Finite State Automata – a concept for data representation

© Marian Mindek, Michal Burda

Department of Computer Science, FEI, VSB - Technical University of Ostrava,
17. listopadu 15, 708 33, Ostrava-Poruba, CZ
{marian.mindek, michal.burda}@vsb.cz

Abstract

In this paper, we introduce finite automata as a tool for matrix specification and compression. We also describe, how to get additional interesting information from such automata. At last, we focus on techniques for storing resultant automata as matrices, tables of an SQL database, or as XML document.

1 Introduction

Finite automata can be used as a tool for efficient matrix storage with the possibility of compression. Since matrices are general purpose data structures, this approach could be used on images, text, sound files etc. Storing suitable data as automaton brings up also the benefit of obtaining the additional interesting information about our data [1, 3, 4, 5, 7, 8].

Such technique is especially useful when dealing with large binary matrices. A traditional approach (compression using common algorithms) solves only a part of the problem – they save a lot of space, but there is no way how to make any changes on the compressed matrix.

In the next chapter, we allege only the necessary background of the automata theory. After that, we describe simple algorithm for compressing matrices by creating the finite state automata.

As one can see in the following sections, we can use the created automaton to get additional interesting information about the compressed data, namely the patterns of similar parts of source matrix. Such patterns may be used in special searching algorithms to find e.g. similar parts of faces, medical pictures, buildings tracing, parts of large sparse matrices, similar noise, similar trends, pieces of text, etc.

Storing matrices as automata is not the read-only compression. It is also possible to modify the compressed matrix. However, this process is not very straightforward, but there exist many variants how to enable update of compressed matrices: we can do the slow re-compression of the overall matrix to get best compression ratio, it is also possible to re-compress the changed part only, which is faster but saves less space.

Proceedings of the Spring Young Researcher's Colloquium on Database and Information Systems SYRCoDIS, St.-Petersburg, Russia, 2005

More about it can be found in chapter *Compression*).

Tail of the paper is intended to discuss the methods of storing automata to usual data tables, matrices or XML documents.

2 How to represent matrix with automaton

2.1 Elementary theory

To understand the following, we allege here the necessary background only. For more about automata theory please read [1, 6].

In the subsequent, we work with images instead of matrices. It is due to simplicity and better understandability of this paper. To work with matrices, no additional effort is needed, as one may realize. Also, for more about using automata as a tool for specifying image, please read [3, 4, 5, 7, 8].

A digitized image of the finite resolution m x n consists of m x n pixels each of which takes a Boolean value (1 for black, 0 for white) for bi-level image, or real value (practically digitized to an integer value from 0 to 256) for a gray-scale image, or 24bit information (RGB) for true-color image.

In the subsequent, we consider only square images of resolution 2^n x 2^n. In order to facilitate the application of finite automata to image description, we can assign unique word (path through the automaton) of length n over the alphabet

$$\Sigma=\{0, 1, 2, 3\}$$

to each pixel of the 2^n x 2^n resolution image.

Each Σ's symbol in the word represents the address of a sub-square of the square addressed with the preceding symbols of the word. We choose ε as an address of the whole unit square.

Single digits, as shown in the left of figure 1, address the quadrants. Thus, the four sub-squares of a square with address w have address *w0*, *w1*, *w2* or *w3*, respectively. The middle of fig. 1 shows addresses of all pixels of a 4 x 4 image. The sub-square (pixel) with address *3203* is shown on the right of figure 1.

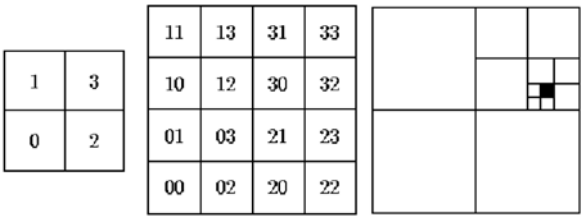

Figure 1. The addresses of the quadrants, of the sub-square of resolution 4 x 4, and the sub-square specified by the string 3203.

We denote Σ^m the set of all words over Σ of the length m, by Σ^* the set of all words over Σ.

In order to specify a black-white image of resolution 2^m x 2^m, we need to specify a language $\boldsymbol{L \subseteq \Sigma^m}$ where the word w belongs to L iff the coresponding sub-square on the image is black.

The automaton coresponding to the given image should be created as to recognize the language L. That is, it must end in accept state iff the sub-square of given address is black.

To be able to compress color images (multi-valued matrices), each ending state of the automaton must contain the color (value) of each pixel (cell) in the sub-square.

Now, we are ready to give some examples. We assume that the reader is familiar with the elementary facts about finite automata and regular sets – see e.g. [1, 6].

Example. Consider the 2 x 2 chess-board in the left of figure 2. Its automata could be described by a regular expression $\{1,2\}\Sigma^*$. Please note, the regular set also describes the 2 x 2 chess board in arbitrary resolutions (concretely, 2n x 2n for any positive integer n).

The 8 x 8 chess-board in the right of figure 2 can be described by the regular expression $\Sigma^2\{1,2\}\Sigma^*$ or by automaton A in Fig. 3.

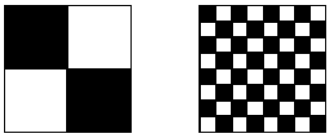

Figure 2. 2 x 2 and 8 x 8 chess-boards

Notice that here we used the fact that the regular expression $\Sigma^2\{1,2\}\Sigma^*$ is the concatenation of two regular expressions Σ^2 and $\{1,2\}\Sigma^*$.

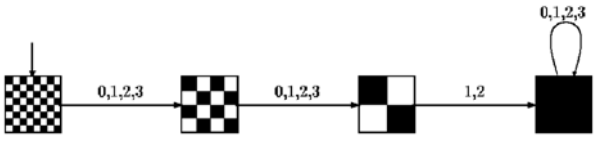

Figure 3. Finite automaton A defining the 8 x 8 chess-board.

Example. Clearly, $L_1 = \{1,2\}^*0$ are addresses of the infinitely many squares illustrated at the left of Fig. 4. If we place the completely black square defined by $L_2 = \Sigma^*$ into all these squares we get the image specified by the concatenation $L_1L_2=\{1,2\}^*0\Sigma^*$ which is the triangle shown in the middle of Fig. 4.

Figure 4. The squares specified by $\{1,2\}^*0$, a triangle defined by $\{1,2\}^*0\Sigma^*$, and the corresponding automaton.

Example. By placing the triangle $L= L_1L_2$ from the previous example into all the squares with addresses $L_3=\{1,2,3\}^*0$ we get the image $L_3=\{1,2,3\}^*0\{1,2\}^*0\Sigma^*$ shown at the left of Fig. 5.

Zooming [5] is easily implemented for images represented by regular expressions and is very important for matrix compression shown in next section.

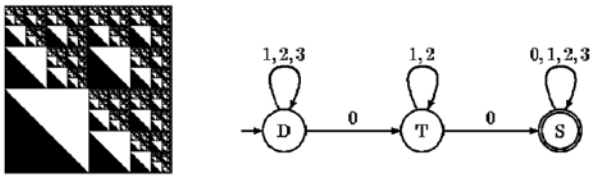

Figure 5. The diminishing triangles defined by $\{1,2,3\}^*0\{1,2\}^*0\Sigma^*$, and the corresponding automaton.

2.2 Construction of Finite Automaton

We have just shown that a necessary condition for black and white multi-resolution image (is evident, that the same reads for binary matrices) to be represented by a regular expression is that it must have only a finite number of different sub-images in all the sub-squares with addresses from Σ^*. We will show that this condition is also sufficient. Therefore, matrices that can be perfectly (i.e. with infinite precision) described by regular expressions (finite automata) are images of regular or fractal character (matrices with many same part). Self-similarity is a typical property of fractals. Any image can by approximated by a regular expression (finite automaton) however; an approximation with a smaller error might require a larger automaton.

Now we will give a theoretical procedure which, given a multi-resolution image or multi-frequency sound, finds a finite automaton perfectly specifying it, if such an automaton exists. (Multi-resolution principle

can be applied to mathematical binary matrices, but only occasionally. For text is not useful!)

Procedure *Construct Automaton*
For given matrix M, we denote M_w the zoomed part of M in the square addressed w. The (sub) matrix represented by state numbered x is denoted by u_x.

Procedure Construct Automaton
  $i = j = 0$
  create state 0 and assign $u_0 = M$
  assume $u_i = M_w$
  loop
    for $k \in \{0,1,2,3\}$ **do**
      if $M_{wk} = u_q$ for some state q **then**
        create an edge labelled k from state i to state q
      else
        $j = j + 1$
        $u_j = M_{wk}$
        create an edge labelled k from state i to the new state j
      end if
    end for
    if $i == j$ **than**
      Stop (all states have been processed)
    else
      $i = i + 1$
    end if
  end loop
end procedure

The procedure *Construct Automaton* terminates if there exists an automaton that perfectly specifies the given matrix and produces a deterministic automaton with the minimal number of states. Our algorithm for non-binary matrix (e.g. grey-scale image, sound, text) is based on this procedure, but it will use evaluated finite automata (as like WFA) introduced in the next section and only replacing binary information 0/1 to a real value (e.g. 256 colour, or greyness image), no creating loop and add some option for set-up compression.

Example: For the Image *diminishing triangles* in Fig. 5, the procedure constructs the automaton shown at the right-hand side of Fig. 5. First, the initial state D is created a processed. For 0 a new state T is created, for 1,2 and 3 a loop to itself. Then state T is processed for 0 a new state S is created, for 1 and 2 a loop to T. There is no edge labelled 3 coming out of T since the quadrant 3 for T (triangle) is empty. Finally the state S (square) is processed by creating loops back to S for all 4 inputs.

In this example we can see, that state D does not contain edge labelled 0 and state T edge labelled 0 and 3. If either one of them is not present several cases may happen, as shown in figure 6. The same example, but without loop is shown in figure 7. Where *0 ... X, X∈ {1,2,3,4}* denote some state of automaton (not necessarily adjoining), and *a, b, c* represent symbol from word w ($a, b, c \in \{0,1,2,3\}$ - represent sub-square of matrices).

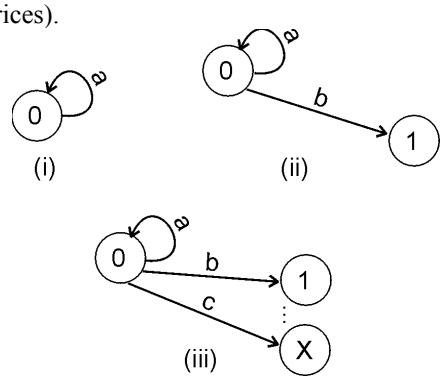

Figure 6. Analysis of possible cases in constructed automaton.

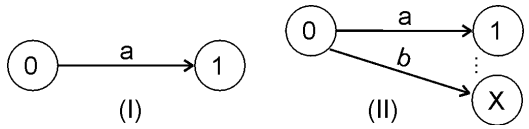

Figure 7. Analysis of possible cases in constructed automaton from figure 6 without loop.

The cases (ii) and (iii) in figure 6 correspond with case (II) in figure 7. We can depress increasing count of state at approach where is not loop.

Procedure for reconstruction matrix from automaton is very simple, than we describe only procedure *Reconstruct matrix* for compressed matrix at next section.

3 Compression

3.1 Basic compression

In this part of chapter, we will demonstrate in brief a method for matrix compression / decompression applicable on construction of matrix storage, or matrix database. Our algorithm is based on algorithm shown in previous section. There lead 4 edges from each node at most and they are labelled with numbers representing matrix part. Every state can store information of average value of given sub-square.

The procedure *Construct Automaton for compression* terminates if there exists an automaton that perfectly (or with the defined error) specifies the given matrix and produces a deterministic automaton with the minimal number of states. The number of states can be a little reduced or extended by changing error threshold. This principle is naturally useful only for matrices where it is possible to apply loss-compression. Changing the part (or only one matrix element) in source matrix changes the number of states in resultant automata.

In the following, we propose one of the simpler and faster algorithms for reconstructing the matrix from the automata. It is recursive algorithm, but it does not dive very deeply.

Procedure *Construct Automaton for Compression*
For given matrix M, we denote M_w the zoomed part of M in the square addressed w. The matrix represented by state numbered x is denoted by u_x.

Procedure Construct Automaton for Compression
$i = j = 0$
create state 0 and assign $u_0 = M$ (matrix represented by empty word and define average value of M)
assume $u_i = M_w$
loop
for $k \in \{0,1,2,3\}$ **do**
if $M_{wk} = u_q$ (or with small error) or if the matrix M_{wk} can be expressed as a part or expanded part of the matrix u_q for some state q **then**
create an edge labelled k from state i to state q
else
$j = j + 1$
$u_j = M_{wk}$
create an edge labelled k from state i to the new state j
end if
end for
if $i == j$ **than**
Stop (all states have been processed)
else
$i = i + 1$
end if
end loop
end procedure

The procedure *Reconstruct Matrix* stops for every automata computed by a procedure *Construct Automaton* (*for Compression*), or other similar algorithm. The procedure computes original matrix only if we process every word from input alphabet and alphabet was computed by loss free procedure. Otherwise, we obtain approximately same matrix. The percent similarity is in relation with length of processed word and original computed word. In figure eight is depicted other approach to storing and reconstructing the image (real evaluated matrix).

Procedure *Reconstruct Matrix* (***RC***)
For given automata A, make matrix M_w the zoomed part of M in the square addressed w. The matrix represented by state number x is denoted by u_x.

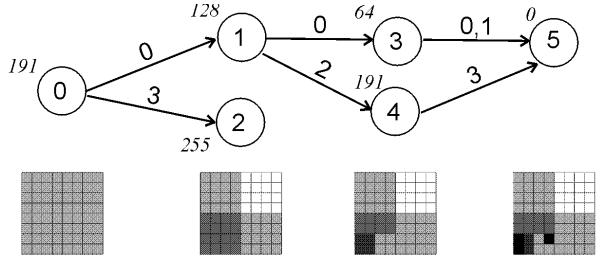

Figure 8. Reconstructing image from root to the leaf node.

Procedure Reconstruct Matrix
$q_o = w = \{\varepsilon\} = M$ (matrix represented by the empty word)
$i(q_o)=1$
$t(q_o)=\varnothing(\varepsilon)$ (average greyness of the matrix M)

recursively for state q **do**
for $u = u0, u1, u2, u3$**do**
$u = M_{ua}$
if $u = t(q_o)$ **and** word has shorter then requested **then**
RC $q = uX$
else
RC $q = q(uY)$
end if
end for
if $u = \{\varepsilon\}$ **or** $u =$ {requested word}
stop
end if
end recursively

where: X denotes part of image
Y is a next part of image

$\varnothing$(x) is average value of the matrix part M_{ux}, we can change to computed value, if we wont that.

Example: Original computed word is $w = \{3203\}$, see figure 1. The length of word is 4 $\Rightarrow$ the number of elements of matrix is 256 (clearly x^2 where x is y^2 where y is z^2 where z is 2). If we process all words we obtain all matrix elements, if we process only first 2 symbols from the word we obtain only matrix with 2^2 elements. The average value of element (if value is from interval 0-255) with address $w = \{32\}$ is approximately 16, then we obtain matrix, where sub-square with address $w = \{32\}$ have every elements equal 16. This principle is useful only for images, sounds or el. signals. Text will be destroyed. The procedure *Reconstruct Matrix* can reconstruct all matrix elements (same dimension as original) with defined error if we use word with first 2 symbol and append empty word $w=\{32\varepsilon\varepsilon\}$.

3.2 Compression and changes

Now we describe, how to re-compress the matrix after some updates.

If we compress the matrix with traditional algorithm and some element is changed, we must re-compress all matrices every time. But if we represent matrix as a Finite State Automata, we can change / re-compress only corresponding part with changed element.

There exist at least three basic solutions for selection of corresponding part of Finite State Automata or corresponding sub-square of source matrix:

1) Re-calculating the biggest corresponding sub-square:

This approach leads to a big quantum of data manipulation (as much as one quarter), but with this approach we can reach the high compression ratio. Method is useful for all types of source matrices.

2) Re-calculating the least corresponding sub-square:

This approach re-calculates the least quantum of data (radix ten elements), but with this approach we gain very small compression ratio and often changes lead to the grow in size of the automata. Compression is after that disutility, but if we want only obtain the interesting information from our resultant automata, this method is useful too. This approach is useful for all types of source matrices.

3) Re-calculating the optimal corresponding sub-square:

Retrieval of such sub-square may be difficult, but in most cases, it shows that it hasn't sense to work with sub-square greater than three or four least corresponding sub-squares. Naturally, it greatly depends on the character of the matrices. If we know that our matrices contain many equal blocks (e.g. wiring gate in microprocessor), we can state the amount of levels, which we should still take in consideration. This choice naturally has not influence on algorithm, but only on machine time and resultant size of compressed matrices.

4 Resultant automata

4.1 Coalescence

There exist many methods for assembling matrices represented by finite automata. We will depict one of better ones, namely assembling resultant automata in direction from leaf node to the root. Suppose a couple of matrices and their representation by means of final state automata computed with procedure 1, depicted in figure 9. For simplicity, black and white pictures with resolution 8 × 8 pixel were used. Moreover, the states of automata contain also the average greyness of corresponding picture parts. These values are in interval 0 − 255, where 0 represents black and 255 white colour.

Example: It is clear that images in figure 9 have some common parts, highlighted in following figure 10. These common parts could be joined into the same state in composed resultant automata (see figure 10 on the bottom). The corresponding automata with images in figure 9 have some common state and edge labelled with same symbol. There exists some similar or the equal word *w* depicting the way from the root to the corresponding part represented by a state.

This principle can be used for no-loss compression. Additional information can be obtained from the structure of the resultant automaton, for example the information about the similarity of the stored matrices, common lines, and alternatively equal parts in matrix. We can easily get the group of equal matrix parts. The algorithm for assembling resultant automata together is very simple.

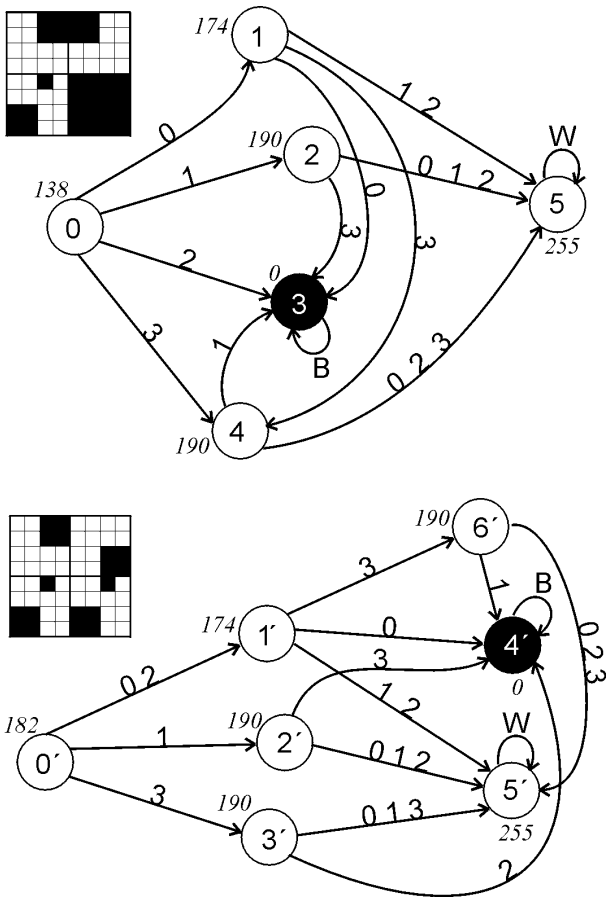

Figure 9. Two sample images with common parts and the corresponding automata.

Procedure *Composition Automaton for Storage*

For given automaton *A* and automaton *B* - resultant automaton from previously composition compute new resultant automaton $B' \in A \cup B$ and combine similar parts of both.

Procedure Composition Automaton (Automaton *A*, Stored automaton *B*)
Assign state q_x from *A* to the corresponding state in stored automaton *B*.
If such case does not exists, assign a new state and take q_{x+1} from *A*.
end if
foa all state of automaton *A* **do**
if not exist edge from state q_i labelled with same word *w* as edge from correspond state in stored automaton **then**
create a new edge labelled *w* to a new state *i*
otherwise
take next edge
end if

if all edge from actual state is processed, take next state
end if
end for
end procedure

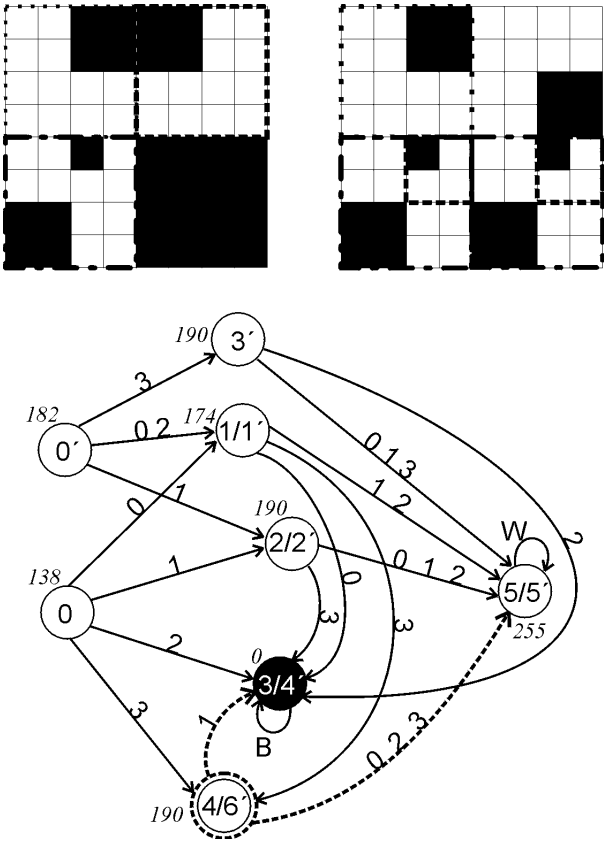

Figure 10. Composite automata from figure 9 respecting the common parts.

The procedure *Composition Automaton for Storage* stops for every automaton computed by a procedure *Construct Automaton for Compression* or other similar algorithm. Resultant automaton is in most cases smaller than original automata and contains interesting information from both automata. In some cases, the resultant automaton could be larger, but if we will store more and more matrices, the resultant automaton will increase less and less. On the other side we can store two (or more) almost same matrices in automaton with almost non-increasing number of states. It is clear, when we are storing matrices of the same domain (e.g. medical super-sound radiograph, X-ray, pictures of building, el. signal, similar noise, etc.) we could obtain an interesting knowledge about stored matrices hand in hand with saving the memory space.

4.2 Interesting information in matrix

If we store some similar matrices in one automaton, we can obtain interesting information about changes in resultant automaton. Concretely, if our resultant automaton has more states then the new states represent the differences between input matrices. With procedure *Construct Automaton for Compression,* we can control the type of acceptable (or unacceptable) changes, e.g. trivial differences in value, small noise, etc. With this, we can separate interesting changes from those that are rather to be ignored.

For example, in medicine, we can obtain many similar images and only medical specialist can mark an interesting parts. We can help him or her to reduce the number of non-interesting changes and of course notice him or her to some small differences, which can be important, but hard to find.

With procedure *Composition Automaton for Storage,* we can increase the ability to make the difference between interesting and non-interesting parts of the resultant automaton. We can even do it by involving the tolerance in value (average value), or similarity of words, etc.

If we store our source matrix in more than one automaton, we can focus on our interesting part of the matrix and there compute the profundity automaton. On other part of matrix, we can compute automaton with less number of states. For this purpose, we can use the pattern matrix shown in table 1, where the values in cells are the counts of profundity of automaton, which represents that part of matrix. This principle can be used only for loss compression (e.g. images, signals, etc.). The part with less count of states stores much fewer information than the part with more states.

It is clear, with this principle we can save much more space and also preserve high information value of our data. We can transfer only interesting part of matrix or any nearest part and save machine time or network capacity. It is sufficient to choose a state from resultant automata, which represents our interesting part of the matrix, and operate with this as with the root. This principle is used with the principle automata composition. Pattern may be arbitrary.

Now we have a background for using finite state automata as a database with included information.

3	3	3	3	3	3
3	5	5	5	5	3
3	5	10	10	5	3
3	5	10	10	5	3
3	5	5	5	5	3
3	3	3	3	3	3

Table 1: Pattern of approach with 36 automata.

4.3 Representation

In this chapter we introduce ideas how to represent Finite State Automata as a matrix, as a table for SQL database and as a XML structure.

1) Matrix representation

Suppose automata as depicted in figure 5. Such automata could be transformed into matrix with at least

four columns. The first to four columns contain destination for each transition. We can append fifth column with the source state. If the matrix is encoded then the average value is usually stored in the sixth column. To support additional features we may store subsequent information but for basic functionality, it is enough. The matrix itself is built using following procedure:

Procedure *Store Automaton to Matrix*
Input is resultant automaton from procedure *Construct Automaton for Compression* or other similar algorithm.

Procedure Store Automaton to Matrix (automaton A)
  for all $q_i \in Q$ **do**
    create new line in matrix and assign value *i* in *source* if appended
    for all edge e_j outgoing from q_i **do**
      assign value *i* in column *j* *
    end for
  end for
end procedure

where: *Q* is se of states of automaton A
j starts from 0 but 3 at most
***** unused transitions leave empty
source - number of processed state

The matrix representation of the automata presently used is straightforward encoding of states and transitions that could be supplied with additional information.

For automaton in figure 5:
state D = 1
T = 2
S = 3 and S is final state

2	1	1	1
3	2	2	
3	3	3	3

Table 2: Matrix representation

Simple resultant matrices without additional information contain only integral numbers from interval (0 - *count of state*). To reconstruct the source matrix from such representation is very simple and fast. We can use procedure *Reconstruct Matrix* above-mentioned or other similar procedure. This representation of FSA is useful for direct hardware-based processing or for running on machines with simple operating system. It is easy to convert it to any other representation and to manipulate with it. Computation costs no many machine time and space. Over all this advantages, the matrix representation is not so good for database manipulation (e.g. searching, updating, etc.).

For such purposes, there are some other representations.

2) Table representation

Table representation is very similar to matrix representation. Instead of matrix, we use common table to store all information needed to describe the automaton. We store simply the *source* state, average value, etc. in the table. Procedure *Store Automaton to Table* is adequate to procedure *Store Automaton to Matrix* then we describe only resultant table at Table 3.

The first to fifth columns contain information about the state and the sixth column contains the average value (in our example it contains the average colour of sub-square in image depicting the diminishing triangles, where background of image is black colour denoted with 0 and other colour is only white - 255.

0	1	2	3	source	a. value
T	D	D	D	D	96
S	T	T		T	128
S	S	S	S	S	255

Table 3: Table representation of automaton in fig. 5

This representation is best for storing in SQL database particularly if we append additional interesting information in additional column (e.g. average value, state depth, most similar state, etc.) witch emphasise included benefit. It is very fine for indexing and facilitates manipulating very large resultant structure (composed FSA).

3) XML representation

This representation is very useful for using Finite State Automata as means to database with added interesting information (e.g. multimedia database, database of medical images, el. signals, trends, etc.) and is best for sharing resultant FSA on internet/intranet. It is very easy to manipulate with only a part of the automaton, in this structure. There are many approaches of how to solve the problem and we offer the on of the simpler. For more about XML please read [9].

Example: The automaton representing a diminishing triangles in figure 5 represented by simplified XML structure.

```
<automaton>
  <root>
    <state average="96" w0="1">
      <state average="128" w0="1" w3="0">
        <state average="255">
        </state>
      </state>
    </state>
  </root>
</automaton>
```

Example: The exemplary composed automaton represented by simplified XML structure.

```
<automaton>
  <root>
```

```
    <state average="X" w0="S" w1="S" w2="S"
w3="S">
      <state average="X " w0="S" . . .  w3="S">
        <state . . . >
          . . .
            <state average="X ">
              <!-- final state for average X -->
            </state>
          . . .
            <state average="Y">
              <!-- final state for average Y -->
            </state>
          . . .
        </state>
      </state>
    </state>
  <!-- one family of matrices -->
  </root>
. . .
  <root>
    <state average="X" w0="S" w1="S" w2="S"
w3="S">
      <state ....>
          . . .
          . . .
          . . .
      </state>
    </state>
  <!-- other family of matrices -->
  </root>
. . .
  <!-- other family of matrices -->
</automaton>
```

Where:

X is average value of part of matrix

wC, C∈{0,1,2,3} is sub-square (pixel), if wC is not present, then sub-square with symbol C is represented itself (with the state witch represent - recursively)

S is value: 0 - not present, 1 - present

In this structure we can store every automata computed by procedure *Construct Automaton for Compression* or *Composition Automaton for Storage* or other similar algorithm. Final state can obtain other information e.g. following automaton increasing deep.

5 Conclusions and future work

In this paper we have presented a Finite State Automata for storing and compressing data represented as a matrix. FSA allow us to capture a large class of data represented as matrices and obtaining interesting information without using other algorithm to compute it. Additionally, data stored in FSA save more space, make it possible to access and manipulate with them without the decompressing process. Storing data in automaton allows us to send via network only the part of data of our interest. We do not need to have available the whole matrix (e.g. picture) if we are interested in a small part of it only. Additional benefit is included. Composition allows us to use FSA as a database of matrices (e.g. multimedia database) with the ability to search of interesting parts of our data.

In our future work, we focus on manipulating with data in database (at various structures, e.g. table or XML), describe language or a set of tools that is able to query stored data.

References

[1] Alur, R. and Dill, D. L. A Theory of Timed Automata. In *Theoretical Computer Science*, 126(2):183–235, 1994.

[2] Daniela BERARDI, Fabio DE ROSA, Luca DE SANTIS and Massimo MECELLA. Finite State Automata as Conceptual Model for E-Services. *In Integrated Design and Process Technology, IDPT-2003*, June 2003.

[3] K. Culik II and J. Kari. Image compression using weighted finite automata. In *Computers & Craphics*, 17:305–313, 1993.

[4] K. Culik II and V. Valenta. Finite automata based compression of bi-level and simple color images. In *Computers & Craphics*, 21:61–68, 1997.

[5] K. Culik II and J. Kari. Image compression Using Weighted Finite Automata, in Fractal Image Compression. In *Theory a Techniques*, Ed. Yuval Fisher, Springer Verlag, pp 243-258, 1994.

[6] J.E.Hopcroft and J.D.Ullman. Introduction to automata theory, languages and computation. In *Addison-Wesley*, 1979.

[7] Marian Mindek. Finite State Automata and Images. In *WOFEX 2004*, PhD Workshop, Ed. V. Snášel, ISBN: 80-248-0596-0, 2004

[8] Marian Mindek. Finite State Automata and Image Recognition. In *DATESO 2004*, Ed. V. Snášel, J. Pokorný, K. Richta, pp 132-143, ISBN: 80-248-0457-3, 2004

[9] W3C (2004) XML Protocol. *XML Protocol Web Page* (link checked January, 10th 2005): http://www.w3.org/XML/