ANALYZE THE EFFICIENCY OF KEYWORD SEARCH ALGORITHMS

DANIELA ENACHESCU, ANDREEA RADULESCU
UNIVERSITATEA PETROL-GAZE, PLOIESTI, UNIVERSITÉ LIBRE DE BRUXELLES, BRUXELLES denachescu22@yahoo.com

Abstract:
This paper intends to analyze, through developed software, the answer of various algorithms to keyword search in random strings problems within the constraint of time and memory space used.
The program developed is able to choose, depending on the algorithm used, the maximum number of characters used to comply with data constraints. Afterwards, the performance of algorithms in operations of keyword search in random strings is analyzed.
Tests were performed by generating random texts of different lengths, using different character sets.
Results can be used to minimize search time, such as in the FIND function used by text editors.

Key words: *String matching, keyword search, text editors (word processing)*

JEL classification: C6

INTRODUCTION

The purpose of this study was to develop a software in order to analyze the answer of three different string matching algorithms. In order to test each algorithm, the software is capable to generate random strings of different lengths and using alphabets with a specific size. This method has a complexity of O(n), n being the number of the characters of the string you want to generate. This process depending only on the length of the strings, will not influence differently the time response of the three algorithms in a test with the same length of the chains.

This software could be used to analyze any keyword matching algorithm and in order to make it more efficient for the user we decided to automate all treatments. The program is able to choose, depending on the algorithm, the maximum number of characters to use so that the time constraints (maximum 2 minutes) and space constraints (max. 8MB) will be respected. Afterwards, it computes the different size of strings that can be tested which will respect the constraints.

The three algorithm are using two different chains named S and T. The lengths of these strings, n_s and n_t, are pre-calculated before each algorithm.

In the begging of the paper we will present the 3 different algorithms used in the software, their complexity and their structure. The results of the different tests will be presented in the fourth part of the paper along with the complexity deduced for the test curves.

1st algorithm

The first algorithm presented is inspired from the Boyer-Moore algorithm and it consists in finding the longest suffix between each possible sub-strings of S and T.

This algorithm starts by filling a table with the numbers of the common consecutive characters for each pair (n_s,n_t). Then, it searches in the same table the maximum value and returns it.

In order to find the number of common consecutive characters, the algorithm will compute the longest common suffix between the substring from the 1st character to the ith character of the chain S and the substring from 1 to jth of the string T (i can take values between 1 and the length of S and j from 1 to the length of T). Thus, if i=1 then the algorithm will make a single comparison for each j from 1 to n_t. If i=2, only one comparison will be make for j=1 and 2 comparison for j from 2 to n_t. More generally, for a value of i=k, the algorithm will make, for each value of j from 1 to n_t, min(k,j) comparison. Therefore, we can deduce that the complexity of the algorithm is $O(n_s*n_t)$.

The algorithm contains a method that fills the matrix with the number of characters, LSuff. In the worst case, this method compares n characters, n being the minimum value between the lengths of the two chains. The maximum number of comparisons occurs when one of the chains is the suffix of another string.

```
Method LSuff(S: string, T: string, $n_s$: integer, $n_t$: integer)
      if S[i] = T[j] then
            nb := 1
      else
            nb := 0
      endif
      if nb ≠ 0 and i > 0 and j > 0 then
            nb := nb + LSuff(S, T, i - 1, j - 1)
      endif
      return nb
```

```
Method algoA1(S: string, T: string, $n_s$: integer, $n_t$: integer)
      for i from 0 to $n_s$-1 do
            for j from 0 to $n_t$-1 do
                  MLSuff[i,j] := LSuff(S, T, i, j)
            endfor
      endfor
      max := 0
      for i from 0 to $n_s$-1 do
            for j from 0 to $n_t$-1 do
                  if MLSuff[i,j] > max then
                        max := MLSuff[i,j]
                  endif
            endfor
      endfor
      return max
```

2nd algorithm

The second algorithm represents an improvement of the naïve algorithm.

The main idea for this algorithm is to compare all sub-chains of the first string to all the sub-chains of the second. The algorithm will compare, for each sub-word with a number of characters from 1 to n_s, belonging to S and to T, each character of a sub-chain of S to each character of a sub-chain of T. Thus, for a length of the sub-chains equals to 1, the algorithm will make one comparison for n_s*n_t sub-chains. For a length of 2, the algorithm will make two comparisons for $(n_s-1)*(n_t-1)$ sub-chains. More generally, for a sub-word of length of l, the algorithm will make l comparisons, for $(n_s-l+1)*(n_t-l+1)$ sub-chains, with l taking values between 1 and n_s. The overall complexity is $O(n_s*n_s*n_t)$.

```
Method algoA2(S: string, T: string, $n_s$: integer, $n_t$: integer)
      max := 0
      for l from 1 to $n_s$ do
            for i from 0 to $n_s$-l do
                  for j from 0 to $n_t$-l do
                        found := true
                        for k from 0 to l-1 do
                              if S[i + k] ≠ T[j + k] then
```

```
                                        found := false
                                  endif
                            endfor
                            if found = true then
                                  max := 1
                            endif
                      endfor
                endfor
          endfor
```

3rd algorithm

The third algorithm is inspired for the Knuth-Morris-Pratt algorithm and the length of the fist string, S, has to be smaller that the length of the string T.

This algorithm allows finding the length of the longest substring of S which is common to T. For each sub-word of S, the algorithm has a phase of pre-treatment. In this part, the table PMK_{ij} allows to find faster if there is a correspondence between a sub-string from S in T. At the end, the algorithm returns the maximum between all the sub-strings of S that appears in T.

```
Method algoA3(S: string, T: string, $n_s$: integer, $n_t$: integer)
      for i from 0 to $n_s$-1 do
            for j from i to $n_t$ do
                  $M_{ij}$ := S[i..j]
                  a := 0
                  b := -1
                  $PMK_{ij}$ [0] := -1
                  while a < $m_{ij}$ do
                        while b > -1 and $M_{ij}$[a] ≠ $M_{ij}$[b] do
                              b := $PMK_{ij}$[b]
                        endwhile
                        a := a + 1
                        b := b + 1
                        if a < $m_{ij}$ and $M_{ij}$[b] then
                              $PMK_{ij}$[a] := $PMK_{ij}$[b]
                        else
                              $PMK_{ij}$[a] := b
                        endif
                  endwhile
                  bool_res := false
                  a := 0
                  for b from 1 to $n_t$ do
                        while a > -1 and $M_{ij}$[a] ≠T[b] do
                              a := $PMK_{ij}$
                        endwhile
                        a := a + 1
                        if a ≥ $m_{ij}$ then
                              bool_res := true
                              a := $PMK_{ij}$[a]
                        endif
                  endfor
            endfor
      endfor
```

Results and Conclusion:

In order to test the results of the algorithm in function of the length of the data, we tested each algorithm with strings randomly generated and using different numbers of letters of the alphabet. If in the case of a 26 letters alphabet, the length of the longest sub-string common to each pair of S and T can be small, we have also performed tests with a smaller alphabet. The complexity of the algorithm for generating strings is the same, no matter what number of letters contains the alphabet used for S and T. Knowing that we can analyze in the same way the influence of the types of input chains in the algorithms.

Because the number of the letters in the alphabet does not influence much the behavior of the algorithms, we are going to present only the tests with alphabets of 26 and 10 different letters. As we can see in the graphics, the difference in response times between the two types of test series was low. For tests using randomly generated strings with only 10 different letters of the alphabet, the algorithm response time is only slightly larger than the tests using 26 letters.

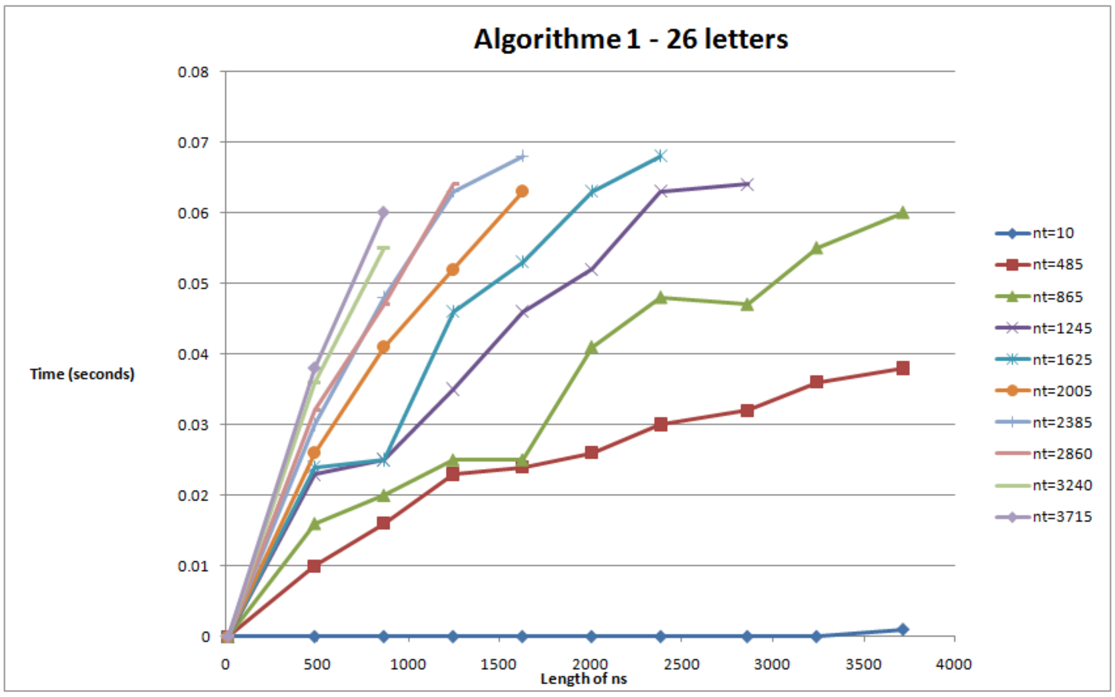

Fig. no.1

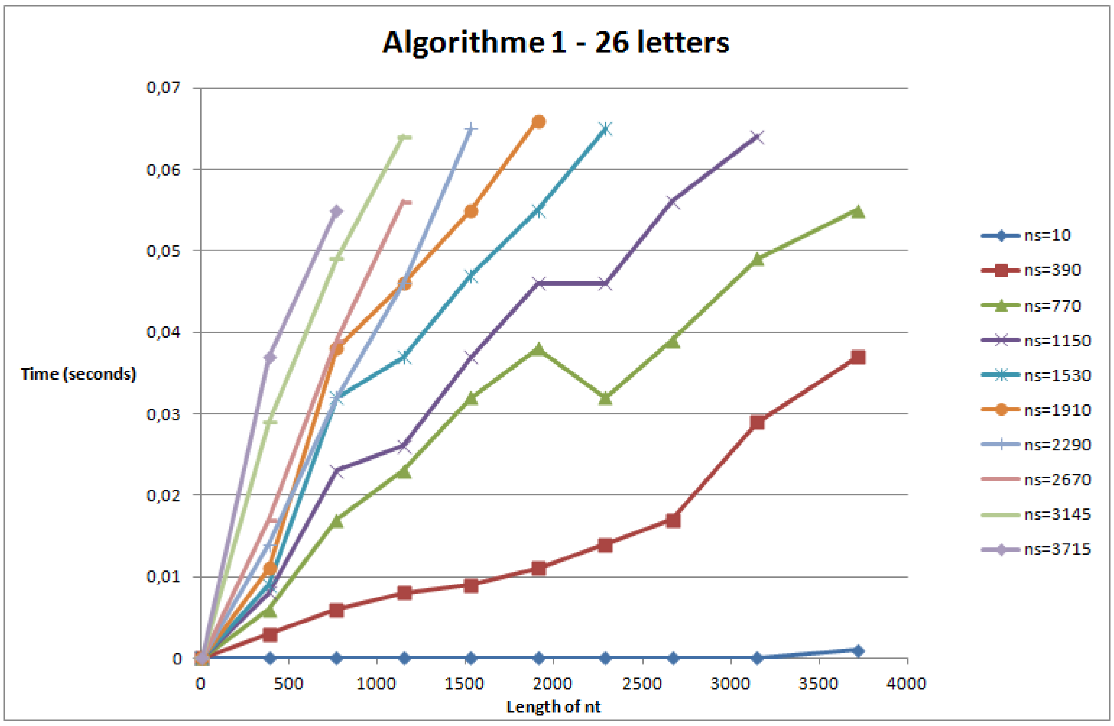

Fig. no.2

In the above curves, we can see that the first algorithm runs linearly both in relation to the length of the string S or T. This implies that the complexity of this algorithm is polynomial. By analyzing the different curves of the execution time we could found the equation of the function associated with the complexity of this algorithm:

$$4 * 10^{-5}x – 1,4 * 10^{-3}$$

We can deduce than, that the execution time of the first algorithm is $O(n^2)$.

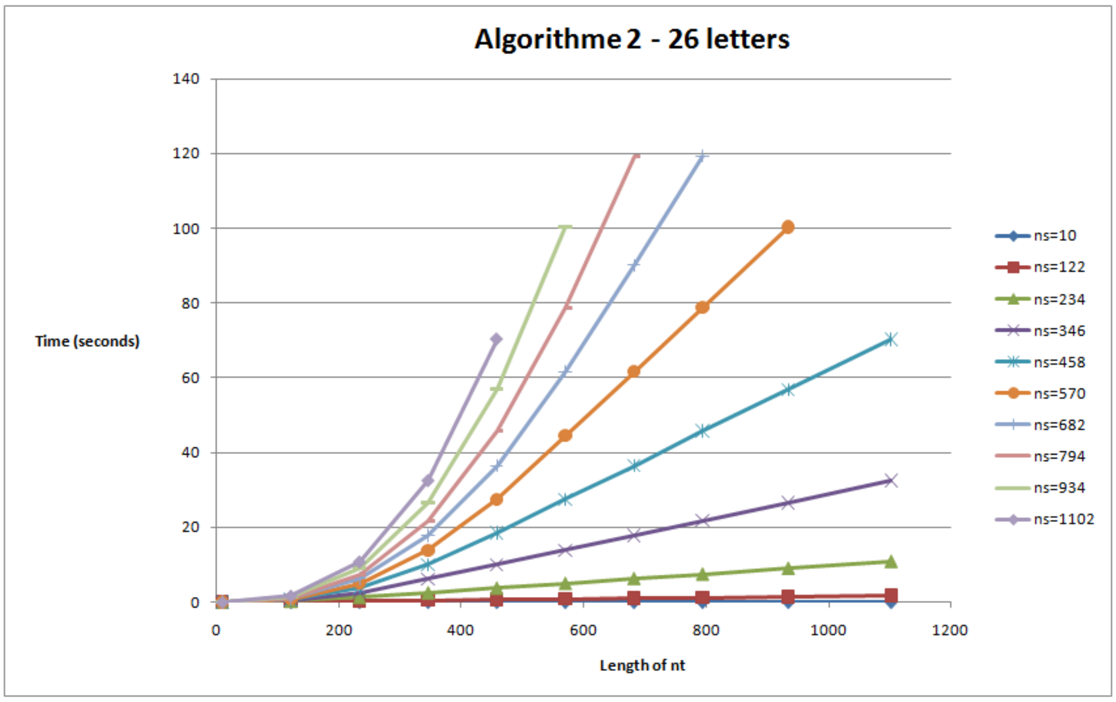

Fig. no. 3

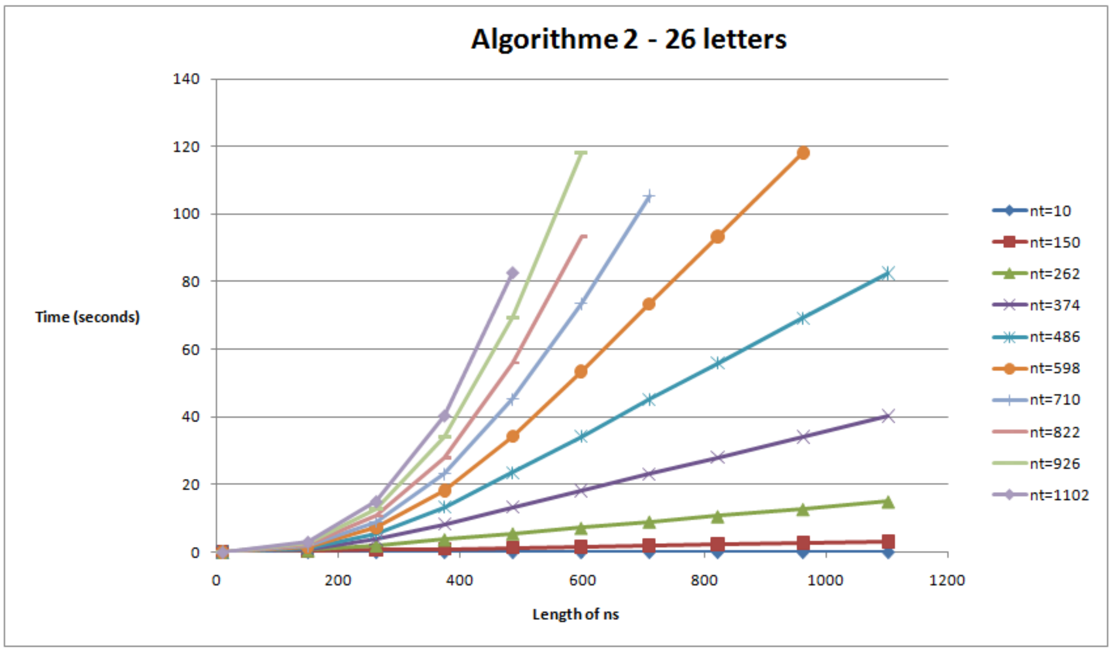

Fig.no.4

For the second algorithm we can see in the graphics that the execution time increases faster than for the first algorithm, regarding to the length of S and also to the length of T. This implies that the complexity of this algorithm is polynomial. The function associated to the complexity of the algorithm is:

$$5*10^{-5}x^2 + 7,3 * 10^{-2}x + 1,57$$

For the third algorithm we can see a small improvement in the run time.

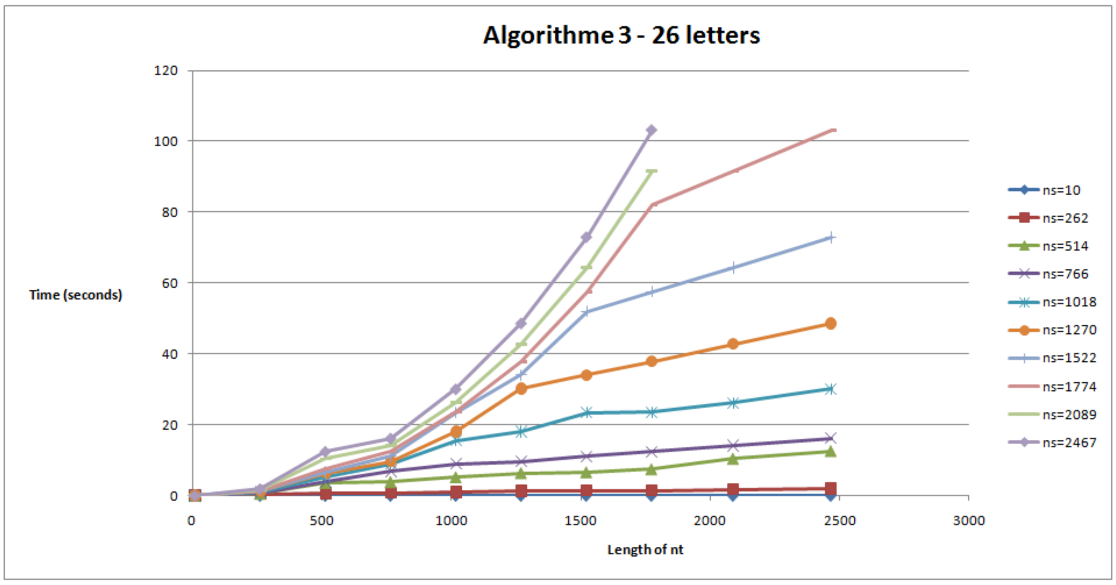

Fig. no.5

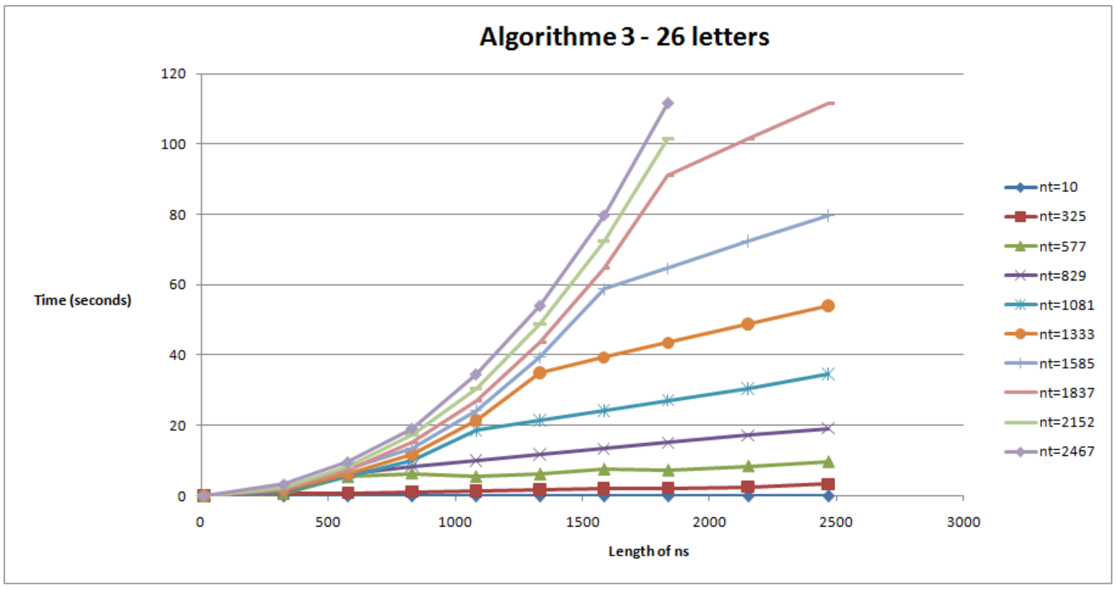

Fig.no.6

In order to see the impact of the amount of letters of the alphabet used in the string generation, we have tested each algorithm with different size alphabets.

As we can see in the graphics of the first algorithm with a 10 letters alphabet, the behavior of the curves resembles a lot with the tests using an alphabet using 26 letters. After numerous tests, we could conclude that the size of the alphabet is not an important factor in the execution time for any of the three algorithms.

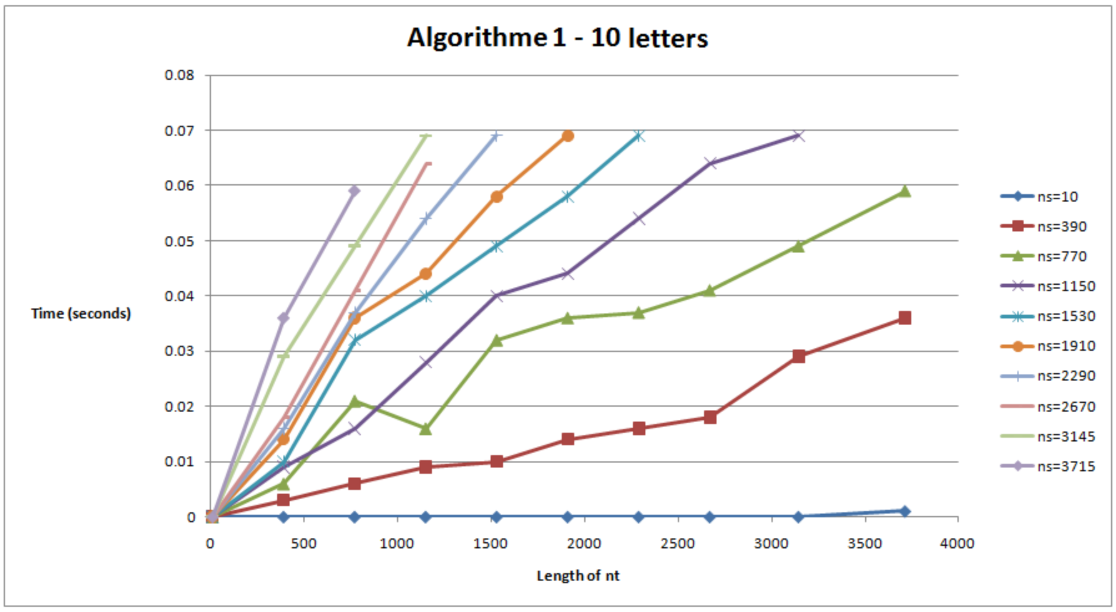

Fig.no.7

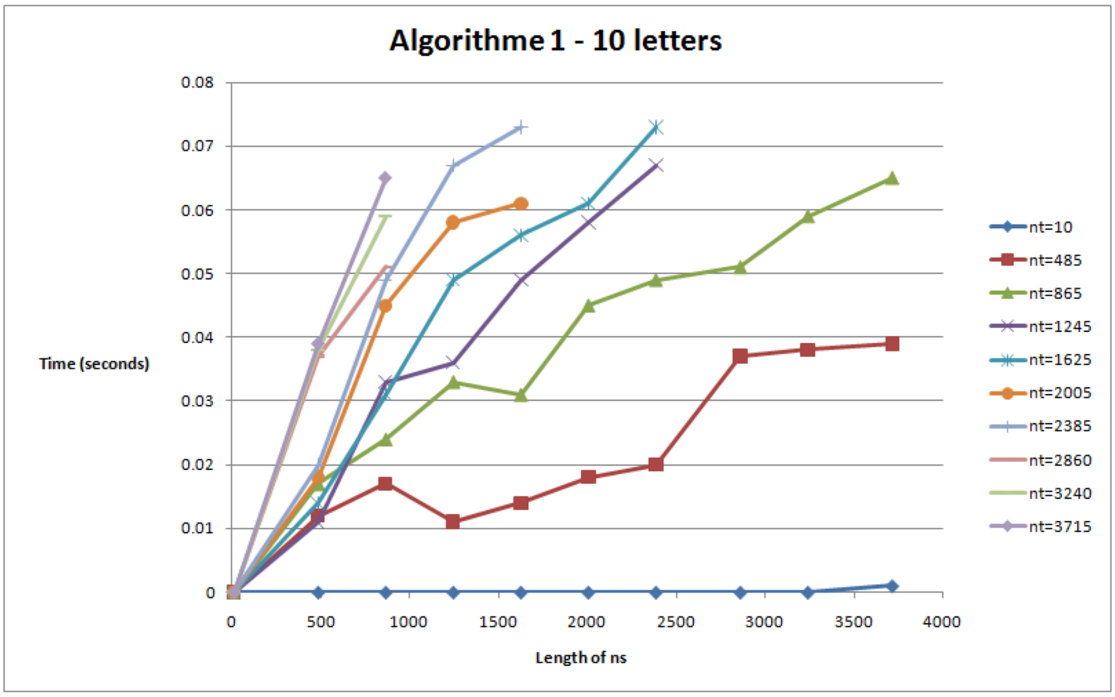

Fig. no.8

The goal of this work is to analyze the efficiency of algorithms in search operations using strings of different lengths in time constraints (maximum 2 minutes) and space constraints (max. 8MB) and the possibility of using these results in operations such as Find in text editors, browsers, etc..

BIBLIOGRAPHY

Aho A.V., Corasick M.J., *Efficient string matching: An aid to bibliographic search*, Communications of the ACM, vol. 18, issue 6, pages 333–340, juin 1975;

Aho A.V, *Algorithms for finding patterns in strings*. in *Handbook of Theoretical Computer Science, Volume A, Algorithms and complexity*, J. van Leeuwen ed., Chapter 5, pp 255-300, Elsevier, Amsterdam, 1990;

Baase, S., Van Gelder, A., *Computer Algorithms: Introduction to Design and Analysis*, 3rd Edition, Chapter 11, Addison-Wesley Publishing Company, 1999;

Cormen, T. H.; Leiserson, C. E., Rivest, R. L., Stein, Clifford *Introduction to Algorithms* (3rd ed.). MIT Press and McGraw-Hill,1990 (2009);

Crochemore, M., *Off-line serial exact string searching*, in *Pattern Matching Algorithms*, ed. A. Apostolico and Z. Galil, Chapter 1, pp 1-53, Oxford University Press, 1997;

Fertin G, Rusu I., *Algorithms in Computational Molecular Biology: Techniques, Approaches and Applications*, in *Computing Genomic Distances: An Algorihtmic Viewpoint*, Wiley Science, 2011;

Gonnet G.H., Baeza-Yates R.A., *Handbook of Algorithms and Data Structures in Pascal and C*, 2nd Edition, Chapter 7, Addison-Wesley Publishing Company, 1991;

Knuth D, James H. Morris, Jr. et Vaughan Pratt. *Fast pattern matching in strings*. *SIAM Journal on Computing*, 6(2):323–350. 1977;

Morris (Jr) J.H.**,** Pratt V.R.**,** *A linear pattern-matching algorithm*, Technical Report 40, University of California, Berkeley, 1970;

Sedgewick, R., *Algorithms in C*, Chapter 19, Addison-Wesley Publishing Company, 1988;

Stephen, G.A., , *String Searching Algorithms*, World Scientific, 1994;

Wirth, N., *Algorithms & Data Structures*, Chapter 1, pp. 17-72, Prentice-Hall, 1986.