USING ADAPTIVE AUTOMATA IN GRAMMAR-BASED TEXT COMPRESSION TO IDENTIFY FREQUENT SUBSTRINGS

Newton Kiyotaka Miura and João José Neto

Escola Politécnica da Universidade de São Paulo, São Paulo, Brazil

ABSTRACT

Compression techniques allow reduction in the data storage space required by applications dealing with large amount of data by increasing the information entropy of its representation. This paper presents an adaptive rule-driven device - the adaptive automata - as the device to identify recurring sequences of symbols to be compressed in a grammar-based lossless data compression scheme.

KEYWORDS

Adaptive Automata, Grammar Based Data Compression

1. INTRODUCTION

New applications are continuously being created to take advantage of computing system's increasing computational power and storage capacity. Currently, social media Internet applications data analysis and genome database processing are examples of applications that require handling of huge amount of data. Despite all advances in computer hardware technology, the necessity for optimizing its use is economically justifiable. Such optimization can be achieved by increasing the information entropy of the representation of the data, using data compression techniques.

Grammar-based text compression uses a context-free grammar (CFG) as defined in Chomsky's hierarchy [1] for representing a sequence of symbols. This CFG has rules that generates only one sequence: the original text. It is based on the idea that a CFG can compactly represent the repetitive patterns within a text. Intuitively, greater compression would be obtained for input strings containing a greater number of repeated substrings to be represented by the same grammatical production rule. Examples of data with these characteristics are the sequences of genes of the same species and texts in version control systems.

The intrinsic hierarchical definition of a CFG allows string-manipulation algorithms to perform operations directly on their compressed representations without the need for a prior decompression [2] [3] [4] [5]. A potential reduction in the temporary storage space required for data manipulation, and shorter execution times can be expected by decreasing the amount of data to be processed [2]. These features are attractive for improving efficiency in processing large volume of data.

In the literature, we can find propositions for may direct operations in grammatically compressed texts [6] such as string search, edit distance calculation, string or character occurrence frequency calculation, access to a specific position of the original string, obtaining the first occurrence of a substring, and indexing for random access. Examples of grammar-based compression applications such as recurring pattern searching, pattern recognition, data mining, tree manipulation are cited in [7], [3], [2] and [8].

This work focuses on adopting an adaptive device guided by rules [10] for the identification of repetitive data for grammar-based text compression.

Adaptive rule-driven device [10] has the ability of self-modification, that is, it changes the rules of its operation at execution time without the need for external intervention. The adaptive automaton is an example of this type of device. It is a state machine with the capability of changing its configuration based on the input string. It has a computational power equivalent to the Turing machine [11].

In Section 2 the basic concept of this data compression technique is presented with some algorithms found in the literature. In Section 3 an adaptive automaton-based algorithm for identifying frequently occurring substrings is presented.

2. Background

2.1. Grammar based compression

This compression technique uses a CFG $G = (\Sigma, V, D, X_S)$, where Σ is the finite set of terminal symbols and $m = |\Sigma|$. V is the set of nonterminal (or variable) symbols with $\Sigma \cap V = \varnothing$. $D \subset V \times (V \cup \Sigma)^*$ is the finite set of rules of production with size $n = |V|$. $X_S \in V$ is the non-terminal that represents the initial nonterminal symbol of the grammar.

The grammatical compression of a sequence S of terminal symbols is a CFG that produces deterministically only a single sequence S. That is, for any $X \in V$ there is only one production rule in D, and there are no cycles. The syntax tree of G is an ordered binary tree in which the inner nodes are labelled with the non-terminal symbols of V and the leaves with the terminals of Σ, that is, the sequence of labels in the sheets corresponds to the input string S. Each internal node Z corresponds to a production rule $Z \rightarrow XY$ with the child nodes X on the right and Y on the left. As G can be expressed in the normal form of Chomsky [1] any compression grammar is a Straight-Line Program (SLP) [12] [13] [2] which is defined as a grammatical compression on $\Sigma \cup V$ and production rules in the form $X_k \rightarrow X_iX_j$ where $X_k, X_i, X_j \in (\Sigma \cup V)$ and $1 \leq i, j < k \leq n + m$. A string S of the size $l = |S|$ is compressed when l is greater than the sum of the size of the representation of grammar G that generates it and the size of the compressed sequence.

To illustrate this process,
Figure presents the compression of the string S = (a, b, a, a, a, b, c, a, b, a, a), resulting in the compressed sequence $S_c = \{X_4, c, X_3\}$ and the derivation dictionary $D = \{X_1 \rightarrow ab, X_2 \rightarrow aa, X_3 \rightarrow X_1X_2, X_4 \rightarrow X_3,X_1\}$, corresponding to the forest of syntactic trees. The dashed lines of the same colour identify portions of the tree that have parent nodes with the same label.

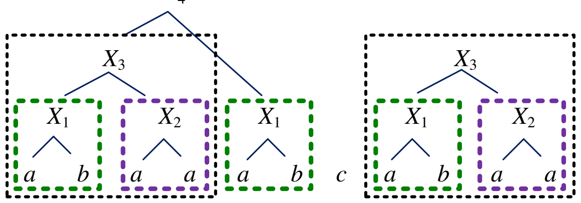

Figure 1. Example of grammar based compression.

The main challenge of grammar-based compression is to find the smallest CFG that generates the original string to maximize the compression rate. This problem has been shown to be intractable. Storer and Szymanski demonstrated [7] that given a string *S* and a constant *k*, obtaining a CFG of size *k* that generates *S* is an NP-complete problem. Furthermore, Charikar et al. [14] demonstrated that the minimum CFG can be approximated by a logarithmic rate and calculated that *8569/8568* is the limit of this approximation rate of the algorithms to obtain the lowest value of *k*, if $P \neq NP$. Research effort has been focused on finding algorithms to infer the approximate minimal grammar, searching for grammars and data structures with characteristics suitable to support operations directly in the compressed data [2] [3] [4] [5]. Regarding the dictionary representation, as discussed by [3] several initial algorithms have adopted Huffman coding to compact it, although it does not allow random access of the strings. More recently the succinct data structure method has been used. A brief description of some researches are presented below.

2.2. Compression algorithms

In the following lines, *l* is the size of the input string, *g* is the minimum CFG size, and *log* refers to log_2.

Nevill-Manning and Witten [15] proposed the Sequitur algorithm, which operates incrementally in relation to the input string with the restriction that each bigram is present only once in a derivation rule of the inferred grammar and that each rule is used more than once. Sequitur operates in a linear space and execution time relative to the size of the input string.

The RePair algorithm developed by Larsson and Moffat [16] constructs a grammar by iteratively replacing pairs of symbols, either terminal or non-terminal, with a non-terminal symbol by doing an off-line processing of the complete text, or long phrases, and adopting a compact representation of the dictionary. It has the following simple heuristic to process a sequence *S*:

1. Identify the most frequent pair *ab* in *S*.
2. Add a rule $X \rightarrow ab$ to the dictionary of productions, where *X* is a new symbol that is not present in *S*.
3. Replace every occurrence of the pair *ab* in *S* by the new nonterminal symbol *X*.
4. Repeat this procedure until any pair in *S* occurs just once.

Although it requires a memory space above 10 times the size of the input string, the algorithm presents a linear execution time, being efficient mainly in the decompression, facilitating search operations in the compressed data.

Rytter [17] and Charikar et al. [14] have developed algorithms that approximate the size of the CFG obtained in $O(log\ l/g)$ by transforming the representation of the data with LZ77 method [18] to the CFG. Rytter [17] proposed the use of a grammar whose derivation tree is an AVL self-balancing binary search tree in which the height of two daughter sub-trees differ only by one unit, which favours pattern matching operations. Charikar et al. [14] imposed the condition that the binary derivation tree be balanced in width, favouring operations such as union and division.
Sakamoto [19] developed an algorithm based on RePair [16] encoding scheme. He obtained an algorithm requiring a smaller memory space, performing iterative substitution of pairs of distinct symbols and repetitions of the same symbol, using double-linked lists for storing the input string and a priority queue for the frequency of the pairs.

Jez [20] modified Sakamoto's algorithm [19] obtaining a CFG of size $O(g\ log\ (l/g))$ for input strings with alphabet Σ identified by numbers of $\{1, \ldots, l^c\}$ for a constant *c*. Unlike the previous research, it was not based on Lempel-Ziv representation.

More recently, Bille et al. [21] proposed two algorithms that improved the space required by the RePair algorithm, one with linear time complexity and other with *O(l log l)*.

Maruyama et al. [5] developed an algorithm based on a context-dependent grammar subclass Σ-*sensitive* proposed to optimize the pattern matching operation on the compacted data.

Tabei et al. [4] has devised a scalable algorithm for least-squares partial regression based on a grammar-packed representation of high-dimensional matrices that allows quick access to rows and columns without the need for decompression. Compared to probabilistic techniques, this approach showed superiority in terms of precision, computational efficiency and interpretability of the relationship between data and tag variables and responses.

The text compression is also a focus of research such as the fully-online compression algorithm (FOLCA) proposed by Maruyama et al. [22] which infers partial parsing trees [17] whose inner nodes are traversed post-order and stored in a concise representation. For class $C = \{x_1, x_2, \ldots, x_n\}$ of n objects, log_n is the minimum of bits to represent any $x_i \in C$. If the representation method requires $n + (n)$ bits for any $x_i \in C$, the representation is called succinct [3]. They presented experimental results proving the scalability of the algorithm in terms of memory space and execution time in processing human genomes with high number of repetitive texts with the presence of noise.

Another online algorithm was proposed by Fukunaga et al. [12] to allow approximate frequent pattern searching in grammatically compressed data using less memory consumption compared to offline methods. They used *edit-sensitive parsing* [23], which measures the similarity of two symbol strings by the edit distance, for comparison of grammars subtrees.

2.3. Grammar inference using adaptive rule-driven device

An adaptive rule-driven device has the self-modifying capability [10], that is, it changes the rules of its operation according to the input data at run time without the need for external intervention. An example is the adaptive automaton [24] which consists of a traditional automaton with the addition of an adaptive mechanism. This mechanism allows modifications in the configuration of the underlying traditional automaton by invoking adaptive functions which can change its set of rules.

Grammar-based compression is a specific case of grammatical inference whose purpose is to learn grammar from information available in a language [9]. In this case, the available information corresponds to the text to be compressed which is the only sentence of a given language. José Neto and Iwai [23] proposed the use of adaptive automaton to build a recognizer with the ability to learn a regular language from the processing of positive and negative samples of strings belonging to that language. The recognizer was obtained by agglutinating two adaptive automata. One automaton constructs the prefix tree from the input strings and the other produces a suffix tree from the inverse sequence of the same string. Matsuno [26] implemented this algorithm and she also presented an application of the Charikar algorithm [14] to obtain a CFG from samples of the language defined by this grammar.

In this paper, we applied the adaptive automaton to identify repetitive sequence of symbols to infer a grammar to generate the compressed version of a string.

3. GRAMMAR BASED COMPRESSION WITH ADAPTIVE IMPLEMENTATION

The RePair [16] algorithm inspired our approach [27] in using the adaptive automaton in the process of finding pairs of symbols to be substituted by a grammar production rule.

The adaptive automaton modifies the configuration of the traditional finite automaton. It is represented by a tuple (Q, Σ, P, q_0, F). For a quick reference, Table 1 describes the meaning of the elements of the tuple, along with other elements used in this work. The notation used by Cereda et al. [28] is adopted.

Modification in the underlying traditional automaton occurs through adaptive functions to be executed either before or after the transitions, according to the consumed input string symbols. The adaptive function executed before the transition is represented by the character '$\cdot$' written after the function name (e.g. $\mathcal{A}\cdot$) and the function executed after the transition is represented by the same character written before the name (e.g. $\cdot\mathcal{B}$). They can modify the automaton configuration by performing elementary adaptive actions for searching, exclusion or insertion of rules. The adaptive functions perform editing actions in the automaton using variables and generators. Variables are filled only once in the execution of the adaptive function. Generators are special types of variables, used to associate unambiguous names for each new state created in the automaton and they are identified by the '*' symbol, for example, g_1^*, g_2^*.

Table 1. List of elements

Element	Meaning
Q	set of states
$F \subset Q$	subset of accepting states
$q_0 \in Q$	initial state
Σ	input alphabet
D	set of adaptive functions
P	P: $D \cup \{\varepsilon\} \times Q \times \Sigma \rightarrow Q \times \Sigma \cup \{\varepsilon\} \times D \cup \{\varepsilon\}$, mapping relation
$\sigma \in \Sigma$	any symbol of the alphabet
$\mathcal{A}\ (q, x) \cdot$	adaptive function $\mathcal{A} \in$ D with arguments q, x triggered before a symbol consumption
$\cdot\mathcal{B}\ (y, z)$	adaptive function $\mathcal{B} \in$ D arguments y, z triggered after the symbol consumption
g_i*	generator used in adaptive functions that associates names with newly created states
$-\ (q_i, \sigma) \rightarrow (q_j)$	elementary adaptive action that removes the transition from q_i to q_j and consumes σ
$+\ (q_i, \sigma) \rightarrow (g_i^*, \varepsilon), \mathcal{A}\cdot$	rule-inserting elementary adaptive action that adds a transition from state q_i, consuming the symbol σ, and leading to a newly created state g_i^* with the adaptive function $\mathcal{A}$ to be executed before the consumption of σ
Out_i	semantic action to be performed in state q_i, as in the Moore machine [1]

This paper presents an adaptive automaton that analyses trigrams contained in the original input string, searching for the most appropriate pair of symbols to be to be replaced by a grammar production rule as in an iteration of the RePair [16] algorithm. From the initial state to any final state it has a maximum of 3 transitions because it analyses at most 3 symbols. Thus, for each trigram, the automaton restarts its operation from the initial state q_0. This requires obtaining the set of all the trigrams present in the input string, and the last bigram. For example, considering the sequence *abcab*, the input is the set {*abc, bca, cab, ab*}.

The automaton helps counting the occurrence of each trigram in the input sequence and its containing prefix and suffix bigrams. This is accomplished by counters that are incremented by semantic action functions Out_i executed in the states in which a bigram or trigram is recognized. Considering the trigram *abc*, it counts the occurrence of the trigram itself and the occurrence of *ab* and *bc* as the prefix and suffix bigrams.

Each bigram has 2 counters, one for prefix and one for suffix. After processing all the trigrams, these counters are used to select the pair to be replaced by a nonterminal symbol. The pair that occurs most frequently inside the most frequent trigram, or the prefix or suffix bigram of this trigram are examples of criteria for the pair selection.

Starting from the terminal state in which the most frequent trigram is recognized, it is possible to identify the constituent bigrams and get the values of their counters by traversing the automaton in the opposite direction of the transactions, towards the starting state. The bigram counter associated with the accepting state stores the total number of occurrences of the suffix bigram. The counter associated with the preceding state stores the number of occurrences of the prefix bigram.

The use of adaptive technique guides the design of this automaton by considering how it should be incrementally built as the input symbols are consumed in run-time, performing the actions just presented above. A conventional finite state automaton to achieve the same results would require the prediction of all combinations of the alphabet symbols in design time.

At the end of an iteration, the chosen bigram is replaced by a nonterminal symbol *X* in the input sequence to prepare it for the next iteration. All bigram and trigram counters are reset to zero, and the automaton is further modified to be used in the next iteration.

It is modified to include the nonterminal *X* in the alphabet and to allow transitions consuming *X* by exploring its directed acyclic graph topology. The automaton can be traversed in a depth first way to identify the chosen bigram. Transitions and nodes that consumes the bigram is replaced by a single transition that consumes the new nonterminal *X*. New nodes and transitions must be created when a prefix of the chosen sequence is the same for other sequences starting from the same node. This approach reduces the processing time for partially rebuilding the automaton in the next iteration.

To better illustrate the configuration evolution of the automaton, we describe the processing of the trigram $\sigma_0\sigma_1\sigma_2$ of a hypothetical input string in the following lines.

Figure 1 shows a simplified version of the initial configuration of the automaton composed by a single state q_0 and transitions for each symbol $\sigma \in \Sigma$ that executes the adaptive function $\mathcal{A}_0\cdot(\sigma, q_0)$ before its consumption. For ease of visualization, the representation of the set of these transitions has been replaced by a single arc in which the symbol to be consumed is indicated as

$\forall\sigma$. This same representation simplification was adopted in the description of the Algorithm 1

```
Algorithm 1: Adaptive function 𝒜0
adaptive function 𝒜0 (s,qx)
    Generators: g1*
    − (qx, s) → (qx)
    + (qx, s) → (g1*, ε)
    + (g1*, ∀σ) → (g1*, ε), 𝒜1(∀σ,g1*)·
end
```

(
Figure 2) of the adaptive function $\mathcal{A}_0(s, q_x)\cdot$.

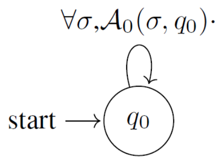

Figure 1. Initial topology of the adaptive automaton

Function $\mathcal{A}_0$ modifies the automaton by removing the transition that consumes the first symbol σ_0 of the trigram, and creating three new elements: the state q_1, the transition from q_0 to q_1 to allow the consumption of σ_0 and the loop transition from q_1 associating it with the consumption of $\forall\sigma \in \Sigma$ and another adaptive function $\mathcal{A}_1(s, q_x)\cdot$.

```
Algorithm 1: Adaptive function 𝒜0
adaptive function 𝒜0 (s,qx)
    Generators: g1*
    − (qx, s) → (qx)
    + (qx, s) → (g1*, ε)
    + (g1*, ∀σ) → (g1*, ε), 𝒜1(∀σ,g1*)·
end
```

Figure 2. Algorithm 1: Adaptive function $\mathcal{A}_0\cdot$

Figure 3 shows the new adaptive automaton topology after consumption of the first symbol σ_0.

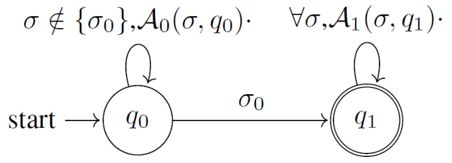

Figure 3. Topology after consuming the first symbol σ_0

The Algorithm 2

```
Algorithm 2: Adaptive function 𝒜1
adaptive function 𝒜1 (s,qx)
    Generators: g1*
    − (qx, s) → (qx)
    + (qx, s) → (g1*, ε)
    + (g1*, ∀σ) → (g1*, ε), 𝒜2(∀σ,g1*)·
end
```

(
Figure 4) presents the adaptive function $\mathcal{A}_1(s, q_x)\cdot$. It operates similarly to $\mathcal{A}_0\cdot$ creating a new state q_2. It also prepares the consumption of a third symbol by inserting a transition with the

adaptive function $\mathcal{A}_2\cdot$, which is described in the algorithm 3

```
Algorithm 3: Adaptive function $\mathcal{A}_2$
adaptive function $\mathcal{A}_2\,(s, q_x)$
    Generators: $g_1^*$
    $-\,(q_x, s) \rightarrow (q_x)$
    $+\,(q_x, s) \rightarrow (g_1^*, \varepsilon)$
End
```

(
Figure 5).

```
Algorithm 2: Adaptive function $\mathcal{A}_1$
adaptive function $\mathcal{A}_1\,(s, q_x)$
    Generators: $g_1^*$
    $-\,(q_x, s) \rightarrow (q_x)$
    $+\,(q_x, s) \rightarrow (g_1^*, \varepsilon)$
    $+\,(g_1^*, \forall\sigma) \rightarrow (g_1^*, \varepsilon), \mathcal{A}_2(\forall\sigma, g_1^*)\cdot$
end
```

Figure 4. Algorithm 2: Adaptive function $\mathcal{A}_1\cdot$

The Algorithm 3 of the adaptive function $\mathcal{A}_2(s, q_x)\cdot$ in

```
Algorithm 3: Adaptive function $\mathcal{A}_2$
adaptive function $\mathcal{A}_2\,(s, q_x)$
    Generators: $g_1^*$
    $-\,(q_x, s) \rightarrow (q_x)$
    $+\,(q_x, s) \rightarrow (g_1^*, \varepsilon)$
End
```

Figure 5 modifies the automaton configuration by removing the transition from q_2 to itself by consuming the symbol σ_2 (the third symbol of the trigram) creating a new state q_3 and the transition to it consuming σ_2. The newly created state q_3 is associated with the semantic function Out_1.

```
Algorithm 3: Adaptive function $\mathcal{A}_2$
adaptive function $\mathcal{A}_2\,(s, q_x)$
    Generators: $g_1^*$
    $-\,(q_x, s) \rightarrow (q_x)$
    $+\,(q_x, s) \rightarrow (g_1^*, \varepsilon)$
End
```

Figure 5. Algorithm 3: Adaptive function $\mathcal{A}_2\cdot$

Figure 6 shows the automaton topology after consumption of the second and third symbol of the input string.

States q_2 and q_3 are associated with output functions Out_0 and Out_1 respectively. They are related to semantic actions in these states. Out_0 is the function responsible for incrementing the occurrence counter of the prefix bigram $\sigma_0\sigma_1$. Out_1 is responsible for incrementing the occurrence counter of the suffix bigram $\sigma_1\sigma_2$, and the counter of the trigram $\sigma_0\sigma_1\sigma_2$.

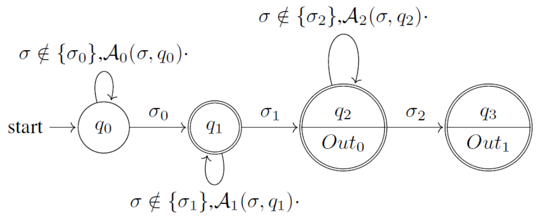
start
q_0
q_1
q_2
Out_0
q_3
Out_1
σ_0
σ_1
σ_2
$\sigma \notin \{\sigma_0\}, \mathcal{A}_0(\sigma, q_0)\cdot$
$\sigma \notin \{\sigma_1\}, \mathcal{A}_1(\sigma, q_1)\cdot$
$\sigma \notin \{\sigma_2\}, \mathcal{A}_2(\sigma, q_2)\cdot$

Figure 6. Topology after consuming the initial trigram $\sigma_0\sigma_1\sigma_2$

To better illustrate the operation of the adaptive automaton,
Figure 7 shows its configuration after processing the sample string *abcabac*. The index *i* of the states q_i corresponds to the sequence in which they were entered by the algorithm.

On the assumption that the most frequent bigram is the pair *ab* and it is going to be replaced by the nonterminal X_1, the new input string for the next iteration is going to be X_1cX_1ac. Figure 8 shows the resulting new topology of the automaton for this case. It was modified to consider X_1 no longer as a nonterminal, but as a member of the input alphabet, and consume it instead of the pair *ab* of the previous iteration. A new transition from state q_0 to q_2 was created, but the transition q_1 was preserved because it is the path to consume the pair *ac*.

4. EXPERIMENTS

A test system is being developed using a publicly available Java language library for adaptive automaton[1] proposed by [29] as the core engine, with the surrounding subsystems such as the semantic functions and the iterations that substitutes the frequently occurring patterns.

Publicly available corpora from http://pizzachili.dcc.uchile.cl/repcorpus.html [30] is going to be used as the test data.

Different criteria in choosing frequently occurring substrings to be replaced by a grammar rule will be experimented. Compression rates, execution time and temporary storage requirements will be measured and compared with other techniques.

[1] https://github.com/cereda/aa

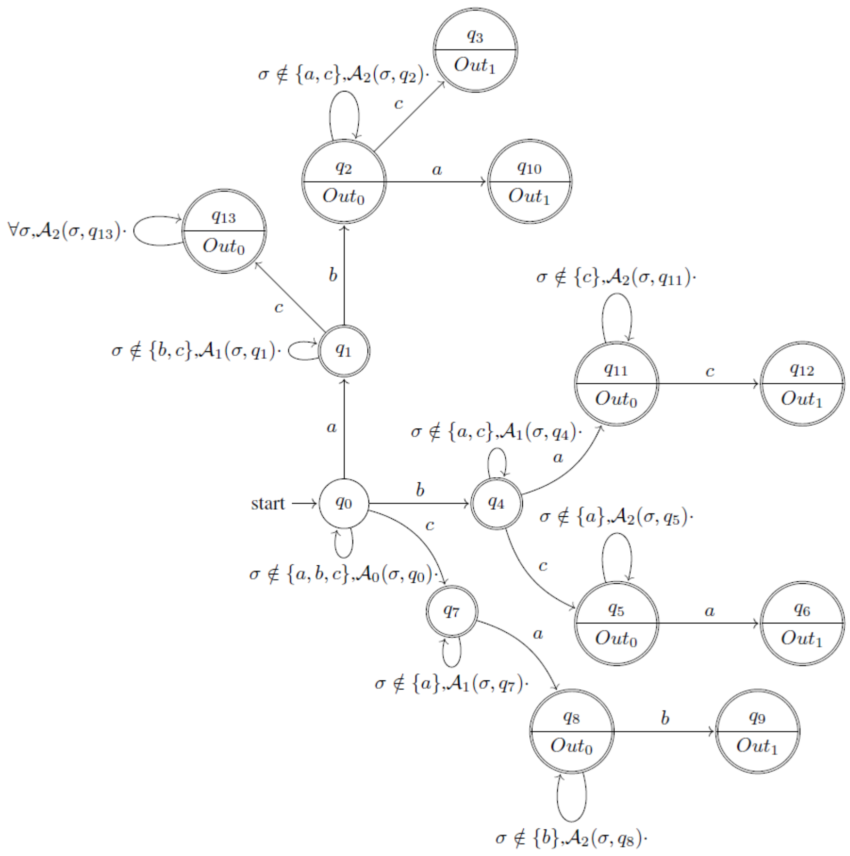
q_3 / Out_1
$\sigma \notin \{a,c\}, \mathcal{A}_2(\sigma, q_2)\cdot$
c
q_2 / Out_0
a
q_{10} / Out_1
$\forall\sigma, \mathcal{A}_2(\sigma, q_{13})\cdot$
q_{13} / Out_0
b
c
$\sigma \notin \{c\}, \mathcal{A}_2(\sigma, q_{11})\cdot$
$\sigma \notin \{b,c\}, \mathcal{A}_1(\sigma, q_1)\cdot$
q_1
q_{11} / Out_0
c
q_{12} / Out_1
a
$\sigma \notin \{a,c\}, \mathcal{A}_1(\sigma, q_4)\cdot$
a
start
q_0
b
q_4
$\sigma \notin \{a\}, \mathcal{A}_2(\sigma, q_5)\cdot$
c
$\sigma \notin \{a,b,c\}, \mathcal{A}_0(\sigma, q_0)\cdot$
c
q_7
q_5 / Out_0
a
q_6 / Out_1
a
$\sigma \notin \{a\}, \mathcal{A}_1(\sigma, q_7)\cdot$
q_8 / Out_0
b
q_9 / Out_1
$\sigma \notin \{b\}, \mathcal{A}_2(\sigma, q_8)\cdot$

Figure 7. Automaton topology after processing *abcabac*.

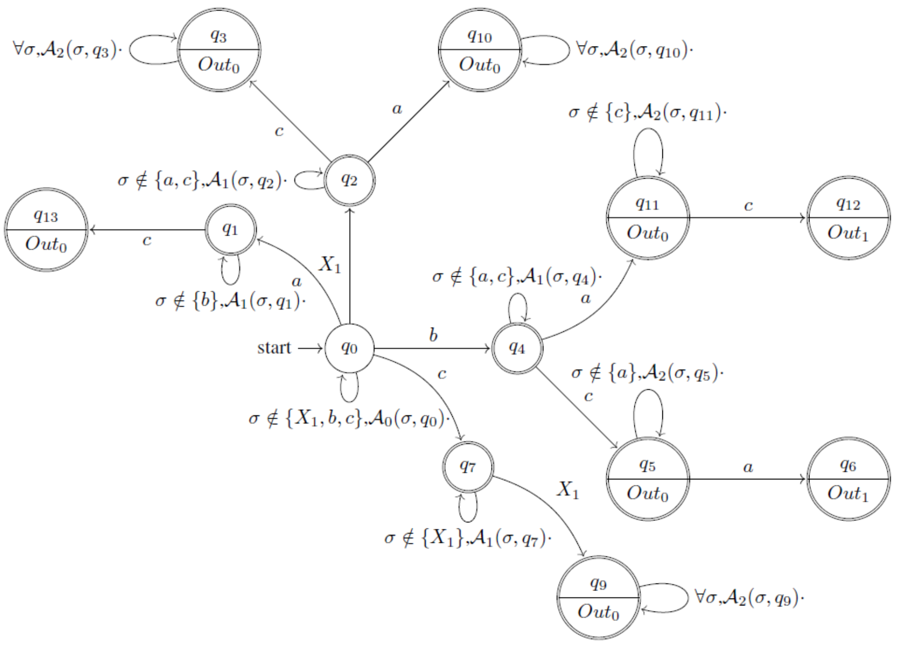
$\forall\sigma, \mathcal{A}_2(\sigma, q_3)\cdot$
q_3 / Out_0
q_{10} / Out_0
$\forall\sigma, \mathcal{A}_2(\sigma, q_{10})\cdot$
a
c
$\sigma \notin \{c\}, \mathcal{A}_2(\sigma, q_{11})\cdot$
$\sigma \notin \{a,c\}, \mathcal{A}_1(\sigma, q_2)\cdot$
q_2
q_{13} / Out_0
c
q_1
q_{11} / Out_0
c
q_{12} / Out_1
X_1
a
$\sigma \notin \{b\}, \mathcal{A}_1(\sigma, q_1)\cdot$
$\sigma \notin \{a,c\}, \mathcal{A}_1(\sigma, q_4)\cdot$
a
start
q_0
b
q_4
c
$\sigma \notin \{a\}, \mathcal{A}_2(\sigma, q_5)\cdot$
c
$\sigma \notin \{X_1,b,c\}, \mathcal{A}_0(\sigma, q_0)\cdot$
q_7
X_1
q_5 / Out_0
a
q_6 / Out_1
$\sigma \notin \{X_1\}, \mathcal{A}_1(\sigma, q_7)\cdot$
q_9 / Out_0
$\forall\sigma, \mathcal{A}_2(\sigma, q_9)\cdot$

Figure 8. Automaton topology after replacing the pair *ab* by the nonterminal X_1.

5. Conclusion

In this work, we presented the adaptive automaton as a device to find the most repeated bigram inside the most frequent trigram in a sequence of symbols. It is used to infer a substitution rule for a grammar based compression scheme. This bigram can be specified to be whether prefix or suffix of the trigram. After processing the input sequence of iteration, the same automaton is further modified to incorporate the non-terminal symbol used by the grammar rule to substitute the chosen repetitive pair in a new alphabet. This can reduce the next iteration execution time by preserving part of the automaton that would be adaptively built in run-time.

As a future work, the adaptive automaton can be further expanded to analyse n-grams larger than trigrams. In addition, a comparative performance study can be done with other techniques. Another point to be investigated is the adoption of an adaptive grammatical formalism [31] in the description of the inferred grammar with the aim of making some operation in the compressed data.

Adaptive rule-driven device allows the construction of large and complex system by simplifying the representation of the problem. It allows the automation of the construction of large structures by programming the steps of growth in run-time instead of predicting all the combinatorial possibilities in design-time. This type of automaton is designed by specifying how the device must be incrementally modified in response to the input data, from a simple initial configuration, considering its desired intermediate configurations and the output to be obtained by associating semantic actions.

Acknowledgement

The first author is supported by Olos Tecnologia e Sistemas.

References

[1] Sipser, M. (2006) “Introduction to the theory of computation”, Thomson Course Technology.
[2] Lohrey, M. (2012) “Algorithmics on SLP-compressed strings: A survey.” Groups Complexity Cryptology, vol. 4, no. 2, pp. 241–299.
[3] Sakamoto, H. (2014) “Grammar compression: Grammatical inference by compression and its application to real data”, in Proceedings of the 12th International Conference on Grammatical Inference, ICGI 2014, Kyoto, Japan, September 17-19, 2014., pp. 3–20.
[4] Tabei,Y., Saigo, H., Yamanishi, Y. & Puglisi, S. J. (2016) “Scalable partial least squares regression on grammar-compressed data matrices”, in 22nd KDD, pp. 1875—1884.
[5] Maruyama, S., Tanaka, Y., Sakamoto, H., & Takeda, M. (2010) “Context-sensitive grammar transform: Compression and pattern matching”, IEICE Transactions on Information and Systems, vol. E93.D, no. 2, pp. 219–226.
[6] Tabei, Y. (2016) “Recent development of grammar compression”, Information Processing Society of Japan Magazine, vol. 57, no. 2, pp. 172–178 (in Japanese).
[7] Jez, A. (2016) “A really simple approximation of smallest grammar”, Theoretical Computer Science, vol. 616, pp. 141–150.
[8] Lohrey, M. (2015) “Grammar-based tree compression”, in International Conference on Developments in Language Theory. Springer, pp. 46–57.
[9] De la Higuera, C. (2010) Grammatical inference: learning automata and grammars. Cambridge University Press.
[10] José Neto, J. (2001) “Adaptive rule-driven devices - general formulation and case study”, in CIAA 2001 6th International Conference on Implementation and Application of Automata, ser. Lecture Notes in Computer Science, B. W. Watson and D. Wood, Eds., vol. 2494. Pretoria, South Africa: Springer-Verlag, pp. 234–250.

[11] Rocha, R. L. A. & José Neto, J. (2000) “Adaptive automaton, limits and complexity compared to the Turing machine”, in Proceedings of the I LAPTEC, pp. 33–48.

[12] Fukunaga, S., Takabatake, Y., I, T. & Sakamoto, H. (2016) “Online grammar compression for frequent pattern discovery”, CoRR, vol. abs/1607.04446.

[13] Takabatake, Y., Tabei, Y., & Sakamoto, H. (2015) Online Self-Indexed Grammar Compression. Springer International Publishing, pp. 258–269.

[14] Charikar, M., Lehman, E., Liu, D., Panigrahy, R., Prabhakaran, M., Sahai, A. & Shelat, A. (2005) “The smallest grammar problem.” IEEE Trans. Inf. Theory, vol. 51, no. 7, pp. 2554–2576.

[15] Nevill-Manning, C. G. & Witten, I. H. (1997) “Identifying hierarchical structure in sequences: A linear-time algorithm”, J. Artif. Intell. Res. (JAIR) vol. 7, pp. 67–82.

[16] Larsson, N. J. & Moffat, A. (1999) “Offline dictionary-based compression”, in Data Compression Conference, 1999. Proceedings. DCC ’99, pp. 296–305.

[17] Rytter, W. (2002) Application of Factorization to the Approximation of Grammar-Based Compression. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 20–31.

[18] Ziv, J. & Lempel, A. (2006) “A universal algorithm for sequential data compression”, IEEE Trans. Inf. Theor., vol. 23, no. 3, pp. 337–343.

[19] Sakamoto, H., (2005) “A fully linear-time approximation algorithm for grammar-based compression”, Journal of Discrete Algorithms, vol. 3, no. 2–4, pp. 416–430.

[20] Jez, A. (2015) “Approximation of grammar-based compression via recompression”, Theoretical Computer Science, vol. 592, pp. 115–134.

[21] Bille, P. & Gørtz, I. L. & Prezza, N. (2016) “Space-efficient re-pair compression,” CoRR, vol. abs/1611.01479.

[22] Maruyama, S. & Tabei, Y. (2014) “Fully online grammar compression in constant space.” in DCC, Bilgin, A., Marcellin, M. W., Serra-Sagristà, J. & Storer, J. A. Eds. IEEE, pp. 173–182.

[23] Cormode, G. & Muthukrishnan, S. (2007) “The string edit distance matching problem with moves”, ACM Transactions on Algorithms (TALG) vol. 3, no. 1, pp. 2:1–2:19.

[24] José Neto, J. (1994) “Adaptive automata for context-sensitive languages”, SIGPLAN Notices, vol. 29, no. 9, pp. 115–124.

[25] José Neto, J. & Iwai, M. K. (1998) “Adaptive automata for syntax learning” in Anais da XXIV Conferência Latinoamericana de Informática - CLEI 98, pp. 135–149.

[26] Matsuno, I. P. (2006) “Um estudo dos processos de inferência de gramáticas regulares e livres de contexto baseados em modelos adaptativos”, Master’s Thesis, Escola Politécnica da Universidade de São Paulo, São Paulo, Brazil (in Portuguese).

[27] Miura, N. K. & José Neto, J. (2017) “Adaptive automata for grammar based compression” in Proceedings of the 4th International Conference on Computer Science and Information Technology: CoSIT 2017, pp. 173–183

[28] Cereda, P. R. M. & José Neto, J. (2015) “A recommendation engine based on adaptive automata”, in Proceedings of the 17th International Conference on Enterprise Information Systems - Volume 2: ICEIS, pp. 594–601.

[29] Cereda, P. R. M. & José Neto, J. (2016) “AA4J: uma biblioteca para implementação de autômatos adaptativos” in Memórias do X Workshop de Tecnologia Adaptativa – WTA 2016, 2016, pp. 16–26 (in Portuguese).

[30] Ferragina, P. & González, R. & Navarro, G. & Venturini, R. (2009) “Compressed text indexes: From theory to practice”, Journal of Experimental Algorithmics (JEA), vol. 13, pp. 12–31.

[31] Iwai, M. K. (2000) “Um formalismo gramatical adaptativo para linguagens dependentes de contexto”, PhD Thesis, Escola Politécnica da Universidade de São Paulo, São Paulo, Brazil (in Portuguese).

AUTHORS

Newton Kiyotaka Miura is a researcher at Olos Tecnologia e Sistemas and a PhD candidate in Computer Engineering at Departamento de Engenharia de Computação e Sistemas Digitais, Escola Politécnica da Universidade de São Paulo. He received his Electrical Engineering degree from Escola Politécnica da Universidade de São Paulo (1989) and holds a master's degree in Systems Engineering from University of Kobe, Japan (1993). His research interests include adaptive technology, adaptive automata, adaptive devices and natural language processing.

João José Neto is an associate professor at Escola Politécnica da Universidade de São Paulo and coordinates the Language and Adaptive Technology Laboratory of Departamento de Engenharia de Computação e Sistemas Digitais. He received his Electrical Engineering degree (1971), master's degree (1975) and PhD in Electrical Engineering (1980) from Escola Politécnica da Universidade de São Pa ulo. His research interests include adaptive devices, adaptive technology, adaptive automata and applications in adaptive decision making systems, natural language processing, compilers, robotics, computer assisted teaching, intelligent systems modelling, automatic learning processes and adaptive technology inferences.