the unprocessedportion of the input that matches the expansionof a secondary nonterminal, and append that nonterminalto the start rule.
3. Otherwise, if no prefix matchesthe expansion of a secondary nonterminal, append the first terminal in the unprocessed portion to the startrule.
4. In either case, if the last pair of symbols in thestart rule already occurs at some non-overlapping positionin the grammar, then replace both occurrences by anew nonterminal whose definition is that pair.
5. Finally,if some nonterminal is used only once after this substitution,then replace it by its definition, and delete thecorresponding rule.

2.5.3.6 Global Algorithms:

The remaining algorithms analyzed here all belongto a single class, which is being referred to as global algorithms. The upper-bound of the approximation ratio of every global algorithm stands by$O((n/\log n)^{2/3})$with a single theorem [1]. However, ourlower bounds are all different, complex, and weak. Moreover,the lower bounds rely on strings over unbounded alphabets.Thus, it may be that every global algorithm has an excellent approximationratio. Because they are so natural and our understandingis so incomplete, global algorithms are one of the mostinteresting topics related to the smallest grammar problem thatdeserve further investigation.

1) ***The Procedure:*** A global algorithm begins with the grammar $S \rightarrow \sigma$. The remaining work is divided into rounds. During each round, one selects a ***maximal string*** γ. (Global algorithms differ only in the way they select a maximal string in each round.)

A maximal string has three properties.

(M1) It has length at least two.
(M2) It appears at least twice on the right side of thegrammar without overlap.
(M3) No strictly longer string appears at least as many timeson the right side without overlap.

1. After a maximal string is selected, a new rule $T \rightarrow \gamma$ is added to the grammar.
2. This rule is then applied by working left-to-right through the right side of every other rule, replacing each occurrence of γ by the symbol T .
3. The algorithm terminates when no more maximal strings exist.

2) ***Upper Bound:*** The approximation ratio of every globalalgorithm is $O((n/\log n)^{2/3})$. This follows from the fact thatgrammars produced by global algorithms are particularly wellconditioned; not only are they irreducible, but they also possessan additional property described by Lemma's [1].

2.5.3.7 Greedy:

Apostolico and Lonardi [28] considered Greedy algorithmsfor grammar-based data compression. The procedureworks as follows [22]:

1. Their idea is to begin with a grammar where the definition of thestart symbol is the entire input string.
2. Then one repeatedlyadds the rule that decreases the size of thegrammar as much as possible.
3. Each rule is added bymaking a pass through the string from left to right andreplacing each occurrence of the definition of the rule byits nonterminal.
4. Greedy terminates when no rule canbe added without enlarging the grammar.

2.5.3.8 Recursive Pairing (re-pair) :

The phrase derivation algorithm used in re-pairconsists of replacing the most frequent pair of symbols in the source message bya new symbol, re-evaluating the frequencies of all of the symbol pairs with respectto the extended alphabet, and then repeating the process until there is no pairof adjacent symbols that occurs twice [12]:

1. Identify symbols*a* and *b*such that *ab*is the most frequent pair of adjacentsymbols in the message. If no pair appears more than once, stop.
2. Introduce a new symbol A and replace all occurrences of ab with A .
3. Repeat from step 1.

The message is reduced to a new sequence of symbols, each of which representseither a unit symbol or a pair of recursively defined symbols. A zero-order entropycode for the reduced message is the final step in the compression process; and thepenultimate step is, of course, transmission of the dictionary of phrases.

We have not specified in which order pairs should be replaced when there areseveral pairs of equal maximum frequency. While this does influence the outcome ofthe algorithm, it appears to be of minor importance. The current implementationresolves ties by choosing the least recently accessed pair, which avoids skewness inthe hierarchy by discriminating against recently created pairs.

2.5.4 Chomsky normal form :

In formal language theory, a context-free grammar*G*is said to be in Chomsky normal form (CNF) (first described by Noam Chomsky) if all of its production rules are of the form:

$A \to BC$, or
$A \to a$, or
$S \to \varepsilon$,

Where A, B andC are nonterminal symbols,
a is a terminal symbol (a symbol that represents a constant value),
S is the start symbol,
and ε denotes the empty string.

Also, neither B nor C may be the start symbol, and the third production rule can only appear if ε is in Language of *G*(L(G)) , namely, the language produced by the context-free grammar G.

Every grammar in Chomsky normal form is ***context-free***, and conversely, every context-free grammar can be transformed into an equivalent one which is in Chomsky normal form and has a size no larger than the square of the original grammar's size.

2.5.5 Straight-Line Program (SLP) :

Definition :

A straight-line program (SLP) [13] over the terminalalphabet$\Gamma$is a context-free grammar A = (V, Γ, S, P)
(Where, V is the set of nonterminals,
Γ is the setof terminals,
S $\in$ V is the initial nonterminal,
and P $\subseteq$ V $\times$ (V $\cup\Gamma)^*$ is the set of productions)
such that the following two conditions hold:

(1) For every A $\in$ V there exists exactly one production of the form (A, α) $\in$ P for $\alpha \in (V \cup\Gamma)^*$.
(2) The relation {(A,B) | (A, α) $\in$ P,B $\in$alph(α)} is acyclic.

A production (A, α) is also written as A $\rightarrow \alpha$. Clearly, the language generated by the SLP Aconsists of exactly one word that is denoted by eval(A). More generally, from every nonterminalA$\in$ V we can generate exactly one word that is denoted by eval_A(A) (thus eval(A) = eval_A(S)).We omit the index Aif the underlying SLP is clear from the context.

The derivation tree of the SLP A = (V, Γ, S, P) is a finite rooted ordered tree, where everynode is labeled with a symbol from V $\cup\Gamma$. The root is labeled with the initial nontermial S andevery node that is labeled with a symbol fromΓ is a leaf of the derivation tree. A node that islabeled with a nonterminal Asuch that (A $\rightarrow \alpha 1 \cdots \alpha n$) $\in$ P (where $\alpha 1, \ldots, \alpha n \in V \cup\Gamma$) has*n* children that are labeled from left to right with $\alpha 1, \ldots, \alpha n$.

The size of the SLP A = (V, Γ, S, P) is |A| =$\sum_{(A,\alpha)\in P}|\alpha|$. Every SLP can be transformed inlinear time into an equivalent SLP in Chomsky normal form, where every production has the form(A, a) with a $\in\Gamma$or (A,BC) with B,C $\in$ V .

Example :

Consider the SLP A over the terminal alphabet {a, b} that consists of the followingproductions: $A_1 \rightarrow b$, $A_2 \rightarrow a$, and $A_i \rightarrow A_{i-1}A_{i-2}$for 3 $\leq$ i $\leq$ 7. The start nonterminal isA_7. Then eval(A) = *abaababaabaab*, which is the 7th Fibonacci word. The SLP A is in Chomsky Normal Form and |A| = 12.

A simple induction shows that for every SLP A of size *m* one has $|\text{eval}(A)| \leq O(3^{m/3})$. On the other hand, it is straightforward to define an SLP B in Chomsky normal formof size 2n such that $|\text{eval}(B)| \geq 2^n$. Hence, an SLP can be seen as a compressed representation ofthe string it generates, and exponential compression rates can be achieved in this way.

One may also allow exponential expressions of the form A^i for $A \in V$ and $i \in N$ in right-handsides of productions. Here the number i is coded binary. Such an expression can be replaced by a sequence of ordinary productions, where the length of that sequence is bounded polynomiallyinthe length of the binary coding of*i* .

For some applications, an extension of SLPs called composition systems in are useful.A composition system A is defined as an SLP but may also contain productions of the form$X \rightarrow Y [i, j]$ for $i, j \in N$ with $1 \leq i \leq j$. Assume that the string $w = \text{eval}_A(Y)$ is already defined.Then we let $\text{eval}_A(X) = w[i,\max\{|w|, j\}]$. A composition system is in Chomsky normal form ifall productions have the form $X \rightarrow a$, $X \rightarrow Y Z$ or $X \rightarrow Y [i, j]$ for nonterminals X, Y , and Zand a terminal symbol a. The following result was shown by Hagenah in his PhD thesis.

2.5.5.1 Theorem : 2

From a given composition system A in Chomsky normal form with n nonterminalsone can compute in time $O(n^2)$ an SLP B of size $O(n^2)$ such that eval(B) = eval(A) [13] .

CHAPTER : 3
METHODOLOGY OF PROPOSED WORK

3.1 Algorithm :

3.1.1 Notation conventions :

The input sequence to be represented by a context-free grammar is $T \in \Sigma*$,
I have used the same letter also for the text currently kept by the algorithm .
By N we denote the initial length of T ,
By |T| the current one.
The smallest grammar generating the input sequence is denoted by G and its size |G| , measured as the length of the productions, is g .

The algorithm TtoG introduces new symbols to the instance, those symbols are the nonterminals of the constructed grammar. However, these are later treated exactly as the original letters, so we insist on calling them letters as well and use common set Σ for both letters and nonterminals. I assumed that T is represented as a doubly-linked list, so that removal and replacement of its elements can be performed in constant time (assuming that we have a link to such an occurrence). Note though that if we were to store Tin a table, the running time would be the same.

3.1.2 Grammar :

The crucial part of the analysis is the modification of Gaccording to the compression performed on T. Still, when a new 'letter' a is introduced to Twe need to estimate the length of the 'productions' in the constructed grammar that are needed for a(note that we can of course use all letters previously used in T). The 'productions' introduced for ais called a representation of a lettera, the sum of lengths of those 'productions' is a cost of rep-resentation of a letter a (or simply: representation cost).

For example, in production (1a) then the representation cost is 2 (as we have only one rule c→ab) and in a rule (1b) we have a cost l; the latter cost can be significantly reduced, for instance for a12we can have a representation cost of 8instead of 12, when we use a subgrammar $a_2 \to aa$, $a_3 \to a_2a$, $a_6 \to a_3a_3$ and $a_{12} \to a_6a_6$. Note that when creplaces a pair (as in (1a)), its representation cost is always 2, but when areplaces a block of letters, say a^l, the cost might be larger than constant.

3.1.3 The algorithm:

The algorithm "**TtoG**"is divided into phases: in each phase :

- we first list all letters
- and for each of them we perform the block compression
- and then again list all letters, choose appropriate partition
- and perform the pair compression for each pair from this partition that occurs in the text.

Algorithm 1:
TtoG: outline

```
1: while|T|>1do
2:      L ←list of letters in T
3:      for each a ∈ L do                       >Blocks compression
4:      compress maximal blocks of a            >O(|T|)
5:      P←list of pairs
6:      find partition ofΣ into Σ_l and Σ_r     >Covering at least 1/2of
                                                occurrences of letters in T
7:                                              >O(|T|), see Lemma5
8:      for ab ∈ P∩Σ_lΣ_r do                    >These pairs do not overlap
9:      compress pair ab                        >Pair compression
10: returnthe constructed grammar
```

Before we make any analysis, we note that at the beginning of each phase we can make a linear-time preprocessing that guarantees that the letters in Tform an interval of numbers (which makes them more suitable for sorting using RadixSort).

3.1.4 Blocks compression :

The blocks compression is very simple to implement: We read T, for a maximal block of a's of length greater than 1. We create a record (a, l, p), where $l$is a length of the block, and pis the pointer to the first letter in this block. We then sort these records lexicographically using RadixSort (ignoring the last component). There are only O(¦T¦) records and we assume that _can be identified with an interval , this is all done in O(¦T¦) . Now, for a fixed letter a, the consecutive tuples with the first coordinate acorrespond to all blocks of a, ordered by the size. It is easy to replace them in O(¦T¦) time with new letters. Clearly, the space consumption is linear as well.

In the following we shall also use a simple property of the block compression: since no two maximal blocks of the same letter can be next to each other, after the block compression there are no blocks of length greater than 1in T.

3.1.5 Pair compression :

The pair compression is performed similarly as the block compression. However, since the pairs can overlap, compressing all pairs at the same time is not possible. Still, we can find a subset of non-overlapping pairs in Tsuch that a constant fraction (1/4) of letters Tis covered by occurrences of these pairs. This subset is defined by a partition of Σ into $\Sigma_l$and $\Sigma_r$and choosing the pairs with the first letter in Σ_l and the second in Σ_r; for a choice of $\Sigma_l\Sigma_r$we say that occurrences of $ab \in P \cap \Sigma_l\Sigma_r$ are covered by $\Sigma_l\Sigma_r$. The existence of a partition covers at least one fourth of the occurrences.

In order to make the selection effective, the algorithm "***GreedyPairs***" keeps an up to date counters $count_l[a]$and $count_r[a]$, denoting, respectively, the number of occurrences of pairs from $a\Sigma_l \cup \Sigma_l a$and $a\Sigma_r \cup \Sigma_r a$in T(for the current assignment of letters to $\Sigma_l$and Σ_r). Those counters are updated as soon as a letter is assigned to $\Sigma_l$orΣ_r.

Algorithm 2:
GreedyPairs

```
1: L←set of letters used in P
2: $\Sigma_l \leftarrow \Sigma_r \leftarrow \emptyset$                          >Organised as a bit vector
3: **for** $a \in L$ **do**
4:     $count_l[a] \leftarrow count_r[a] \leftarrow 0$                >Initialisation
5: **for** $a \in L$ **do**
6:     **if** $count_r[a] \geq count_l[a]$ **then**             >Choose the one that guarantees
                                                            larger cover
7:         *choice*←*l*
8:     **else**
9:         *choice*←*r*
10:    $\Sigma_{choice} \leftarrow \Sigma_{choice} \cup \{a\}$
11:    **for**each abor baoccurrence in T**do**
12:        $count_{choice}[b] \leftarrow count_{choice}[b] + 1$
13: **if** # occurrences of pairs from $\Sigma_r\Sigma_l$ in T > # occurrences of pairs from $\Sigma_l\Sigma_r$in T **then**
14:    switch $\Sigma_r$ and $\Sigma_l$
15: **return**($\Sigma_l$, $\Sigma_r$)
```

By the argument given above, when Σ is partitioned into $\Sigma_l$and $\Sigma_r$by GreedyPairs, at least half of the occurrences of pairs from Tare covered by $\Sigma_l\Sigma_r \cup \Sigma_r\Sigma_l$. Then one of the choices $\Sigma_l\Sigma_r$ or $\Sigma_r\Sigma_l$ covers at least one fourth of the occurrences.

3.1.6 Size and running time :

It remains to estimate the total running time, summed over all phases. Clearly each subprocedure in a phase has arun-ning time O(|T|)so it is enough to show that |T|is reduced by a constant factor per phase.

3.2 Size of the grammar: SLPs and recompression :

To bound the cost of representation of letters introduced during the construction of the grammar, we start with the smallest grammar G generating (the input) T and then modify the grammar so that it generates T (i.e. the current string kept by TtoG) after each of the compression steps. Then the cost of representing the introduced letters is paid by various credits assigned to G. Hence, instead of the actual representation cost, which is difficult to estimate, we calculate the total value of issued credit. Note that this is entirely a mental experiment for the purpose of the analysis, as G is not stored or even known to the algorithm. We just perform some changes on it depending on the TtoG actions.

3.2.1 Definitions :

We assume that grammar G is a Straight Line Programme (SLP), however, we relax the notion a bit (and call it an **SLP with explicit letters**, when an explicit reference is needed):

• then onterminals are $X_1, \ldots, X_m$;

• each nonterminal has exactly one rule, which has at most two nonterminals in its body (i.e. there are two, one or none nonterminals and an arbitrary number of letters in the rule's body);

• if $X_i \to \alpha_i$ is a rule and X_j occurs in $\alpha_i$then $j < i$.

Note that every CFG generating a unique string can be transformed into an SLP with explicit letters, with the size increased only by a constant factor:

• The renaming of nonterminals is obvious, we also remove the useless nonterminals.

• If a nonterminal X with a rule $X \to \alpha$ has more than two nonterminals in α, we can replace a substringYwZin α by a new nonterminal $X'$with a rule $X' \to YwZ$. In this way the number of nonterminals in α drops by 1 and the size of the grammar increases by 1.

• As only one string is generated, we can reorder the nonterminals.

We call the letters (strings) occurring in the productions the ***explicit letters*** (***strings***, respectively). The unique string derived by $X_i$is denoted by val(X_i); the grammar G shall satisfy the condition val(X_m) = T , where m = |T| . We do not assume that val(X_i) ≠ , however, if val(X_i) = then $X_i$is not used in the productions of G (as this is a mental experiment, such $X_i$can be removed from the rules and in fact from the SLP).

3.2.2 Intuition and road map:

3.2.2.1 Paying the representation cost: credit

With each explicit letter we associate two units of ***credit*** and pay most of the cost of representing the letters introduced during TtoG with these credits. More formally: when the algorithm modifies G and in the process it creates an occurrence of a letter, we ***issue*** (or pay) 2 new credits. On the other hand, if we do a compression step in G, then we remove some occurrences of letters. The credit associated with these occurrences is then ***released*** and can be used to pay for the represen-tation cost of the new letters introduced by the compression step as well as for the credit for the newly introduced letters (so that the algorithm does not issue new credit). For ***pair compression*** the released credit indeed suffices to pay both the credit of the new letters occurrences and their representation cost, but for ***chain compression*** the released credit does not suffice, as it is not enough to pay the representation cost. Here we need some extra amount that is estimated separately later on. In the end, the total cost is the sum of credit that was issued during the modifications of G plus the value that we estimate separately later on.

3.2.2.2 Additional cost :

The additional cost of representing letters during the block compression is estimated separately. In most cases, the cost of creating blocks can be cover by released credit, the only exception is when two long blocks of a are joined together. This can happen only between nonterminals in some rule of G and then the additional cost is charged towards this rule. Then we show that one rule has only O(log N) cost charged to it: if we charge $\sum_i log l_i$ cost to a rule, then it originally derived a word of length at least $\prod_i l_i$. This is described in detail later on.

3.2.2.3 Modifying the grammar :

Recall that whenever we say nonterminal, rule, production etc., we mean one of G. When we replace each occurrence of the pair ab in T , we should also do this in G. However, this may not be possible, as some abs generated by G do not come from explicit pairs in G but rather are ‘between’ a nonterminal and a letter, for instance in a simple grammar $X_1 \rightarrow a$, $X_2 \rightarrow X_1 b$ the pair *ab* has such a problematic occurrence. If there are no such occurrences, it is enough to replace each explicit ab in G and we are done. To deal with the problematic ones, we need to somehow change the grammar, in the example above we replace $X_1$with a, leaving only $X_2 \rightarrow ab$, for which the previous procedure can be applied. It turns out that this ad-hoc approach can be turned into a systematic procedure that deals with all such problems at once, by removing appropriate letters from rules and introducing them to other rules. This procedure is the main ingredient of this section and it is given in Section 3.4. Similar problems occur also when we want to replace maximal blocks of *a* and the solution to this problem is similar and it is given in Section 3.5.

Note that in the example above, when $X_1$is replaced with *a*, 2 credit for the occurrence of a in $X_1 \rightarrow a$ is released and wasted. Then we issue 2 credit for the new occurrence of a in the rule X_2. When *ab* is replaced with *c*, 4 credit is released when *ab* is removed from the rule, 2 of this credit is used for the credit of c and the remaining 2 can be used to pay the representation cost for $c \rightarrow ab$.

3.2.2.4 Pair compression :

A pair of letters *ab* has a crossing occurrence in a nonterminal X_i(with a rule $X_i \rightarrow \alpha_i$) if *ab* is in val(X_i) but this occurrence does not come from an explicit occurrence of *ab* in $\alpha_i$nor it is generated by any of the nonterminals in α_i. A pair is non-crossing if it has no crossing occurrence. Unless explicitly written, we use this notion only to pairs of different letters.

By $PC_{ab \rightarrow c}(w)$we denote the text obtained from w by replacing each ab by a letter c (we assume that a ≠b). We say that a procedure (that changes a grammar G with nonterminals $X_1, \ldots, X_m$ to $G'$with nonterminals $X_1', \ldots, X_m'$) properly implements the pair compression of *ab* to *c*, if $\text{val}(X_m') = PC_{ab \rightarrow c}(\text{val}(X_m))$ and $G'$is an SLP with explicit letters. When a pair *ab* is noncrossing the procedure that implements the pair compression is easy to give: it is enough to replace each explicit *ab* with *c*.

Algorithm 3:

PairCompNCr(ab, c):

compressing a non-crossing pair *ab*.

```
1: replace each explicit $ab$ in $G$ by $c$
```

In order to distinguish between the nonterminals, grammar, etc. before and after the application of compression of ab (or, in general, any procedure) we use ***‘primed’*** letters, i.e. X_i', G', $T'$for the nonterminals, grammar and text after this compression and ***‘unprimed’***, i.e. X_i, G, T for the ones before.

If all pairs in $\Sigma_l\Sigma_r$are non-crossing, iteration of PairCompNCr(*ab,c*)for each pair *ab*in $\Sigma_l\Sigma_r$properly implements the pair compression for all pairs in $\Sigma_l\Sigma_r$(note that as $\Sigma_l$and $\Sigma_r$are disjoint, occurrences of different pairs from $\Sigma_l\Sigma_r$cannot overlap and so the order of replacement does not matter). So it is left to assure that indeed the pairs from $\Sigma_l\Sigma_r$are all noncrossing. It is easy to see that $ab \in \Sigma_l\Sigma_r$is a crossing pair if and only if one of the following three ‘***bad***’ situations occurs:

CP1 : there is a nonterminal X_i, where $i < m$, such that val(X_i)begins with *b*and *a*$X_i$occurs in one of the rules;
CP2 : there is a nonterminal X_i, where $i < m$, such that val(X_i)ends with *a*andX_i*b*occurs in one of the rules;

CP3 : there are nonterminals X_i, X_j, where i, j <m, such that val(X_i)ends with *a*andval(X_j)begins with *b*and $X_iX_j$occurs in one of the rules.

Consider (CP1), let *bw* =val(X_i). Then it is enough to modify the rule for $X_i$so that val(X_i) =*w*and replace each $X_i$in the rules by bX_i, we call this action the ***left-popping****b*from X_i. Similar operation of ***right-popping*** a letter *a*from$X_i$is symmetrically defined. It is shown in Lemma [4] that they indeed take care of all crossing occurrences of *ab*.

Furthermore, left-popping and right-popping can be performed for many letters in parallel: the below procedure Pop(Σ_l, Σ_r)'***uncrosses***' all pairs from the set $\Sigma_l \Sigma_r$, assuming that Σ_l and Σ_r are disjoint subsets of Σ(and we apply *Pop(* Σ_l, Σ_r*)*only in the cases in which they are).

Algorithm 4 :
Pop(Σ_l ,Σ_r):
Popping letters from Σ_l and Σ_r.

```
1: for i ←1 … (m −1) do
2:      let the production for $X_i$be $X_i \to \alpha_i$
3:      ifthe first symbol of $\alpha_i$is $b \in \Sigma_r$ then            >Left-popping b
4:           remove this bfrom αi
5:           replace$X_i$in G's productions by $bX_i$
6:           ifval($X_i$) = ε then
7:                remove$X_i$from G's productions
8: for i ←1 … (m −1) do
9:      let the production of $X_i$be $X_i \to \alpha_i$
10:     ifthe last symbol of $\alpha_i$is $a \in \Sigma_l$then              >Right-popping a
11:     remove this afrom αi
12:     replace$X_i$in G's productions by $X_ia$
13:     if val($X_i$) = ε then
14:          remove$X_i$from G's productions
```

In order to compress pairs from $\Sigma_l \Sigma_r$ it is enough to first uncross them all using Pop(Σ_l, Σ_r)and then compress them all by PairCompNCr(*ab, c*) for each $ab \in \Sigma_l \Sigma_r$.

Algorithm 5 :
PairComp(Σ_l, Σ_r):
compresses pairs from Σ_l Σ_r .

```
1: run Pop($\Sigma_l$, $\Sigma_r$ )
2: for ab∈$\Sigma_l$ $\Sigma_r$ do
3:       runPairCompNCr($ab, c$)                          >$c$is a fresh letter
```

Lemma :

PairCompimplements pair compression for each ab∈Σ_l Σ_r. It issues O(m)new credit to G, where *m*is the number of nonterminals of G. The credit of the new letters introduced to Gand their representation costs are covered by the credit issued or released by PairComp.

Using this Lemma [4] we can estimate the total credit issued during the pair compression.

Corollary 1:

The compression of pairs issues in total O(m logN)credit during the run of TtoG; the credit of the new letters introduced to Gand their representation costs are covered by the credit issued or released during PairComp [4].

3.2.2.5 Blocks compression:

Similar notions and analysis as the ones for pairs are applied for blocks. Consider occurrences of maximal a-blocks in *T*and their derivation by *G*. Then a block a^{l}has a ***crossing occurrence***inX$_i$with a rule X$_i$→α$_i$, if it is contained in val(X$_i$)but this occurrence is not generated by the explicit as in the rule nor in the substrings generated by the nonterminals inα$_i$. If a-blocks have no crossing occurrences, then ***ahas no crossing blocks***. As for noncrossing pairs, the compression of ablocks, when it has no crossing blocks, is easy: it is enough to replace each explicit maximal a-block in the rules of G. We use similar terminology as in the case of pairs: we say that a subprocedure properly implements a block compression for a.

Algorithm 6 :
BlockCompNCr(a),
which compresses ablocks when ahas no crossing blocks.

```
1:foreach$a^{l_m}$do
2:replace every explicit maximal block$a^{l_m}$inGby$a_{l_m}$
```

Note that we do not yet discuss the issued credit, nor the cost of the representation of letters representing blocks. It is left to ensure that no letter has a crossing block. The solution is similar to Pop, this time though we need to remove the whole prefix and suffix from val(X$_i$)instead of a single letter. The idea is as follows: suppose that ahas a crossing block because *aX*$_i$occurs in the rule and val(X$_i$)begins with a. Left-popping adoes not solve the problem, as it might be that val(X$_i$)still begins with a. Thus, we keep on left-popping until the first letter of val(X$_i$)is not *a*, i.e. we remove the *a*-prefix of val(X$_i$). The same works for suffixes.

Algorithm 7 :
RemCrBlocks:
removing crossing blocks.

```
1: for  i ←1 … (m −1)  do
2:        let  $a, b$ be the first and last letter of val(X$_i$)
3:        let $l_i$ , $r_i$ be the length of the $a$-prefix and $b$-suffix of val(X$_i$)
4:        If val(X$_i$) ∈$a^*$  then
5 :                 $r_i$=0and$l_i$=| val(Xi)|
6:        remove $a^{l_i}$ from the beginning and $b^{r_i}$ from the end of α$_i$
7:        replace X$_i$ by  $a^{l_i}$X$_i$$b^{r_i}$ in the rules
8:        if  val(X$_i$) = ε  then
9:                 remove X$_i$ from the rules
```

The compression of all blocks of letters is done by first running ***RemCrBlocks***and then compressing each of the block by ***BlockCompNCr***. Note that we do not compress blocks of letters that are introduced in this way. Concerning the number of credit, the arbitrary long blocks popped by RemCrBlocksare compressed (each into a single letter) and so at most 8credit per rule is issued.

Algorithm 8 :
BlockComp:
compresses blocks of letters.

```
1: run  RemCrBlocks
2: L ←list of letters in T
3: for each a ∈ L do
4:         run  BlockCompNCr($a$)
```

Corollary 2.

During the whole TtoGtheBlockCompissues in total O(m logN)credit. The credit of the new letters introduced to Gis covered by the issued credit.

Note that the cost of representation of letters replacing blocks is not covered by the credit, this cost is separately estimated in the next subsection 3.2.2.6.

3.2.2.6 Calculating the cost of representing letters in block compression :

The issued credit is enough to pay the 2 credit for occurrences of letters introduced during TtoGand the released credit is enough to pay the credit of the letters introduced during the pair compression and their representation cost. However, credit alone cannot cover the representation cost of letters replacing blocks. The appropriate analysis is presented in this section. The overall plan is as follows: firstly, we define a scheme of representing the letters based on the grammar Gand the way Gis changed by *BlockComp(the G-based representation)*. Then for such a representation schema, we show that the cost of representation is O(glogN). Lastly, it is proved that the actual cost of representing the letters by *TtoG (the TtoG-based representation)* is smaller than the G-based one, hence it is also O(glogN).

3.2.2.6.1 G-based representation :

The intuition is as follows: while the ablocks can have exponential length, most of them do not differ much, as in most cases the new blocks are obtained by concatenating letters athat occur explicitly in the grammar and in such a case the released credit can be used to pay for the representation cost. This does not apply when the new block is obtained by concatenating two different blocks of *a*(popped from nonterminals) inside a rule. However, this cannot happen too often: when blocks of lengthp$_1$, p$_2$, ...,p$_l$are compressed (at the cost of
O($\sum^{l}_{i=1}(1+\log p_i)$) = O(log($\prod^{l}_{i=1} p_i$)), as each$p_i \geq 2$) , the length of the corresponding text in the input text is $\prod^{l}_{i=1} p_i$, which is at most N. Thus O($\sum^{l}_{i=1}(1+\log p_i)$)=O(log($\prod^{l}_{i=1} p_i$))=O(logN)cost per nonterminal is scored.

Getting back to the representation of letters: we create a new letter for each ablock in the rule $X_i \rightarrow \alpha_i$afterRemCrBlockspopped prefixes and suffixes from X_1, ..., X_{i-1}but before it popped letters from X_i. (We add the artificial empty block ε to streamline the later description and analysis.) Such a block is a powerif it is obtained by concatenation of two a-blocks popped from nonterminals inside a rule (and perhaps some other explicit letters a), note that this power may be then popped from a rule (as it may be a prefix or suffix in this rule). This implies that in the rule $Xi \rightarrow uX_jvX_kw$the popped suffix of $X_j$and popped prefix of $X_k$are blocks of the same letter, say *a*, and furthermore $v \in a^*$. Note that it might be that one (or both) of $X_j$and$X_k$were removed in the process (in this case the power can be popped from a rule as well). For each block a^lthat is not a power we may uniquely identify another block a^k(perhaps ε , not necessarily a power) such that $a^l$was obtained by concatenating$l-k$explicit letters to a^kin some rule.

Lemma :

For each block a^lrepresented in the G-based representation that is not a power there is block a^k(perhaps k =0) such that $a^k$is also represented in G-based representation and $a^l$was obtained in a rule by concatenating $l–k$explicit letters that existed in the rule to a^k.

Note that the block a^k is not necessarily unique: it might be that there are several a^lblocks in Gwhich are obtained as different concatenations of $a^k$and $l−k$explicit letters.

We represent the blocks as follows:

1. for a block a^l that is a power we represent $a_l$using the binary expansion, which costs O (1 + log l);
2.for a block a^lthat is obtained by concatenating $l−k$explicit letters to a block a^k (see above Lemma [4]) we represent a_l as $a_k a^{l-k}$, which has a representation cost of $l-k+1$, this cost is covered by the $2(l-k) \geq l-k+1$credit released by the$l-k$ explicit letters a. Note that the credit released by those letters was not used for any other purpose. (The 2units of credit per occurrence of $a_l$in the rules of grammar are already covered by the credit issued by BlockComp.)

We refer to cost in 1 as the cost of ***representing powers***and redirect this cost to the nonterminal in whose rule this power is created. The cost in 2, as marked there, is covered by released credit.

3.2.2.6.2 Cost of G-based representation:

We now estimate the cost of representing powers. The idea is that if nonterminal $X_i$is charged the cost of representing powers of length $p_1, p_2, ..., p_l$, which have representation cost $O(\sum^l_{i=1}(1+\log p_i)) = O(\log(\prod^l_{i=1} p_i))$, then in the input this nonterminal generated a text of length at least$p_1, p_2, ..., p_l \leq N$and so the total cost of representing powers is O(logN)(per nonterminal). This is formalised in the lemma below.

Corollary 3.

The cost of G-based representation is$O(g+g\log N)$.

3.2.2.6.3 Comparing the G-based representation cost and TtoG-based representation cost :

We now show that the cost of TtoG-based representation is at most as high as G-based one. We first represent G-based representation cost using a weighted graph G_G, such that the G-based representation is (up to a constant factor) $w(G_G)$, i.e. the sum of weights of edges of G_G.

Similarly, the cost of TtoG-based representation has a graph representation G_{TtoG}.

We now show that $G_G$can be transformed to G_{TtoG}without increasing the sum of weights of the edges. This is done by simple redirection of edges and changing their cost.

Corollary 4.

The total cost of TtoG-representation is O(*g logN*).

3.3 Improved Analysis:

Intuitively, each "reasonable" grammar should have size $O(|T|)$: application of a rule $X \to \alpha$makes the current text longer by at least $|\alpha| - 1$, so the sum of all lengths of right-hand sides (so $|\alpha|$s) cannot be shorter than the input text. In some extreme cases this estimation might be better than $O(g\log N)$guaranteed by TtoG, thus TtoG should have an approximation guarantee $O(\min(N, g\log N))$. This approach can be further improved: the trivial upper bound applies to any intermediate string obtained during TtoGand we can choose any of those estimations. We choose a specific point, where $|T| \approx g$. As a result, we divide the analysis of a computation of TtoGinto two stages: the first one lasts while $|T| \geq g$and then the second one begins. We separately estimate the cost of representation in the first stage, by $O(g\log(N/g))$, and in the second, by $O(g)$. In total this yields $O(g+g\log(N/g))$; this matches the best known results for the smallest grammar problem[18,1,19]and is not worse than both $O(g\log N)$and $O(g)$.

3.3.1 Theorem : 3

The TtoGruns in linear time and returns a grammar of size $O(g+g\log (N/g))$, where gis the size of the optimal grammar for the input text.

Note that the time analysis was done already in Theorem1, in the rest of this section we focus on the improved size analysis.

3.3.2 Outline:

Firstly, we show that indeed any reasonable grammar for a text Thas size $O(T)$. This follows by simple calculation and shows that it is enough to calculate the cost of representation for the grammar when $|T| \geq g$. From Corol-lary1and Lemma13we know that those costs are covered by the issued credit and the additional representation cost for a-blocks. The analysis for credit is easy: since in each phase we introduce $O(m) \leq O(g)$credit, it is enough to bound the number of phases and this follows from the fact that we shorten the text in each phase.

On the other hand, the analysis of the representation cost for blocks is much more involved. The general outline remain as it was as in Section3.6: we again use the G-based representation as a middle step, estimate its cost and compare it with the TtoG-based one. The difference is in the estimation of the G-based representation cost. We no longer can simply charge $O(\log N)$cost to a rule, we need a more subtle analysis. Instead of direct charging to a rule $X_i \to \alpha_i$, we associate the cost with some of the letters (of the original text) generated by X_i. To this end we 'mark' those letters and distinguish between different such markings. We ensure that such markings are disjoint, there are at most 2of them per non-terminal and that the cost of representation is related to the total size of the markings, to be more precise, when we have markings of lengths p_1, p_2, ..., $p_k$then the G-representation cost is $O(\sum_{i=1}^{k} 1+\log p_i)$. Then the estimation of the size of the whole grammar is just a matter of calculation. The markings and the analysis of representation cost using them is performed in Section4.4. For technical reasons we also consider the cost of representation in the phase in which |T|is reduced from more thangto smaller than *g*separately, the analysis is a simple combination of the case when |T| >*g*and when |T| <*g*and is done in Section4.5. Wrapping up all estimations and giving the proof of Theorem2is done in Section4.6. What is left is to describe the way we modify the markings to ensure their properties. This technical construction is presented separately.

3.3.3 Linear bound:

We begin with formalizing the argument that any "reasonable" grammar has size O(|T|).

Lemma:

Let SLP *G*contain no production X→αwith |α| ≤1and assume that every production is used in the derivation defined by G. Then |G| ≤2|T| −1 [4].

In particular, if at any point the letters created so-far by TtoGhave representation cost kand the remaining text is Tthen the final grammar for the input tree has size at most k +2|T| −1.

Note that the grammar produced by TtoGclearly has the properties assumed by Lemma19: we introduce new letters in place of substrings of length at least 2and each of them is used in the derivation of the input text.

In the following analysis we focus on the phase such that the text before it has length greater or equal to gand after it is smaller than g. Such phase exists: clearly N≥g(as we can take the grammar with Ton the right-hand side) and so initially |T| ≥gand in the end Tis reduced to a single letter.

Lemma :

There is a phase in computation of TtoGsuch that at the beginning of the phase |T| ≥gand at the end of the phase |T| <g [4].

We separately estimate the cost of representation (i.e. issued credit and the cost of TtoG-based representation) up to the phase from Lemma20, in this phase and after it. For the first two we show an upper bound of O(g+glog(N/g)), for the latter we use Lemma19to get an estimation O(g)on the representation cost.

3.3.4 Credit and pair compression when text is long :

Lemma :

If at the beginning of the phase |T| ≥g then O(g+glog(N/g)) credit was issued.

From the above Lemma we know that the representation cost of letters introduced by pair compression is covered by the credit. Thus

Corollary 5:

Suppose that at the beginning of the phase |T| ≥g. Then the representation cost of letters introduced by pair compression till this phase is O(g+glog(N/g)).

3.3.5 Cost of representing blocks when text is long:

For the cost of representing blocks, we define the G-based and TtoG-based representations in the same way as previously. However, we slightly extend the notion: we consider those representations at any point of TtoG, not only at the end; this does not affect those notions in any way.

For both the G-based representation and the TtoG-based representation we again define graphs $G_G$and$G_{TtoG}$and by Lemma16the cost of G-based representation is $\Theta(w(G_G))$and by Lemma17the cost of TtoG-based representation is $\Theta(w(G_{TtoG}))$. Then Lemma18shows that we can transform$G_G$toG_{TtoG}without increasing the sum of weights. Hence it is enough to show that the G-based representation cost is at most O(g+log(N/g)).

The G-based representation cost consists of some released credit and the cost of representing powers, see Lemma16. The former was already addressed in Lemma21(the whole issued credit is O(g+glog(N/g))) and so it is enough to estimate the latter, i.e. the cost of representing powers.

The outline of the analysis is as follows: when a new power a_is represented, we mark some letters of the input text (and perhaps modify some other markings) those markings are associated with nonterminals and are named Xi-pre-power marking and Xi-in marking. The markings satisfy the following conditions:

M1 : each marking marks at least 2letters, no two markings mark the same letter;
M2 : for each $X_i$there is most one X_i-pre-power marking and at most one X_i-in marking;
M3 : when the substrings of length p_1, p_2, ..., $p_k$are marked, then the so-far cost of representing the powers by G-based representation is $c\sum_{i=1}^{k}(1+\log p_i)$ (for some fixed constant c).

We show that when we have a marking satisfying (M1)–(M3) then indeed the cost of representing blocks is O(g+glog(N/g)). The construction of the markings and the analysis of it, is technical and does not affect further estimations of the grammar size.

3.3.6 Intermediate phase:

We bounded the representation cost before the phase from Lemma20and after it, so it is left to estimate the cost within this phase.

Lemma :

The cost of representing letters in the phase from Lemma20is O(g+glog(N/g)).

3.3.7 Markings' modification:

The idea of preserving (M1) – (M3) is as follows: if a new power of length l is represented, this yields a cost O(1 +log$l$) =O(log l), we can choose c in (M3) so that this is at most $c$log$l$ (as $l \geq 2$). Then either we mark newl letters or we remove some marking of length l' and mark $l \cdot l'$ letters, it is easy to see that in this way (M1)–(M3) is preserved.

Whenever we are to represent powers a^{l_1}, a^{l_2}, ..., for each power a^{l} , where l>1, we find the right-most maximal block a^{l} inT. Let $X_i$be the smallest nonterminal that derives (before **RemCrBlocks**) this right-most occurrence of maximal a^{l} (clearly there is such a non-terminal, as $X_m$derives it). It is possible that this particular a^{l} in X_is' rule was obtained as a concatenation of$l$–$k$explicit letters to a^{k}(so, not as a power). In such a case we are lucky, as the representation of this $a_l$is paid by the credit and we do not need to separately consider the cost of representing power a^{l}. Otherwise the a_in this rule is obtained as a power and we mark some of the letters in the input that are 'derived' by this a^{l}. The type of marking depends on the way this particular a^{l} is 'derived': If one of the nonterminals in X_is'production was removed during RemCrBlocks, this marking is an **X_i-pre-power marking**. Otherwise, this marking is an **X_i-in marking**.

Consider the a^{l} and the 'derived' substring $w^{l}$of the input text. We show that if there are markings inside w^{l}, they are all inside the last among those w's.

Lemma :

Let a^{l}be an occurrence of a maximal block to be replaced with $a_l$which 'generates' w^{l} in the input text. If there is any marking within thisw^{l}then it is within the last among those w's. [4]

We now demonstrate how to mark letters in the input text. Suppose that we replace a power a^{l}, let us consider the right-most occurrence of this a^{l} inTand the smallest $X_i$that generates this occurrence. This a^{l}generates some w^{l}in the input text. If there are no markings inside w^{l} then we simply mark any $l$letters within w^{l}. In the other case, by Lemma25we know that all those markings are in fact in the last w. If any of them is the (unique) X_i-in marking, let us choose it. Otherwise choose any other marking. Let l'denote the length of the chosen marking. Consider, whether this marking in wis unique or not

uniquemarking :Then we remove it and mark arbitrary $l \cdot l'$letters inw^l; this is possible, as $|w| \geq l'$and so $|w^l| \geq l \cdot l'$. Since $\log(l \cdot l') = \log l + \log l'$, the(M3) is preserved, as it is enough to account for the $1 + \log l \leq c \log l$representation cost of a^las well as the $c \log l'$cost associated with the previous marking of length l'.

notunique :Then $|w| \geq l' + 2$(the 2for the other markings, see(M1)). We remove the marking of length l', let us calculate how many unmarked letters are in w^lafterwards: inw^{l-1}there are at least $(l-1)\cdot(l'+2)$letters (by the Lemma25: none of them marked) and in the last $w$there are at least l'unmarked letters (from the marking that we removed):

$$
\begin{aligned}
(l-1)\cdot(l'+2) + l' &= (l l' + 2l - l' - 2) + l' \\
&= l l' + 2l - 2 \\
&> l l'
\end{aligned}
$$

We mark those $l.l'$letters, as in the previous case, the associated $c\log(l\, l')$is enough to pay for the cost.

There is one issue: it might be that we created an X_i-in marking while there already was one, violating (M2). However, we show that if there were such a marking, it was within w^l(and so within the last w [4]) and so we could choose it as the marking that was deleted when the new one was created. Consider the previous X_i-in marking. It was introduced for some powerb^k, replaced by $b_k$that was a unique letter between the nonterminals in the rule for X_i, by Lemma24. Consider the rightmost substring of the input text that is generated by the explicit letters between nonterminals in the rule for X_i. The operations performed on Gcannot shorten this substring, in fact they often expand it. When $b_k$is created, this substring is generated by b_k, by Lemma24. When a_l is created, it is generated by a_l, [4] i.e. this is exactly w^l. So in particular w^lincludes the marking for b_k. This shows that (M1)–(M3) are preserved.

CHAPTER : 4
Comparative Study of Different Developed Model

The hardness of the smallest grammar problem naturally leads to two directions of research: on the one hand, several heuristics are considered [12,11,16], on the other, approximation algorithms, with a guaranteed approximation ratio, are proposed; in this paper we consider only the latter approach. Note that the heuristical algorithms can work differently depending on the distribution of letters in the input (and often the principle behind them assumes that the data has some sort of regularity). On the other hand, the approximation guarantees shown for the latter algorithms are universal, in the sense that they do not depend on the distribution of letters or any other properties of the provided text.

The first two algorithms with an approximation ratio $O(\ \log(N/g))$were developed independently (and simultaneously) by Rytter [18] and Charikar et al. [1]. They followed a more or less the similar approach. At first Rytter's Algorithm is presented.

4.1 Rytter's Algorithm in details :

Rytter's algorithm [18] applies the LZ77 compression to the input string and then transforms the obtained LZ77 represeNtation to an O(l log(N/l)) size grammar, where l is the size of the LZ77 representation. It is easy to show that $l \leq$ g and as f (x) = x *log*(N/x) is increasing, the bound O(g log(N/g)) on the size of the grammar follows (and so a bound O(log(N/g)) on approximation ratio). The crucial part of the construction is the requirement that the intermediate constructed grammar defines a derivation tree satisfying the AVL condition. The bound on the running time and the approximation guarantee are all consequences of the balanced form of the derivation tree and of the known algorithms for merging, splitting, etc. of AVL trees (in fact these procedures are much simpler in this case, as we do not store any information in the internal nodes [18]). Note that also the final grammar for the input text is balanced, which makes it suitable for later processing. Since the construction of LZ77 representation can be performed in linear time (assuming that the letters of the input word can be sorted in linear time), also the running time of the whole algorithm can be easily bounded by a linear function.

4.1.1 Construction of small grammar-based compression: [18]

Assume we have an LZ-factorization $f_1 f_2 \dots f_k$of w.We convert it into a grammarwhose size increases by a logarithmic factor. Assume we have LZ-factorizationw=$f_1 f_2 \dots f_k$and we have already constructed good (AVL-balanced and of size*O(i log n))*grammar Gfor the prefix$f_1 f_2 \dots f_{i-1}$. If $f_i$is a terminal symbol generatedby a nonterminal A then we set G := Cancat(G, A) [18] . Otherwise we locate thesegment corresponding to $f_i$in the prefix$f_1 f_2 \dots f_{i-1}$.

Due to the fact that G is balanced we can Jnd a logarithmic number of nonterminalsS_1 , S_2 ,… , $S_{t(i)}$of G such that f_i=val(S_1).val(S_2). … val($S_{t(i)}$). The sequenceS_1 , S_2 ,… , $S_{t(i)}$is called the grammar decomposition of the factor f_i .

We concatenate the parts of the grammar corresponding to this nonterminals with G,using the operation *Concat*. Assume the first |Σ| nonterminalscorresponds to letters of the alphabet, so they exist at the beginning. We initialize Gto the grammar generating the Jrst symbol of w and containing all nonterminals forterminal symbols, they do not need to be initially connected to the string symbol. Thealgorithm starts with the computation of LZ-factorization, this can be done using suffix trees in*O(n log |Σ|)*time, see [39].

If LZ-factorization is too large (exceeds*n/log n*) then we neglect it and write atrivial grammar of size *n* generating a given string. Otherwise we have only$k \leq n \log n$ factors, they are processed from left to right.

We have*t(i) = O(log n)* [18] , so the number of two-arguments concatenationsneeded to implement single step (2) is*O(log n)*, each of them adding *O(log n)* nonterminals. Steps (1) and (3) can be done in *O(log n)*time, since the height of thegrammar is logarithmic. Hence the algorithm gives $O(log^2(n))$ -ratio approximation.

At the cost of slightly more complicated implementation of step (2) $log^2 n$ -ratio canbe improved to a *log n*-ratio approximation. The key observation is that the sequence ofheights of subtrees corresponding to segments $S_i$of next LZ-factor is bitonic [18].We can split this sequence into two subsequences: height-nondecreasing sequenceR_1 , R_2 ,…, R_k, called ***right-sided***, and height-nonincreasing sequence L_1; L_2 ,…, L_r, called ***left-sided***.

4.2 Charikar's Algorithm in details :

Charikar et al. [1] followed more or less the same path, with a different condition imposed on the grammar: it was required that its derivation tree is length-balanced, i.e. for a rule $X \rightarrow Y Z$ the lengths of words generated by Y and Z are within a certain multiplicative constant factor from each other. For such trees efficient implementation of merging, splitting etc. operations were given (i.e. constructed from scratch) by the authors and so the same running time as in the case of the AVL trees was obtained.

4.2.1 An *$O(\log^3 N)$* Approximation Algorithm: [1]

To begin, we describe a useful grammar construction, proveone lemma, and cite an old result that we shall use later.

The substring construction generates a set of grammar rulesenabling each substring of a string $\eta = x_1 \ldots x_p$ to be expressedwith at most two symbols.

The construction works as follows. First, create a nonterminalfor each suffix of the string $x_1 \ldots x_k$and each prefix of $x_{k+1} \ldots x_p$, where $k = \lceil(p/2)\rceil$. Note that each such nonterminalcan be defined using only two symbols: the nonterminalfor the next shorter suffix or prefix together with one symbolx_i. Repeat this construction recursively on the two halves of theoriginal string $x_1 \ldots x_k$ and $x_{k+1} \ldots x_p$.The recursion terminateswhen a string of length one is obtained. This recursion has*$\log p$*levels, and nonterminals are defined at each level. Sinceeach definition contains at most two symbols, the total cost ofthe construction is at most$2p \log p$.

Now we show that every substring $\alpha = x_i \ldots x_j$ of η is equalto $< AB >$, where $A$and $B$are nonterminals defined in the construction.There are two cases to consider. If αappears entirelywithin the left-half of or entirely within the right-half, then wecan obtain $A$and B from the recursive construction on $x_1 \ldots x_k$ and $x_{k+1} \ldots x_p$. Otherwise, let $k = \lceil(p/2)\rceil$as before, and let$A$be the nonterminal for $x_i \ldots x_k$, and let Bbe the nonterminalfor $x_{k+1} \ldots x_j$.

4.2.2 Charikar's Appoximate Algorithm:

In this algorithm [1] , the focus is on certain sequences ofsubstrings of α . In particular, we construct $\log n$ sequencesC_n , $C_{n/2}$, $C_{n/4}$, C_2, where the sequence $C_k$consists ofsome substrings of σ that have length at most k . These sequencesare defined as follows. The sequence $C_n$is initializedto consist of only the string σ itself. In general, the sequenceCkgenerates the sequence $C_{k/2}$via the following operations,which are illustrated in **Fig. 1.**

1) Use the greedy -approximation algorithm of Blum et al.to form a superstring $\rho_k$containing all the distinct strings in C_k .
2) Cut the super string ρ_k into small pieces. First, determinewhere each string in C_k nded up inside ρ_k , and then cut $\rho_k$at the left endpoints of those strings.
3) Cut each piece of ρ_k that has length greater than $k/2$ atthe midpoint. During the analysis, we shall refer to thecuts made during this step as extra cuts.

The sequence $C_{k/2}$ is defined to be the sequence of pieces of $\rho_k$generated by this three-step process. By the nature of Blum'salgorithm, no piece of $\rho_k$can have length greater than k afterstep 2), and so no piece can have length greater than $k/2$ afterstep 3). Thus, $C_{k/2}$is a sequence of substrings of that havelength at most $k/2$ as desired.

Now we translate these sequences of strings into a grammar.To begin, associate a nonterminal with each string in each sequenceC_k. In particular, the nonterminal associated with thesingle string in C_n(which is σ itself) is the start symbol of thegrammar.

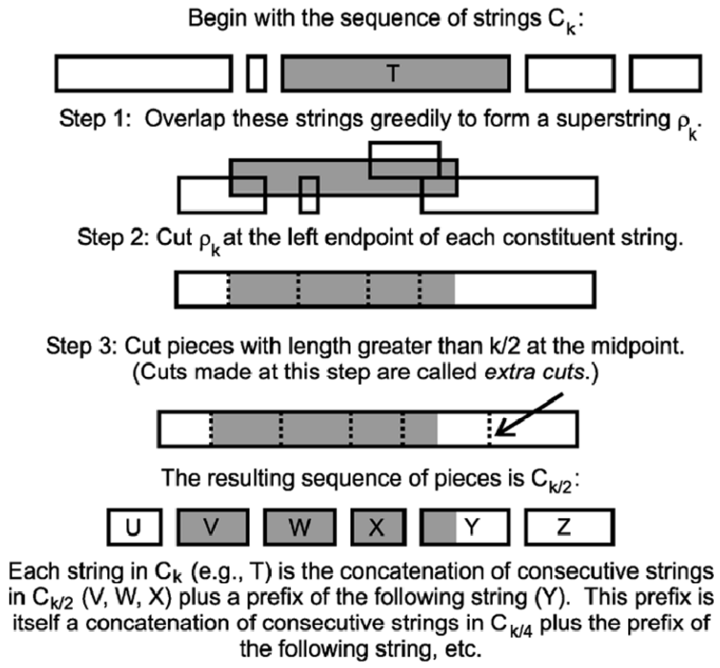
Begin with the sequence of strings C_k:
T
Step 1: Overlap these strings greedily to form a superstring ρ_k.
Step 2: Cut ρ_k at the left endpoint of each constituent string.
Step 3: Cut pieces with length greater than k/2 at the midpoint.
(Cuts made at this step are called *extra cuts*.)
The resulting sequence of pieces is $C_{k/2}$:
U
V
W
X
Y
Z
Each string in C_k (e.g., T) is the concatenation of consecutive strings in $C_{k/2}$ (V, W, X) plus a prefix of the following string (Y). This prefix is itself a concatenation of consecutive strings in $C_{k/4}$ plus the prefix of the following string, etc.

Figure : 2

All that remains is to define these nonterminals. In doing so,the following observation is key: each string in $C_k$is the concatenationof several consecutive strings in $C_{k/2}$together with aprefix of the next string in $C_{k/2}$. This is illustrated in the figureabove, where the fate of one string in C_k(shaded and marked T) is traced through the construction of $C_{k/2}$. In this case, T isthe concatenation of V, W, X , and a prefix of Y . Similarly, theprefix of Y is itself the concatenation of consecutive strings in $C_{k/4}$ together with a prefix of the next string in $C_{k/4}$. This prefixis, in turn, the concatenation of consecutive strings in $C_{k/8}$ togetherwith a prefix of the next string in $C_{k/8}$, etc. As a result,we can define the nonterminal corresponding to a string in $C_k$as a sequence of consecutive nonterminals from $C_{k/2}$, followedby consecutive nonterminals from $C_{k/4}$, followed by consecutivenonterminals from $C_{k/8}$, etc. For example, the definitionof Twould begin $T \rightarrow V W X \ldots$and then contain sequencesof consecutive nonterminals from $C_{k/4}$, $C_{k/8}$, etc. As a specialcase, the nonterminals corresponding to strings in $C_k$can be definedin terms of terminals.

We can use the substring construction to make these definitionsshorter and hence the overall size of the grammar smaller.In particular, for each sequence of strings C_k , we apply the substringconstruction on the corresponding sequence of nonterminals.This enables us to express any sequence of consecutivenonterminals using just two symbols. As a result, we can defineeach nonterminal corresponding to a string in C_k using only twosymbols that represent a sequence of consecutive nonterminalsfrom $C_{k/2}$, two more that represent a sequence of consecutivenonterminals from $C_{k/4}$, etc. Thus, every nonterminal can nowbe defined with *O(log n)*symbols on the right.

4.3 Sakamoto's Algorithm in details :

Lastly, **Sakamoto** [19] proposed a different algorithm, based on RePair [12], which is one of the practically implemented and used algorithms for grammar-based compression. His algorithm iteratively replaced pairs of different letters and maxi-mal blocks of letters (a^l is a maximal block if it cannot be extended by*a* to either side). A special pairing of the letters was devised, so that it is 'synchronizing': if w has 2 disjoint occurrences in the text, then those two occurrences can be represented as w1 w′w2, where w1, w2 = O(1), such that both occurrences of w′in text are paired and compressed in the same way. The analysis was based on considering the LZ77 representation of the text and proving that due to 'synchronization' the factors of LZ77 are compressed very similarly as the text to which they refer.

4.3.1 Sakamoto's Approximation algorithm: [19]

The approximation algorithm **LEVELWISE-REPAIR**for the grammar-based Compression is presented. This algorithm calls two procedures repetition(,) and arrangement(,) .

Outline of the algorithm

The algorithm contains two procedures ***repetition*** and ***arrangement***. They are calledby the algorithm for each execution of the outer-loop.

The task of ***repetition*** is to replaceany repetition $w[i, j] = a^+$ in the input string by an appropriate nonterminal. More precisely,if an input string contains a repetition $w[i, j] = a^k$, then w[i, j] is replaced bya nonterminal $A_{(a,k)}$and the production $A_{(a,k)} \rightarrow B_{(a,k)}C_{(a,k)}$is defined. The nonterminals$B_{(a,k)}$,$C_{(a,k)}$ and their productions are also defined recursively depending on k;$B_{(a,k)} = C_{(a,k)} = A_{(a,k/2)}$if k is even and $B_{(a,k)}A_{(a,k-1)}$and $C_{(a,k)} = a$ otherwise.

On the other hand, the task of ***arrangement*** is to decide whether the algorithm replacea segment w[i, i + 1] = ab by a nonterminal for each pair ab $\in \Sigma^2$, where $a \neq b$. Thisprocess is executed in the frequent order of all pairs stored in a priority queue indicated bylist in line 3 of Fig. 3. This order is fixed until all elements are popped according to the
following process.

We next briefly explain the task of arrangement. The complete description and an exampleare shown in the next subsection. Taking a most frequent pair ab from the priorityqueue and a unique index $id^{ab} = \{d_1^{ab}, d_2^{ab}\}$ is set for ab, where the index is simply denotedby id = $\{d_1, d_2\}$ if it is not necessary to indicate the pair. Let S be the set of segmentsw[i, i + 1] such that w[i, i + 1] = ab. The task is to assign either$d_1$or $d_2$to each s $\in$ S.Such an index is used to decide the replacement of the adjoining segment of *s* . Similarly,the replacement of s itself is decided by the index of its adjoining segment, which is alreadyassigned. After the set $S' \subseteq S$ of segments to be replaced is decided, arrangementcreates an appropriate nonterminal A and the production $A \rightarrow ab$.

After all pairs are popped from the priority queue, the algorithm actually replaces all thesegments by their corresponding nonterminals. The obtained string *w*is given to the algo rithm as a next input and the two procedures are executed for *w*.The algorithm continuesthis process until there is no more pair ab such that $\#(ab,w) \geq 2$ [19].

However **Autur Jez** found that the presented analysis [19] is incomplete, as the cost of nonterminals introduced when maximal blocks are replaced is not bounded at; the bound that Jez was able to obtain using the approach of Sakamoto is $O(\log(N/g)^2)$, so worse than claimed [4].

4.3.2 Comparison with Sakamoto's algorithm :

The general approach is similar to Sakamoto's method but there are separate analyses and estimations for (variants of) pair compression and block compression. However, the pairing of letters seems more natural here and the analysis is simpler. Also, the construction of nonterminals for blocks of letters is different. Note, that the analysis for block compression mentioned here is much more involved than the one for *pair compression*. On the other hand, the connection to the addition chains suggests that the compression of blocks is the difficult part of the smallest grammarproblem.

CHAPTER : 5
CONCLUSION

5.1 Advantages and disadvantages of the proposed technique :

The proposed algorithm is interesting, as it is very simple and its analysis for the first time does not rely on LZ77 representation of the string. Potentially this can help in both design of an algorithm with a better approximation ratio and in showing a logarithmic lower bound: Observe that LZ77 representation is known to be at most as large as the smallest grammar, so it might be that some algorithm produces a grammar of size $o(g\log(N/g))$, even though this is of size$\Omega(l\log(N/l))$, where$l$is the size of the LZ77 representation of the string. Secondly, as the analysis 'considers' the optimal grammar, it may be much easier to observe, where every approximation algorithm performs badly, and so try to approach a logarithmic lower bound. This is much harder to imagine, when the approximation analysis is done in terms of the LZ77.

Unfortunately, the obtained grammar is not balanced in any sense, in fact it is easy to give examples on which it returns grammar of height $\Omega(\sqrt{N})$(note though that the same applies also to grammar returned by Sakamoto's algorithm). This makes the obtained grammar less suitable for later processing; on the other hand, the practically used grammar-based compressors [12,11,16]also do not produce a balanced grammar, nor do they give a guarantee on its height.

On the good side, there is no reason why the optimal grammar should be balanced, neither can we expect that for an unbalanced grammar a small balanced one exists. Thus it is possible that while $o(\log(N/g))$ approximation algorithm exists, there is no such an algorithm that always returns a balanced grammar.

We note that the reason why the grammar returned by the proposed algorithm can have large height is only due to block compression: if we assume that the nonterminal generating a_has height one, the whole grammar has height $O(\log N)$. It looks reasonable to assume that many data structures for grammar representation of text as well as later processing of it can indeed process a production $a_l \to a^l$ in constant time.

Lastly, the proposed method seems to be much easier to generalize then the LZ77-based ones: generalizations of SLPs to grammars generating other objects (mostly: trees) are known but it seems that LZ77-based approach does not generalize to such settings, as LZ77 ignores any additional structure (like: tree-structure) of the data. In recent work of Lohrey and the author the algorithm presented in this paper is generalized to the case of tree-grammars, yielding a first provable approximation for the smallest tree grammar problem [9].

5.2 Note on computational model :

The presented algorithm runs in linear time, assuming that the Σ can be identified with a continuous subset of natural numbers of size O(N^C)for some constant *c* and the RadixSort can be performed on it. Should this not be the case for the input, we can replace the original letters with such a subset, in O($n \log |\Sigma|$)time (by creating a balanced tree for letters occurring in the input string).

REFERENCES

[1]M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, A. Shelat, The smallest grammar problem, IEEE Trans. Inform. Theory 51(7) (2005) 2554–2576.

[2]P. Gawrychowski, Pattern matching in Lempel–Ziv compressed strings: fast, simple, and deterministic, in: C. Demetrescu, M.M. Halldórsson (Eds.), ESA, in: LNCS, vol.6942, Springer, 2011.

[3] L. Gasieniec, M. Karpinski, W. Plandowski, W. Rytter, Efficient algorithms for Lempel–Ziv encoding, in: R.G. Karlsson, A. Lingas (Eds.), SWAT, in: LNCS, vol.1097, Springer, 1996.

[4] A. Jez, Approximation of grammar-based compression via recompression, in: J. Fischer, P. Sanders (Eds.), CPM, in: LNCS, vol.7922, Springer, 2013, full version available at http://arxiv.org/abs/1301.5842.

[5]A. Jez, Recompression: a simple and powerful technique for word equations, in: N. Portier, T. Wilke (Eds.), STACS, in: LIPIcs, vol.20, SchlossDagstuhl–Leibniz ZentrumfuerInformatik, Dagstuhl, Germany, 2013, full version available at http://arxiv.org/abs/1203.3705, accepted to J. ACM, http://drops.dagstuhl.de/opus/volltexte/2013/3937.

[6] A. Jez, The complexity of compressed membership problems for finite automata, Theory Comput. Syst. 55 (2014) 685–718, http://dx.doi.org/10.1007/s00224-013-9443-6.

[7] A. Jez, One-variable word equations in linear time, Algorithmica (2015), http://dx.doi.org/10.1007/s00453-014-9931-3, in press.

[8]A. Jez, Faster fully compressed pattern matching by recompression, ACM Trans. Algorithms 11(3) (2015) 20:1–20:43, http://doi.acm.org/10.1145/2631920.

[9] A. Jez, M. Lohrey, Approximation of smallest linear tree grammar, in: E.W. Mayr, N. Portier (Eds.), STACS, in: LIPIcs, vol.25, SchlossDagstuhl–Leibniz-ZentrumfuerInformatik, 2014.

[10] M. Karpinski, W. Rytter, A. Shinohara, Pattern-matching for strings with short descriptions, in: CPM, 1995.

[11]J.C. Kieffer, E.-H. Yang, Sequential codes, lossless compression of individual sequences, and Kolmogorov complexity, IEEE Trans. Inform. Theory 42(1) (1996) 29–39.

[12]N.J. Larsson, A. Moffat, Offline dictionary-based compression, in: Data Compression Conference, IEEE Computer Society, 1999.

[13]M. Lohrey, Algorithmics on SLP-compressed strings: a survey, Groups Complex. Cryptol. 4(2) (2012) 241–299.

[14]K. Mehlhorn, R. Sundar, C. Uhrig, Maintaining dynamic sequences under equality tests in polylogarithmic time, Algorithmica 17(2) (1997) 183–198.

[15]M. Mitzenmacher, E. Upfal, Probability and Computing: Randomized Algorithms and Probabilistic Analysis, Cambridge University Press, 2005.

[16]C.G. Nevill-Manning, I.H. Witten, Identifying hierarchical structurein sequences: a linear-time algorithm, J. Artificial Intelligence Res. 7 (1997) 67–82.

[17]W. Plandowski, Testing equivalence of morphisms on context-free languages, in: J. van Leeuwen (Ed.), ESA, in: LNCS, vol.855, Springer, 1994.

[18]W. Rytter, Application of Lempel–Ziv factorization to the approximation of grammar-based compression, Theoret. Comput. Sci. 302(1–3) (2003) 211–222.

[19]H. Sakamoto, A fully linear-time approximation algorithm for grammar-based compression, J. Discrete Algorithms 3(2–4) (2005) 416–430.

[20]J.A. Storer, T.G. Szymanski, The macro model for data compression, in: R.J. Lipton, W.A. Burkhard, W.J. Savitch, E.P. Friedman, A.V. Aho (Eds.), STOC, ACM, 1978.

[21]A.C.-C. Yao, On the evaluation of powers, SIAM J. Comput. 5(1) (1976) 100–103.

[22] Eric Lehman, AbhiShelat, Approximation Algorithms for Grammar-Based Compression, SODA '02 Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms , Pages 205-212

[23] Guy E. Blelloch, Computer Science Department, Carnegie Mellon University, Introduction to Data Compression, Algorithms in the real world, 2000

[24] Hiroshi Sakamoto, Grammar Compression: Grammatical Inference by Compression and Its Application to Real Data, Proceedings of the 12th ICGI, JMLR: Workshop and Conference Proceedings 34: 3-20, 2014

[25] Shmuel T. Klein, Efficient recompression techniques for dynamic full-text retrieval systems, SIGIR '95 Proceedings of the 18th annual international ACM SIGIR conference on Research and development in information retrieval, Pages 39-47

[26] C. de Marcken. The Unsupervised Acquisition of aLexicon from Continuous Speech.MIT AI Memo 1558.November 1995.

[27] J. C. Kieffer, E. Yang, G. J. Nelson, P. Cosman.Universal Lossless Compression via Multilevel PatternMatching. IEEE Transactions on Information Theory,vol. 46 (2000), pp. 1227-1245.

[28] A. Apostolico and S. Lonardi. Some Theory and Practice of Greedy O_-Line Textual Substitution. DCC 1998, pp 119-128.

[29] J. Ziv and A. Lempel.A Universal Algorithm for Sequential Data Compression. IEEE Transactions onInformation Theory, vol. 23 (1977), pp. 337-343.

[30] J. Ziv and A. Lempel.Compression of Individual Sequences via Variable-Rate Coding. IEEE Transactions on Information Theory, vol. 24 (1978), pp. 530-536.

[31] C. Nevill-Manning.Inferring Sequential Structure.PhD thesis, University of Waikato, 1996.

[32] E. H. Yang and J. C. Kieffer, “Efficient universal lossless data compressionalgorithms based on a greedy sequential grammar transform—Partone: Without context models,” IEEE Trans. Inf. Theory, vol. 46, no. 3,pp. 755–777, May 2000.

[33]J. C. Kieffer and E. H. Yang, “Grammar based codes: A new class ofuniversal lossless source codes,” IEEE Trans. Inf. Theory, vol. 46, no. 3,pp. 737–754, May 2000.

[34] G. Nelson, J. C. Kieffer, and P. C. Cosman, “An interesting hierarchicallossless data compression algorithm,” in Proc. IEEE Information TheorySociety Workshop, Rydzyna, Poland, Jun. 1995. Invited Presentation.

[35] R. E. Bryant, “Graph-based algorithms for Boolean function manipulation,”IEEE Trans. Comput., vol. C-35, no. 8, pp. 677–691, Aug. 1986.

[36] J. C. Kieffer, P. Flajolet, and E.-H. Yang, “Data compression via binarydecision diagrams,” in IEEE Int. Symp. Information Theory, vol. 46, Jun.2000, p. 296.

[37] C.-H. Lai and T.-F. Chen, “Compressing inverted files in scalable informationsystems by binary decision diagram encoding,” in Proc. 2001Conf. Supercomputing, Denver, CO, Nov. 2001, p. 60.

[38] T. A. Welch, “A technique for high-performance data compression,”Computer Mag. Computer Group News of the IEEE Computer GroupSoc., vol. 17, no. 6, pp. 8–19, 1984.

[39] M. Crochemore, W. Rytter, Text Algorithms, Oxford University Press, New York, 1994.

[40] http://brasil.cel.agh.edu.pl/~11sustrojny/en/formal-grammar/index.html