TEI of Crete

Technological Educational Institute of Crete

First project to subject Computational Intelligence

Solving Substitution Ciphers with Genetic Algorithms

January 21, 2018

Authors: Ladislav Šulák, xsulak04@stud.fit.vutbr.cz
Krisztián Benko, kristianbnk@gmail.com

Contents

1 Introduction

This documentation describes a design and a development of solving Vigenere substitution cipher by using genetic algorithm. Every aspect of the development is described - from analyzing of the issue, choosing the solution and dataset, design of application, implementation of the final cipher cracker to results and proposal improvements. The aim of this project was to show that there is a sense to implement genetic algorithm instead of brute force technique, while trying to crack the cipher.

The report is conceived as follows: in the section 2 we describe the main points of our assignment such as the motivation, dataset and some details about the this project and it's steps. In the next section 3 it is described how we approached to this problem and there are also details about the algorithm we created. In the section 4 we describe each particular part of the application in more details. Section 5 deals with testing scenarios on a dataset we chose. Section 6 we analyze obtained results, conclusions and possible future improvements. The last section 7 summaries the observations we found.

2 Analysis

The main requirement of this project was to implement an application, which involves evolutionary computation technique and also to choose a dataset for the input of this application. So first we had to do a research to find out what possible applications are, where it makes sense to implement evolutionary computation technique. After we found the application, we decided what will be the input and how we will get the dataset. The whole project can be decomposed to more separated tasks we were dealing with.

2.1 Research work

At first, we were looking for suitable applications with the usage of some kind of evolutionary computation technique. Either it was: genetic algorithm (GA), genetic programming (GP), evolutionary programming (EP), evolution strategies (ES) or Particle Swarm Optimization (PSO). In the best case we wouldn't have to implement the application, but just use it, create the dataset and work on testing and results. Our choice was an application for cracking the polyalphabetic substitutional cipher. To be more particular, we chose to decrypt text which is already encrypted with Vigenere cipher. The best evolutionary computation technique we could apply on this particular problem was genetic algorithm according to the literature. We were inspirated by following papers and articles: [1], [2], [4], [5], [3]. We found some similar applications, but we were not satisfied with their function or the genetic algorithm was used in a strange way, what means it didn't contain the typical genetic algorithm's parts. So our choice was, that we are going to develop the application from scratch.

2.2 Dataset

The next part of the project's assignment was to find out the suitable data for our application. In our case we needed a lot of English text what we would be able to be divided into smaller parts and from those we would create separate tests. Each test will consist of the original plain text, cryptographic key, which will be generated pseudo-randomly, and encrypted text, what will be the input of the application and what we will try to crack. Regarding our set of plaintexts, we used[1] for text generation and regarding cryptographic keys, we wrote our own manually or used random generator from standard library of Python programming language.

2.3 Decomposition of the project

The whole project from our perspective was broken down into following parts:

1. Find suitable application
2. Define the data
3. Presentation of the previous two steps
4. Create a software design of the application
5. Implement the application

[1]http://randomtextgenerator.com/

6. Testing and experimentation
7. The results and its analysis
8. Presentation of the application and the results
9. This final report

3 Design

Now as we know what application we are going to create, we have to design how we will implement each part of the application. At the start we take a look at how we should done the cracking of the cipher and then we determine how it could be improved by genetic algorithm.

3.1 Cryptoanalysis of Vigenere cipher

Vigenere cipher is a polyalphabetic substitution cipher, created by Blaise de Vigenère. In general it uses a (cryptographic) key and (Vigenere) table, which allows to replace a given letter from plaintext to an another letter (or the same one). This means that each letter in the plaintext can be transformed into 26 possible letters (length of alphabet). So if the input plaintext has n letters, there is 26^n possible options how to decrypt text since encrypted and decrypted texts have the same length. This means that going through the whole space of all possible solutions requires a great computing power. The length of the key can be smaller than the plaintext. In that case the key is being repeated so the final key has always the length of the plaintext and encrypted text.

For decrypting purposes the most common techniques use statistical methods in order to find the key length and then a simple frequency analysis for finding the key itself. Since the algorithm and Vigenere table is known, the key was the only secret needed for the decryption. Our solution is basically a heuristic approach which deals with guessing the correct key.

Kasiski test

Kasiski test consists of finding a repeating sequences of letters in the cipher text. The fact that repeating letters can be found means two things: either a same sequence of letters of the plaintext is encrypted with the same part of the key, or different sequences letters are encrypted with different parts of the key but they ends with the same encrypted letters. The second possibility is poorly probable.

By analyzing the gaps between two identical redundant sequences, an attacker can find more probable key lengths. By analyzing each gap in the term of number of letters and by calculating divisors, an attacker can deduct the size of the key with a high probability.

3.2 The way of using genetic algorithm

Since we are going to implement the cracker using genetic algorithm, the knowledge of the key length should be enough for the start.

As we know, the genetic algorithm starts with generating of a new population. So we take the first (in case that there will be more key lengths) possible key length and create an initial population. Population will consist of pseudo-randomly generated keys of the same key length and the size of population will be determined by user's choice. One key represents one member of the population.

Now we came to fitness function. We need to calculate the fitness value for each population member, what means we have to evaluate each cryptographic key in a given population. There were more possibilities how to do it, but we chose that we are going to use classifier based on Markov Chain[2] and PyEnchant[3] library in Python. The first one is basically a gibberish detector. It means that it can tell with a certain confidence if the word is similar to word from some human language. The second one is just a checker if the input word belong to the specified language, in our case English. It should be noted that it is easy to work with more languages than just English. So during fitness measurement we combine both approaches and we calculate the probability or the confidence if the word belongs to English language.

Next we are going to reproduce selected individuals to form new population. The population size should be always even, because we are going to choose a pairs of members by roulette wheel selection algorithm. Each key gets it's own part in the roulette wheel. The size of this part depends on key's fitness value, bigger value - bigger part in roulette wheel or higher probability that bigger value will be chosen. By this process the pairs of population members are being created.

We can now apply crossover and mutation to obtained pairs, but first we have to convert the keys from string representation to binary representation. After this we will first apply crossover. We can chose the type of crossover in the very beginning of the application - it can be either one point crossover or two points crossover. After that the mutation will be next. In this part there will be an exchange of two bits between a two keys. Positions of the bits are pseudo-randomly generated.

From these new created members (keys) we form a new population and we will iterate until the end of number of generations is reached or the threshold value is exceeded. At the end we will get the best solution.

[2]https://github.com/rrenaud/Gibberish-Detector

[3]https://pypi.python.org/pypi/pyenchant/

3.3 Algorithm

In this section the algorithm we designed is depicted in the figure 1. It uses genetic algorithm which was detailed more in the previous section.

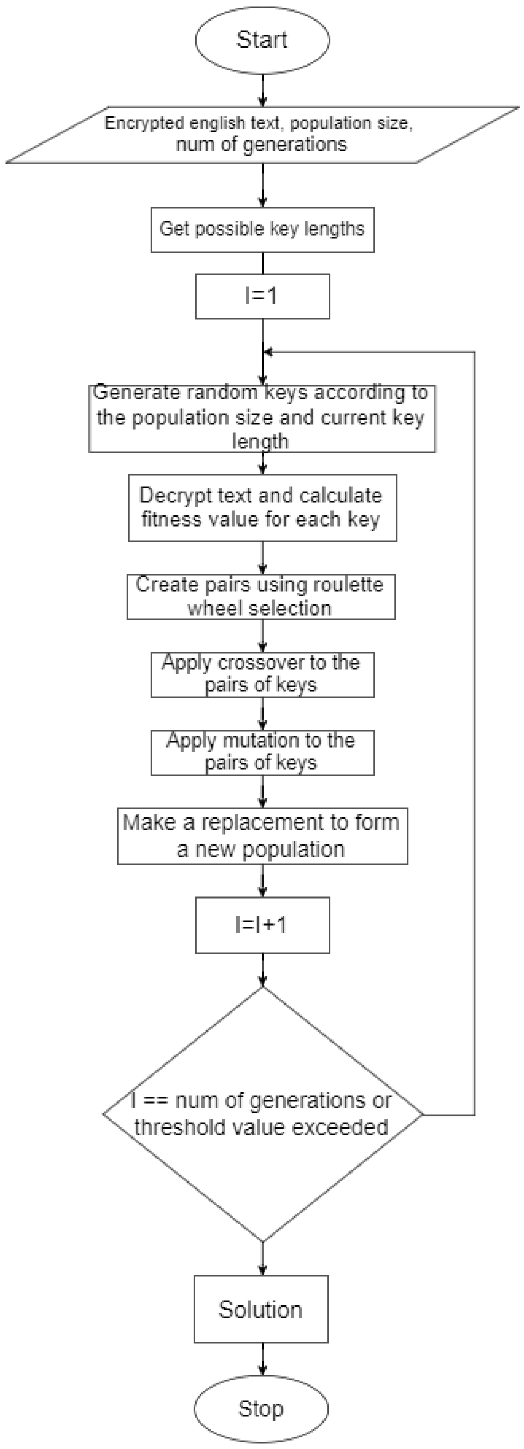
Start
Encrypted english text, population size, num of generations
Get possible key lengths
I=1
Generate random keys according to the population size and current key length
Decrypt text and calculate fitness value for each key
Create pairs using roulette wheel selection
Apply crossover to the pairs of keys
Apply mutation to the pairs of keys
Make a replacement to form a new population
I=I+1
I == num of generations or threshold value exceeded
Solution
Stop

Figure 1: The algorithm of final application

4 Implementation

In this section we are going to describe the implementation of the application. Here it will be described what prerequisites were needed, how we implemented each part of the genetic algorithm, what modification had to be done in each part of algorithm to make working the cracker.

For implementation purposes we chose Python programming language in version 3.6. Also for easier experimentation we used Bash scripting language. The whole project was implemented and tested under Fedora 26 and Ubuntu 16.04 operating systems, both were 64-bit. Regarding other than standard libraries from Python, we use following libraries of 3rd parties: frogress, pycipher, enchant and Markov Chain classifier[4]. First three are available via *pip*[5].

4.1 Key length

According to Kasiski test, which was mentioned in the chapter 3, we created a procedure which is trying to guess all the possible key lengths. From the array of key lengths we try to guess a correct key length by using a genetic algorithm. The key length can be specified in command line by argument too, which was helpful mainly in experimentation and testing phase.

4.2 Initialization of population

We have used an uniform pseudo-random generator from standard library from Python for generation of initial population. So we have the keys, one by one, and now we can try to decrypt the encrypted text using all generated keys.

4.3 Roulette Wheel Selection

During implementation of Roulette wheel selection we were heavily inspired by it's description from lecture slides of the subject Computational Intelligence from TEI of Crete. Regarding to pseudo-random number generator, we used a one from standard library of Python.

4.4 Crossover and Mutation

Crossover and Mutation operations were inspired from [3] with the difference that we implemented not just one point crossover like they did, but also two point crossover and we did experiments with both (always just one of them during 1 runtime). Regarding to mutation, we implemented a very simple one, which is working only with 2 bits.

4.5 Fitness function

The first thing we have to mention is that we worked only with plaintexts containing English words. It is easy to adapt our application to other languages, but it was out of the scope of this work. In the section 3.2, we are discussing our design of evaluating fitness value.

Briefly, we wanted to evaluate a cryptographic key which represents one population member. We decrypt the input decrypted text with such key and we have plaintext which we will

[4]https://github.com/rrenaud/Gibberish-Detector

[5]Python packaging manager, see https://pip.pypa.io/en/stable/

evaluated. Our idea is to determine how similar is the resulting plaintext to English language which will be representing by 1 number. If this number is high, that means that there is a similarity with English language or even it already contains some English words. If this fitness value is small, that means that our key is useless and far from final solution, because decrypted text contains mostly gibberish text.

To be more in detail, we train Markov Chain classifier with English version of the book of `Sir Arthur Conan Doyle - The Adventures of Sherlock Holmes` with some additional words for improving a vocabulary. This Markov Chain classifier works with n-grams (we chose $n = 2$) and re-running our application does only need a model which is already trained and saved as a local file.

However, Markov Chain classifier is just an approximation of the problem of determining if the word is similar to some language or not. Therefore we also implemented an algorithm, which is very simple in its nature and it takes an advantage of both Markov Chain classifier and PyEnchant library. On its input it has 1 word which is being analyzed and it returns a floating point number representing a confidence about how a word fits to English:

1. if the length(word) >= 4 and word is from English dictionary (enchant lib), return a high confidence value, else continue
2. obtain all possible substrings from the input word if length(substr) >= 4, else skip such substrings
3. check all obtained substrings on a match with English dictionary (enchant lib). If there is at least 1 match, all such occurences will be saved as $len(substr)/len(word)$ value and the maximum value will be stored
4. calculate score of the input word via Markov Chain classifier
5. compare score obtained from step 3 and 4 and return a bigger value

Then for a fitness value we just sums all of such values and result represents the measure of the whole decrypted text - how the whole text fits to English.

5 Testing

For the tests and experiments we used dataset we created and which is described in 2.2. The encrypted files were the input and the used key for encrypting should be the output in case of passed test. However, genetic algorithm works with random generations and random numbers, so we usually can't expect the exact key on the output. We can expect that we will get maybe only a couple of letters from the key. So we are not going to evaluate the tests just as passed and failed, but we will evaluate the percentage of match between original key and key on the output. We will also try to measure time, which was needed for finding the key.

5.1 Percentage of match

This section contains table 1 and table 2 which contains some of our experiments. The main observation is that with the small key lengths and small or relative bigger encrypted text our solution is quite precise, but when the key is bigger our solution is just a poor approximation of this problem with mostly gibberish words in the resulted encrypted text. In the figures 2 and 3 there are charts which display both tables.

Test	Test1	Test2	Test3	Test4	Test5	Test6
Length of encoded text	40B	11B	38B	40B	28B	170B
Size of key	3B	8B	8B	10B	6B	60B
Original key	key	superkey	password	superkeyys	supkey	*too long*
Output key	key	supekkey	qesswprd	afgerkeysi	supkey	*too long*
% of match	100%	87%	62%	60%	100%	20%

Table 1: Population size 40, number of generations 160 - % of match between original and output key

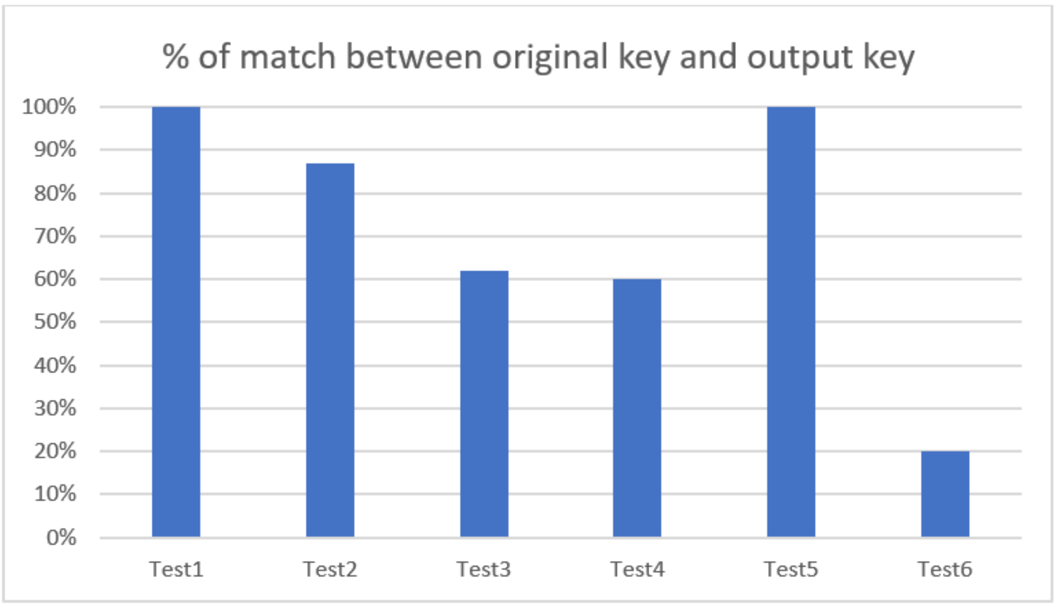

Figure 2: The chart for table 1

Test	Test7	Test8	Test9
Length of encoded text	60B	183B	257B
Size of key	7B	10B	9B
Original key	habbaba	longggkeyy	simplekey
Output key	halbaha	lonsggkeyy	simplekey
% of match	71%	90%	100%

Table 2: Population size 40, number of generations 300 - % of match between original and output key

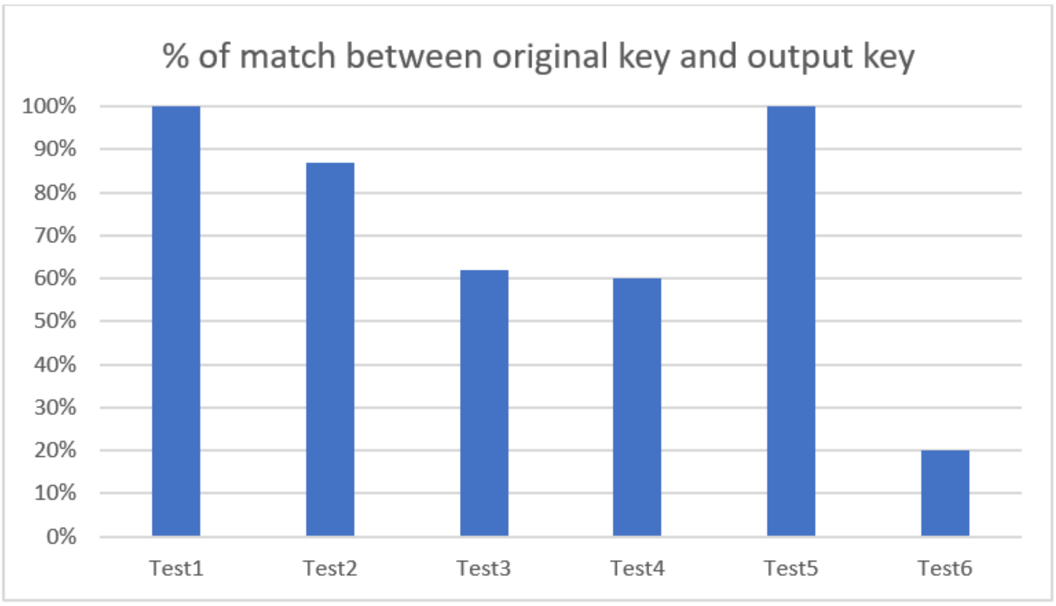

Figure 3: The chart for table 2

5.2 Spent time

This subsection contains table 3 and a chart 4 for having a better look and comparison of each spent time.

Test	Test1	Test2	Test3	Test4	Test5	Test6	Test7	Test8	Test9
Time [min]	6	40	50	55	25	2000	30	60	70

Table 3: Spent time in each test during decryption.

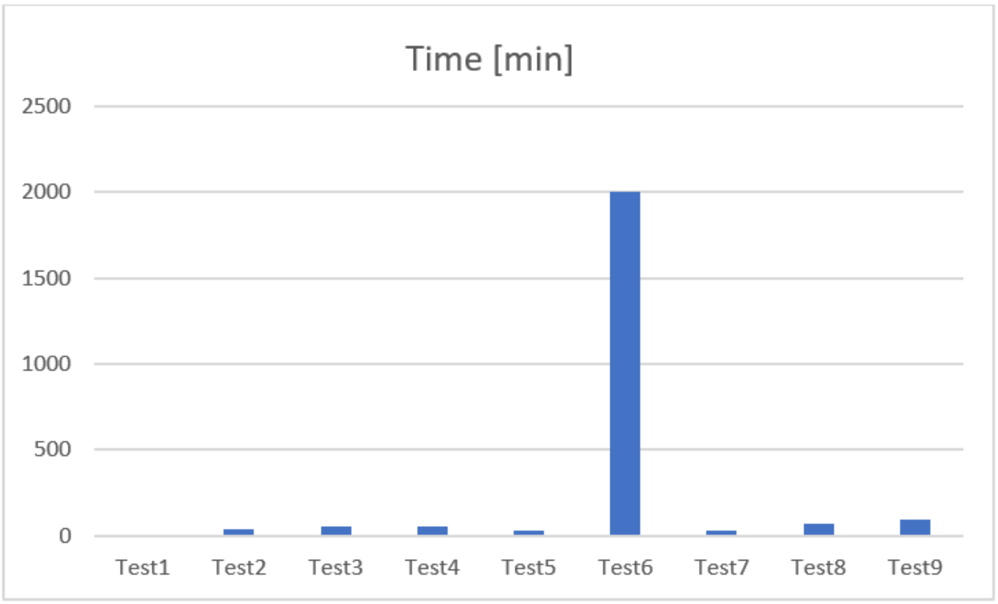

Figure 4: The chart for table 3

5.3 Spent time by brute force method

Another method was created, by which we were trying to crack the key by brute force. This method helped us to compare brute force method and genetic algorithm, and find out if genetic algorithm was useful. With the determination if the key is a correct one we used similar approach as in our fitness measurement - we use Markov Chain classifier and PyEnchant library to determine if decrypted text is in English language. That is why the time required for brute forcing was so great. In cases when the key length is bigger than 5 or 6 characters, the time required for finding a correct key was in couple of days or even hundreds of days in case of bigger text and bigger key. For some examples we provide figure 4 with its chart for better visualization of estimated time in figure 5.

The time was being estimated automatically with the frogress library in Python. Since it is known how many keys will be generated per each key length and based on generation and evaluation time of some relatively small number of keys the final time required per each key length and decrypted text were estimated. It should be noted that this can differ on other CPU and this brute force computation can be heavily distributed and paralyzed. However, for our demonstration purposes we are working only with an estimation times, without a parallel computation and on Intel Core i5-3210M.

Test	Test1	Test2	Test3	Test4	Test5	Test6	Test7	Test8	Test9
Time [days]	0.0007	48000	48500	28000000	60	1.5E78	1500	30000000	1100000

Table 4: Spent time in each test during decryption by brute force method.

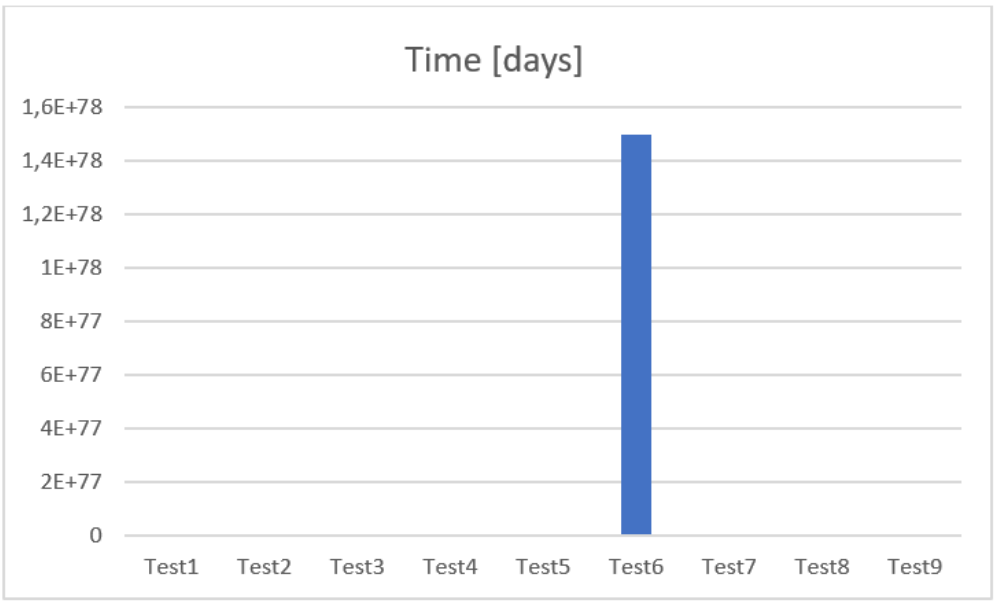

Figure 5: The chart for table 4

6 Results

After first tests some improvements were done in certain parts of genetic algorithm for better performance and to maximize the success of key acquiring. In this section you will also see, why it was useful to implement genetic algorithm for solving this problem and also some suggestions for possible future improvements.

6.1 Improvements during our experiments

At the start after very first tests, when the results were worse than results in section 5, we started to improve our application. The improvements were mainly focused on the genetic algorithm and its operations and also on the evaluation of fitness function. First we replaced one-point crossover for two-point crossover. Then we tuned the size of population and number of generation. The best results we found with population of size 40 members. With higher size of population we usually stuck at local minimum and with lower there were not enough combinations and variability. As you can see in table 1 and in table 2, when we used higher number of generations, we got better results. But also according to table 3, more generations meant more time spent by genetic algorithm and it showed that it doesn't necessary mean that it will lead to solution. Regarding improvements in fitness evaluation, we tuned parameters in Markov Chain classifier, we experimented with a word check via PyEnchant and tried to find a best way how to merge these two checks to determine 1 floating point value which would represent the final fitness value. It can be considered as a confidence if a given word belongs to English language.

6.2 Notes and future improvements

In table 4 you can see the sense of this work. Brute force method was not able to find the key in the reasonable amount of time. That means that for automatic obtaining the correct cryptographic key the approximation is definitely needed. There are more ways how to achieve this goal, for example to use neural network or other machine learning techniques mainly focused on the supervised learning. This or also some other improvements in our algorithm could lead to more precise and faster Vigenere decryptor on machine learning basis.

7 Conclusion

The aim of this project was to find, design and implement a practical application with the usage of evolutionary computation techniques. It has been chosen a decryption of text encrypted by Vigenere cipher with an usage of genetic algorithm. We found out that in comparison to brute force method it is a good approximation regarding computation time, since determination of key by brute force method can run for a couple or even a hundreds of days when the key length is bigger because it is growing exponentially with increasing key length. With our approach it could be find with a couple of minutes or hours, depending on key length and probability since there is a lot of randomness in genetic algorithm and its operations mainly. Regarding the correctness of the solution, in brute forcing method it is always found, but with genetic algorithm approach the correct solution is found only with the small key lengths. With the bigger keys it is only the approximation to this problem, which means in practical way that the final decrypted text could contain many nonsense, gibberish words, but also some fragments of correct plaintext. Our work was primarily focused on English text, but it could be easily adapted to other language. However, with our approach it is not possible to decrypt text other than a one in some of human languages, because of measurement of each member of the population via fitness function.

References

[1] Bagnall A. The application of genetic algorithm in cryptanalysis, 1996.

[2] Joe Gester. Solving substitution ciphers with genetics algorithm. 11 2017.

[3] Safaa Omran, A.S. Al-Khalid, and Dalal Alsaady. A cryptanalytic attack on vigenère cipher using genetic algorithm, 09 2011.

[4] Karel P. Bergmann, Renate Scheidler, and Christian Jacob. Cryptanalysis using genetic algorithms, 01 2008.

[5] Ragheb Toemeh and Subbanagounder Arumugam. Applying genetic algorithms for searching key- space of polyalphabetic substitution ciphers. 5:87–91, 01 2008.