# A Simulated Annealing Approach to Cryptogram Classification 

Guilherme Martins Amado<br>Mestrado em Segurança Informática<br>Departamento de Ciência de Computadores<br>2022

## Orientador

Rogério Reis, Faculdade de Ciências

## Coorientador

António Machiavelo, Faculdade de Ciências

${ }^{\mathrm{F}} \mathrm{C}$
FACULDADE DE CIÊNCIAS UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, $\qquad$ 1


## Declaração de honra

Eu, Guilherme Martins Amado, inscrito no Mestrado em Segurança Informática da Faculdade de Ciências da Universidade do Porto declaro, nos termos do disposto na alínea a) do artigo $14 .{ }^{\text {º }}$ do Código Ético de Conduta Académica da U.Porto, que o conteúdo da presente dissertação de tese reflete as perspetivas, o trabalho de investigação e as minhas interpretações no momento da sua entrega.

Ao entregar esta dissertação de tese, declaro, ainda, que a mesma é resultado do meu próprio trabalho de investigação e contém contributos que não foram utilizados previamente noutros trabalhos apresentados a esta ou outra instituição.

Mais declaro que todas as referências a outros autores respeitam escrupulosamente as regras da atribuição, encontrando-se devidamente citadas no corpo do texto e identificadas na secção de referências bibliográficas. Não são divulgados na presente dissertação de tese quaisquer conteúdos cuja reprodução esteja vedada por direitos de autor.

Tenho consciência de que a prática de plágio e auto-plágio constitui um ilícito académico.

Guilherme Martins Amado
Porto, 17 de Outubro de 2022

## Abstract

In this work, we present a machine-learning model for cipher classification.
For a very long time, humanity has needed to send messages in secret. For just as long, there has also been a need to reveal the contents of those secret messages from other parties. To ensure that messages remained secret when intercepted, ciphers were invented. While older "classical" ciphers are no longer in use, there is still much interest in developing tools that help crack these, for both historical and recreational reasons.

Throughout history, multiple heuristics have been invented to help identify a given ciphered text's cipher. One can start to have an idea of which ciphers may have been used by looking at the results of heuristics aimed at identifying different ciphers' traces. To help with this task, multiple cipher classifiers were created to provide an automatic educated guess. However, these often rely on neural networks or only work for very particular variations of the ciphers, not giving much control to the cryptanalyst.

In this work, we present a simulated annealing cipher classifier that, along with giving more control to the cryptanalyst, also shows which heuristics are more helpful at cipher identification. This opens up the possibility for the cryptanalyst to improve or discard certain heuristics depending on their performance. To measure our model performance we built an example program to identify 12 ciphers using 6 heuristics, whose performance was also measured. Our example program correctly guesses the cipher used around $64 \%$ of the time, on average.

Finally, we make some observations regarding how well the classifier performed and how this performance compares to our expectations. We also comment on how the classifier compares to the state of the art, and on how our model could be improved upon.

## Resumo

Neste trabalho apresentamos um modelo de machine-learning de classificador de cifras.
A humanidade tem, há muito tempo, a necessidade de enviar mensagens em segredo. Por outro lado, quase tão velha é a necessidade de revelar o conteúdo dessas mensagens secretas por adversários. Para garantir que as mensagens permanecessem secretas quando intercetadas, as cifras foram inventadas. Embora cifras mais velhas (clássicas) tenham caído em desuso, há ainda muito interesse em desenvolver ferramentas para descodificar estas sem a chave.

Ao longo dos tempos várias heurísticas foram inventadas para ajudar a identificar a cifra usada num dado texto cifrado. Combinando os resultados de várias heurísticas cujo objetivo é detetar traços de várias cifras, é possível começar a ter uma ideia de quais terão sido usadas. Para ajudar nesta tarefa, vários classificadores de cifras automáticos foram criados para dar um palpite instruído sobre qual cifra terá sido usada. No entanto, estes frequentemente utilizam redes neuronais ou funcionam apenas para variações específicas de cifras para funcionar, não dando muito controlo ao criptoanalista.

Neste trabalho apresentamos um classificador de cifras que usa "simulated-annealing" e que, para além de dar mais controlo ao criptoanalista, também mostra quais heurísticas mais contribuem para a identificação da cifra correta. Isto torna possível ao criptoanalista melhorar ou até remover heurísticas consoante o seu desempenho no processo de identificação. Para medir a performance do modelo construímos um programa exemplo, capaz de identificar até 12 cifras usando 6 heurísticas, cuja performance também foi medida. O nosso modelo exemplo é capaz de adivinhar corretamente a cifra usada $64 \%$ das vezes em média.

No final, fazemos algumas observações sobre a performance do classificador, e como esta se compara às nossas expectativas iniciais de performance. Para além disto, também fazemos considerações sobre de que forma e que o nosso modelo se compara ao estado-de-arte e de que forma poderá ser melhorado.

## Agradecimentos

Em primeiro lugar, quero agradecer aos meus orientadores, o professor Rogério Reis e o professor António Machiavelo, por todo apoio e paciência ao longo da tese. Estou certo que se tivessem cabelos para puxar os teriam puxado muitas vezes.

Quero também agradecer à minha família por sempre se ter preocupado com a minha educação, não foi barata!

Finalmente quero agradecer aos meus amigos: aos que fizeram tese ao mesmo tempo, e procrastinaram comigo, e aos outros que fizeram os possíveis para que eu não procrastinasse.

## Contents

Abstract ..... i
Resumo ..... iii
Agradecimentos ..... v
1 Introduction ..... 1
1.1 State of the art ..... 3
1.1.1 Tools and classifiers available on the Internet ..... 3
1.2 Structure of the thesis ..... 7
2 Background ..... 9
2.1 Terminology ..... 9
2.2 Enciphering methods of classical ciphers ..... 10
2.3 Cipher keys ..... 11
2.3.1 Tabula Recta ..... 12
2.3.2 5x5 Polybius square ..... 12
2.4 Caesar cipher ..... 14
2.5 Vigenère cipher ..... 15
2.6 Autokey ciphers ..... 15
2.7 Bifid cipher ..... 16
2.8 Trifid cipher ..... 18
2.9 Homophonic substitution ciphers ..... 19
2.9.1 Numbered key cipher ..... 20
2.9.2 Chequerboard cipher ..... 20
2.10 Phillips cipher ..... 22
2.11 Playfair cipher ..... 23
2.12 Nihilist Substitution cipher ..... 24
2.13 Nihilist Transposition cipher ..... 25
3 Heuristics ..... 27
3.1 Character frequency ..... 28
3.1.1 Alphabet size of the cryptogram ..... 29
3.1.2 Index of coincidence ..... 31
3.1.3 Detecting transpositions with character frequency ..... 33
3.2 Discovering periodicity with the Index of Coincidence ..... 36
3.2.1 Automatically detect periodicity ..... 37
3.3 Phillips signature ..... 41
3.3.1 Finding the period ..... 41
3.3.2 A better approach to find the period ..... 42
3.4 Non-connected digraphs ..... 45
3.4.1 Enumerating all possible cases for non-connected digraphs ..... 46
3.4.2 Probability of character occurrence within a ciphering block ..... 49
3.4.3 Probability of homogeneous non-connected digraph occurrences ..... 51
3.4.4 Distribution of the standard deviation for non-connected digraphs ..... 53
3.4.5 Building an heuristic for the non-connected digraphs strategy ..... 55
4 Building an automated classifier for classical ciphers ..... 57
4.1 Overview ..... 58
4.2 Automatic identification process ..... 59
4.3 Finding a good weight tuple ..... 60
4.3.1 Score interpretation ..... 61
4.3.2 Tuple rating ..... 62
4.3.3 Weight mutation ..... 63
4.3.4 Simulated annealing approach ..... 63
4.3.5 Further improvements to the algorithm ..... 66
5 Results ..... 69
5.1 Setup for computing results ..... 69
5.2 Expectations ..... 71
5.3 Measuring the performance of the algorithm ..... 73
5.3.1 Algorithm convergence speed ..... 73
5.3.2 Tuple rating vs number of correct guesses ..... 75
5.4 Comparing heuristics ..... 76
6 Conclusions ..... 81
6.1 Expectations and Reality ..... 81
6.2 Comparing our model to the state of the art ..... 82
6.3 Further work ..... 83
A Box-and-whisker plots ..... 85
B Code ..... 87
Bibliography ..... 163

## Chapter 1

## Introduction

For thousands of years, there has been a need to send messages in secret. For just as long, there have been third parties interested in the contents of those messages. To ensure that messages remained private when intercepted, ciphers were invented.

Ciphers are used to encrypt information, that is, to transform information into a cryptogram, making the message unreadable to everyone except the sender and intended receiver. This is done by following a procedure that uses some sort of secret key. Using the same secret, the receiver can revert the cryptogram back to the original message. This is called deciphering. In this context, the only aim of encryption is to deny the intelligible content to a would-be interceptor [Gai89, p. 10].

Cryptography refers to the study of systems for communication in the presence of adversaries, and cryptanalysis refers to the science of breaking these systems [Riv91, p. 720]. Cryptology is the union of both. As long as there has been Cryptography, there has been Cryptanalysis, that is, the analysis of ciphers and cryptograms with the intention of deciphering the messages without the key. To combat this, new "techniques" have been invented over the years, creating a multitude of ciphers.

In the context of our work, we only look at older ciphers, specifically those that belong to the field of classical cryptography. Unlike modern cryptography, where the ciphers deal with complex bit operations, classical cryptography ciphers the texts by character, and using simpler operations that can be done by hand. "Modern cryptography" as an area of scientific interest appeared around the 1970s, and until then (classical) cryptography was mostly reserved for military use.

However, there is still a lot of interest in the study of "classical" ciphers for multiple reasons. Since these ciphers were originally made to hide military messages, one reason for interest is that there are still many historical cryptograms to decipher [DS20, Dun20, Tom22]. Finding the original content of these messages could improve our understanding of history.

The study of classical ciphers helps one to realise the flaws of older ciphers, making them
a good introduction to the study of cryptography. Because of this, classical ciphers are still often present in capture-the-flag competitions. These are competitions for computer security enthusiasts that consist in multiple puzzles where the players have to find "flags", hidden in vulnerable programs, websites, and, more relevant for our case, ciphered texts [ŠČVB21]. Players often have to break classical ciphers in these challenges in order to find the flag.

Interest in cryptanalysis as an activity for educational and recreation purposes is not new. For example, the American Cryptogram Association was founded in 1930 with the objective of promoting Cryptology popularity. This association has been responsible for the publication of a bimonthly periodic called "The Cryptogram" that includes articles and challenge cryptograms for readers to break. Moreover, the association is also responsible for publishing "The ACA and you", a handbook in which, along with other resources, a summary of a panoply of ciphers is given.

For all these reasons, there is a need for tools that can help cryptanalysts decipher cryptograms. Provided the cipher used is already known to the cryptanalyst, there are many tools already available online to help him "crack" the cipher. The self-titled "Cyber Swiss Army Knife" CyberChef [Gov22], is one of those. It offers multiple mechanisms that allow one to "manipulate data in complex ways without having to deal with complex tools or algorithms", such as:

- Data formatting, such as encoding and decoding operations;
- Encryption and decryption operations of various ciphers;
- Operations related to modern cryptography, such as signing and signature verification;
- Arithmetic and logic operations;
- Networking related operations, such as changing the IP format;
- And many others.

Another popular website is dcode.fr [DCo22], which features tools to help puzzle enthusiasts solve word games, mathematical games, and other types of puzzles. This website offers a vast collection of cipher implementations, so that the user can play with them, as well as an explanation of how each cipher of those works.

However, granted these tools are useful, the first task of the cryptanalyst is, more often than not, to detect what was the cipher used in the production of a particular enciphered text. Furthermore, while some tools already mentioned could be used to that end, these still heavily rely on the human contribution, making what could be a small step in the deciphering process more troublesome.

Therefore, a tool capable of automatically identifying the cipher used on a given text is needed. In this work, we propose a model for such a tool. Although our approach is machine learning, we use a different technique from the already existing written work which gives more control to the user. An analysis of the existing state-of-the-art follows.

### 1.1 State of the art

In our research, we tackle the problem of cipher identification by resorting to a machine learning algorithm designed by us. Machine learning is a research field focused on analysing and building methods that "learn", that is, methods that use data to improve the performance of their tasks. Machine learning is useful in cipher classification, since one can easily generate a lot of cryptograms for an algorithm to learn from.

Regarding machine learning, there are two possible approaches to our problem: feature engineering and feature learning. Feature engineering refers to the practise of constructing suitable features from information that, by itself, could not be used to improve predictive performance. Much of the actual effort in this approach often consists of designing preprocessing mechanisms that can produce suitable features for the machine learning algorithm to use, rather than designing the machine learning algorithm itself. This approach takes advantage of human ingenuity and prior knowledge to compensate for the fact that current learning algorithms have difficulties extracting and organising discriminative information from data [BCV13, p. 1]. Learning algorithms less dependent on feature engineering do not require so much time spent on the designing of the features. This makes them highly desirable since new applications can be built faster with them. This leads to feature learning, where the underlying idea is to automatically discover the information needed for feature detection and classification of raw data. This methodology allows the machine to learn the features without manual intervention.

However, in this work, we use a feature engineering approach, since, in this context, it has the key benefit of allowing one to model features according to known cipher weaknesses in order to train a machine learning algorithm $\left[\mathrm{LKE}^{+} 21\right.$, p. 118]. The work presented in $\left[\mathrm{LMK}^{+} 21\right]$ also took a similar approach.

Unlike these, however, we do not use a neural network, and instead use simulated annealing. The way our model is designed allows one to alter the importance given to each feature in the cryptogram classification after training. This is unlike the previously mentioned neural network approach, which works like a black box where the features are given as input. Because of this, we have a greater control over our classifier.

With this work, we were hoping to be able to tell which features are more or less useful. This can be helpful since one may be unsure of how effective a given feature is. We were also hoping our model would allow us to know which features could be added to improve the model's performance.

### 1.1.1 Tools and classifiers available on the Internet

In order to find the current state-of-the-art on automatic cryptogram classifiers, we started by searching classifiers available online, with the goal of finding related written work. Unfortunately we were initially unsuccessful at finding related written-work, so we mostly focused on classifiers
available online with no written-work. Because of this, we also missed a chance to build on what these projects had already achieved. Of the classifiers we found, we only found four worth mentioning, since the overwhelming majority of the others suffered from the following two main problems:

- The classifier was unable to classify any ciphers relevant for this work;
- There was no information about them available other than the code.

These projects were found by searching on GitHub and Google, using keywords such as "cipher classifier" or "cipher identifier". We only considered projects that had a clear explanation on how to use them.

Enigmator The approach taken by this classifier [Cry17] is to check for the presence or lack thereof of different attributes in the cryptogram. Some of these attributes are the cryptograms alphabet size, the value of the index of coincidence and the existence of certain letters in the cryptogram. This is done sequentially, and the classifier may attribute the cryptogram to a given cipher known to produce the attributes already found without checking all the attributes the classifier is prepared to identify. This means that one could produce cryptograms that can easily fool the classifier, especially since one of the attributes taken into consideration is the presence of certain letters as an indications of what ciphers may have been used. On the other hand, the classifier is often sufficient to identify ciphers from capture-the-flag challenges. The classifier includes the following ciphers:

| Affine | Beaufort | Caesar | Columnar transposition |
| :--- | :--- | :--- | :--- |
| Double transposition | Gronsfeld | Hill cipher (3x3) | Input Autokey |
| Nihilist substitution | Playfair | Polybius cipher | Railfence |
| Scytale | Simple substitution | Vigenère. |  |

The code for this classifier is available online on GitHub.

Classifier available on dcode.fr One of the many tools available on the dcode.fr website is a cipher classifier. We contacted the creator and maintainer of the classifier to ask some questions about it, since at the time there was no information available, which has since been rectified. Upon asking how the classifier worked, we were told that the classifier has a multilayer perceptron neural network-type architecture, that is, a fully connected feedforward neural-network. This type of neural-network is an example of feature learning. On the input layer of the network, the coded messages are given, and each node of the output layer corresponds to a different cipher available on the website. This classifier stands out for the very large number of ciphers it is capable of identifying: around 200 different codes and ciphers.

When asked how the performance of the classifier is measured, and we were told that it is using F1-score. The F1-score is a way of combining recall and precision to get a single measure
which falls between recall and precision [Sas, p. 1]. These are, respectively:

$$
\begin{aligned}
\text { Precision } & =\frac{\# \text { of true positives }}{\# \text { of true positives }+\# \text { of false positives }}, \\
\text { Recall } & =\frac{\# \text { of true positives }}{\# \text { of true positives }+\# \text { of false negatives }} .
\end{aligned}
$$

F1-score is the harmonic average between the two:

$$
F 1-\text { Score }=\frac{2 \times \text { Precision } \times \text { Recall }}{\text { Precision }+ \text { Recall }} .
$$

Overall, this classifier scores around 0.8 using F1-score.
More ciphers are added regularly to the website and the training database is regularly updated. We were also told that the network is trained on a personal computer and that it takes several hours to train.

Boxentriq Another classifier that we found is one that is available on the boxentriq website [ $\AA$ hhl22]. This classifier is capable of telling apart 25 different ciphers and codes. Among these are the following classical ciphers:

| ADFGVX | ADFGX | Atbash |
| :--- | :--- | :--- |
| Beaufort | Beaufort autokey | Bifid |
| Caesar | Columnar transposition | Four-square |
| Input autokey | Monoalphabetic substitution | Playfair |
| Railfence | Two-square horizontal | Two-square vertical |
| Vigenère. |  |  |

Again, we contacted the creator of the classifier to ask if he could tell us how it worked. He told us that it uses a random forest algorithm that takes less than an hour to train on a laptop, with an accuracy of around $80 \%$.

Cryptool and related written work Most of the written work we have found describes a neural network approach to tackle the problem of cipher identification. Of these, the most sophisticated approach we found is Cryptool NCID, which is available online and is described in [ $\left.\mathrm{LKE}^{+} 21\right]$. This classifier uses a neural network model capable of telling apart the following 56 ciphers, most of which are specified by the ACA and you:

| Amsco | Autokey | Baconian | Bazeries |
| :--- | :--- | :--- | :--- |
| Beaufort | Bifid | Cadenus | Checkerboard |
| Cmbifid | Columnar transposition | Condi | Digrafid |
| Four square | Fractional morse | Grandpré | Grille |
| Gromark | Gronsfeld | Headlines | Homophonic |
| Key phrase | Mnmedinome | Morbit | Myskowski |
| Nicodemus | Nihilist substitution | Nihilist transposition | Null |
| Numbered key | Per. Gromark | Phillips | Phillips rc |
| Plaintext | Playfair | Pollux | Porta |
| Portax | Progkey | Quagmire1 | Quagmire2 |
| Quagmire3 | Quagmire4 | Ragbaby | Railfence |
| Redefence | Route transposition | Running key | Seriated playfair |
| Slidefair | Swagman | Tridigital | Trifid |
| Trisquare | Two square | Variant | Vigenère. |

The classifier has an accuracy of $80.24 \%$. This work is a natural next step to that of Nuhn and Knight [NK14], since it covers more ciphers and with better accuracy, using a similar model. In the original paper, a comparison is made between different activation functions and results from previous work, and it is shown how this classifier is better than the previous written-work.

Although it is not written-work [Cip22], shows that support vector machines can also be used to classify ciphers.

The following table resumes the previously described differences between the classifiers.
Table 1.1: Comparison between the different classifiers found.

|  | Ciphers covered | Strategy for identification |
| :--- | :--- | :--- |
| dcode.fr | Around 200 ciphers | Neural Network |
| Enigmator | Common classical ciphers, encodings <br> and hashes | Short program that checks for dif- <br> ferent properties on the cryptogram <br> using if-else clauses |
| Boxentriq | 25 common ciphers and encodings | Random Forest |
| Cryptool | 56 ciphers from the ACA and You | Neural Network |


|  | Time spent in training <br> the classifier | Written work | Learning type |
| :--- | :--- | :--- | :--- |
| dcode.fr | Several hours | No | Feature Learning |
| Enigmator | N/A | No | N/A |
| Boxentriq | Less than an hour | No | Feature Engineering |
| Cryptool | Several hours | Yes | Feature Engineering |

### 1.2 Structure of the thesis

Our work is structured as follows. In the next chapter, we start by giving the reader some background on the ciphers we try to automatically identify using our classifier, as well as a short historical background. We also show the differences between these ciphers, some of which are exploited by our classifier later on. In addition, we present the terminology that we will use throughout the rest of the work.

Chapter 3 covers the heuristics that the classifier uses to tell the ciphers apart. Heuristics are key elements for the cipher classifier to work. Each heuristic attributes a property found in the enciphered text to a group of ciphers. The classifier outputs the most likely cipher to have been used by putting the heuristics guesses together. In this chapter we show how each heuristic was built, using statistical observations of different properties found in the enciphered texts.

In Chapter 4, we explain how the classifier model works and how it is trained using simulatedannealing, a machine learning technique. This training consists in finding a balance between the importance given to each heuristic in the classifier result.

After this, in Chapter 5, we present our implementation of the designed model. We analyse the training and classification performance of the implementation. In addition, we explain the expectations that we had beforehand regarding the performance results, and the motives for these expectations.

Finally, in Chapter 6 we draw some conclusions regarding both the expectations we had, how our work compares to the state-of-the-art, and what we missed or could have been done better.

## Chapter 2

## Background

In this work, we present a program that automatically attempts to identify which cipher, from a set of ciphers, was used to produce a given cryptogram. Given this, in this chapter, we present the ciphers that we use in that program, along with some historical context of their use and some known variations. We also present an overview of those ciphers characteristics, and how these are exploited in our program.

We start by presenting the terminology that we use throughout this work. This is done to avoid any confusion since in the cryptography field terminology varies from author to author, and everyday words can assume a different meaning.

### 2.1 Terminology

This work focuses on the analysis of ciphers, specifically their classification. Ciphers are used to modify secret messages in such a way that only the receiver, who knows what modifications were done, can read them. This transformation follows an algorithm, that is, a series of precise steps which are followed as a procedure. The input to this algorithm, that is, the original message, is called plaintext. The output of the algorithm, that is, the ciphered message, is called cryptogram or ciphertext. The process of transforming a plaintext into a ciphertext is called encipherment. The reverse process, that is, the process of transforming the cryptogram into a plaintext, is called decipherment.

The transformation from plaintext to ciphertext requires one key, a secret piece of information shared between the correspondents. The encipherment process can be compared to that of locking a message within a safe box using a key. Furthermore, in a general sense, using the same plaintext with two different keys results in two different cryptograms. We use the words crack or attack to designate the process of deciphering without knowing the key. Using the previous safe box analogy, cracking would be the act of picking the lock of the box. The person that attempts to crack a cipher is called the attacker.

Depending on the language used, plaintexts are written using a given set of characters. Here, we call alphabet to the ordered set of all the characters that can make up a given domain. For example, the cryptogram alphabet is the set comprised of all the characters of a given cryptogram. For example, if one were to write messages using both English letters and digits, the plain text alphabet would consist of all English letters and digits that could be used in a message. Furthermore, if we ciphered the message using only digits as characters, the cryptogram alphabet would be the set of all digits.

Some ciphers presented in this work rely on the alphabetic order at some point during the enciphering process. However, it is also possible for the encipherer to use a permutation of the alphabet instead. Should the encipherer keep this permutation a secret, the permutation works as a key as well. For this reason, and for every cipher, we assume that instead of the alphabet in its usual order, a secret permutation of the alphabet is used.

Symbols are the characters' visual representation. It is important not to confuse symbols with characters. For example, in the Roman language, the character for the eighth cardinal is represented using four symbols: VIII.

The set of characters that can make up a plaintext for a given cipher is the plaintext vocabulary. The set that can make up a ciphertext is the ciphertext vocabulary.

### 2.2 Enciphering methods of classical ciphers

Classical ciphers, in general, can be divided into three different types [Gai89, p. 1]:
Concealment ciphers Ciphers that disguise or hide the message within another message and are intended to pass without raising suspicions of secret communication. This work does not cover these.

Transposition ciphers Ciphers that employ a character-moving system, so that the ciphertext constitutes a permutation of the plaintext.

Substitution ciphers Ciphers that replace the characters of the original messages with substitutes, keeping the original order.

To add to these, there are combinations of types and combinations of different subtypes belonging to the same type. In this work, we focus on transposition and substitution ciphers. We can further divide substitution ciphers into the following five major ciphering techniques. The reason these five were chosen is that they are the most prevalent among ciphers.

Monoalphabetic substitution This technique is a simple character substitution, and each character is always substituted by the same character. The substituting characters can have a different number of symbols than the plaintext characters. Each character is always replaced by the same character, independently of its location in the ciphertext.

Polyalphabetic substitution These ciphers assign an alphabet permutation to each character position of the plaintext, and then cipher each character using a substitution that takes into consideration the positions' assigned permutation. Thus, generally speaking, the same character at a different position is ciphered differently. Because of this, these ciphers are harder to crack than monoalphabetic ones.

Homophonic substitution These are ciphers that, similarly to polyalphabetic ciphers, can substitute the same character in different ways. However, unlike polyalphabetic ciphers, these ciphers do it independently of where the characters are located in the plaintext. Instead, the substitution is selected in a non-deterministic way, meaning that the encipherer chooses what is the substitute in each instance.

Polygraphic substitution Ciphers in which groups of characters are replaced with other groups of characters. Each sequence of characters has a pre-established replacement.

Fractional substitution These are ciphers that substitute each character with more than a single character, expanding the text. After this expansion, the substitutes are broken and these fractions are subject to further encipherment. An example of this are ciphers that first expand the text by substituting letters for pairs of digits, and on a second phase, convert the digits back to letters using a different method of grouping digits.

In Gaines' book [Gai89], substitution ciphers are only divided into four major methods [Gai89, p. 68]. However, we found it relevant to admit a fifth major method, homophonic substitution, given its prevalence among many classical ciphers. Furthermore, this technique is also notably good at hindering statistical analysis, making it more relevant in the context of this work.

Table 2.1 shows, for the ciphers that we cover in this work, what ciphering methods each cipher uses.

### 2.3 Cipher keys

As previously mentioned in section 2.1 , ciphers use a secret piece of information called the key. Keys may be composed of a single component or multiple components. These components mainly take shape as one of the following three forms:

- A keyword, that is, a phrase, word or code;
- A number, typically used as a shift or as a period;
- A table, filled with characters.

Since any alphabet permutation can be used other than the English alphabetic order, as mentioned in section 2.1 , we assume that the alphabet permutation used is always randomly

Table 2.1: Methods used in each cipher.

|  |  | Ciphering methods |  |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  |  |  | $\begin{aligned} & 0 \\ & 0 \\ & 0 \\ & 0 \\ & 0 \\ & 0 \\ & 0 \\ & 0 \\ & 0 \\ & 0 \end{aligned}$ |  |  |  |  |
|  | Caesar | X |  |  |  |  |  |
|  | Chequerboard | X | X |  |  |  |  |
|  | Numbered key |  | X |  |  |  |  |
|  | Vigenère |  |  | X |  |  |  |
|  | Autokey Input |  |  | X |  |  |  |
| $0$ | Autokey Output |  |  | X |  |  |  |
| 훙 | Phillips |  |  | X |  |  |  |
|  | Nihilist substitution |  |  | X |  |  |  |
|  | Playfair |  |  |  | X |  |  |
|  | Bifid |  |  |  | X | X | X |
|  | Trifid |  |  |  | X | X | X |
|  | Nihilist transposition |  |  |  |  |  | X |

chosen. Since the procedure to make key-tables often relies on the alphabetic permutation in some part of the process, this also applies to key-tables. We now present two of the most popular kind of tables used as components of the key.

### 2.3.1 Tabula Recta

The tabula recta is a table that is filled with characters using the alphabet. The first line of the table is the chosen alphabet permutation, and each row of the table is made by shifting the previous one to the left one character. An example of a tabula recta is shown in Figure 2.1. Ciphers that use a tabula recta are notably similar, as we will see later.

### 2.3.2 5x5 Polybius square

The Polybius square is a $5 \times 5$ table filled with different characters. Usually, ciphers use the coordinates of the characters within the square to perform ciphering and deciphering operations. Its creation is attributed to the ancient Greeks Cleoxenus and Democleitus and was later further developed by the famous Greek historian Polybius, thus bearing his name [Pol89, p. 44]. Apparently, the square was first created for fire signalling, but later found its use both in


Figure 2.1: A Tabula Recta.

Telegraphy and Cryptography.
Given that the square only has 25 slots for characters, the English alphabet does not fit in it. The usual solution is to combine the letters "I" and "J" in the same cell, given that the letter "J" has a very low frequency and both letters look similar.

One way to fill the square, commonly used in the past, is to start by inserting a keyword, removing character repetitions, and then filling the rest of the square with the remaining characters of the alphabet in their usual order. Since there can be no repetitions, it can be filled with characters in 25 ! different ways. The following are some examples of filling schemes. Figure 2.2 displays an example of each of those schemes, using the keyword "triumvirate".

Horizontal We start by filling in the square using the keyword from left to right and top to bottom. Then we continue with the remaining alphabet characters.

Vertical Identical to the horizontal scheme, but first top to bottom and then left to right.
Horizontal serpentine Similar to the horizontal scheme, but the filling direction is inverted in the even rows. We can also do this for the vertical scheme, making a vertical serpentine.

Clockwise spiral The outer squares are filled clockwise and the process is repeated for the remaining inner squares in a spiral way. Again, first, the keyword characters are used and only then the remaining alphabet. The spiral can also be done anticlockwise.

Some ciphers may number the rows and columns differently from the example in Figure 2.2, or even use characters instead of numbers. An example of this occurs in the Phillips cipher (Section 2.10).

|  | Horizontal |  |  |  |  | Vertical |  |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | 1 | 2 | 3 | 4 | 5 |  | 1 | 2 | 3 | 4 | 5 |
| 1 | T | R | I | U | M | 1 | T ${ }^{\text {r }}$ | V | $\mathrm{D}^{\text {r }}$ | L | S |
| 2 | $V$ | A | E | B | C | 2 | R | A | F | $\mathrm{N}^{\prime}$ | W |
| 3 | D | F | G | H | K | 3 | I | $\mathrm{E}^{\prime}$ | $\mathrm{G}^{\prime}$ | $\mathrm{O}^{\prime}$ | $\mathrm{X}^{\prime}$ |
| 4 | L | N | O | P | Q | 4 | $\mathrm{U}_{1}^{\prime}$ | B | ${ }^{\prime}$ | P | Y |
| 5 | S | W- | X | Y | Z | 5 | M: | Ci | K! | Q | Z |


| Horizontal Serpentine |  |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: |
|  | 1 | 2 | 3 | 4 | 5 |
| 1 | T | R | I- | U | M |
| 2 | C | B | E | A | V |
| 3 | D | F | G | H | K |
| 4 | Q | P | O | N | L |
| 5 | 次 | W | X | Y | Z |


|  | Clockwise Spiral |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: |
|  | 1 | 2 | 3 | 4 | 5 |
| 1 | T | R | I | U | M |
| 2 | L | N | O | P | V |
| 3 | K | Y | Z | Q | A |
| 4 | $\mathrm{H}^{\prime}$ | X | W | S | E |
| 5 | G | F | D | C | B |

Figure 2.2: Some Polybius square fill schemes.

### 2.4 Caesar cipher

Suetonius, a historian in ancient Rome, described a cipher used between Caesar and Cicero, along with others [Kah96, p. 83-84]. In order to cipher, they would replace each letter of the message with the one three places after in the alphabet to cipher their messages. Nowadays, this ciphering method is commonly known as Caesar cipher, even if the distance used is other than three [Kah96, p. 83-84]. A particular case is known as ROT13, in which the distance between letters is exactly thirteen and the alphabet used is the English. The ROT47 is another known case, where the ASCII alphabet is used with a distance of forty-seven.

The Caesar cipher is a rather weak cipher, since the number of possible keys is limited by the size of the alphabet. Consequently, it is easy to attack this cipher using brute-force and this can be done by hand. The cipher can also be recognised just by looking at the cryptogram, due to the fact that the cipher does not properly hide the languages' signature, even to the naked eye.

### 2.5 Vigenère cipher

Vigenère was a French cryptographer, alchemist, and diplomat that lived during the 1500s. Although he did invent a polyalphabetic ciphering method, his name has been wrongly associated with another, weaker, polyalphabetic cipher [Kah96, p. 145]. The Vigenère cipher uses a Tabula recta and a keyword to cipher and decipher. For each pair of characters plaintext/key, the following process is repeated:

1. The intersection of the column with the plaintext character on top and the row with the key character on the left is found, this being the corresponding ciphertext character;
2. The process is repeated for the next plaintext character.
3. When the keyword is used up, the cipher process starts from the beginning of the keyword.

The keyword can also be seen as a sequence of shift keys, where each character corresponds to a different character shift, as in the Caesar's cipher. Figure 2.3 shows an example for the plaintext "Fortune favours the bold", using the key "Password" and the tabula recta from Figure 2.1.

| Plaintext: | F | O | R | T | U | N | E | F | A | V | O | U | R | S | T | H | E | B | O | L | D |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Keyword: | P | A | S | S | W | O | R | D | P | A | S | S | W | O | R | D | P | A | S | S | W |
| Ciphertext: | A | S | N | X | C | H | G | B | T | I | Q | H | Q | Q | C | L | I | L | Q | M | M |

Figure 2.3: Vigenère ciphering example.

### 2.6 Autokey ciphers

Autokey ciphers can be described as ciphers in which the original key is expanded through a cyclic process. Usually, in the ciphering process, after the initial key has been used, either the plaintext or the ciphertext itself are used to cipher the next part of the cryptogram.

As stated before, the Vigenère cipher is wrongly attributed to Blaise de Vigenère. In reality, Vigenère did create a similar polyalphabetic cipher, albeit more complex and harder to break. This latter cipher was one of the first autokey ciphers [Kah96, p. 145]. It works like the Vigenère cipher, but once the initial keyword has been used, the plaintext starts being used as key throughout the entirety of the remaining text. We call this input autokey cipher. An example of the cipher is shown in Figure 2.4, where the keyword "Password" is used to cipher the phrase "Fortune favours the bold", using the tabula recta from Figure 2.1.

Other than this cipher, Vigenère also developed another similar cipher, which we call output autokey cipher. This cipher works similarly to the input autokey, but instead of using the plaintext as the key, it uses the ciphertext. This has the advantage of using a key that "looks

| Plaintext: | F | O | R | T | U | N | E | F | A | V | O | U | R | S | T | H | E | B | O | L | D |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Keyword: | P | A | S | S | W | O | R | D | F | O | R | T | U | N | E | F | A | V | O | U | R |
| Ciphertext: | A | S | N | X | C | H | G | B | H | J | U | G | K | Z | W | D | X | Y | Y | A | V |

Figure 2.4: Input autokey example.
random", meaning one cannot know the content of the message by having just the key. An example is shown in Figure 2.5, where the same key and phrase from the previous example are used.

| Plaintext: | F | O | R | T | U | N | E | F | A | V | O | U | R | S | T | H | E | B | O | L | D |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Keyword: | P | A | S | S | W | O | R | D | A | S | N | X | C | H | G | B | R | P | H | B | I |
| Ciphertext: | A | S | N | X | C | H | G | B | R | P | H | B | I | W | Q | S | G | C | I | G | K |

Figure 2.5: Output autokey example.

These ciphers are much less vulnerable to statistical analysis thanks to the key generation process, which avoids the creation of patterns in the cryptogram.

### 2.7 Bifid cipher

The bifid cipher was invented by the French cryptographer Félix Marie Delastelle (18401902) [Kah96, p. 243]. This cipher is a simple yet effective application of the fractional substitution method. In fact, the name bifid comes from the French "bifide", meaning "split in two".

The bifid cipher uses a Polybius square as key, as well as an agreed period $\ell$, normally greater than six [MR07, p. 2]. First, the plaintext is divided into blocks of size $\ell$. Then, for each block, the coordinates of each plaintext character are written underneath it. Taking, for example, the following text and Polybius square and $\ell=9$ :
"Friends, Romans, countrymen, lend me your ears"

|  | 0 | 1 | 2 | 3 | 4 |
| :---: | :---: | :---: | :---: | :---: | :---: |
| 0 | S | H | A | K | E |
| 1 | P | R | B | C | D |
| 2 | F | G | I | L | M |
| 3 | N | O | Q | T | U |
| 4 | V | W | X | Y | Z |

one gets:

| F | R | I | E | N | D | S | R | O | $/$ | M | A | N | S | C | O | U | N | T | $/$ |  |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| 2 | 1 | 2 | 0 | 3 | 1 | 0 | 1 | 3 |  | 2 | 0 | 3 | 0 | 1 | 3 | 3 | 3 | 3 |  |  |
| 0 | 1 | 2 | 4 | 0 | 4 | 0 | 1 | 1 |  | 4 | 2 | 0 | 0 | 3 | 1 | 4 | 0 | 3 |  |  |
| R | Y | M | E | N | L | E | N | D | $/$ | M | E | Y | O | U | R | E | A | R | $/$ | S |
| 1 | 4 | 2 | 0 | 3 | 2 | 0 | 3 | 1 |  | 2 | 0 | 4 | 3 | 3 | 1 | 0 | 0 | 1 |  | 0 |
| 1 | 3 | 4 | 4 | 0 | 3 | 4 | 0 | 4 |  | 4 | 4 | 3 | 1 | 4 | 1 | 4 | 2 | 1 |  | 0 |

In order to obtain the ciphertext, each block is recoded using the same square, by reading pairs of coordinates horizontally, from left to right, as the following scheme suggests. If the period is odd, the last coordinate of the first line is paired with the first coordinate of the second line.


Thus, we would have the following coordinates.

| 21 | 20 | 31 | 01 | 30 | 12 | 40 | 40 | 11 | $/$ |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| 20 | 30 | 13 | 33 | 34 | 20 | 03 | 14 | 03 | $/$ |
| 14 | 20 | 32 | 03 | 11 | 34 | 40 | 34 | 04 | $/$ |
| 20 | 43 | 31 | 00 | 14 | 43 | 14 | 14 | 21 | $/$ |
| 00 |  |  |  |  |  |  |  |  |  |

Which would then be substituted by the corresponding characters in the square, resulting in the following ciphertext.

## GFOHNBVVRFNCTUFKDKDFQKRYVUEFXOSDYDDGS.

If the last block has a different size it is still ciphered using the same scheme. This means that, should the last block have only one character, it remains unciphered. This can be avoided by adding padding to the plaintext.

The name "bifid" is also used to describe the family of ciphers that form a pattern, constituted by two substitution operations and two transposition operations [Wil39, p. 178]. The pattern is as follows.

1. First, a process of substitution takes place, in which each plaintext character is replaced by two components, of a "bipartite alphabet". This is done so that each character is represented by two symbols instead of just one, so that these can be split apart later.
2. Then, there is a process of transposition, in which the components originally paired together are separated.
3. Next, a process of transposition takes place, in which the separated components are combined to form new pairs.
4. Finally, there is a substitution in which each new pair of symbols is substituted back to other alphabet, usually the original alphabet.

### 2.8 Trifid cipher

The trifid cipher is a fractional substitution cipher. It works similarly to the bifid cipher, but instead of partitioning every character representation in two, it partitions in three, hence the name. The cipher is in the "bifid family of ciphers" previously mentioned.

For this cipher, a 27 character alphabet is used, along with a period. This cipher requires a tripartite alphabet since the cipher partitions the characters' representations into three parts. The cipher uses a table as a component of the key. The table is built as follows.

1. The first row of the table contains the characters of the original alphabet. To determine the character order of the first row a keyword is picked, its repetitions are removed, and then the remaining alphabet is added, analogous to what was done before with the Polybius square.
2. Below each character is its respective substitution in the tripartite alphabet. This tripartite alphabet is ternary, ranging from $(0,0,0)$ to $(2,2,2)$.

Figure 2.6 shows an example of a table for the keyword "Pompey Magnus".

| P | O | M | E | Y | A | G | N | U | S | B | C | D | F | H | I | J | K | L | Q | R | T | V | W | X | Z | \# |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 0 | 0 | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 0 | 0 | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 0 | 0 | 0 | 1 | 1 | 1 | 2 | 2 | 2 |
| 0 | 1 | 2 | 0 | 1 | 2 | 0 | 1 | 2 | 0 | 1 | 2 | 0 | 1 | 2 | 0 | 1 | 2 | 0 | 1 | 2 | 0 | 1 | 2 | 0 | 1 | 2 |

Figure 2.6: Trifid table example.

Although the table is a more convenient format, one can also visualize it as a three-dimensional Polybius, as follows.


The following figure shows an example of the trifid cipher, where the phrase "Cease quoting laws to us that have swords girt about us!" is ciphered using the previous table and a period of 8. In this example the letter "\#" to separate the words in the plaintext.

| Plaintext: | CEASE\#QU | OTING\#LA | WS\#TO\#US |  |
| :---: | :---: | :---: | :---: | :---: | :---: |
|  | Ciphertext: | SCLDIXLW | NMLKXDER | WRYNUGQX |
| Plaintext: | \#THAT\#HA | VE\#SWORD | S\#GIRT\#A | BOUT\#US |
| Ciphertext: | ZUCFJIX\# | RIVISDRI | IKL\#ACMU | SXSWQIX |

Figure 2.7: Trifid cipher example.

### 2.9 Homophonic substitution ciphers

Generally speaking, a cipher is called homophonic when there are multiple alternatives to substitute the same plaintext character. Until the invention of polyalphabetic ciphers, homophonic ciphers were the most popular, and remained in use around three hundred years after the invention of polyalphabetic ciphers [Kah96, p. 150]. This is explained by the fact that polyalphabetic ciphers have certain features that made them inadequate to use when they were created, most notably:

Slow to use At the time they were invented, ciphering was made by hand, with no help from machines. Given their complexity, this made the process slow and error-prone.

Mistakes made them unreadable A ciphering error meant the message would be unreadable, or at least part of it. This was very troublesome since it meant sending a messenger back
to retrieve a correctly ciphered message. On the other hand, homophonic ciphers are still readable even if a character or two are mistaken.

Since each plaintext character can be substituted in multiple ways, it is important to avoid ambiguity when ciphering. Ambiguity can occur if two different plaintext characters can be substituted by the same character. This can severely hinder their decipherment. To avoid it, the ciphertext vocabulary must be larger than the plaintext vocabulary, if they are equal in size then it becomes just a monoalphabetic substitution.

Ambiguity may also take place if the substituting characters are represented by multiple symbols, and some have more symbols than others. In that case, a character representation cannot have as prefix another characters' representation, otherwise, there can be ambiguity when deciphering.

### 2.9.1 Numbered key cipher

The numbered key cipher is a homophonic cipher. The cipher substitutes characters using pairs of digits, with some characters having more than one possible substitution. The substitutions are determined in the following manner:

1. A keyword is picked. This can be a phrase, a word, etc., written using the plaintext alphabet.
2. A string is made by appending the remainder alphabet characters not present in the keyword to the keyword. Note that there can be character repetitions in the string if the keyword has repetitions.
3. Each character of the string is then numbered consecutively, using two digits, starting from 1. The numbering process can start somewhere in the middle of the string and wrap around. We call this the numbered key.
4. To cipher, each character of the plaintext is substituted with one of its corresponding numbers in the numbered key.

The process of creating a numbered a key can be seen in figure 2.8. Note that the letters "o", "r", "c" and "i", have multiple substitution options when ciphering.

To decipher, the substituting process is reversed.

### 2.9.2 Chequerboard cipher

The chequerboard cipher is a homophonic substitution cipher, which also has a monoalphabetic substitution variant. We start by explaining the latter variant, since it is simpler. In this variant,

Keyword: "I found Rome a city of bricks"

String: ifoundromeacityofbricksghlpqvwxyz

| i | f | o | u | n | d | r | o | m | e | a | c | i | t | y | o |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 |

Numbered key:

| f | b | r | i | c | k | s | g | h | l | p | q | v | w | x | z |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |

Figure 2.8: Example of a numbered key.
each character is substituted with its coordinates in the Polybius square. The coordinates of the cells in the square do not have to be expressed as digits; in fact, using letters is common. It is also possible to use a different vocabulary for rows and columns coordinates, for example, only digits for rows and only letters for columns. An example of the cipher can be seen in Figure 2.9.

|  | W | H | I | T | E |
| :---: | :---: | :---: | :---: | :---: | :---: |
| B | A | U | G | S | T |
| L | B | C | D | E | F |
| A | H | I | K | L | M |
| C | N | O | P | Q | R |
| K | V | W | X | Y | Z |


| Plaintext: | A | P | P | L | A | U | D | A | S | I | E | X | I | T |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Ciphertext: | BW | CI | CI | AT | BW | BH | LI | BW | BT | AH | LT | KI | AH | BE |

Figure 2.9: Monoalphabetic chequerboard ciphering example.

The other, more complex, variant of the cipher allows each character coordinate to be represented in multiple ways, meaning there can be multiple substitutions for the same character, making it poligraphic. The order of each pair of substituting characters may also be switched, if there are no characters whose coordinates, when switched, are equal to other character coordinates. Figure 2.10 shows an example of this. Note that, since the rows and columns vocabulary is completely distinct, it is possible to switch the order of the coordinates when ciphering without risking any ambiguity.

|  |  | B | L | A | C | K |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  |  | W | H | I | T | E |  |
| 0 | 5 | A | U | G | S | T |  |
| 1 | 6 | B | C | D | E | F |  |
| 2 | 7 | H | I | K | L | M |  |
| 3 | 8 | N | O | P | Q | R |  |
| 4 | 9 | V | W | X | Y | Z |  |


| Plaintext: | A | P | P | L | A | U | D | A | S | I | E | X | I | T |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Ciphertext: | 5B | I8 | 3 A | 2 C | 0B | H5 | A6 | B5 | C5 | 7H | T6 | A9 | H2 | 5 K |

Figure 2.10: Homophonic chequerboard ciphering example.

### 2.10 Phillips cipher

The Phillips is a polyalphabetic cipher. It was "described in an early issue of The Cryptogram as having been used for military purposes, and was called the Phillips system" [Gai89, p. 185]. The cipher uses both a period $\ell$ and a Polybius square as key. It works as follows.

1. Each character is substituted by the one diagonally down to its right in the square. If the character is at the border, the character at the other side is chosen, looping around.
2. After $\ell$ characters, the cipher changes the Polybius square, by moving a row down. This is done seven times. First, the initial first row is moved, until it becomes the last. Then, the initial second row, now the first, is moved until it becomes the fourth. At the eighth iteration, instead of moving rows, the initial setup is restored and the process of moving rows starts again. The changes in row order can be seen in table 2.2 .

Table 2.2: Polybius square row order for each period in the Phillips cipher.

| Polybius square iteration | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | $9=1$ | $10=2$ | $\ldots$ |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | 1 | 2 | 2 | 2 | 2 | 3 | 3 | 3 | 1 | 2 |  |
|  | 2 | 1 | 3 | 3 | 3 | 2 | 4 | 4 | 2 | 1 |  |
| Corresponding row order | 3 | 3 | 1 | 4 | 4 | 4 | 2 | 5 | 3 | 3 | $\ldots$ |
|  | 4 | 4 | 4 | 1 | 5 | 5 | 5 | 2 | 4 | 4 |  |
|  | 5 | 5 | 5 | 5 | 1 | 1 | 1 | 1 | 5 | 5 |  |

Figure 2.11 shows a ciphering example of the phrase "Better a cautious commander, and not a rash one" with a period of five.

|  | 1 | 2 | 3 | 4 | 5 |
| :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | A | G | R | I | P |
| 2 | B | C | D | E | F |
| 3 | H | K | L | M | N |
| 4 | O | Q | S | T | U |
| 5 | V | W | X | Y | Z |


| Plaintext: | BETTE | RACAU | TIOUS | COMMA |
| :---: | :---: | :---: | :---: | :---: |
| Ciphertext: | KNZZN | MKRKV | ZUWVY | LGUUW |
|  |  |  |  |  |
| Plaintext: | NDERA | NDNOT | ARASH | ONE |
| Ciphertext: | OMNEC | BTBWZ | KMKEQ | WOP |

Figure 2.11: Phillips cipher example.

### 2.11 Playfair cipher

The Playfair cipher was invented in 1854 by Charles Wheatstone, an English scientist of the Victorian era. However, the name that remained attached to the cipher is of his friend Lyon Playfair, Baron of St. Andrews, who recommended it to high-ranking government and military persons [Bau02, p. 62]. The cipher found use in multiple wars throughout the following decades.

The Playfair cipher is a poligraphic substitution cipher. The cipher uses a Polybius square as key, preferably not filled with the horizontal filling scheme [Gai89, p. 200]. The Playfair cipher enciphers the characters in pairs, using the following four rules.

1. Each pair must have two different characters. If two characters are the same, a "blank" character has to be inserted between them. The "blank" character is a different character from the ones it is separating. If the text has an uneven number of characters, a blank should be inserted to make it even.
2. If the two characters are in the same column of the Polybius, each is substituted with the one directly below it. The bottom cycles to the top.
3. If the two characters are in the same row, each is substituted with the character directly to the right. The right cycles to the first character of the same row.
4. If both characters are in different rows and columns, they are substituted using the two characters which form a rectangle with them. The first character is the one in the same row as the first of the pair.

Note that the last three rules are cyclic. Thus, as long as the order $0-1-2-3-4$ is maintained in both columns and rows, it makes no difference shifting the columns or the rows [Gai89, p. 200]. For each key-square there are 5 possible column shifts and 5 possible row shifts, meaning that for each key-square there are other 24 equivalent key-squares. Thus, instead of 25 ! possible different key-squares for this cipher, there are 24 !.

Figure 2.12 exhibits an example of the Playfair cipher. In the example, the phrase "let the

|  | 0 | 1 | 2 | 3 | 4 |
| :---: | :---: | :---: | :---: | :---: | :---: |
| 0 | R | U | B | I | C |
| 1 | O | N | A | D | E |
| 2 | F | G | H | K | L |
| 3 | M | P | Q | S | T |
| 4 | V | W | X | Y | Z |


| Plaintext: | LE | TX | TH | ED | IE | BE | CA | ST |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Ciphertext: | TL | QZ | QL | OE | CD | CA | BE | TM |

Figure 2.12: Playfair cipher example.
die be cast" is ciphered using the given square. In the example, a blank is used to separate the two T's.

### 2.12 Nihilist Substitution cipher

The nihilist substitution cipher bears the name of the anarchistic opponents of the czarist regime, who may have invented it [Kah96, p. 620]. It uses a Polybius square and a keyword as a key. Originally, the Polybius square used was a $6 \times 6$ square, to encompass all the old Russian characters.

The cipher works as follows. Each character of the plaintext is paired with a character of the keyword, in order. When the keyword is used up it cycles back to the start of the keyword. Then, the coordinates of each pair of characters are added, and used to substitute the plaintext character. Since the cipher only uses pairs of digits to substitute, should the addition result be between 100 and 110 , the first digit is omitted in the ciphertext. To avoid any risk of ambiguity, the coordinates should go from 1 to 5 . This way, any number less than 11 can be identified as a 3-digit number whose first digit was ignored.

Figure 2.13 shows an example of the ciphering process. The rows in the example are, respectively, the keyword, the plaintext, the keyword letters coordinates, the plaintext letters coordinates and the ciphertext.

Some variations of the cipher include:

- Inversion of coordinates - instead of pairs of (row, column) the cipher uses (column, row);
- Different coordinate representations - the digits from 1 to 5 can be substituted with other characters.

|  | 1 | 2 | 3 | 4 | 5 |
| :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | S | I | M | P | L |
| 2 | E | A | B | C | D |
| 3 | F | G | H | K | N |
| 4 | O | Q | R | T | U |
| 5 | V | W | X | Y | Z |


| Plaintext: <br> Keyword: |  | T | H | E | E | A | R | L | Y | B | I | R | D |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  |  | E | A | S | Y | E | A | S | Y | E | A | S | Y |
|  |  | 44 | 33 | 21 | 21 | 22 | 43 | 15 | 54 | 23 | 12 | 43 | 25 |
|  | $+$ | 21 | 22 | 11 | 54 | 21 | 22 | 11 | 54 | 21 | 22 | 11 | 54 |
| Ciphertext: |  | 65 | 55 | 32 | 75 | 43 | 65 | 26 | 08 | 44 | 34 | 54 | 79 |

Figure 2.13: Nihilist substitution cipher example.

### 2.13 Nihilist Transposition cipher

This nihilist transposition cipher works as follows. First, a square is filled with the plaintext, usually up to a hundred characters. If the text length is not a square number, blanks are used to fill in the rest of the square. The rows and columns are numbered in ascending order, from 1 to the length of the square side. The cipher key is a permutation of this sequence of numbers. For a $4 \times 4$ square, a possible key would be 2-1-4-3, for example. In a first phase, the order of the columns is switched in order to match the key, then, in a second phase, the same is done for the rows. The final square contains the ciphertext.

|  | 1 | 2 | 3 | 4 |
| :---: | :---: | :---: | :---: | :---: |
| 1 | L | E | N | D |
| 2 | M | E | Y | O |
| 3 | U | R | E | A |
| 4 | R | S | X | X |


|  | 2 | 1 | 4 | 3 |
| :---: | :---: | :---: | :---: | :---: |
| 1 | E | L | D | N |
| 2 | E | M | O | Y |
| 3 | R | U | A | E |
| 4 | S | R | X | X |


|  | 2 | 1 | 4 | 3 |
| :---: | :---: | :---: | :---: | :---: |
| 2 | E | M | O | Y |
| 1 | E | L | D | N |
| 4 | S | R | X | X |
| 3 | R | U | A | E |

Figure 2.14: Nihilist transposition cipher example.

The process can also be done the other way around: the rows and columns are ordered using the key and then switched to match the ascending order. An example of the ciphering process can be seen in Figure 2.14, where the key is 2-1-4-3 is used with the plaintext "lend me your ears".

Other variants of the cipher exist. For example, in Gaines' book [Gai89] a variant is described where the text is partitioned to fit in multiple squares, that are then ciphered using the same
steps [Gai89, p. 17-24]. In this book it is also described the possibility of using a keyword instead of a number sequence as key. Here, the letters of the keyword are numbered according to their order of appearance in the alphabet, resulting in a number sequence, that is then used in a similar way as described above.

## Chapter 3

## Heuristics

Ciphers are known to produce certain properties in their cryptograms that allow cipher identification. Thus, the fewer properties produced, and the less notable those produced, the better the cipher. Some features specific to a language are called language signatures. Once a text is ciphered, some of these features may remain evident depending on the cipher. We consider the inability to hide the language signature a trait of the cipher as well.

To determine which cipher was used in a given cryptogram, we use heuristics. A heuristic is an approach to a problem that follows a procedure, but, unlike an algorithm, an optimal solution is not guaranteed. Instead, a heuristic finds a reasonably working solution not guaranteed to be correct. In the context of cipher identification, heuristics are used to detect properties in cryptograms. Since there can be false positives and false negatives, heuristics here are employed as educated guesses as to what cipher was used. When compiling the results of different heuristics, it is possible to form an idea of what cipher may have been used. This is the underlying idea of the cipher classifier, on which this work is focused.

We use the following notation throughout this work.
$C$ - The set of all ciphers; a cipher in $C$ is denoted $c$.
$x-\mathrm{A}$ cryptogram.
$P$ - The set of all plaintexts; a plaintext in $P$ is denoted $p$.
$\Sigma$ - The input alphabet of all ciphers; every plaintext is in $\Sigma^{*}$.
$\Gamma$ - The output alphabet of all ciphers; every cryptogram is in $\Gamma^{*}$.

Let $N$ be the set of the heuristics' names and let $n$ be a name in $N$. For each heuristic there is a function $t_{n}(x)$, that attempts to recognize a given property in the cryptogram $x$, outputting a threshold value $v$. Let $m$ be a function that, given a name of a heuristic $n$ and a value $v$,
returns a set of ciphers $C^{\prime}$ known to produce the property the heuristic tries to detect, as follows:

$$
\begin{align*}
m: N \times \mathbb{R} & \longrightarrow 2^{C}  \tag{3.1}\\
n, v & \longrightarrow C^{\prime}
\end{align*}
$$

We define $h$, a function that encapsulates all heuristics, as follows. The function takes as input the name of the heuristic and a cryptogram, and outputs a pair of elements. The first element of this output pair is a value that is related to the probability that the property was produced by a subset of ciphers, and the second element is that subset.

$$
\begin{align*}
h: N \times \Gamma^{*} & \longrightarrow[0,1] \times 2^{C} \\
(n, x) & \longrightarrow\left(\operatorname{Pr}\left[\exists c \in m\left(n, t_{n}(x)\right), \exists p \in P: x=c(p)\right], m\left(n, t_{n}(x)\right)\right) \tag{3.2}
\end{align*}
$$

Here, we assume that no two ciphers create the same cryptogram with different or equal plaintexts.
In the future, we will use $h_{n}(x)$ to designate $h(n, x)$. Moreover, we will use $H$ to refer to the set of all heuristics, that is, $H=\left\{h_{n} \mid n \in N\right\}$.

In this chapter, we formulate heuristics, relying on observations made by finding patterns on batches of cryptograms. To generate each batch of cryptograms, a batch of texts was sampled from an English corpus of size 12 MB . Then, each text was ciphered using the ciphers presented in the previous chapter.

The sampled texts only contained characters belonging to the English alphabet. At this point, it is important to recall the difference in terminology between characters and symbols, previously established in Section 2.1. For ciphers using a Polybius square, the letter "i" was used instead of the letter " j ".

Unless explicitly stated otherwise, each plaintext is 500 characters long. The alphabet permutation, the Tabula Rectas, and the Polybius squares are random. For ciphers that require a keyword, the keyword length ranges from 5 to 20 characters, and it is randomly generated. For ciphers that have a period or a shift, it ranges from 5 to 20 .

For the chequerboard cipher, only the last variant was used, and for the nihilist cipher, we use the variant presented. For the Playfair cipher, we used "X" as the blank character.

### 3.1 Character frequency

There are many strategies in cryptoanalysis that rely on counting the characters of cryptograms. Assuming that one knows the original language in which the plaintext was written, one can use the character frequency of a cryptogram to infer which cipher may have been used.

An attacker does not know a priori how many symbols constitute each character, in order to count the characters. Assuming that each character is composed of only one symbol is a good start, especially since that is the usual for most ciphers.

### 3.1.1 Alphabet size of the cryptogram

The size of a cryptogram alphabet can be used to infer facts about its cipher. To denote the set of characters in a cryptogram $x$, we use the notation $\alpha(x)$ throughout this work.

A cipher like the numbered key, whose codomain is the set of all digits, can easily be distinguished from the others, even if the digits are disguised as letters, since each character is composed of only one symbol, then no more than 10 different characters are used. Ciphers that employ a Polybius square are limited to 25 different characters. Here, if the cryptogram has more than 25 different characters, one can exclude these ciphers from having been used.

We devised an heuristic following this observation. We began by analysing how the ciphers were distributed regarding the size of the cryptograms' alphabet. For this, a set of 60000 cryptograms was used and the alphabet size of each cryptogram was calculated. In addition to this set, we also computed a ten more to ensure that the results remained consistent. The graph of the distribution for each cipher is shown in Figure 3.1. The graph is a box-and-whisker plot, and an explanation for how these graphs work can be found in the Appendix A.


Figure 3.1: Box-and-whisker plots of the distribution of the cryptograms alphabet size of each cipher.

To build an effective heuristic, each heuristic was separated into branches, each corresponding to an interval of threshold values and a set of ciphers. The underlying idea is that if a value is within a given interval, then the cryptogram used to produce the value was likely produced with a cipher within the associated set of ciphers. Both the intervals and their associated sets of ciphers were chosen taking into account the distribution of values for each cipher, presented in Figure 3.1.

Table 3.1 shows the intervals and sets of ciphers chosen for this heuristic, along with a ratio of how many values within each interval were produced with a cipher of the corresponding set. Taking the case $\alpha(x) \leq 9$, for example, we see that approximately $99 \%$ of the alphabet size
values within that interval were produced from nihilist substitution cryptograms. We can think of the ratio as the accuracy of the chosen intervals and sets of ciphers.

Table 3.1: Performance measure of $h_{\alpha}$ branches.
$\left.\left.\begin{array}{c|c|c}\text { Ratio } & \text { Interval } & \text { Set of ciphers } \\ \hline 0.987879 & \alpha(x) \leq 9 & \text { \{Nihilist substitution\} } \\ 1.000000 & \alpha(x)=10 & \text { \{Numbered key, Nihilist substitution\} } \\ 0.996214 & \alpha(x)=20 \\ 0.988422 & 21 \leq \alpha(x)<24 & \text { \{Chequerboard\}} \\ 0.990430 & \alpha(x)=24 & \left\{\begin{array}{c}\text { \{Caesar, Nihilist transposition\} }\end{array}\right. \\ 0.999280 & \alpha(x)=25 & \left\{\begin{array}{c}\text { Nihilist transposition, Phillips } \\ \text { Caesar, Playfair }\end{array}\right\} \\ 0.965222 & \alpha(x)=26 & \{(x)=27\end{array}\right\} \begin{array}{c}\text { Nihilist transposition, Phillips, } \\ \text { Caesar, Playfair, Bifid }\end{array}\right\}$

In the previous table the value ranges of $1-7$ and 11-19 were ignored, since there were very few cryptograms within these ranges. Since the ratios are used in the heuristic we are about to specify, we can also think of them as the performance measurement of each branch of the heuristic. The heuristic follows:

$$
\begin{aligned}
& h_{\alpha}: \Gamma^{*} \longrightarrow {[0,1] \times 2^{C} } \\
& \begin{cases}(0.98,\{\text { Nihilist substitution }\}), & \text { if } \alpha(x) \leq 9 \\
(1.00,\{\text { Numbered key, Nihilist substitution }\}), & \text { if } \alpha(x)=10 \\
(1.00,\{\text { Chequerboard }\}), & \text { if } \alpha(x)=20 \\
(1.00,\{\text { Caesar, Nihilist transposition }\}), & \text { if } 21 \leq \alpha(x)<24 \\
\left(0.99,\left\{\begin{array}{c}
\text { Nihilist transposition, Phillips } \\
(\text { Caesar, Playfair }
\end{array}\right\}\right), & \text { if } \alpha(x)=24 \\
\left(1.00,\left\{\begin{array}{c}
\text { Nihilist transposition, Phillips, } \\
(\text { Caesar, Playfair, Bifid }
\end{array}\right\}\right), & \text { if } \alpha(x)=25 \\
(0.96,\{\text { Vigenere, Input autokey, Output autokey }\}), & \text { if } \alpha(x)=26 \\
(1.00,\{\text { Trifid }\}), & \text { if } \alpha(x)=27 \\
(0.00, \emptyset), & \text { otherwise. }\end{cases}
\end{aligned}
$$

It is important to note that this heuristic is only fit to be used in cryptograms produced with texts of size approximately 500 or larger. This is because, for some ciphers, the least frequent characters are so rare that, for smaller cryptograms, they may not show at all. It is easy to see how this would affect an heuristic that relies on the size of the cryptograms' alphabet.

As an example, Figure 3.2 shows the distributions of the size of the cryptograms' alphabet for a sample of 60000 cryptograms, each produced with a text of 100 characters. We can see that,
for almost all ciphers, the distribution range is larger, making the task of identifying ciphers much harder one.


Figure 3.2: Distribution of alphabet size for cryptograms with only 100 characters.

### 3.1.2 Index of coincidence

The index of coincidence is the probability of drawing two equal letters by randomly selecting two letters from a given text [MVOV97, p. 249]. This technique was first published by William F. Friedman.

The index of coincidence, or IC, allows one to deduce the original language of a monoalphabetic substitution cryptogram without deciphering it [Bau02, p. 302]. This is possible since every language has a characteristic index of coincidence value, part of the language signature, that is not hidden by a monoalphabetic substitution. The same applies to transposition ciphers, since the characters are moved around, but their frequency remain the same. On the other hand, polyalphabetic ciphers are known to hide the language signature more effectively, giving the text a "random semblance" that hinders statistical analysis. Thus, the IC can be used to tell ciphers apart, most notably monoalphabetic and transposition ciphers from polialphabetic ciphers.

The IC is calculated as follows. Let $a$ represent a letter of the cryptogram alphabet and let $|x|_{a}$ denote its absolute frequency in the cryptogram $x$. The probability of choosing character $a$ from ciphertext $x$ is $\frac{|x|_{a}}{|x|}$. The probability of picking it twice at different positions is $\frac{|x|_{a}\left(|x|_{a}-1\right)}{|x|(|x|-1)}$. Thus, the index of coincidence is given by:

$$
\begin{equation*}
I C(x)=\sum_{a \in \alpha(x)} \frac{|x|_{a}\left(|x|_{a}-1\right)}{|x|(|x|-1)} \approx \sum_{a \in \alpha(x)}\left(\frac{|x|_{a}}{|x|}\right)^{2} \tag{3.3}
\end{equation*}
$$

To observe the difference of IC between the ciphers we use in this thesis, a set of 60000 cryptograms was used, and each cryptogram IC was measured. With the calculated data, shown
in Figure 3.3, it is possible to see the variation of the IC for each cipher. In addition to this batch, the data was calculated for more batches to ensure that the results remained consistent.


Figure 3.3: Index of coincidence distribution of each cipher.

Using the observed data, the following branches of the heuristic were made, shown in Table 3.2. The table was computed similarly to Table 3.1. The reason why there is a $10 \%$ drop in performance

Table 3.2: Performance measure of $h_{I C}$ branches.

| Ratio | Interval | Set of ciphers |
| :---: | :---: | :---: |
| 0.999331 | $0 \leq I C(x)<0.055$ | $\left\{\begin{array}{c}\text { Output autokey, Trifid, Input autokey, } \\ \text { Vigenère, Bifid, Playfair } \\ \text { Phillips, Chequerboard }\end{array}\right\}$ |
| 0.899418 | $0.055 \leq I C(x)<0.11$ | \{Chequerboard, Nihilist transposition, Caesar\} <br> 0.997904 |
| $0.11 \leq I C(x)$ | \{Nihilist substitution, Numbered key\} |  |

for the second interval is that there are many outliers from the ciphers of the first set that end up within the second interval. This is also the reason why we did not create more intervals. Figure 3.4, shows the same data as Figure 3.3, but the outliers are included.

The chequerboard cipher is the only cipher featured in two different sets of the heuristic. This is because its distribution is too close to 0.055 , and by including it in only one of the sets, the heuristic would miss a lot for that cipher cryptograms. This way, it is still possible to tell the cipher apart from the last set of ciphers without making any compromise regarding the first two sets.

The ratios presented were the best performance that we achieved by tuning the heuristic


Figure 3.4: Index of coincidence distribution of each cipher, showing the outliers.
intervals and sets. The heuristic is as follows:

$$
\begin{aligned}
& h_{I C}: \quad \Gamma^{*} \longrightarrow[0,1] \times 2^{C} \\
& \left.x \longmapsto\left\{\begin{array}{ll}
\left(1.00,\left\{\begin{array}{c}
\text { Output autokey, Trifid, } \\
\text { Input autokey, Vigenère, } \\
\text { Bifid, Playfair, } \\
\text { Phillips, Chequerboard }
\end{array}\right.\right.
\end{array}\right\}\right), \quad \text { if } 0 \leq \operatorname{IC}(x)<0.055 \\
& \text { (1.00, }\{\text { Nihilist substitution, Numbered key\}), if } 0.11 \leq I C(x) \text {. }
\end{aligned}
$$

### 3.1.3 Detecting transpositions with character frequency

The relative frequency of the characters is a characteristic of the language. If a cryptogram character frequency is identical to that of the language, then the most likely cipher to have been used is a nihilist transposition cipher, since this cipher does not hide the character frequency of the language at all. Let $f_{x}(a)$ be the relative frequency of the character $a$ in the cryptogram $x$ and $f_{l}(a)$ be the relative frequency of the character $a$ in the language $l$. Let $\alpha(l)$ designate the alphabet of the language $l$. The following function $f$ calculates the mean of the squared difference of the relative frequency of both language and cryptogram, in an attempt at measuring
how much a cryptogram looks like a transposition.

$$
\begin{aligned}
f: \quad \Gamma^{*} & \longrightarrow[0,1] \\
x & \longmapsto 1-\sum_{a \in \alpha(l)} \frac{\left|f_{x}(a)-f_{l}(a)\right|^{2}}{|\alpha(l)|} .
\end{aligned}
$$

In the function, the square is used to make it easier to tell apart the values of the nihilisttransposition from the values of the other ciphers. Depending on the ciphers and languages considered, one may need to increase this power, to make the difference of the function distribution between ciphers more evident. We found the square to be sufficient in our case.


Figure 3.5: Value distribution of function $f$ for each cipher.

The function was applied to a batch of 60000 cryptograms, and the result distribution is shown in figure 3.5. In addition to this, other batches were used to ensure that the data remained consistent. Similarly to what was done for Table 3.1, Table 3.3 shows a possible separation of the data into intervals and the effectiveness of this separation.

Table 3.3: Performance measurement of $h_{\text {trans }}$ branches.

| Ratio | Interval | Set of ciphers |
| :---: | :---: | :---: |
| 0.993372 | $0 \leq f<0.99955$ | $C \backslash\left\{\begin{array}{c}\text { Nihilist transposition, Numbered key, } \\ \text { Nihilist substitution }\end{array}\right\}$ |
| 0.991561 | $0.99955 \leq f<1$ | \{Nihilist transposition\} |
| 1.000000 | $f=1$ | \{Numbered key, Nihilist substitution\} |

From this the heuristic follows:

$$
\begin{aligned}
h_{\text {trans }}: \Gamma^{*} \longrightarrow \begin{cases}(0,1] \times 2^{C} \\
\left(0.99, C \backslash\left\{\begin{array}{c}
\text { Nihilist transposition, } \\
\text { Numbered key, } \\
\text { Nihilist substitution }
\end{array}\right\}\right), & \text { if } 0 \leq f<0.99955 \\
(0.99,\{\text { Nihilist transposition\} }), & \text { if } 0.99955 \leq f<1 \\
(1.00,\{\text { Numbered key, Nihilist substitution\} }), & \text { if } f=1 .\end{cases}
\end{aligned}
$$

Before settling with function $f$, we also attempted to use the following function $f^{\prime}$ instead:

$$
\begin{aligned}
f^{\prime}: \Gamma^{*} & \longrightarrow[0,1] \\
x & \longmapsto 1-\left(\frac{\sum_{a \in \alpha(l)} g(x, l, a)}{|\alpha(l)|}\right)^{2}, \text { where } g(x, l, a)= \begin{cases}\frac{f_{x}(a)}{f_{l}(a)} & : f_{l}(a) \geq f_{x}(a) \\
\frac{f_{l}(a)}{f_{x}(a)} & : f_{l}(a)<f_{x}(a) .\end{cases}
\end{aligned}
$$

This function calculates the mean of function $g$ for all characters of the language. Function $g$ estimates how similar the frequency of a character of the cryptogram and the frequency of the same character in the language are. The mean is squared so that the value of the function, when applied to a transposition, is further apart from the value when applied to other ciphers. We did not settle with this version since it did a poorer job at separating the nihilist-transposition cryptograms from the others, which was our main goal. Figure 3.6 shows, for the same batch of cryptograms, the distribution of the resulting values of $f^{\prime}$.


Figure 3.6: Value distribution of the alternative function $f^{\prime}$ for each cipher.

### 3.2 Discovering periodicity with the Index of Coincidence

Discovering if a cipher is periodic is essential towards identifying it. Some periodic ciphers produce patterns on cryptograms that, once identified, can narrow down the group of ciphers that may have produced them. One such cipher is the Vigenère cipher, on which we focus on throughout this section.

Let $x$ be a Vigenère cryptogram of size $|x|$. Let:

$$
x=a_{0} a_{1} \ldots a_{|x|-1} \text {, where } a_{i} \text { is the } i \text { th ciphered character. }
$$

A Vigenère with keyword of size $\ell$, applies up to $\ell$ different Caesar cipher substitutions, each to a different plaintext substring. The $i$ th substring corresponds to the part of the text ciphered with the $i$ th keyword character, that is, all characters $i$ modulo $\ell$. One can visualise each substring as the characters of a column, if the text was separated into $\ell$ columns, as follows.

$$
x=\begin{array}{c|c|c|cc}
a_{0} & a_{1} & \ldots & a_{\ell-1} & \\
a_{\ell} & a_{\ell+1} & \ldots & a_{2 \ell-1} & \\
\vdots & \vdots & & \vdots & , k \in \mathbb{N} \\
a_{(k-1) \ell} & a_{(k-1) \ell+1} & \ldots & a_{k \ell-1} &
\end{array}
$$

The cipher is periodic since two characters at distance $\ell$ are ciphered using the same character of the key. Since the cipher is a polyalphabetic substitution cipher, its index of coincidence is generally considerably lower than that of the original plaintext. However, if the cryptogram is long enough, the index of coincidence of each substring should remain approximately the same value of that of the original language. We denote the cryptogram restricted to the characters in a given set of positions as $x[s e t]$. Let $\pi$ be a given period. The mean of the IC values of all columns is computed as follows:

$$
\begin{aligned}
I C_{\text {cols }}: \Gamma^{*} \times \mathbb{N} & \longrightarrow \mathbb{Q}_{0}^{+} \\
(x, \pi) & \longrightarrow \sum_{j=0}^{\pi-1} \frac{I C(x[\{i: i \equiv j(\bmod \pi) \wedge i \in\{0, \ldots,|x|-1\}\}])}{\pi} .
\end{aligned}
$$

Assuming $x$ is a large enough cryptogram, if one were to calculate the value of $I C_{\text {cols }}$ for multiple periods, the maximum value should occur for $\ell$ or one of its multiples. This would occur since, for the correct period (or one of its multiples), the ICs summed in $I C_{\text {cols }}$ are calculated over substrings ciphered with only one key each. Furthermore, the maximum value would be close to that of the plaintext IC. Thus, we have that:

$$
\begin{equation*}
\max \left(\left\{I C_{\text {cols }}(x, \pi): \pi \in\left\{\pi_{\min }, \ldots, \pi_{\max }\right\}\right\}\right) \in\left\{I C_{\text {cols }}(x, \ell \times k): k \in \mathbb{N}\right\} \tag{3.4}
\end{equation*}
$$

where $\pi_{\text {min }}$ and $\pi_{\text {max }}$ are the minimum and maximum periods one considers, respectively. The larger the value of $\pi$, the more substrings the ciphertext is divided into. Since the IC generally increases with fewer characters, the larger the value of $\pi$, the less useful the resulting value of $I C_{\text {cols }}(x, \pi)$.


Figure 3.7: Outcome of the period analysis using IC for a Vigenère cryptogram of key size 7.

Periodicity can be seen if one applies $I C_{\text {cols }}$ to every possible period of a Vigenère cryptogram. This is shown in figure 3.7, where a Vigenère cryptogram of size 150, produced with a key of size 7, was used. In the figure, the peaks for each period are actually considerably higher than the IC value of the original plaintext. This is because the cryptogram is short, meaning that each substring is also short, which, as previously explained, increases the value of the IC. In spite of this, as long as the cryptogram is not too short, this analysis works to show the periodicity. Although $I C_{\text {cols }}$ does not always produce as clear results as the one seen in Figure 3.7, one can expect it to work for most cryptograms.

Periodicity can be observed with these results, but we still need a reliable and automatic way to tell if these results are periodic to model a heuristic after this observation.

### 3.2.1 Automatically detect periodicity

We begin by finding the likelihood that a cryptogram has a given period $\pi$, using the values produced by $I C_{\text {cols }}$ for every possible period, calculated as before. The following steps are taken.

1. A list of IC values, one for each period, is computed for cryptogram $x$. Let $\operatorname{list}_{I C}(x)$ designate the resulting list of values, we have that:

$$
\operatorname{list}_{I C}(x)=\left[I C_{\text {cols }}(x, \pi): \pi \in\left\{\pi_{\min }, \ldots, \pi_{\max }\right\}\right]
$$

2. Given a list and an integer $l$, we separate the list into windows of size $l$ using the following function:

$$
\text { windows }(l i s t, l)=\left\{l i s t[i, \ldots, i+l]: i \in\left\{a \times l: a \in\left\{0, \ldots,\left\lfloor\frac{\mid \text { list } \mid}{l}\right\rfloor\right\}\right\}\right\} .
$$

This function is used to compute windows $\left(\operatorname{list}_{I C}(x), \pi\right)$. To make things easier, we consider the period of size 1 to have an index of coincidence of 0 . Should one want to consider a minimum possible period, the values up to that period are assumed to be 0 as well.
3. The maximum value of each window is computed.
4. Let index be a function that, given a number and a window, returns all indices of the window with that number, with the indices starting at 0 . Moreover, let $\operatorname{win}(x, \pi)$ designate the set of windows resultant of calculating windows $^{\left(\text {iist }_{I C}(x), \pi\right)}$.
The following function likely serves to calculate the likelihood that a given period was used in a given cryptogram. In the function, function index is used to find if at least one of the maximums is in the last position, that is, the period. In our work, we borrow the notation from [GKP94, p. 24]: if a condition enclosed in brackets is true, then the result is 1 , otherwise it is 0 .
The likelihood that period $\pi$ was used in cryptogram $x$, likely $(x, \pi)$, is defined as:

$$
\begin{align*}
& \text { likely: } \quad \Gamma^{*} \times \mathbb{N} \longrightarrow \mathbb{Q}_{0}^{+} \\
& x, \pi \quad \longrightarrow \frac{1}{|\operatorname{win}(x, \pi)|} \times \sum_{w \in \operatorname{win}(x, \pi)}[\pi-1 \in \operatorname{index}(\max (w), w)] . \tag{3.5}
\end{align*}
$$

These steps can be repeated for every possible period to find the most likely period. We define the likelihood that a given cryptogram is periodic as equal to the likelihood of the most likely period. The size of the period is inversely proportional to the number of windows, which results in two scenarios:

1. For shorter periods the windows are smaller, and as a consequence also in a larger quantity, which in turn increases the chance of finding maximums at positions not multiple of the period. To prevent this unbalance for shorter periods, a minimum period $\pi_{\text {min }}$ is established.
2. For larger periods, there are fewer windows, meaning there is less scrutiny and less granularity in the resulting likelihood. A larger period needs fewer maximums at the right positions to look very likely. On the other hand, each maximum holds more weight, thus, a missed maximum results in a larger loss in likelihood. To avoid this, we define a likelihood threshold $l t$, and preference is given to shorter periods that meet this threshold.

Following this observation, function $\phi_{\text {likely }}$ is designed to calculate the likelihood that a given cryptogram shows periodicity when $I C_{\text {cols }}$ is applied:

$$
\begin{aligned}
\phi_{\text {likely }}: & \Gamma^{*} \longrightarrow[0,1] \\
& x \longrightarrow\left\{\begin{array}{cl}
\operatorname{likely}(x, \min (\{\pi: \operatorname{likely}(x, \pi) \geq l t\})), & \text { if }|\{\pi: \operatorname{likely}(x, \pi) \geq l t\}|>0 \\
0, & \text { otherwise },
\end{array}\right.
\end{aligned}
$$

with $\pi \in\left\{\pi_{\text {min }}, \ldots,\left\lfloor\frac{\left\lfloor l i s t_{I C}(x) \mid\right.}{2}\right\rfloor\right\}$. Note that $\phi_{\text {likely }}$ only aims to find the most likely period up to $\left\lfloor\frac{|l i s t|}{2}\right\rfloor$. This is because at least two windows are necessary to consider a given period. Thus,
if one wants to consider $\pi_{\max }$ as the highest possible period, one needs to calculate $\operatorname{list}_{I C}(x)$ for $\pi \in\left\{\pi_{\min }, \ldots, 2 \times \pi_{\max }\right\}$ at least.

Figure 3.8 shows the results of the application of function $\phi_{\text {likely }}$ to a set of 12000 cryptograms. Here, $l t=\frac{1}{2}$ was used, and one can see that it was good enough to separate most ciphers into two groups. A higher value of $l t$ worsens the results, meaning that it becomes harder to tell ciphers apart, and that is why we settled for a lower value.


Figure 3.8: Distribution of function $\phi_{\text {likely }}$ for each cipher.

For many ciphers there are outliers at 0 , which is to be expected given that if the function does not find any periodicity it returns 0 . After observing the results, and since these remain consistent between different sets of cryptograms, we defined intervals and corresponding sets of ciphers, and measured the performance of identification. For this purpose, a threshold of 0.7 was used, as seen in Table 3.4.

Table 3.4: Performance measurement of $h_{\text {ICperiod }}$ branches.

| Ratio | Interval | Set of ciphers |
| :---: | :---: | :---: |
| 0.943286 | $\phi_{\text {likely }}(x)<0.7$ | $\left\{\begin{array}{c}\text { Nihilist transposition, Caesar, } \\ \text { Input autokey, Output autokey, } \\ \text { Phillips, Numbered key, Chequerboard, } \\ \text { Playfair, Bifid, Trifid, Nihilist } \\ \text { substitution, Vigenère }\end{array}\right\}$ |
| 0.557166 | $0.7 \leq \phi_{\text {likely }}(x)$ | $\left\{\begin{array}{c}\text { Bifid, Trifid, Nihilist substitution } \\ \text { Vigenère }\end{array}\right\}$ |

Note that both Bifid and Trifid are included in both intervals, since these ciphers are hard to distinguish from the others with the chosen threshold. In this way, the ciphers are not ignored,
but no information is given about them either. From this, the heuristic follows:

$$
h_{\text {ICperiod }}: \quad \Gamma^{*} \longrightarrow[0,1] \times 2^{C}
$$

For the same batch used to specify the heuristic, for each computed $\phi_{\text {likely }}$, we also examined the most likely periods. We assume that, if $\phi_{\text {likely }}(x)=0$, then the cipher is not periodic. Table 3.5 shows the percentage, for each cipher, of correctly guessed periods. The number of correctly guessed non-periodic ciphers is also counted. In the table, one sees that there is a correlation between the highest likelihoods and the number of correctly guessed periods. We can also see that this strategy, by itself, is not very good at guessing which cryptograms do not have any period. Looking at the table, we can see that with this strategy one can guess the period of

Table 3.5: Fraction of correctly guessed periods using the underlying strategy of $\phi_{\text {likely }}$.

| Cipher | Percentage |
| ---: | :--- |
| Vigenère | 0.444 |
| Nihilist substitution | 0.441 |
| Trifid | 0.413 |
| Bifid | 0.412 |
| Phillips | 0.226 |
| Input autokey | 0.152 |
| Output autokey | 0.149 |
| Nihilist transposition | 0.102 |
| Caesar | 0.051 |
| Checkerboard | 0.038 |
| Numbered key | 0.033 |
| Playfair | 0.025 |

both Trifid and Bifid ciphers almost as well as for the Vigenère. This was a surprising result for us, and it is because the strategy worked so well for these two ciphers that we were unable to put them in just one branch of the heuristic. Both ciphers hide the language signature much more effectively than the Vigenère cipher, and, although they do not have a key changing system like the autokey ciphers, we did not expect to find any relation between the index of coincidence and the period length for them.

### 3.3 Phillips signature

The Phillips cipher is a periodic cipher, and although it is somewhat similar to the Vigenère cipher, the previous approach is not effective against Phillips cryptograms. In this section, we explain the reason for this, and how a similar approach (in the sense that we look at the periodicity in a similar manner) can be used to tackle this specific cipher.

### 3.3.1 Finding the period

Since for the Vigenère cipher the IC approach to finding periodicity works as expected, we start by drawing a comparison between the two ciphers. Both ciphers are made up of multiple monoalphabetic substitutions, each with a different key. In the Vigenère cipher, although each character has its own monoalphabetic substitution, this substitution is repeated every period length. On the other hand, in the Phillips cipher, all characters within a period length block are ciphered using the same monoalphabetic substitution. If a Vigenère cipher has a keyword of $\ell$ characters, then, at most, there are $\ell$ different monoalphabetic substitutions occurring in the text when the cipher is applied. On the other hand, in the Phillips cipher, there are at most 7 different monoalphabetic substitutions occurring in the text when the cipher is applied. This is because, even though there are eight keys in rotation, two of these are equivalent, as seen in Table 2.2.

Consider $c_{j}$ to be the monoalphabetic substitution with the $j$ th key, $a_{i}$ to be the ciphertext character at position $i$, and $a_{i}^{\prime}$ the character at the same position of the original plaintext. Let $x$ be a cryptogram of length $k \ell, k \in \mathbb{N}$. To calculate $I C_{c o l s}(x, \pi)$ for a given period $\pi$, the text is separated into substrings. Figure 3.9 shows the separation into columns for the Vigenère and Phillips ciphers, respectively.

The underlying idea of $I C_{\text {cols }}$ is that, for the correct period, only one key is used for each column. For the Phillips cipher, this does not happen at $\pi=\ell$, but at $\pi=8 \ell$. This may not be very useful, given that $8 \ell$ can be a large number and the cryptogram may not be large enough to see this. Another IC peak that can occur is at $\pi=4 \ell$. In this case, although most substrings are ciphered with more than one key, the first $\ell$ substrings are ciphered with only one, since the first and fifth squares are equivalent keys. Both behaviours are shown in Figure 3.10, where the function $I C_{\text {cols }}$ is applied to a cryptogram of size 1000, produced with period 5. Separating the

| $c_{0}\left(a_{0}^{\prime}\right)$ | $c_{1}\left(a_{1}^{\prime}\right)$ | $\cdots$ | $c_{\ell-1}\left(a_{\ell-1}^{\prime}\right)$ |
| :---: | :---: | :---: | :---: |
| $c_{0}\left(a_{\ell}^{\prime}\right)$ | $c_{1}\left(a_{\ell+1}^{\prime}\right)$ | $\cdots$ | $c_{\ell-1}\left(a_{2 \ell-1}^{\prime}\right)$ |
| $\vdots$ | $\vdots$ |  | $\vdots$ |
| $c_{0}\left(a_{(k-1) \ell}^{\prime}\right)$ | $c_{1}\left(a_{(k-1) \ell+1}^{\prime}\right)$ | $\cdots$ | $c_{\ell}\left(a_{k \ell-1}^{\prime}\right)$ |

```
    \(c_{(0 \bmod 8)}\left(a_{0}^{\prime}\right) \quad c_{(0 \bmod 8)}\left(a_{1}^{\prime}\right) \quad \cdots \quad c_{(0 \bmod 8)}\left(a_{\ell-1}^{\prime}\right)\)
    \(c_{(1 \bmod 8)}\left(a_{\ell}^{\prime}\right) \quad c_{(1 \bmod 8)}\left(a_{\ell+1}^{\prime}\right) \quad \ldots \quad c_{(1 \bmod 8)}\left(a_{2 \ell-1}^{\prime}\right)\)
\(c_{(\ell-1 \bmod 8)}\left(a_{(k-1) \ell}^{\prime}\right) \quad c_{(\ell-1 \bmod 8)}\left(a_{(k-1) \ell+1}^{\prime}\right) \quad \ldots \quad c_{(\ell-1 \bmod 8)}\left(a_{k \ell-1}^{\prime}\right)\)
```

Figure 3.9: Separation into columns for the Vigenère and for the Phillips ciphers, for $\pi=\ell$.


Figure 3.10: Resulting values of $I C_{\text {cols }}$ applied to a Phillips cryptogram of period 5 .
text into $8 \ell$ or $4 \ell$ columns leaves few characters per column. Therefore, the calculated IC of each substring is often too inaccurate to draw conclusions. Thus, for shorter cryptograms this observation is of little value, and a different, more precise approach is required to find the correct period.

### 3.3.2 A better approach to find the period

Separating the text into $\ell$ substrings does not separate characters ciphered with different keys in the Phillips cipher. However, it is possible to see in Figure 3.9 that for the $\ell$ columns, each row only has one key.

If one concatenates the ciphertext characters of each $k$ th row modulo eight, there are eight strings of text, each the result of a monoalphabetic substitution of a different key. Similarly to what was done before with $I C_{\text {cols }}$, a method to find the period in a cryptogram using the rows
instead follows:

where, $p r$ stands for "phillips rows". The fraction numerator in the expression is the measurement of the Index of coincidence of row $k$.

Figure 3.11 shows this function applied to the same cryptogram used in Figure 3.10. Although this strategy performs better than the previous one, it is not reliable enough for shorter cryptograms, similarly to $I C_{\text {cols }}$. Another problem with the function $I C_{p r}$ is that the IC


Figure 3.11: Application of $I C_{p r}$ to a Phillips cryptogram of period 5.
peaks on multiples of the period are not as noticeable as the ones produced by $I C_{\text {cols }}$, meaning that using function likely, from the previous section, is unfeasible in this context. Even though this is not a viable strategy to find the likelihood of periodicity, it is a viable strategy to find the correct period, since the peak at the period is usually very noticeable. An alternative way to calculate the likelihood of a given period for this cipher follows. We borrow the notation from [GKP94, p. 24] once again.

$$
\begin{aligned}
& \text { likely }_{\text {phillips }}: \Gamma^{*} \times \mathbb{N} \longrightarrow\left\{0, \frac{1}{8}, \frac{2}{8}, \frac{3}{8}, \frac{4}{8}, \frac{5}{8}, \frac{6}{8}, \frac{7}{8}, 1\right\} \\
&(x, \pi) \longrightarrow\left\{\begin{array}{cl}
0, & \text { if } \operatorname{IC}(x) \geq 0.065 \\
\sum_{k \in\{0, \ldots, \pi-1\}}[\text { condition }] \times \frac{1}{8}, & \text { otherwise, }
\end{array}\right.
\end{aligned}
$$

where, condition $=\left(\operatorname{IC}\left(x\left[\left\{i:\left\lfloor\frac{i}{\pi}\right\rfloor \bmod 8=k \wedge i \in\{0, \ldots,|x|\}\right\}\right]\right) \geq 0.065\right)$.
Function likelyphilips separates the text into 8 strings of text, each ciphered with one of the 8 keys, and then counts how many of these had an IC inferior to that of the language. Given that
the Phillips cipher is a polyalphabetic cipher, the function returns 0 if the IC, of the cryptogram as a whole, has an IC equal or greater to that of the language. The highest likelihood among all possible periods is used as the likelihood that the cryptogram was produced by a Phillips cipher. From this, we specify function $f$ as follows:

$$
\begin{aligned}
f: \Gamma^{*} & \longrightarrow\left\{0, \frac{1}{8}, \frac{2}{8}, \frac{3}{8}, \frac{4}{8}, \frac{5}{8}, \frac{6}{8}, \frac{7}{8}, 1\right\} \\
x & \longrightarrow \max \left(\left\{\text { likely }_{\text {phillips }}(x, \pi): \pi \in\left\{\pi_{\min }, \ldots, \pi_{\max }\right\}\right\}\right),
\end{aligned}
$$

where $\pi_{\min }$ and $\pi_{\max }$ are the minimum and maximum periods one wants to consider, respectively. Figure 3.12 shows the distribution of the results of $f$ for each cipher, calculated over a batch of 60000 cryptograms. The figure shows there is a gap between the Phillips cipher and the other ciphers. This analysis was done over multiple batches and remained consistent.


Figure 3.12: Distribution of results of function $f$.

Table 3.6 shows, similarly to what was done for the other heuristics, the branches we use in the following heuristic and the performance that is expected from each. These results remained

Table 3.6: Performance measurement of $h_{\text {phillips }}$ branches.
$\left.\begin{array}{c|c|c}\text { Ratio } & \text { Interval } & \text { Set of ciphers } \\ \hline 0.931587 & f(x)<0.12 & \left\{\begin{array}{c}\text { Numbered key, Nihilist substitution, Caesar, } \\ \text { Output autokey, Nihilist transposition, } \\ \text { Input autokey, Vigenère, Trifid }\end{array}\right\} \\ 0.599270 & 0.12 \leq f(x)<0.6 & \text { \{Bifid, Playfair, Chequerboard\} } \\ 0.893824 & f(x) \leq 1 & \text { \{Phillips\} }\end{array}\right\}$
consistent between multiple batches. Thus, instead of having a heuristic that only tells apart the Phillips cipher from the rest, we can create more branches to take advantage of the fact that
there are other ciphers that can also be told apart. The heuristic follows:

$$
\begin{aligned}
h_{\text {phillips }}: \Gamma^{*} \longrightarrow & \left.\left\{\begin{array}{ll}
(0,1] \times 2^{C} \\
x & \longrightarrow\left\{\begin{array}{c}
\text { Numbered key, Nihilist } \\
\text { substitution, Caesar, Output } \\
\text { autokey, Nihilist transposition, } \\
\text { Input autokey, Vigenère, Trifid }
\end{array}\right.
\end{array}\right\}\right), \\
\left(0.60,\left\{\begin{array}{l}
\text { Bifid, Playfair, } \\
\text { Chequerboard }
\end{array}\right\}\right), & \text { if } 0.12 \leq f(x)<0.12 \\
(0.89,\{\text { Phillips }\}), & \text { if } f(x) \leq 1 .
\end{aligned}
$$

### 3.4 Non-connected digraphs

In this section, we present the approach originally described in [MR07], to find the period. Since it was originally presented as a way to decrypt bifid cryptograms, we focus on this cipher before expanding to other ciphers. You may recall from Section 2.7 that the bifid ciphers characters by block, using the following scheme, shown in Table 3.7.

| plaintext character | $\sigma_{0}$ | $\sigma_{1}$ | $\sigma_{2}$ | $\ldots$ | $\sigma_{\ell-3}$ | $\sigma_{\ell-2}$ | $\sigma_{\ell-1}$ |
| ---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| first coordinate | $x_{0}$ | $x_{1}$ | $x_{2}$ | $\ldots$ | $x_{\ell-3}$ | $x_{\ell-2}$ | $x_{\ell-1}$ |
| second coordinate | $y_{0}$ | $y_{1}$ | $y_{2}$ | $\ldots$ | $y_{\ell-3}$ | $y_{\ell-2}$ | $y_{\ell-1}$ |

$\downarrow$

| ciphertext character | $\tau_{0}$ | $\tau_{1}$ | $\ldots$ | $\tau_{\frac{\ell-3}{2}}$ | $\tau_{\frac{\ell-1}{2}}$ | $\tau_{\frac{\ell+1}{2}}$ | $\ldots$ | $\tau_{\ell-1}$ |
| ---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| first coordinate | $x_{0}$ | $x_{2}$ | $\ldots$ | $x_{\ell-3}$ | $x_{\ell-1}$ | $y_{1}$ | $\ldots$ | $y_{\ell-2}$ |
| second coordinate | $x_{1}$ | $x_{3}$ | $\ldots$ | $x_{\ell-2}$ | $y_{0}$ | $y_{2}$ | $\ldots$ | $y_{\ell-1}$ |

Table 3.7: bifid ciphering scheme for the odd period on a block of size $\ell$.

The block size $\ell$ of a bifid cryptogram can be found by computing, for different distances $d$, the frequency at which pairs of equal characters occur at a distance $d$. The name given to two equal characters separated by a distance $d$ is a non-connected digraph. The plot of this as a function of $d$ should result in a graph of approximately sinusoidal shape, with peaks at $d=\ell \times k$ for $k \in \mathbb{N}$, as explained next. This occurs for both even and odd periods and what follows is an explanation for the odd case.

### 3.4.1 Enumerating all possible cases for non-connected digraphs

To facilitate the explanation, we start by counting all the possible cases of non-connected digraph occurrences, depending on the position of the two characters of the digraph within the ciphered block. Let B and A denote the segments Before and After the centre of the block, and C the centre itself. We differentiate between these segments since there are noticeable differences between them regarding the cryptogram:

- The segment Before is made up of only the first coordinates of the block characters;
- The segment Centre is made up of the last first coordinate and the first second coordinates of the block characters;
- The segment After is made up of only the second coordinates of the block characters.

Taking into account that we are only considering odd periods, the B section consists of the characters from index 0 to $\frac{\ell-1}{2}$, the Centre character is at $\frac{\ell-1}{2}$ and the A segment consists of the remainder characters up to $\ell$. We refer to segments A and B as the halves.

Let $j$ be a chosen index. We evaluate the probability that the character in position $j$ and the character in position $j+d$ are the same, where $d$ is the distance between these two characters. We separate this analysis into four different cases:

MM - When the two chosen positions are both at the centre of the blocks;
HM - When one position is in one of the halves and the other at the centre;
HH - When the two chosen positions are in the same half;
$\mathbf{H H}^{\prime}$ - When both indexes are in different halves.

For each, we first layout the constraints that both $j$ and $j+d$ have to fulfil so that we may count all possible positions for both characters. We then calculate, given a distance $d$, the probability of each case, $\mathcal{P}_{d}^{(X Y)}$, where $X Y \in\left\{M M, H M, H H, H H^{\prime}\right\}$. This probability depends only on $d$ modulo $\ell$. We do not consider the case $d=0$ since the distance is always greater than zero. On the other hand, we consider the case $d=\ell$ when calculating the probability, to cover the case where the two characters have the same position within the blocks. There are a total of $\ell$ cases: $j \in\{0,1, \ldots, \ell-1\}$, for $d \in\{d \in 1,2, \ldots, \ell\}$.

MM We begin with the case where the two chosen positions are at the centre of the blocks.
In this case, $j$ is at the centre and $j+d$ is exactly at the block size distance. In other words, $d=\ell$ with $j$ at $C$. Since:

$$
j=\frac{\ell-1}{2} \wedge j+d=\ell+\frac{\ell-1}{2} \Longrightarrow j=\frac{\ell-1}{2}
$$



Figure 3.13: Both $j$ and $j+d$ are both at the centre of different blocks.
the probability that two characters are the same at a distance $d$ for the $M M$ scenario is:

$$
P_{d}^{(M M)}= \begin{cases}0 & \text { if } d<\ell \\ \frac{1}{\ell} & \text { if } d=\ell .\end{cases}
$$

HM In this case, one of the characters is at the centre and the other in one of the other segments. Figure 3.14 shows four possible scenarios (BC, CA, CB and AC), which can be reduced to two cases: the case where the first letter is in one of the halves and the second at the centre and the case where the first letter is at the centre, and the second in one of the halves. Thus, we


Figure 3.14: One of the positions is at the centre and the other is in one of the halves.
have the following intervals for $j$ :

$$
\left(\frac{\ell-1}{2}<j<\ell+\frac{\ell-1}{2} \wedge j+d=\ell+\frac{\ell-1}{2}\right) \vee\left(j=\frac{\ell-1}{2} \wedge \frac{\ell-1}{2}<j+d<\ell+\frac{\ell-1}{2}\right),
$$

which, since the first part of the second parcel is within the first part of the first parcel, and since the second part of the first parcel is within the second part of the second parcel, can be simplified to:

$$
j+d=\ell+\frac{\ell-1}{2} \vee j=\frac{\ell-1}{2}
$$

Thus, we have that there are only two possible values for $j$. From this, the probability that one character is in one of the halves and the other is in the middle, at a distance $d$, follows:

$$
P_{d}^{(H M)}= \begin{cases}\frac{2}{\ell} & \text { if } d<\ell \\ 0 & \text { if } d=\ell .\end{cases}
$$

HH In this case, both characters are in the same segment. As shown in Figure 3.15, there are two possible cases. We may have two characters in segments within the same block or in two separate blocks, as long as the distance between both characters is less than $\ell$. From this we


Figure 3.15: Both characters of the digraph are in the same segment.
deduce the following.

- If $d \leq \frac{\ell-1}{2}$, we must have that:

$$
\left(0 \leq j<\frac{\ell-1}{2} \wedge j+d<\frac{\ell-1}{2}\right) \vee\left(\frac{\ell-1}{2}<j<\ell \wedge j+d<\ell\right) .
$$

By picking the elements, from each parcel, that most constrain $j$, one can reduce this to the following:

$$
\left(0 \leq j<\frac{\ell-1}{2}-d\right) \vee\left(\frac{\ell-1}{2}<j<\ell-d\right)
$$

Thus, the number of $j$ 's that satisfy the above conditions is:

$$
\left(\frac{\ell-1}{2}-d\right)+\left(l-d-\frac{\ell-1}{2}-1\right)=\ell-2 d-1 .
$$

- On the other hand, if $d>\frac{\ell-1}{2}$, we have the following:

$$
\left(0 \leq j<\frac{\ell-1}{2} \wedge j+d<\ell+\frac{\ell-1}{2}\right) \vee\left(\frac{\ell-1}{2}<j<\ell \wedge \ell+\frac{\ell-1}{2}<d+j<2 \ell\right)
$$

By picking the elements from each parcel that most constrain $j$, one can simplified this to:

$$
\left(\ell-d \leq j<\frac{\ell-3}{2}\right) \vee\left(\ell+\frac{\ell-1}{2}-d<j<\ell\right)
$$

From this, one can calculate the cases for which $j$ satisfies the conditions:

$$
\left(d-\frac{\ell-1}{2}-1\right)+\left(d-\frac{\ell-1}{2}-1\right)=-\ell+2 d-1
$$

From this analysis it follows that:

$$
\mathcal{P}_{d}^{(H H)}=\left\{\begin{array}{ll}
\frac{l-2 d-1}{\ell} & \text { if } d \leq \frac{\ell-1}{2} \\
\frac{-l+2 d-1}{\ell} & \text { if } \frac{\ell-1}{2}<d<\ell
\end{array}= \begin{cases}1-\frac{2 d}{\ell}-\frac{1}{\ell} & \text { if } d \leq \frac{\ell-1}{2} \\
-1+\frac{2 d}{\ell}-\frac{1}{\ell} & \text { if } \frac{\ell-1}{2}<d<\ell,\end{cases}\right.
$$

which we can be summarised as:

$$
\mathcal{P}_{d}^{(H H)}=\left|1-\frac{2 d}{\ell}\right|-\frac{1}{\ell} .
$$

HH' This last case is the one where the two characters are in different segments of the block. Either the first character is at $B$ and the second at $A$, or the first is at $A$ and the second at $B$. Figure 3.16 shows the two possible scenarios.


Figure 3.16: The two different scenarios for characters in both halves.

For this case, we have the following:

$$
\left(0 \leq j<\frac{\ell-1}{2} \wedge \frac{\ell-1}{2}<j+d<\ell\right) \vee\left(\frac{\ell-1}{2}<j<\ell \wedge \ell \leq j+d<\ell+\frac{\ell-1}{2}\right)
$$

As was done previously, one can simplify the expression to the most restrictive constraints on $j$. In order to find the number of values of $j$ that satisfy these conditions, we separate this into two cases, $d \leq \frac{\ell-1}{2}$ and $d>\frac{\ell-1}{2}$, yielding:

$$
\begin{cases}\left(\frac{\ell-1}{2}-d<j<\frac{\ell-1}{2}\right) \vee(\ell-d-1<j<\ell) & \text { if } d \leq \frac{\ell-1}{2} \\ (-1<j<\ell-d) \vee\left(\frac{\ell-1}{2}<j<\ell+\frac{\ell-1}{2}-d\right) & \text { if } d>\frac{\ell-1}{2}\end{cases}
$$

Therefore, the number of different $j$ 's is:

$$
\left\{\begin{array}{ll}
\left(\frac{\ell-1}{2}-\frac{\ell-1}{2}+d-1\right)+(\ell-\ell+d+1-1) & \text { if } d \leq \frac{\ell-1}{2} \\
(\ell-d+1-1)+\left(\ell+\frac{\ell-1}{2}-d-\frac{\ell-1}{2}-1\right) & \text { if } d>\frac{\ell-1}{2}
\end{array}= \begin{cases}2 d-1 & \text { if } d \leq \frac{\ell-1}{2} \\
2 \ell-2 d-1 & \text { if } d>\frac{\ell-1}{2}\end{cases}\right.
$$

The probability of finding the same character in two different segments can now be determined simply by dividing by the number of all possible positions by $\ell$ :

$$
\mathcal{P}_{d}^{\left(H H^{\prime}\right)}=\left\{\begin{array}{ll}
-\frac{1}{\ell}+\frac{2 d}{\ell} & \text { if } d \leq \frac{\ell-1}{2} \\
-\frac{1}{\ell}-\frac{2 d}{\ell} & \text { if } \frac{\ell-1}{2}<d<\ell \\
0 & \text { if } d=\ell
\end{array}= \begin{cases}1-\frac{1}{\ell}-\left|1-\frac{2 d}{\ell}\right| & \text { if } d<\ell \\
0 & \text { if } d=\ell\end{cases}\right.
$$

### 3.4.2 Probability of character occurrence within a ciphering block

To facilitate the following explanation, we introduce the following quantities. For $\alpha \in \Sigma$ let:
$\operatorname{row}(\boldsymbol{\alpha})$ denote the row of the key table to which $\alpha$ belongs to;
$\boldsymbol{\operatorname { c o l }}(\boldsymbol{\alpha})$ denote the column of the key table to which $\alpha$ belongs to;
$\boldsymbol{p}(\alpha)$ the probability of occurrence of $\alpha$ in the text;
$\boldsymbol{p}(\boldsymbol{\alpha} \boldsymbol{\beta})$ the probability of occurrence of the digraph $\alpha \beta$ in the text;
$\boldsymbol{\alpha}^{t}$ the transposed of $\alpha$, that is, the key-table entry that satisfies the condition:

$$
\operatorname{row}\left(\alpha^{t}\right)=\operatorname{col}(\alpha) \quad \wedge \quad \operatorname{col}\left(\alpha^{t}\right)=\operatorname{row}(\alpha) .
$$

We also use the following notation:

- $\rho_{\alpha}=\sum_{\beta \in \text { row( } \alpha)} p(\beta)$, the probability of a character in the same row (in the key-table) of $\alpha$ occurring in the text.
- $\kappa_{\alpha}=\sum_{\beta \in \operatorname{col}(\alpha)} p(\beta)$, the probability of a character in the same column as $\alpha$ (in the key-table) occurring in the text.
- $B_{\alpha}=\sum_{\beta \in \operatorname{row}(\alpha), \gamma \in \operatorname{row}\left(\alpha^{t}\right)} p(\beta \gamma)$, the probability of digraph $\beta \gamma$ occurring in the plaintext in such a way that, when ciphered, a character becomes $\alpha$ in the first part of the block (before the centre).
- $C_{\alpha}=\rho_{\alpha} \kappa_{\alpha}$, the probability of a digraph occurring in the plaintext in such a way that, when ciphered, a character becomes $\alpha$ at the middle of the block.
- $A_{\alpha}=\sum_{\beta \in \operatorname{col}(\alpha), \gamma \in \operatorname{col}\left(\alpha^{t}\right)} p(\beta \gamma)$, the probability of a digraph $\beta \gamma$ occurring in such a way that, when ciphered, a character becomes $\alpha$ at the last part of the block (after the centre).

Recalling from Table 3.7 the bifid ciphering scheme for odd periods, using the notation introduced, we can determine the probability that the $i$ th character of the ciphertext is equal to a particular character $\alpha \in \Sigma$. This probability depends on where character $\alpha$ occurs in the ciphertext block. As shown in Table 3.8, we have three cases to take into account. Let $0 \leq i, j<\frac{\ell-1}{2}$, we have:

- the case where $\tau_{i}=\alpha$, that is, the case where the character occurs before the centre;
- the case where $\tau_{\frac{\ell-1}{2}}=\alpha$, that is, the case where the character occurs at the centre;


Table 3.8: Block division into 3 parts.

| B |  |  | C | A |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| $\ldots$ | $\tau_{i}$ | $\ldots$ | $\tau_{\frac{\ell-1}{2}}$ | $\ldots$ | $\tau_{\frac{\ell+1}{2}+j}$ | $\cdots$ |
|  | $x_{2 i}$ | $\cdots$ | $x_{\ell-1}$ | $\cdots$ | $y_{2 j-1}$ | $\cdots$ |
|  | $x_{2 i+1}$ |  | $y_{0}$ |  | $y_{2 j}$ | $\ldots$ |

For the first case, we have the following:

$$
\begin{equation*}
\mathcal{P}\left(\tau_{i}=\alpha\right)=\mathcal{P}\left(\sigma_{2 i-1} \in \operatorname{row}(\alpha) \wedge \sigma_{2 i} \in \operatorname{row}\left(\alpha^{t}\right)\right)=B_{\alpha} . \tag{3.6}
\end{equation*}
$$

For the centre case, we have the following:

$$
\begin{equation*}
\mathcal{P}\left(\tau_{\frac{\ell-1}{2}}=\alpha\right)=\mathcal{P}\left(\sigma_{\ell} \in \operatorname{row}(\alpha) \wedge \sigma_{1} \in \operatorname{col}(\alpha)\right) \simeq \rho_{\alpha} \kappa_{\alpha}=C_{\alpha} . \tag{3.7}
\end{equation*}
$$

The probability that both a character of the same row of $\alpha$ and a character of the same column of $\alpha$ occur in the text in the two extremes of the block can be influenced by the language itself, and thus the two events may not be independent. This is why $\simeq$ is used instead of $=$ in 3.7.

For the last case, we have:

$$
\begin{equation*}
\mathcal{P}\left(\tau_{\frac{\ell-1}{2}+j}=\alpha\right)=\mathcal{P}\left(\sigma_{2 j} \in \operatorname{col}\left(\alpha^{t}\right) \wedge \sigma_{2 j+1} \in \operatorname{col}(\alpha)\right)=A_{\alpha} . \tag{3.8}
\end{equation*}
$$

### 3.4.3 Probability of homogeneous non-connected digraph occurrences

One can now calculate the probability that a non-connected digraph occurs in the ciphertext. Let $\mathcal{P}_{d}$ denote the probability that two characters at a distance $d$ are the same. We can express this with the quantities defined previously, $B_{\alpha}, C_{\alpha}$, and $A_{\alpha}$, as follows:

$$
\begin{align*}
\mathcal{P}_{d}=\sum_{\alpha}\left(B_{\alpha}^{2}+A_{\alpha}^{2}\right) \cdot \mathcal{P}_{d}^{(H H)}+\sum_{\alpha} & C_{\alpha}^{2} \cdot \mathcal{P}_{d}^{(M M)} \\
& +\sum_{\alpha} C_{\alpha}\left(B_{\alpha}+A_{\alpha}\right) \cdot \mathcal{P}_{d}^{(H M)}+\sum_{\alpha} B_{\alpha} A_{\alpha} \cdot \mathcal{P}_{d}^{\left(H H^{\prime}\right)} . \tag{3.9}
\end{align*}
$$

Note that the first parcel is the summation of the case where the non-connected digraph occurs in the first halve, $\sum_{\alpha} B_{\alpha}^{2} \cdot \mathcal{P}_{d}^{(H H)}$, with the case where it occurs in the second halve, $\sum_{\alpha} A_{\alpha}^{2} \cdot \mathcal{P}_{d}^{(H H)}$. We denote the coefficients of $\mathcal{P}_{d}^{(H H)}, \mathcal{P}_{d}^{(M M)}, \mathcal{P}_{d}^{(H M)}$, and $\mathcal{P}_{d}^{\left(H H^{\prime}\right)}$ by $r, s, u$, and $v$, respectively.

Recalling the probabilities from 3.4.1, for $1 \leq d \leq \ell$, we have:

$$
\begin{aligned}
& \mathcal{P}_{d}^{(H H)}=\left|1-\frac{2 d}{\ell}\right|-\frac{1}{\ell}, \\
& \mathcal{P}_{d}^{(M M)}= \begin{cases}0 & \text { if } d<\ell \\
\frac{1}{\ell} & \text { if } d=\ell,\end{cases} \\
& \mathcal{P}_{d}^{(H M)}= \begin{cases}\frac{2}{\ell} & \text { if } d<\ell \\
0 & \text { if } d=\ell,\end{cases} \\
& \mathcal{P}_{d}^{\left(H H^{\prime}\right)}= \begin{cases}1-\frac{1}{\ell}-\left|1-\frac{2 d}{\ell}\right| & \text { if } d<\ell \\
0 & \text { if } d=\ell,\end{cases}
\end{aligned}
$$

which we can plug in the formula for $\mathcal{P}_{d}$ from 3.9, to obtain:

$$
\mathcal{P}_{d}= \begin{cases}r+\frac{1}{\ell}(2 u-v-r)-\frac{2 d}{\ell}(r-v) & \text { if } d \leq \frac{\ell-1}{2} \\ 2 v-r+\frac{1}{\ell}(2 u-v-r)+\frac{2 d}{\ell}(r-v) & \text { if } \frac{\ell-1}{2}<d<\ell \\ r+\frac{1}{\ell}(s-r) & \text { if } d=\ell .\end{cases}
$$

Notice that:

- $r-v=\sum_{\alpha}\left(B_{\alpha}^{2}+A_{\alpha}^{2}-B_{\alpha} A_{\alpha}\right)=\sum_{\alpha}\left(\left(B_{\alpha}-\frac{1}{2} A_{\alpha}\right)^{2}+\frac{3}{4} A_{\alpha}^{2}\right) \geq 0$;
- $\mathcal{P}_{i}=\mathcal{P}_{\ell-i}$ for $i=1,2, \ldots, \frac{\ell-1}{2}$.

These two observations help one to better understand how $\mathcal{P}_{d}$ behaves. Thus, one concludes that:

$$
\begin{equation*}
\mathcal{P}_{1} \geq \mathcal{P}_{2} \geq \cdots \geq \mathcal{P}_{\frac{\ell-1}{2}}=\mathcal{P}_{\frac{\ell+1}{2}} \leq \cdots \leq \mathcal{P}_{\ell-2} \leq \mathcal{P}_{\ell-1}=\mathcal{P}_{1} . \tag{3.10}
\end{equation*}
$$



Figure 3.17: Expected behaviour of function $\mathcal{P}_{d}$.
On the other hand,

$$
\begin{aligned}
\ell\left(\mathcal{P}_{\ell}-\mathcal{P}_{1}\right) & =(\ell r+(s-r))-(\ell r+(2 u-v-r)-2(r-v)) \\
& =2 r+s-2 u-v \\
& =\sum_{\alpha}\left(2 B_{\alpha}^{2}+2 A_{\alpha}^{2}+C_{\alpha}^{2}-2 C_{\alpha} B_{\alpha}-2 C_{\alpha} A_{\alpha}-B_{\alpha} A_{\alpha}\right) .
\end{aligned}
$$

This means that $\ell\left(\mathcal{P}_{\ell}-\mathcal{P}_{1}\right)$ is of the form $2 x^{2}+2 y^{2}+z^{2}-2 z x-2 z y-y x$, which is not positive definite, that is, a function that is always positive or equal to zero, as shown in [MR07, p. 6]. However, for $\mathcal{P}_{\ell}-\mathcal{P}_{2}$, we have the following.

$$
\begin{aligned}
\ell\left(\mathcal{P}_{\ell}-\mathcal{P}_{2}\right) & =(\ell r+(s-r))-(\ell r+(2 u-v-r)-4(r-v)) \\
& =4 r+s-2 u+5 v \\
& =\sum_{\alpha}\left(4 B_{\alpha}^{2}+4 A_{\alpha}^{2}+C_{\alpha}^{2}-2 C_{\alpha} B_{\alpha}-2 C_{\alpha} A_{\alpha}+5 B_{\alpha} A_{\alpha}\right) .
\end{aligned}
$$

Let $f(x, y, z)=4 x^{2}+4 y^{2}+z^{2}-2 z x-2 z y+5 x y$. We can verify that it is positive definite by finding the critical points of $f$, using the gradient of $f$, and then checking the concavity at those points, as follows.

$$
\nabla f=\mathbf{0} \Longleftrightarrow\left[\begin{array}{c}
8 x+5 y-2 z \\
5 x+8 y-2 z \\
-2 x-2 y+2 z
\end{array}\right]=\left[\begin{array}{l}
0 \\
0 \\
0
\end{array}\right] \Longleftrightarrow\left\{\begin{array}{l}
y=0 \\
x=0 \\
z=0
\end{array} \Longrightarrow(0,0,0)\right. \text { is the only critical point. }
$$

By calculating the reduced row echelon form of the Hessian matrix, using the Gaussian elimination technique, we can understand the function behaviour at the point $(0,0,0)$.

$$
\operatorname{Hess} f(x, y, z)=\left(\begin{array}{ccc}
\frac{\partial^{2} f}{\partial x^{2}} & \frac{\partial^{2} f}{\partial y \partial x} & \frac{\partial^{2} f}{\partial z \partial x} \\
\frac{\partial^{2} f}{\partial x \partial y} & \frac{\partial^{2} f}{\partial y^{2}} & \frac{\partial^{2} f}{\partial z \partial y} \\
\frac{\partial^{2} f}{\partial x \partial z} & \frac{\partial^{2} f}{\partial y \partial z} & \frac{\partial^{2} f}{\partial z^{2}}
\end{array}\right)=\left(\begin{array}{ccc}
8 & 5 & -2 \\
5 & 8 & -2 \\
-2 & -2 & 2
\end{array}\right)
$$

The matrix above can be reduced to the identity matrix using Gaussian elimination, whose function behaviour is known: the function is positive definite. Thus, the only critical point, $(0,0,0)$, is the minimum, and is also positive definite. A similar approach to $\ell\left(\mathcal{P}_{\ell}-\mathcal{P}_{1}\right)$ does not yield the same result.

This proves that $f: d \longmapsto \mathcal{P}_{d}$, evaluated over a bifid cryptogram of period $\ell$, is approximately a periodic function with period $\ell$. For more details see [MR07].

### 3.4.4 Distribution of the standard deviation for non-connected digraphs

The strategy presented in the previous section does not always yield clear results. This subsection covers a possible strategy to make the results clearer, to find the correct period, originally presented in [MR07]. Figure 3.18 shows the function applied to two different bifid cryptograms, both of size 500 . While one can see that in the first cryptogram there is a period of 9 , the second cryptogram does not look periodic at all.


Figure 3.18: Results of applying $f$ to two different bifid cryptograms with period 9 .

Furthermore, for some cryptograms, the graph of the distribution function $f: d \longmapsto \mathcal{P}_{d}$ is especially flat, making the previous method useless. A better alternative is to take the function $f^{\prime}: d \longmapsto s t d_{d}$, where $s t d_{d}$ is the standard deviation of the frequencies of the non-connected digraphs of distance $d$. The graph will usually reveal half of the period as its maximum value. In Figure 3.19 we can see, for a bifid cryptogram of size 500 , produced with period 12 , the non-connected digraphs approach and the standard deviation approach, respectively.

The flattening of the distribution comes from the fact that pairs of characters not at distances equal to half the period in the ciphertext (around half the period for odd periods) come from


Figure 3.19: Usage of the standard deviation on a cryptogram that defeats the non-connected digraphs method.
non-contiguous characters in the plaintext, as shown in Figure 3.20. However, for those distances, the statistical peculiarities of the digraphs of the original language cause a slight increase of the standard deviation.

| $x_{0}$ | $y_{0}$ | $x_{1}$ | $y_{1}$ | $x_{2}$ | $y_{2}$ | $x_{3}$ | $y_{3}$ |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| $x_{4}$ | $y_{4}$ | $x_{5}$ | $y_{5}$ | $x_{6}$ | $y_{6}$ | $x_{7}$ | $y_{7}$ |


| $x_{0}$ | $y_{0}$ | $x_{1}$ | $y_{1}$ | $x_{2}$ | $y_{2}$ | $x_{3}$ |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| $y_{3}$ | $x_{4}$ | $y_{4}$ | $x_{5}$ | $y_{5}$ | $x_{6}$ | $y_{6}$ |


| $x_{0}$ | $y_{0}$ | $x_{1}$ | $y_{1}$ | $x_{2}$ | $y_{2}$ | $x_{3}$ |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| $y_{3}$ | $x_{4}$ | $y_{4}$ | $x_{5}$ | $y_{5}$ | $x_{6}$ | $y_{6}$ |

Figure 3.20: Interference of coordinates for even (8 characters) and odd (7 characters) periods.

For odd periods, half-period would be a noninteger, and this has consequences on the distribution of the standard-deviation. Instead of a singular maximum, as seen in Figure 3.19, there are two consecutive peaks around the half-period. This is shown in Figure 3.21.


Figure 3.21: Standard deviation test for a cryptogram of period 7.

In [MR07] it is also shown how to determine the cipher key, but this is already beyond the scope of this work.

### 3.4.5 Building an heuristic for the non-connected digraphs strategy

With what we have covered in this section, it is possible for one to build an heuristic that uses the non-connected digraphs strategy to tell some ciphers apart. Consider the following function $n c d$, which counts the frequency of non-connected digraphs for a given distance $d$ in a given cryptogram $x$.

$$
\begin{aligned}
n c d: \Gamma^{*} \times \mathbb{N} & \longrightarrow \mathbb{N} \\
(x, d) & \longrightarrow \sum_{i \in\{0, \ldots,|x|-d\}}\left[x_{i}=x_{i+d}\right] .
\end{aligned}
$$

Using function $n c d$ one can calculate a list list $_{n c d}$, a list of frequencies for each possible period. Let $\pi_{\text {min }}$ and $\pi_{\text {max }}$ denote, respectively, the minimum and maximum period we are considering, then

$$
\text { list }_{n c d}(x)=\left[n c d(x, \pi): \pi \in\left\{\pi_{\min }, \ldots, \pi_{\max }\right\}\right] .
$$

Having this list, one can check for periodicity, employing the same strategy as the one presented in 3.2.1. Thus, one can use function $\phi_{\text {likely }}$ again, using list $_{n c d}$ instead of $l i s t_{I C}$.

Figure 3.22 shows the results of the application of function $\phi_{\text {likely }}$ to a set of 60000 cryptograms. Here, the likelihood threshold, $l t$, is equal to $\frac{6}{10}$. This value was picked for the threshold in an attempt to separate the ciphers into different groups. Recall that the dots to the right of the image are outliers, as explained in Appendix A.


Figure 3.22: Likelihood distribution for non-connected digraphs.

The strategy does not seem very effective against bifid cryptograms, since many of these do not show signs of periodicity when this strategy is used. However, the strategy is still useful for other ciphers. Using the data shown in Figure 3.22, we came up with the following separation of ciphers into branches, shown in Table 3.9, similarly to what was already done for the other heuristics.

Table 3.9: Performance measurement of $h_{n c d}$ branches.

| Percentage | Interval | Set of ciphers |
| :---: | :---: | :---: |
| 0.598838 | $\phi_{\text {likely }}(x)=0$ |  |
| 0.727378 | $\phi_{\text {likely }}(x) \neq 0$ |  |\(\left\{\begin{array}{c}\left\{\begin{array}{c}Nihilist transposition, Phillips, <br>

Input autokey, Output autokey, <br>
Caesar, Playfair\end{array}\right\} <br>
bifid, Trifid, Numbered key, <br>
Chequerboard, Nihilist substitution, <br>
Vigenère\end{array}\right\}\)

From this, the heuristic follows:

$$
\begin{aligned}
h_{n c d}: \Gamma^{*} \longrightarrow & {[0,1] \times 2^{C} } \\
x & \left(\begin{array}{ll}
\left.0.60,\left\{\begin{array}{c}
\text { Nihilist transposition, Phillips, } \\
\text { Input autokey, Output autokey, } \\
\text { Caesar, Playfair }
\end{array}\right\}\right), & \text { if } \phi_{\text {likely }}(x)=0 \\
\left(0.73,\left\{\begin{array}{c}
\text { bifid, Trifid, Numbered key, } \\
\text { Chequerboard, Nihilist substitution, } \\
\text { Vigenère }
\end{array}\right\}\right), & \text { otherwise. }
\end{array}\right.
\end{aligned}
$$

Similarly to what was done in 3.2 .1 , we examined the most likely periods to see how well the strategy performs regarding finding the correct period. Table 3.10 shows the result for each cipher. Even though the strategy does not perform well in finding the period of more complex ciphers, it can be useful in finding some non-periodic ciphers, most notably the nihilist-transposition.

Table 3.10: Fraction of correctly guessed periods using the underlying strategy of $\phi_{\text {likely }}$.

| Cipher | Percentage |
| ---: | :--- |
| Nihilist transposition | 0.9694 |
| Caesar | 0.7978 |
| Playfair | 0.7670 |
| Vigenère | 0.6770 |
| Numbered key | 0.6462 |
| Chequerboard | 0.6192 |
| Nihilist substitution | 0.4060 |
| Trifid | 0.1202 |
| bifid | 0.1046 |
| Input autokey | 0.0628 |
| Output autokey | 0.0440 |
| Phillips | 0.0120 |

## Chapter 4

## Building an automated classifier for classical ciphers

The first step towards deciphering a cryptogram is to identify the cipher that was used to encrypt it. Having identified the cipher, the attacker can attempt to crack the cryptogram, exploiting the properties of the cipher. However, identifying the cipher used to produce a given cryptogram is not an easy task, particularly for sophisticated ciphers whose properties are harder to spot on a cryptogram.

In this chapter, we describe a program that automatically attempts to identify which cipher, from a set of ciphers, was used to produce a given cryptogram. We call it the classifier program. This is done using heuristics that indicate which set of ciphers may have been used to produce the cryptogram. By combining all the heuristics' guesses, it is possible to get an idea of what ciphers were most likely used to produce the cryptogram. However, heuristics can differ in both consistency, precision and splitting-ability. These are, respectively, how prone to fail, how accurate and how capable of telling ciphers apart, each heuristic is. Since they differ in these three attributes, it is necessary to attribute a weight to each heuristic accordingly. These weights have to be fine-tuned beforehand so that the classifier can work properly, that is, have a good chance at identifying ciphers.

Fine-tuning the heuristics' weights is a difficult task. Because of this, we wrote another program to help us find a good tuple of weights to be used by the classifier. This fine-tuning program uses a simulated annealing strategy to find a good weight tuple. It rates the weight tuples based on how good the classifier performs using each weight tuple. To measure the performance, the program starts by generating a list of cryptograms. Then it counts the number of correctly guessed ciphers of the cryptograms. As the program gets closer to finding a good tuple of weights, more cryptograms are added so that the weight tuple is as good as possible.

For the program implementation, the set of ciphers used was exactly the one introduced in Chapter 2, and the set of heuristics are the ones presented in Chapter 3.

We start by presenting an overview of the classifier, followed by an explanation of how it works. After this, we explain the other program, used to fine-tune the classifier.

### 4.1 Overview

We divide the classifier program into four different phases, as depicted in Figure 4.1.


Figure 4.1: Inner stages of the identification process.

## 1 - Input

- The cryptogram whose corresponding cipher we want to guess. The cryptogram is assumed to be in a standard format. In other words, every letter, digit or other symbol, is considered part of the cryptogram alphabet past this phase. This also means that the case variation should be discarded beforehand; otherwise, the uppercase and lowercase of a letter will be interpreted as different characters.
- A set of ciphers the program can identify.
- A list of heuristics that the classifier program will use. Each heuristic gives an indication of what subset of ciphers may have been used, from the set of ciphers the program can identify.
- A weight for each heuristic. These weights indicate how significant each heuristic is. Weights are used to find a balance between heuristics. Since heuristics differ in accuracy, precision and splitting ability, as explained before at the start of the chapter, it makes sense to give them more or less weight accordingly. Well-attributed weights improve the quality of the programs results. The process used to find a good weight tuple is explained in 4.3.

2 - Applying heuristics Each cipher induces properties on the cryptograms they produce. Each heuristic, when applied to a cryptogram, detects whether the cryptogram exhibits a given property. The result of each heuristic is a value that relates to the probability that the property was produced by a subset of ciphers known to produce it.
$\mathbf{3}$ - Scoring the ciphers The ciphers are scored according to the values calculated with the heuristics and the weights of the heuristics used to find the properties.

4 - Output The list of possible ciphers used to create the cryptogram, from most to least likely to have been used, and their respective score. Each score is a probability that the corresponding cipher was used.

### 4.2 Automatic identification process

In this section, we explain how the classifier program works. The classifier program compiles the results of different heuristics to form an idea of what cipher may have been used. You may recall from 3.2 that the function that encapsulates all heuristics is as follows:

$$
\begin{aligned}
h: N \times \Gamma^{*} & \longrightarrow[0,1] \times 2^{C} \\
(n, x) & \longrightarrow\left(\operatorname{Pr}\left[\exists c \in m\left(n, t_{n}(x)\right), \exists p \in P: x=c(p)\right], m\left(n, t_{n}(x)\right)\right),
\end{aligned}
$$

where:
$\Gamma$ is the output alphabet for all ciphers;
$N$ is the set of all the heuristics' names;
$m$ is a function that takes a name of a heuristic $n$, and a threshold value of that heuristic computed over the cryptogram $t_{n}(x)$, and outputs a set of ciphers likely to have been used to produce $x$.

Since heuristics are not all equally effective, we can further improve our guessing ability by introducing weights, one for each heuristic. The weights are values in $[0,1]$ and their sum should be equal to 1 .

Before explaining the automatic identification process, we present some notation that we will use henceforth.
$C$ - A set of ciphers; a cipher in $C$ is denoted $c$.
$p-\mathrm{A}$ plaintext.
$x_{c, p}$ - The resulting cryptogram from applying a cipher $c$ to the plaintext $p$; the same as $c(p)$.
$\Sigma$ - The input alphabet of all ciphers; every plaintext is in $\Sigma^{*}$.
$\Gamma$ - The output alphabet of all ciphers; every cryptogram is in $\Gamma^{*}$.
$H$ - The tuple of heuristics; a heuristic in $H$ is denoted as $h$.
$r_{h, x}, K_{h, x}$ - When an heuristic $h$ is applied to a cryptogram $x$ it results in a tuple: a value $r_{h, x}$ and a set of ciphers $K_{h, x}$. We also use $r_{h, c, p}$ to refer to the resulting value of applying heuristic $h$ to a cryptogram produced by computing $c(p)$.
$W$ - The tuple of weights, one for each heuristic; $w_{h}$ denotes the weight corresponding to heuristic $h$.
$S$ - The tuple of scores, one for each cipher. These express the likelihood of each cipher having been used to produce the cryptogram. Moreover, we denote the score of cipher $c$ as $s_{c}$.

The following entities are present for the program execution.

- A set $C$ of ciphers possibly used to produce the cryptogram.
- The cryptogram, $x$.
- A tuple $W$ of heuristic weights.
- A tuple of scores, $S$, one for each cipher. At the start of the identification, $s=\frac{1}{|S|}$ for any $s \in S$.

The program starts by applying each heuristic to the given cryptogram. After calculating all the resulting values, the program scores the ciphers.

The scoring works as follows: for each heuristic, if the cipher is within the resulting set of ciphers $K_{h, x}$, then the resulting value $r_{h, x}$ is multiplied by that heuristic weight $w_{h}$, and this is added to the score of the cipher. If the cipher is not within $K_{h, x}$ then nothing is added. This can be expressed as:

$$
s_{c, x}=\sum_{h \in H}\left\{\begin{array}{cl}
r_{h, x} \cdot w_{h}, & \text { if } c \in K_{h, x}  \tag{4.1}\\
0, & \text { otherwise }
\end{array}\right.
$$

Finally, the scores are normalised; that is, each score is divided by the sum of all scores. Thus, each score is within $[0,1]$ and the sum of all scores is equal to one. The closer the score of cipher $c$ is to one, the more likely it is that $x$ was ciphered using $c$.

The classifier then returns all the ciphers and their respective scores in descending order.

### 4.3 Finding a good weight tuple

As mentioned above, in Section 4.2, each heuristic has a corresponding weight. A poor configuration of the weights will result in an inferior performance of the classifier program. Finding a good tuple of weights is essential in order to identify the majority of cryptograms.

Heuristics differ in multiple factors, making the task of assigning weights difficult. We narrowed these factors down to three main ones:

Consistency Heuristics are not perfect, and so sometimes they fail. Thus, one of the factors to consider when giving weight to a heuristic is how often they fail. Failing a lot does not necessarily mean that the heuristic is useless, as it can still be used if, for example, there is no other heuristic detecting the same property. On the other hand, it may be pointless to have the heuristic if there is another, more consistent heuristic, that attempts to detect the same property in a similar way.

Precision Each heuristic tries to assign the production of a given property in the cryptogram to a subset of ciphers of those that the classifier can identify. These sets can vary in size, and precision is related to the size of that set. Having two heuristics that attempt to detect the same property, the one that can better narrow down the group of ciphers that produced the property is better.

Splitting-ability If a heuristic is good at detecting some property but cannot be used to distinguish between two ciphers, it is useless as long as identification depends on it for distinguishing those two ciphers. Moreover, some heuristics may be added to the classifier just to tell specific ciphers apart.

It can be hard to compare the consistency, precision, and splitting-ability of different heuristics, especially if they measure different properties of different ciphers. Another factor to consider is that although an heuristic can be consistent and detect a property that a cipher produces, it may be the case that the cipher does not produce the property often enough to justify using the heuristic to identify this cipher. The task of fine-tuning becomes even harder as more heuristics are added. Given these difficulties, it makes sense to create a tool to help us find a well-balanced weight tuple. In this section, we cover the core ideas for building this tool in 4.3.1, 4.3.2 and 4.3.3, explain how to build it in 4.3.4 and how to further improve it in 4.3.5.

### 4.3.1 Score interpretation

The weights of the heuristics balance the importance of each heuristic in the identification process. Because of this, and assuming the heuristics are adequate for the ciphers the program has to identify, the weight tuple is the element that dictates how good of a performance the classifier program will have at identifying.

Given this, it is necessary to find a method to rate how good a tuple is to find a working one. In order to do this, one must first decide which information from the ciphers' scores is to be taken into account to define a weight tuple as good or bad, in other words, how the scores of the ciphers are to be interpreted. This can be as simple as checking if the highest scored cipher is the original cipher, or it can take into account other factors, such as the scoring of similar ciphers or the number of ties in score. Thus, we define two possible systems, one more simple and the other more complex.

A simple approach We start by considering the most basic system possible, one that only recognizes a weight tuple as a good when it gives the highest scoring cipher to the original cipher, namely using the function:

$$
i d_{1}\left(x_{c, p}, S\right)= \begin{cases}1 & \text { if } \max (S)=\left\{s_{c}\right\} \\ 0 & \text { otherwise }\end{cases}
$$

The biggest problem with this approach is that, should the original cipher not be the given
the highest score, it makes no difference whether it is the second highest scored or the one with the lowest score. For both, the function returns 0 .
Since ciphers share properties, ciphers that share a lot of properties with the original cipher are bound to have a similar score to that of the original cipher. Furthermore, since the heuristics that pick up on these properties are not infallible, it is possible that the highest scoring cipher is not the original cipher but one that shares many properties with it.

Taking the cipher rank into account In order to fill this shortcoming, our second approach takes the ranking of the original cipher into account. Given a cipher $c$ and the list of scores of all ciphers, $S$, let $\operatorname{Rank}(c, S)$ denote the position of cipher $c$ score in the list, with the values of $S$ put in descending order. For the highest scored cipher, $\operatorname{Rank}(c, S)=0$, for the second highest scored $\operatorname{Rank}(c, S)=1$, and so on. However, if there are any draws between ciphers, the Rank of the set of ciphers with the same score value is the highest position with that score value.
From this, we define the following function:

$$
i d_{2}\left(x_{c, p}, S\right)=1-\frac{\operatorname{Rank}(c, S)}{|C|} .
$$

This function returns a greater value the lower the rank of the cipher, which gives us more information about the quality of the scoring. Furthermore, it penalises draws, given that, in the case of a draw, the highest rank is taken into account instead of the lowest.

### 4.3.2 Tuple rating

Having a function $i d$ to interpret if the scoring is good or bad, what follows is a method that measures the performance of the classifier for a given weight tuple.

1. First, a list $P$ of plaintexts is sampled from a corpus. These can vary in length, but should be long enough to allow for any feasible kind of guessing.
2. The list $P$ is used to produce a list of cryptograms, $X$, using a set of ciphers $C$, with keys selected at random. To do this, the following function enc is used.

$$
\text { enc: } \begin{aligned}
C \times P & \longrightarrow \Gamma^{*} \\
(c, p) & \longmapsto c(p) .
\end{aligned}
$$

The function is applied to every plaintext in $P$ each of the ciphers in $C$, producing $|P| \cdot|C|$ cryptograms. We denote the list of cryptograms produced as $X=\operatorname{Im}(e n c)=e n c(C \times P)$. The heuristics are applied to each cryptogram, each resulting in a tuple of results. This tuples are compiled into a list $\mathcal{R}$ of tuples of results. We denote the tuple of results corresponding to the cryptogram $x$ as $R_{x}$.
3. Finally, for each of the cryptograms in $X$, the classifier attempts to identify the producing cipher. Let $\mathcal{S}$ designate a function that takes a tuple of weights and a set of heuristics
results, calculates the scores using the two, and returns in a tuple, one for each cipher, as explained in 4.2. It is possible to express how good a weight tuple is by the ratio between the number of correctly guessed cryptograms and the total number of cryptograms, by:

$$
\begin{equation*}
\operatorname{Rating}(W, \mathcal{R}, X)=\sum_{x_{c, p} \in X} \wedge_{R_{c, p} \in \mathcal{R}} \frac{i d\left(x_{c, p}, \mathcal{S}\left(W, R_{x_{c, p}}\right)\right)}{|X|} . \tag{4.2}
\end{equation*}
$$

The number of cryptograms must be large enough to make the rating accurate.

We call this procedure weight tuple rating. This, along with the weight mutation which we describe next, is used as a tool to improve identification.

### 4.3.3 Weight mutation

By altering the way the weights are distributed, one can improve the classifier program performance in identifying the ciphers. The improvement can be noted by using the previous rating method to rate the tuple before and after the alteration. This allows one to see if the change improved the tuple. The weights are changed in the following manner:

1. The weight tuple is rated.
2. One of the weights of the tuple is chosen at random, and a pre-defined increment value is added to that weight. All weights are then divided by one plus the increment (i.e. the weight distribution is normalised so that each weight remains within $[0,1]$ ).
3. The new weight tuple is rated using the tuple rating method. We can now compare it to the initial rating and check if there was an improvement in identification. If there was, we use the new tuple instead.

The process described in step two is designated as weight mutation. When repeated, the process converges towards a functioning weight tuple.

### 4.3.4 Simulated annealing approach

The technique previously presented in 4.3.3 is called simulated annealing and has many applications, as seen in [KGV83]. The following algorithm uses this technique to find a good weight tuple for the classifier program with a given list of cryptograms.

To serve as input to the algorithm, a list of tuples of results of heuristics is calculated the following way:

1. A list $X$ of cryptograms is created, similarly to what was done before using function enc in step 2 in Section 4.3.2.
2. The heuristics are applied to each cryptogram and compiled into a list $\mathcal{R}$, similar to what was done before in step 2 in Section 4.3.2.

An illustration of the algorithm can be seen in Figure 4.2.


Figure 4.2: Simulated annealing algorithm.

The algorithm starts with the following entities.

- A list $\mathcal{R}$ of tuples of results from applying the heuristics to a list of cryptograms $X$, as previously explained.
- A weight tuple $W$ with $w=\frac{1}{|W|}, \forall w \in W$.
- An acceptable fallback ratio. This is how much the tuple rating is allowed to get worse from one weight tuple $W$, to the next $W^{n}$. Between iterations, the following condition has to be met:

$$
a f r \geq 1-\frac{\operatorname{Rating}\left(W^{n}, \mathcal{R}, X\right)}{\operatorname{Rating}(W, \mathcal{R}, X)}
$$

where $a f r$ is the acceptable fallback ratio. By allowing the tuples to get slightly worse, one can avoid stopping the algorithm at a smaller rating local maximum.

- A sliding window of previous tuples. This is how many previous weight tuples the algorithm takes into consideration to check if it has converged into a good weight tuple. The tuples are kept in the window until they are no longer useful.
- A minimum improvement value. This is a value used to check if the algorithm is still converging or has stabilised. The algorithm stops if the current weight tuple rating minus the average of ratings within the window is lower than that value, that is:

$$
\operatorname{miv} \leq \operatorname{Rating}(W, \mathcal{R}, X)-\frac{\sum_{W^{\prime} \in \text { window }} \operatorname{Rating}\left(W^{\prime}, \mathcal{R}, X\right)}{\mid \text { window } \mid}
$$

where miv is the minimum improvement value.

- A maximum number of attempts the algorithm will try to improve a weight tuple by mutation, before finally stopping.
- The increment given to a random weight for the tuple mutations.

After the above input is given, the algorithm proceeds with the following steps:

1. The initial weight tuple $W$ is rated. The higher the value of $\operatorname{Rating}(W, \mathcal{R}, X)$, the better the starting weight tuple is;
2. The current tuple of weights is mutated using the input increment;
3. The new weight tuple, the mutation result, is rated. The algorithm then compares the rating of the new tuple $W^{n}$ with the rating of the previous tuple $W$.
(a) If the rating of the new tuple is better, or at least within the acceptable fallback ratio, two things can be done:
i. If the rating is close enough to the average of the ratings of the weight tuples within the window (and the window is full), the algorithm considers the weight tuple rating to have converged and returns it. To check if it is close enough, the minimum improvement value is used as previously described;
ii. Otherwise, the search for a good tuple continues, using the mutated tuple, and the old tuple is added to the window. If the window is already full, the oldest tuple is discarded. The algorithm jumps back to step two, repeating the process with the new set;
(b) On the other hand, if the rating is not within the acceptable fallback, the new tuple is discarded. The algorithm then jumps back to step two and tries another mutation. After the maximum number of attempts, the algorithm stops and returns the previous tuple.

From now on, we call the above algorithm the simulated annealing application.

### 4.3.5 Further improvements to the algorithm

The previously explained algorithm can calculate a good weight tuple. However, there is a risk that the weight tuple obtained is only good for the given list. In order to solve this problem, one can repeat the simulated annealing algorithm for a gradually growing sample of cryptograms. At the same time, one can also gradually shrink the weight increment when a better rating is not obtained. This enables the change to the weight tuples to become smaller as the algorithm converges to a local maxima, giving it more precision. With this in mind, one can create an improved algorithm that takes this into account. An illustration of this algorithm can be seen in Figure 4.3.


Figure 4.3: Fine-tuning algorithm.

The algorithm starts with the following entities present:

- The necessary input for the simulated annealing application, as before.
- A corpus to sample plaintexts from, the ciphers to produce the cryptograms, and the heuristics to apply to the cryptograms.
- A value $\alpha$ of how much the ratings of the tuples must improve between iterations.
- A constant $N$, the maximum number of times the algorithm is allowed to fail consecutively to achieve a substantial improvement in rating.

The algorithm proceeds as follows.

1. Cryptograms are added to the list.
2. The simulated annealing application is run with the results from the previous step, the current weight tuple, and the remainder of its input.
3. The current weight tuple is updated.
4. The new weight tuple rating is checked to see if there was a substantial improvement. There is a substantial improvement if the following condition holds:

$$
\operatorname{Rating}\left(W^{\prime}, \mathcal{R}, X\right) \geq \operatorname{Rating}(W, \mathcal{R}, X)+\alpha
$$

If there is substantial improvement the algorithm jumps back to step 1 , adding more results of heuristics to $\mathcal{R}$.

If there is not, and it is the $N$ th consecutive time there is not, the algorithm stops, returning the current weight tuple and its rating. If it is not the $N$ th consecutive time, the weight increment is reduced to half, and the algorithm jumps back to step 2.

The reason the algorithm is allowed to fail up to $N$ consecutive times is that it may be the case the rating is not improving because the weight increment is too big. As the weight tuple achieves a better rating and the number of cryptograms increases, finer adjustments are necessary to continue improving. On the other hand, since more results are added every time there is a significant improvement in rating, there is an assurance that the rating did not increase by chance.

## Chapter 5

## Results

In this chapter, we present the following:

- The performance results obtained using the algorithm to compute the weight tuples. Given that the fine-tuning algorithm improves the weight tuple based on the performance of the classifier itself, just by analysing the results of the fine-tuning we can already have an idea of how both the algorithm and the classifier perform;
- The input that is given to our setup to get the results, as well as some details regarding the implementation of our classifier;
- An overview of how good the heuristics are at telling ciphers apart, focussing on their precision and splitting-ability. This is in contrast to what was done in Section 4.3, where we focussed on the consistency factor.


### 5.1 Setup for computing results

To put our classifier into action, we programmed the algorithm in Python along with the generation of cryptograms. The language was chosen for its simplicity.

In our program we avoided the use of floating-point variables in order to avoid precision loss in our calculations. Instead, we used rational numbers from the fractions Python module. The downside of this is that operations with fractions take more time. Furthermore, after operating (adding, multiplying, etc.) a lot of fractions, there is a tendency for the numerator and denominator to become larger and larger, making the operations slower. This also makes it impossible for the user to understand the order of magnitude of these values. For these reasons, we did not fully remove floating-point usage from our code, but avoided it as much as possible where it made sense to do so.

In order to make the observations that follow, the simulated annealing algorithm was run 200 times, 100 times for each of the two identification measurement functions. Information about
the algorithm performance was collected on each run of the algorithm and compiled into a data set, which had 200 entries. Each entry had information of a different fine-tuning program run, namely:

- The identification function;
- The elapsed time;
- The number of iterations to converge;
- The number of cryptograms used;
- The rating of the final weight tuple;
- The number of correct guesses for the final weight tuple;
- The number of correct guesses plus the number of ties at first place for the final weight tuple;
- The weights of each heuristic in the final weight tuple.

The setup that follows is what we used to run the algorithm. The corpus used was the same as that we used for our observations in Chapter 3. In Chapter 4, two types of input are presented for the algorithm: the input necessary for the simulated annealing part of the algorithm and the input of the fine-tuning part of the algorithm, the "outer layer". The input to the fine-tuning part of the algorithm was the following:

- Necessary improvement between iterations: $\frac{1}{100}$;
- Number of simulated annealing attempts: 3;
- Initial weight increment: $\frac{1}{6},\left(=\frac{1}{|H|}\right)$;
- Initial weights: $\frac{1}{|C|}$ for $c \in C$;
- Number of cryptograms added per iteration: 3600. This number comes from the fact that for each iteration 300 texts are selected and each is ciphered with all ciphers in $C$, resulting in $300 \times 12=3600$ cryptograms. The cryptograms that were added in the last iteration of simulated annealing are not taken into account for our observations since they have no impact on the results.

For the cryptogram generation of periodic ciphers, we defined the minimum period to be 5 and the maximum 20. The size of the plaintexts used to create the cryptograms was 500 . The input for the simulated annealing part of the algorithm was the following:

- Acceptable fallback ratio: $\frac{1}{20}$;
－Size of the sliding window of the previous tuples：12；
－Minimum improvement value：$\frac{1}{200}$ ；
－Maximum number of attempts to improve the rating for simulated annealing： 3 ．

These inputs were chosen through trial and error，taking in consideration both the rating and the amount of correctly identified cryptograms．The program runtime was also taken into consideration．

## 5．2 Expectations

As explained in Section 1．1，we were optimistic that using our classifier we would be able to tell which heuristics are more or less important in the identification process．This importance is related to the consistency，precision，and splitting ability of the heuristic，as explained in Section 4．3．Our previous analysis to each heuristic in Chapter 3 was mostly consistency－wise， however，we can also speculate about the classifier performance regarding both precision and splitting－ability．In this regard，Table 5.1 shows in which branches of the heuristics each cipher is present．By comparing the columns of the table for each pair of ciphers，we can get an idea of

Table 5．1：Table showing the relation between the heuristics＇branches and the ciphers．

## Ciphers

|  |  | $\begin{aligned} & \text { ت゙ } \\ & \text { だ } \\ & \text { む̃ } \end{aligned}$ |  |  | 4 0 0 0 0 0 0 0 0 0 0 3 | $\begin{aligned} & \vec{H} \\ & \end{aligned}$ | ت |  |  | $\frac{0}{\rightrightarrows}$ |  | $\text { uo!̣łnł!̣৭sqns } \ddagger \text { S!!! }!!N$ |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | $h_{\alpha}$ | 4，5， 6 | 7 | 7 | 7 | 6 | 8 | 2 | 3 | 5， 6 | 5， 6 | 1， 2 | 4，5， 6 |
|  | $h_{\text {IC }}$ | 2 | 1 | 1 | 1 | 1 | 1 | 3 | 1， 2 | 1 | 1 | 3 | 2 |
|  | $h_{\text {trans }}$ | 1 | 1 | 1 | 1 | 1 | 1 | 3 | 1 | 1 | 1 | 3 | 2 |
|  | $h_{\text {ICperiod }}$ | 1 | 2 | 1 | 1 | 1， 2 | 1， 2 | 1 | 1 | 1 | 1 | 2 | 1 |
|  | $h_{\text {phillips }}$ | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 2 | 3 | 2 | 1 | 1 |
|  | $h_{n c d}$ | 1 | 2 | 1 | 1 | 2 | 2 | 2 | 2 | 1 | 1 | 2 | 1 |

which ciphers the classifier may have the hardest time distinguishing．The more dissimilar two columns of the table are，the easier it is to tell apart the ciphers of those columns，and vice versa． Thus，it is easy to see that it will be impossible for the classifier to tell both autokey ciphers apart，since none of our heuristics separates these ciphers into different branches．This，of course， ignores the consistency factor of each heuristic，since it does not take into account how often the heuristics fail．

Let $b_{h, c}$ denote the set of branches of heuristic $h$ for which cipher $c$ is in the branch set. To create the heatmap shown in Figure 5.1, the following function heat was calculated for every pair of ciphers, using the fact that the more similar the columns of the two ciphers are in the table, the closer the value returned by the function is to 1 , and vice versa.

$$
\text { heat: } \begin{aligned}
C \times C & \longrightarrow[0,1] \\
\left(c_{1}, c_{2}\right) & \longrightarrow \frac{1}{|H|} \times \sum_{h \in H} \frac{\left|b_{h, c_{1}} \cap b_{h, c_{2}}\right|}{\left|b_{h, c_{1}} \cup b_{h, c_{2}}\right|} .
\end{aligned}
$$

Looking at the heatmap, we can expect that the most difficult pairs to distinguish to be those


Figure 5.1: Heatmap showing the expected difficulty of telling apart each pair of ciphers.
with the darkest squares. Given that the heuristics generally achieved a good performance, as shown in Chapter 3, we can, for the most part, ignore the consistency difference between heuristics.

### 5.3 Measuring the performance of the algorithm

In this section, we measure the performance of the algorithm with respect to different factors. For each of these factors, we also compare the two alternative functions, $i d_{1}$ and $i d_{2}$, that we use to interpret the scores, as explained in Section 4.3.1. We draw this comparison on the basis of the impact they make on the algorithm performance itself.

### 5.3.1 Algorithm convergence speed

Using the computed dataset, as described in Section 5.1, the distribution of the number of iterations that the algorithm takes to converge was calculated. Since the algorithm only converges as long as there is substantial improvement, this distribution shows the number of times that cryptograms were added to the algorithm per run. Figure 5.2 shows this distribution. In the distribution, we see that the algorithm generally converges faster for $i d_{2}$. We decided not to compute a larger dataset since this would take a long time.


Figure 5.2: Comparison between functions regarding the number of iterations the algorithm takes to converge.

Figure 5.3 shows the distribution of the algorithm run-time for each of the $i d$ functions, run on a machine with a 2.300 GHz CPU . In the figure, one sees that for the function $i d_{2}$ the algorithm has, generally speaking, a longer runtime. This is not surprising given that $i d_{2}$ does a lot more operations than $i d_{1}$.


Figure 5.3: Comparison between functions regarding the real time the algorithm takes to converge.

We used dPython to profile the code and verify this. We did this to two runs of the algorithm, one using $i d_{1}$ and the other $i d_{2}$. We made sure to use two executions where the same number of
iterations of the algorithm occurred, in this case 3 . Table 5.2 and 5.3 show the data for both $i d_{1}$ and $i d_{2}$, respectively. The following functions are present in the tables:
<module> - The entire program;
cipher_and_calc_heuristics - As the name indicates, the function that is called to cipher and calculate the heuristics for all texts, when they are added to the algorithm;
get_weights_rating - Computes the rating of the weight tuple;
get_scores - Function that, given a weight tuple, calculates the cipher scores;
_add - Addition of fractions, which requires calculating the greatest common divisor;
id 1 and id_2 - The $i d$ function used;
get_ranking - Computes the ciphers' ranking, and is only called by id_2.

For each, the table shows:

1. Name of the function;
2. Number of times it was called during execution;
3. Time the function takes to execute - in milliseconds and in percentage relative to the whole program;
4. Own time, that is, the time the function takes to execute, ignoring the time of the functions called by that function - in milliseconds and in percentage relative to the whole program.

Both tables are sorted by time.
Table 5.2: Profiling results for id_1.

| Function name | Call count | Time ms | $\%$ | Own time ms | $\%$ |
| :--- | :--- | ---: | ---: | ---: | :---: |
| <module> | 1 | 389962 | 100.00 | 4 | 0.00 |
| cipher_and_calc_heuristics | 600 | 256290 | 65.72 | 49 | 0.00 |
| get_weights_rating | 74 | 111573 | 28.61 | 781 | 0.00 |
| get_scores | 446400 | 89339 | 22.91 | 10163 | 0.03 |
| _add | 16251480 | 58693 | 15.05 | 29592 | 0.08 |
| id_1 | 442800 | 20292 | 5.20 | 2096 | 0.01 |

Looking at the tables, one can see that:

- The majority of time is spent creating more cryptograms and computing the heuristics over them in either case;

Table 5.3: Profiling results for id_2.

| Function name | Call count | Time ms | $\%$ | Own time ms | $\%$ |
| :--- | :--- | ---: | ---: | ---: | :---: |
| <module> | 1 | 475235 | 100.00 | 3 | 0.00 |
| cipher_and_calc_heuristics | 600 | 242462 | 51.02 | 32 | 0.01 |
| get_weights_rating | 67 | 211221 | 44.45 | 774 | 0.16 |
| id_2 | 424800 | 122336 | 25.74 | 874 | 0.18 |
| get_ranking | 428400 | 119568 | 25.16 | 5263 | 1.11 |
| get_scores | 428400 | 86855 | 18.28 | 9559 | 2.01 |
| _add | 15617784 | 57394 | 12.08 | 28400 | 5.98 |

- Function $i d_{2}$ spends about the same time with fractional operations as $i d_{1}$;
- The need to rank ciphers with function get_ranking is the biggest reason for $i d_{2}$ being slower than $i d_{1}$, as expected.

Although we did not take advantage of this, one convenience of our fine-tuning program is that it can be implemented to take advantage of threading. One way to do this is to run in parallel the computation of the heuristics for different cryptograms, as well as the generation of the cryptograms themselves.

### 5.3.2 Tuple rating vs number of correct guesses

Using the dataset, we make a comparison between the rating and the number of correct guesses for both functions $i d$. Figure 5.4 shows a box plot of the rating distribution for the computed tuple of weights. Figure 5.5 shows the distribution of the ratio between the number of correct cipher guesses and the total number of guesses in the final iteration of the algorithm. It is important to note that both distributions were taken from the same algorithm runs. Thus, for each rating, there is a corresponding ratio of correct guesses. Also note that correct guesses here do not include ties between ciphers at the first position.


Figure 5.4: Distribution of rating values.

Let $c g c$ be a function that counts the number of correctly guessed cryptograms, taking as input the weight tuple and the heuristics results. We can define this function as $\operatorname{cgc}(W, \mathcal{R})=\sum_{x_{c, p} \in X} \wedge R_{x_{c, p} \in \mathcal{R}} i d_{i}\left(x_{c, p}, \mathcal{S}\left(W, R_{x_{c, p}}\right), i \in\{1,2\}\right.$, taking advantage of the already


Figure 5.5: Distribution of the ratio of correct guesses.
defined function $i d_{1}$ to count the number of correct guesses. Using the function, we can define the ratio between the number of correctly guessed cryptograms and the total number of correctly guessed cryptograms as $\frac{\operatorname{cgd}(W, \mathcal{R})}{|X|}$. With this, we can conclude that, for $i d_{1}$, the rating and the number of correct guesses should be the same.

$$
\begin{gathered}
\frac{\operatorname{cgc}(W, \mathcal{R})}{|X|}=\frac{\sum_{x_{c, p} \in X \wedge R_{x_{c, p}} \in \mathcal{R}} i d_{1}\left(x_{c, p}, \mathcal{S}\left(W, R_{x_{c, p}}\right)\right)}{|X|}= \\
\sum_{x_{c, p} \in X \wedge R_{x_{c, p}} \in \mathcal{R}} \frac{i d_{1}\left(x_{c, p}, \mathcal{S}\left(W, R_{x_{c, p}}\right)\right)}{|X|}=\operatorname{Rating}(W, \mathcal{R}, X)
\end{gathered}
$$

Thus, the distribution of both ratings and correct guesses for $i d_{1}$ is the same.
Since the ratio of correct guesses was so low, we figured that, given that there were few heuristics, there may have been many ties at the first position. In fact, if we account for the ties in the first position as correct guesses, we can see that, for both functions, the algorithm is capable of correctly identifying more than $70 \%$ cryptograms. Figure 5.6 shows this.


Figure 5.6: Distribution of the ratio of correct guesses and ties at first position.

### 5.4 Comparing heuristics

The Figure 5.7 shows the distribution of the weights for each different score interpretation function. All heuristics follow a normal distribution. In the figure, one can see that the weights remain mostly the same between the functions $i d_{1}$ and $i d_{2}$. On the other hand, for the function $i d_{1}$ the differences between the heuristics are much more noticeable, especially for the heuristic $h_{n c d}$. This is to be expected since the function $i d_{1}$ penalises the heuristics much more for their inaccuracy.


Figure 5.7: Distribution of the weights of the heuristics using function $i d_{1}, i d_{2}$, respectively.

The figure shows that for both functions, $h_{n c d}$ is, in general, given less weight. This is easily explained by the fact that $h_{n c d}$ is, of the presented heuristics, the least accurate. On the other hand, $h_{\alpha}$ is the one given more weight since not only is it very accurate, it also has more branches than all the other heuristics.

Since every heuristic follows a normal distribution, we can use Pearson correlation coefficients to try to find correlations between the heuristics. Correlation is a measure of a monotonic association between two variables. A monotonic relationship between two variables is one of the following:

- If one variable increases in value, so does the other;
- If one variable increases in value, the other decreases.

In this context, we attempt to find correlation between heuristics, meaning we are trying to find any indication that any two heuristics have a monotonic relationship, in either direction. In the direction where both values increase, we call it a positive correlation, and in the other direction,
we call it a negative correlation.
The degree to which the change in one continuous variable is associated with a change in another continuous variable can mathematically be described in terms of the "covariance of the variables" [SBS18, p. 1]. It is calculated as follows:

$$
\operatorname{cov}(X, Y)=\sum_{x \in X \wedge y \in Y}\left(x-\mu_{X}\right)\left(y-\mu_{Y}\right)
$$

where $\mu_{X}$ and $\mu_{Y}$ are the means of $X$ and $Y$, respectively.
However, covariance depends on the scale of the variables. Thus, to facilitate interpretation, a Pearson correlation coefficient is commonly used instead:

$$
\rho(X, Y)=\frac{\operatorname{cov}(X, Y)}{\sigma_{X} \sigma_{Y}}
$$

where $\sigma_{X}$ and $\sigma_{Y}$ are the standard deviation of $X$ and $Y$, respectively. This coefficient values range from 1 to -1 :

- The closer to 1 , the stronger the positive correlation is;
- The closer to -1 , the stronger the negative correlation is;
- The closer to 0 , in either direction, the less correlation there is.

Furthermore, it is generally agreed that values between -0.1 and 0.1 constitute a weak correlation, and values lower than -0.9 or higher than 0.9 constitute a strong correlation [SBS18, p. 3].

It is important to note that the correlation coefficient does not take into consideration the "strength of agreement" between variables. For example, two variables may correlate, but one may always be significantly higher than the other.

However, the coefficient is only intended for linear relationships. We assume that any linear combination of the normal distributions of the heuristics is normal, since these are all normal distributions in $[0,1]$. Thus, we have that the heuristics are jointly normal too. Since if there is a relationship between jointly normally distributed data, it is always linear [Kut05, p. 81], we can use the coefficient.

It is often hard to get a feel of what the coefficient represents in terms of how close to a linear function the relationship between two variables is. To get that feel, one can plot the scatter-plot of the two variables to see how much correlation there is between the two and examine it with the naked eye [Kut05, p. 5]:

- If the dots are close to forming a line with positive slope, it is indicative of a positive correlation;
- If the dots are close to forming a line with negative slope, it is indicative of a negative correlation;
- If the dots are scattered through the plot, then there is very weak or no correlation.

Figure 5.8 shows the scatter-plot for two different pairs for $i d_{1}$. The first pair, $h_{\text {phillips }}$ and $h_{\text {ICperiod }}$ has a coefficient of 0.01 , and its scatter-plot shows no indication of correlation between the two. The second pair, $h_{\alpha}$ and $h_{i c}$ has a coefficient of -0.43 , and its scatter-plot shows some indication of a negative correlation between the two, although it is still unclear.


Figure 5.8: Two examples of scatter-plots with the heuristics values.

Finally, a Pearson correlation matrix can be computed, by calculating the coefficient for every cipher pair. Figure 5.9 shows the matrices for the results of $i d_{1}$ and $i d_{2}$, respectively. Although
Heuristic

Figure 5.9: Correlation matrices of $i d_{1}$ and $i d_{2}$, respectively.
the matrices differ from each other, no coefficient has a value significant enough to indicate there is an evident correlation.

## Chapter 6

## Conclusions

### 6.1 Expectations and Reality

Our results were significantly better than the expectations presented in Section 5.2. To test how well the ciphers were being identified by the classifier, we run the classifier with a weight tuple found by the fine-tuning algorithm. We put the classifier to test against 3600 cryptograms, 300 for each cipher.

Figure 6.1 shows a heatmap, where each square shows the ratio between the number of cryptograms of the row cipher misidentified with the column cipher and the total number of cryptograms produced by the row cipher. Let misid be a function that, given a cipher $c$ and a cryptogram $x$, not produced with $c$, returns 1 if the classifier wrongly classifies $x$ as having been produced by cipher $c$, and 0 otherwise. We use the following function to produce the heatmap:

$$
\begin{aligned}
\text { heat }^{\prime}: C \times C & \longrightarrow[0,1] \\
c_{1}, c_{2} & \longrightarrow \frac{1}{\left|X_{c_{1}}\right|} \times \sum_{x \in X_{c_{1}}} \operatorname{misid}\left(c_{2}, x\right),
\end{aligned}
$$

where $X_{c_{1}}$ is the list of cryptograms whose original cipher was $c_{1}$. Note that, since we include draws of the highest score as incorrect guesses, for a cryptogram in a draw with more than 1 incorrect cipher, multiple ciphers will be counted as incorrectly guessed for that cryptogram.

The heat map shown in Figure 6.1 is significantly different from the one shown in Figure 5.1. This is a testament to the accuracy of the heuristics, since even in cases where there is only one heuristic to tell apart the two ciphers, it is sufficiently accurate for the classifier to be able to tell them apart from each other in the majority of cases. An example of this is the Caesar and Nihilist transposition ciphers: only $6 \%$ of the Nihilist transposition cryptograms were incorrectly guessed as Caesars, and none were incorrectly guessed the other way around.

This data also allowed us to see that the relation between ciphers is not symmetric. For example, only $31 \%$ of Vigenère cryptograms are incorrectly guessed as Input autokey cryptograms, while $43 \%$ of Input autokey cryptograms are incorrectly guessed as Vigenères.


Figure 6.1: Heatmap showing the which ciphers were the hardest to tell apart for a computed weight tuple.

### 6.2 Comparing our model to the state of the art

Unfortunately we only found related written work close to the end of our project, and by that point our project had already deviated a lot from the written work. Because of this, we missed an opportunity of picking a similar method of cryptogram generation to that of [LKE ${ }^{+} 21$, p. 5], that is, using the ciphers exactly as they are described in [Ame05] and ciphered the cryptograms using random keywords, but filling in the remaining alphabet in its normal order. Furthermore, our work is a different approach from that of the other projects presented in Chapter 1.1.

Our approach can not be fooled by the presence of certain symbols. This contrasts with some of the presented projects in Chapter 1.1, which use the presence of certain symbols as an indication of what cipher may have been used.

The analysis described in Section 6.1, together with the results obtained on the performance of the classifier, allowed us to see that feature engineering is not a bad approach, since even with a few heuristics, we can identify most of the 12 ciphers, some of which are quite difficult to distinguish. To add to this, we are able to extract valuable information about the heuristics we are using, such as how useful the heuristics are at telling the given ciphers apart. Moreover, our approach allows one to see what heuristics could potentially improve the classifiers performance if added, by examining information such as the heatmaps previously presented.

### 6.3 Further work

During this thesis there were many elements about which we would have liked to research further, but for which we did not have the time to. Hoping that someone will continue the presented work, a list of these follows, as well as an explanation of why this would be interesting.

Adding more ciphers Regarding our classifier, we think it would be interesting to expand it to include all the ciphers of "The ACA and You" [Ame05], as described there, since this would make it possible to compare its performance with the performance of [LKE $\left.{ }^{+} 21\right]$. This would mean updating the heuristics we present to include all ciphers and adding new heuristics. We think it would be interesting if one could find heuristics capable of:

- distinguishing both autokeys;
- distinguishing the Bifid cipher from the CMBifid cipher.

In addition to this, updating the heuristics to work for any cryptogram length would also be interesting.

Simplifying our model In Figure 5.2, it is shown that the average number of iterations, before the algorithm stabilises, is only 2 . Thus, we find that our model may be a bit over-engineered and that one could probably produce a similar, simpler model, capable of achieving the same results.

Guessing the family of ciphers We also think it would be interesting if the user were given the most likely family of ciphers in addition to the most likely cipher. This could be achieved by using a similar approach to ours, where instead of trying to identify the cipher, the classifier would try to identify the family of ciphers. This would be useful when the cipher is not included in the set of ciphers given to our classifier, since this second classifier would give a broader idea of what ciphers may have been used.

Finding the period Similarly to the previous item, it would also be useful to have another classifier for the period length. This would mean training a classifier that, instead of identifying a cipher, tries to identify the period if there is one.

Index of coincidence and bifid ciphers As explained at the end of 3.2.1, it is possible to detect the period for both the Bifid and Trifid ciphers on certain occasions. We did not research this further, but we think it would be interesting for someone to.

## Appendix A

## Box-and-whisker plots

Boxplots are a useful way of representing univariate data that follows a normal distribution. An example of a boxplot is shown in Figure A.1. These plots are generally useful for depicting the following properties of the data [dSS86, p. 29-30]:

Locality Where the values of the variables in the data are located;
Spread How much the data varies from the mean;
Asymmetries How distant the mean and the median are from each other. Usually, an asymmetry in the data creates skewness, that is, the two tails of the Bell curve of the data are different, not symmetric.

The following figure, taken from wikipedia's page on Boxplots, shows the boxplot of a simple normal distribution and how it relates to its Bell curve plot.


Figure A.1: Comparison between a box-and-whisker plot and a Bell curve.

The vertical lines of the box in Figure A. 1 correspond to, respectively:

Q1 The median of the lower half of the dataset, known as the first quartile;
Q2 The middle value of the dataset, known as the second quartile;
Q3 The median of the upper half of the dataset, known as the third quartile.

When the boxplot is shown vertically, these are, respectively, the bottom of the box, the middle line, and the top of the box.

The difference between $Q 3$ and $Q 1$ is called Inter Quartile Range (IQR). The horizontal lines on the side of the box are called whiskers. These are a step away from the box, where a step corresponds to $1.5 \times I Q R$, following Tukey's suggested criteria for outliers [Tuk77, p. 44].

Values not within the whiskers are considered outliers, and small dots are sometimes used to indicate their presence. The following figure shows an example featuring outliers.


Figure A.2: Two boxplots, each showing the distribution of tips for a different meal.

## Appendix B

## Code

Figure B. 1 shows how the code for the classifier and the classifier fine-tuning programs should be organised into folders to work as intended. The corpus one wants to use to sample plaintexts from should be place inside the corpora folder.

```
thesis_code
    classifier
    _classifier_tools.py
        _heuristics.py
        _input_objects.py
    __simulated_annealing.py
    __tuple_rating.py
    coding
    __cipher.py
    __corpus.py
        _generatecrypto.py
        _handy_vars.py
        _polybius.py
        _statistics.py
        tools.py
    corpora
    L_<corpus>
    demonstration
        _algorithm_to_get_weights.py
        _cipher_classifier.py
        _create_example_cryptogram.py
        _sa_demonstration.py
```

Figure B.1: Folder hierarchy of the code.

Under the folder demonstration there are four different small programs to experiment with the model, these are:
algorithm_to_get_weights.py - which is the fine-tuning program;
sa_demonstration.py - which is the simulated-annealing part of the fine-tuning program; create_example_cryptogram.py - which creates a cryptogram to be used as example by the classifier;
cipher_classifier.py - which is the classifier program.

Before running any of these, one should make sure that the input is configured to one's needs by editing the file. To run any file one should do the following steps:

1. Run Python on the top folder of the repository;
2. Execute run import os within Python;
3. Execute run exec (open('file_to_run').read()), where file_to_run is the file one wants to execute.

The code of each file follows.

```
thesis_code/classifier/classifier_tools.py
```

```
from fractions import Fraction
from pandas import DataFrame
from classifier.input_objects import SAInput, HInput, FTInput
from classifier.simulated_annealing import weight_training
from classifier.tuple_rating import get_ranking
def get_weights(
    sa_input: SAInput, he_input: HInput, ft_input: FTInput, ciphers_tuple:
            tuple[str]
) -> tuple[dict[str, Fraction], DataFrame, Fraction, float]:
    " " "
    Given the input, runs the algorithm to calculate the weights and returns,
        along with the weights,
    a dataframe with the cipher scores for each generated cryptogram, the
        rating and the elapsed time
    :param ft__input: The fine-tuning input
    :param sa__input: The simulated annealing input
    :param he_input: The heuristics input
    :param ciphers_tuple: The tuple of ciphers to be considered
    :return: Tuple with the weights, dictionary, a dataframe, the rating and
        the elapsed time, respectively
    " " "
    calculated_weights, rating, calculated_heuristics_list, elapsed_time =
        weight_training(
        ft_input,
```

```
        sa_input,
        he_input,
        ciphers_tuple
    )
    scores_df = DataFrame(columns=["original_cipher"] + list(ciphers_tuple))
    for ch in calculated_heuristics_list:
        scores_dict = ch.get_scores(ciphers_tuple, calculated_weights)
        scores_df.loc[len(scores_df.index)] = [ch.original_cipher] +
        [scores_dict[a] for a in ciphers_tuple]
    return calculated_weights, scores_df, rating, elapsed_time
def scores_df_to_ranks_df(scores_df: DataFrame) -> DataFrame:
    " " "
    Takes the scores dataframe and turns it into a ranks dataframe.
    This means it substitutes the scores with the ranking positions.
    :param scores__df: The scores dataframe
    :return: The rankings dataframe
    " " "
    ranks_df = DataFrame(columns=scores_df.columns)
    ciphers_list = list(scores_df.columns)[1:]
    scores_dict = scores_df.to_dict(orient="index")
    for i in range(len(scores_df.index)):
        og_cipher = scores_dict[i].pop("original_cipher")
        ranks = get_ranking(scores_dict[i])
        new_row = [og_cipher] + [ranks[cipher] for cipher in ciphers_list]
        ranks_df.loc[len(ranks_df.index)] = new_row
    return ranks_df
def calc_correct_guesses(ranks_df: DataFrame) -> int:
    " " "
    Given a ranks dataframe calculates how many correct guesses there were.
    : param ranks_df: The ranks dataframe
    :return: The number of correct guesses
    " " "
    correct_guesses = 0
    for i, row in ranks_df.iterrows():
        og_cipher = row["original_cipher"]
        if row[og_cipher] == 0:
            correct_guesses += 1
    return correct_guesses
def calc_tied_guesses(ranks_df: DataFrame) -> int:
    " " "
    Given a ranks dataframe calculates how many ties at first guess there were.
    : param ranks_df: The ranks dataframe
```

```
: return: The number of ties at first guess
" " "
ties = 0
ciphers = list(ranks_df.columns)[1:]
for i, row in ranks_df.iterrows():
    og_cipher = row["original_cipher"]
    min_pos = min([row[cipher] for cipher in ciphers])
    if row[og_cipher] == min_pos and min_pos != 0:
                ties += 1
return ties
```

thesis_code/classifier/heuristics.py

```
from fractions import Fraction
import pandas as pd
import coding.statistics
from classifier.input_objects import HInput
from coding import tools, handy_vars, corpus
from coding.generatecrypto import cipher_text_with_given_ciphers
from coding.statistics import calc_character_abs_frequency, calc_ic,
    calc_character_rel_frequency
from coding.tools import split_text_into_list
def get_alphabet_and_split_text(text, symbols_per_character) ->
    tuple[tuple[str], list[str]]:
        " ""
        Given a text and the number of symbols per character returns the alphabet
        in a tuple and the text in a list.
        :param text: The text
        : param symbols__per_character: The number of symbols per character
        :return: Tuple with the alphabet tuple and the text list
        " " "
        alphabet = tools.infer_alphabet_from_text(text, symbols_per_character)
        split_text = split_text_into_list(text, alphabet)
        return alphabet, split_text
def h_alpha(cryptogram_alphabet: tuple[str]) -> tuple[Fraction, list[str]]:
        " " "
        Heuristic h__alpha: given the alphabet of the cryptogram returns a tuple
        with a value between 0 and 1,
    and the list of related ciphers.
    : param cryptogram__alphabet: The cryptogram's alphabet
    :return: Returns a value between 0 and 1 and a list of related ciphers
    " " "
    length = len(cryptogram_alphabet)
    if length <= 9:
        return Fraction(98, 100), ["nihilist-substitution"]
```

```
    elif length == 10:
    return Fraction(1), ["numbered-key", "nihilist-substitution"]
elif length == 20:
    return Fraction(1), ["checkerboard"]
elif 21 <= length < 24:
    return Fraction(1), ["caesar", "nihilist-transposition"]
elif length == 24:
        return Fraction(99, 100), ["nihilist-transposition", "phillips",
            "caesar", "playfair"]
elif length == 25:
    return Fraction(1), ["nihilist-transposition", "phillips", "caesar",
            "playfair", "bifid"]
elif length == 26:
    return Fraction(96, 100), ["vigenere", "autokey-input",
                "autokey-output"]
elif length == 27:
    return Fraction(1), ["trifid"]
else:
    return Fraction(0), []
def h_ic(index_of_coincidence: float) -> tuple[Fraction, list[str]]:
    " " "
    Heuristic h__ic: given the index of coincidence of the cryptogram returns a
        tuple with a value between 0 and 1,
    and the list of related ciphers.
    :param index_of_coincidence: The cryptogram index of coincidence
    :return: Returns a value between 0 and 1 and a list of related ciphers
    " " "
    if Fraction(0) <= index_of_coincidence < Fraction(55, 1000):
        return Fraction(1), ["autokey-output", "trifid", "autokey-input",
            "vigenere",
                                    "bifid", "playfair", "phillips", "checkerboard"]
    elif Fraction(55, 1000) <= index_of_coincidence < Fraction(11, 100):
        return Fraction(90, 100), ["checkerboard", "nihilist-transposition",
            "caesar"]
    elif Fraction(11, 100) <= index_of_coincidence:
        return Fraction(1), ["nihilist-substitution", "numbered-key"]
def h_trans(cryptogram_char_rel_freq: dict[str, Fraction],
                language_char_rel_freq: dict[str, Fraction]) -> tuple[Fraction,
                    list[str]]:
    " " "
    Heuristic h__trans: given the cryptogram character relative frequency and
        the language relative frequency
    returns a tuple with a value between 0 and 1,
    and the list of related ciphers.
    :param cryptogram__char__rel__freq: A dictionary with the relative frequency
        of each character in the cryptogram
    : param language_char_rel__freq: A dictionary with the relative frequency of
```

```
        each character in the language
    :return: Returns a value between 0 and 1 and a list of related ciphers
    " " "
    _sum = Fraction(0)
    for character in cryptogram_char_rel_freq
        if character not in language_char_rel_freq.keys():
            continue
        a = language_char_rel_freq[character]
        b = cryptogram_char_rel_freq[character]
        _sum += abs(a - b) ** 2
    _h_trans = 1 - (_sum / len(language_char_rel_freq.keys()))
    if Fraction(0) <= _h_trans < Fraction(99955, 100000):
        return Fraction(99, 100), ["caesar", "phillips", "playfair",
            "vigenere", "autokey-input",
                                    "autokey-output", "checkerboard", "trifid",
                                    "bifid"]
elif Fraction(99955, 100000) <= _h_trans < Fraction(1):
    return Fraction(99, 100), ["nihilist-transposition"]
elif _h_trans == Fraction(1):
        return Fraction(1), ["numbered-key", "nihilist-substitution"]
def h_ic_period(split_text: list[str], alphabet: tuple[str], p_min: int, p_max:
    int) -> tuple[Fraction, list[str]]:
        " " "
    Heuristic h_icPeriod: given the text in a list, the alphabet and a minimum
        and maximum period
    returns a tuple with a value between 0 and 1, and the list of related
        ciphers.
    :param split_text: A list with the text
    : param alphabet: The alphabet
    : param p_min: The minimum period
    : param p_max: The maximum period
    :return: Returns a value between 0 and 1 and a list of related ciphers
    " " "
    periods_dict = coding.statistics.period_with_ic_cols(split_text, alphabet,
        p_max * 2)
    lt = Fraction(1, 2)
    _, period_likelihood = coding.statistics.fi_likelihood(
        [periods_dict[a] for a in range(p_max + 1)], p_min, p_max, lt
    )
    if period_likelihood < 0.7:
        return Fraction(94, 100), ["caesar", "autokey-input", "autokey-output",
            "bifid", "phillips", "checkerboard",
                                    "trifid", "numbered-key", "playfair",
                                    "nihilist-transposition"]
    elif 0.7 <= period_likelihood:
        return Fraction(56, 100), ["vigenere", "bifid", "trifid",
            "nihilist-substitution"]
```

```
def h_phillips(split_text: list[str], alphabet: tuple[str]) -> tuple[Fraction,
    list[str]]:
        " " "
    Heuristic h__phillips: given the text in a list and the alphabet
    returns a tuple with a value between 0 and 1, and the list of related
        ciphers.
    :param split__text: A list with the text
    : param alphabet: The alphabet
    :return: Returns a value between 0 and 1 and a list of related ciphers
    " " "
    _, period_likelihood = coding.statistics.likely_phillips(split_text,
        alphabet)
    if period_likelihood < Fraction(12, 100):
        return Fraction(93, 100), ["numbered-key", "nihilist-substitution",
            "caesar", "autokey-output",
                                    "nihilist-transposition", "autokey-input",
                                    "vigenere", "trifid"]
    elif Fraction(12, 100) <= period_likelihood < Fraction(6, 10):
        return Fraction(6, 10), ["bifid", "playfair", "checkerboard"]
    elif Fraction(6, 10) <= period_likelihood <= Fraction(1):
        return Fraction(89, 100), ["phillips"]
def h_ncd(split_text: list[str], alphabet: tuple[str], p_min, p_max) ->
    tuple[Fraction, list[str]]:
        " " "
        Heuristic h_ncd: given the text in a list, the alphabet, the maximum and
        minimum periods,
    returns a tuple with a value between 0 and 1, and the list of related
        ciphers.
    :param split_text: A list with the text
    : param alphabet: The alphabet
    : param p_min: The minimum period
    : param p_max: The maximum period
    :return: Returns a value between 0 and 1 and a list of related ciphers
    " " "
    dict_digraphs =
        coding.statistics.calc_non_connected_digraphs_multiple_distances(split_text,
        alphabet, p_max * 2)
    df_aux = pd.DataFrame(data=list(dict_digraphs.items()), columns=["period",
        "likelihood"])
    lt = Fraction(6, 10)
    _, period_likelihood =
        coding.statistics.fi_likelihood(list(df_aux["likelihood"].values),
        p_min, p_max, lt)
    if period_likelihood == Fraction(0):
        return Fraction(6, 10), ["nihilist-substitution", "phillips",
            "autokey-input",
```

```
                                    "autokey-output", "caesar", "playfair"]
    else:
        return Fraction(73, 100), ["bifid", "trifid", "numbered-key",
            "checkerboard",
                "nihilist-substitution", "vigenere"]
class CalculatedHeuristics:
    " " "
    Object that holds the results of the computed heuristics of a cryptogram.
    " " "
    def
        __init___
        _(
            self,
            text: str,
            symbols_per_character: int,
            min_period_guess: int,
            max_period_guess: int,
            heuristics_tuple,
            original_cipher: str
    ) :
        Heuristics object. This object upon creation calculates the heuristics
        of a given text.
        The results can later be scored with the scoredHeuristics object.
        :param text: The text on which we want to calculate the heuristics
        : param symbols_per_character: The number of symbols per character we
        are considering
    : param max_period__guess: The maximum guess for the period
    : param heuristics_ciphers_dict: Dictionary that maps the heuristics to
        the ciphers they affect
    :param heuristics_tuple: The heuristics we will take into consideration
        (we may not want all of them)
    " " "
    self.original_cipher: str = original_cipher
    self.heuristics_tuple: tuple[str] = heuristics_tuple
    self.heuristics_vals: dict[str, Fraction] = {}
    self.heuristics_ciphers_affected: dict[str, list[str]] = {}
    # Variables required for multiple heuristics (so we don't repeat their
        calculation)
    cryptogram_alphabet, split_text = get_alphabet_and_split_text(text,
        symbols_per_character)
    cryptogram_char_abs_freq = calc_character_abs_frequency(split_text,
        cryptogram_alphabet)
    cryptogram_char_rel_freq =
        calc_character_rel_frequency(cryptogram_char_abs_freq)
    # HEURISTICS PRE-CALCULATION
```

```
    # index of coincidence
    ic = calc_ic(cryptogram_char_abs_freq, len(split_text))
    # HEURISTICS CALCULATION
    for heuristic in heuristics_tuple:
        val = Fraction(0)
    ciphers_affected = ()
    if heuristic == "h_alpha":
        val, ciphers_affected = h_alpha(cryptogram_alphabet)
    elif heuristic == "h_ic":
        val, ciphers_affected = h_ic(ic)
    elif heuristic == "h_ic_period":
        val, ciphers_affected = h_ic_period(split_text,
            cryptogram_alphabet, min_period_guess, max_period_guess)
    elif heuristic == "h_ncd":
        val, ciphers_affected = h_ncd(split_text, cryptogram_alphabet,
            min_period_guess, max_period_guess)
    elif heuristic == "h_phillips":
        val, ciphers_affected = h_phillips(split_text,
            cryptogram_alphabet)
    elif heuristic == "h_trans":
        val, ciphers_affected = h_trans(cryptogram_char_rel_freq,
            coding.handy_vars.ENGLISH_LETTER_FREQ)
    self.heuristics_vals[heuristic] = val
    self.heuristics_ciphers_affected[heuristic] = ciphers_affected
def print(self):
    for heuristic in self.heuristics_vals:
        print(heuristic + ":", self.heuristics_vals[heuristic],
                self.heuristics_ciphers_affected[heuristic][1])
def get_scores(
    self,
    ciphers_tuple: tuple[str],
    weights_dict: dict[str, Fraction]
) -> dict[str, Fraction]:
    " " "
    Given the tuple of ciphers and the weights, calculates the score of
        each cipher.
    :param ciphers_tuple: Tuple of ciphers
    : param weights_dict: Weights dictionary, heuristic }->\mathrm{ weight
    :return: Score of each cipher, cipher }->\mathrm{ score
    " " "
    cipher_scores = {}
```

```
    for cipher in ciphers_tuple:
    cipher_scores[cipher] = Fraction(0)
for heuristic in self.heuristics_tuple:
    cur_weight = weights_dict[heuristic]
    val_times_weight = self.heuristics_vals[heuristic] * cur_weight
    for cipher in self.heuristics_ciphers_affected[heuristic]:
            if cipher in ciphers_tuple:
                cipher_scores[cipher] += val_times_weight
    return cipher_scores
def cipher_and_calc_heuristics(
    text: str,
    symbols_per_character: int,
    min_period_guess: int,
    max_period_guess: int,
    ciphers_tuple: tuple[str],
    heuristics_tuple,
) -> list[CalculatedHeuristics]:
    " " "
    Given a text ciphers it with the given ciphers and calculates the
        heuristics for each cryptogram.
    Returns a list of said calculated heuristics.
    : param ciphers_tuple: Ciphers to be used to create ciphered texts to
        calculate heuristics with
    : param text: The text on which we want to calculate the heuristics
    : param symbols_per_character: The number of symbols per character we are
        considering
    :param min_period__guess: The maximum guess for the period
    :param max_period_guess: The maximum guess for the period
    :param heuristics_tuple: The heuristics we will take into consideration (we
        may not want all of them)
    : return: Tuple of heuristics calculated from the various cryptogram
    " " "
    cryptograms_dict = cipher_text_with_given_ciphers(text, ciphers_tuple)
    calculated_heuristics_list = []
    for cipher in ciphers_tuple:
        cryptogram = cryptograms_dict[cipher]
        calculated_heuristics_list.append(
            CalculatedHeuristics(
                text=cryptogram,
                symbols_per_character=symbols_per_character,
                min_period_guess=min_period_guess,
                max_period_guess=max_period_guess,
                heuristics_tuple=heuristics_tuple,
                original_cipher=cipher
            )
        )
    return calculated_heuristics_list
```

```
def get_more_heuristics(
    he: HInput,
    ciphers_tuple: tuple[str, ...]
) -> list[CalculatedHeuristics]:
    " " "
    Given a cleaned text to sample from, creates n samples,
    ciphers them with all the ciphers and calculates their heuristics.
    : param he: An HeuristicsInput object, containing the input necessary
    :param ciphers_tuple: Ciphers for which we will create a cryptogram
    :return: List with tuple of heuristics for each text
    " " "
    calculated_heuristics = []
    texts_to_cipher = corpus.get_random_texts_from_file(
        he.clean_corpus_path,
        he.clean_corpus_length,
        (he.min_text_length, he.max_text_length,),
        he.n_texts
    )
    for i in range(len(texts_to_cipher)):
        calculated_heuristics.extend(
            cipher_and_calc_heuristics(
                texts_to_cipher[i],
                he.symbols_per_character,
                he.min_period_guess,
                he.max_period_guess,
                    ciphers_tuple,
                he.heuristics_tuple
            )
        )
    return calculated_heuristics
```

thesis_code/classifier/input_objects.py

```
from fractions import Fraction
from typing import Callable
class SAInput:
    "|
    Object that stores the input used in the simulated annealing part of the
        algorithm.
    " " "
    def __init__(
            self,
            acceptable_fallback_ratio: Fraction,
            minimum_improvement_value: Fraction,
            max_failed_attempts_tolerated: int,
```

```
        ratings_window_size: int,
        identification_function: Callable,
        verbose: bool = False
    ):
Input for the simulated-annealing part of the algorithm
:param ratings__window_size: The size of the previous ratings window
: param acceptable_fallback__ratio: How much the weights are allowed to
    worsen in percentage
: param minimum_improvement__value: How much the weights should improve
    in percentage
: param max_failed__attempts__tolerated: The number of failed attempts at
    improving before finally stopping
" " "
# simulated annealing input
self.acceptable_fallback_ratio = acceptable_fallback_ratio
self.minimum_improvement_value = minimum_improvement_value
self.max_failed_attempts = max_failed_attempts_tolerated
self.ratings_window_size = ratings_window_size
self.identification_function = identification_function
self.verbose = verbose
class FTInput:
    " " "
    Object that stores the input used in the fine-tuning part of the algorithm.
    " " "
    def
        __init___
        (
            self,
            necessary_improvement_between_iterations: Fraction,
            max_failed_attempts: int,
            weight_increment: Fraction,
            initial_weights: dict[str, Fraction],
            verbose: bool = False
    ):
        Input for the fine-tuning (overview) of the algorithm.
        : param necessary_improvement__between__iterations: The necessary
        improvement in percentage
        that has to occur during iterations
        : param max_failed__attempts: The maximum number the algorithm will fail
        before stopping
        :param weight_increment: The increment given to the weights for the
        mutations
        : param initial__weights: The initial weights, should we want some
        " " "
    # fine-tuning input
    self.necessary_improvement_between_iterations =
        necessary_improvement_between_iterations
        self.max_failed_attempts = max_failed_attempts
        self.initial_weight_increment = weight_increment
```

```
            self.initial_weights = initial_weights
self.verbose = verbose
class HInput:
    " " "
    Object that stores the input used for the heuristics.
    " " "
    def
        __init___
        self,
        clean_corpus_path: str,
        clean_corpus_length: int,
        min_text_length: int,
        max_text_length: int,
        n_texts: int,
        symbols_per_character: int,
        min_period_guess: int,
        max_period_guess: int,
        heuristics_tuple: tuple[str, ...]
    ):
        Input for the heuristics generated.
        :param max_text_length: Maximum length for the text
        :param min_text_length: Minimum length for the text
        : param clean_corpus_length: Number of characters of the corpus
        :param clean_corpus_path: Path to corpus, after the corpus has been
        cleaned of unwanted characters
        : param n_texts: The number of texts we want to sample
        : param symbols_per_character: The number of symbols per character we
        are considering
    : param max_period_guess: The maximum guess for the period
    : param heuristics__ciphers__dict: Dictionary that maps the heuristics to
        the ciphers they affect
        :param heuristics__tuple: The heuristics we will take into consideration
        (we may not want all of them)
    " " "
    # heuristics input
    self.clean_corpus_path = clean_corpus_path
    self.clean_corpus_length = clean_corpus_length
    self.min_text_length = min_text_length
    self.max_text_length = max_text_length
    self.n_texts = n_texts
    self.symbols_per_character = symbols_per_character
    self.min_period_guess = min_period_guess
    self.max_period_guess = max_period_guess
    self.heuristics_tuple = heuristics_tuple
```

thesis_code/classifier/simulated_annealing.py

```
from __future__ import annotations
```

```
import random
import time
from fractions import Fraction
from typing import Any
from classifier.heuristics import CalculatedHeuristics, get_more_heuristics
from classifier.input_objects import SAInput, HInput, FTInput
from classifier.tuple_rating import get_weights_rating
def mutate_weights(
    weights_dict: dict[str, Fraction],
    weight_increment: Fraction,
    chosen_heuristic: str = None,
) -> dict[str, Fraction]:
    " " "
    Mutates a tuple of weights. The change can be at most the value of
        mutation_threshold.
    : param weight__increment: How much we want to increase the chosen weight
        before normalizing the weights
    :param weights_dict: Tuple with weights
    : param chosen__heuristic: The heuristic for which the weight should be
        increased; by default one is picked at random
    :return: Returns mutated weights.
    " " "
    mutated_weights_dict = weights_dict.copy()
    if chosen_heuristic is None:
        chosen_heuristic = random.choice(list(mutated_weights_dict))
    sum_to_divide_with = Fraction(1) + weight_increment
    mutated_weights_dict[chosen_heuristic] += weight_increment
    for heuristic in mutated_weights_dict.keys():
        mutated_weights_dict[heuristic] =
            Fraction(mutated_weights_dict[heuristic], sum_to_divide_with)
    return mutated_weights_dict
class RatingsWindow:
    " " "
    Object to hold the latest ratings and act as a window.
    " " "
    def __init__(self, window_size: int):
        self.window: list[Fraction, ...] = []
        self.window_size: int = window_size
        self.mean: Fraction = Fraction(0)
    def push_rating(self, new_rating):
        if len(self.window) < self.window_size:
```

```
        # we calculate like this to avoid re-adding every element to the
                mean
        # given its very time-consuming using fractions
        self.mean = (self.mean * len(self.window) + new_rating) /
        (len(self.window) + 1)
        self.window.append(new_rating)
    else:
        rating_to_be_removed = self.window[0]
        # we calculate like this to avoid re-adding every element to the
            mean
        # given its very time-consuming using fractions
        self.mean = (self.mean * self.window_size - rating_to_be_removed +
            new_rating) / self.window_size
        self.window = self.window[1:]
        self.window.append(new_rating)
    def get_mean(self) -> Fraction:
        return self.mean
    def get_last_rating(self) -> Fraction | None:
        if self.window is []:
            return None
        else:
            if len(self.window) == 0:
            return Fraction(0)
        else:
            return self.window[len(self.window) - 1]
    def is_the_window_full(self) -> bool:
    return len(self.window) >= self.window_size
    def print(self):
        print("window of size", self.window_size, ":", self.window)
        print(self.mean)
def is_within_afr_or_better(
        acceptable_fallback_ratio: Fraction,
        current_rating: Fraction,
        previous_rating: Fraction
) -> bool:
    " " "
    Checks if the rating is within the acceptable fallback ratio.
    : param acceptable_fallback__ratio: The value of the acceptable fallback ratio
    :param current__rating: The current rating
    : param previous__rating: The previous rating
    :return: True if is within, false otherwise
    " " "
    if previous_rating == 0:
        return True
    elif current_rating > previous_rating:
```

```
        return True
    else:
        return acceptable_fallback_ratio >= (1 - current_rating /
            previous_rating)
def is_within_miv(
        minimum_improvement_value: Fraction,
        current_rating: Fraction,
        ratings_window: RatingsWindow
) -> bool:
    " " "
    Checks if the current rating is within the minimum improvement value.
    : param minimum_improvement__value: The minimum improvement value
    : param current__rating: The current rating
    :param ratings_window: The window where the previous ratings are stored
    :return: True if is within, false otherwise
    " " "
    return minimum_improvement_value <= current_rating -
        ratings_window.get_mean()
def simulated_annealing(
        calculated_heuristics_list: [CalculatedHeuristics, ...],
        current_weight_increment: Fraction,
        ciphers_tuple: tuple[str, ...],
        initial_weights: dict[str, Fraction],
        initial_rating: Fraction,
        sa: SAInput,
) -> tuple[dict[str, Fraction], Fraction]:
    " " "
    Does the simulated annealing part of the algorithm to find working weights
        for the classifier.
    :param sa: Input object for the simulated-annealing with the necessary input
    :param initial__rating: The initial rating for the weights, so that we can
        have a minimum to start from
    : param current__weight__increment: Increment to be applied during the weight
        mutations
    :param calculated__heuristics__list: A list of pre-calculated heuristics over
        cryptograms
    :param ciphers_tuple: A tuple of the ciphers to take into consideration
    :param initial__weights: The initial weights for the simulated annealing
    :return: The resulting weights and their rating
    " " "
    ratings_window = RatingsWindow(sa.ratings_window_size)
    current_weights = initial_weights
    current_rating = initial_rating
    if sa.verbose:
        print("initial weights for simulated annealing:")
        for heuristic in initial_weights:
            print(initial_weights[heuristic], heuristic)
```

```
    print("==" * 20)
# Failed attempts = 0
failed_attempts = 0
while True:
    # getting new weights through mutation
    new_weights = mutate_weights(current_weights, current_weight_increment)
    # calculating the new rating (identification attempt)
    new_rating = get_weights_rating(
        new_weights, calculated_heuristics_list, ciphers_tuple,
            sa.identification_function
    )
    if sa.verbose:
        for heuristic in new_weights.keys():
            print(new_weights[heuristic], heuristic)
        print("rating: ", new_rating, float(new_rating))
    # does the new set improve identification or is within an acceptable
        fallback?
    within_afr = is_within_afr_or_better(sa.acceptable_fallback_ratio,
        new_rating, ratings_window.get_last_rating())
    if sa.verbose: print("AFR", within_afr)
    if within_afr:
        # if it does improve identification then...
        current_weights = new_weights
        current_rating = new_rating
        failed_attempts = 0
        if sa.verbose: print("FAILED =0")
        # is the new set within the minimum improvement value?
        within_miv = is_within_miv(sa.minimum_improvement_value,
            new_rating, ratings_window)
        window_full = ratings_window.is_the_window_full()
        if sa.verbose: print("WINDOW FULL AND NOT MIV", window_full and not
            within_miv)
        if window_full and not within_miv:
                if sa.verbose: print("SA STOPPED HERE")
                return current_weights, current_rating
        else:
            ratings_window.push_rating(new_rating)
    else:
        # if it does NOT improve identification then...
        failed_attempts += 1
        if sa.verbose: print("FAILED +1")
        # has the number of attempts been exceeded?
        if failed_attempts > sa.max_failed_attempts:
                return current_weights, current_rating
```

```
    if sa.verbose: print("==" * 20)
def weight_training(
    ft: FTInput,
    sa: SAInput,
    he: HInput,
    ciphers_tuple: tuple[str, ...],
) -> tuple[dict[str, Fraction], Fraction, list[Any], float]:
    " " "
    Trains a tuple of weights
    :param ft: The fine-tuning input
    : param sa: The simulated annealing input
    : param he: The heuristics input
    :param ciphers_tuple: The tuple of ciphers
    :return: Returns tuple with weights, their rating, the list of computed
        heuristics objects and the elapsed time
    " " "
    start_time = time.time()
    current_weights = ft.initial_weights
    print(current_weights)
    current_rating = Fraction(0)
    current_weight_increment = ft.initial_weight_increment
    failed_attempts = 0
    calculated_heuristics_list = get_more_heuristics(he, ciphers_tuple)
    while True:
        if ft.verbose:
            print("CURRENT RATING:", float(current_rating))
        new_weights, new_rating = simulated_annealing(
            calculated_heuristics_list,
            current_weight_increment,
            ciphers_tuple,
            current_weights,
            current_rating,
            sa
    )
    # checking to see if the rating has improved enough
    if new_rating >= current_rating +
            ft.necessary_improvement_between_iterations:
            # if there was enough improvement...
            current_weights = new_weights
            current_rating = new_rating
            calculated_heuristics_list += get_more_heuristics(he, ciphers_tuple)
            failed_attempts = 0
        else:
            # if there was not enough improvement...
```

```
failed_attempts += 1
if failed_attempts >= ft.max_failed_attempts:
    elapsed_time = time.time() - start_time
    calculated_heuristics_list_actually_used =
        calculated_heuristics_list[:-he.n_texts *
        len(ciphers_tuple)]
    return current_weights, current_rating,
        calculated_heuristics_list_actually_used, elapsed_time
else:
    current_weight_increment = Fraction(current_weight_increment,
thesis_code/classifier/tuple_rating.py
```

from fractions import Fraction
from typing import Callable
from classifier.heuristics import CalculatedHeuristics
from coding.tools import get_reverse_dict
def id_1(original_cipher: str, scores: dict[str, Fraction]) -> Fraction:
The first evaluation function presented in the thesis.
Only checks if the original cipher is one with the highest score, does not
take ties into account.
:param original__cipher: The original cipher
:param scores: The scores of each cipher
:return: 1 if the highest score is the original cipher, 0 otherwise
" " "
max_score = max(scores.values())
list_of_keys = list()
\# Iterate over all the items in dictionary to find keys with max value
for cipher, value in scores.items():
if value == max_score:
list_of_keys.append(cipher)
if len(list_of_keys) == 1:
if list_of_keys[0] == original_cipher:
\# print("highest score cipher:", list_of_keys[0])
return Fraction(1)
return Fraction(0)
def id_2(original_cipher: str, scores: dict[str, Fraction]) -> Fraction:
" " "
Improved evaluation function.
Calculates the 1 - (cipher_rank / how_many_ciphers_there_are).
This means that the best returned value possible is 1, and the worst is 1 /
how__many__ciphers_there__are.
:param original__cipher: The original cipher

```
```

    :param scores: The scores of each cipher
    :return: The score as explained, 1 - (cipher__rank /
        how__many__ciphers_there__are)
    " " "
    ranks = get_ranking(scores)
    position = ranks[original_cipher]
    return 1 - Fraction(position, len(scores.keys()))
    def apply_identification_function(
heuristics: CalculatedHeuristics,
scores: dict[str, Fraction],
identification_function: Callable
) -> Fraction:
" " "
Calculates how good the weights are using the evaluation function passed.
:param identification_function: Chosen evaluation function for the scores
:param scores: The score of each cipher
: param heuristics: Heuristics object
:return: Value returned by the evaluation function for the computed scores
" " "
return identification_function(heuristics.original_cipher, scores)
def get_weights_rating(
weights_dict: dict[str, Fraction],
calculated_heuristics_list: list[CalculatedHeuristics],
ciphers_tuple: tuple[str],
identification_function
) -> Fraction:
" " "
Given the weights for the heuristics and a list of tuples of heuristics for
each cryptogram
returns the number of correct cryptograms guessed out of the given
cryptograms
(each cryptogram heuristic is a cryptogram).
:param weights_dict: Dictionary of the weights of the heuristics
:param calculated_heuristics_list: List of heuristics
:param ciphers_tuple: Ciphers for which we will try to guess
:param identification_function: Chosen evaluation function for the scores
:return: Score of the set of weights in the given heuristics
" " "
id_function_result = 0
for calculated_heuristics in calculated_heuristics_list:
scores = calculated_heuristics.get_scores(ciphers_tuple, weights_dict)
id_function_result +=
apply_identification_function(calculated_heuristics, scores,
identification_function)

```
```

    return Fraction(id_function_result, len(calculated_heuristics_list))
    def get_ranking(scores_dict: dict[str, Fraction]) -> dict[str, int]:
" " "
Given a dictionary with scores of each cipher returns the ranking of each
cipher,
according to the definition of rank given in the thesis.
:param scores_dict: The dictionary with the scores of the ciphers, Cipher
-score
:return: A dictionary with the ranking of each cipher, Cipher }->\mathrm{ Rank
" " "
sorted_scores = list(set(scores_dict.values()))
sorted_scores.sort(reverse=True)
inv_dict = get_reverse_dict(scores_dict)
ranks = {}
cur_rank = 0
for score in sorted_scores:
ciphers_with_this_score = inv_dict[score]
for cipher in ciphers_with_this_score:
ranks[cipher] = cur_rank + len(ciphers_with_this_score) - 1
cur_rank += len(ciphers_with_this_score)
return ranks

```
thesis_code/coding/cipher.py
```

import collections
import random
from copy import deepcopy
from typing import Optional
from coding.polybius import Polybius, Polybius3d
from coding.tools import closest_perfect_square, split_text_into_list
def caesar_cipher(text: str, alphabetic_key: tuple[str, ...], shift: int) ->
str:
" " "
Given a text returns the text ciphered with the caesar cipher.
To decipher instead of the shift give it the length of the alphabet -
encryption shift
:param text: Text to be ciphered/deciphered
:param alphabetic_key: Alphabet order
:param shift: Shift
:return: Ciphered/Deciphered text
" " "

```
```

    ciphered = ""
    text_list = split_text_into_list(text, alphabetic_key)
    for character in text_list:
        ciphered_index = (alphabetic_key.index(character) + shift) %
            len(alphabetic_key)
        ciphered = ciphered + alphabetic_key[ciphered_index]
    return ciphered
    def substitution_cipher(text: str, mapping: {str}) -> str:
" " "
Given a dictionary, with a mapping [character }->\mathrm{ substitution
character(s)], applies it.
:param text: Text to be ciphered/deciphered
:param mapping: Dictionary with mapping
:return: Ciphered text
" " "
return ''.join([mapping[letter] for letter in split_text_into_list(text,
mapping.keys())])
def substitution_decipher(text: str, inv_mapping: {str}) -> str:
" " "
Given a dictionary, with a mapping [substitution character(s) ->
character], applies it.
: param text: Text to decipher
:param inv_mapping: Dictionary with mapping
: return: Deciphered text
" " "
return ''.join([inv_mapping[letter] for letter in
split_text_into_list(text, inv_mapping.keys())])
def homophonic_substitution_cipher(text: str, mapping: {set}) -> str:
" " "
Substitution cipher that happens to be homophonic.
:param text: Text to cipher
:param mapping: Dictionary with mapping [character }->\mathrm{ substitution
characters]
:return: Ciphered text
" " "
ciphered = ""
text_list = split_text_into_list(text, mapping.keys())
for character in text_list:
s = mapping[character]
ciphered += random.sample(s, 1)[0]
return ciphered

```
```

def homophonic_substitution_decipher(text: str, inv_mapping: {str}) -> str:
" " "
Substitution cipher that happens to be homophonic.
: param text: Text to decipher
: param inv_mapping: Dictionary with mapping [substitution characters }
character]
:return: Deciphered text
" " "
ciphered = ""
text_list = split_text_into_list(text, inv_mapping)
for character in text_list:
ciphered += inv_mapping[character]
return ciphered
def vigenere_cipher_character(a: str, b: str, alphabetic_key: tuple[str, ...])
-> str:
" " "
Given 2 characters and an alphabetic key returns the ciphered result.
To decipher instead of b give it the character in
[length of the alphabetic key - index of b]
:param a: Plaintext letter
:param b: Key letter
: param alphabetic__key: Alphabet order
:return: Ciphered/Deciphered letter
" " "
a_index = alphabetic_key.index(a)
b_index = alphabetic_key.index(b)
c_index = (a_index + b_index) % len(alphabetic_key)
return alphabetic_key[c_index]
def vigenere_cipher(text: str, alphabetic_key: tuple[str, ...], key: tuple[str,
...]) -> str:
" " "
Ciphers the text with a generic vigenere
:param text: The text to cipher
: param alphabetic__key: Alphabet order
: param key: Key used to cipher in a list format
:return: Ciphered text
" " "
ciphered = ""
text_list = split_text_into_list(text, alphabetic_key)
i = 0
for text_character in text_list:
key_character = key[i % len(key)]
ciphered += vigenere_cipher_character(text_character, key_character,
alphabetic_key)
i += 1

```
```

    return ciphered
    def vigenere_decipher(text: str, alphabetic_key: tuple[str, ...], key:
tuple[str, ...]) -> str:
" " "
Deciphers the text with a generic vigenere.
:param text: The text to decipher
:param alphabetic_key: Alphabet order
: param key: Key used to cipher in a list format
: return: Deciphered text
" " "
deciphered = ""
text_list = split_text_into_list(text, alphabetic_key)
i = 0
for text_character in text_list:
key_character = key[i % len(key)]
\# necessary to decipher instead of ciphering
key_character_inv = alphabetic_key[
(len(alphabetic_key) - alphabetic_key.index(key_character)) %
len(alphabetic_key)
]
deciphered += vigenere_cipher_character(text_character,
key_character_inv, alphabetic_key)
i += 1
return deciphered
def autokey_output_fed_cipher(text: str, alphabetic_key: tuple[str, ...], key:
tuple[str]) -> str:
" " "
Ciphers the text using vigenere autokey ciphertext fed.
:param text: The text to cipher
: param alphabetic__key: Alphabet order
:param key: Initial key used to cipher
:return: Ciphered text
" " "
ciphered = []
text_list = split_text_into_list(text, alphabetic_key)
buffered_key = collections.deque(key)
for character in text_list:
c = vigenere_cipher_character(character, buffered_key[0],
alphabetic_key)
ciphered.append(c)
buffered_key.append(c)
buffered_key.popleft()

```
```

    return ''.join(ciphered)
    def autokey_output_fed_decipher(text: str, alphabetic_key: tuple[str, ...],
key: tuple[str]) -> str:
" " "
Deciphers the text using vigenere autokey ciphertext fed.
:param text: The text to decipher
: param alphabetic__key: Alphabet order
:param key: Initial key used to cipher
:return: Plaintext with deciphered text
" " "
deciphered = []
text_list = split_text_into_list(text, alphabetic_key)
buffered_key = collections.deque(key)
for character in text_list:
key_character_inv = alphabetic_key[
(len(alphabetic_key) - alphabetic_key.index(buffered_key[0])) %
len(alphabetic_key)
]
d = vigenere_cipher_character(character, key_character_inv,
alphabetic_key)
deciphered.append(d)
buffered_key.append(character)
buffered_key.popleft()
return ''.join(deciphered)
def autokey_input_fed_cipher(text: str, alphabetic_key: tuple[str, ...], key:
tuple[str]) -> str:
" " "
Ciphers the text using vigenere autokey plaintext fed.
:param text: The text to cipher
: param alphabetic__key: Alphabet order
:param key: Beginning of the key used to cipher
:return: Ciphertext with ciphered text
" " "
ciphered = []
text_list = split_text_into_list(text, alphabetic_key)
buffered_key = collections.deque(key)
i = 0
for character in text_list:
c = vigenere_cipher_character(character, buffered_key[0],
alphabetic_key)
ciphered.append(c)
buffered_key.append(text_list[i])

```
```

        buffered_key.popleft()
        i += 1
    return ''.join(ciphered)
    def autokey_input_fed_decipher(text: str, alphabetic_key: tuple[str, ...], key:
tuple[str]) -> str:
" " "
Deciphers the text using vigenere autokey plaintext fed.
: param text: The text to decipher
: param alphabetic__key: Alphabet order
: param key: Beginning of the key used to cipher
return: Plaintext with deciphered text
" " "
deciphered = []
text_list = split_text_into_list(text, alphabetic_key)
buffered_key = collections.deque(key)
for character in text_list:
key_character_inv = alphabetic_key[
(len(alphabetic_key) - alphabetic_key.index(buffered_key[0])) %
len(alphabetic_key)
]
d = vigenere_cipher_character(character, key_character_inv,
alphabetic_key)
deciphered.append(d)
buffered_key.append(d)
buffered_key.popleft()
return ''.join(deciphered)
def __bifid_periodless_cipher(text: str, polybius: Polybius) -> str:
" " "
Given a string ciphers the string with the periodless bifid cipher.
:param text: The text to cipher
:param polybius: Tuple containing the polybius in dictionaries (row and
column)
: return: The ciphered text
" " "
size = len(text)
ciphered = []
for i in range(0, len(text) * 2, 2):
x = polybius.t[text[i % size]][i // size]
y = polybius.t[text[(i + 1) % size]][(i + 1) // size]
ciphered.append(polybius.t_inv[(x, y)])
return ''.join(ciphered)
def _bifid_periodless_decipher(text: str, polybius: Polybius) -> str:

```
```

    " " "
    Given a string deciphers the string with the periodless bifid cipher.
    :param text: The text to decipher
    :param polybius: Tuple containing the polybius in dictionaries (row and
        column)
    :return: The deciphered text
    " " "
    deciphered = []
    size = len(text)
    for i in range(0, len(text)):
        x = polybius.t[text[i // 2]][i % 2]
        y = polybius.t[text[(i + size) // 2]][(i + size) % 2]
    deciphered.append(polybius.t_inv[(x, y)])
    return ''.join(deciphered)
    def bifid_cipher(text: str, polybius: Polybius, period: int) -> str:
" " "
Ciphers the text using the bifid cipher.
:param text: The text to cipher
:param polybius: Tuple containing the polybius in dictionaries (row and
column)
: param period: The cipher's period
:return: The ciphered text
" " "
last_block = len(text) % period
ciphered_text = ""
for p in range(0, len(text) - last_block, period):
ciphered_text += _bifid_periodless_cipher(text[p:(p + period)],
polybius)
if last_block > 0:
ciphered_text += _bifid_periodless_cipher(text[-last_block:], polybius)
return ciphered_text
def bifid_decipher(text: str, polybius: Polybius, period: int) -> str:
" " "
Deciphers the text using the bifid cipher.
:param text: The text to decipher
:param polybius: Tuple containing the polybius in dictionaries (row and
column)
: param period: The cipher's period
:return: The deciphered text
" " "
deciphered_text = ""
last_block = len(text) % period
for p in range(0, len(text) - last_block, period):
deciphered_text += _bifid_periodless_decipher(text[p:(p + period)],
polybius)
if last_block > 0:
deciphered_text += _bifid_periodless_decipher(text[-last_block:],

```
```

    polybius)
    return deciphered_text
    def _phillips_periodless_cipher(text: str, polybius: Polybius, dig_val: int) ->
str:
" " "
Given a string ciphers/deciphers the string with the phillips cipher.
:param text: The text to cipher/decipher
:param polybius: Tuple containing the polybius in dictionaries (row and
column)
:param dig_val: 1 if ciphering, -1 if deciphering
:return: The ciphered/deciphered text
" " "
ciphered = []
for i in range(0, len(text)):
letter_xy = polybius.t[text[i]]
ciphered_letter_xy = ((letter_xy[0] + dig_val) % 5, (letter_xy[1] +
dig_val) % 5)
ciphered.append(polybius.t_inv[ciphered_letter_xy])
return ''.join(ciphered)
def _phillips_move_row_down(polybius: Polybius, row: int) -> None:
" " "
Given a polybius table made with two dictionaries and given a row, lowers
that row once.
: param row: The row to be moved down
" " "
x = row % 4
for }y\mathrm{ in range(0, 5):
letral = polybius.t_inv[(x, y)]
letra2 = polybius.t_inv[((x + 1) % 5, y)]
polybius.t_inv[(x, y)] = letra2
polybius.t_inv[((x + 1) % 5, y)] = letral
polybius.t[letra1] = ((x + 1) % 5, y)
polybius.t[letra2] = (x, y)
def phillips_cipher(text: str, polybius: Polybius, period: int) -> str:
polybius2 = deepcopy(polybius)
ciphered_text = ""
row = 0
last_row_to_switch = 4
counter = 1
for }p\mathrm{ in range(0, len(text), period):
ciphered_text += _phillips_periodless_cipher(text[p:p + period],
polybius2, 1)
_phillips_move_row_down(polybius2, row)

```
```

        row += 1
        counter += 1
        if counter % 8 == 1: # 8 is the number of tables
        polybius2 = deepcopy(polybius)
        last_row_to_switch = 4
        row = 0
    if last_row_to_switch == 0:
last_row_to_switch = 4
if row == last_row_to_switch:
row = 0
last_row_to_switch = last_row_to_switch - 1
return ciphered_text
def phillips_decipher(text: str, polybius: Polybius, period: int) -> str:
polybius2 = deepcopy(polybius)
deciphered_text = ""
row = 0
last_row_to_switch = 4
counter = 1
for }p\mathrm{ in range(0, len(text), period):
deciphered_text += _phillips_periodless_cipher(text[p:p + period],
polybius2, -1)
_phillips_move_row_down(polybius2, row)
row += 1
counter += 1
if counter % 8 == 1: \# 8 is the number of tables
polybius2 = deepcopy(polybius)
last_row_to_switch = 4
row = 0
if last_row_to_switch == 0:
last_row_to_switch = 4
if row == last_row_to_switch:
row = 0
last_row_to_switch = last_row_to_switch - 1
return deciphered_text
def checkerboard_cipher(text: str, polybius: Polybius) -> str:
" " "
Given a string ciphers the string with the checkerboard cipher.
:param text: The text to cipher
:param polybius: Tuple containing the polybius in dictionaries (row and
column)
: return: The ciphered text
" " "
def get_checkerboard_indexes(horizontal_keywords: [str], vertical_keywords:
[str]) -> tuple:
k_letters_row = {}
k_letters_col = {}

```
```

    for i in range(5):
        letter_list = [word[i] for word in horizontal_keywords]
        k_letters_col[i] = letter_list
    for i in range(5):
        letter_list = [word[i] for word in vertical_keywords]
        k_letters_row[i] = letter_list
    return k_letters_col, k_letters_row
checkerboard_indexes = get_checkerboard_indexes(polybius.col_words,
polybius.row_words)
keyword_letters_col = checkerboard_indexes[0]
keyword_letters_row = checkerboard_indexes[1]
ciphered = []
for letter in text:
(x, y) = polybius.t[letter]
ciphered.append(random.choice(keyword_letters_row[x]))
ciphered.append(random.choice(keyword_letters_col[y]))
return ''.join(ciphered)
def checkerboard_decipher(text: str, polybius: Polybius) -> str:
" " "
Given a string deciphers the string with the checkerboard cipher.
: param text: The text to decipher
:param polybius: Tuple containing the polybius in dictionaries (row and
column)
:return: The deciphered text
" " "
def get_checkerboard_indexes_inv(horizontal_keywords: [str],
vertical_keywords: [str]) -> tuple:
k_letters_row_inv = {}
k_letters_col_inv = {}
for i in range(5):
letter_list = [word[i] for word in horizontal_keywords]
for letter in letter_list:
k_letters_col_inv[letter] = i
for i in range(5):
letter_list = [word[i] for word in vertical_keywords]
for letter in letter_list:
k_letters_row_inv[letter] = i
return k_letters_col_inv, k_letters_row_inv
checkerboard_indexes_inv = get_checkerboard_indexes_inv(polybius.col_words,
polybius.row_words)
keyword_letters_col_inv = checkerboard_indexes_inv[0]
keyword_letters_row_inv = checkerboard_indexes_inv[1]
deciphered = []
for i in range(0, len(text), 2):
letter_for_row = text[i]
letter_for_col = text[i + 1]

```
```

        row = keyword_letters_row_inv[letter_for_row]
    col = keyword_letters_col_inv[letter_for_col]
deciphered.append(polybius.t_inv[(row, col)])
return ''.join(deciphered)
def _trifid_periodless_cipher(text: str, polybius: Polybius3d) -> str:
" " "
Given a string ciphers the string with the periodless trifid cipher
:param text: The text to cipher
:param polybius: 3 dimensional polybius
: return: The ciphered text
" " "
size = len(text)
ciphered = []
for i in range(0, len(text) * 3, 3):
x = polybius.t[text[i % size]][i // size]
y = polybius.t[text[(i + 1) % size]][(i + 1) // size]
z = polybius.t[text[(i + 2) % size]][(i + 2) // size]
ciphered.append(polybius.t_inv[(x, y, z)])
return ''.join(ciphered)
def _trifid_periodless_decipher(text: str, polybius: Polybius3d) -> str:
" " "
Given a string deciphers the string with the periodless trifid cipher.
:param text: The text to decipher
:param polybius: 3 dimensional polybius
:return: The deciphered text
" " "
deciphered = []
size = len(text)
for i in range(0, len(text)):
x = polybius.t[text[i // 3]][i % 3]
y = polybius.t[text[(i + size) // 3]][(i + size) % 3]
z = polybius.t[text[(i + 2 * size) // 3]][(i + 2 * size) % 3]
deciphered.append(polybius.t_inv[(x, y, z)])
return ''.join(deciphered)
def trifid_cipher(text: str, polybius: Polybius3d, period: int) -> str:
" " "
Ciphers the text using the trifid cipher.
: param text: The text to cipher
:param polybius: 3 dimensional polybius
: param period: The cipher's period
:return: Ciphertext with ciphered text
" " "
last_block = len(text) % period
ciphered_text = ""
for p in range(0, len(text) - last_block, period):

```
```

        ciphered_text += _trifid_periodless_cipher(text[p:(p + period)],
        polybius)
    if last_block > 0:
        ciphered_text += _trifid_periodless_cipher(text[-last_block:], polybius)
    return ciphered_text
    def trifid_decipher(text: str, polybius: Polybius3d, period: int) -> str:
" " "
Deciphers the text using the trifid cipher.
:param text: The text to decipher
:param polybius: 3 dimensional polybius
: param period: The cipher's period
:return: The deciphered text
" " "
deciphered_text = ""
last_block = len(text) % period
for p in range(0, len(text) - last_block, period):
deciphered_text += _trifid_periodless_decipher(text[p:(p + period)],
polybius)
if last_block > 0:
deciphered_text += _trifid_periodless_decipher(text[-last_block:],
polybius)
return deciphered_text
def numbered_key_cipher(text: str, key: tuple) -> str:
" " "
Ciphers the text using the numbered key cipher.
: param text: The text to cipher
:param key: Tuple with the dictionaries
:return: The ciphered text
" " "
ciphered = []
t = key[0]
for letter in text:
numbers = t[letter]
chosen_number = random.choice(numbers)
if chosen_number < 10:
ciphered.append(str(0))
ciphered.append(str(chosen_number))
return ''.join(ciphered)
def numbered_key_decipher(text: str, key: tuple) -> str:
" " "
Deciphers the text using the numbered key cipher.
:param text: The text to decipher
:param key: Tuple with the dictionaries
:return: The deciphered text
" " "

```
```

    deciphered = []
    t_inv = key[1]
    for i in range(0, len(text), 2):
        number = int(text[i:i + 2])
        deciphered.append(t_inv[number])
    return ''.join(deciphered)
    def playfair_cipher(text: str, polybius: Polybius, blank_char: str) -> str:
" " "
Ciphers with the playfair cipher. The filler char is used to separate pairs
of equal letters.
:param text: Text to cipher
:param polybius: Tuple containing the polybius in dictionaries (row and
column)
:param blank_char: Char to use to separate equal letters
: return: Ciphered text
" " "
ciphered = ""
for i in range(0, len(text) - 1, 2):
if text[i] == text[i + 1]:
text = text[:i + 1] + blank_char + text[i + 1:]
if len(text) % 2 != 0:
text += blank_char
for i in range(0, len(text), 2):
coord1 = polybius.t[text[i]]
coord2 = polybius.t[text[i + 1]]
if coord1[0] == coord2[0]:
ciphered += polybius.t_inv[(coord1[0], (coord1[1] + 1) % 5)]
ciphered += polybius.t_inv[(coord2[0], (coord2[1] + 1) % 5)]
elif coord1[1] == coord2[1]:
ciphered += polybius.t_inv[((coord1[0] + 1) % 5, coord1[1])]
ciphered += polybius.t_inv[((coord2[0] + 1) % 5, coord2[1])]
else:
ciphered += polybius.t_inv[(coord1[0], coord2[1])]
ciphered += polybius.t_inv[(coord2[0], coord1[1])]
return ciphered
def playfair_decipher(text: str, polybius: Polybius) -> str:
" " "
Decipher with the playfair decipher. There can be some residue from the
filler characters.
:param text: Text to decipher
:param polybius: Tuple containing the polybius in dictionaries (row and
column)
: return: Deciphered text

```
```

    " " "
    deciphered = ""
    for i in range(0, len(text), 2):
        coord1 = polybius.t[text[i]]
        coord2 = polybius.t[text[i + 1]]
        if coord1[0] == coord2[0]:
        deciphered += polybius.t_inv[(coord1[0], (coord1[1] - 1) % 5)]
        deciphered += polybius.t_inv[(coord2[0], (coord2[1] - 1) % 5)]
    elif coord1[1] == coord2[1]:
        deciphered += polybius.t_inv[((coord1[0] - 1) % 5, coord1[1])]
        deciphered += polybius.t_inv[((coord2[0] - 1) % 5, coord2[1])]
    else:
        deciphered += polybius.t_inv[(coord1[0], coord2[1])]
        deciphered += polybius.t_inv[(coord2[0], coord1[1])]
    return deciphered
    def nihilist_substitution_cipher(text: str, polybius: Polybius, key: tuple[str,
...]) -> str:
" " "
Ciphers using the nihilist substitution cipher.
:param text: Text to cipher
:param polybius: Polybius to be used
:param key: Keyword to be used
:return: Ciphered text
" " "
def add_indexes(first: tuple[int, int], second: tuple[int, int]) -> str:
" " "
Adds indexes given as tuples. Also transforms 105 to 05, for example,
during the calculations.
:param first: First tuple
:param second: Second tuple
: return: Tuple resultant of addition
" " "
first = int(str(first[0]) + str(first[1]))
second = int(str(second[0]) + str(second[1]))
r = first + second
if r >= 100:
r_aux = str(r)
return r_aux[1:
elif r < 10:
return "0" + str(r)
else:
return str(r)
key_indexes = [(polybius.t[letter][0] + 1, polybius.t[letter][1] + 1) for
letter in key]

```
```

    ciphered_list = [add_indexes((polybius.t[text[i]][0] + 1,
        polybius.t[text[i]][1] + 1), key_indexes[i % len(key)])
            for i in range(len(text))]
    return "".join(ciphered_list)
    def nihilist_substitution_decipher(text: str, polybius: Polybius, key:
tuple[str, ...]) -> str:
" " "
Deciphers with the nihilist substitution decipher.
: param text: Text to be deciphered
:param polybius: Polybius used to cipher
: param key: Keyword used to cipher
: return: Deciphered text
" " "
def sub_indexes(first: tuple[int, int], second: tuple[int, int]) ->
tuple[int, int]:
" " "
Subtracts indexes given as tuples. Also transforms 05 to 105, for
example, during the calculations.
: param first: First tuple
:param second: Second tuple
:return: Tuple resultant of the subtraction
" " "
\# transforming to ints
first = int(str(first[0]) + str(first[1]))
second = int(str(second[0]) + str(second[1]))
\# checking if bigger than 100, since 105, for example, would be
ciphered to 05 instead of 105
if first <= 10:
first = int("10" + str(first))
r = first - second
\# transforming back to tuples of coordinates
real_indexes = int(str(r) [0]), int(str(r) [1])
return real_indexes
key_indexes = [polybius.t[letter] for letter in key]
deciphered = ""
for i in range(0, len(text), 2):
coord = int(text[i]), int(text[i + 1])
k = key_indexes[(i // 2) % len(key)]
k = (k[0] + 1, k[1] + 1)
indexes = sub_indexes(coord, k)
indexes = indexes[0] - 1, indexes[1] - 1

```
```

        deciphered += polybius.t_inv[indexes]
    return deciphered
    def nihilist_transposition_cipher(text: str, alphabet: tuple[str, ...], key:
tuple[int, ...],
blank_character: Optional[str] = None) -> str:
" " "
Ciphers using the transposition cipher.
: param alphabet: The alphabet to be used. All the characters should have
the same number of symbols.
:param text: The text to cipher/decipher
:param key: The key to be used - should be an array of integers in some
order from 0 to square__len - 1
: param blank__character: Character to fill the rest of the message with, to
the size of the square - by default it's filled with random characters
:return: The resulting text
" " "
import math
text_list = split_text_into_list(text, alphabet)

# adding extra chars to the text if needed

n_missing_chars = closest_perfect_square(len(text_list)) - len(text_list)
extra_chars = []
if n_missing_chars > 0:
if blank_character is None:
extra_chars = [random.choice(alphabet) for _ in
range(n_missing_chars)]
else:
extra_chars = [blank_character for _ in range(n_missing_chars)]
text_list = text_list + extra_chars
square_side_len = math.ceil(math.sqrt(len(text_list)))

# ordering by rows

ordered_by_rows = [text_list[row_number * square_side_len:(row_number + 1)
* square_side_len] for row_number in key]

# ordering by cols

ordered_by_rows_and_cols = [[] for _ in range(square_side_len)]
for col_number in range(square_side_len):
for row in ordered_by_rows:
ordered_by_rows_and_cols[col_number].append(row[key[col_number]])

# putting everything together

ciphered = ""
for i in range(square_side_len):
for j in range(square_side_len):
ciphered += ordered_by_rows_and_cols[j][i]

```
```

    return ciphered
    def nihilist_transposition_decipher(text: str, alphabet: tuple[str, ...], key:
tuple[int, ...]) -> str:
" " "
Deciphers using the transposition cipher.
: param alphabet: The alphabet to be used. All the characters should have
the same number of symbols.
:param text: The text to cipher/decipher
:param key: The key to be used - should be an array of integers in some
order from 0 to square_len - 1
:return: The resulting text
" " "
import math
text_list = split_text_into_list(text, alphabet)
square_side_len = math.ceil(math.sqrt(len(text_list)))
\# ordering by rows
ordered_by_rows = [text_list[row_number * square_side_len:(row_number + 1)
* square_side_len] for row_number in key]
\# ordering by cols
ordered_by_rows_and_cols = [[] for _ in range(square_side_len)]
for col_number in range(square_side_len):
for row in ordered_by_rows:
ordered_by_rows_and_cols[col_number].append(row[key[col_number]])
\# putting everything together
ciphered = ""
for i in range(square_side_len):
for j in range(square_side_len):
ciphered += ordered_by_rows_and_cols[j][i]
return ciphered

```
thesis_code/coding/corpus.py
```

import os
import random
def clean_text(
text: str, alphabet: tuple[str]
) -> str:
Given a text and an alphabet returns the text in lowercase and removes any
character not in the alphabet.
:param text: The text to clean
:param alphabet: The accepted alphabet

```
```

    :return: The clean text
    " " "
    clean = [item.lower() for item in text]
    clean = [c for c in clean if c in alphabet]
    return ''.join(clean)
    def clean_corpus(
corpus_path: str, alphabet: tuple[str], clean_corpus_path: str
) -> int:
" " "
Given the path for a file with a corpus, cleans it with the given alphabet
[see clean_text()] and outputs it to
the given output path. Returns the number of characters in the new file.
: param corpus__path: Path for the input file
:param alphabet: Alphabet to clean corpus with
: param clean__corpus__path: Path for the output
:return: The number of characters in the outputted file
" " "
if os.path.isfile(corpus_path):
\# reading the original text
input_file = open(corpus_path, "r")
input_text = input_file.read()
\# cleaning the text
output_text = clean_text(input_text, alphabet)
\# writing to file
output_file = open(clean_corpus_path, "w")
output_file.write(output_text)
output_file.close()
return len (output_text)
else:
raise Exception("File not found!")
def get_random_texts_from_file(
file_path: str, file_length: int, text_length: int | tuple[int, int] |
list[int, int], n_texts: int = 1
) -> dict[int, str]:
" " "
Given the path to a file returns a list of random strings, with the given
parameters.
:param file_path: The path to the file
:param file_length: The length of the file
:param text__length: The length of the text, can be a single int,
or two ints in a tuple for a random length within that interval
:param n_texts: Number of texts
:return: List with texts
" " "
text_list = {}

```
```

    with open(file_path, "r") as text_file:
        for i in range(n_texts):
        # getting the text length
        if type(text_length) is int:
                sample_length = text_length
        elif type(text_length) is list:
                sample_length = random.randint(text_length[0], text_length[1])
        elif type(text_length) is tuple:
            sample_length = random.randint(text_length[0], text_length[1])
        # getting a random place to start the text
        start_index = random.randint(0, file_length - sample_length)
        end_index = start_index + sample_length
        # getting the text and appending to the list
        text_file.seek(start_index)
        text_sample = text_file.read(end_index - start_index)
        text_list[i] = text_sample
    return text_list
    def get_random_text_from_file(
file_path: str, file_length: int, text_length: int | tuple[int, int] |
list[int, int]
) -> str:
" " "
Given the path to a file returns a random string, with the given parameters.
: param file__path: The path to the file
: param file_length: The length of the file
:param text_length: The length of the text, can be a single int,
or two ints in a tuple for a random length within that interval
:return: String with text
" " "
return get_random_texts_from_file(file_path, file_length, text_length, 1)[0]
def get_number_of_chars_in_file(
file_path: str
) -> int:
" " "
Given a file returns the number of characters within the file.
: param file_path: The path to the file
:return: The number of characters in the file
" " "
file = open(file_path, "r")
data = file.read()
length = len(data)
file.close()
return length

```
thesis_code/coding/generatecrypto.py
```

import random
from random import randint
from typing import Optional, Union
import numpy
from coding.polybius import Polybius, Polybius3d
import coding.handy_vars as handy_vars
from coding.tools import closest_perfect_square

# defaults for generator functions

DEFAULT_PERMUTE_ALPHABET = True
DEFAULT_ALPHABET = list(handy_vars.ENGLISH_ALPHABET)
DEFAULT_ALPHABET_POLYBIUS = list(handy_vars.ENGLISH_ALPHABET_NO_J)
DEFAULT_ALPHABET_TRIFID = list(handy_vars.ENGLISH_ALPHABET_27)
DEFAULT_MIN_KEY_LENGTH = 5
DEFAULT_MAX_KEY_LENGTH = 20 \# the max length itself can be used
DEFAULT_BLANK_CHARACTER = "x"
def create_caesar_cryptogram(
text: str,
alphabetic_key_list: Optional[list[str]] = None,
permute_alphabet: Optional[bool] = DEFAULT_PERMUTE_ALPHABET,
key: Optional[int] = None
-> tuple[str, tuple[str, ...], int]:
Creates a cryptogram using the caesar cipher.
:param text: Text to cipher
:param alphabetic_key_list: The alphabet list in its order _ None uses
DEFAULT ALPHABET
: param permute_alphabet: True to permute, false otherwise
: param key: Key if you want to define one, otherwise a random will be chosen
:return: Tuple with (ciphered text, Alphabetic key, key)
" " "
\# setting the alphabet
alphabetic_key_list = alphabetic_key_list if alphabetic_key_list else
DEFAULT_ALPHABET
\# permuting the alphabetic key if necessary
if permute_alphabet:
random.shuffle(alphabetic_key_list)
alphabetic_key = tuple(alphabetic_key_list)
\# setting the shift if necessary
if key is None:
key = randint(1, len(alphabetic_key_list) - 1) \# starts in 1, ends in
len(alphabetic__key) - 1
from coding.cipher import caesar_cipher
ciphertext = caesar_cipher(text, alphabetic_key, key)
return ciphertext, alphabetic_key, key

```
```

def create_vigenere_cryptogram(
text: str,
alphabetic_key_list: Optional[list[str, ...]] = None,
permute_alphabet: Optional[bool] = DEFAULT_PERMUTE_ALPHABET,
key: Optional[tuple[str, ...]] = None,
key_len: Optional[int] = None,
) -> tuple[str, tuple[str, ...], tuple[str, ...]]:
" " "
Creates a cryptogram using the vigenere cipher.
:param text: The text to cipher
:param alphabetic_key_list: The alphabet list in its order - None uses
DEFAULT_ALPHABET
:param permute_alphabet: True to permute, false otherwise
:param key: Keyword, None is a random one with size key_len
:param key_len: Size for the random key, None ranges from
DEFAULT_MIN_KEY_LENGTH to DEFAULT_MAX_KEY_LENGTH
:return: Tuple with (ciphered text, key, alphabetic key)
" " "
\# setting the alphabet
alphabetic_key_list = alphabetic_key_list if alphabetic_key_list else
DEFAULT_ALPHABET
\# permuting the alphabetic key if necessary
if permute_alphabet:
alphabetic_key_list = list(alphabetic_key_list)
random.shuffle(alphabetic_key_list)
alphabetic_key = tuple(alphabetic_key_list)
\# setting the key
if key is None:
if key_len is None:
key_len = randint(DEFAULT_MIN_KEY_LENGTH, DEFAULT_MAX_KEY_LENGTH)
key = tuple(numpy.random.choice([char for char in alphabetic_key],
size=key_len, replace=True))
from coding.cipher import vigenere_cipher
ciphertext = vigenere_cipher(text, alphabetic_key, key)
return ciphertext, alphabetic_key, key
def create_autokey_cryptogram(
text: str,
mode: Optional[str] = "input",
alphabetic_key_list: Optional[list[str, ...]] = None,
permute_alphabet: Optional[bool] = DEFAULT_PERMUTE_ALPHABET,
key: Optional[Union[tuple[str, ...], str]] = None,
key_len: Optional[int] = None
) -> tuple[str, tuple[str, ...], tuple[str, ...]]:

```
```

    Creates a cryptogram using the autokey ciphers.
    :param text: Text to cipher
    : param mode: "input" for autokey plaintext fed; "output" for autokey
        ciphertext fed
    :param alphabetic_key_list: The alphabet list in its order - None uses
        DEFAULT_ALPHABET
    : param permute__alphabet: True to permute, false otherwise
    :param key: Keyword, None is a random one with size key_len
    :param key_len: Size for the random key, None ranges from
        DEFAULT_MIN_KEY_LENGTH to DEFAULT_MAX_KEY_LENGTH
    : return: Tuple with (ciphered text, alphabetic key, key)
    " " "
    if mode not in ["input", "output"]:
        raise Exception("Invalid mode")
    # setting the alphabet
    alphabetic_key_list = alphabetic_key_list if alphabetic_key_list else
        DEFAULT_ALPHABET
    # permuting the alphabetic key if necessary
    if permute_alphabet:
        alphabetic_key_list = list(alphabetic_key_list)
        random.shuffle(alphabetic_key_list)
    alphabetic_key = tuple(alphabetic_key_list)
    # setting the key
    if key is None:
        if key_len is None:
            key_len = randint(DEFAULT_MIN_KEY_LENGTH, DEFAULT_MAX_KEY_LENGTH)
        key = tuple(numpy.random.choice([char for char in alphabetic_key],
            size=key_len, replace=True))
    ciphertext = ""
    if mode == "input":
        from coding.cipher import autokey_input_fed_cipher
        ciphertext = autokey_input_fed_cipher(text, alphabetic_key, key)
    elif mode == "output":
from coding.cipher import autokey_output_fed_cipher
ciphertext = autokey_output_fed_cipher(text, alphabetic_key, key)
return ciphertext, alphabetic_key, key
def create_bifid_cryptogram(
text: str,
polybius: Optional[Polybius] = None,
keyword: Optional[tuple[str, ...]] = None,
alphabetic_key_list: Optional[list[str, ...]] = None,
permute_alphabet: Optional[bool] = DEFAULT_PERMUTE_ALPHABET,
period: Optional[int] = None
) -> tuple[str, Polybius, int]:

```
```

    " " "
    Creates a cryptogram using the bifid cipher.
    :param text: Text to cipher
    :param polybius: A polybius to use, None to generate a random one
    :param keyword: Initialization key for the Polybius
    :param alphabetic_key_list: The alphabet list in its order - None uses
        DEFAULT_ALPHABET_POLYBIUS
    : param permute_alphabet: True to permute, false otherwise
    : param period: The period for the cipher, None ranges from
        DEFAULT_MIN_KEY_LENGTH to DEFAULT_MAX_KEY_LENGTH
    : return: Tuple with (ciphered text, polybius, period)
    " " "
    # setting the polybius
    if polybius is None:
        polybius = get_polybius(permute_alphabet, keyword, alphabetic_key_list)
    # setting the period
    if period is None:
        period = randint(DEFAULT_MIN_KEY_LENGTH, DEFAULT_MAX_KEY_LENGTH)
    from coding.cipher import bifid_cipher
    ciphertext = bifid_cipher(text, polybius, period)
    return ciphertext, polybius, period
    def create_phillips_cryptogram(
text: str,
polybius: Optional[Polybius] = None,
keyword: Optional[tuple[str, ...]] = None,
alphabetic_key_list: Optional[list[str, ...]] = None,
permute_alphabet: Optional[bool] = DEFAULT_PERMUTE_ALPHABET,
period: Optional[int] = None
) -> tuple[str, Polybius, int]:
" " "
Creates a cryptogram using the phillips cipher.
:param text: Text to cipher
:param polybius: A polybius to use, None to generate a random one
:param keyword: Initialization key for the Polybius
:param alphabetic_key_list: The alphabet list in its order - None uses
DEFAULT_ALPHABET_POLYBIUS
: param permute_alphabet: True to permute, false otherwise
: param period: The period for the cipher, None ranges from
DEFAULT_MIN_KEY_LENGTH to DEFAULT_MAX_KEY_LENGTH
: return: Tuple with (ciphered text, Polybius, period)
" " "
\# setting the alphabet
alphabetic_key_list = alphabetic_key_list if alphabetic_key_list else
DEFAULT_ALPHABET_POLYBIUS
\# setting the polybius

```
```

    if polybius is None:
        polybius = get_polybius(permute_alphabet, keyword, alphabetic_key_list)
    # setting the period
    if period is None:
        period = randint(DEFAULT_MIN_KEY_LENGTH, DEFAULT_MAX_KEY_LENGTH)
    from coding.cipher import phillips_cipher
    ciphertext = phillips_cipher(text, polybius, period)
    return ciphertext, polybius, period
    def create_trifid_cryptogram(
text: str,
polybius3d: Optional[Polybius3d] = None,
keyword: Optional[tuple[str, ...]] = None,
alphabetic_key_list: Optional[list[str, ...]] = None,
permute_alphabet: Optional[bool] = DEFAULT_PERMUTE_ALPHABET,
period: Optional[int] = None
) -> tuple[str, Polybius3d, int]:
" " "
Creates a cryptogram using the trifid cipher.
:param text: Text to cipher
:param polybius3d: A (3-dimensional) polybius to use, None to generate a
random one
:param keyword: Initialization key for the Polybius
:param alphabetic_key_list: The alphabet list in its order - None uses
DEFAULT_ALPHABET__TRIFID
: param permute__ alphabet: True to permute, false otherwise
: param period: The period for the cipher, None ranges from
DEFAULT_MIN_KEY_LENGTH to DEFAULT_MAX_KEY_LENGTH
:return: Tuple with (ciphered text, Polybius, period)
" " "
\# setting the alphabet
alphabetic_key_list = alphabetic_key_list if alphabetic_key_list else
DEFAULT_ALPHABET_TRIFID
\# setting the three-dimensional polybius
if polybius3d is None:
polybius3d = get_polybius3d(permute_alphabet, keyword,
alphabetic_key_list)
if period is None:
period = randint (DEFAULT_MIN_KEY_LENGTH, DEFAULT_MAX_KEY_LENGTH)
from coding.cipher import trifid_cipher
ciphertext = trifid_cipher(text, polybius3d, period)
return ciphertext, polybius3d, period
def create_checkerboard_cryptogram(
text: str,

```
```

            polybius: Optional[Polybius] = None,
        keyword: Optional[tuple[str, ...]] = None,
        alphabetic_key_list: Optional[list[str, ...]] = None,
        permute_alphabet: Optional[bool] = DEFAULT_PERMUTE_ALPHABET
    ) -> tuple[str, Polybius]:
" " "
Creates a cryptogram using the checkerboard cipher.
:param text: Text to cipher
:param polybius: A polybius to use, None to generate a random one.
Should have indexes attributed or else random ones will be attributed.
: param keyword: Initialization key for the Polybius
: param alphabetic__key_list: The alphabet list in its order _ None uses
DEFAULT_ALPHABET_POLYBIUS
: param permute__alphabet: True to permute, false otherwise
:return: Tuple with (ciphered text, Polybius)
" " "
\# setting the polybius
if polybius is None:
polybius = get_polybius(permute_alphabet, keyword, alphabetic_key_list)
if polybius.has_indexes() is False:
\# necessary to set up a list to use for the indexes
if alphabetic_key_list is None:
alphabetic_key_list = DEFAULT_ALPHABET_POLYBIUS
alphabet2 = alphabetic_key_list.copy()
random.shuffle(alphabet2)
alphabet2 = "".join(alphabet2)
hor = [alphabet2[:5], alphabet2[5:10]]
ver = [alphabet2[10:15], alphabet2[15:20]]
polybius.add_indexes(hor, ver)
from coding.cipher import checkerboard_cipher
ciphertext = checkerboard_cipher(text, polybius)
return ciphertext, polybius
def create_numbered_key_cryptogram(
text: str,
keyword: Optional[tuple] = None,
key_len: Optional[int] = None,
alphabetic_key_list: Optional[list[str, ...]] = None,
permute_alphabet: Optional[bool] = DEFAULT_PERMUTE_ALPHABET
) -> tuple[str, tuple]:
" " "
Ciphers using the numbered key cipher
:param text: Text to cipher
:param keyword: Initialization key for the numbered key
:param permute_alphabet: True if you wish the alphabet to be permuted,

```
```

        False otherwise
    :param key_len: Size for the random key, None ranges from
        DEFAULT_MIN_KEY_LENGTH to DEFAULT_MAX_KEY_LENGTH
        : param alphabetic_key_list: The alphabet list in its order - None uses
        DEFAULT_ALPHABET
        : return: The ciphered text, numbered key
        " " "
        # setting the alphabet
        alphabetic_key_list = alphabetic_key_list if alphabetic_key_list else
        DEFAULT_ALPHABET
    # permuting the alphabetic key if necessary
    if permute_alphabet:
random.shuffle(alphabetic_key_list)
if keyword is None:
if key_len is None:
key_len = randint(DEFAULT_MIN_KEY_LENGTH, DEFAULT_MAX_KEY_LENGTH)
keyword = tuple(numpy.random.choice(alphabetic_key_list, size=key_len,
replace=True))
from coding.tools import get_numbered_key
numbered_key = get_numbered_key(keyword, alphabetic_key_list, randint(1,
len(alphabetic_key_list) - 1))
from coding.cipher import numbered_key_cipher
ciphertext = numbered_key_cipher(text, numbered_key)
return ciphertext, numbered_key
def create_playfair_cryptogram(text: str,
polybius: Optional[Polybius] = None,
keyword: Optional[tuple[str, ...]] = None,
alphabetic_key_list: Optional[list[str, ...]] =
None,
permute_alphabet: Optional[bool] =
DEFAULT_PERMUTE_ALPHABET,
blank_character: Optional[str] =
DEFAULT_BLANK_CHARACTER
) -> tuple[str, Polybius, str]:
Creates a cryptogram using the playfair cipher.
:param text: Text to cipher
:param polybius: A polybius to use, None to generate a random one.
:param keyword: Initialization key for the Polybius
: param alphabetic__ey_list: The alphabet list in its order - None uses
DEFAULT_ALPHABET_POLYBIUS
: param permute__ alphabet: True to permute, false otherwise
:param blank_character: Character to use as blank character, default uses
DEFAULT BLANK CHARACTER
:return: Tuple with (ciphered text, polybius, blank character)
" " "

```
```

    # setting the polybius
    if polybius is None:
        polybius = get_polybius(permute_alphabet, keyword, alphabetic_key_list)
    from coding.cipher import playfair_cipher
    ciphertext = playfair_cipher(text, polybius, blank_character)
    return ciphertext, polybius, blank_character
    def create_nihilist_substitution_cryptogram(
text: str,
polybius: Optional[Polybius] = None,
key: Optional[tuple[str, ...]] = None,
key_len: Optional[int] = None,
keyword: Optional[tuple[str, ...]] = None,
alphabetic_key_list: Optional[list[str, ...]] = None,
permute_alphabet: Optional[bool] = DEFAULT_PERMUTE_ALPHABET
) -> tuple[str, Polybius, tuple[str, ...]]:
" " "
Creates a cryptogram using the nihilist substitution cipher.
:param text: Text to cipher
:param polybius: A polybius to use, None to generate a random one.
:param key: Key to be used when ciphering
:param key__len: Size for the random key, None ranges from
DEFAULT_MIN_KEY_LENGTH to DEFAULT_MAX_KEY_LENGTH
:param keyword: Initialization key for the Polybius
: param alphabetic_key_list: The alphabet list in its order - None uses
DEFAULT_ALPHABET_POLYBIUS
: param permute_alphabet: True to permute, false otherwise
:return: Tuple with (ciphertext, polybius, key)
" " "
\# setting the alphabet
alphabetic_key_list = alphabetic_key_list if alphabetic_key_list else
DEFAULT_ALPHABET_POLYBIUS
\# setting the polybius
if polybius is None:
polybius = get_polybius(permute_alphabet, keyword, alphabetic_key_list)
if key is None:
from coding.tools import get_random_string
if key_len is None:
key_len = randint(DEFAULT_MIN_KEY_LENGTH, DEFAULT_MAX_KEY_LENGTH)
key = get_random_string(alphabetic_key_list, key_len)
from coding.cipher import nihilist_substitution_cipher
ciphertext = nihilist_substitution_cipher(text, polybius, key)
return ciphertext, polybius, key
def create_nihilist_transposition_cryptogram(

```
```

    text: str,
    key: Optional[tuple[int, ...]] = None,
    alphabetic_key_list: Optional[list[str, ...]] = None,
    blank_character: str = DEFAULT_BLANK_CHARACTER
    ) -> tuple[str, tuple[str, ...], tuple[int, ...]]:
" " "
Creates a cryptogram using the nihilist transposition cipher.
: param blank_character: Character to use as blank character, default uses
DEFAULT_BLANK_CHARACTER
:param text: Text to cipher
: param key: Key to be used when ciphering
:param alphabetic_key_list: The alphabet list in its order - None uses
DEFAULT_ALPHABET
:return: Tuple with (ciphertext, polybius, transposition key)
" " "
from coding.cipher import nihilist_transposition_cipher
\# setting the alphabet
alphabetic_key_list = alphabetic_key_list if alphabetic_key_list else
DEFAULT_ALPHABET
alphabetic_key = tuple(alphabetic_key_list)
next_square = closest_perfect_square(len(text))
from math import sqrt, ceil
square_len = ceil(sqrt(next_square))
if key is None:
key_list = list(range(0, square_len))
random.shuffle(key_list)
key = tuple(key_list)
ciphertext = nihilist_transposition_cipher(text, alphabetic_key, key,
blank_character)
return ciphertext, tuple(alphabetic_key_list), key
def cipher_text_with_given_ciphers(
text: str,
ciphers_tuple: tuple[str, ...],
period_or_key_len=None
) -> dict[str, str]:
" " "
Given a text returns a tuple of the text ciphered with each cipher
:param text: The text we want ciphered
: param ciphers_tuple: List with ciphers to be used
:param period__or_key_len: Integer with the period length or for a random
key length
:return: Tuple of ciphered text
" " "
alphabet_list = list(handy_vars.ENGLISH_ALPHABET)
alphabet_list_polybius = list(handy_vars.ENGLISH_ALPHABET_NO_J)
alphabet_list_trifid = list(handy_vars.ENGLISH_ALPHABET_27)

```
```

if len(alphabet_list) == 25:
alphabet_list_polybius = alphabet_list
if len(alphabet_list) == 27:
alphabet_list_trifid = alphabet_list
ciphers_and_cryptograms = {cipher: "" for cipher in ciphers_tuple}
for cipher in ciphers_tuple:
if cipher == "caesar":
ciphers_and_cryptograms[cipher] = create_caesar_cryptogram(
text,
alphabetic_key_list=alphabet_list,
key=period_or_key_len
) [0]
elif cipher == "vigenere":
ciphers_and_cryptograms[cipher] = create_vigenere_cryptogram(
text,
alphabetic_key_list=alphabet_list,
key_len=period_or_key_len
) [0]
elif cipher == "autokey-input":
ciphers_and_cryptograms[cipher] = create_autokey_cryptogram(
text,
mode="input",
alphabetic_key_list=alphabet_list,
key_len=period_or_key_len
) [0]
elif cipher == "autokey-output":
ciphers_and_cryptograms[cipher] = create_autokey_cryptogram(
text,
mode="output",
alphabetic_key_list=alphabet_list,
key_len=period_or_key_len
) [0]
elif cipher == "bifid":
ciphers_and_cryptograms[cipher] = create_bifid_cryptogram(
text.replace("j", "i"),
alphabetic_key_list=alphabet_list_polybius,
period=period_or_key_len
) [0]
elif cipher == "phillips":
ciphers_and_cryptograms[cipher] = create_phillips_cryptogram(
text.replace("j", "i"),
alphabetic_key_list=alphabet_list_polybius,
period=period_or_key_len
) [0]
elif cipher == "checkerboard":
ciphers_and_cryptograms[cipher] = create_checkerboard_cryptogram(
text.replace("j", "i"),
alphabetic_key_list=alphabet_list_polybius
)[0]

```
```

    elif cipher == "trifid":
        ciphers_and_cryptograms[cipher] = create_trifid_cryptogram(
            text,
            alphabetic_key_list=alphabet_list_trifid,
            period=period_or_key_len
        ) [0]
    elif cipher == "numbered-key":
        ciphers_and_cryptograms[cipher] = create_numbered_key_cryptogram(
            text,
            alphabetic_key_list=alphabet_list,
            key_len=period_or_key_len
        ) [0]
    elif cipher == "playfair":
        ciphers_and_cryptograms[cipher] = create_playfair_cryptogram(
            text.replace("j", "i"),
            alphabetic_key_list=alphabet_list_polybius
        ) [0]
    elif cipher == "nihilist-substitution":
        ciphers_and_cryptograms[cipher] =
            create_nihilist_substitution_cryptogram(
            text.replace("j", "i"),
            alphabetic_key_list=alphabet_list_polybius,
            key_len=period_or_key_len
        ) [0]
    elif cipher == "nihilist-transposition":
        ciphers_and_cryptograms[cipher] =
            create_nihilist_transposition_cryptogram(
            text,
            alphabetic_key_list=alphabet_list
        ) [0]
        else:
        Exception("CIPHER DOESNT EXIST")
    return ciphers_and_cryptograms
    def get_polybius(
permute_alphabet: bool = True,
keyword: Optional[tuple[str, ...]] = None,
alphabetic_key_list: Optional[list[str, ...]] = None
) -> 'Polybius' :
" " "
Method to generate a random polybius according to the given input.
:param permute_alphabet: True to permute, false otherwise
:param keyword: Initialization key for the Polybius
: param alphabetic_key_list: The alphabet list in its order _ None uses
DEFAULT_ALPHABET_POLYBIUS
:return: Generated polybius
" " "
alphabetic_key_list = alphabetic_key_list if alphabetic_key_list else
DEFAULT_ALPHABET_POLYBIUS

```
```

    # permuting the alphabetic key if necessary
    if permute_alphabet:
        alphabetic_key_list = list(alphabetic_key_list)
    random.shuffle(alphabetic_key_list)
    alphabetic_key = tuple(alphabetic_key_list)
    return Polybius.horizontal_from_keyword(
        keyword,
        alphabetic_key,
        forbidden_characters="j",
    )
    def get_polybius3d(
permute_alphabet: bool = True,
keyword: Optional[tuple[str, ...]] = None,
alphabetic_key_list: Optional[list[str, ...]] = None
) -> 'Polybius3d':
" " "
Method to generate a random (3-dimensional) polybius according to the given
input.
:param permute_alphabet: True to permute, false otherwise
:param keyword: Initialization key for the Polybius
:param alphabetic_key_list: The alphabet list in its order - None uses
DEFAULT_ALPHABET_POLYBIUS
:return: Generated polybius
" " "
\# permuting the alphabetic key if necessary
if permute_alphabet:
alphabetic_key_list = list(alphabetic_key_list)
random.shuffle(alphabetic_key_list)
alphabetic_key = tuple(alphabetic_key_list)
\# setting the polybius
polybius3d = Polybius3d.from_keyword(
keyword,
alphabetic_key,
)
return polybius3d

```
thesis_code/coding/handy_vars.py
```

" " "
This file contains a collection of handy reoccurring variables.
" " "
from fractions import Fraction
MONOALPHABETIC_MIN_IC: Fraction = Fraction(57, 1000) \#0.057
ENGLISH_ALPHABET: tuple[str, ...] = \
('a', 'b', 'c', 'd', 'e', '£', 'g', 'h', 'i', 'j',

```
```

    'k', ' l', 'm', 'n', 'o', 'p', ' q', 'r', ' s', ' t', ' u', 'v', 'w', ' x', ' y',
    'z')
    ENGLISH_ALPHABET_NO_J: tuple[str, ...] = \
('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i',

```

```

        'z')
    ENGLISH_ALPHABET_27: tuple[str, ...] = \
('a', 'b', 'c', ' d', 'e', 'f', 'g', 'h', 'i', ' j',

```

```

        'z', '#')
    PORTUGUESE_ALPHABET: tuple[str, ...] = \
('a', 'b', 'c', 'd', ' ' ', 'f', 'g', 'h', 'i', 'j',

```

```

        'z')
    ALL_CRYPTOGRAMS_ALPHABET: tuple[str, ...] = ('a', 'b', 'c', 'd', 'e', 'f', 'g',
'h', 'i', 'j',
'k', ' l', 'm', 'n', 'o', 'p', ' q',
'r', 's', 't', 'u', 'V', 'w',
' x', ' y',
'z', '\#',
'1', '2', '3', '4', '5', '6', '7',
\prime8', '9', '0')
ENGLISH_LETTER_FREQ = {
' a': Fraction(41, 500),
'b': Fraction(3, 200),
'C': Fraction(7, 250),
'd': Fraction(43, 1000),
'e': Fraction(13, 100),
'f': Fraction(11, 500),
' g': Fraction(1, 50),
'h': Fraction(61, 1000),
'i': Fraction(7, 100),
'j': Fraction(3, 2000),
'k': Fraction(77, 10000),
' l': Fraction(1, 25),
'm': Fraction(3, 125),
'n': Fraction(67, 1000),
'O': Fraction(3, 40),
'p': Fraction(19, 1000),
' q': Fraction(19, 20000),
'r': Fraction(3, 50),
's': Fraction(63, 1000),
't': Fraction(91, 1000),
'u': Fraction(7, 250),
'v': Fraction(49, 5000),
'w': Fraction(3, 125),

```
```

    'x': Fraction(3, 2000),
    'Y': Fraction(1, 50),
    'z': Fraction(37, 50000)
    }

# 

# ENGLISH_LETTER_FREQ: dict[str, float] = {'a': 8.2, 'b ': 1.5, 'c ': 2.8, 'd':

    4.3, 'e': 13,
    
# 'f': 2.2, 'g': 2, 'h': 6.1, 'i ': 7,

    'j':0.15,
    
# ' ' ': 0.77, 'l':4, 'm': 2.4, 'n':

    6.7, 'o ': 7.5,
    
# 6.3, 't': 9.1,

# 'u': 2.8, 'v':0.98, 'w': 2.4, 'x':

    0.15, 'y': 2,
    
# 'z':0.074}

PORTUGUESE_LETTER_FREQ: dict[str, float] = {'a': 13.49, 'b': 1.01, 'c': 3.75,
'd': 4.21, 'e': 14.07,
'f': 1.07, 'g': 1.08, 'h': 1.22,
'i': 5.67, 'j': 0.30,
'k': 0.13, ' l': 3, 'm': 5.07, 'n' :
5.02, 'o': 10.44,
'p': 3.01, 'q': 1.10, 'r': 6.73,
's': 7.35, 't': 5.07,
'u': 4.57, 'v': 1.72, 'w': 0.05,
' X': 0.28, 'Y': 0.04,
'z'}=0.45
ALL_HEURISTICS_TUPLE = (
"h_alpha",
"h_ic",
"h_ic_period",
"h_ncd",
"h_phillips",
"h_trans",
)
ALL_CIPHERS_TUPLE = (
"caesar",
"vigenere",
"autokey-input",
"autokey-output",
"bifid",
"phillips",
"checkerboard",
"trifid",
"numbered-key",
"playfair",

```
```

    "nihilist-substitution",
    "nihilist-transposition",
    )
ALL_CIPHERS_TUPLE_NICE_NAME =
"Caesar",
"Vigenere",
"Input autokey",
"Output autokey",
"Bifid",
"Phillips",
"Chequerboard",
"Trifid",
"Numbered key",
"Playfair",
"Nihilist substitution",
"Nihilist transposition",
)
CIPHERS_NICE_NAME_DICT = {
'caesar': 'Caesar',
'vigenere': 'Vigenere',
'autokey-input': 'Input autokey',
'autokey-output': 'Output autokey',
'bifid' : 'Bifid',
'phillips': 'Phillips',
'checkerboard': 'Chequerboard',
'trifid': 'Trifid',
'numbered-key': 'Numbered key',
'playfair': 'Playfair',
'nihilist-substitution': 'Nihilist substitution',
'nihilist-transposition': 'Nihilist transposition'
}
PERIODIC_CIPHERS_TUPLE = (
"vigenere",
"autokey-input",
"autokey-output",
"bifid",
"phillips",
"trifid",
"nihilist-substitution"
)
NON_PERIODIC_CIPHERS_TUPLE = (
"caesar",
"checkerboard",
"numbered-key",
"playfair",

```
```

    "nihilist-transposition"
    ```
)
thesis_code/coding/polybius.py
```

from typing import Optional
class Polybius:
def __init__(
self,
content: tuple[str, ...],
col_words: Optional[list[str, ...]] = None,
row_words: Optional[list[str, ...]] = None
):
self.t = {}
self.t_inv = {}
for i in range(25):
current_tuple = (i // 5, i % 5)
self.t[content[i]] = current_tuple
self.t_inv[current_tuple] = content[i]
self.row_words = row_words
self.col_words = col_words
@classmethod
def from_content_list(
cls,
content: tuple[str],
col_words: Optional[list[str]] = None,
row_words: Optional[list[str]] = None
):
Constructs a Polybius square from the 25 first symbols of a given list.
Does not check for repeats.
:param content: The characters to fill the square with
:param col_words: The words to fill the horizontal indexes much like
for the checkerboard cipher
:param row_words: The words to fill the vertical indexes much like for
the checkerboard cipher
:return: Polybius square with given characters
" " "
return cls(content, row_words=row_words, col_words=col_words)
@classmethod
def horizontal_from_keyword(
cls,
keyword: tuple[str, ...],
alphabetic_key: tuple[str, ...],
col_words: Optional[tuple[str]] = None,

```
```

            row_words: Optional[tuple[str]] = None,
            forbidden_characters: Optional[str] = None,
    ) -> 'Polybius':
    " " "
    Construction of a polybius square using the usual horizontal method.
    :param keyword: Keyword to initiate the Polybius
    :param alphabetic_key: The alphabet to fill the Polybius
    :param col__words: The words to fill the horizontal indexes much like
        for the checkerboard cipher
    :param row__words: The words to fill the vertical indexes much like for
        the checkerboard cipher
    : param forbidden__characters: Characters we don't want to include in the
        Polybius
    : return:
    " " "
    from coding.tools import get_alphabet_with_key
    content = get_alphabet_with_key(keyword, alphabetic_key)
    if forbidden_characters is not None:
        content = "".join([a for a in content if a not in
            forbidden_characters])
    return cls(content, row_words=row_words, col_words=col_words)
    def print_polybius(self) -> None:
    " " "
    Prints the Polybius square.
    " " "
    for i in range(5):
        for j in range(5):
            print(self.t_inv[(i, j)], end=" ")
        print()
    if self.row_words is not None and self.col_words is not None:
        print("HORIZONTAL WORDS:", self.col_words)
        print("VERTICAL WORDS:", self.row_words)
    def has_indexes(self) -> bool:
    return self.col_words is not None and self.row_words is not None
    def add_indexes(
self,
col_words: list[str],
row_words: list[str]
) :
self.col_words = col_words
self.row_words = row_words
class Polybius3d:
def
init__(
self,
content: tuple[str, ...]

```
```

):
a = 0
b}=
c = 0
self.t = {}
self.t_inv = {}
for letter in content:
self.t[letter] = (a, b, c)
self.t_inv[(a, b, c)] = letter
c += 1
c}=c%
if c % 3 == 0:
b += 1
b}=\textrm{b}%
if b % 3 == 0 and c% 3 == 0:
a += 1
@classmethod
def from_keyword(
cls,
keyword: tuple[str, ...],
alphabet: tuple[str, ...],
) -> 'Polybius3d':
from coding.tools import get_alphabet_with_key
content = get_alphabet_with_key(keyword, alphabet)
return cls(content)

```
thesis_code/coding/statistics.py
```

from fractions import Fraction
from numbers import Number
from typing import TypeVar, Any
import numpy as np
import pandas
import coding.handy_vars
from coding.tools import get_ith_blocks_flatten, get_ith_blocks
T = TypeVar('T') \# Declare type variable
def calc_non_connected_digraphs_frequency(
split_text: list[str],
alphabet: tuple[str],
distance: int
) -> dict[str, int]:

```
```

    For a given distance, calculates the frequency for which a non-connected
        digraph occurs for each letter.
    :param split__text: Text in a list format
    : param alphabet: The alphabet used
    :param distance: Distance between different characters
    :return: Dictionary with frequencies
    " " "
    count = {}
    for i in range(0, len(alphabet)):
        count[alphabet[i]] = 0
    for i in range(0, len(split_text) - distance):
        if split_text[i] == split_text[i + distance]:
            count[split_text[i]] += 1
    return count
    def calc_non_connected_digraphs_multiple_distances(
split_text: list[str],
alphabet: tuple[str],
max_distance: int
) -> dict[int, int]:
" " "
From 1 to a given maximum distance, calculates the frequency for which
non-connected digraphs occur.
:param split_text: Text in list format
: param alphabet: The alphabet used
: param max_distance: Maximum distance between non-connected digraphs that
will be taken into account
- must be greater than 0
: return: Dictionary with frequencies sums
" " "
count = {0: 0, 1:0}
for distance in range(2, max_distance + 1):
count[distance] = sum(calc_non_connected_digraphs_frequency(split_text,
alphabet, distance).values())
return count
def calc_non_connected_digraphs_standard_deviation(
split_text: list[str],
alphabet: tuple[str],
max_distance: int
) -> dict[int, float]:
" " "
Given the max distance between non-connected digraphs to examine,
calculates the standard deviation for each distance,
in an effort to better expose the key size period for certain ciphers.
:param split_text: The text in a list format
: param alphabet: The alphabet of the text's language

```
```

    :param max_distance: Maximum distance between non-connected
    digraphs that will be taken into account - must be greater than 0
    :return: Dictionary with the standard deviation for each
    distance within the given limit
    " " " "
    results = {}
    for distance in range(1, max_distance + 1):
        count = calc_non_connected_digraphs_frequency(split_text, alphabet,
            distance)
    mean = sum(count.values())
    mean = mean / len(count)
    squared_differences_sum = 0
    for key in count.keys():
            squared_difference = (count[key] - mean) ** 2
            squared_differences_sum += squared_difference
    almost = squared_differences_sum / (len(count) - 1)
    results[distance] = almost ** (1 / 2)
    return results
def calc_character_abs_frequency(
split_text: list[Any, ...],
alphabet: tuple[Any, ...] = None
) -> dict[Any, int]:
" " "
Returns a dictionary with absolute frequencies of each character in the
alphabet.
: param alphabet: The alphabet if you want to ensure all characters are in
the frequency dictionary
: param split_text: Text as a list of characters
:return: Dictionary with the format (character }->\mathrm{ respective absolute
frequency)
" " "
if alphabet is not None:
frequency_dict = {a: 0 for a in alphabet}
else:
frequency_dict = {}
for char in split_text:
if char not in frequency_dict.keys():
frequency_dict[char] = 1
else:
frequency_dict[char] += 1
return frequency_dict
def calc_character_rel_frequency(absolute_frequency: dict[Any, int]) ->
dict[Any, Fraction]:
frequencies_sum = sum(absolute_frequency.values())

```
```

    return \{char: Fraction(absolute_frequency[char], frequencies_sum) for char
        in absolute_frequency\}
    def calc_ic(abs_frequency: dict[str, int], text_length: int) -> float:
" " "
Calculates the index of coincidence of a text given the character absolute
frequencies.
: param abs_frequency: Character absolute frequencies' dictionary
: param text_length: The length of the text, needed for the calculation of
the $I C$
: return: IC in a Fraction
" " "
aux $=$ text_length * (text_length - 1) \# len (text) * (len (text) -1$)$
addition $=0$
for letter in abs_frequency:
addition += abs_frequency[letter] * (abs_frequency[letter] - 1)
return addition / aux
def sort_for_plot(df: pandas.DataFrame, to_sort, sort_by, how_to_sort: str) ->
list [Any]:
" " "
Returns attributes sorted according to input.
Options are "lower_quartile", "upper_quartile", "lower_whisker",
"upper__whisker", "min", "max", "mean", "median".
: param df: The dataframe
: param to_sort: Columns with attributes we want sorted
: param sort_by: Column with the values with which to sort
: param how_to_sort: Options are "lower_quartile", "upper_quartile",
"lower_whisker", "upper_whisker", "min", "max", "mean", "median"
: return: List with sorted attributes
" " "
attributes_and_vals = \{\}
for attribute in df[to_sort].unique():
df_aux = df[df[to_sort] == attribute]
data $=$ df_aux[sort_by].copy().astype(float)
upper_quartile = np.percentile(data, 75)
lower_quartile = np.percentile(data, 25)
if how_to_sort == "lower_quartile":
attributes_and_vals[attribute] = lower_quartile
continue
elif how_to_sort == "upper_quartile":
attributes_and_vals[attribute] = upper_quartile
continue
elif how_to_sort == "min":
attributes_and_vals[attribute] = df_aux[sort_by].min()
continue

```
```

        elif how_to_sort == "max":
        attributes_and_vals[attribute] = df_aux[sort_by].max()
        continue
    elif how_to_sort == "mean":
        attributes_and_vals[attribute] = df_aux[sort_by].mean()
        continue
    elif how_to_sort == "median":
        attributes_and_vals[attribute] = df_aux[sort_by].median()
        continue
    # the only thing left are whiskers, that require some extra calculation
    iqr = upper_quartile - lower_quartile
    upper_whisker = data[data <= upper_quartile + 1.5 * iqr].max()
    lower_whisker = data[data >= lower_quartile - 1.5 * iqr].min()
    if how_to_sort == "lower_whisker":
        attributes_and_vals[attribute] = lower_whisker
    elif how_to_sort == "upper_whisker":
        attributes_and_vals[attribute] = upper_whisker
    else:
        raise Exception("INVALID SORT!")
    sorted_attributes = sorted(attributes_and_vals,
    key=attributes_and_vals.get, reverse=False)
    return sorted_attributes
    def period_with_ic_cols(split_text: list[str], alphabet: tuple[str],
max_period_guess: int) -> dict[int, float]:
Calculates, for a given text, the likelihood of each period using the index
of coincidence.
:param split__text: The text in a list
: param alphabet: The alphabet
: param max_period__guess: The maximum period that may have been used
:return: Returns dictionary with the likelihood for each period
" " "
periods_list = [i for i in range(2, max_period_guess + 1)]
periods_dict = {0: 0.0, 1: 0.0}
for period in periods_list:
mean_ic = 0.0
for i in range(0, period):
text_from_column = get_ith_blocks_flatten(split_text, i, 1, period)
column_abs_frequency =
calc_character_abs_frequency(text_from_column, alphabet)
ic = calc_ic(column_abs_frequency, len(text_from_column))
mean_ic += float(ic)
mean_ic = mean_ic / period \# calculating the mean
periods_dict[period] = mean_ic

```
```

    return periods_dict
    def fi_likelihood(
list_of_values: list[Number],
min_period: int = 5,
max_period: int = 20,
min_likelihood: Fraction = Fraction(90, 100)
) -> tuple[int, Fraction]:
" " "
Given a list of values in a dataframe, searches for a period using
increasingly bigger windows.
Here it is expected for the peaks to occur at the last place of the window
:param list__of__values: List with values
:param min_period: A minimum period, should there be one. By default, it is
5.
:param max_period: A maximum period, should there be one. By default, it is
20.
:param min__likelihood: The minimum likelihood to stop searching.
:return: Returns the period guess and its likelihood
" " "
best_likelihood = Fraction(0)
for p in range(min_period, max_period + 1):
\# We calculate the max for every p values (lets call this a block).
\# If the maxes are equidistant (have the same position in each block),
\# then there should be a period
likelihood = get_period_likelihood(list_of_values, p)
if likelihood > best_likelihood:
best_likelihood = likelihood
\# If the likelihood meets our threshold we return the period.
if likelihood >= min_likelihood:
return p, likelihood
\# If no period was detected, the period returned is 0 with a likelihood of 0
return 0, Fraction(0)
def get_period_likelihood(
list_of_values: list[Number], period: int
) -> Fraction:
" " "
Given a list of values and a period returns the likelihood of that period.
: param list__of__values: List with numeric values
: param period: The period we want to obtain the likelihood of
:return: The likelihood
\# We calculate the max for every period number of values (lets call this a
block).
\# If the maxes are equidistant (have the same position in each block),

```
```

    # then there should be a period.
    blocks_list = get_ith_blocks(list_of_values, 1, period, period)
    popularity = 0
    for block in blocks_list:
        block_max = max(block)
        if block[period - 1] == block_max:
            popularity += 1
    likelihood = Fraction(popularity, len(blocks_list))
    return likelihood
    def period_with_ic_phillips(split_text: list[str], alphabet: tuple[str],
min_period_guess: int = 2,
max_period_guess: int = 20) -> dict[int, float]:
" " "
Function that attempts to guess the phillips period using the ic.
:param split_text: The text in a list
: param alphabet: The alphabet
: param min__period__guess: The maximum period
:param max__period__guess: The minimum period
: return: A dictionary with the ic for each period
" " "
periods_dict = {0: 0.0, 1: 0.0}
for period in list(range(min_period_guess, max_period_guess + 1)):
mean_ic = 0.0
for i in range(0, 8):
text_from_square = get_ith_blocks_flatten(split_text, i * period,
period, period * 8)
if not text_from_square:
break
column_abs_frequency =
calc_character_abs_frequency(text_from_square, alphabet)
ic = calc_ic(column_abs_frequency, len(text_from_square))
mean_ic += ic
mean_ic = mean_ic / 8
periods_dict[period] = mean_ic
return periods_dict
def likely_phillips(
split_text: list[str],
alphabet: tuple[str],
min_monoalphabetic_ic: Fraction =
coding.handy_vars.MONOALPHABETIC_MIN_IC,
min_period_guess: int = 5,
max_period_guess: int = 20
) -> tuple[int, Fraction]:

```
```

" " "
Calculates how likely it is for the text to have been ciphered with the
phillips cipher.
: param split__text: The text in a list
: param alphabet: The alphabet
:param min_monoalphabetic__ic: The minimum index of coincidence for a
monoalphabetic ciphers
:param min_period__guess: The minimum period
:param max_period__guess: The maximum period
: return: The most likely period to have been used and how likely it is;
if it is not likely it returns a period of }0\mathrm{ and a likelihood of 0
" " "
text_abs_freq = calc_character_abs_frequency(split_text, alphabet)
text_ic = calc_ic(text_abs_freq, len(split_text))
if text_ic >= coding.handy_vars.MONOALPHABETIC_MIN_IC:
return 0, Fraction(0)
period_likelihood = {}
for period in range(min_period_guess, max_period_guess + 1):
n_of_ics = 0
for i in range(0, 8):
text_from_square = get_ith_blocks_flatten(split_text, i * period,
period, period * 8)
if not text_from_square:
break
column_abs_frequency =
calc_character_abs_frequency(text_from_square, alphabet)
ic = calc_ic(column_abs_frequency, len(text_from_square))
if ic >= min_monoalphabetic_ic:
n_of_ics += 1
period_likelihood[period] = Fraction(n_of_ics, 8)

# for a in period_likelihood.keys():

# print(a, float(period__likelihood[a]))

best_period = max(period_likelihood, key=period_likelihood.get)
best_likelihood = period_likelihood[best_period]
return best_period, best_likelihood

```
thesis_code/coding/tools.py
```

import math
import os
import typing

# for generic objects

from fractions import Fraction
from typing import TypeVar
import pandas as pd

```
```

from pandas import DataFrame
T = TypeVar('T') \# Declare type variable
A = TypeVar('A') \# Declare type variable
def get_alphabet_with_key(key: tuple[str, ...], alphabet: tuple[str, ...]) ->
tuple[str, ...]:
" " "
Given a keyword and an alphabet creates a new alphabet that starts with the
key,
similarly to what most polybius schemes do.
:param key: The keyword
: param alphabet: The alphabet
:return: The new alphabet order
" " "
key_with_no_repetitions = []
if key is not None:
key_with_no_repetitions = list(dict.fromkeys(key))
return tuple(key_with_no_repetitions + [letter for letter in alphabet if
letter not in key_with_no_repetitions])
def get_numbered_key(key: typing.Iterable[str], alphabet: list[str], start_pos:
int) -> tuple:
" " "
Given a keyword and an alphabet returns two translation tables for the key
concatenatedf
with the alphabet with a shift so that it starts in initial__pos.
:param key: The keyword
: param alphabet: The alphabet
:param start_pos: The starting position for the numeration
:return: A dictionary with the format letter }->\mathrm{ position }x\mathrm{ and an
identical inverted dictionary.
" " "
new_key = [a for a in key]
new_key = new_key + [letter for letter in alphabet if letter not in key]
new_key = new_key[-start_pos:] + new_key[:-start_pos]
t = {}
for letter in new_key:
t[letter] = []
t_inv = {}
for i in range(len(new_key)):
t[new_key[i]].append(i)
t_inv[i] = new_key[i]
return t, t_inv
def get_ith_letters(text, where_to_start, period) -> str:

```
```

    " " "
    Returns all letters that show in a given period.
    : param text: The text
    : param where_to_start: The index of where it should start
    : param period: The period
    :return:String with the concatenated letters
    " " "
    return ''.join([text[i] for i in range(where_to_start, len(text), period)])
    def get_ith_blocks_flatten(
elements_list: list[T],
where_to_start: int,
block_size: int,
period: int,
allow_last_block_of_different_length: bool = False
) -> list[T]:
" " "
Returns blocks of elements of the given size within a given period.
: param elements_list: The list of elements
: param where_to_start: The index of where it should start
: param block_size: Size of each block every period
: param period: The period
:param allow_last_block__of_different_length: True to get the last block,
even if it is of different length
:return: List of blocks, flattened
" " "
end = len(elements_list) - ((len(elements_list) - where_to_start) %
block_size)
list_of_blocks = []
for i in range(where_to_start, end, period):
if i + block_size > end and not allow_last_block_of_different_length:
break
block = elements_list[i: i + block_size]
for b in block:
list_of_blocks.append(b)
return list_of_blocks
\# return [
\# text_list[i: i + block__size]
\# for i in range(where_to__start, len(text__list) - (len(text_llist) %
block_size), period)]
def get_ith_blocks(
elements_list: list[typing.Any],
where_to_start: int,
block_size: int,
period: int,
allow_last_block_of_different_length: bool = False
) -> list[list[typing.Any]]:

```
```

    " " "
    Returns blocks of elements of the given size within a given period.
    : param elements_list: The list of elements
    : param where_to_start: The index of where it should start
    :param block_size: Size of each block every period
    : param period: The period
    :param allow_last_block__of_different_length: True to get the last block,
    even if it is of different length
    :return: List of blocks, each block is a list within
    " " "
    end = len(elements_list) - ((len(elements_list) - where_to_start) %
    block_size)
    list_of_blocks = []
    for i in range(where_to_start, end, period):
        if i + block_size > end and not allow_last_block_of_different_length:
        break
    block = elements_list[i: i + block_size]
    list_of_blocks.append(block)
    return list_of_blocks
def infer_alphabet_from_text(text: str, symbols_per_character: int) ->
tuple[str, ...]:
" " " "
Given a text returns all the different characters within.
:param text: The text
: param symbols__per_character: Number of symbols per character
sometimes it makes sense to consider multiple characters a single symbol.
: return: The alphabet
" " "
from ordered_set import OrderedSet
letters = OrderedSet(text[i:i + symbols_per_character] for i in range(0,
len(text), symbols_per_character))
return tuple(letters) \# we preserve the order, might come in handy
def get_reverse_dict(dict_to_reverse: dict[T, A]) -> dict[A, list[T]]:
" " "
Given a map reverses it.
:param dict__to__reverse: The dictionary to reverse
: return: Reversed dictionary
" " "
inv_map = {}
for k, v in dict_to_reverse.items():
inv_map[v] = inv_map.get(v, []) + [k]
return inv_map
def get_random_string(alphabet: typing.Union[tuple[str, ...], list[str, ...]],
size: int) -> str:

```
```

    " " "
    Returns a random string of the given alphabet with the given size
    : param alphabet: The alphabet for the string
    :param size: The size of the random string
    : return: Random string
    " " "
    import numpy
    return ''.join(numpy.random.choice(list(alphabet), size=size, replace=True))
    def closest_perfect_square(number: int):
\# first we check if it is already a square
square_root = number ** 0.5
modulus_1 = square_root % 1
is_perfect_square = modulus_1 == 0
if is_perfect_square:
return number
else:
\# the next square is calculated otherwise
next_n = math.floor(math.sqrt(number)) + 1
return next_n * next_n
def split_text_into_list(text: str, alphabet: tuple[str] | list[str]) ->
list[str]:
" " "
Splits the text into a list of characters of the alphabet.
This is needed since we may consider a character to have multiple symbols
: param text: The text to split
:param alphabet: The alphabet to be used
:return: List of characters
" " "
text_position = 0
text_list = []
while text_position != len(text):
no_corresponding_char = True
for alphabet_character in alphabet:
text_character = text[text_position:text_position +
len(alphabet_character)]
if text_character == alphabet_character:
text_list.append(text_character)
text_position += len(text_character)
no_corresponding_char = False
if no_corresponding_char:
\# sanity check to see if there are characters in the text not
present in the alphabet
raise Exception("There are characters in the text that are not in
the alphabet.")

```
```

    return text_list
    def check_current_dir(expected_parent_dir: str) -> bool:
" " "
Method that, given a directory, checks if the program is being run from
said directory.
Useful given that Pycharm seems to be unable to save the default working
directory, which is quite annoying.
: param expected__parent__dir: The directory one is expecting the program to
be run from.
:return: True if the directory is the expected one, false otherwise
" " "
\# get current directory
path = os.getcwd()
current_dir = os.path.basename(path)
\# compare the two
return current_dir == expected_parent_dir
def filter_with_condition_function(df: DataFrame, condition_function:
typing.Callable, column: str) -> DataFrame:
" " "
Given a dataframe and a function that given a value, returns a boolean,
filters using the function.
The rows whose values in the column, when applied the function are true,
are kept in the returned dataframe.
The other rows are ignored.
: param df: The dataframe to filter
:param condition_function: The boolean function
:param column: The column of df where the values with which we want to
filter are
:return: New dataframe with filtered values
" " "
new_df = pd.DataFrame(columns=df.columns)
for idx, row in df.iterrows():
if condition_function(row[column]):
new_df.loc[(len(new_df.index))] = row
return new_df
def easy_dict_print(the_dict: dict[str, Fraction], n_decimals: int = 2,
ignore_fraction: bool = False) -> None:
" " "
Print for dictionaries of the format str }->\mathrm{ Fraction so that one can see
data easily.
: param the_dict: The dictionary
: param n_decimals: The number of decimal digits in the printed float
:param ignore_fraction: If true the fraction is not printed at the end
: return: None
" " "

```
```

for key in the_dict.keys():
f_val = float(the_dict[key])
format_float = ("{:." + str(n_decimals) + "f}").format(f_val)
if ignore_fraction:
print(format_float, key)
else:
print(format_float, key, the_dict[key])

```
thesis_code/demonstration/algorithm_to_get_weights.py
```

import os
from fractions import Fraction
from classifier.classifier_tools import get_weights, scores_df_to_ranks_df,
calc_correct_guesses
from classifier.input_objects import HInput, SAInput, FTInput
from classifier.tuple_rating import id_1, id_2
from coding import handy_vars, corpus
from coding.tools import check_current_dir, easy_dict_print
if ___name___ == '___main___':
\# Import the data
if not check_current_dir("thesis_code"):
raise Exception("Running from wrong directory!")
\# CLEANING UP A CORPUS TO BE USED FOR CRYPTOGRAMS
print("cleaning the corpus if needed...")
corpus_path = "corpora/english_corpus.txt"
clean_corpus_path = "output/cleaned_text.txt"
output_path_to_file = "output/files/scores.csv"
if not os.path.exists(os.path.dirname(output_path_to_file)):
os.makedirs(os.path.dirname(output_path_to_file))
alphabet_to_clean = handy_vars.ENGLISH_ALPHABET
if not os.path.exists(clean_corpus_path):
file_length = corpus.clean_corpus(corpus_path, alphabet_to_clean,
clean_corpus_path)
else:
file_length = corpus.get_number_of_chars_in_file(clean_corpus_path)
\# INPUT
identification_function = id_1
heuristics_tuple = handy_vars.ALL_HEURISTICS_TUPLE
ciphers_tuple = handy_vars.ALL_CIPHERS_TUPLE
sa_input = SAInput(
acceptable_fallback_ratio=Fraction(1, 20),
minimum_improvement_value=Fraction(1, 200),
max_failed_attempts_tolerated=3,
ratings_window_size=12,
identification_function=identification_function

```
```

)
he_input = HInput(
clean_corpus_path=clean_corpus_path,
clean_corpus_length=file_length,
min_text_length=500,
max_text_length=500,
n_texts=300,
symbols_per_character=1,
min_period_guess=5,
max_period_guess=20,
heuristics_tuple=handy_vars.ALL_HEURISTICS_TUPLE
)
initial_weights = {}
for heuristic in heuristics_tuple:
initial_weights[heuristic] = Fraction(1, len(heuristics_tuple))
ft_input = FTInput(
necessary_improvement_between_iterations=Fraction(1, 100),
max_failed_attempts=3,
weight_increment=Fraction(1, len(heuristics_tuple)),
initial_weights=initial_weights,
verbose=True
)

# COMPUTING

computed_weights, scores_df, rating, elapsed_time = get_weights(sa_input,
he_input, ft_input, ciphers_tuple)

# OUTPUTTING

print("SCORES DATAFRAME:")
print(scores_df)
print(float(rating), rating)
print(elapsed_time)
scores_df.to_csv(output_path_to_file)
easy_dict_print(computed_weights)
output_path_to_file = "output/files/ranks.csv"
ranks_df = scores_df_to_ranks_df(scores_df)
ranks_df.to_csv(output_path_to_file)
print("Number of correctly guessed:", calc_correct_guesses(ranks_df))
print("Number of cryptograms in total:", len(ranks_df.index))
ratio = Fraction(calc_correct_guesses(ranks_df), len(ranks_df.index))
print("Ratio between them:", float(ratio), ratio)

```
thesis_code/demonstration/cipher_classifier.py
```

from classifier.heuristics import CalculatedHeuristics
from classifier.tuple_rating import get_ranking
from coding import handy_vars
from coding.tools import check_current_dir, easy_dict_print
if ___name___ == '___main___':
\# Import the data
if not check_current_dir("thesis_code"):
raise Exception("Running from wrong directory!")
\# CLEANING UP A CORPUS TO BE USED FOR CRYPTOGRAMS
print("cleaning the corpus if needed...")
corpus_path = "corpora/english_corpus.txt"
clean_corpus_path = "output/cleaned_text.txt"
\# READING THE CRYPTOGRAM FROM THE FILE
cryptogram_file = "output/files/example_cryptogram.txt"
text_file = open(cryptogram_file, "r")
cryptogram = text_file.read()
text_file.close()
\# INPUT
symbols_per_character = 1
min_period_guess = 5
max_period_guess = 20
heuristics_tuple = handy_vars.ALL_HEURISTICS_TUPLE
ciphers_tuple = handy_vars.ALL_CIPHERS_TUPLE
weights_dict = {
"h_alpha": Fraction(3030623103620308676227, 9387480337647754305649),
"h_ic": Fraction(2855499837626176807254, 9387480337647754305649),
"h_ic_period": Fraction(694248090544448628480, 9387480337647754305649),
"h_ncd": Fraction(236052463877338152960, 9387480337647754305649),
"h_phillips": Fraction(544118476419313509888, 9387480337647754305649),
"h_trans": Fraction(2026938365560168530840, 9387480337647754305649)
}
if sum(weights_dict.values()) != 1.0:
raise Exception("The sum of the weights of the heuristics must be 1.")
ch = CalculatedHeuristics(
text=cryptogram,
symbols_per_character=symbols_per_character,
min_period_guess=min_period_guess,
max_period_guess=max_period_guess,
heuristics_tuple=heuristics_tuple,
original_cipher="NA"
)
scores = ch.get_scores(
ciphers_tuple=ciphers_tuple,

```
```

    weights_dict=weights_dict
    )
ranking = get_ranking(scores)
print("scores:")
easy_dict_print(scores)
print()
print("ordered by ranking:")
easy_dict_print(ranking, n_decimals=0, ignore_fraction=True)

```
thesis_code/demonstration/create_example_cryptogram.py
```

import os
from coding import corpus, handy_vars
from coding.generatecrypto import cipher_text_with_given_ciphers
from coding.tools import check_current_dir
if ___name___ == '___main___' :
\# Import the data
if not check_current_dir("thesis_code"):
raise Exception("Running from wrong directory!")
\# CLEANING UP A CORPUS TO BE USED FOR CRYPTOGRAMS
print("cleaning the corpus if needed...")
corpus_path = "corpora/english_corpus.txt"
clean_corpus_path = "output/cleaned_text.txt"
output_path_to_file = "output/files/example_cryptogram.txt"
if not os.path.exists(os.path.dirname(output_path_to_file)):
os.makedirs(os.path.dirname(output_path_to_file))
alphabet = handy_vars.ENGLISH_ALPHABET
print("cleaning the corpus if needed...")
if not os.path.exists(clean_corpus_path):
file_length = corpus.clean_corpus(corpus_path, alphabet,
clean_corpus_path)
else:
file_length = corpus.get_number_of_chars_in_file(clean_corpus_path)
\# INPUT
text_length = 500
cipher_to_use = "phillips"
period_or_key_len = 13
\# CRYPTOGRAM GENERATION
text_to_cipher = corpus.get_random_texts_from_file(
clean_corpus_path,
file_length,

```
```

        (text_length, text_length,),
        1
    ) [0]
print("chosen text:")
print(text_to_cipher)
cipher_tuple = (cipher_to_use,)
ciphertext = cipher_text_with_given_ciphers(text_to_cipher, cipher_tuple,
period_or_key_len) [cipher_to_use]
print("ciphered text:")
print(ciphertext)

# writing to file

output_file = open(output_path_to_file, "w")
output_file.write(ciphertext)
output_file.close()

```
thesis_code/demonstration/sa_demonstration.py
```

import os
from fractions import Fraction
from pandas import DataFrame
from classifier.classifier_tools import get_weights, scores_df_to_ranks_df,
calc_correct_guesses
from classifier.heuristics import get_more_heuristics
from classifier.input_objects import HInput, SAInput, FTInput
from classifier.simulated_annealing import simulated_annealing
from classifier.tuple_rating import id_1, id_2
from coding import handy_vars, corpus
from coding.tools import check_current_dir, easy_dict_print
if ___name___ == '___main___':
\# Import the data
if not check_current_dir("thesis_code"):
raise Exception("Running from wrong directory!")
\# CLEANING UP A CORPUS TO BE USED FOR CRYPTOGRAMS
print("cleaning the corpus if needed...")
corpus_path = "corpora/english_corpus.txt"
clean_corpus_path = "output/cleaned_text.txt"
output_path_to_file = "output/files/scores.csv"
if not os.path.exists(os.path.dirname(output_path_to_file)):
os.makedirs(os.path.dirname(output_path_to_file))
alphabet_to_clean = handy_vars.ENGLISH_ALPHABET
if not os.path.exists(clean_corpus_path):
file_length = corpus.clean_corpus(corpus_path, alphabet_to_clean,
clean_corpus_path)
else:

```
```

        file_length = corpus.get_number_of_chars_in_file(clean_corpus_path)
    
# INPUT

identification_function = id_1
heuristics_tuple = handy_vars.ALL_HEURISTICS_TUPLE
ciphers_tuple = handy_vars.ALL_CIPHERS_TUPLE
sa_input = SAInput(
acceptable_fallback_ratio=Fraction(1, 20)
minimum_improvement_value=Fraction(1, 200),
max_failed_attempts_tolerated=3,
ratings_window_size=12,
identification_function=identification_function,
verbose=True
)
he_input = HInput(
clean_corpus_path=clean_corpus_path,
clean_corpus_length=file_length,
min_text_length=500,
max_text_length=500,
n_texts=300,
symbols_per_character=1,
min_period_guess=5,
max_period_guess=20,
heuristics_tuple=handy_vars.ALL_HEURISTICS_TUPLE
)
weight_increment = Fraction(1, len(heuristics_tuple))
initial_rating = Fraction(0)
initial_weights = {}
for heuristic in heuristics_tuple:
initial_weights[heuristic] = Fraction(1, len(heuristics_tuple))

# COMPUTING

calculated_heuristics_list = get_more_heuristics(he_input, ciphers_tuple)
computed_weights, rating = simulated_annealing(
calculated_heuristics_list=calculated_heuristics_list,
current_weight_increment=weight_increment,
ciphers_tuple=ciphers_tuple,
initial_weights=initial_weights,
initial_rating=initial_rating,
sa=sa_input
)
scores_df = DataFrame(columns=["original_cipher"] + list(ciphers_tuple))
for ch in calculated_heuristics_list:
scores_dict = ch.get_scores(ciphers_tuple, computed_weights)

```
```

            scores_df.loc[len(scores_df.index)] = [ch.original_cipher] +
            [scores_dict[a] for a in ciphers_tuple]
    
# OUTPUTTING

print("SCORES DATAFRAME:")
print(scores_df)
print(float(rating), rating)
scores_df.to_csv(output_path_to_file)
easy_dict_print (computed_weights)
output_path_to_file = "output/files/ranks.csv"
ranks_df = scores_df_to_ranks_df(scores_df)
ranks_df.to_csv(output_path_to_file)
print("Number of correctly guessed:", calc_correct_guesses(ranks_df))
print("Number of cryptograms in total:", len(ranks_df.index))
ratio = Fraction(calc_correct_guesses(ranks_df), len(ranks_df.index))
print("Ratio between them:", float(ratio), ratio)

```

\section*{Bibliography}
[Åhl22] Johan Åhlén. Cipher Identifier (online tool) | Boxentriq. https://www.boxentriq. com/code-breaking/cipher-identifier, 2022.
[Ame05] American Cryptogram Association. The ACA and You -- a Handbook for the Members of the American Cryptogram Association. Haverford, 2005.
[Bau02] Friedrich Ludwig Bauer. Decrypted Secrets: Methods and Maxims of Cryptology. Springer, Berlin ; New York, 3rd, rev. and updated ed edition, 2002.
[BCV13] Y. Bengio, A. Courville, and P. Vincent. Representation Learning: A Review and New Perspectives. IEEE Transactions on Pattern Analysis and Machine Intelligence, 35(8):1798-1828, August 2013.
[Cip22] Ciphertext-Classification-Ensemble-SVM. https://github.com/ HappyHackingOrange/Ciphertext-Classification-Ensemble-SVM, April 2022.
[Cry17] Crypto Identifier - Enigmator. https://merricx.github.io/enigmator/cryptanalysis/ crypto_identifier.html, January 2017.
[DCo22] dCode - Solveurs, Crypto, Maths, Codes, Outils en Ligne. https://www.dcode.fr/, 2022.
[DS20] Elonka Dunin and Klaus Schmeh. Codebreaking: A Practical Guide. Robinson, London, 2020.
[dSS86] S. H. C. du Toit, A. G. W. Steyn, and R. H. Stumpf. Graphics for Univariate and Bivariate Data. In S. H. C. du Toit, A. G. W. Steyn, and R. H. Stumpf, editors, Graphical Exploratory Data Analysis, pages 13-35. Springer, New York, NY, 1986.
[Dun20] Elonka Dunin. Elonka's List of Famous Unsolved Codes and Ciphers. https://elonka. com/UnsolvedCodes.html, December 2020.
[Gai89] Helen Fouché Gaines. Cryptanalysis: A Study of Ciphers and Their Solution. Dover Publ, New York, corr. ed edition, 1989.
[GKP94] Ronald L. Graham, Donald Ervin Knuth, and Oren Patashnik. Concrete Mathematics: A Foundation for Computer Science. Addison-Wesley, Reading, Mass, 2nd ed edition, 1994.
[Gov22] Government Communications Headquarters. CyberChef. https://github.com/gchq/CyberChef, June 2022.
[Kah96] David Kahn. The Codebreakers: The Story of Secret Writing. Scribner, New York, rev. ed. edition, 1996.
[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. Science, 220(4598):671-680, May 1983.
[Kut05] Michael H. Kutner, editor. Applied Linear Statistical Models. The McGraw-Hill/Irwin Series Operations and Decision Sciences. McGraw-Hill Irwin, Boston, 5th ed edition, 2005.
\(\left[L^{2}{ }^{+} 21\right]\) Ernst Leierzopf, Nils Kopal, Bernhard Esslinger, Harald Lampesberger, and Eckehard Hermann. A Massive Machine-Learning Approach For Classical Cipher Type Detection Using Feature Engineering. International Conference on Historical Cryptology, pages 111-120, August 2021.
[LMK \({ }^{+}\)21] Ernst Leierzopf, Vasily Mikhalev, Nils Kopal, Bernhard Esslinger, Harald Lampesberger, and Eckehard Hermann. Detection of Classical Cipher Types with FeatureLearning Approaches. In Yue Xu, Rosalind Wang, Anton Lord, Yee Ling Boo, Richi Nayak, Yanchang Zhao, and Graham Williams, editors, Data Mining, Communications in Computer and Information Science, pages 152-164, Singapore, 2021. Springer.
[MR07] António Machiavelo and Rogério Reis. Automated Ciphertext-Only Cryptanalysis of the Bifid Cipher. Cryptologia, 31(2):112-124, March 2007.
[MVOV97] A. J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. Handbook of Applied Cryptography. CRC Press Series on Discrete Mathematics and Its Applications. CRC Press, Boca Raton, 1997.
[NK14] Malte Nuhn and Kevin Knight. Cipher Type Detection. In Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), pages 1769-1773, Doha, Qatar, October 2014. Association for Computational Linguistics.
[Pol89] Polybius. The Histories of Polybius. Translated from the Text of F. Hultsch by Evelyn S. Shuckburgh. London Macmillan, 1889.
[Riv91] Ronald L. Rivest. Cryptography. In Handbook of Theoretical Computer Science (Vol. A): Algorithms and Complexity, pages 617-755. MIT Press, Cambridge, MA, USA, 1991.
[Sas] Yutaka Sasaki. The truth of the F-measure. page 5.
[SBS18] Patrick Schober, Christa Boer, and Lothar A. Schwarte. Correlation Coefficients: Appropriate Use and Interpretation. Anesthesia \(\mathcal{E}^{2}\) Analgesia, 126(5):1763-1768, May 2018.
[ŠČVB21] Valdemar Švábenský, Pavel Čeleda, Jan Vykopal, and Silvia Brišáková. Cybersecurity knowledge and skills taught in capture the flag challenges. Computers \(\mathcal{\xi}\) Security, 102:102154, March 2021.
[Tom22] Satoshi Tomokiyo. Unsolved Historical Ciphers. http://cryptiana.web.fc2.com/code/ unsolved.htm, July 2022.
[Tuk77] John W. (John Wilder) Tukey. Exploratory Data Analysis. Reading, Mass. : AddisonWesley Pub. Co., 1977.
[Wil39] William F. Friedman. Military Cryptanalysis, Part IV - Transposition And Fractionating Systems; By William F. Friedman For The War Department's Chief Signal Officer. NSA, 1939.```

