Word-based Compression Method with Direct Access[1]

Jiří Dvorský, Václav Snášel

Computer Science Department, Palacky University of Olomouc, Tomkova 40, 779 00 Olomouc, Czech Republic

e-mail: {jiri.dvorsky,vaclav.snasel}@upol.cz

Abstract. Compression method (WRAC) based on finite automata is presented in this paper. Simple algorithm for constructing finite automaton for given regular expression is shown. The best advantage of this algorithm is the possibility of random access to a compressed text. The compression ratio achieved is fairly good. The method is independent on source alphabet i.e. algorithm can be character or word based.

Key words: word-based compression, text databases, information retrieval, HuffWord, WLZW

1 Introduction

Data compression is an important part of the implementation of full text retrieval systems. The compression is used to reduce space occupied by indexes and text of documents. There are many popular algorithms to compress a text, but none of them can perform direct access to the compressed text. This article presents an algorithm, based on finite automaton, which allows such type of access. The definition of finite automata is given in the first section. Compression algorithm itself is described in the second section and the third section shows some experimental results. At the end the conclusion is given.

2 Finite automata

Definition 1 *A deterministic finite automaton (DFA) [5] is a quintuple* (Q, A, δ, q_0, F)*, where* Q *is a finite set of states,* A *is a finite set of input symbols (input alphabet),* δ *is a state transition function* $Q \times A \rightarrow Q$*,* q_0 *is the initial state,* $F \subseteq Q$ *is the set of final states.*

[1]This work was done under grant from the Grant Agency of Czech Republic, Prague No.: 201/00/1031

Definition 2 *Regular expression U on alphabet A is defined as follows:*

1. *$\emptyset$, ε and a are regular expression for all $a \in A$*
2. *If U, V are regular expression on A then $(U+V), (U \cdot V)$ and $(U)^*$ are regular expression on A.*

Definition 3 *Value $h(U)$ of regular expression U is defined as:*

$$\begin{aligned} h(\emptyset) &= \emptyset \\ h(\varepsilon) &= \{\varepsilon\} \\ h(a) &= \{a\} \\ h(U+V) &= h(U) \cup h(V) \\ h(U \cdot V) &= h(U) \cdot h(V) \\ h(U^*) &= (h(U))^* \end{aligned}$$

Definition 4 *Derivative $\frac{dU}{dx}$ of regular expression U by $x \in A^*$ is defined as:*

1. $$\frac{dU}{d\varepsilon} = U$$
2. *$\forall a \in A$ it holds:*
$$\begin{aligned} \frac{d\varepsilon}{da} &= \emptyset \\ \frac{d\emptyset}{da} &= \emptyset \\ \frac{db}{da} &= \begin{cases} \emptyset & \text{if } a \neq b \\ \varepsilon & \text{otherwise} \end{cases} \\ \frac{d(U+V)}{da} &= \frac{dU}{da} + \frac{dV}{da} \\ \frac{d(U \cdot V)}{da} &= \frac{dU}{da} \cdot V + \frac{dV}{da} \text{ if } \varepsilon \in h(U) \\ \frac{d(U \cdot V)}{da} &= \frac{dU}{da} \cdot V \text{ if } \varepsilon \notin h(U) \\ \frac{d(V^*)}{da} &= \frac{dV}{da} \cdot V^* \end{aligned}$$
3. *For $x = a_1 a_2 \ldots a_n$, where $a_i \in A$ it holds:*
$$\frac{dV}{dx} = \frac{d}{da_n}\left(\frac{d}{da_{n-1}}\left(\cdots \frac{d}{da_2}\left(\frac{dV}{da_1}\right)\cdots\right)\right)$$

Derivative of regular expression V by string x is an equivalent

$$h\left(\frac{dV}{dx}\right) = \{y : xy \in h(V)\}$$

In other words, derivative of V by x is expression U such $h(U)$ contains strings which arise from strings in $h(V)$ by cutting prefix x.

Example 1 *Let be $h(V) = \{abccabb, abbacb, babbcab\}$. Then $h(\frac{dV}{da}) = \{bccabb, bbacb\}$.*

	$\frac{dV}{d0}$	$\frac{dV}{d1}$
$(0+1)^* \cdot 01$	$(0+1)^* \cdot 01 + 1$	$(0+1)^* \cdot 01$
$(0+1)^* \cdot 01 + 1$	$(0+1)^* \cdot 01 + 1$	$(0+1)^* \cdot 01 + \varepsilon$
$(0+1)^* \cdot 01 + \varepsilon$	$(0+1)^* \cdot 01 + 1$	$(0+1)^* \cdot 01$

Table 1: Construction of DFA for $V = (0+1)^* \cdot 01$

2.1 Construction of DFA for given regular expression V

Theorem 1 *When DFA accepts, in state q, language defined by V then accepts in state $\delta(q,a)$ language defined by $\frac{dV}{da}$, for all $a \in A$ (see [5]).*

For given regular expression V we construct $DFA(V) = (Q, A, \delta, q_0, F)$, where

- Q is a set of regular expressions
- A is given alphabet
- $\delta(q,a) = \frac{dq}{da}, \forall a \in A$
- $q_0 = V$
- $F = \{q \in Q : \varepsilon \in q\}$

Example 2 *Let's construct automaton for $V = (0+1)^* \cdot 01$ – words ending with 01. See table 1. Final state is $(0+1)^* \cdot 01 + \varepsilon$ only.*

3 Random access compression

Let be $A = \{a_1, a_2, \ldots, a_n\}$ an alphabet. Document D of length m can be written as sequence $D = d_0, d_1, \ldots, d_{m-1}$, where $d_i \in A$. For each position i we are able to find out which symbol d_i is at this position. We must save this property to create compressed document with random access.

A set of position $\{i; 0 \leq i < m\}$ can be written as a set of binary words $\{b_i\}$ of fixed length. This set can be considered as language $L(D)$ on alphabet $\{0,1\}$. It can be easy shown that the language $L(D)$ is regular ($L(D)$ is finite) and it is possible to construct DFA which accepts the language $L(D)$. This DFA can be created, for example, by algorithm given in section 2. Regular expression is formed as $b_0 + b_1 + \cdots + b_{m-1}$.

Compression of the document D consists in creating a corresponding DFA. But decompression is impossible. The DFA for the document D can only decide, whether binary word b_i belongs to the language $L(D)$ or not. The DFA does not say anything about a symbol which appears in position i. Inorder to do this, the definition of DFA must be extended.

Definition 5 *A deterministic finite automaton with output (DFAO) is a 7-tuple* $(Q, A, B, \delta, \sigma, q_0, F)$*, where* Q *is a finite set of states,* A *is a finite set of input symbols (input alphabet),* B *is a finite set of output symbols (output alphabet),* δ *is a state transition function* $Q \times A \to Q$*,* q_0 *is the initial state,* σ *is an output function* $F \to B$*,* $F \subseteq Q$ *is the set of final states.*

This type of automaton is able to determine for each of the accepted words b_i which symbol lies on position i. To create an automaton of such a type the algorithm mentioned in section 2 must be extended too. Regular expression V, which is input into the algorithm, consists of words b_i. Each b_i must carry its output symbol d_i. Regular expression is now formed as $b_0 d_0 + b_1 d_1 + \cdots + b_{m-1} d_{m-1}$,

Example 3 *Let be for example document* $D = abracadabra$*,* $m = 11$*. Regular expression* V *will be*

$$\begin{aligned} V &= 0000a + 0001b + 0010r + 0011a + \\ &\quad 0100c + 0101a + 0110d + 0111a + \\ &\quad 1000b + 1001r + 1010a \end{aligned}$$

$DFAO(V) = (Q, A, B, \delta, \sigma, q_0, F)$ *will be constructed. For the construction of DFAO see following table.*

State q	V	dV/d0	dV/d1
0	0000a, 0001b, 0010r, 0011a, 0100c, 0101a, 0110d, 0111a, 1000b, 1001r, 1010a	000a, 001b, 010r, 011a, 100c, 101a, 110d, 111a	000b, 001r, 010a
1	000a, 001b, 010r, 011a, 100c, 101a, 110d, 111a	00a, 01b, 10r, 11a	00c, 01a, 10d, 11a
2	000b, 001r, 010a	00b, 01r, 10a	$\emptyset$
3	00a, 01b, 10r, 11a	0a, 1b	0r, 1a
4	00c, 01a, 10d, 11a	0c, 1a	0d, 1a
5	00b, 01r, 10a	0b, 1r	0a
6	0a, 1b	εa	εb
7	0r, 1a	εr	εa
8	0c, 1a	εc	εa
9	0d, 1a	εd	εa
10	0b, 1r	εb	εr
11	0a	εa	$\emptyset$
12	εa	$\emptyset$	$\emptyset$
13	εb	$\emptyset$	$\emptyset$
14	εr	$\emptyset$	$\emptyset$
15	εc	$\emptyset$	$\emptyset$
16	εd	$\emptyset$	$\emptyset$

$Q = \{q_0, q_1, \ldots q_{16}\}$, $A = \{0, 1\}$, $B = \{a, b, c, d, r\}$, $F = \{q_{12}, q_{13}, q_{14}, q_{15}, q_{16}\}$,

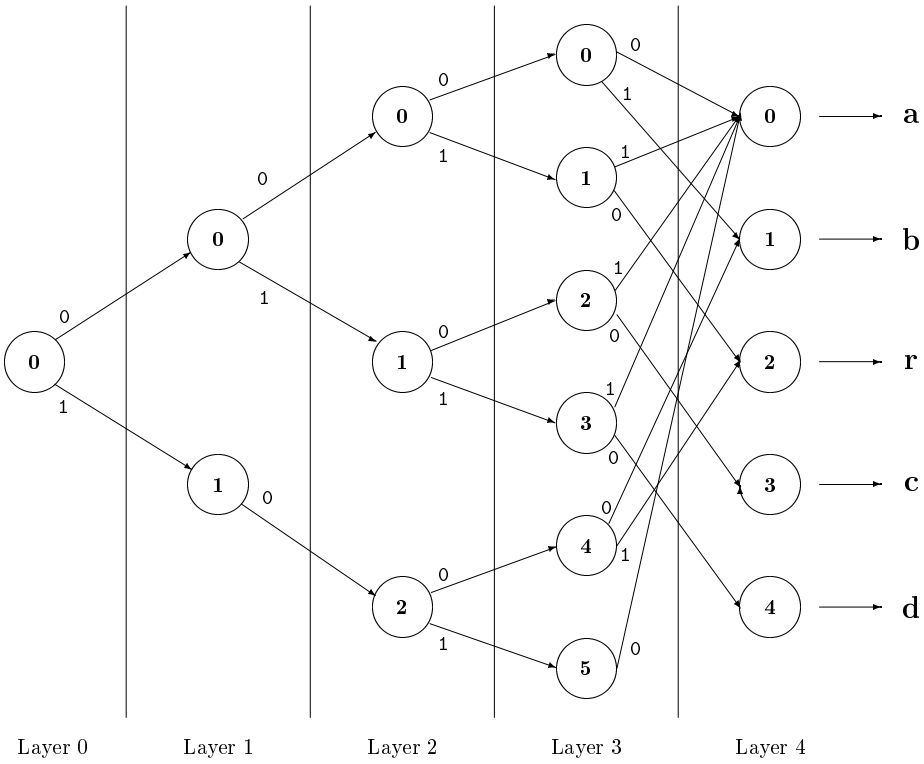

Figure 1: Automaton for expression V from example 3

Such constructed automaton have following properties:

1. there are no transitions from final states,
2. let be $|q|$ for $q \in Q$ the length of words in appropriate regular expression. If $\delta(q_i, a) = q_j$, where $q_i, q_j \in Q$, $a \in A$, then $|q_i| > |q_j|$. In other words, the state transition function contain only forward transitions. There are no cycles.

The set of states Q of the automaton $DFAO(V)$ is divided into disjunct subsets (so called *layers*). Transitions are done only between two adjacent layers. Thus states can be numbered locally in those layer. Final automaton from our example is drawn in figure 1.

Final automaton is stored on disk after construction. All layers are stored sequentially. Three methods of storing layers are available now:

Raw – the layer is stored as a sequence of integer numbers. Appropriate for short layers.

Bitwise – maximum state number max in layer is found. The layer is stored as a sequence of integer numbers, each $\lceil \log_2 max \rceil$ bits long.

Linear – linear prediction of transitions is made. Parameters of the founded line and a correction table are stored.

Let's remark, that algorithm of construction of automaton is independent with respect to its output alphabet. There are two possibilities. The first is a classic character

NT	LT	NS	LCF	CR
10	29	39	580	2000
100	258	103	780	302.33
1,000	2,636	589	2,016	76.48
10,000	25,793	5,018	13,548	52.53
50,000	129,728	22,418	59,676	46.00
100,000	259,571	41,593	113,976	43.91
200,000	522,872	74,872	206,728	39.54
300,000	788,773	106,775	294,448	37.33
400,000	1,053,040	139,900	402,840	38.25
500,000	1,314,038	173,126	492,448	37.48
800,000	2,120,924	274,495	797,292	37.59
1,000,000	2,651,385	340,020	999,920	37.71
1,535,710	4,077,774	511,678	1,480,884	36.32

Table 2: Experimental results for file bible.txt
Where **NT** is the number of tokens, **LT** is the length of tokens in bytes, **NS** is the number of states, **LCF** is the length of compressed text in bytes (automaton in memory has the same size) and **CR** is the compression ratio $(LCF/LT)\cdot 100\%$

based version. Algorithm is one-pass and output alphabet is a standard ASCII. For the text retrieval systems word-based version (the second possibility) is more advantageous because of the character of natural languages.

4 Experimental results

To allow practical comparison of algorithm, experiments have been performed on some compression corpus. For test has been used Canterbury Compression Corpus (large files), especially King's James Bible (bible.txt) file which is $4,077,774$ bytes long. There are $153,5710$ tokens and $13,461$ of them are distinct.

A word-based version of algorithm has been used for a test. Compression algorithm has been tested for different input size. The length of the input tokens, the number of states of the resulting automaton, the size of the compressed file and the compression ratio have been observed. Results are given in table 2.

Another tests were done with Czech text file. This file is $11,076,629$ bytes long. There are $2,332,127$ and $70,177$ of them are distinct. Results are given in table 4.

Both tests were done on Pentium II/350Mhz with 128MB of RAM. Program was compiled by MS Visual C++ 6.0 as 32-bit console application under MS Windows 2000. Implementation of this method is described in [4].

5 Conclusion and Future works

Some advanced techniques, how to store layers (i.e. sequences of numbers) will be adopted, such as differential encoding etc.

Compression utility	Compressed text [bytes]	Ratio [%]
WRAC	1,480,884	36.32
ARJ 2.41a	1,207,114	29,6
WINZIP 6.2	1,178,869	28,91
GZip (UNIX)	1,178,757	28.9
WinRAR 2.6	994,346	24,38
Jar32 1.01	908,547	22,28
WLZW (word-based LZW)	896,956	22
WRAC (estimation for nonperiodic texts)	816,000	20

Table 3: Comparison with other compression utilities (bible.txt)

NT	**LT**	**NS**	**LCF**	**CR**
1,000	4,684	585	2,132	45,52
10,000	45,977	7,075	21,172	46,05
100,000	432,181	58,526	167,780	38,82
1,000,000	4,872,038	562,766	1,817,820	37,31
2,332,127	11,076,629	1,235,086	4,152,888	37,49

Table 4: Experimental results for Czech text
Where **NT** is the number of tokens, **LT** is the length of tokens in bytes, **NS** is the number of states, **LCF** is the length of compressed text in bytes (automaton in memory has the same size) and **CR** is the compression ratio $(LCF/LT)\cdot 100\%$

Compression utility	Compressed text [bytes]	Ratio [%]
WRAC	4,152,888	37,49
ARJ 2.41a	2,890,560	26,1
WINZIP 6.2	2,816,458	25,43
WinRAR 2.6	2,379,730	21,48
WLZW (word-based LZW)	2,294,576	20,72

Table 5: Comparison with other compression utilities (Czech text file)

From figure 1 can be seen, that the most piece of information is between layer 3 and layer 4. The transitions from state 0 in layer 3 represents the first and the second character in document. The transitions from state 1 in layer 3 represents the third and the fourth character in document and so on. For such automaton layers 0, 1 and 2 can be omitted. There is at least one problem of course. Let's consider the text *abraabraara*. When the two last layers would be stored decompressed text is *abraara*. It is due to that there are two equal substrings *abra* which length are equal to power of 2 and its positions are powers of 2 too. These positions differ only in one bit and this bit make a multiple transition (by 0 and 1). The result is that only one of these substrings appears in decompressed text. The position of multiple transition have to be stored.

But while automaton would be smaller because equal substrings have the same sub-automata i.e. compression ratio would be better. Compression ratio is estimated at 20 percent.

Compression methods suitable for the use in textual databases have certain special properties:

- these methods must be very fast in decompression;
- the information necessary for decompression should be usable for text searching.

It is important to realise that this method does not actually depend on text encoding. This means that it performs successfully for a text encoded in UNICODE as well.

Several word-based compression algorithms were developed for the text retrieval systems. There is well-known Huffword [1, 6], and WLZW [2, 3] (our version of word-based, two-phase LZW).

References

[1] R. B. Yates, B. R. Neto. *Modern Information Retrieval.* Addison Wesley 1999

[2] J. Dvorský, V. Snášel, J. Pokorný: *Word-based Compression Methods for Text Retrieval Systems.* Proc. DATASEM'98, Brno 1998

[3] J. Dvorský, V. Snášel, J. Pokorný. *Word-based Compression Methods for Large Text Documents.* Data Compression Conferencs - DCC '99, Snowbird, Utah USA.

[4] J. Dvorský. *Text Compression with Random Access.* Workshop ISM 2000, Czech Republic, ISBN 80-85988-45-3

[5] G. Rozenberg, A.Salomaa, Ed. *Handbook of Formal Language.* Springer Verlag 1997, Vol. I.–III.

[6] I. H. Witten, A. Moffat, T. C. Bell: *Managing Gigabytes: Compressing and Indexing Documents and Images.* Van Nostrand Reinhold, 1994.