Implementation of DAWG

Miroslav Balík

Department of Computer Science and Engineering
Faculty of Electrical Engineering
Czech Technical University
Karlovo nám. 13
121 35 Prague 2
Czech Republic

e-mail: balikm@cslab.felk.cvut.cz

Abstract. Let T be a text over a fixed alphabet A. Then an automaton can be created in a linear time that accepts all *substrings* that occur in text T. The ratio of the size of the implementation of this automaton *(factor automaton, DAWG)* and of the input text is in usual cases *14:1*. This paper shows a method of implementing *DAWG* that reduces this ratio down to *4:1* while preserving good qualities of the automaton, which is linear time of its construction with respect to the length of the input text and linear time of checking that a pattern is present in the text with respect to the length of the pattern.

Key words: finite automata, approximate string matching, DAWG, factor automaton

1 Introduction

Let $T = t_1 t_2 \dots t_n$ be a *text* over a given alphabet A. An alphabet is a finite set of symbols. A *word* (string) over a given alphabet is a finite sequence of symbols. An empty sequence of symbols is called an empty word and it will be denoted as ε. A pattern $P = p_1 p_2 \dots p_m$ is a substring of a text T iff such two natural numbers i, j exist that $P = t_i t_{i+1} \dots t_j$. To answer whether a pattern P is a substring (subpattern, subword, factor) of a text T is a look-up problem.

A graph that represents a finite automaton accepting all substrings occurring in a given text is called *DAWG* (Directed Acyclic Word Graph).

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,
- it can be constructed in the time $\mathcal{O}(n)$,
- it allows to check whether a pattern occurs in a text in $\mathcal{O}(m)$, where m is the length of the pattern. Algorithm is shown on Fig. 1.

The basic idea of the implementation that is about to be described is that because the majority of edges contained in *DAWG* connect neighbouring vertices (according to a given topological order), these edges are worth implementing as a single bit saying whether such an edge is present or not. Another *DAWG* property is that all edges ending at such vertex are labeled by the same symbol of the alphabet, thus the labelling symbol can be transferred to vertices. Finally, when a statistical analysis of symbols and of the number of edges starting at a given vertex is performed, a suitable encoding can be employed to yield another reduction of *DAWG* size.

1. State $Q := Q_0$; $i := 1$;
2. ***if***$(i = m + 1)$**END** - Pattern occurs in Text
3. $Q := Successor(Q, P[i])$; $i := i + 1$
4. $if(Q = nil)$ **END** - Pattern does not occur in Text
   else goto(2)

Figure 1: Matching Algorithm

2 Implementation

The approach presented here creates a *DAWG* structure in three phases. The first phase is the construction of the usual *DAWG* graph, the second phase is topological ordering (or re-ordering) of vertices, 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. More details about the implementation and the results presented here can be found in [Bal98].

2.1 Construction of *DAWG*

There are many ways of constructing *DAWG* from text, more details can be found for example in [Cro94]. The method used in this article is the on-line construction algorithm. An example of *DAWG* constructed using this algorithm for an input text $T = aabbabb$ is shown below:

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.

2.2 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 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.

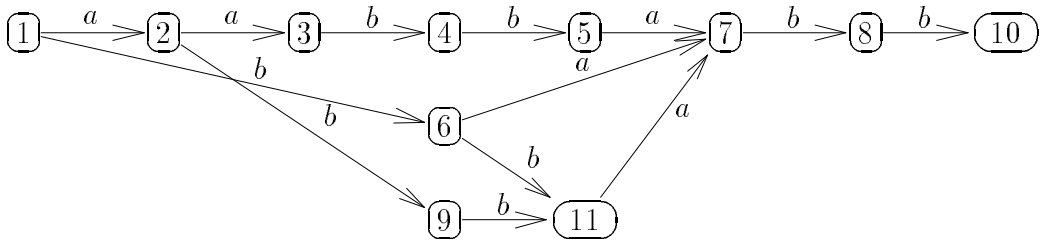

Figure 2: *DAWG* for the text $T = aabbabb$.

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 determined, 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 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, for more details see [Bal98].

The original *DAWG* shown in Fig. 1 will be reordered using the LIFO strategy and the resulting graph is depicted in Fig. 3

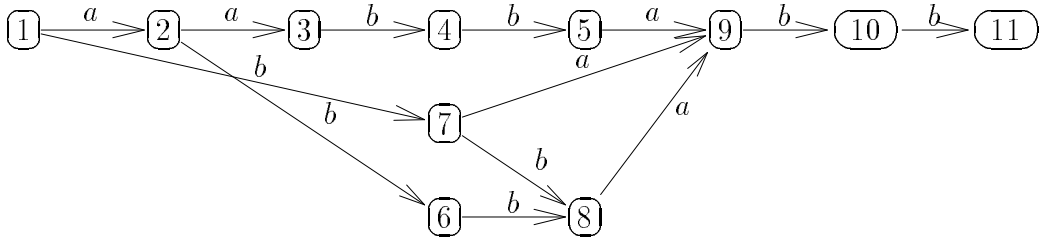

Figure 3: The result of reordering.

2.3 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.

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 an

information on a label of all edges ending at it. A Huffman code is used for coding of the respective symbol 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 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 to two parts, one describing the length of the other part, which is a binary encoded address.

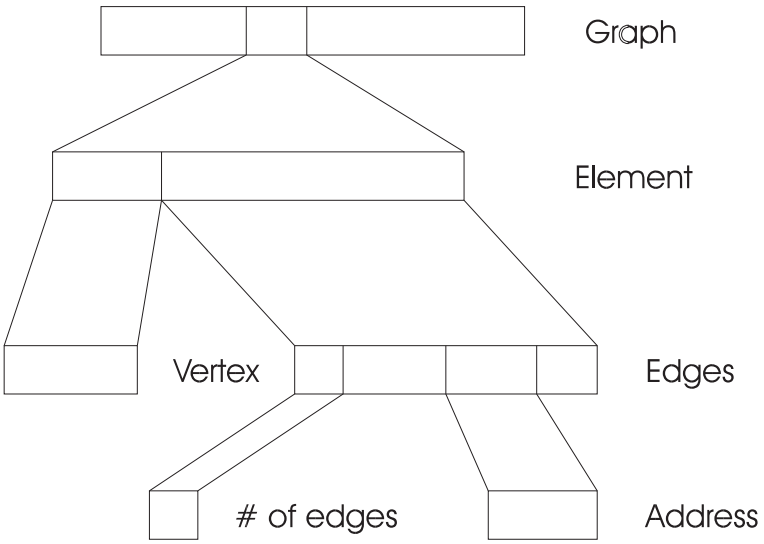

Figure 4: Implementation - Data Structures

2.4 Matching Algorithm

1. Build coding trees from the $CodeFile$
2. $Ptr := 0$; {Ptr . . .pointer into the $CodeFile$}
   $i := 1$; initialize $Stack$
3. ***if***$(i = m + 1)$**END** - Pattern occurs in Text
4. Decode num . . .number of edges starting in state Q, update Ptr.
5. ***if*** $(num =$ "*Only one edge to the next vertex*") $Push(Stack, 0)$
   else while$(num > 0)$ {decode one edge and push it to $Stack$; $num := num - 1$; update Ptr }
6. ***if*** ($Stack$ is empty)**END** - Pattern does not occur in Text
7. $Ptr := Ptr + Pop(Stack)$
8. Decode $label$ from Ptr
9. ***if*** $(label = P[i])$ { update Ptr; i := i+1; ***goto***(3)}
   else goto(6)

2.5 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, which can be based either on counts of symbol occurrences in the text, or proportionate representation at individual vertices. 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.

2.6 Symbol Decoding

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 encountered, the corresponding symbol is output.

2.7 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 the Fig. 6.

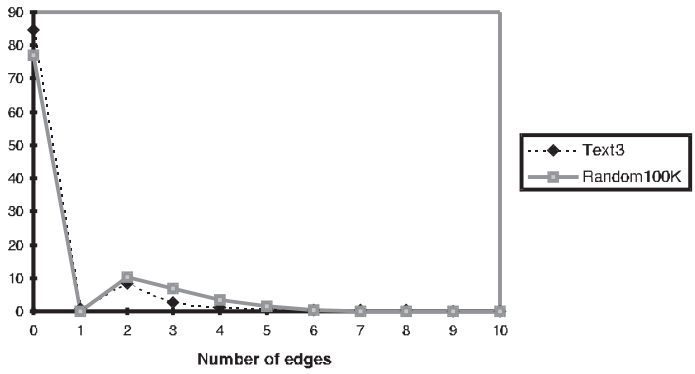

Figure 5: Edge count distribution (# of edges, # of vertices - %)

In the figure Fig. 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 code word 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 [Cro97].

2.8 Number of Edges Decoding

The process is similar to Symbol decoding.

2.9 Edge Encoding

The last part of the graph element contains references to vertices that can be accessed from the current vertex. These references are realised 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 $k = 1, 2, ..., t$, 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 $i-th$ 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

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

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

The above table 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 the following figure.

The figure 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 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. The other two figures depict the real distribution of address

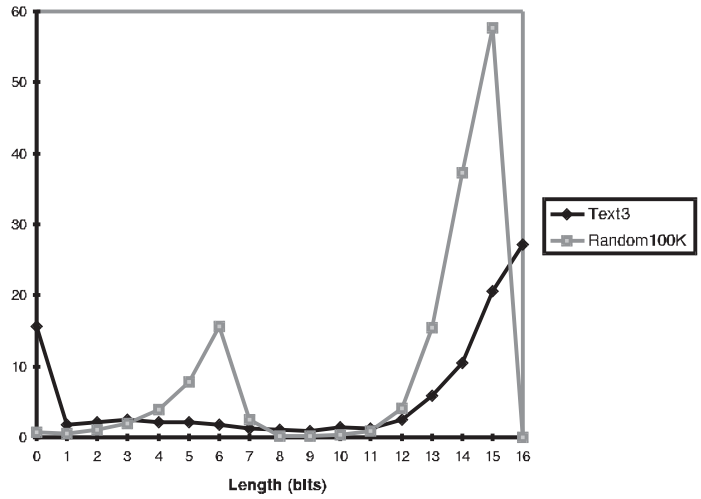

Figure 6: Edge Length Distribution (Length - bits, # of edges - %)

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.

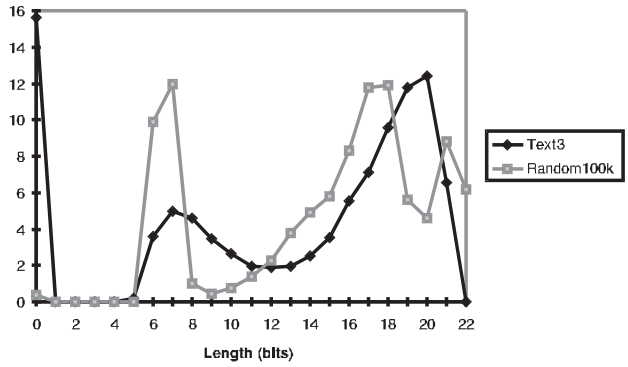

Figure 7: Address length distribution - Two categories (Length - bits, # of addresses - %)

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.

2.10 Edge Decoding

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 $CodeFile$. Than, we read n bits from $CodeFile$ as an integer, and this number is the address.

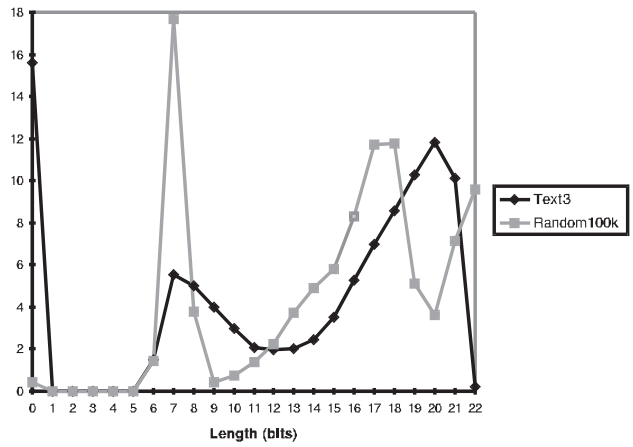

Figure 8: Address length distribution - Eight categories (Length - bits, # of addresses - %)

The following picture describes the contribution of individual parts to the overall length of the resulting code.

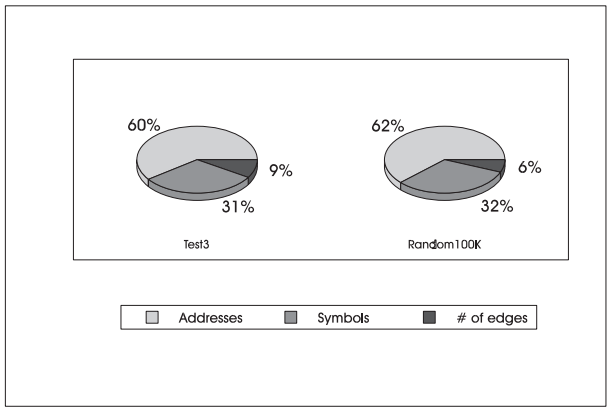

Figure 9: The influence of code lengths to the overall length of the code file

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.

2.11 Complexity

DAWG can be created using the on-line construction algorithm in $\mathcal{O}(n)$ time [Cro94]. Vertex re-ordering can be also done in $\mathcal{O}(n)$ time, encoding of *DAWG* elements as described above can also be done in $\mathcal{O}(n)$ [Bal98]. Moreover, vertex re-ordering can be done during the first or third phases. This means that the described *DAWG* construction can be performed in $\mathcal{O}(n)$.

The time complexity of searching in such an encoded *DAWG* is $\mathcal{O}(m)$ [Bal98].

3 Results

File Name	$\|X\|$	$\|Y_1\|$	$\|Y_2\|$	$\frac{\|Y_1\|}{\|X\|}$	$\frac{\|Y_2\|}{\|X\|}$
TEXT1	21818	602928	500385	345.4 %	286.7 %
TEXT2	53801	1459973	1201342	339.2 %	279.1 %
TEXT3	81054	2304026	1906376	355.3 %	294.0 %
MOD4005.TXT	1246946	-	11818341	-	118.5 %
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 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.

4 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 input file of the maximal size up to 150MB, which is a more than three-times better result than the one obtained by the classical approach.

References

[Ada89] J. Adamek: *Coding.* MVŠT XXXI, SNTL, Prague, 1989, in Czech.

[Bal98] M. Balík: *String Matching in a Text.* Diploma Thesis, CTU, Dept. of Computer Science & Engineering, Prague, 1998.

[Cro94] M.Crochemore, W.Rytter: *Text Algorithms*, Oxford University Press, New York, 1994.

[Cro97] M.Crochemore and R.Vérin: *Direct Construction Of Compact Directed Acyclic Word Graphs.* in (CPM97, A. Apostolico and J. Hein, eds., LNCS 1264, Springer–Verlag, 1997) pp 116-129.

[Me95] B. Melichar: *Approximate String Matching By Finite Automata.* Computer Analysis of Images and Patterns, LNCS 970, Springer, Berlin 1995.

[Me96] B. Melichar: *Fulltext Systems.* Publishing house CTU, Prague, 1996, in Czech.

[Me97] B. Melichar: *Pattern Matching and Finite Automata.* Proceedings of the Prague Stringology Club Workshop '97, Prague, 1997.