

# Fast Practical Multi-Pattern Matching

Maxime CROCHEMORE \*      Artur CZUMAJ †

Leszek GASIENIEC †      Stefan JAROMINEK †

Thierry LECROQ ‡      Wojciech PLANDOWSKI †

Wojciech RYTTER †

## Abstract

The main result of the paper is the construction of a very fast multi-pattern matching algorithm, called in the paper DAWG-MATCH. The algorithm is of Boyer-Moore type. Previous algorithm of this type is the Commentz-Walter algorithm. The DAWG-MATCH algorithm behaves better than Commentz-Walter algorithm. We combine the ideas of two algorithms: the Aho-Corasick algorithm, and the Reverse Factor algorithm from Crochemore et alii. The new algorithm performs at most  $2|text|$  inspections of text characters, and is very fast on the average. We give some experimental evidence of its good behavior for random words, against the Commentz-Walter algorithm. The algorithm is especially simple for a single pattern: in this case the Aho-Corasick algorithm can be replaced by the strategy of the Knuth-Morris-Pratt algorithm. The basic tool in the algorithm DAWG-MATCH is the directed acyclic word graph. This graph is usually used as representation of the text to be scanned, but, in our case we use it to represent the set of reverse patterns.

## 1 Introduction

We consider the multiple string matching problem: finding all occurrences of a finite set  $P$  of string patterns in a text  $t$  of length  $n$ . Finding all occurrences of elements of  $P$  is a problem that appears in bibliographic search and in information retrieval. The first algorithm to solve this problem in  $O(n)$  was the Aho-Corasick (AC algorithm, for short) [AC 75], which can be viewed as a generalization of the Knuth-Morris-Pratt algorithm (KMP algorithm) [KMP 77], designed for a single pattern. As for one pattern, the Boyer-Moore algorithm (BM algorithm) [BM 77] has a better behavior in practice than the KMP algorithm. Commentz-Walter developed an algorithm combining the ideas of AC and BM algorithms ([Co 79a, Co 79b]). A complete version can be found in [Ah 90]. Later, Uratani [Ur 88], and Baeza-Yates and Régnier [BR 90] developed similar algorithms.

In this paper, we show how to use the power of directed acyclic word graphs (DAWG's) for finding a finite set of patterns. Such graphs are used to represent all factors (subwords) of a given word.

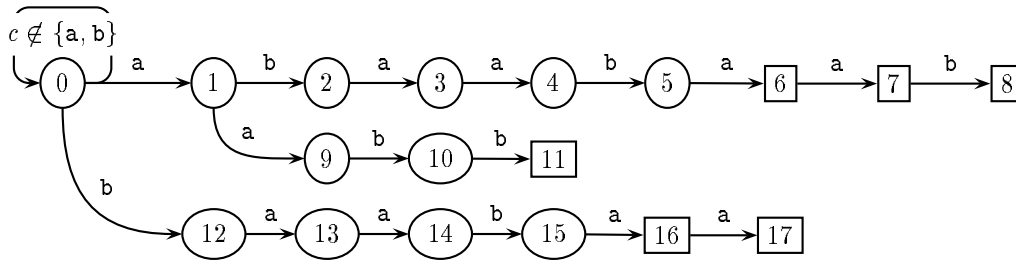
We recently presented a family of algorithms for finding one pattern using the DAWG of the reverse pattern [C-R 94]. Direct extension of this algorithm to solve the multi-pattern matching problem gives an algorithm running in quadratic time in

---

\*Institut Gaspard Monge, Université de Marne-la-Vallée, 2 rue de la Butte Verte, 93166 Noisy-le-Grand Cedex, France

†Institute of Informatics, Warsaw University, ul. Banacha2, 00-913 Warsaw 59, Poland

‡Laboratoire d'Informatique de Rouen, Université de Rouen, Facultés des Sciences et des Techniques, 76821 Mont-Saint-Aignan Cedex, France



$s$	0	1	2	3	4	5	6	7	8
$f(s)$	0	0	12	13	14	15	16	17	5
$s$	9	10	11	12	13	14	15	16	17
$f(s)$	1	2	12	0	1	9	10	3	4
$s$	6	7	8	11	16	17			
$output(s)$	baaba	baabaa	abaabaab	aabb	baaba	baabaa			

Figure 1: The Aho-Corasick machine for  $P = \{abaabaab, aabb, baabaa, baaba\}$ .

the worst case. However, the algorithm has a good behavior in practice. Combining this idea with the AC algorithm, we present a new algorithm which performs at most  $2n$  inspections of text characters, and which is simultaneously very fast on the average.

In the case of one pattern, the same technique applies. It gives a new algorithm combining the strategies of the KMP algorithm and the Reverse Factor algorithm ([Le 92, C-R 94]). This new algorithm performs at most  $2n$  inspections of text characters, and it is more simple than the Turbo Reverse Factor algorithm presented in [C-R 94]. Like the Reverse Factor type algorithms, the new algorithm is optimal on the average, with  $O((n \log m)/m)$  inspections of text characters ( $m$  is the length of the pattern).

## 2 The preprocessing phase

The preprocessing phase of DAWG-MATCH algorithm concerns the set  $P$  of patterns. It consists both in building the Aho-Corasick machine  $A$  for all the patterns of  $P$ , and in building the DAWG  $D$  for the reverse patterns of  $P$ . We shortly describes these two data structures.

An Aho-Corasick machine  $A$  is a deterministic finite state automaton  $(Q, \delta, f, s_0, T)$  where  $Q$  is a finite set of states,  $\delta$  is the transition function,  $f$  is the failure function,  $s_0$  is the initial state, and  $T$  is a set of the accepting states (see [Ah 90] for details). In Figure 1 we present the machine for the example set of patterns  $P = \{abaabaab, aabb, baabaa, baaba\}$ . Accepting states are in square boxes.

The DAWG-MATCH algorithm needs to be able to compute a shift corresponding to each state of the AC machine. It will enable the algorithm to perform the optimal shift which corresponds to the matched prefix associated with the state. Each state  $s$  is associated with a prefix  $w$  of words in  $P$ , which is composed by the characters spelling the unique path between  $s_0$  and  $s$ . The length of the shift associated to  $s$  is  $|p| - |w|$ , where  $p$  is the shortest pattern in  $P$  that has prefix  $w$ . The table *Shift* can be defined intuitively as follows:

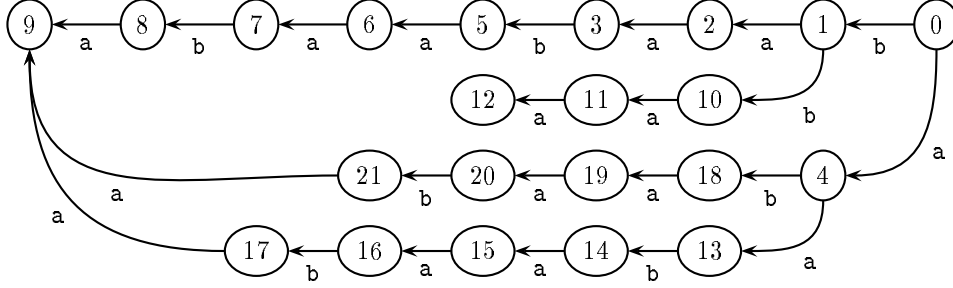


Figure 2: The DAWG for reverse patterns of  $P = \{abaabaab, aabb, baabaa, baaba\}$ .

for state  $s$ ,  $Shift[s]$  is the minimal shift which guarantees that there is no occurrence of a pattern in the skipped area, assuming that  $s$  corresponds to the longest prefix of a pattern which is a suffix of the text scanned so far (the whole information from the scanned part relevant to further processing).

More formally,  $Shift[s]$  is the length of the minimal path between state  $s$  and an accepting state different from  $s$  (the path can use either the transition function or the failure links).

It is easy to see that, if we are given the pattern matching machine with failure links in the table  $f$ , then the table  $Shift$  can be constructed in time proportional to the number of states the following rule :

during the computation of the trie, when the letter  $p[i]$  of a pattern  $p$  of length  $m$  is processed,

if a new state  $s$  is created then, if  $i \neq m$  then  $Shift[s] := m - i - 1$  else  $Shift[s] := m$ ,

if the corresponding state  $s$  already exists then  $Shift[s] := \min(m - i - 1, Shift[s])$ ;

afterwards, during the breadth-first traversal of the trie for computing the failure links, when the value of  $f(s)$  is available,

$Shift[s] := \min(Shift[s], Shift[f(s)])$ .

For the above example of the Figure 1, values of the table  $Shift$  are:

$s$	0	1	2	3	4	5	6	7	8
$Shift[s]$	4	3	4	3	2	1	1	1	1
$s$	9	10	11	12	13	14	15	16	17
$Shift[s]$	2	1	4	4	3	2	1	1	2

The other data structure used in the DAWG-MATCH algorithm is a DAWG. The DAWG is also a deterministic finite state automaton. It recognizes all the factors of the reverse patterns of  $P$ . For the above example of set  $P$ , the automaton is presented in Figure 2. To avoid reversing the pattern in the picture, transitions go from right to left.

The aim of the preprocessing is to construct the two following functions :

**function** AC(*initpos*, *minpos*, *state*);

**begin**

scan the text  $t[initpos, minpos]$  left to right with the Aho-Corasick machine  $A$  starting with state *state*;

continue scanning the text with the Aho-Corasick machine  $A$

until  $Shift[state] \geq$  length of the shortest pattern divided by 2;

```

    report all positions, as matches, where  $A$  is in an accepting state;
    return the last position scanned, and the last state of  $A$ ;
end function;

```

```

function DAWG( $pos, critpos$ );
  begin
    scan the text  $t[critpos, pos]$  from right to left with the DAWG,
    until there is no transition for the next symbol or the position
     $critpos$  is reached;
    return the last position scanned;
  end function;

```

### 3 The search phase

The search phase combines the techniques used by the AC and RF algorithms. Its strategy consists first in reading from right to left a segment of the text as far as it is a factor of at least one pattern of  $P$ . Then, using the fact that this segment is a part of one pattern, it is read from left to right using the Aho-Corasick machine, in order to both, report matches, and compute lengths of shifts. The tool that enables us to match segments of the text from right to left against factors of patterns of  $P$  is the DAWG for all reverse patterns of  $P$  (see [B-M 87]).

Assume that both the AC machine  $A$  for the patterns, and the dawg  $D$  for the reverse patterns are constructed. We describe the general situation encountered during the search:

we have recognized a prefix  $u$  of length  $l$  of at least one pattern from  $P$ . The occurrence of  $u$  in the text ends at position  $critpos$ . We suppose that we also know the corresponding state in the AC machine, denoted by  $state$ . We denote by  $m$  the length of the shortest pattern  $p$  in  $P$  such that  $u$  is a prefix of  $p$ . Let  $pos$  be equal to  $critpos + m - l$ . The action at this stage can be described informally as follows:

**Substage I** Scan with  $D$  the characters  $t[critpos + 1, pos]$  from right to left. If we are able to reach successfully the critical point  $critpos$  using  $D$ , it means that the factor  $t[critpos + 1, pos]$  is factor of one pattern in  $P$ .

**Substage II** If we reach the critical point during Substage I, then, we use the AC machine starting with  $state$  and with the factor  $t[critpos + 1, pos]$  from left to right until a sufficiently large shift is possible.

If  $t[critpos + 1, pos]$  is not a factor of any pattern of  $P$ , it means that, at a character  $t[pos - k]$  with  $pos - k > critpos$ , there is no transition in  $D$ . Then, we use the AC machine starting with the initial state and with the factor  $t[pos - k + 1, pos]$  from left to right.

After that, it gives a new prefix  $u$ , and a new value for  $state$ . The next stage starts here (see Figure 3).

At the first stage, we scan the factor  $t[1..m_{inf}]$  of the text from right to left using the DAWG  $D$ . Length  $m_{inf}$  is the length of the shortest pattern in  $P$ . The first value of  $state$  is the initial state  $s_0$  of the AC machine  $A$ . Figure 4 shows the succession of stages of the algorithm DAWG-MATCH.

The algorithm DAWG-MATCH is presented below.

```

Algorithm DAWG-MATCH;
  begin

```

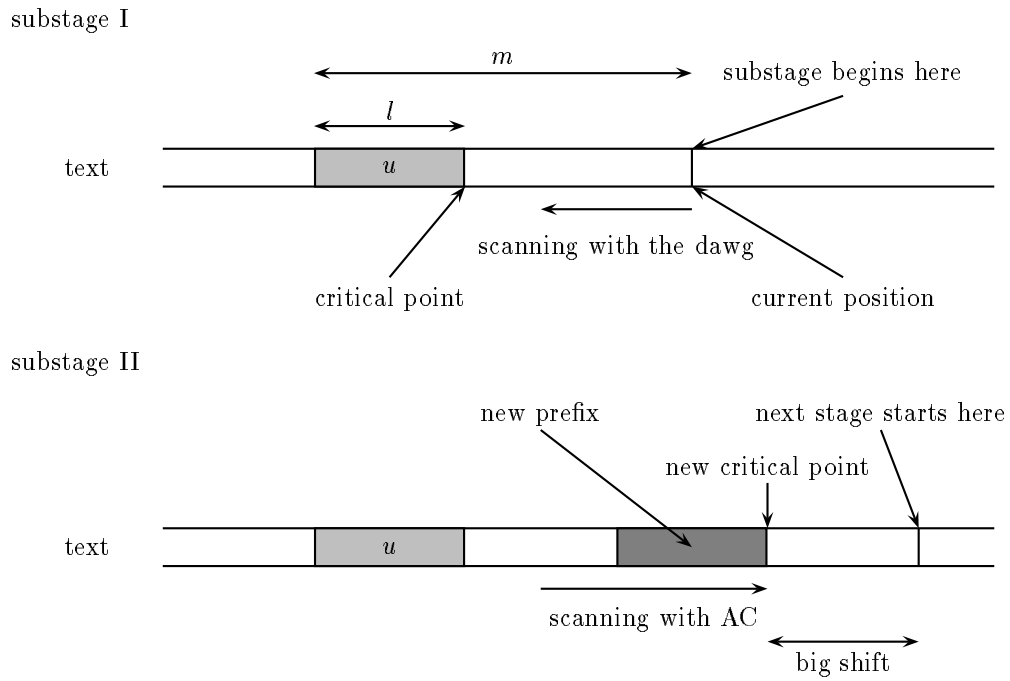


Figure 3: General situation during the search phase.

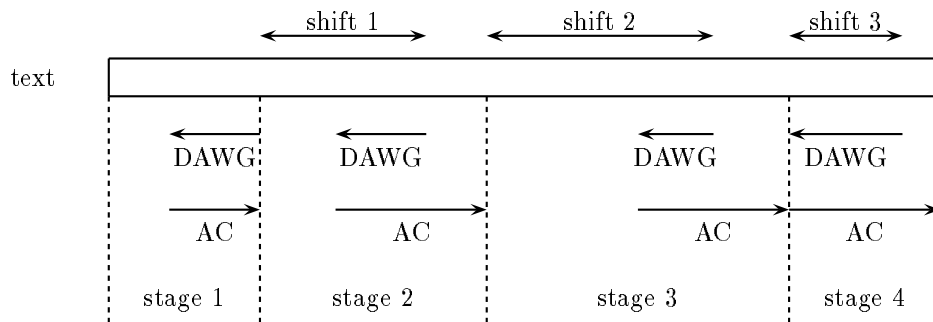


Figure 4: A succession of stages of the search phase.

```

preprocessing phase:
  build the Aho-Corasick pattern matching machine  $A$  with
    the table  $Shift$ ;
  construct the DAWG  $D$  for reverse patterns;
search phase:
   $pos :=$  length of shortest pattern;
   $critpos := 0$ ;
   $state :=$  initial state of  $A$ ;
  while  $pos \leq n$  do {
     $newpos :=$  DAWG( $pos, critpos + 1$ );
    if  $newpos > critpos$  then  $state :=$  initial state of  $A$ ;
    ( $pos, state$ )= $AC(newpos, pos, state)$ ;
     $critpos := pos$ ;
     $pos := pos + Shift[state]$ ;
  }
end.

```

### Example

$P = \{abaabaab, aabb, baabaa, baaba\}$ ,  
 $t = abaabaabac\dots$

**stage 1**  $critpos = 0$  and  $pos = 4$ , the algorithm DAWG-MATCH first scans  $t[1..4] = abaa$  from right to left with the DAWG starting with the initial node and stopping with node 15. Then it scans  $t[1..4]$  from left to right with the AC machine starting with the initial state and ending with state 4.  $Shift[4] = 2$  so it shifts to the right by 2.

**stage 2**  $critpos = 4$  and  $pos = 6$ , the algorithm scans  $t[5..6] = ba$  from right to left with the DAWG starting with the initial node and stopping at node 18. Then it scans  $t[5..6]$  from left to right with the AC machine starting with state 4 and ending at state 6 (it outputs the pattern **baaba**).  $Shift[6] = 1 <$  length of the shortest pattern divided by 2 ( $= 2$ ), then it scans  $t[7] = a$  and reaches state 7 (it outputs the pattern **baabaa**),  $Shift[7] = 1$  so it scans  $t[8] = b$  and reaches state 8 (it outputs the pattern **abaabaab**),  $Shift[8] = 1$  so it scans  $t[9] = a$  (it takes the failure link from state 8 to state 5) and reaches state 6 (it outputs the pattern **baaba**). Then it scans  $t[10] = c$  and reaches state 0,  $Shift[0] = 4$  so it shifts to the right by 4.

## 3.1 Worst-case time complexity analysis

During one stage of the search phase of the DAWG-MATCH algorithm, each text character between positions  $critpos + 1$  and  $pos$  are scanned once with the DAWG, and once with the AC machine. At the end of the stage, some characters at the right of position  $pos$  are scan only once with the AC machine. For the next stage,  $critpos$  is set to the rightmost position already scanned, and no character at the left of  $critpos$  is scanned again. So, obviously, the algorithm DAWG-MATCH performs at most  $2n$  inspections of text characters.

## 3.2 Average-case time complexity analysis

The average complexity of the algorithm DAWG-MATCH is similar to the average complexity of the RF algorithm (see [C-R 94]). Denote the length of the shortest pattern by  $m$ , and the total length of all patterns by  $M$ . Assume that  $M$  is polynomial with respect to  $m$ ,  $M \leq m^k$ , where  $k$  is constant. Let  $s \geq 2$  be the size of the alphabet. The text (in which the pattern is to be found) is random.

If the shortest pattern is short, for example if it is a single-letter pattern, then the average complexity is  $\Omega(n)$ . Also if  $M$  is big, for example when the set of patterns consists of almost all strings of size  $m$ , then  $M = \Omega(s^m)$ , and the average complexity is  $\Omega(n)$ . Hence the sublinear average complexity of the algorithm DAWG-MATCH can be expected if  $m$  is reasonably big and  $M$  is reasonably small (polynomial on  $m$ ).

**Definition 1** *The length of a shift is the number of symbols between new critical position  $critpos$  and new position  $pos$ . This is the number of new symbols of the text to be read in the next iteration.*

**Proposition 2** *Each shift in the algorithm has length at least  $\Omega(m)$ .*

**Proof.**

Each iteration ends on work of the AC automaton to obtain a prefix of the multipattern  $P$  which is shorter than  $m/2$ . After that the number of elements between new critical position  $critpos$  and new position  $pos$  is at least  $m/2$ . This completes the proof.

By the proposition 2 it is obvious that there are not more than  $O(n/m)$  shifts in the algorithm. We will show that the expected number of comparisons with symbols of the text per one iteration is  $O(\log_s m)$ .

**Proposition 3** *There exists a constant  $C$  such that reading  $C \log_s m$  new symbols of the text we obtain a subword of the multipattern  $P$  with the probability not greater than  $1/m^k$ .*

**Proof.**

There is at most  $m^{2^k}$  different subwords in the multipattern  $P$  and there exist at least  $s^l$  different words of length not greater than  $l$  over the size of the alphabet. Each of these words can be obtained with equal probability during reading new symbols of the text because the text is random.  $s^l \geq m^{3^k}$  if  $l \geq \log_s(m^{3^k}) \geq C \log_s m$  for  $C \geq k$ . Then the probability that the read word is a subword of the multipattern is less than  $1/m^k$ . This completes the proof.

**Lemma 4** *The expected number of inspections of text characters of the AC machine to obtain a prefix shorter than  $m/2$  is  $O(\log_s m)$ .*

**Proof.**

Let us assume that the probability of the event that the number of inspections of text characters of the AC machine to obtain a prefix shorter than  $m/2$  is between  $rC \log_s m$  and  $(r+1)C \log_s m$  is less than  $(1/m^k)^r$  by the proposition 3. It follows that the expected number of inspections of text characters  $\leq C \log_s m + \sum_r (r+1)(1/m^k)^r C \log_s m \leq O(\log_s m)$ .

**Lemma 5** *The expected number of inspections of text characters in one stage of the algorithm is  $O(\log_s m)$ .*

**Proof.**

There are two cases.

Case A

The DAWG stops before it has read  $C \log_s m$  symbols. The probability of this case is not less than  $1 - 1/m^k$  by the proposition 3. The number of inspections of text characters in this case is obviously not greater than  $2C \log_s m$  because the AC machine starts with the empty prefix and has only at most  $C \log_s m$  symbols to read.

Length of patterns	Commentz-Walter	DAWG-MATCH
10	4.4497	1.1576
20	2.555	1.6819
30	2.2186	1.1075
40	1.8997	0.8458
50	1.7599	0.7016
60	1.5846	0.5077
70	1.5605	0.5222
80	1.4681	0.5171
90	1.4185	0.4512
100	1.3866	0.3
10-50	3.34	1.96
50-100	1.58	0.63

Table 1: Results for a binary alphabet.

#### Case B

The DAWG does not stop before the  $C \log_s m^{th}$  symbol. The probability of this case is less than  $1/m^k$  by the proposition 3. In the worst case the DAWG achieves the critical position *critpos*. Then the expected number of text characters inspections in this case is not greater than  $m^k$  for the DAWG and  $m^k$  for the AC machine to achieve the position *pos* and then eventually  $C \log_s m$  for the AC machine to achieve a prefix shorter than  $m/2$  (by lemma 4).

Thus we can bound the expected number of text characters inspections by the formula  $(1-1/m^k)2C \log_s m + (1/m^k)(2m^k + C \log_s m) = O(\log_s m)$ . This completes the proof.

Proposition 2 and Lemma 5 together imply directly the following result.

**Theorem 6** *Under our assumptions on  $P$  the algorithm DAWG-MATCH makes on average  $O(n \log_s m/m)$  inspections of text characters.*

## 4 Experimental results

In order to verify the good practical behavior of the DAWG-MATCH algorithm we have tested it against the Commentz-Walter algorithm. The two algorithms were implemented in C. We implemented the simple Commentz-Walter algorithm which is quadratic in the worst case (see [Hu 90]). Tests on these two algorithms have been performed with three kinds of alphabet: binary alphabet, alphabet of size 4, and alphabet of size 8.

For each alphabet size, we randomly build a text of 50000 characters. Then, we first made experiments with sets of patterns of the same length: for each length of pattern we randomly build a set of 100 patterns of the same length.

After that we build sets of patterns of different length (the length is random in a interval): one set with 100 patterns of lengths between 10 and 50, and one set of 100 patterns of lengths between 50 and 100.

Then, for the two algorithms, we count the number of inspections per one text character. The results are presented in Figures 5, 6 and 7 and in Tables 1, 2 and 3.

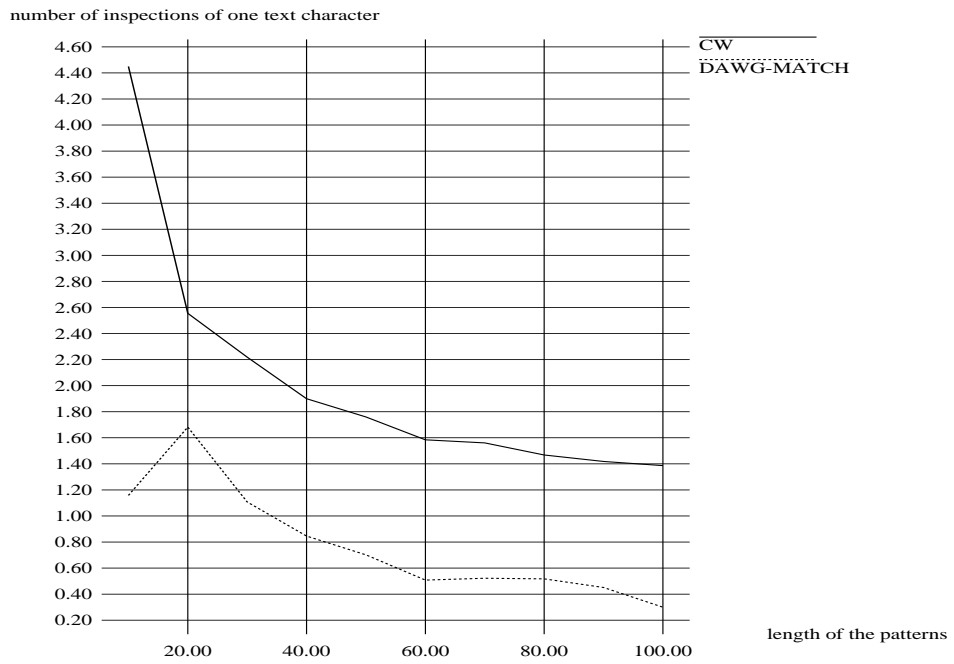


Figure 5: Results for an alphabet of size 2.

Length of patterns	Commentz-Walter	DAWG-MATCH
10	1.9887	1.4938
20	1.5612	0.6884
30	1.3593	0.47
40	1.2843	0.3457
50	1.2094	0.2785
60	1.1371	0.2351
70	1.1205	0.205
80	1.0521	0.3402
90	1.0376	0.2285
100	1.0258	0.1462
10-50	1.83	1.34
50-100	1.2	0.27

Table 2: Results for an alphabet of size 4.

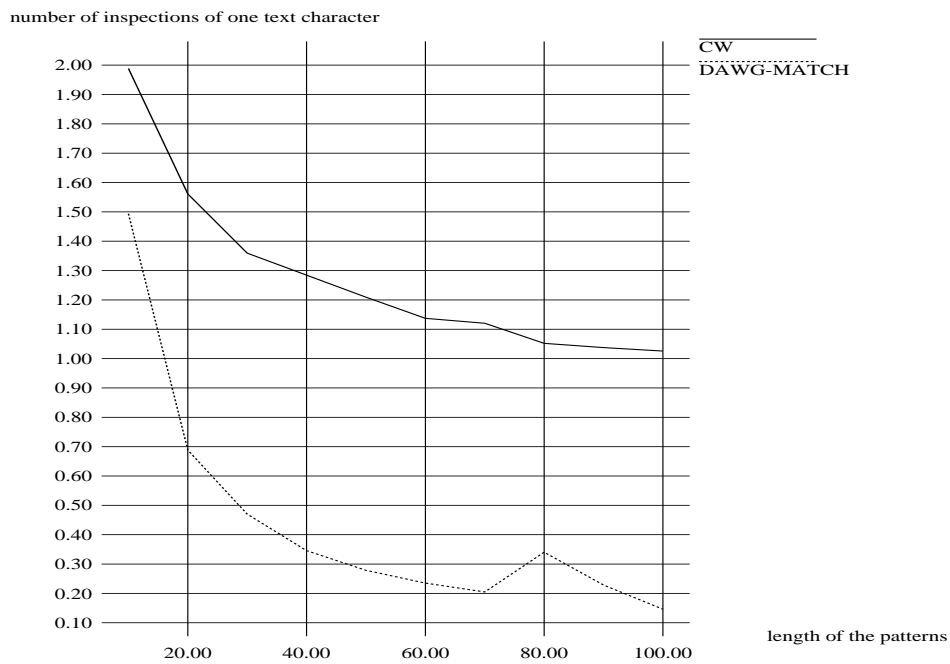


Figure 6: Results for an alphabet of size 4.

Length of patterns	Commentz-Walter	DAWG-MATCH
10	1.5611	0.8749
20	1.255	0.4313
30	1.2007	0.2923
40	1.1144	0.223
50	1.0541	0.181
60	1.0335	0.1828
70	1.0138	0.1964
80	0.946	0.2053
90	0.9296	0.1065
100	0.8906	0.0968
10-50	1.5	0.87
50-100	1.04	0.18

Table 3: Results for an alphabet of size 8.

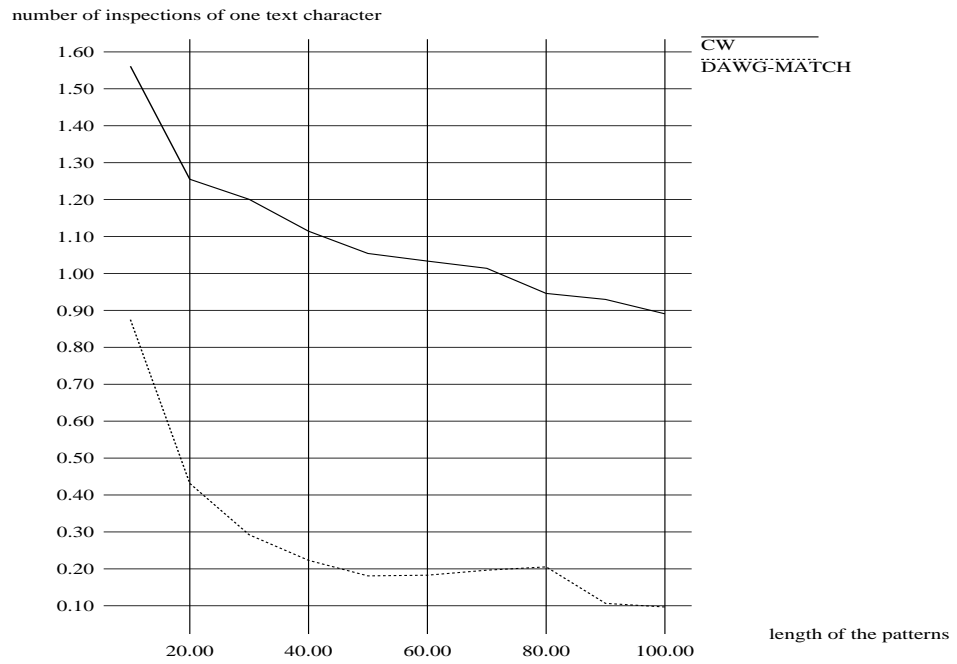


Figure 7: Results for an alphabet of size 8.

On the binary alphabet, the results for the DAWG-MATCH algorithm is better for length 10 than for length 20 because of the added scanning part with the AC machine until the shift is big enough which saves a lot of inspections.

From these results it appears that for small alphabet the DAWG-MATCH algorithm is much better than the simple Commentz-Walter algorithm. This is due to the fact that the Commentz-Walter algorithm computes its shifts with the suffixes it recognizes in the text, but when the set of patterns is big the probability that those suffixes reappear close to the right end of at least one pattern is very large; so, the shifts computed by the Commentz-Walter are small.

**Remark** (on single-pattern matching)

The previous algorithm is especially simple and efficient for one pattern. In this case we can replace the Aho-Corasick algorithm by the Knuth-Morris-Pratt algorithm. The preprocessing phase consists in building the failure function of Knuth-Morris-Pratt and the DAWG for the reversed pattern. Then during the search phase the shift are computed as the difference between the length of the pattern and the position of the character of the pattern which is compared when the algorithm stops scanning from left to right.

## References

- [Ah 90] A. V. AHO, Algorithms for finding patterns in strings, In: (J. VAN LEEUWEN, editor, *Handbook of Theoretical Computer Science*, vol A, *Algorithms and complexity*, Elsevier, Amsterdam, 1990) 255–300.
- [AC 75] A. V. AHO, M. CORASICK, Efficient string matching: an aid to bibliographic search, *Comm. ACM* **18** (1975) 333–340.
- [BR 90] R. A. BAEZA-YATES, M. RÉGNIER, Fast algorithms for two-dimensional and multiple pattern matching, In: (R. KARLSSON, J. GILBERT, editors, *Proceedings of the 2nd Scandinavian Workshop in Algorithmic Theory*, Lecture Notes in Computer Science 447, Springer-Verlag, Berlin, 1990) 332–347.
- [B-M 87] A. BLUMER, J. BLUMER, A. EHRENFUCHT, D. HAUSSLER, R. MCCONNELL, Completed inverted files for efficient text retrieval and analysis, *J. ACM* **34** (1987) 578–595.
- [BM 77] R. S. BOYER, J. S. MOORE, A fast string searching algorithm, *Comm. ACM* **20** (1977) 762–772.
- [C-R 94] M. CROCHEMORE, A. CZUMAJ, L. GASIENIEC, S. JAROMINEK, T. LECROQ, W. PLANDOWSKI, W. RYTTER, Speeding up two string matching algorithms, *Algorithmica* **12**(4/5) (1994) 247–267.
- [Co 79a] B. COMMENTZ-WALTER, A string matching algorithm fast on the average, Technical Report 79.09.007, I.B.M. Heidelberg Scientific Center, Germany, 1979.
- [Co 79b] B. COMMENTZ-WALTER, A string matching algorithm fast on the average, In: (*Proceedings of the 6th International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1979) 118–132.
- [Hu 90] A. HUME, Keyword matching, Internal report of Bell Laboratories, NJ, 1990.
- [KMP 77] D. E. KNUTH, J. H. MORRIS JR, V. R. PRATT, Fast pattern matching in strings, *SIAM J. Comput.* **6** (1977) 323–350.

- [Le 92] T. LECROQ, A variation on Boyer-Moore algorithm, *Theoret. Comput. Sci.* **92**(1) (1992) 119–144.
- [Ur 88] N. URATANI, A fast algorithm for matching patterns, Internal report of IECEJ, Japan, 1988.
- [Ya 79] A.C. YAO, The complexity of pattern matching for a random string, *SIAM J. Comput.* **8** (1979) 368–387.