S. Krajči (ed.): ITAT 2018 Proceedings, pp. 181–187
CEUR Workshop Proceedings Vol. 2203, ISSN 1613-0073, © 2018 Tatiana Jajcayová and Zuzana Majeríková

A note on some interesting test results for the Rabin-Karp string matching algorithm

Tatiana Jajcayová, Zuzana Majeríková

Department of Applied Informatics, Comenius University, Bratislava, Slovakia
jajcayova@fmph.uniba.sk
z.majerikova19@gmail.com

Abstract: While analyzing and testing some of the well known string matching algorithms, we came across rather interesting phenomenon in tests results for the Rabin-Karp algorithm. Testing which moduli work best in the algorithm, we realized that on some texts some moduli perform much worse then others. This caught our attention, and we continued testing. In this paper, we summarize tests using different moduli on the same text, tests run on different English texts (fiction, scientific, law, newspaper), and tests run on different languages and alphabets (English, Slovak, French, German and Russian). We include analysis of the obtained results and state some hypotheses.

1 The Rabin-Karp algorithm

We start with a brief review of the Rabin-Karp algorithm and its computational complexity. This string matching algorithm is based on the use of elementary number-theoretic notions such as the equivalence of two numbers modulo a third number. Rabin and Karp [5] proposed the algorithm to improve performance of string matching algorithms in practice. This algorithm generalizes to other algorithms for related problems, such as two dimensional pattern matching.

To simplify the explanations, it will be convenient to assume for a moment that characters are decimal digits, so we interpret the characters of strings as both digits and also graphical symbols. Thus, a string of length k can be represented as a decimal number. For instatnce, a string $s =$ "12345" of length 5 corresponds to a decimal number 12 345. In general case, we assume to have an alphabet Σ which consists of characters that are digits from the number system over the base $b = |\Sigma|$.

Let us have a pattern $P[1...m]$ and text $T[1...n]$. The decimal value of pattern P will be denoted as p. Similarly, the decimal value of every substring of a text T of length m starting at a position s will be denoted as t_s. We say that pattern P occurs in text T if and only if one can find at least one t_s that has the same decimal value as p, i.e. $t_s = p$ and $T[s+1...s+m] = P[1...m]$, $s = 0, 1, \ldots, n-m$. In that case we call s a valid shift.

We can compute decimal value of p in time $\Theta(m)$ by using the Horner's Rule:

$$p = P[m] + 10(P[m-1] + 10(P[m-2] + 10(P[m-3] + \ldots$$
$$+10(P[2] + 10P[1])...)$$

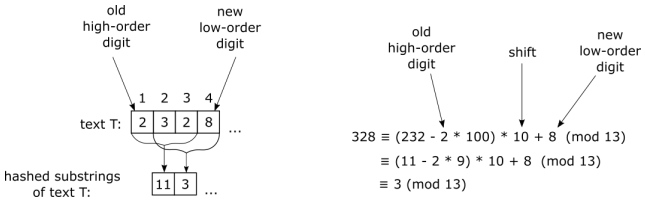

Figure 1: Computation of the value for a window based on a previous window in a constant time. The first window has a value "232". We drop the high-order digit that is "2" and multiply the rest of the window by 10. Then we add the low-order digit that is "8" and we get new value that is "328". The value of the first window is 11 and the value of the new window is 3, because all computations are performed with out hash function that is modulo 13.

Similarly, again by the Horner's Rule, the value t_0 can be computed in $\Theta(m)$.

For every s, the value t_{s+1} can be then computed from the previous value t_s:

$$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1].$$

For example, to compute the value t_1 from t_0 for our text T = "12345" from the previous paragraph and pattern P of length $m = 3$, we use the value t_0 which is 123. In this computation $s = 0$. A character at the position T[s+1] is "1", and its numerical value is 1, a character at position T[s+m+1] is "4" and its value is 4. The computation is as follows:

$$t_1 = 10(t_0 - 10^{3-1}T[0+1]) + T[0+3+1]$$
$$t_1 = 10(123 - 10^2 T[1]) + T[4]$$
$$t_1 = 10(123 - 100) + 4$$
$$t_1 = 234$$

We see that by computing $t_s - 10^{m-1}T[s+1]$, we remove the highest order digit from value t_s. By subsequent multiplication of the remainder of the value by 10, we shift the number to the left. 10 is used because we work with the decimal system. In the general case, where base b is $|\Sigma|$, we would have to shift by b. By adding $T[s+m+1]$ we add the low-order digit and we get the final value of t_{s+1}. (Figure 1)

If the constant 10^{m-1} is precomputed, then for each $s = 1, 2, \ldots, n-m$, the computation of the value t_s is done in constant time. Thus the computation of all the remaining values t_1, t_2, t_3, …, t_{n-m} is done in $\Theta(n-m)$. That

means, that by comparing value of p with every value of t_s, we can get all valid shifts in time $\Theta(m)+\Theta(n-m+1)$.

A problem may occur, when value p and values t_s are too large. Some mathematical operations with large numbers may take a long time, and the assumption that they take only constant time may be unreasonable. One possible remedy for this is, what Rabin-Karp algorithm does: instead of comparing the original values, it computes values p and t_s modulo some suitable modulus q, and then compare these new modulated numbers. The modulated numbers are not bigger than $q-1$. Computation of $p \pmod q$ takes again time $\Theta(m)$ and computation of all $t_s \pmod q$ values together takes $\Theta(n-m+1)$.

The modulus q is usually chosen to be a prime such that the value $b \cdot q$ fits within a computer word. (b is the base, the number of characters in the alphabet Σ). According to this, the previous equation for computing t_s+1 changes to:

$$t_{s+1} \equiv (b(t_s - T[s+1]h) + T[S+m+1]) \pmod q,$$

where

$$h \equiv d^{m-1} \pmod q$$

It is clear that if $p \not\equiv t_s \pmod q$ then P does not start at position s in T. So if this happens, we know that s is not a valid shift. On the other hand, it is possible that $p \equiv t_s \pmod q$, and still the pattern P does not start at s. That means that to complete this algorithm, we must not only compare the modulated values and find a match, but also test further to see if the substring $T[s+1...s+m]$ is the same as the pattern P. This additional but indispensable comparison takes another $\Theta(m)$ time. The simple graphical example of the Rabin-Karp algorithm and modulation of the values of t_s is shown in Figure 2.

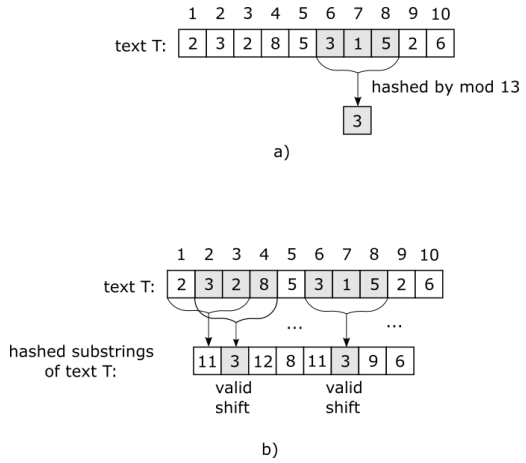

Figure 2: Example of the Rabin-Karp string matching with alphabet Σ in which every character is a decimal digit. a) an example of a hashed pattern $P = 315$ that is located in text T. Part b) demonstrates hashing of every possible substring of T with the length of P. Two substrings that have the same hash value as the pattern are found and they are denoted as valid shifts. Only the substrings with the valid shift are compared further with the pattern by characters. For this case we have found one occurrence of pattern P in text T at position 6.

In our version of the Rabin-Karp algorithm the preprocessing time takes $\Theta(n)$. The processing time takes $\Theta((n-m+1)m)$ in the worst case, and only $\Theta(n+1)$ in the best case of running. The worst case occurs when all characters of pattern P and all characters of text T are the same unique symbol. For example, if we have a pattern P="aaa" and a text T="aaaaaaaaa", then algorithm considers every possible shift as valid and behaves as the naive string matching algorithm.

2 Implementation

In this section we sketch how we implemented the Rabin-Karp algorithm. The particulars of the implementation can become interesting when discussing the test results in the last section of the paper.

We have chosen Java as the programming language. Java is an object-oriented language, and its biggest advantage is its platform independence, which means that the Java code can be run without modifying at any platform that supports Java. As a development tool we chose Eclipse Java Oxygen with a WindowBuilder component. WindowBuilder is designed to create Java GUI applications in an easy way.

2.1 Implementation of the Main Application

We study and test Rabin-Karp algorithm in a context of other string matching algorithms. To organize the testing of the implemented algorithms, we needed to create the main application with Graphical User Interface. This application is build to work with all the studied algorithms and to manage the input texts, patterns and all of the output results. The main components that we need in our application:

- **Input pattern** - text input field, where the user can write or paste the pattern that he wants to find in the text **Input text**
- **Input text** - text input field, where the user can write or paste the text where he wants to find all the occurrences of the pattern from the **Input pattern**
- **Load text from file** - button, that allows the user to load text from a file to the **Input text**
- **Algorithm** - selector, where the user can choose which algorithm he wants to test
- **Modulo** - number input field, where the user can choose the modulo that will be used in the Rabin-Karp algorithm
- **Output text** - output text field, where the text is rewritten from the **Input text**, but with highlighted occurrences of the pattern from **Input pattern**

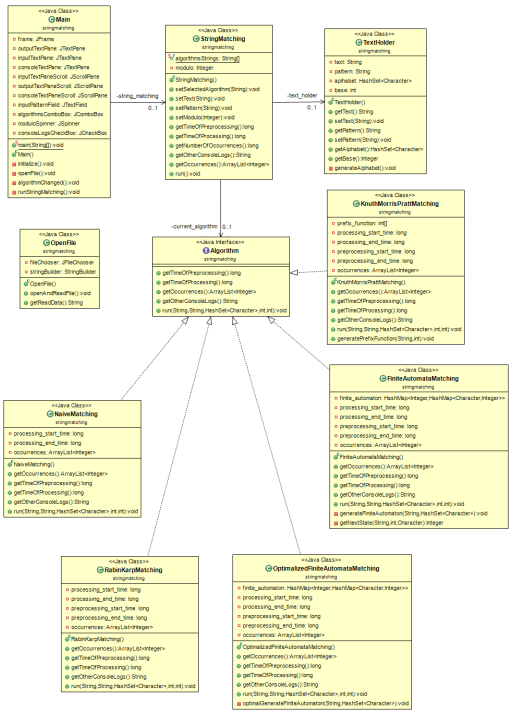

Figure 3: Class diagram of the main application generated by The ObjectAid UML Explorer for Eclipse

- **Additional console logs** - check field, if it is checked it means that in the **Console** there will be written some additional logs for the given algorithm alongside with the proccessing and preprocessing time
- **Console** - after running an algorithm, in the console there is shown it's processing and preprocessing time and when the **Additional console logs** is checked there are some additional logs for every algorithm
- **Run** - button that runs the program, where with a selected **Algorithm**, search in **Input text** for an **Input pattern**

A class diagram is shown in Figure 3.

The application consists of the following classes that work as follows:

Main is the main class that runs the application. Its purpose is to initialize the contents of the frame of GUI window. When users run the application, the Main class gets the value of the input text, the input pattern, the input modulo and the selected algorithm. The class passes all this values to class **StringMatching** and calls the function *run* of the class **StringMatching**. After the *run* is completed, the class fills the output text and console with the results of running.

TextHolder is the class that stores the text and the pattern that the application got on input. It also generates the alphabet and the base based on the given text and pattern.

OpenFile is the class that allows the user to load a text from a file with a *JFileChooser*.

Algorithm is an interface for implemented algorithms. Every algorithm needs to have implemented functions *getTimeOfPreprocessing*, *getTimeOfProcessing*, *getOccurrences*, *getOtherConsoleLogs* and *run(text, pattern, alphabet, base, modulo)*.

NaiveMatching, RabinKarpMatching, FiniteAutomataMatching, OptimalizedFiniteAutomataMatching, KnuthMorrisPrattMatching are the classes where the implemented algorithms that we used are.

StringMatching is the class that gets the selected algorithm, input text, input pattern and modulo. First, it processes the text and the pattern with **TextHolder** and gets the alphabet and the base generated from the text. Then, it creates a new instance of the selected algorithm and calls it's function *run*.

The review of the implemented application and it's GUI is shown in Figure 4, where a simple running of the program with the naive string matching algorithm with pattern "the" and text of the book "The Project Gutenberg EBook of The Adventures of Sherlock Holmes by Sir Arthur Conan Doyle" is demonstrated.

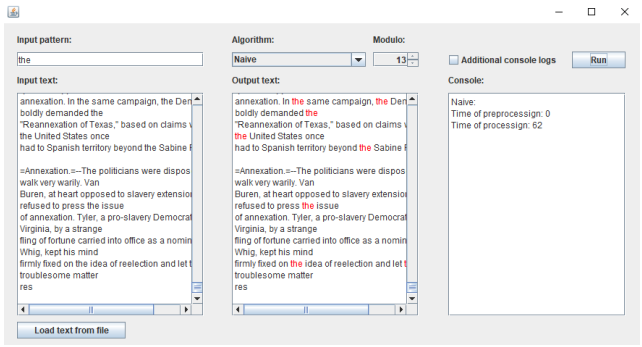

Figure 4: The main application, where the user can test the implemented algorithms.

2.2 The Rabin-Karp algorithm implementation

For the Rabin-Karp string matching procedure, after it receives the input text, the pattern, the value of d that represents the $radix - d$ notation that is uses in its matching and modulo, we need to initialize the value of the high-order digit position of an m-digit window. This is shown in Listing 1 on line 1. After that the preprocessing of this algorithm follows. We need to compute the value of p and the value of the first substring t_0 of the text with length of the pattern. Implementation of this is shown in Listing 1 on lines $3-6$, where the *pattern_hash* is our value of p and *sub_text_hash* represents the value of t_0. As a default, these variables were preset to value 0.

Listing 1: Calculation of Hashed Values of Value h, p and t_0

```
1  int h = (int) Math.pow(base, pattern_length-1) % modulo;
2
3  for (int i=0; i<pattern_length; i++) {
4      pattern_hash = (base*pattern_hash + pattern.charAt(i)) % modulo;
5      sub_text_hash = (base*sub_text_hash + text.charAt(i)) % modulo;
6  }
```

How the processing is implemented can seen in Listing 2. It starts by iterating through all possible shifts s in the text. On line 3 there is a check whether value p matches value of t_s at shift s where the loop is executed. If so, then the character by character comparison between pattern P and substring of text $T[s+1...s+m]$ follows. If these two strings are equal, we add shift to the array of all of the occurrences, where we store all shifts in which we have found a positive match. The check on line 9 will be true on every t_s, except the last one t_{n-m}. That is because the main loop will execute one more time and in that case we need to compute the value of upcoming t_{s+1}. This computation is implemented on line 10. In the case that the value of t_{s+1} is smaller that 0, we add q to this value on lines $13-15$.

Listing 2: Processing of the Rabin-Karp String Matching Algorithm

```
1  for (int s=0; s<(text_length - pattern_length); s++) {
2
3      if (pattern_hash == sub_text_hash) {
4          String text_substring = text.substring(s, (s + pattern_length));
5          if (Objects.equals(pattern, text_substring))
6              occurrences.add(s);
7      }
8
9      if (s < (text_length - pattern_length)) {
10         sub_text_hash = (base*(sub_text_hash - text.charAt(s)*h) +
11                          text.charAt(s+pattern_length)) % modulo;
12
13         if (sub_text_hash < 0)
14             sub_text_hash = (sub_text_hash + modulo);
15     }
16 }
```

3 Testing moduli in the Rabin-Karp algorithm

While testing a performance of the implemented string matching algorithms, we were interested whether a choice of the modulus q in the Rabin-Karp algorithm affects the performance of the algorithm. Recall that for the given base $b = |\Sigma|$, the modulus q is usually chosen to be a prime number such that the value $b \cdot q$ fits within a computer word. In this section we summarize results of our testing of moduli in the Rabin-Karp algorithm.

We had some preconceived ideas how the test results should look like. These expectations were met while testing some of the texts, for an example see Figure 5. The sample text in Figure 5 is an English text. It is an artificial text in the sense that it was generated in an online random text generator, with even some further random changes and additions. The text is 400 000 characters long.

A suprise came when we tested the Rabin-Karp algorithm on *The adventures of Sherlock Holmes* by Sir Arthur Conan Doyle. The text was obtained from The Project Gutenberg. Release Date: March, 1999 [EBook #1661] [Most recently updated: November 29, 2002], Edition: 12, Language: English, Character set encoding: ASCII.

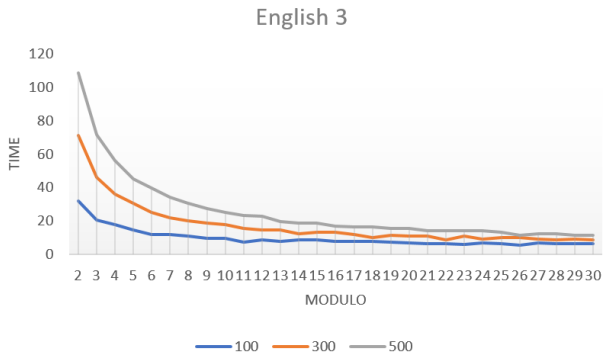

Figure 5: The Rabin-Karp algorithm on an English text generated by an online random text generator. Three different colors represent three different lengths of a searched pattern P.

The text was again about 500 000 characters long, and we tested with different patterns and different lengths of patterns. As we see in Figure 6, modulus $q = 13$ performed significantly worse than the other moduli.

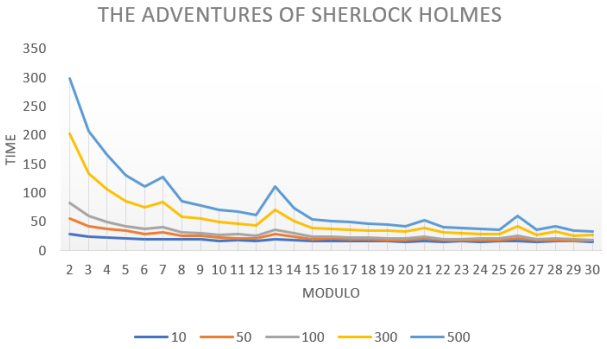

Figure 6: The Rabin-Karp algorithm on English fiction: The adventures of Sherlock Holmes by Sir Arthur Conan Doyle.

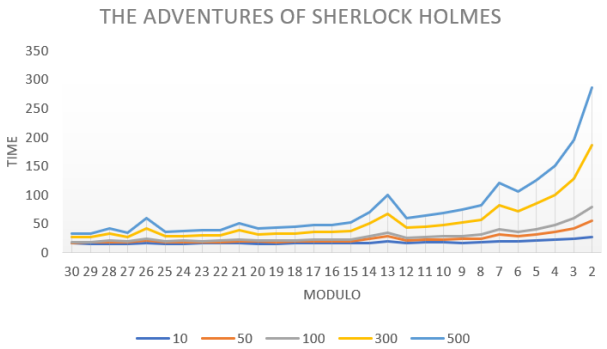

Figure 7: The Rabin-Karp algorithm on English fiction: The adventures of Sherlock Holmes by Sir Arthur Conan Doyle.

This caught our attention. We tested the same text for the second time, now with the moduli ordered in the descending order (see Figure 7) to make sure that a possible error is not caused in the program (language Java can

sometimes slow down the performance of the program because of the Garbage Collector). Since we got the two graphs that are exactly mirror images of each other, we see that the garbage collecting in Java is not the reason for the unexpected behavior.

Our first hypothesis was that the natural language has different characteristics than randomly generated texts. So we continue testing on many different types of English texts and patterns. In Figure 8, we see the results of the testing on a law text The Common Law, by Oliver Wendell Holmes, Jr., and in Figure 9 on a scientific text from the book Ancient And Modern Physics by Thomas E. Willson. (Both these text were again obtained from the Project Gutenberg.) As we see the charts for all these texts are different. To definitely show that our first hypothesis does not hold, in Figure 10 is yet another *randomly generated* English text that has $q = 13$ behaving as it is in the fiction in Figure 6.

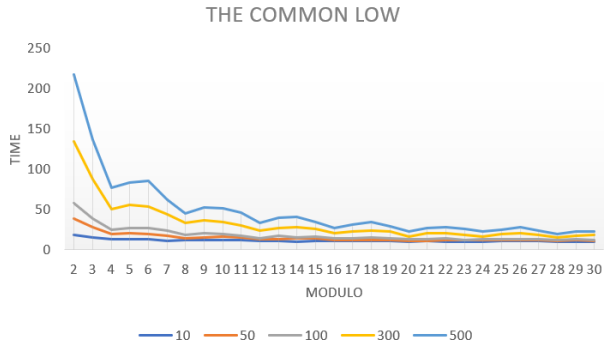

Figure 8: The Rabin-Karp algorithm on an English law text: The Common Law, by Oliver Wendell Holmes, Jr.

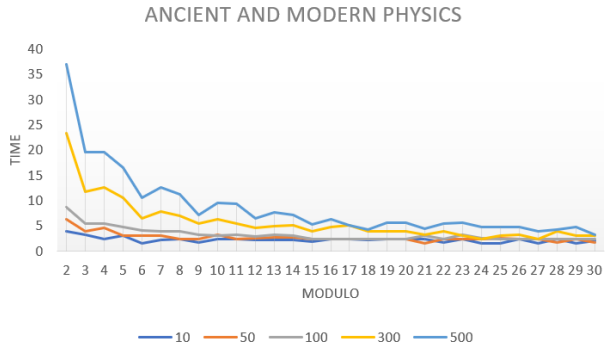

Figure 9: The Rabin-Karp algorithm on an English scientific text: Ancient And Modern Physics by Thomas E. Willson

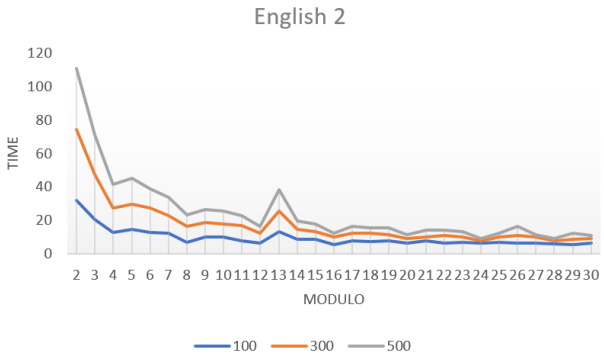

Figure 10: The Rabin-Karp algorithm on another randomly generated English text.

Our another guess was that the performance depending on moduli may somehow be related to the language or to the alphabet of the text. Therefore we tested on other languages, including German (Figure 11), and other languages using other alphabets, including Russian (Figures 12 and 13.) We see that we can get all types of behavior.

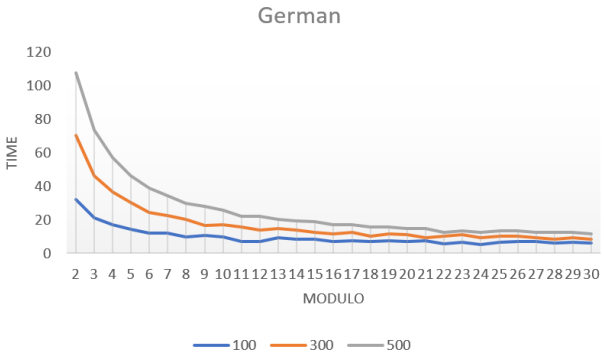

Figure 11: The Rabin-Karp algorithm on a randomly generated German text.

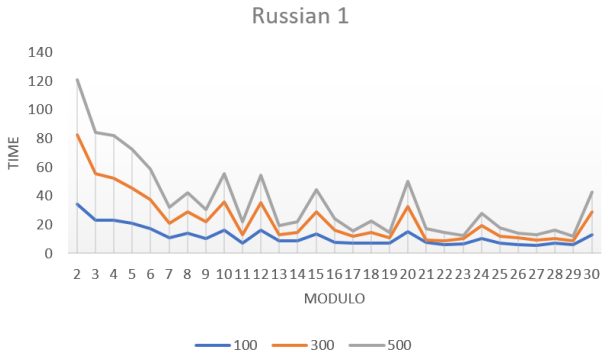

Figure 12: The Rabin-Karp algorithm on the first Russian text (randomly generated).

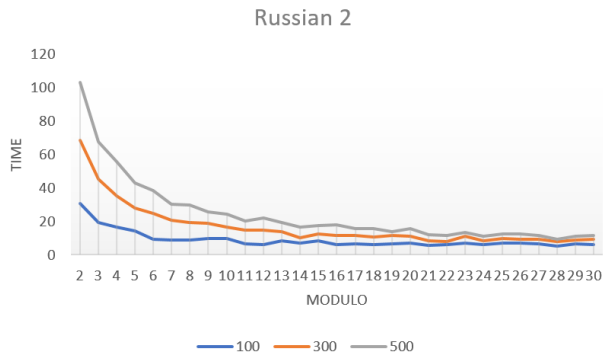

Figure 13: The Rabin-Karp algorithm on the second Russian text (randomly generated).

We were also interested what is happening in Slovak language. Below you see the results of three tests. The first one, Figure 14, is on a randomly generated Slovak text, the second one, Figure 15, is on the original Slovak fiction, Adam Šangala by Ladislav Nádaši-Jégé, and the third one, Figure 16, is on the Slovak translation of French Molier's Lakomec (L'Avare).

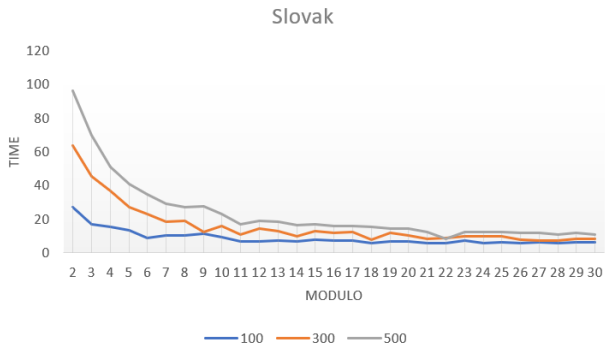

Figure 14: The Rabin-Karp algorithm on a randomly generated Slovak text.

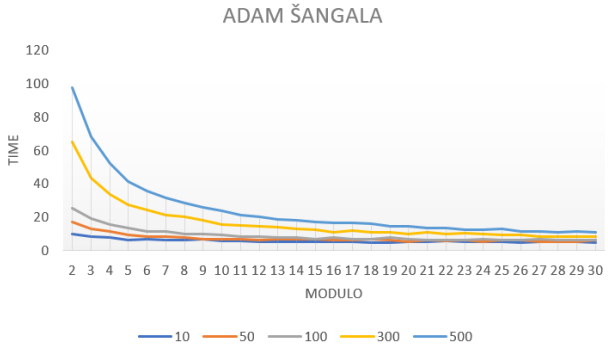

Figure 15: The Rabin-Karp algorithm on the original Slovak fiction, Adam Šangala by Ladislav Nádaši-Jégé

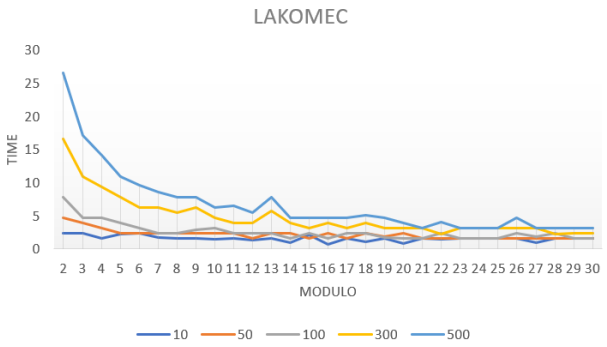

Figure 16: The Rabin-Karp algorithm on the Slovak translation of Molier's Lakomec (L'Avare).

Here it seems that $q = 13$ is again behaving worse that other moduli. (Why?) Just to make a comparison, we also include the results of the testing on the original French text. As we see below, Figure 17, modulus $q = 13$ is not different from others, while $q = 7$ and in some extend its multiples are now the moduli with the worst performance.

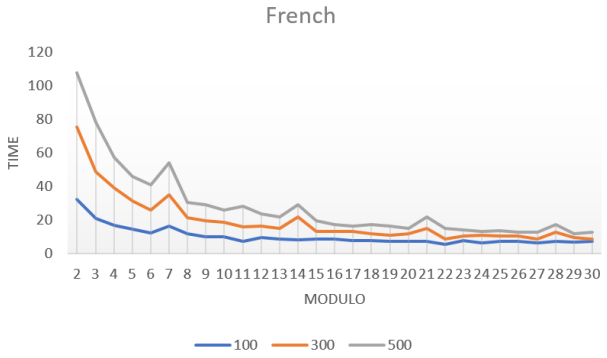

Figure 17: The Rabin-Karp algorithm on the French text.

We were hoping that by systematically running tests on different types of texts, on different languages and alphabets, with different patterns to search for, we would be able to understand the relationship between the computational time and the modulus for Rabin-Karp algorithm. At this moment we see that this tests are not sufficient to explain this relationship, and that other, more involved (probably statistical) methods would be needed to completely understand the phenomenon.

Acknowlegment

The first author acknowledges the support by VEGA 1/0039/17 and VEGA 1/0719/18

References

[1] Marc GOU, Algorithms for String matching, July 30, 2014

[2] Nimisha Singla, Deepak Garg, January 2012, String Matching Algorithms and their Applicability in various Applicationsl, International Journal of Soft Computing and Engineering (IJSCE) ISSN: 2231-2307, Volume-I, Issue-6, January 2012

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, 2009, Introduction to Algorithms: Third Edition, Massachusetts Institute of Technology, Cambridge, Massachusetts 02142 <http://mitpress.mit.edu>

[4] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974

[5] Richard M. Karp and Michael O. Rabin, Efficient randomized pattern-matching algorithms, IBM Journal of Research and Development, 31(2):249-260, 1987

[6] Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt, Fast pattern matching in strings, SIAM Journal on Computing, 6(2):323–350, 1977

[7] Edward M. Reingold, Kenneth J. Urban, and David Gries, K-M-P string matching revisited, Information Processing Letters, 64(5):217–223, 1997

[8] Zvi Galil and Joel Seiferas, Time-space-optimal string matching, Journal of Computer and System Sciences, 26(3):280–294, 1983

[9] Vidya SaiKrishna, Prof. Akhtar Rasool and Dr. Nilay Khare, String Matching and its Applications in Diversified Fields, IJCSI International Journal of Computer Science Issues, Vol. 9, Issue 1, No 1, January 2012