KYBERNETIKA — VOLUME *38* (2002), NUMBER 1, PAGES 91–103

IMPLEMENTATION OF DIRECTED ACYCLIC WORD GRAPH

MIROSLAV BALÍK

An effective implementation of a Directed Acyclic Word Graph (*DAWG*) automaton is shown. A *DAWG* for a text T is a minimal automaton that accepts all substrings of a text T, so it represents a complete index of the text. While all usual implementations of *DAWG* needed about 30 times larger storage space than was the size of the text, here we show an implementation that decreases this requirement down to four times the size of the text. The method uses a compression of *DAWG* elements, i. e. vertices, edges and labels. The construction time of this implementation is linear with respect to the size of the text, a search for a specific pattern is done in a linear time with respect to the size of the pattern. This implementation preserves both good properties of the *DAWG* automaton.

1. INTRODUCTION

String matching is one of the most frequently used tasks in text processing. With the increased volume of processed data, which usually consists of unstructured texts, the importance of data qualification technologies is increasing. This is the reason why indexing structures are constructed for static texts that support pattern matching in a linear time with respect to the length of the pattern.

Although some indexing structures have a linear size with respect to the length of the text, this size is high enough to disable practical implementation and usage. This size depends on implementation details, on the type of the text and on the type of the automaton used. For *suffix tree* the size is rarely smaller than $10n$ bytes, where n is the length of the text. Other structures are a Directed Acyclic Word Graph (*DAWG*) automaton (size about $30n$ bytes) and its compact version *CDAWG* (size about $10n$ bytes). Stefan Kurtz in [10] shows a number of known implementations of these automata together with experimental evaluations and a number of references.

Other types of indexing structures are usually smaller, *suffix arrays* [6] (size $5n$ bytes), level compressed tries [2] (size about $11n$ bytes), *suffix cactuses* – a combination of suffix trees and suffix arrays [9] (size $9n$ bytes), and *suffix binary search trees* [8] (size about $10n$ bytes).

An automaton is usually stored as a graph, vertices represent states of the automaton and edges represent transitions. A state is then represented by an index into a transition table or as a memory position referenced by edges. Edges are stored as a part of the vertex they start from. Thus it is possible to locate an edge with a specific label in a constant time with respect to the number of vertices and edges. Each edge contains an information about the vertex it leads to, and its label, which is one symbol in the case of DAWG, or a sequence of symbols in the case of CDAWG and suffix trees. Every sequence used as a label is a substring of a text, so it can be represented by a starting and ending positions of this substring.

An implementation presented in this paper uses a compression of elements of the graph representing the automaton to decrease space requirements. The 'compression' is not a compression of the whole data structure, which would mean to perform decompression to be able to work with it, but it is a compression of individual elements, so it is necessary to decompress only those elements that are necessary during a specific search. This method is applicable for all homogeneous automata[1] and it can be generalized to all automata accepting a finite set of strings.

The whole graph is a sequence of bits in a memory that can be referenced by pointers. A vertex is a position in the bit stream where a sequence of edges originating from the vertex begins. These edges are pointers into a bit stream, they point to places where the corresponding terminal vertices are located. Vertices are stored in a topological ordering[2], which ensures that a search for a pattern is a one-way pass through the implementation of the graph structure.

Each vertex contains an information about labels of all edges leading to it and the number of edges that start from it. Since it is possible to construct a statistical distribution of all symbols in the text, we can store edge labels using a Huffman code [7]. We can use it also to encode the number of edges starting from a vertex and because the most frequent case is when only one edge starts from a vertex, it is dealt with as a special case.

The approach presented here creates a *DAWG* structure in three phases. The first phase is the construction of the usual *DAWG* (Section 3), the second phase is topological ordering (or re-ordering) of vertices (Section 4), which ensures that no edge has a negative "length", where length is measured as a difference of vertex numbers. The final phase is encoding and storing the resulting structure. Encoding of each element is described in Section 5.

We have used six files in the experiments. The former three files are texts on UNIX. This files are written in English. The later three files are randomly generated texts. This texts have similar properties as DNA sequences.

2. BASIC DEFINITIONS

An alphabet is a finite set of symbols. A *string* over a given alphabet is a finite sequence of symbols. An empty sequence of symbols is called an *empty string* and it will be denoted as ε. Let $T = t_1t_2\dots t_n$ be a string (*text*) over a given alphabet

[1]Homogeneous automata have all transitions to a specific state labeled with the same symbol.

[2]A topological ordering of a vertex is such a numbering of vertices that ensures that each edge starts from a vertex with a lower number and ends at a vertex with a higher number.

A. A pattern $P = p_1p_2 \dots p_m$ is a substring of a text T iff such two natural numbers i, j, $j = i+m+1$, $i > 0$, $j \leq n$ exist such that $P = t_it_{i+1} \dots t_j$. To answer whether a pattern P is a substring (subpattern, subword, factor) of a text T is a pattern matching problem.

The major advantages of *DAWG* are:

— it has a linear size limited by the number of vertices ,which is less than $2|n|-2$; the number of edges is less than $3|n|-4$, where $n > 1$ is the length of the text [4],

— it can be constructed in $\mathcal{O}(n)$ time [4],

— it allows to check whether a pattern occurs in a text in $\mathcal{O}(m)$ time, where m is the length of the pattern. Pattern matching algorithm is shown in Figure 1.

```
1. State $q := q_0$; $i := 1$;
2. while(($i < m + 1$) and ($q \neq NULL$))
3.     do $q := Successor(q, P[i])$; $i := i + 1$; end
4. if($i < m + 1$) then NO // Pattern does not occur in Text
   else YES // Pattern occurs in Text
```

Fig. 1. Matching Algorithm.

Index i points to the processed symbol of the pattern, function $Successor(q, P[i])$ returns successor of state q using the edge labeled by symbol $P[i]$. If there is no edge from state q labeled by this symbol, the *NULL* value is returned.

The whole algorithm consists of the mail loop. The number of iteration is in the worst case m. In each iteration appropriate edge is searched. This search depends only on the size of the input alphabet, it is independent on the number of edges in the whole graph. The resulting time complexity of the algorithm is $\mathcal{O}(m)$.

3. CONSTRUCTION OF *DAWG*

There are many ways of constructing *DAWG* from text, more details can be found for example in [4]. The method used here is the on-line construction algorithm. An example of *DAWG* constructed using this algorithm for an input text $T = aabbabb$ is shown in Figure 2.

During this phase a statistical distribution of symbols in the text is created. A statistical distribution of the number of edges at respective vertices is also created.

4. TOPOLOGICAL ORDERING

The *DAWG* structure is a directed acyclic graph. This means that its vertices can be ordered according to their interconnection by edges. Such an implementation

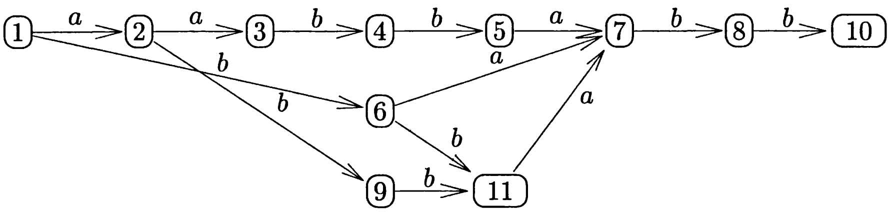

Fig. 2. *DAWG* for the text $T = aabbabb$.

that keeps all the information about edges starting from a vertex only in the vertex concerned while storing the vertices in a given order guarantees that every pattern matching will result in a single one-way pass through this structure.

The problem of such topological ordering can be solved in linear time. At first, for each vertex its input degree (the number of edges ending at the vertex) is computed, next a list of vertices having an input degree equal to zero (the list of roots) is constructed. At the beginning, this list will contain only the initial vertex. One vertex is chosen from the list, gets the next number in the ordering and for all vertices accessible by an edge starting at this vertex their input degree is decreased by one. Then such vertices that have a zero input degree are inserted into the list. And this goes on until the list is empty. The order of the vertices, which determines the quality of the final implementation, obtained this way depends on the strategy of choosing a vertex from the list. Several strategies were tested and the best results were obtained using the LIFO (last in – first out) strategy, because using this strategy for vertices with only one outgoing edge gets its successor next number (if this successor has been inserted into the roots list). With the vertices having only one edge, and this edge points to the successor in topological ordering, is dealt as a special case, see Section 5.

The original *DAWG* shown in Figure 2 has been topologically ordered using the LIFO strategy and the resulting graph is depicted in Figure 3.

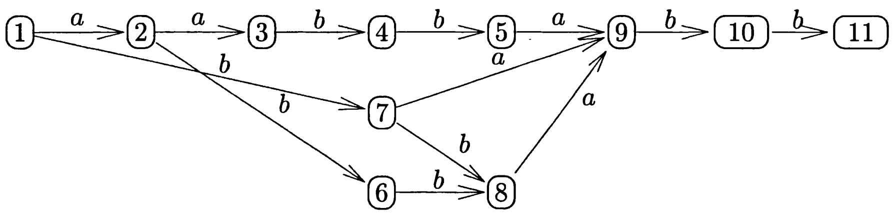

Fig. 3. The results of topological ordering.

5. ENCODING

The *DAWG* graph is encoded element by element (elements are described later in this section). It starts with the last vertex according to the topological order (as described above) and progresses in the reverse order, ending with the first vertex of the order. This ensures that a vertex position can be defined by the first bit of its representation and that all edges starting at the current vertex can be stored because all ending vertices have already been processed and their address is known.

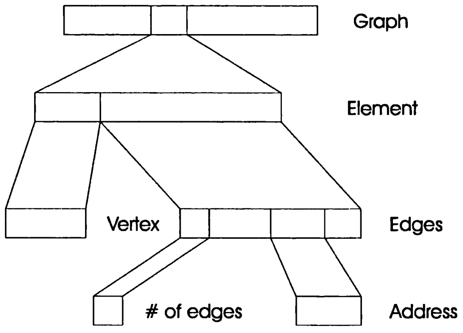

Fig. 4. Implementation – Data Structures.

The highest building block is a *graph.* It is further divided into single *elements.* Each *element* consists of two parts: a *vertex* and an *edge.* A *vertex* carries out information on a label of all edges ending at it. A Huffman code is used for coding symbols of the alphabet. An *edge* is further split into a *header* and an *address order.* A *header* carries out information on the *number of addresses* – edges belonging to a respective vertex. A distribution of edge counts for all vertices can be obtained during the construction of *DAWG.* This makes possible to use a Huffman code for header encoding, but Fibonacci encoding ([12]) is sufficient as well, though one must expect a substantial amount of small numbers. An *address* is the address of the first bit of the element being pointed to by an appropriate edge. It is further split into two parts, one describing the length of the other part, which is a binary encoded address.

5.1. Symbol encoding

A code of an *element* (vertex and corresponding edges) starts with a code of the symbol for which it is possible to enter the vertex. The best code is the Huffman code. The following table shows experimental results. The second column shows the size of text. The Huffman code can be based either on counts of symbol occurrences in the text (third column), or proportionate representation at individual vertices in the *DAWG* graph (fourth column). The latter better suits the implementation.

File Name	|X|	Symbol Count $\frac{\|Bits\|}{\|Symbol\|}$	Proportionate Repre. $\frac{\|Bits\|}{\|Symbol\|}$
TEXT1	21818	4.771186	4.770672
TEXT2	53801	4.264782	4.264746
TEXT3	81054	4.588081	4.587633
RANDOM1K	1000	7.532672	7.531844
RANDOM10K	10051	7.809383	7.807136
RANDOM100K	100447	7.831894	7.831680

The average number of bits necessary to store one symbol is calculated for symbols representing the vertices of the graph. It can be observed that the two methods of encoding provide similar results. For example, using the latter method for encoding the file TEXT3 will result in improvement of only 0.00045 bits per symbol, which is 0.0098 % with respect to the value obtained using the first method.

Decoding begins at the *root* of the coding tree, and follows a left edge when a '0' is read or a right edge when a '1' is read. When a leaf is reached, the corresponding symbol is output.

5.2. Encoding of number of edges

The code of the *number of edges* is another item. Even this value can be obtained prior to encoding. A typical example of a distribution of numbers of edges for two input text files is shown in Figure 5.

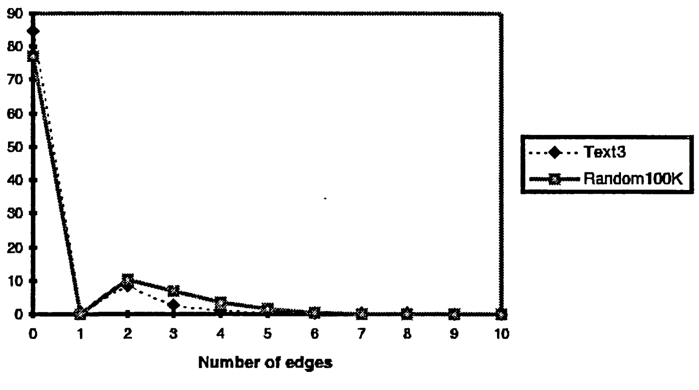

Fig. 5. Edge count distribution (# of edges, # of vertices in percent).

In Figure 5 vertices with just one edge starting at them were further divided into two groups: the first group is formed by vertices having just one edge leading to the next vertex according to given vertex ordering (included in the group Edge count = 0), and the second group is formed by vertices having just one edge leading anywhere else (Edge count = 1). The first group can be easily encoded by the value of Edge count.

The figure also shows that more than 84 % of all vertices belong to the first group. This means that the codeword describing this fact should be very short. It will be only one bit long using Huffman coding. Other values of edge counts are represented by more bits according to the structure of the input text.

The smallest element of *DAWG* represents a vertex with just one edge ending at the next vertex. For TEXT3 it is 5.6 (4.6 per symbol + 1 bit per edges) bits on average. The fact that *DAWG* consists mainly of such elements was used in the construction of the *Compact DAWG structure (CDAWG)* derived from the general *DAWG*, more details can be found in [5].

The process of decoding of Number of edges is similar to Symbol decoding.

5.3. Edge encoding

The last part of the graph element contains references to vertices that can be accessed from the current vertex. These references are realized as relative addresses with respect to the beginning of the next element. The valid values are non-negative numbers. To evaluate them it is necessary to know the ending positions of corresponding edges. This is why the code file is created by analysing *DAWG* from the last vertex towards the root in an order that excludes negative edges. If we wanted to work with these edges, we would have to reserve an address space to be filled in later when the position of the ending vertex is known.

The address space for a given edge depends on the number of bits representing the elements (vertices) lying between the starting and ending vertices. As the size of these elements is not fixed (the size of the dynamic part depends mainly on element addresses), it is impossible to obtain an exact statistical distribution of values of these addresses, which we obtained for symbols and edges. A poor implementation of these addresses will result in the fact that elements will be more distant and the value range broader.

Yet it is possible to make an estimation based on the distribution of edge lengths (measured by the number of vertices between the starting and ending vertices). In this case the real address value might be only q-times higher on average, where q is an average length of one *DAWG* element. The first estimation of optimal address encoding is based on the fact that the number of addresses covered by k bits is the same for $k = 1, 2, \ldots, t$, where t is the number of bits of the maximum address. We will use an address consisting of two parts: the first part will determine the number of bits of the second part, the second part will determine the distance of the ending vertex in bits. The simplest case is when the addresses are of a fixed length, then the length of an average address field is $r = s + t$, where $s = 0$, which means that $r = t$ actually. Another significant case is a situation when the number of categories is t, then $s = \lceil \log_2 t \rceil$.

When s is chosen from an interval $s \in \langle 0, \lceil \log_2 t \rceil \rangle$, the number of categories is 2^s, the number of address bits of the ith category is $\frac{t*i}{2^s}$. An average address field length is then

$$r = s + \sum_{i=1}^{2^s} \frac{t * i}{2^s}.$$

When we rearrange this formula, we obtain

$$r = s + t\frac{2^s + 1}{2^{s+1}}.$$

When the address length is fixed and the number of categories varies, this function has a local minimum for

$$2^s = \frac{t \ln 2}{2}.$$

If we know t, we can calculate s as

$$s = \log_2(t \ln 2) - 1$$

Table 1. Address encoding.

s	t	Optimal |X|
1	6	3B
1 and 2	8	11B
2	12	171B
2 and 3	16	2.7kB
3	23	350kB
3 and 4	32	180MB
4	46	2.9TB
4 and 5	64	$8 \cdot 10^{17}$B
5	92	$2 \cdot 10^{26}$B

Table 1 shows optimal values of t for given values of s as well as address limits when it does not matter if we use a code for s or $s+1$ categories. The estimation of the input file length assumes that the code file is three times greater than is the length of the input text, and that the code file contains the longest possible edge, which connects the initial and the last vertices. This observation is based on experimental evaluation.

It can be seen that the value $s = 3$ is sufficient for a wide range of input text file lengths, which guarantees a simple implementation, yet it leaves some space for doubts about the quality of the approach used. Or is it so that edge lengths are not spread uniformly in the whole range of possible edge lengths (1 to the maximum length)? The answer can be found in Figure 6.

Figure 6 does not contain edges ending at the next vertex (with respect to the actual vertex) as they are dealt with in a different way. It can be clearly observed

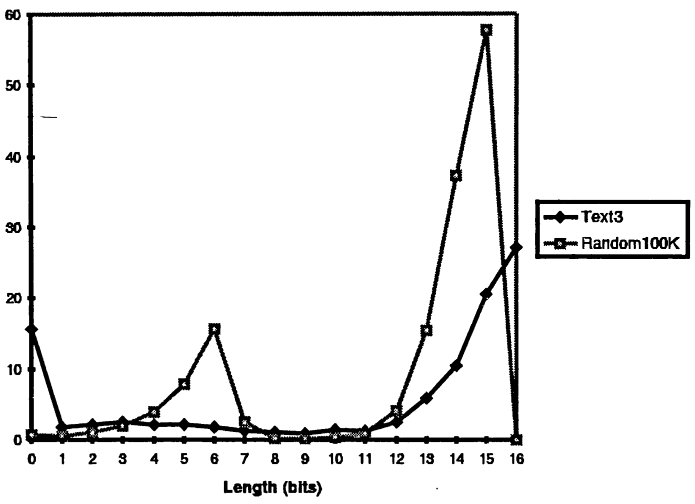

Fig. 6. Edge Length Distribution (Length – bits, # of edges in percent).

that the assumption of uniformity of the distribution is not quite fulfilled. Nevertheless categories can be constructed in the way that supports the requirement of the minimal average code word length. Figures 7 and 8 depict the real distribution of address lengths for two ways of encoding. The first is a code with two categories, one encoding addresses with 15 bits, the other with 30 bits. The second way regularly divides address codes into eight categories by four bits.

Both ways of address encoding provide similar results..The relevancy with respect to the statistical distribution of edges is obvious, the peaks being shifted by three or four bits to the right.

Edge decoding depends on the number of categories used for encoding. When eight categories are used, three bits are used for symbol length code – $s = 3$. We read these three bits as an integer n. Then we calculate the number of bits that represent an edge address as $t := (n + 1) * const$, where *const* is based on the length of *CodeFile*. Then, we read n bits from *CodeFile* as an integer, and this number is the address.

Figure 9 shows the contribution of individual parts to the overall length of the resulting code.

The biggest portion is occupied by edge encoding, even though the majority of edges is included in the edge count encoding. The test was performed for encoding with eight address categories.

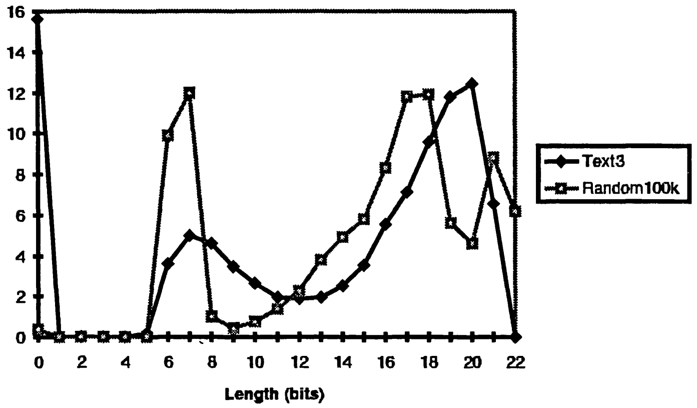

Fig. 7. Address length distribution – Two categories (Length – bits, # of addresses in percent).

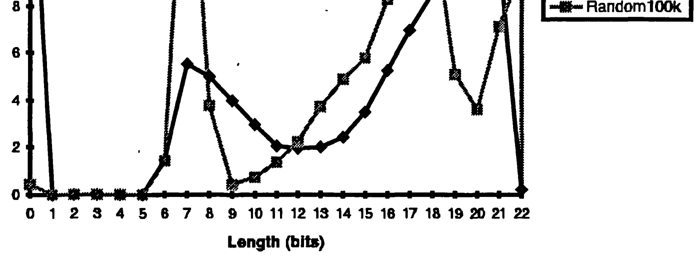

Fig. 8. Address length distribution – Eight categories (Length – bits, # of addresses in percent).

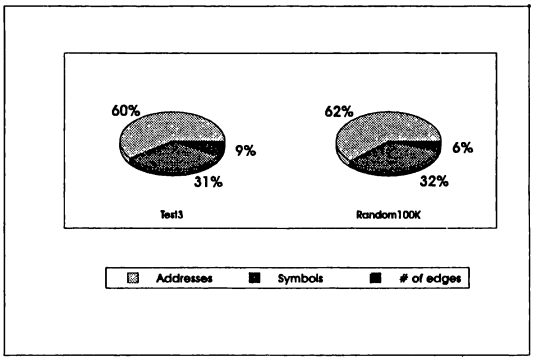

Fig. 9. The influence of code lengths to the overall length of the code file The influence of code lengths to the overall length of the code file.

5.4. Matching algorithm over the implementation

The matching algorithm over the implementation is shown in Figure 10. Each element (symbol labeling, number of edges encoding, edge encoding) is decoded in the moment of use.

Variable *Ptr* points to the implementation, each bit of the implementation is addressable using *Ptr*. The beginning of the implementation has the value $Ptr = 0$. Function *DecodeNum*(*Ptr*) decodes number of edges. Decoding process starts from the root of Huffman tree, the input bit stream is taken from the position *Ptr*. The number of used bits is stored for the function *Update*(*Ptr*). The executing process in function *DecodeLabel*(*Ptr*) is analogical, but for decoding is used Huffman tree constructed for edge labeling instead of Huffman tree constructed for number of edges.

If the appropriate edge from the currently processed vertex is searched there are all possible vertices visited. Each symbol by the visited vertex is compared with the labeling of the appropriate edge. The maximal number of visited vertices equals to the maximal number of edges from a vertex, to the size of the input alphabet. The time complexity of execution of this function is $\mathcal{O}(m)$.

5.5. Complexity

DAWG can be created using the on-line construction algorithm in $\mathcal{O}(n)$ time [4]. Vertex re-ordering can be also done in $\mathcal{O}(n)$ time, encoding of *DAWG* elements as described above can also be done in $\mathcal{O}(n)$ time. Moreover, vertex re-ordering can be done during the first or third phase. This means that the described *DAWG* construction can be performed in $\mathcal{O}(n)$ time.

The time complexity of searching in such an encoded *DAWG* is $\mathcal{O}(m)$, see [3].

```
 1. $Build\ decoding\ trees\ from\ the\ implementation$;
 2. $Ptr := 0$; $//Ptr\ldots$pointer into the implementation
 3. $i := 1$;
 4. $initialize\ Stack$;
 5. $\boldsymbol{while}((i < m + 1)\ \boldsymbol{and}\ (Ptr \neq NULL))\ \boldsymbol{do}$
 6.  $\boldsymbol{begin}\ num := DecodeNum(Ptr); Update(Ptr)$;
 7.    $\boldsymbol{if}(num =$ “$Only\ one\ edge\ to\ the\ next\ vertex$”$)\ \boldsymbol{then}\ Push(Stack, 0)$;
 8.      $\boldsymbol{else}\ \boldsymbol{while}(num > 0)\ \boldsymbol{do}$
 9.        $\boldsymbol{begin}\ Push(Stack, DecodeEdgeLength(Ptr))$;
10.          $Update(Ptr)$;
11.          $num := num - 1$;
12.        $\boldsymbol{end}$; //end while
13.    $found := FALSE$;
14.    $\boldsymbol{while}((Empty(Stack) \neq TRUE)\ \boldsymbol{and}\ (found \neq TRUE))\ \boldsymbol{do}$
15.      $\boldsymbol{begin}\ Ptr := Ptr + Pop(Stack)$;
16.        $\boldsymbol{if}(DecodeLabel(Ptr) = P[i])\ \boldsymbol{then}$
17.          $\boldsymbol{begin}\ Update(Ptr)$;
18.            $i := i + 1$;
19.            $found := TRUE$;
20.            $initialize\ Stack$;
21.          $\boldsymbol{end}$; //end if
22.      $\boldsymbol{end}$; //end while
23.    $\boldsymbol{if}(found = FALSE)\boldsymbol{then}\ Ptr := NULL$;
24.  $\boldsymbol{end}$; //end while
25. $\boldsymbol{if}(i < m + 1)\ \boldsymbol{then}$ NO // Pattern does not occur in Text
26.        $\boldsymbol{else}$ YES // Pattern occurs in Text
```

Fig. 10. Matching Algorithm over the implementation.

6. RESULTS

The following table shows the results for the set of test files.

File Name	$\lvert X \rvert$	$\lvert Y_1 \rvert$	$\lvert Y_2 \rvert$	$\frac{\lvert Y_1 \rvert}{\lvert X \rvert}$	$\frac{\lvert Y_2 \rvert}{\lvert X \rvert}$
TEXT1	21818	602928	500385	345.4 %	286.7 %
TEXT2	53801	1459973	1201342	339.2 %	279.1 %
TEXT3	81054	2304026	1906376	355.3 %	294.0 %
RANDOM1K	1000	25687	23258	321.1 %	290.7 %
RANDOM10K	10051	244703	219157	304.3 %	272.6 %
RANDOM100K	100447	3843810	3177465	478.3 %	395.4 %

The size of the text is denoted as X. The size of the code file for two sets of addresses is denoted as $|Y_1|$, $|Y_2|$ is relevant for the code using eight address categories. Both values are in bits and do not contain information on the Huffman encoding used. The size of these data does not depend on the size of the input file.

7. CONCLUSION

The results show that the ratio of code file size vs. the input file size is 3:1. This number changes very little with the rising size of the input file to the detriment of the code file. If the ratio rose as high as 4:1, a CD-ROM with the capacity of 600MB could contain one code file for an text of the maximal size up to 150MB, which is a more than seven-times better result than the one obtained by the classical approach.

ACKNOWLEDGEMENTS

I would like to thank the referees for their valuable remarks to the original version of this paper.

(Received May 12, 2000.)

REFERENCES

[1] J. Adámek: Coding. MVŠT XXXI, SNTL, Prague 1989 (in Czech).
[2] A. Anderson and S. Nilson: Efficient implementation of suffix trees. Software–Practice and Expirience *25* (1995), 129–141.
[3] M. Balík: String Matching in a Text. Diploma Thesis, CTU, Dept. of Computer Science and Engineering, Prague 1998.
[4] M. Crochemore and W. Rytter: Text Algorithms. Oxford University Press, New York 1994.
[5] M. Crochemore and R. Vérin: Direct construction of compact directed acyclic word graphs. In: CPM97 (A. Apostolico and J. Hein, eds., Lecture Notes in Computer Science 1264), Springer–Verlag, Berlin 1997, pp. 116–129.
[6] G. H. Gonnet and R. Baeza-Yates: Handbook of Algorithms and Data Structures. Pascal and C. Addison–Wesley, Wokingham 1991.
[7] D. A. Huffman: A method for construction of minimum redundancy codes. Proc. IRE *40* (1952), 9, 1098-1101.
[8] R. W. Irving: Suffix Binary Search Trees, Technical Report TR-1995-7, Computing Science Department, University of Glasgow 1995.
[9] J. Kärkkäinen: Suffix cactus: A cross between suffix tree and suffix array. In: Proc. 6th Symposium on Combinatorial Pattern Matching, CPM95, 1995, pp. 191–204.
[10] S. Kurtz: Reducing the Space Requirment of Suffix Trees. Software–Practice and Experience *29* (1999), 13, 1149-1171.
[11] B. Melichar: Approximate string matching by finite automata. In: Computer Analysis of Images and Patterns (Lecture Notes in Computer Science 970), Springer–Verlag, Berlin 1995.
[12] B. Melichar: Fulltext Systems. Publishing House CTU, Prague 1996 (in Czech).
[13] B. Melichar: Pattern matching and finite automata. In: Proceedings of the Prague Stringology Club Workshop '97, Prague 1997.

Ing. Miroslav Balík, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University, Karlovo nám. 13, 121 35 Praha 2. Czech Republic.
e-mail: balikm@fel.cvut.cz