Grammar-based Compression

1. Introduction

Large-scale, highly repetitive data collections are becoming more and more ubiquitous. Examples are sequences of individual human genomes, source code in repositories and version-controlled documents. In the ever-expanding landscape of data generation and transmission, the pursuit of efficient storage and communication methods has become paramount. One intriguing avenue within the realm of data compression is that of grammar compression.

Grammar compression is a technique to represent a string or a data structure by a context-free grammar that generates only that object. Grammar compression can achieve exponential compression and with additional existing algorithms on compressed data has many applications in bioinformatics, data networks, data mining, pattern recognition, and more. The main goal of grammar compression is to find the smallest grammar that represents a given string, which is known as the smallest grammar problem. This problem is NP-hard, and therefore, various approximation algorithms have been proposed to solve it efficiently.

Grammar compression consists of two phases: grammar construction and grammar encoding. In the grammar construction phase, the input string is transformed into a context-free grammar. In the grammar encoding phase, the grammar is encoded into a succinct data structure, which uses space close to the information-theoretic lower bound and supports efficient operations. The combination of these two phases results in a compressed representation of the input string that can be manipulated and decompressed easily.

In this paper, we focus on two variants of fully online grammar compression algorithms, FOLCA and SOLCA, which build an SLP and directly encode it into a succinct representation in an online manner. FOLCA and SOLCA work in constant space and linear time, which makes them suitable for large-scale, noisy repetitive texts. We explain how they work, compare them with other grammar-based compression algorithms, and show some experimental results and evaluations on different datasets.

2. Basic concepts

Firstly, we introduce some of the basic concepts and building blocks that are used in grammar-based compression. We assume that the reader is familiar with the notions of strings, alphabets, languages, and grammars.

Definition 1. Context-free grammar (CFG). A context-free grammar (CFG) is a 4-tuple $G = (V, \Sigma, S, R)$, where V is a finite set of variables (or nonterminals), Σ is a finite set of terminals, $S \in V$ is the start variable, and R is a finite set of production rules. Each production rule has the form $A \rightarrow \alpha$, where $A \in V$ and $\alpha \in (V \cup \Sigma)^*$. A CFG can generate a language $L(G) \subseteq \Sigma$ by applying the production rules starting from the start variable, until no more variables are left.

Definition 2. Chomsky normal form. A context-free grammar $G = (V, \Sigma, P, S)$ is in Chomsky normal form if every production in P is of one of the following forms:

$A \rightarrow BC$, where $A, B, C \in V$ and $B, C \neq S$

$A \rightarrow a$, where $A \in V$ and $a \in \Sigma$

S → ε, where ε is the empty string

Definition 3. Straight line program (SLP). A straight-line program (SLP) is a special kind of CFG in Chomsky normal form that generates exactly one string. An SLP can be seen as a sequence of expressions, where each expression is either a terminal, or the concatenation of two previous expressions. The last expression in the sequence is the output of the SLP. An SLP can be used to represent the structure and patterns of a string in a compact way. SLP has a unique derivation tree, which is a balanced binary tree.

Definition 4. Ordered directed acyclic graph (ODAG). An ordered directed acyclic graph (ODAG) is a directed acyclic graph (DAG) that has a total order on its vertices. The order is defined such that for any two vertices u and v, u < v if and only if u is reachable from v by following the edges in the DAG. An ODAG can be used to encode an SLP, by assigning each vertex a label that is either a terminal, or the index of two previous vertices in the order. The last vertex in the order is the root of the ODAG, and its label is the output of the corresponding SLP. An ODAG can be encoded into a succinct data structure, which uses space close to the information-theoretic lower bound and supports efficient operations. For example, the string ababab can be represented by the following SLP and ODAG:

SLP: A -> a, B -> b, C -> AB, D -> CC, E -> CD

ODAG:

> The ODAG has five vertices, labeled from 1 to 5. The order is defined by the topological sort of the DAG, which is 1, 2, 3, 4, 5. The labels of the vertices are either terminals (a or b) or the indices of two previous vertices (3, 4, or 5). The last vertex (5) is the root of the ODAG, and its label (CD) is the output of the corresponding SLP. The ODAG can be encoded into a succinct data structure, such as a balanced parenthesis sequence, which uses space close to the information-theoretic lower bound and supports efficient operations.

Definition 5. Longest Common Ancestor (LCA). The longest common ancestor (LCA) of two nodes in a tree or a DAG is the lowest node that has both nodes as descendants. In a DAG, the LCA is well-defined if there is a unique path from the root to each node. The LCA can be used to measure the similarity or distance between two nodes, or between two strings represented by SLPs or ODAGs. For example, the LCA of two nodes in an ODAG can be used to find the longest common substring of the corresponding strings, or the edit distance between them. The LCA problem is to find the LCA of two given nodes in a tree or a DAG. There are efficient algorithms and data structures for solving the LCA problem in various settings.

3. Basic theory

Let's review some of the basic theory behind grammar-based compression, and how it relates to information theory, complexity theory, and succinct data structures. We also introduce the smallest grammar problem, which is the core of grammar-based compression, and some of the approximation algorithms for it.

3.1 Information theory and grammar compression

Information theory is the study of the quantification, storage, and communication of information. One of the central concepts in information theory is entropy, which measures the amount of uncertainty or randomness in a source of information. Entropy can be used to define the optimal compression rate for a source, which is the minimum number of bits per symbol needed to encode the source without loss of information. The optimal compression rate is also known as the information-theoretic lower bound, and it can be achieved by using a prefix code that assigns codewords to symbols according to their probabilities.

Grammar-based compression can be seen as a way of exploiting the structure and regularity of a source to achieve compression rates close to the information-theoretic lower bound. By representing a source as a context-free grammar, grammar-based compression can capture the patterns and substructures of the source and reduce the redundancy and repetition in the source. The size of the grammar can be used as a measure of the complexity or information content of the source, and the goal of grammar-based compression is to minimize the size of the grammar while preserving the information of the source.

3.2 Smallest grammar problem and NP-hardness

The smallest grammar problem is to find the smallest context-free grammar that generates exactly one given string. The size of the grammar can be defined as the total number of symbols (terminals and nonterminals) in the production rules of the grammar. The smallest grammar problem is the core of grammar-based compression, as it aims to optimize the compression ratio of the grammar.

The smallest grammar problem is known to be NP-hard, which means that there is no efficient algorithm that can solve it exactly in polynomial time, unless P = NP. The NP-hardness of the smallest grammar problem can be shown by reducing it from other NP-hard problems, such as the shortest common super sequence problem or the minimum set cover problem.

3.3 Approximation algorithms for the smallest grammar problem

Since the smallest grammar problem is NP-hard, various approximation algorithms have been proposed to solve it efficiently. An approximation algorithm is an algorithm that produces a solution that is close to the optimal solution, within some guaranteed factor or bound. The approximation ratio of an algorithm is the worst-case ratio between the size of the grammar produced by the algorithm and the size of the optimal grammar for any input string.

Some of the most popular and widely used approximation algorithms for the smallest grammar problem are based on the idea of finding and replacing repeated substrings in the input string. These algorithms include Ziv-Lempel, Re-Pair, Sequitur, LZW, FOLCA, and SOLCA. These algorithms have different trade-offs between speed, space, and quality, and different approximation ratios. For example, Ziv-Lempel has an approximation ratio of O (log n), where n is the length of the input string, and runs in linear time and space. Re-Pair has an approximation ratio of O (log^2 n) and runs in quadratic time and linear space. Sequitur has an approximation ratio of O (log n) and runs in linear time and space, but with a large constant factor. LZW has an approximation ratio of O (log n), and runs in linear time and space, but requires an additional encoding step. FOLCA and SOLCA have an approximation ratio of O (log n), and run in linear time and constant space, but use a balanced binary tree and a succinct data structure to store the grammar.

3.4 Succinct data structures and grammar compression

A succinct data structure is a data structure that uses space close to the information-theoretic lower bound and supports efficient operations. A succinct data structure can be used to encode a grammar or a string in a compact and query-able way. For example, a balanced parenthesis sequence can be used to encode a balanced binary tree, which can represent an SLP or an ODAG. A balanced parenthesis sequence can be encoded into a bit vector, which uses 2n + o (n) bits for n nodes, and supports operations such as finding the parent, child, sibling, or ancestor of a node in constant time.

Succinct data structures can also be used to support direct operations over the compressed data, such as pattern matching, similarity search, random access, and decompression, without requiring full decompression. For example, a wavelet tree can be used to encode a string over an alphabet of size k, which uses n log k + o (n log k) bits for n symbols, and supports operations such as rank, select, access, and range queries in O (log k) time.

4. Existing algorithms

Now analyze some of the existing algorithms for grammar-based compression, and compare their performance, complexity, and advantages. Algorithms are divided into two categories: offline and online. Offline algorithms require the whole input to be available in memory before processing, while online algorithms can process the input as a stream without storing it. Mail focus is on FOLCA, which is a variant of online grammar compression that works in constant space and linear time.

4.1 Offline algorithms

Offline algorithms for grammar-based compression are algorithms that need to read and store the entire input string before constructing and encoding the grammar. Some of the most popular and widely used offline algorithms are Ziv-Lempel, Re-Pair, Sequitur, and LZW. These algorithms are based on the idea of finding and replacing repeated substrings in the input string and building a context-free grammar or a straight-line program (SLP) that generates the input string. These algorithms have different trade-offs between speed, space, and quality, and different approximation ratios for the smallest grammar problem. For example, Ziv-Lempel has an approximation ratio of O (log n), where n is the length of the input string, and runs in linear time and space.

4.2 Online algorithms

Online algorithms for grammar-based compression are algorithms that can process the input string as a stream, without storing the whole input in memory. Online algorithms are suitable for large-scale, noisy, or dynamic data, where the input size is unknown or changing. Online algorithms can also support direct operations over the compressed data, such as pattern matching, similarity search, random access, and decompression, without requiring full decompression. Some of the online algorithms for grammar-based compression are based on the Edit-Sensitive Parsing (ESP) technique, which partitions the input into substrings of different types and parses them using binary trees. These algorithms include FOLCA and SOLCA, which are variants of fully online grammar compression that work in constant space and linear time.

4.3. Lempel–Ziv–Welch (LZW)

LZW is a universal lossless data compression algorithm that is simple to implement and has the potential for very high throughput in hardware implementations. It is the algorithm of the Unix file compression utility compress and is used in the GIF image format.

The algorithm works by constructing a dictionary of substrings, which we will call "phrases," that have appeared in the text. The idea is to parse the sequence into distinct phrases. The version we analyze does this greedily. Suppose, for example, we have the string

AABABBBABAABABBBABBABB

We start with the shortest phrase on the left that we haven't seen before. This will always be a single letter, in this case A:

A|ABABBBABAABABBBABBABB

We now take the next phrase we haven't seen. We've already seen A, so we take AB:

A|AB|ABBBABAABABBBABBABB

The next phrase we haven't seen is ABB, as we've already seen AB. Continuing, we get B after that:

A|AB|ABB|B|ABAABABBBABBABB

and you can check that the rest of the string parses into

A|AB|ABB|B|ABA|ABAB|BB|ABBA|BB

Because we've run out of letters, the last phrase on the end is a repeated one. That's O.K. Now, how do we encode this? For each phrase we see, we stick it in a dictionary. The next time we want to send it, we don't send the entire phrase, but just the number of this phrase. Consider the following table

1	2	3	4	5	6	7	8	9
A	AB	ABB	B	ABA	ABAB	BB	ABBA	BB
ØA	1B	2B	ØB	2A	5B	4B	3A	7

The first row gives the numbers of the phrase, the second row gives the phrases, and the third row their encodings. The last piece is encoding this string into binary. This gives

001110100101001011100101100111

To see how this works, I've now inserted dividers and commas, to make it more comprehensible)

0, 0|1, 1|10, 1|00, 1|010, 0|101, 1|100, 1|011, 0|0111

We have taken the third row of the previous array, expressed all the numbers in binary (before the comma) and the letters in binary (after the comma). Note that I've mapped A to 0 and B to 1. If you had a larger alphabet, you would encode the letters by more than one bit.

To decode, the decoder needs to construct the same dictionary. To do this, he first takes the binary string he receives, and inserts dividers and commas. This is straightforward. The first two dividers each come after 2 bits. The next two each come after 3 bits. We then get 2^2 of length 4 bits, 2^3 of length 5 bits, 2^4 of length 6 bits, and in general 2^k of length k + 2 bits. This is because when we encode our phrases, if we have r phrases in our dictionary, we use $\log_2 r$ bits to encode the number of the phrase to ensure that the encodings of all phrases use the same number of bits. The decoder then uses the same algorithm to construct the dictionary as the encoder did. He knows phrases 1 through r − 1 when he is trying to figure out what the r-th phrase is, and this is exactly the information he might need to reconstruct the dictionary.

So how well does the Lempel-Ziv algorithm work? We will calculate how well it works in the worst case. The compression is asymptotically optimal. That is, in the worst case, the length of the encoded string of bits is n + o(n). Since there is no way to compress all length-n strings to fewer than n bits, this can be counted as asymptotically optimal. In the second case, the source is compressed to length

$$H(p_1, p_2, . . . , p_\alpha)n + o(n) = n \sum(\alpha, i=1) (-p_i \log_2 p_i) + o(n),$$

which is to first order the Shannon bound. Let’s do the worst case analysis first. Suppose we are compressing a binary alphabet. We ask the question: what is the maximum number of distinct phrases that a string of length n can be parsed into. There are some strings which are clearly worst case strings. These are the ones in which the phrases are all possible strings of length at most k. For example, for k = 1, one of these strings is

$$0|1$$

with length 2. For k = 2, one of them is

$$0|1|00|01|10|11$$

with length 10; and for k = 3, one of them is

$$0|1|00|01|10|11|000|001|010|011|100|101|110|111$$

with length 34. In general, the length of such a string is

$$n_k = \sum(k, j=1)\ j2^j$$

since it contains 2^j phrases of length j. It is easy to check that

$$n_k = (k - 1)2^{k+1} + 2$$

by induction. If we let $c(n_k)$ be the number of distinct phrases in this string of length n_k, we get that

$$c(n_k) = \sum(k, i=1)\ 2^i = 2\ k+1 - 2$$

For n_k, we thus have

$$c(n_k) = 2^{k+1} - 2 \leq ((k - 1)\ 2^{k+1}) / (k - 1) \leq n_k / k - 1$$

Now, for an arbitrary length n, we can write $n = n_k + \Delta$. To get the case where c(n) is largest, the first n_k bits can be parsed into $c(n_k)$ distinct phrases, containing all phrases of length at most k, and the remaining Δ bits can be parsed into into phrases of length k +1. This is clearly the most distinct phrases a string of length n can be parsed into, so we have that for a general string of length n, the number of phrases is at most total is

$$c(n) \le \frac{n_k}{k-1} + \frac{\Delta}{k+1} \le \frac{n_k + \Delta}{k-1} = \frac{n}{k-1} \le \frac{n}{\log_2 c(n) - 3}$$

Now, we have that a general bit string is compressed to around c(n) $\log_2$ c(n)+ c(n) bits, and if we substitute

$$c(n) \le \frac{n}{\log_2 c(n) - 3}$$

we get

$$c(n) \log_2 c(n) + c(n) \le n + 4c(n) = n + O(\frac{n}{\log_2 n})$$

So asymptotically, we don't use much more than n bits for compressing any string of length n. This is good: it means that the Lempel-Ziv algorithm doesn't expand any string by very much. We can't hope for anything more from a general compression algorithm, as it is impossible to compress all strings of length n into fewer than n bits. So if a lossless compression algorithm compresses some strings to fewer than n bits, it will have to expand other strings to more than n bits.

4.4. FOLCA

FOLCA stands for Fully Online LCA, which is a variant of online grammar compression that builds an SLP and directly encodes it into a succinct representation in an online manner. FOLCA works in constant space and linear time, which makes it suitable for large-scale, noisy repetitive texts. FOLCA uses a balanced binary tree to store the SLP and a hash table to store the pairs of non-terminals that appear consecutively in the input stream. FOLCA also uses LCA queries to find the longest repeated substring in the current input and replace it with a new non-terminal. FOLCA achieves a good approximation ratio of O($\log_2$ N) to the smallest grammar compression for an input of length N, while using only O(N log N) bits of space. FOLCA can also support various applications that require efficient processing of highly repetitive texts, such as pattern matching, edit distance computation, q-gram mining, and characteristic substring mining.

FOLCA uses a bottom-up approach to construct a parse tree corresponding to an SLP, and encodes it into two bit arrays: B and L. B is a balanced parenthesis sequence that represents the structure of the parse tree, and L is a label sequence that stores the symbols at the leaves of the parse tree.

The compression process of FOLCA consists of the following steps:

- Initialize an empty hash table H and two empty queues Q1 and Q2.
- Read a character c from the input text and enqueue it to Q1.

- If Q1 contains two symbols x and y, check if H contains an entry for xy. If yes, let z be the non-terminal symbol associated with xy in H. If no, create a new non-terminal symbol z and add an entry for xy -> z in H.
- Dequeue x and y from Q1 and enqueue z to Q2.
- Repeat steps 2 to 4 until Q1 is empty or contains only one symbol.
- If Q2 contains more than one symbol, copy its contents to Q1 and clear Q2. Then go back to step 2.
- If Q2 contains only one symbol, it is the root of the parse tree. Output the bit arrays B and L as follows:
    - For each symbol in Q1 and Q2, append a 0 to B and the symbol to L.
    - For each non-terminal symbol in H, append a 10 to B and the symbol to L.

For example, suppose the input text is "abababa". The compression process of FOLCA is shown below:

Step	Q1	Q2	H	B	L
1	a				
2	a, b				
3		X1	ab -> X1		
4	a	X1	ab -> X1		
5	a, b	X1	ab -> X1		
6		X1, X2	ab -> X1, X1a -> X2		
7	X1	X2	ab -> X1, X1a -> X2		
8	X1, b	X2	ab -> X1, X1a -> X2		
9		X2, X3	ab -> X1, X1a -> X2, X1b -> X3		
10	X2	X3	ab -> X1, X1a -> X2, X1b -> X3		
11	X2, a	X3	ab -> X1, X1a -> X2, X1b -> X3		
12		X3, X4	ab -> X1, X1a -> X2, X1b -> X3, X2a -> X4		
13	X3	X4	ab -> X1, X1a -> X2, X1b -> X3, X2a -> X4		
14	X3, X4		ab -> X1, X1a -> X2, X1b -> X3, X2a -> X4		
15		X5	ab -> X1, X1a -> X2, X1b -> X3, X2a -> X4, X3X4 -> X5		
16			ab -> X1, X1a -> X2, X1b -> X3, X2a -> X4, X3X4 -> X5	0010101011	abaX1X2

The final output is B = 0010101011 and L = abaX1X2. This is a succinct representation of the SLP:

X1 -> ab X2 -> X1a X3 -> X1b X4 -> X2a X5 -> X3X4 which derives the original text “abababa”.

SOLCA

SOLCA (Space-Optimal LCA) is an advanced grammar-based compression algorithm that builds upon the foundation laid by FOLCA. While FOLCA achieves efficient online compression by constructing a Straight-Line Program (SLP) and encoding it into a succinct representation, SOLCA further enhances the process by optimizing space usage and improving performance through various innovative techniques.

Key Changes from FOLCA

1. Space Optimization
    a. Dynamic Succinct Data Structures: SOLCA employs dynamic succinct data structures to represent the parse tree more compactly than FOLCA. This includes a dynamic balanced parenthesis sequence and a dynamic label sequence that efficiently encode the structure and symbols of the parse tree.
    b. Reverse Dictionary: SOLCA introduces a reverse dictionary to facilitate quick lookups of phrases during the compression process. This dictionary allows the algorithm to efficiently determine whether a given pair of symbols has already been encountered and encoded
2. Improved Time Complexity
    a. Expected Time for Phrase Lookup: SOLCA achieves an expected time complexity of O(log(log(N))) for phrase lookups using the reverse dictionary, which is a significant improvement over the time complexity of FOLCA.
3. Constant Space Reverse Dictionary (CRD):
    a. Frequent Production Rules: SOLCA incorporates a constant space reverse dictionary (CRD) that stores frequent production rules. This CRD allows for constant expected time lookups for frequently occurring phrases, further optimizing the compression process.

These key differences were achieved by usage of proper data structures such as:

- Balanced Parenthesis Sequence:
    - A dynamic balanced parenthesis sequence B is used to represent the hierarchical structure of the parse tree. Each opening parenthesis corresponds to the beginning of a non-terminal, and each closing parenthesis corresponds to the end
- Label Sequence
    - A dynamic label sequence L stores the actual symbols at the leaves of the parse tree. Non-terminal symbols are stored as references to entries in the reverse dictionary
- Hash Table
    - Similar to FOLCA, SOLCA uses a hash table to store pairs of non-terminal symbols that appear consecutively in the input stream. This hash table facilitates quick lookups and insertions during the compression process.
- Reverse Dictionary:
    - The reverse dictionary is a key innovation in SOLCA. It maps pairs of symbols or non-terminals to their corresponding non-terminal representation in the grammar. This dictionary is dynamically updated as new pairs are encountered.

- Constant Space Reverse Dictionary (CRD):
    - The CRD is a specialized version of the reverse dictionary that uses constant space algorithms to store and lookup frequent items. This structure is particularly effective for handling repetitive texts where certain phrases occur frequently.

The compression process in SOLCA involves several steps, similar to FOLCA but with enhancements for space efficiency and speed:

1. Initialization:
    a. Initialize an empty hash table H, two empty queues Q1 and Q2, and an empty reverse dictionary RD .
2. Reading Input
    a. Read characters from the input text and enqueue them to Q1
3. Processing Pairs
    a. If Q1 contains two symbols x and y
        i. Check if the pair xy exists in RD. If yes, let z be the associated non-terminal symbol. If no, create a new non-terminal symbol z and add an entry for xy -> z in RD
        ii. Dequeue x and y from Q1 and enqueue z to Q2.
4. Queue Management:
    a. Repeat the processing steps until Q1 is empty or contains only one symbol.
    b. If Q2 contains more than one symbol, copy its contents to Q1 and clear Q2, then continue processing.
    c. If Q2 contains only one symbol, it represents the root of the parse tree.
5. Output Encoding:
    a. Encode the parse tree into the balanced parenthesis sequence B and the label sequence L as follows:
        i. For each symbol in Q1 and Q2, append a `0` to B and the symbol to L.
        ii. For each non-terminal symbol in H, append `10` to B and the symbol to L .

Example

Consider the input string "abababa". The compression process of SOLCA is illustrated below:

Step	Q1	Q2	H	RD
1.	a			
2.	a b			
3.		X1	ab -> X1	ab -> X1
4.	a	X1	ab -> X1	ab -> X1
5.	a b	X1	ab -> X1	ab -> X1
6.		X1X2	ab -> X1 aX1 -> X2	ab -> X1 aX1 -> X2
7.	X1	X2	ab -> X1 aX1 -> X2	ab -> X1 aX1 -> X2
8.	X1b	X2	ab -> X1 aX1 -> X2	ab -> X1 aX1 -> X2

9.		X2X3	ab -> X1 aX1 -> X2 X1b -> X3	ab -> X1 aX1 -> X2 X1b -> X3
10.	X2	X3	ab -> X1 aX1 -> X2 X1b -> X3	ab -> X1 aX1 -> X2 X1b -> X3
11.	X2a	X3	ab -> X1 aX1 -> X2 X1b -> X3	ab -> X1 aX1 -> X2 X1b -> X3
12.		X3X4	ab -> X1 aX1 -> X2 X1b -> X3 X2a -> X4	ab -> X1 aX1 -> X2 X1b -> X3 X2a -> X4
13.	X3	X4	ab -> X1 aX1 -> X2 X1b -> X3 X2a -> X4	ab -> X1 aX1 -> X2 X1b -> X3 X2a -> X4
14.	X3X4		ab -> X1 aX1 -> X2 X1b -> X3 X2a -> X4	ab -> X1 aX1 -> X2 X1b -> X3 X2a -> X4
15.		X5	ab -> X1 aX1 -> X2 X1b -> X3 X2a -> X4 X3X4 -> X5	ab -> X1 aX1 -> X2 X1b -> X3 X2a -> X4 X3X4 -> X5

Metódy optimalizácie SOLCA

To further enhance the efficiency of SOLCA, we can introduce parallel processing and adaptive caching techniques:

1. Parallel Processing:
    - Task Parallelism: Identify independent tasks in the grammar construction process that can be executed in parallel. For example, different sections of the input stream can be processed concurrently, with each processor building a partial grammar. These partial grammars can then be merged in a final step.
    - Pipeline Parallelism: Implement a pipeline where different stages of the compression process (e.g., parsing, LCA computation, grammar construction) are handled by separate processors. This approach can help reduce bottlenecks and improve overall throughput.
2. Adaptive Caching:
    - Dynamic Cache Management: Introduce a dynamic caching mechanism that adapts to the input data characteristics. Frequently accessed non-terminals and their pairs can be cached to speed up lookups and updates.

    - Cache-Efficient Data Structures: Use data structures optimized for cache performance. For example, consider using compact hash tables or trees that fit well into the cache lines, minimizing cache misses and improving access times.

These improvements aim to make SOLCA more efficient in terms of both time and space, making it a robust choice for large-scale data compression tasks.