Optimising Solutions to the Enigma Machine

Submitted April 2019, in partial fulfilment of
the conditions for the award of the degree BSc Computer Science

Jake Learman

14271579

School of Computer Science

University of Nottingham

I hereby declare that this dissertation is all my own work, except as indicated in the text:

Signature _____

Date _____/_____/_____

I hereby declare that I have all necessary rights and consents to publicly distribute this dissertation via the University of Nottingham's e-dissertation archive.

The University of Nottingham

UNITED KINGDOM · CHINA · MALAYSIA

School of Computer Science

Jake Learman

ID Number: 14271579

Supervisor: Graham Hutton

Module Code: G53IDS

2018/19

Abstract

The Enigma machine is one of the greatest electro-mechanical devices in history. Its enormous set of possible configurations provided a mass of cryptographic strength, thus making it a challenge to decrypt at any level. With modern advances in technology, many of these challenges still pose a large computational problem – a problem this project aimed to solve.

In this project, a well-typed implementation of the Enigma machine was designed and implemented in Haskell. This will be done so that a new method of cracking the encryption can be designed using modern Computer Science techniques. The solution developed for this project is then benchmarked in order to assess it success as well as compared with other existing projects with similar aims.

Contents

List of Figures

Chapter 1

Introduction

The Enigma machine is possibly one of the most famous cases of cryptography to have ever been cracked in the Late Modern Era. As computing technology has evolved, the methods used to break encryptions have grown more complex in to keep up with more modern methods of scrambling and enciphering.

Figure 1.1: A M3 Enigma Machine[1]

1.1 Objectives and Motivation

The aim of this project is to recreate the Enigma machine in an attempt to find a more efficient way of cracking the code. The purpose of this project is to see how modern advances in technology and Computer Science affect older Computer Science problems. Furthermore, this project will debate whether or not these advances benefit or detriment the solution of these problems. Additionally, this project will attempt to apply more modern Computer Science techniques in order to optimise the solution. This exercise can be broken down into three main objectives:

[1] `https://goo.gl/UEs5bw`

1. To deconstruct how the Enigma machine works and construct a well-typed functional implementation of its encryption. This will ensure that the process of cracking of the code has a strong and accurately recreated Enigma machine to compete with. This will also enable the overall project aim is met to a high degree of scientific accuracy.

2. To assess how statistical analysis impacted the breaking of the code and how it influenced the rest of the war. This will be of benefit as it will give some understanding of some of the approaches that can be taken when it comes to cracking the Enigma machine. It will also give an understanding of what technology was used during the war.

3. To attempt to find a more efficient way of cracking the Enigma encryption and determine if it would have made a difference to the outcome of WWII. This builds off both the previous objectives as it will both be an attempt to crack a purely implemented Enigma machine as well as provide a basis to see if a more efficient method can be found.

This project will make extensive use of functional programming, cryptographic and mathematical skills in order to both simulate the machine as well as attempting to form a mathematically and functionally sound method of cracking the encryption. Once this has been done, we could further assess the impact of the code breaking and whether or not a better solution could have had a greater impact.

This paper will first cover the background of the project, firstly by examining the structure of the Enigma machine and then analysing how its encryption has been broken in the past.

Chapter 2

Background

In order to find a more optimal route of solving the Enigma conundrum, it is important to fully understand how the Enigma machine works, as well as understanding the complexity of its encryption. Furthermore, by examining more modern attempts of cracking the Enigma machine, one can analyse what they did and attempt to improve upon these solutions.

2.1 Enigma Machine

The Enigma machine is a formidable feat of electro-mechanical engineering (Ostwald & Weierud, 2017). It is a complex mechanical device with a series of rotors and wires which can be rearranged in supernumerary. In order to fully understand how this machine works, it is best to follow the flow of encrypting a phrase as it passes through the various parts. The messages that are sent were typically around two to three short phrases containing direct and straight-forward instructions.

2.1.1 Rotors

The rotors are the main form of scrambling in the Enigma machine. Each rotor is approximately 10cm in diameter and with 26 spring-loaded, electrical contact pins on one side (one for each letter) and corresponding housing for these contacts on the other side for the neighbouring rotors. Inside the body of the rotor is a complex wiring system linking a contact pin to a housing slot on the other side (Deavours & Reeds, 1977).

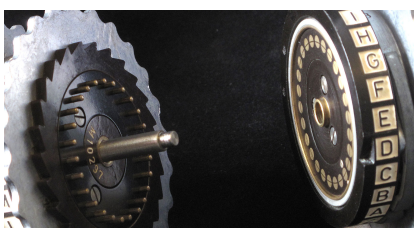

Figure 2.1: A Pair of Enigma Rotors[1]

[1] https://goo.gl/NwzLZH

Using just the rotors, the machine can perform a simple substitution cipher which simply maps one letter to another. The Enigma machine gains security by using a series of three or four rotors in series; this in-turn shifts the encryption from a simple substitution cipher to a more complex poly-alphabetic one, where each rotor would form its own alphabet depending on its position.

In order to avoid the encryption to be cracked easily, a premise of stepping and turnovers were added to the machine to increase its security. When a key is pressed, one or more of the rotors would step by $\frac{1}{26}$ of a rotation. This means that the substitution was different, even if you input the same letter twice. The turnover was another mechanical system hidden in the Enigma machine. It consisted of a ratchet hidden behind each rotor, which was then set up so that when the rotor stepped between two specific letters, the next rotor along would also be stepped.

2.1.2 Reflectors

The Reflectors (or in its manufactured German *Umkehrwalze*), are complex mechanisms that connect the outputs of the last rotor into pairs so that the current could be redirected back through the mechanism via a different route.

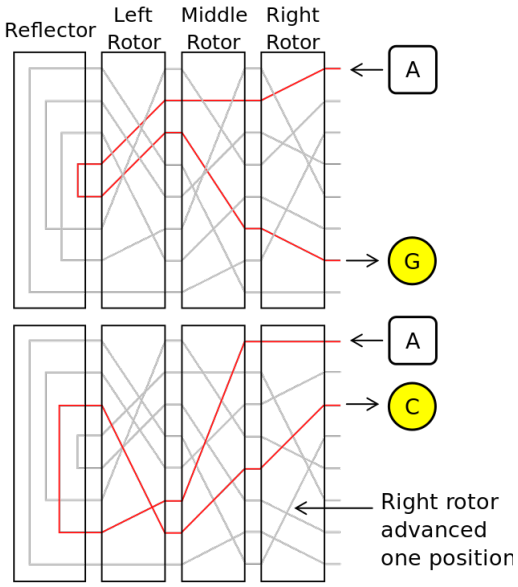

Figure 2.2: The Enigma Machine Encryption Process [2]

The use of a reflector ensures that the Enigma machine was self-reciprocal and obeys the rule of involution: ensuring that encryption was the same as decryption, as well as making sure that no letter could be mapped back to itself. This is a major cryptographic flaw which led to the machine being cracked at Bletchley Park during WWII (Marks & Weierud, 2000).

2.1.3 Plugboard

The Plugboard is the third component of the Enigma machine. It was implemented by the German Navy in order to increase the complexity of the machine's encryption. The

[2] `https://goo.gl/uZbrkm`

machine operator would have a set of wires which could be used to directly map one letter to another, both before and after the signal was passed through the rotors. This means that if you plug the letter E to the letter Q, no matter what gets encrypted to an E, it would be swapped to a Q before it is outputted to the machine operator. This process added a serious amount of cryptographic strength to the machine, which lead to the creation of Alan Turning's Bombe, a much more complex machine then what was seen before in the war (Turing, 1939).

2.2 Related Work

In a post-WWII world, the threats caused by the Enigma machine are minimal, therefore no government felt the need to keep investing in the cracking process. Although the war ended 73 years ago, several of the messages intercepted during the war are still encoded. Now that computers have advanced at such an incredible rate, the concept of cracking these last few messages has become feasible.

2.2.1 M4 Project

German Violinist Stefan Krah started the M4 project to try and crack three original messages sent by the German Navy in 1942. The project made use of distributed computing where a user would download a Python script which connects to a server made up of possible Enigma machine settings. When successful, a C program is downloaded and runs in the background sending feedback to the server if any progress is made. The project began on January 2006 and the first two codes were broken before April 2006. Finally, the project finished with the breaking of the final message in January 2013. Issues did soon arise as the intercepts that were decrypted by the project did contain some 'garble' which were simply used to indicate that some of the text which was decrypted did not actually contain any message information (*Final M4 Letter, 1995*, n.d.). As the timings suggest, this project started out relatively efficient but stalled when a crib was unable to be found. This meant the whole process had to be restarted two or three times before it was finally solved (Campbell, 2011).

2.2.2 Enigma@Home

Enigma@Home is a wrapper between Krah's M4 Project and the University of Berkeley's Open Infrastructure for Network Computing (BOINC). The project is today still attempting to crack messages intercepted from the *Kriegsmarine* during WWII. There is a debate, whether or not these messages were intercepted properly as some letters are missing or due to the encryption of the message being very complex. There are also other theories that the message is a message sent by an officer meaning that it was passed through several Enigma configurations before it was sent. Some messages have been decrypted by improving the algorithm used to solve author Simon Singh's Cipher Challenge (Almgren, Andersson, Granlund, Ivansson, & Ulfberg, 2000), but there is still a need of a greater distributed network to decrypt the final messages.

A comparison between these two projects and this project can be found in Chapter 5 in order to evaluate the success of this project.

Chapter 3

Description Of Work

The work of this project can easily be broken down into three sections based upon the objectives set out in the introduction. The reason for this is due to the fact it allows for greater project planning as well as setting a clear and concise achievable goal for each section. The first section consists of recreating an Enigma machine, this meets the first objective of the project, allowing the other 2 sections to be analysed at the highest degree possible. The second section will consist of implementing a solution to the Enigma machine based upon how it was cracked at Bletchley Park; this sets the standard for what the third section will achieve. The third and final section will consists of analysing and optimising the solution and concluding whether or not modern techniques prove to benefit the decryption process.

3.1 Design & Method

3.1.1 Design of the Project

The overall project itself is designed and implemented as a command line program. This design choice was made as the program will be written in Haskell and is compiled using Glasgow Haskell Compiler (GHC) giving both the Enigma machine and the methods of cracking the encryption unprecedented speed and parallelism (an analysis of which is discussed further in 3.2).

In order to implement the Enigma machine functionally, it is important to first define the problem mathematically such that methods can be deduced intrinsically.

3.1.2 Software Engineering Methods

In order to ensure that this project is completed on time. Several agile development techniques will be used in order to manage time and resource allocation. Breaking down the project into small stages called Sprints allows for a multitude of small easily-achievable goals. Making use of sprints allows for transparency meaning that prioritisation of specific key features can be done in order to see the impact on the rest of the project. Furthermore, having a sprint-based model allows for change, meaning that if any issues occur and sprints need to be pushed back to a later date, it is easy to manage the rest of the project (Martin, 2002).

3.1.3 Enigma Machine

The Enigma machine family is very large and diverse (Hamer, Sullivan, & Weierud, 1993). The Enigma encryption was first used to provide extra security when confidential information was being sent between banks or government institutions. It took just under 10 years of production before any military branch in Germany started using the machine to send messages. Since its adoption in 1929, there were 6 main iterations of the Enigma machine, each one with up to 8 rotors to make combinations (Kruh & Deavours, 2002). There were also 3 reflectors to choose from. For the sake of this project, the Enigma M3 will be simulated, which was used by both the Army and the Navy and was cracked at Bletchley Park in 1941 (Hamer et al., 1993).

Complexity of the Machine

The complexity of an Enigma machine scales with the complexity of its configuration (Ratcliff, 2006). This is demonstrated by the following:

- Selecting 3 rotors out of a possible 5 gives us: $\frac{5!}{(5-3!)}$ meaning that there are 60 combinations of rotors which could be used.
- Then for each rotor there are 26 possible positions of the wiring relative to the rest of the rotor which could be selected, giving $26^3 = 17{,}576$ combinations of starting positions.
- Adding even more complexity, the Plugboard can be given 10 cables which can be configured in a variety of ways, meaning that there is a possibility of $\frac{26!}{(26-20)!\times 2^{10}\times 10!}$ = 108, 531, 557, 954, 820, 000 configurations of the Plugboard

We can therefore state there are:

$$\frac{5!}{(5-3!)} \times 26^3 \times \frac{26!}{(26-20)! \times 2^{10} \times 10!} \tag{3.1}$$

= 158,962,555,217,826,360,000 possible settings of the Enigma machine without accounting for the starting positions of the rotor.

From this, we can see that the Enigma machine is a very complex and intricate device both in terms of the mechanics and mathematics involved (Miller, 1995).

3.1.4 Mathematical Representation of the Enigma Machine

We can also define the transformation of a letter through the Enigma machine as a product of a series of permutations (Rejewski, 1981), for example:

$$E = PRMLReL^{-1}M^{-1}R^{-1}P^{-1} \tag{3.2}$$

where:

- E represents the encryption,
- P represents the Plugboard transformations,
- R, M, L represent the right, middle and left rotors,

- Re represents the reflector.

Once a key is pressed, the rotor turns, changing the transformation such that rotor R is rotated x positions. The transformation can be defined as $\rho^i R \rho^{-i}$ where ρ can be defined as the cyclic permutation of a mapping between 2 letters (Rejewski, 1980). Using this, the Enigma machine can be defined as follows:

$$E = P(\rho^i R \rho^{-i})(\rho^j M \rho^{-j})(\rho^k L \rho^{-k}) U (\rho^k L^{-1} \rho^{-k})(\rho^j M^{-1} \rho^{-j})(\rho^i R^{-1} \rho^{-i}) P^{-1} \tag{3.3}$$

3.2 Implementation and Simulation

All of the implementation of this project will be done using Haskell, a language that is very beneficial for developing programs which makes use of text and the processing of strings and characters. The language's native type safety and type inference make development much more efficient. This reduces problems that can occur with most procedural languages such as null pointer exceptions, or just a general lack of speed due to the nature of how the language is processed, e.g. through a virtual machine. Since this project does not need to run at maximum efficiency, there is no need to use a C-based language since no direct memory access is needed. Furthermore, GHC will make up for any speed lost.

Another reason why a functional language like Haskell is beneficial here is due to its reliance on recursion. When handling long strings of text which need to be operated on, using pure functions allows for immutability meaning that the compiler has greater freedom when it comes to code optimisation. Having this greater freedom means that the compiler can handle function order in a more beneficial manner, as well as perform functions quicker when the tail is called allowing for tail call optimisation.

Implementation of the Enigma Machine

Implementing the Enigma machine is a relatively complex process as there are many intricate and intertwining parts. The implementation in this project begins by declaring a substitution function which takes in a string (in the form of a list of characters) and the letter that is to be substituted. The function then creates a list, paring the string with letters from the alphabet. It then looks up the input character in the list and returns the relevant character. A shift function is then defined, which acts as a Caesar shift, taking the shift key k as an input and returning a character shifted k places in the alphabet. The cycle function is used to wrap the end of the alphabet back to the beginning.

```
> substitute :: String -> Char -> Char
> substitute s c = fromMaybe c $ lookup c $ zip alphabet s

> shift :: Char -> Char -> Char
> shift k = substitute $ dropWhile (/= k) $ cycle alphabet
```

Now that these helper functions have been defined, we can then define a new data type for the Enigma machine itself. This is due to the fact the Enigma machine is made up of various parts. In order to handle its use properly, it is easier to create an object to pass parameters into instead of creating a new Enigma machine object whenever it needs to be used.

As seen in 2.1, the Enigma machine has 3 major components - the rotors, the reflector

and the Plugboard - each of which has its own specific set of settings, all of which are some form of shifting letters from one to another. The *Grundstellung* is the starting position of the rotors. This was chosen by the operator and was different for every message, which is one of the reasons why the encryption is so strong. The *Ringstellung* is the initial ring setting relative to the rotor discs, effectively an initialisation vector for the encryption adding a level of randomness to the machine's encryption. The Plugboard is a variable wiring system that would be manually reconfigured by the operator using patch cables. One could manually map letters together forming a 'steckered pair' of letters, which swapped the letter respectively both before and after the rotor scrambling.

```
> data Enigma = Enigma {
>  rotors :: [(String, String)], reflector :: String,
>  grundstellung :: String, ringstellung :: String,
>  plugboard :: String } deriving (Eq, Show)
```

We can then define the various rotors and reflectors which can be included when configuring the machine. The letters at the end of the rotor definitions are the turnover notches (Hamer, 1997).

```
> rotorI  = ("EKMFLGDQVZNTOWYHXUSPAIBRCJ", "Q")
> rotorII = ("AJDKSIRUXBLHWTMCQGZNPYFVOE", "E")
> rotorIII = ("BDFHJLCPRTXVZNYEIWGAKMUSQO", "V")
> rotorIV = ("ESOVPZJAYQUIRHXLNFTGKDCMWB", "J")
> rotorV  = ("VZBRGITYUPSDNHLXAWMJQOFECK", "Z")
> rotorVI = ("JPGVOUMFYQBENHZRDKASXLICTW", "M")
> rotorVII = ("NZJHGRCXMYSWBOUFAIVLPEKQDT", "Z")
> rotorVIII = ("FKQHTLXOCBJSPDZRAMEWNIUYGV", "M")


> reflectorA = "EJMZALYXVBWFCRQUONTSPIKHGD"
> reflectorB = "YRUHQSLDPXNGOKMIEBFZCWVJAT"
> reflectorC = "FVPJIAOYEDRZXWGCTKUQSBNMHL"
```

An example of the M3 Engima Machine can be defined as follows:

```
> enigmaMachine = Enigma {
>     rotors = [rotorI, rotorII, rotorIII],
>     reflector = reflectorB,
>     grundstellung = "AAA",
>     ringstellung = "AAA",
>     plugboard = alphabet }
```

This will create an Enigma machine with rotors 1-3, reflector B and both the rotors and rotor positions will all be lined as the letter A. There is also no Plugboard defined in this machine so that letters aren't re-scrambled before and after the rotors - this of course means that the rotors are the main cause of the encryption with the exemption the reflector. Now the actual encryption process can be defined. The rotation function is used to simulate a rotor shifting. A case analysis of the boolean value is performed to check what the starting value of each rotor is, and it then also checks if that is an element in next rotor. Here, sR refers to the starting value in that rotor. nR refers to the value in the next rotor.

```
> rotation :: Enigma -> Enigma
> rotation r = r {
>     grundstellung = [bool sR1 (shift 'B' sR1) $ sR2 `elem` nR2,
>     bool sR2 (shift 'B' sR2) $ sR2 `elem` nR2 || sR3 `elem` nR3, shift 'B'
    sR3
>    ]}
>     where
>      [sR1, sR2, sR3] = grundstellung r
>      [nR1, nR2, nR3] = snd <$> rotors r
```

In order to apply the rotation to a string of characters, we first use applyShift and conjugate the string with the alphabet so that when we apply the rotation, the machine's defined settings are applied correctly.

```
> applyShift :: String -> Char -> String
> applyShift cs key = unshift key . substitute cs . shift key <$> alphabet

> applyRotation :: Enigma -> [String]
> applyRotation cs = zipWith applyShift (fst <$> rotors cs)
>     $ zipWith unshift (ringstellung cs) $ grundstellung cs
```

The final part of this process involves passing the string of characters through the reflector and plugboard. This is done by the findMap function which composes all of the previous functions into a helper function for the encryptChar function. This first checks that the input is actually a letter, then processes the character to finally return the machine with the rotate rotors and the encrypted char.

```
> findMap :: Enigma -> Char -> Char
> findMap e = plugboardOutput . unsubstitute rotatedInput . substitute
    (reflector e)
>    .  substitute rotatedInput . plugboardOutput
>    where
>      rotatedInput = foldr1 (.) (substitute <$> applyRotation e) <$> alphabet
>      plugboardOutput = substitute $ plugboard e

> encryptChar :: Enigma -> Char -> (Enigma, Char)
> encryptChar machine c = bool(machine, c)(machine', findMap machine' c) $
    isLetter c
>    where machine' = rotation machine
```

The final two functions used to encrypt a text involves mapping encryptChar over a string, then folding the output leftward to form something in a similar format to the input text. The reason the Traversable type was used is due to its ability to allow effects to run while the data structure is being rebuilt. In this case, while the string is being encrypted and rebuilt, the rotor shift can occur, thus functioning as an Enigma machine should.

```
> encryption :: Traversable t => Enigma -> t Char -> t Char
> encryption machine = snd . mapAccumL encryptChar machine

> runMachine :: Traversable t => t Char -> t Char
> runMachine cs = encryption (enigmaMachine) cs
```

```
-------------------------
|An example of encryption|
-------------------------
ghci Enigma.lhs
*Enigma> runMachine "HASKELL"
"IDPHRIU"
*Enigma> runMachine "IDPHRIU"
"HASKELL"
```

The demonstrated example shows that the machine is self-reciprocal and obeys the rule of involution, therefore proving itself to be a worthy simulation of the machine.

Chapter 4

Cracking the Code

At the time the Enigma machine was first being used, many of its users often underestimated their belief in the encryption and how 'unbreakable' it is. In this chapter, the various attempts and methods of decryption will be analysed and the final of which, the solution derived at Bletchley Park, will be implemented (see 4.1). The aim of this chapter is to meet the second objective of the project and set up the implementation so that code and its optimisations can be made in Chapter 5.

4.0.1 Polish Progress

After the first World War, Poland secretly intercepted radio messages from the German Navy who were active near Polish waters in the 1920s (Bloch & Deavours, 1987). In 1926, the Polish were no longer able to decrypt these messages due to the incorporation of Enigma into the German Navy. It is still unknown how the Polish acquired information about the Enigma machine; it is believed that the Polish Secret Service had a hand in the matter (Rejewski, 1981). When the Mathematician Marian Rejewski was able to derive the rotor wiring using linear algebra (see 3.1.3), this along with other information purchased from German defectors, the Polish built a copy of Enigma ("Polish Mathematicians Finding Patterns in Enigma Messages, author=Christensen, Chris, journal=Mathematics Magazine, volume=80, number=4, pages=247–273, year=2007, publisher=Taylor & Francis", n.d.).

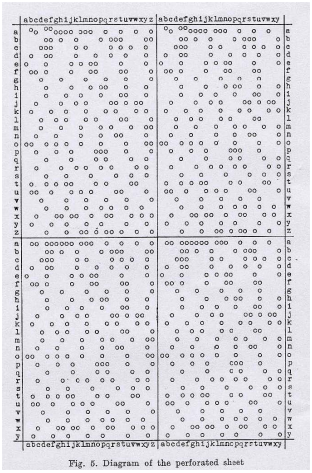

Figure 4.1: A Zygalski Sheet [1]

A Zygalski Sheet was designed and implemented by the Polish Cipher Bureau as a method of simulating the Enigma machine. 26 sheets were placed atop each other per each rotor configuration in the Enigma machine. Each sheet has a 26 x 26 matrix, and each letter was punched as a hole in the top sheet revealing possible paths in sheets below. This method seemed promising originally, but Germany's fast-paced technological advancements quickly rendered this method useless (Stengers, 1984). On 15th December 1938, the German Army added 2 more rotors to the selection choice, changing the number of possible rotor combinations from 6 to 60. As such, the Poles could only read messages that used neither of the new rotors. Rejewski then developed a method of using a stack of 58 Zygalski Sheets as well as a combination of Permutation Theory and Linear Algebra. (Alster, Urbanowicz, & Williams, 2011) Although this method proved to be successful initially, its ability to cut out a lot of the manual work was quickly became redundant, due to the ever-changing nature of Enigma. This is due to the Enigma's constantly changing set of components.

4.0.2 Bletchley Park

In 1938, The British Government Code & Cipher School (GC&CS) moved into Bletchley Park with the aim of cracking Enigma (Beesly, 2000). Along with MI6, they began recruiting mathematicians and cryptologists including Alan Turing and Derek Taunt. Most of the success that the Polish had in cracking Enigma was based around the repetition of the indicator. However once Turing began research, he wanted to find methods that did not rely on this weakness as they did not want the Germans to be aware that the code had been cracked. This led to the development of Crib based decryption(Hodges, 2012).

4.1 Crib-Based Attacks

The strategy of breaking a code by guessing part of the deciphered text and then using that to deduce the process of encryption is called a known-plaintext attack. The team at Bletchley Park used the word crib to describe the part of guessed plain-text. A crib-based attack is best demonstrated using an example.

Suppose we have the following ciphertext: *UAENFVRLBZPWMEPMIHFSRJXFMJKWRAXQEZ*. One deduction that Bletchley made was that intercepted messages often contained text such as weather report or phrases such as "Nothing to report" (in German, *Keine Besonderen Ereignisse*). The first step of a crib-based attack is called alignment. This is done by matching the crib with possible locations within the cipher text, so that all the rules of Enigma are followed. This means that letters cannot be mapped to themselves, but during initial alignment a crash can occur. A crash refers to positions where letters are in the same position in both the crib and the cipher text. This means that the cipher text needs shifting. The difference in positions between the initial alignment and the final alignment is called the offset. The method of calculating which is as follows:

Calculating the offset After shifting the cipher text by 5 places, we can see that there are no crashes between the crib and the cipher text. This means that every letter in the crib can be encrypted to any letter in the cipher text.

[1] `https://goo.gl/k594vE`

$$U\ A\ E\ (N)\ F\ V\ R\ L\ B\ Z\ P\ W\ M\ (E)\ P\ M\ I\ H\ F\ S\ R\ J\ X\ F\ M\ J\ K\ W\ R\ A\ X\ Q\ E\ Z.....$$

$$K\ E\ I\ (N)\ E\ B\ E\ S\ O\ N\ D\ E\ R\ (E)\ N\ E\ R\ E\ I\ G\ N\ I\ S\ S\ E$$

Offset = 1

$$A\ (E)\ N\ F\ V\ R\ L\ B\ Z\ P\ W\ M\ E\ P\ M\ I\ H\ F\ S\ R\ J\ X\ F\ M\ J\ K\ W\ R\ A\ X\ Q\ E\ Z.....$$

$$K\ (E)\ I\ N\ E\ B\ E\ S\ O\ N\ D\ E\ R\ E\ N\ E\ R\ E\ I\ G\ N\ I\ S\ S\ E$$

Offset = 2

$$E\ N\ F\ V\ R\ L\ B\ Z\ P\ W\ M\ (E)\ P\ M\ I\ H\ F\ S\ R\ J\ X\ F\ M\ J\ K\ W\ R\ A\ X\ Q\ E\ Z.....$$

$$K\ E\ I\ N\ E\ B\ E\ S\ O\ N\ D\ (E)\ R\ E\ N\ E\ R\ E\ I\ G\ N\ I\ S\ S\ E$$

Offset = 3

$$N\ F\ V\ R\ L\ (B)\ Z\ P\ W\ M\ E\ P\ M\ I\ H\ F\ S\ R\ J\ X\ F\ M\ J\ K\ W\ R\ A\ X\ Q\ E\ Z.....$$

$$K\ E\ I\ N\ E\ (B)\ E\ S\ O\ N\ D\ E\ R\ E\ N\ E\ R\ E\ I\ G\ N\ I\ S\ S\ E$$

Offset = 4

$$F\ V\ R\ L\ B\ Z\ P\ W\ M\ E\ P\ M\ I\ H\ F\ S\ (R)\ J\ X\ F\ M\ J\ K\ W\ R\ A\ X\ Q\ E\ Z.....$$

$$K\ E\ I\ N\ E\ B\ E\ S\ O\ N\ D\ E\ R\ E\ N\ E\ (R)\ E\ I\ G\ N\ I\ S\ S\ E$$

Offset = 5

$$V\ R\ L\ B\ Z\ P\ W\ M\ E\ P\ M\ I\ H\ F\ S\ R\ J\ X\ F\ M\ J\ K\ W\ R\ A\ X\ Q\ E\ Z.....$$

$$K\ E\ I\ N\ E\ B\ E\ S\ O\ N\ D\ E\ R\ E\ N\ E\ R\ E\ I\ G\ N\ I\ S\ S\ E$$

Figure 4.2: An example of crib alignment

Once the offset has been calculated, we can then attempt to find mappings from the crib to the cipher text. This is done by finding loops. A loop is a closed cycle containing pairings of letters between the crib and the cipher text. Using table 4.1, we generate a graph to visualise the mappings between the crib and the cipher text.

V	R	L	B	Z	P	W	M	E	P	M	I	H	F	S	R	J	X	F	M	J	K	W	R	A
K	E	I	N	E	B	E	S	O	N	D	E	R	E	N	E	R	E	I	G	N	I	S	S	E
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24

Table 4.1: Alignment of the Crib and Cipher Text

Once the menu has been plotted, it becomes clear where loops can be found as they are best represented by cycles found in the graph. The more node the cycle traverses, the more mappings are known between letters. In 4.3, we can see that we have one loop of length 2 (E, R, E), two loops of length 3 (F, I, E, F) and (N, B, P, N), two loops of length 4 (R, J, N, S, R) and (E, R, S, W, E) and one loop of length 6 (E, R, J, N, S, W, E). It is clear that mappings of these letters are only valid when using the offset of 5. This means that if another valid offset is found, a new menu would have to be generated using the new offset in order to find loops.

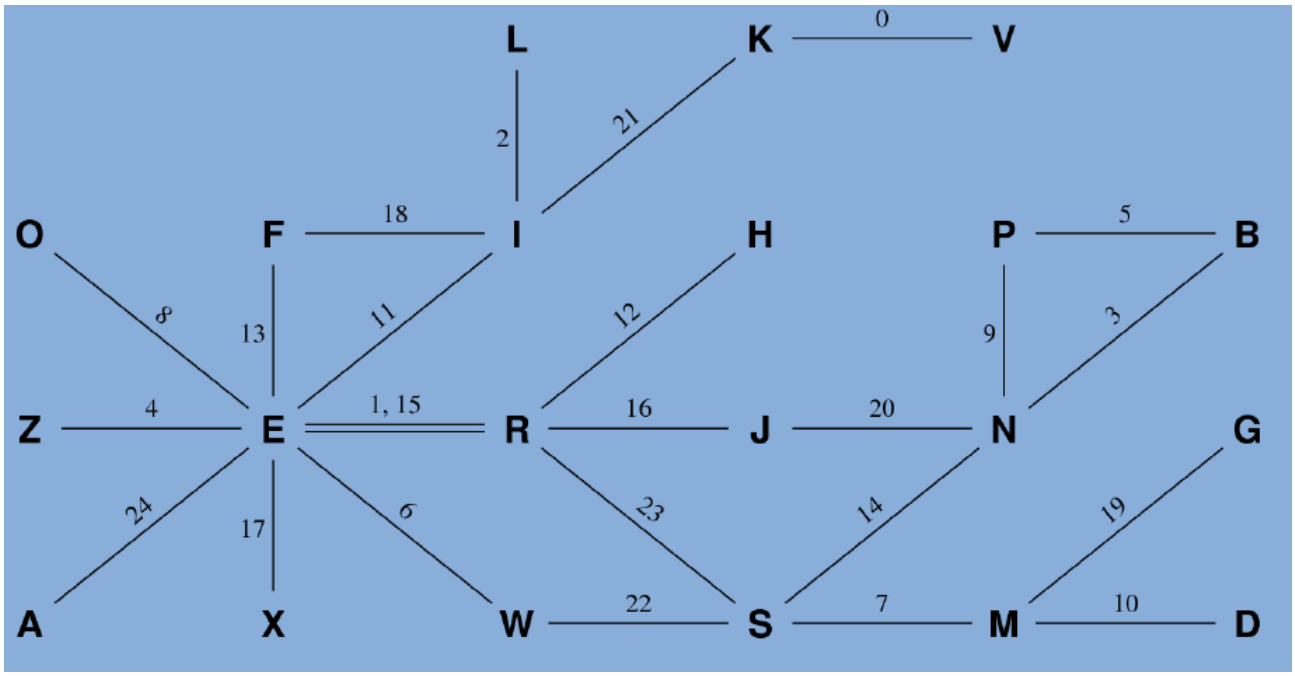

Figure 4.3: A menu showing the mappings between a crib and the cipher text

4.2 Design of the Cracking Algorithm

In order to fully benefit from the use of Haskell for this project, as well as provide a different approach to deciphering Enigma cipher text, a new method will be designed. It begins relatively similar to how the code was cracked at Bletchley Park. Once a menu has been found between a crib and some cipher text, one can judge how certain letters are mapped to others. At this stage a superset of rotor combinations can be found so that the alphabet can be encrypted with every possible solution. Once all the encrypted alphabets have been found, a simple matching algorithm can be used so that the best match of mappings between the menu and a specific alphabet can be found. At this stage the rotors used to encrypt the best matched alphabet can be found. These therefore must be the rotors used to encrypt the cipher text.

4.3 The Bombe

The discovery of finding loops and cycles within menus gave Bletchley Park an extensive breakthrough in the cracking process. The fact that the loop pattern effectively makes the encryption independent of the plugboard means that it is then possible to retrieve the plain text by simply running the Enigma machine using all possible combinations of rotors. To speed this process up and to remove the amount of manual rotor switching needed, the Bombe was created (Welchman, 1982).

The Bombe was set up with a particular set of loops in order to find a set of feasible rotor positions so that Enigma can be decrypted. The Bombe machine effectively was a complex trial and error system where for each cycle it ran, it would assume that the rotor settings at that time were correct (Mackenzie, 1996).

[2] `https://goo.gl/8uMV6P`

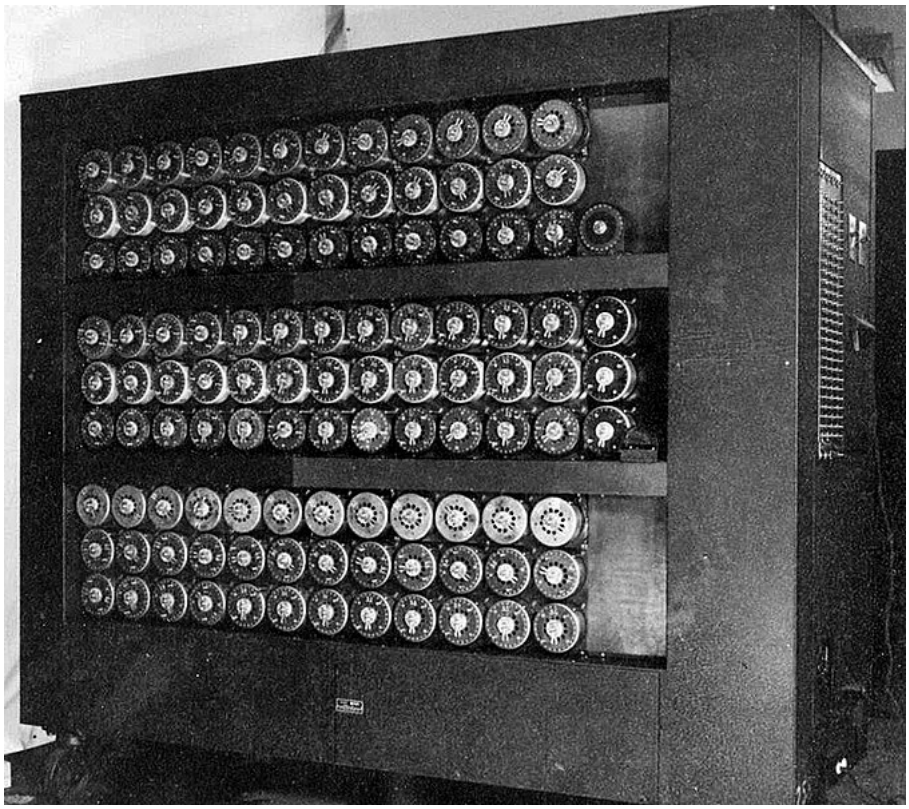

Figure 4.4: A Wartime Bombe in Bletchley Park [2]

If using that series of rotor settings led to a crash or a contradiction in letter mapping, it would shift to the next set of rotor settings and attempt those.

Structure of the Bombe

A standard German Enigma machine has at three rotors in use when encoding and decoding messages. A standard Bombe built at Bletchley Park was able to simulate running 36 Enigma machines, each of which was given three drums in order to represent each rotor. Different colours were using to label each drum where each colour would represent a different rotor found in the set used by the Germans. A Bombe was able to run up to 3 cycles at once, where each cycle would have a unique menu for a crib cipher text combination.

If the machine was able to complete a running cycle without encountering a contradiction, the machine would stop. At this point the machine's operator would record the settings the Bombe outputted; this would become a 'Candidate Solution'. Once several Candidate Solutions were found, they are read by the Bombe several times more in order to reduce as many stops as possible. A design flaw in the Bombe was that it would false stop, often several times per rotor setting, which would lead to several possible candidate solutions being inaccurate meaning that the re-running of these settings were necessary (Gaj & Orłowski, 2003). Once the operator was happy with the candidate, it would be sent to a different hut in Bletchley Park for further cryptanalysis (see 4.3.1). At this point, rotor settings were finalised, and the cipher text was fed through a modified TypeX machine in order to translate the cipher text to the plain text.

[3] `https://goo.gl/CBpJbE`

Figure 4.5: A Modified TypeX Machine able to decipher Enigma [3]

4.3.1 Complexities of the Plugboard

The most complex part of the decryption process was figuring out how the plugboard was set up for each configuration. The plugboard was the main source of the Enigma's cryptographic strength. Turing came up with a solution to test possible plugboard combinations based upon the relationships between the values within (Good, 2000).

For a plugboard P and a scrambler S, there are plugboard connections such that a Typex machine can be set up so the crib letter A can be compared with some cipher text W. If there is a match such that $A \subseteq W$, the next letter in the crib would be tested. Turing provided a solution that if $P(A)$ and $P(W)$ are unknown, the crib still provide a known relationship between the two.

For example: if we have a crib and cipher text alignment where we see that a letter in a crib A aligns with character T in the cipher text at index 10, where P is used to represent the plugboard encryption and $P(A)$ can be represented by X we can represent that as:

$$T = P(S_{10}(P(A)))$$

As we know the Enigma encryption is self-reciprocal and obeys the rule of involution, we can reapply the plugboard P as: $P(T) = S_{10}(P(A))$. This means that for any case, whether the values of $P(A)$ and $P(T)$ are known or not, we can state that:

$$P(T) = S_{10}(P(A)) = S_{10}(X) = A$$

Once one plugboard connection was proven, it became easier to reduce the number of unknown letters thus speeding up the decryption process. While all the mathematical proof works efficiently in theory, the Bombe was often inefficient to run as it required impractically long cribs in order to rule out the majority of settings, and then required more specific and complex operations to reduce the small set remaining to optimal candidate solutions.

4.4 Implementation

The first function used in this stage of cracking the encryption is alignCrib. alignCrib is responsible for comparing the crib and the encrypted string in order to see if any character is in the same index in both strings. This means that if a match is found, a shift will be needed to ensure that no elements in a common index are the same.

```
> alignCrib :: String -> String -> String
> alignCrib [] [] = []
> alignCrib crib encryptedOutput = map fst . filter (uncurry (==)) $ zip crib
    encryptedOutput
```

Once the crib is aligned with the cipher text, this is then tested using a boolean value. The function filter alignment will return a boolean whether or not a crib is aligned with cipher text correctly. If a realignment is needed, the function shiftCrib is called, this will add a space to the beginning of a crib such that the crib can be compared with the cipher text again.

```
> filterAlignment :: String -> String -> Bool
> filterAlignment crib encryptedOutput = if alignCrib crib encryptedOutput ==
    [] then True else False

> shiftCrib :: String -> String
> shiftCrib crib = " " ++ crib
```

Once a valid shift is found, it is then necessary to find the best offset for the crib. This is done by ensuring that there are no crashes for the entire length of the crib when compared with the cipher text. The function findNoCrashes is responsible for this. It takes in a crib and some cipher text and uses an if then else clause to ensure that a filtered alignment is used. If no shift is needed, the function will return a list of tuples, where each list index refers to the letters at each index in both the crib and the cipher text. If a shift is needed then the function is called with the shifted crib.

For example, if we have a string "HELLO" and an encrypted string "ILBDA" (hello encrypted), we would get returned [('I','H'),('L','E'),('B','L'),('D','L')] as a list of pairs.

```
> findNoCrashes :: String -> String -> [(Char, Char)]
> findNoCrashes crib encrypted = if filterAlignment crib encrypted then (zip
    crib encrypted) else findNoCrashes (shiftCrib crib) encrypted
```

As explained in 4.1, once a valid alignment is found, the next step involves finding loops between mappings of letters. setUpForLoop removes any pairs containing spaces such that the list of tuples has already handled the offset. This reduces processing time as effectively null values are removed from the list. The tuples are then ordered by in descending order by either their first or second value in the tuples, which are then grouped by common elements.

```
> setUpForLoop :: [(Char, Char)] -> [(Char, Char)]
> setUpForLoop alignedText = [c | c <- alignedText, fst c /= ' ']

> orderTuples :: String -> String -> ((Char, Char) -> Char) -> [(Char, Char)]
> orderTuples crib encrypted f = sortBy (comparing f) (setUpForLoop
    (findNoCrashes crib encrypted))

> groupTuples :: String -> String -> ((Char, Char) -> Char) -> [[(Char, Char)]]
> groupTuples crib encrypted f = groupBy ((==) `on` f) ((orderTuples crib
    encrypted f))
```

Now that we have the letters paired, we can then begin to set up the letters for menu construction. This is best done by removing all duplicate pairs so that each letter is only paired with another one, reducing the chance of any small closed loops being found.
The flatten function effectively removes any extra depth of the letter pairings ensuring that all the pairs are wrapped in the list type only once. The tuples both ordered by their first and second element are the concatenated to ensure that all possibilities of letters are considered. Finally, groupTuples' removes all duplicate elements from the combined list such that only valid pairs are left, giving the best set of pairings to build the menu with.

```
> flatten :: [[a]] -> [a]
> flatten xs = (\z n -> List.foldr (\x y -> List.foldr z y x) n xs) (:) []

> joinLists :: String -> String -> [[(Char, Char)]]
> joinLists crib encrypted = (groupTuples crib encrypted fst) ++ (groupTuples
    crib encrypted snd)

> groupTuples' :: String -> String -> [(Char, Char)]
> groupTuples' crib encrypted = nub (flatten (joinLists crib encrypted))
```

The first step in menu generation is grouping elements by the vertices they are linked to. This means that we can short-cut building the whole menu by using the vertex with the most edges surrounding it.

```
> groupByVertex :: (Eq a, Ord a) => [(a, b)] -> [(a, [b])]
> groupByVertex = map (\l -> (fst . head $ l, map snd l)) . groupBy ((==) `on`
    fst) . sortBy (comparing fst)
```

The most effective way to represent a menu is by using a list of integers. In order to make the loop finding operations more efficient, a graph-growing algorithm was used instead of generating the whole menu. The findMenu function is the main function used to find the menu for the input list of letter pairs. The boolDups function will return a boolean depending on if duplicates are found in the list input. This is used to further eliminate any extra elements found in the list of pairs.

```
> type Menu = [Int]

> findMenu :: [(Char, Char)] -> [Menu]
> findMenu crib = growMenu (findLink crib)
```

```
> boolDups :: Eq a => [a] -> Bool
> boolDups [] = True
> boolDups [_] = True
> boolDups (h : t)
>  |(elem h t) = False
>  |otherwise = boolDups t
```

As explained above, the findMenu function handles the bulk of menu growth. It makes use of the following two functions: findLink and growMenu. The findLink function finds a list of menus based upon an input of a crib and some cipher text. It begins by splitting the tuple (labelled crib) into segments. For each segment, a series of sub-menus are made using the relationship between the elements within the crib and the pair-link structure of each edge in the menu. growMenu is a more complex function which recursively expands the menu by linking all the smaller sub-menus together if they contain common elements. If there is a single common element then there must be an edge connecting each node, effectively by using Primm's algorithm for finding minimum spanning trees. For the null case, a list of menus is returned where no more joins between menus can be made. If there is a case to be made, i.e. a menu to be linked, a series of list comprehensions are made where small cyclic and irrelevant sub-menus are removed.

```
> findLink :: [(Char, Char)] -> [Menu]
> findLink crib = [[xs, ys] |(ys, y) <- m,(xs,x) <- c, y ==x]
>  where
>   cycle = [0 .. ((length crib) - 1)]
>   (ms, cs) = unzip crib
>   m = zip cycle ms
>   c = zip cycle cs

> growMenu :: [Menu] -> [Menu]
> growMenu ms
>  |null joins = ms
>  |otherwise = growMenu noDupsMs
>  where
>   joins = [(joinMenu x y) | x <-ms, y <- ms, x /= y, (findOverlaps x y) > 0]
>   filteredJoins = [m | m <-joins, boolDups m]
>   filteredMenu = removeSubMenus filteredJoins
>   noDupsMs = nub filteredMenu
```

Due to the size of growMenu, several helper functions are defined. The findOverlaps function will take two lists and return any overlapping elements. The lists are indexed, and the function returns the number of shared elements. The lists are labelled left and right such that when joined later they meet in the correct place. joinMenu effectively concatenates two lists while removing any overlapping elements. removeSubMenus will remove any menus of shorter length than the longest menu, which means that we will find a greater number of mappings between the crib and cipher text.

```
> findOverlaps :: [Int] -> [Int] -> Int
> findOverlaps xs ys = findOverlaps' xs ys 0

> findOverlaps' :: [Int] -> [Int] -> Int-> Int
```

```
> findOverlaps' xs ys n
>  | n == length xs = 0
>  |left == right = (length left)
>  |otherwise = findOverlaps' xs ys (n+1)
>  where
>   left = (drop n xs)
>   right = (take (length left) ys)

> joinMenu :: [Int] -> [Int] -> [Int]
> joinMenu ms ns = (take ((length ms) - n) ms) ++ ns
>  where
>   n = findOverlaps ms ns

> removeSubMenus :: [[Int]] -> [[Int]]
> removeSubMenus ms = [m | m <-ms, null [ns | ns <- ms, (length ns) > (length
    m), m == (take (length m) ns)]]
```

Once findMenu has been called with a valid crib and cipher text combination, it will return a list of letters which represent the biggest loops found in the menu. Once the biggest loop has been found, it is then possible to deduce rotor settings for the machine as a relationship between the cipher text and some of its plain text letters can be derived ("The Cryptanalysis of a ThreeRotor Machine Using a Genetic Algorithm., author=Bagnall, Anthony J and McKeown, Geoff P and Rayward-Smith, Victor J, booktitle=ICGA, pages=712–718, year=1997", n.d.).

Example of Menu Growth

For example, if we have a phrase "BATTLEFIELD" and an encrypted string "ADHFUNDBWPF", we will get the following menu returned:

```
> menuToChar(findMenu(groupTuples' "BATTLEFIELD" "ADHFUNDBWPF"))
["GBACF"]
```

In order to implement the Bombe in an appropriate method to decrypt the Enigma implemented for this project (see 3.2), a Bombe data structure would need to be implemented. It has the same type of Enigma as it will make use of similar inputs. Note that the rotors, *Grundstellung* and *Ringstellung* are left empty. This is due to the fact that this is what the program will calculate. Reflector B is left the same, as reflector B was the main reflector used throughout the war and is the strongest out of the 3 available. A rotor list is also established in order to iterate through and combine, such that a superset can be used in the Bombe ensuring every combination of rotors is used.

```
> bombe :: Enigma
> bombe = Enigma {
>     rotors = [],
>     reflector = reflectorB,
>     grundstellung = [],
>     ringstellung = [],
>     plugboard = alphabet}

> rotorList :: [Rotor]
```

```
> rotorList = [rotorI, rotorII, rotorIII, rotorIV, rotorV, rotorVI, rotorVII,
    rotorVIII]
```

---

groupInNs is a function used to produce a superset of all rotor combinations. In this case as we are using the M3 Enigma, rotors are grouped into sets of 3. The function makes use of a monadic mapping to ensure the higher order function nature of const is maintained and that the grouping size is limited to n. The fetchRotorCombinations will return a set of rotors out of the list of rotors, meaning that when the Bombe is running, it will be easy to iterate through the various combinations.

```
> groupInNs :: Eq a => [a] -> Int -> [[a]]
> groupInNs xs n = filter ((n==) . length . nub) $ mapM (const xs) [1..n]

> fetchRotorCombination :: Eq a => [a] -> Int -> [a]
> fetchRotorCombination rs n = (groupInNs rs 3) !! n
```

Now that all possible rotor combinations have been found, we then need to find every possible alphabet encryption so that we can easily compare the cipher text with all the encrypted alphabets. This is done using the runBombe function. A Bombe (being a sub-class of the Enigma datatype) runs a string through it and encrypts the string using the rotors returned from fetchRotorCombaination.

```
> runBombe :: Traversable t => t Char -> Int -> t Char
> runBombe cs n = encryption (e) cs
>   where e = Enigma {
>     rotors = (fetchRotorCombination rotorList n),
>     reflector = reflectorB, grundstellung = "AAA",
>     ringstellung = "AAA", plugboard = alphabet }
```

The rotorLength is used as a limiter in recursive functions to ensure no out of bounds errors can occur. limitBombe is used to handle any missing return value errors. runBombe' uses list comprehension to iterate through the list of rotors such that every rotor combination encrypts the alphabet once.

```
> rotorLength :: Int
> rotorLength = length (groupInNs rotorList 3)

> limitBombe :: Int -> Maybe Int
> limitBombe n = if n < rotorLength then Just n else Nothing

> runBombe' :: Traversable t => t Char -> [t Char]
> runBombe' cs = [runBombe cs n | n <- [0 .. (rotorLength -1)]]
```

Once all the possible alphabet encryptions have been found, they are then zipped with a decrypted alphabet so that a comparison can be made later when attempting to crack the code. prepBreak makes use of a predicate as a placeholder for the alphabet when the function is called in its prime form. An integer is taken in as well, so that an encrypted alphabet can be fetched from a specific index. This will allow for a recursive variation of the function to be called as prepBreak'. prepBreak' is a traversable version of the above function, which traverses through the list of rotors in order to zip each encrypted alphabet with the standard alphabet in order to create a list of tuple letter pairs where one of which is encrypted.

```
> prepBreak :: p -> Int -> [(Char, Char)]
> prepBreak xs n = zip alphabet $ runBombe alphabet n

> prepBreak' :: Traversable t => t Char -> [[(Char, Char)]]
> prepBreak' xs = [prepBreak x n| x <- runBombe' xs, n <- [0 .. (rotorLength
    -1)]]
```

In order to transpose the menu found from the crib and cipher text into an appropriate form, cribUp is used. cribUp takes a menu and splits it into a list of tuples such that every neighbouring letter is paired together. An equality check is used to ensure that no two letters are the same within a pair. findClosestMatch is used to find the closest match between the mappings of the menu and the encrypted alphabet. findClosestMatch' is used to iterate through the list of encrypted alphabets to ensure that each and every encrypted alphabet is compared to.

```
> cribUp :: Eq a => [a] -> [(a,a)]
> cribUp menu = [(x, y) | x <- menu, y <- menu , x /= y]

> findClosestMatch :: [String] -> Int -> [(Char, Char)]
> findClosestMatch menu n = intersect (cribUp (head menu)) (prepBreak alphabet
    n)

> findClosestMatch' :: [String] -> [[(Char, Char)]]
> findClosestMatch' menu = [findClosestMatch menu n | n <- [0 .. (rotorLength
    -1)]]
```

The matches are then sorted by length such that the alphabet with the most matches is found. fetchClosestMatch uses the list of matches to find the alphabet with the largest match between the menu and the encrypted alphabet. The greater the list of matches, the more direct mappings between letters can be found.

```
> sortMatches :: [[a]] -> [[a]]
> sortMatches = sortBy (comparing length)

> fetchClosestMatch :: [String] -> [(Char, Char)]
> fetchClosestMatch menu = head(reverse(sortMatches(findClosestMatch' menu)))
```

findRotorCombination uses list comprehension in order to find if the closest matches in each menu can be found in a set encrypted alphabet. A maybe wrapper is used to handle any null elements. filterRotors is used to remove all rotor combinations that return the nothing type, meaning that only valid letters indices are found. findRotorCombination is used to find which specific set of rotors is used to decrypt the cipher text used to form the menu back to its original plain-text. If this function returns successfully, Enigma is broken.

```
> findRotorCombination :: [String] -> [Maybe Int]
> findRotorCombination menu = [elemIndex x y | x <- (fetchClosestMatch menu),
    y <- (prepBreak' alphabet)]
```

```
> filterRotors :: [String] -> [Maybe Int]
> filterRotors menu = filter (/= Nothing) (findRotorCombination menu)


> findRotorCombination' :: [String] -> [Rotor]
> findRotorCombination' menu = fetchRotorCombination rotorList ((fromJust(head
    (filterRotors menu))) - 1)
```

Once a valid set of rotors is returned, we can pass them into an Enigma machine in order to retrieve the decrypted text.

```
> breakEnigma :: [(Char, Char)] -> Enigma -> String
> breakEnigma crib = runMachine e (snd(unzip crib))
>  where e = Enigma {
>     rotors = (findRotorCombination' (menuToChar(findMenu crib))),
>     reflector = reflectorB, grundstellung = "AAA",
>     ringstellung = "AAA", plugboard = alphabet }
```

Exampleof breaking the encryption

If we input some cipher text and a suggested crib, we get some plain text returned.

```
> crib = zip "INCOMINGTRANSMISSIONNOTHINGTOREPORT"
    "YIHKDEZRMXNGPMUTYAMVYESKXGYFREEJ"
> breakEnigma crib
"NOTHINGTOREPORTWILLREPLYINANHOUR"
```

Chapter 5

Evaluation

This chapter will discuss the success of the project in terms of how well it meets the specification of breaking the Enigma encryption, as well as discussing how well the software works in terms of its correctness and efficiency. The reason for this discussion is to evaluate how well the software developed for this project meets the criteria for the first two objectives as well justifying the purpose of the third objective. Furthermore, the success of the third objective is reliant on its predecessors as software cannot be optimised if its initial purpose is not met.

5.1 Optimising the Solution

The first obvious distinction to make between the solution developed at Bletchley Park and the solution developed for this project, the latter of these is actively dependent on computing power, while the solution at Bletchley Park was reliant on several large teams of people computing these operations by hand. If we choose to analyse the logical flow of breaking the encryption rather than analysing the resources available at the time, we can see that this project has made some changes to greatly reduce Enigma processing time.

The first major difference between this project and Bletchley Park is the way the menu is generated and used. At Bletchley Park, an entire menu was generated, meaning that no matter the length of the message, a mapping for every letter needed to be found to ensure that a valid crib could be found and successfully used. This project on the other hand, makes use of Primm's MST Algorithm. While not directly used, the logic behind the proof allowed for a menu-growing algorithm to be used. The logic states that if we have a graph, there must be two disjoint non-empty subsets of vertices such that there must be an edge contained within both subsets (Moret & Shapiro, 1992). From this, we can deduce that if we simply check whether or not neighbouring vertices form a chain back to the initial value, then we simply just have to work out the longest chain. This means that not all vertices need to be visited, as only relevant vertices neighbouring already visited vertices need to be analysed.

Another major difference between the two solutions implemented for this project and the solution used at Bletchley Park is the use of Brute Force strategies at the stage of figuring out the rotor combination. At Bletchley Park, several teams in different Huts were responsible for figuring out the missing letter mappings not found in the menu. This was a long and tedious process which often wouldn't be finished by the time the German Military would change the encryption set up for the day. This project on the other hand takes the time to find every possible variation of encrypting the alphabet and compares

the menu to the encrypted alphabet, returning the best possible match and using that rotor combination to attempt to decrypt the cipher text.

5.2 Optimising the Software

5.2.1 Efficiency

In order to test the efficiency of the software, a new script called Main was made. This script will house the major functions which encrypt and decipher the Enigma machine. Its main function is as follows:

```
> main :: IO ()
> main = do
> let output = runMachine (map toUpper input)
> let broken = breakEnigma cribZip
> putStrLn "The Encrypted String is: "
> putStrLn output
> putStr "The Crib is: "
> putStrLn crib
> putStr "The deciphered text is: "
> putStrLn broken
```

Some of the first optimisations can be made when compiling the main file instead of running it via the GHC interpreter.

```
ghc -O -Wall Main.lhs
```

Running the compiler with the -O flags allows the compiler to use its built-in optimisations whilst still generating clean and concise code.

In order to see how the program performs, we can compile the program using the -rtsopts flag. Once the program has compiled, we can generate several details about the program including memory use and heap allocation. The main program generates the following information:

```
Main.exe 20 +RTS -sstderr
      5,494,880 bytes allocated in the heap
         79,240 bytes copied during GC
         76,112 bytes maximum residency (2 sample(s))
         26,704 bytes maximum slop
              2 MB total memory in use (0 MB lost due to fragmentation)
                                     Tot time (elapsed) Avg pause Max pause
  Gen  0         4 colls,    0 par   0.000s  0.000s    0.0000s   0.0001s
  Gen  1         2 colls,    0 par   0.000s  0.000s    0.0001s   0.0002s
  INIT   time    0.000s (  0.000s elapsed)
  MUT    time    0.000s (  0.040s elapsed)
  GC     time    0.000s (  0.000s elapsed)
  EXIT   time    0.000s (  0.000s elapsed)
  Total  time    0.000s (  0.040s elapsed)
  %GC    time      0.0%  (7.9% elapsed)
  Alloc rate   0 bytes per MUT second
  Productivity 100.0% of total user, 86.0% of total elapsed
```

From this data, one can see that the runtime (MUT) is approximately 0.04 seconds and the garbage collection (GC) is a near zero value. In order to optimise this further, time and allocation profiling can be used (done by adding the -P flag before RTS), which increases the productivity from 86% of the time elapsed to 96%.

In order to see how efficient the decrypter was with strings of varying length, a benchmarking tool was used. Criterion is a Haskell library containing tools to measure software performance. It proves to be one of the better benchmarking libraries due to its ability to cleanly handle set of infinite data and recursion-based programs (Hudak & Jones, 1994). The input lengths of messages were: 5, 11, 23, 25, 35, 52, 53. Using a growing length allows us to see how runtime scales with message length. Furthermore, using sporadic distances between the lengths reduces the amount of straight linear projections, as well as providing a more realistic view as messages sent in real life were not a set length. The results from the benchmarking are as follows:

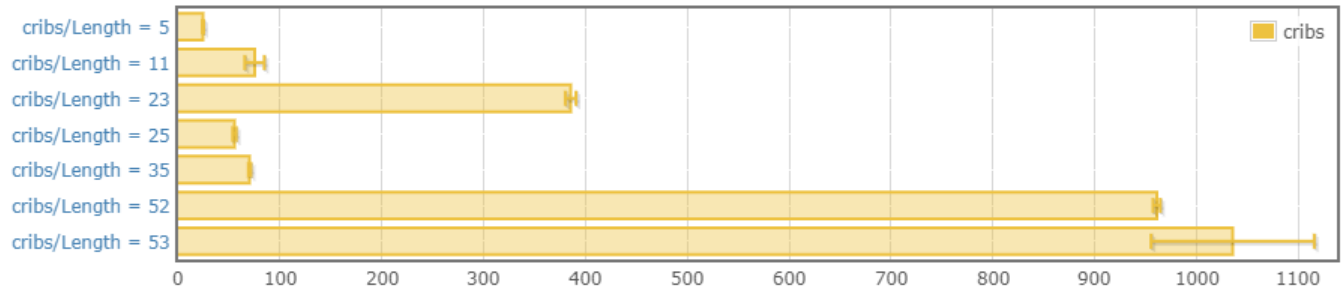

Figure 5.1: Benchmarking results for strings of varying length (ms)

From the above data, it is hard to find a general trend between crib length and time taken to bench mark. Apart from the anomalous result of length 23, one could suggest that smaller length cribs, i.e. less than 50 tend to have a shorter runtime, however there are lots of dependent variables to consider. For example, the hardware the tests are run on may have an effect on the benchmarking due to variables such as memory or heap size. Furthermore, the quality of the crib has to be considered. A higher quality crib could provide a larger menu, therefore allowing a more complete mapping from encryption to alphabet reducing runtime dramatically. It is also important to note that as cribs became longer in length (around 60 characters or more), the program would take several hours to run.

Upon further profiling of the programming, it became clear that the menu finding algorithm used became a detriment to larger length cribs as it took a greater amount of time to process such large mappings of text.

5.2.2 Correctness

As stated in 1.1, several of the objectives are reliant on the correctness of both the implementation of the Enigma machine, as well as providing a correct solution to cracking the encryption. In order to do this, the Haskell library Quickcheck was used. The issue with using tests to find correctness is that the tests can be written in such a way that they always return true, as tests can be subject to bias. This problem of truly determining whether a test is valid or not is solved by Quickcheck. The library itself makes use of an executable specification, meaning that the test value is checked again the value computed by Quickcheck (Claessen & Hughes, 2011).

The most important part of checking correctness was that each part built was being built on the principle that its predecessor was implemented correctly beforehand. This means that no gap in correctness could be made, and that at each stage of development correctness was preserved. When meeting the specification of implementing the Enigma machine, it was important to ensure that the machine was self-reciprocal and obeyed the rule of involution. In order to do this, most of the tests implemented checked that the identity of whatever was input was preserved. The following proposition tests this:

```
> prop_enigmaEncryption = encryption (enigmaMachine { ringstellung = "BBB" })
    "AAAAA" == "EWTYX"
> prop_enigmaDecryption = encryption(enigmaMachine{ringstellung="BBB"})
    "EWTYX" == "AAAAA"
```

As seen in the above code, the first proposition checks that the phrase "AAAAA" becomes encrypted into "EWTYX", while the second proposition ensures that the phrase "EWTYX" becomes encrypted into "AAAAA". From this, we can see that the machine simulated is self-reciprocal and therefore is a correct implementation.

In terms of breaking the machine, several tests were set to ensure that the text that was used as input gets encrypted and decrypted and then checking whether or not the input is the same as the output. This level of correctness is trivial due to the fact that if the output is different to the input then correctness has not been preserved.

5.3 Comparison With Related Work

In order to best assess the success of this project, it is important to compare the solution developed for this project with already existing solutions as discussed in 2.2.

5.3.1 M4 Project

The M4 project made use of an extensive network of distributed systems that allowed Enigma cracking to work in the background of a user's PC, and then would feedback to a server if any progress was made. The major difference between this project and the M4 project is that the M4 project, like the Bletchley Park solution, made use of stops to eliminate incorrect rotor positions. The runtime of the program would be increased relative to the size of the loop. The larger the loop, the longer the time needed to check the loop. This would be due to the fact that each rotor combination would need to be retested and with such a large array of different combinations it would take polynomial time. This project on the other hand makes use of a best-effort solution. There is guaranteed to be at most and at least 1 specific set of rotors providing a specific mapping from some plain-text to some cipher-text, a more brute force solution needed to be used. This was done as GHC provides a strong basis of code optimisations therefore minimising the runtime whenever possible.

Another major difference is that the M4 project is implemented in C while this project is implemented in Haskell. For longer length cribs, it may be more beneficial to implement in C, as you can ensure that the correct amount of memory can be allocated which would reduce the chance of the program freezing as this project does during benchmarking.

5.3.2 Enigma@Home

Similar to the M4 Project, Enigma@Home makes use of distributed computing, but instead of 'piggybacking' off people's personal computers, it made use of the University of Berkeley's BOINC system. Upon further analysis, the success of Enigma@Home surpasses that of the M4 Project. This is mainly due to the greater processing power provided by the BOINC system.

Comparing with this project however, we do get some interesting differences. Enigma@Home makes use of a combination of a brute force technique and a hill climbing algorithm. The project uses these techniques in order to minimise the size of the search space whenever possible by simply rotating the outer rotors while leaving the middle rotor stationary (Ulbricht, 2005). In order to perform the hill climbing required, the algorithm devised uses the alphabet as a look up table where some statistical analysis is used to score the likely hook of one letter mapping to another (Ostwald & Weierud, 2017). This method does tend to operate slower overall, but does provide a more accurate method to cracking Enigma. This differs greatly to the solution provided by this project which uses a best effort model. This does work at a much faster speed but does tend to struggle when messages are longer than 60 characters.

Chapter 6

Summary & Reflections

6.1 Project Management

Throughout the duration of this project, it became clear that a strict and well-planned schedule was needed to ensure that all three of the objectives set in 1.1 were met . Although the schedule felt tighter than initially planned at times, the resulting software developed successfully met all the aims of the objectives and allowed for a deeper analysis of the software than initially planned.

6.1.1 Self-Management

In order to successfully meet the deadlines set a well thought-out and sturdy plan was needed. In order to achieve this, a balance between external commitments, University work and the work for this project needed to be made. This was probably the most difficult part of the project, as there were certain times where a shift in priorities took place between work for this project and other work.

Self-motivation was also another challenge presented by this project. Though its solution is often trivial in its nature, the loss of it tends to happen quicker than expected. During the sprints throughout the winter term, it became clear that a clear lapse in judgement had occurred when attempting to meet goals without properly working out the time needed to do so. This loss of motivation paired with looming coursework deadlines and the approach of examinations led to prioritisation of over work, putting this project on hold until the culmination of examinations. This was finally resolved early into the spring semester as the program became more modular and step-based meaning that each section could be broken down into manageable tasks which could be completed in shorter time frames.

A visit to Bletchley Park during the winter break paired with a smaller work load during the spring semester fuelled my motivation to 'knuckle down' and finish this project. Having fortnightly meetings with my supervisor ensures that my progress was on the right track both from a development standpoint as well as a written one. Furthermore, these meetings allowed small demonstrations of progress of development such that my supervisor was constantly aware of how my project was progressing.

6.1.2 Time-Management

In addition to keeping myself motivated to finish the project, one had to be consciously aware of specific deadlines set. This itself led to a great development of my software engineering skills, especially in planning and maintenance.

By using a Gantt chart, I was able to set clear and well-communicated sprints where at each stage a small part of the project was being completed. Upon completion of development and testing the software, a clear evaluation of how well sprints were met and managed can be made. When the project was initially proposed in October 2018, the Gantt chart provided an optimistic view of the project with the design, development and testing of the Enigma machine to be finished before the winter break; an under-estimation of the workload was clearly made as some of the work had to be pushed into the spring semester. In order to fully evaluate the time management of the project, a discussion of their success can be found below.

Sprint	**Description**	**Date Projected**	**Date Completed**
A	Complete Project Proposal	26/9/2018	8/10/2018
B	Research Enigma machines	15/10/2018	18/10/2018
C	Implement Enigma machine and Test	22/10/2018	29/11/2018
D	Research various methods of cracking Enigma	29/10/2018	14/2/2019
E	Implement a selected solution	5/11/2018	18/3/2019
F	Test the solution	12/11/2018	22/3/2019
G	Write Interim Report	19/11/2018	7/12/2018
H	Analyse initial solution and design improvements	21/1/2019	3/3/2019
I	Implement solution improvements	4/2/2019	24/3/2019
J	Test solution and perform software analysis	11/2/2019	27/3/2019
K	Begin Dissertation	21/1/2019	12/4/2019
L	Write and film video	1/3/2019	12/4/2019
M	Prepare for presentation	12/4/2019	16/5/2019

Table 6.1: The Project's Sprints.

As seen in the above table, and further demonstrated in the projects Gantt charts (see Appendix A), a large time gap grew between sprint C and D. At the time of initial planning, an underestimation was made regarding the amount of time which could be spent on other University work and extra-curricular activities. This meant that prioritisation could be made such that the right work could be completed as necessary. The initial project plan proposed was too optimistic and a decision was made that the first three sprints was an appropriate amount of work to balance between other commitments during the winter term.

The changes made to the project plan set in the interim report provide a clearer and more precise breakdown of what needed to accomplished rather than a more open and vague goal set in the initial proposal. Having clearer milestones, along with a larger amount of time to research viable solutions to breaking Enigma ensured that a concrete method of breaking Enigma could be designed and implemented. Furthermore instead of having two stages as defined in the sprints for implementing solutions, it was decided it would be easier to design both the existing solution and combine it with the designed solution such that a hybridisation of the two would be implemented, providing a clear and working solution to breaking the encryption.

6.1.3 Reflections & Contributions

Throughout the course of the project there have been what can only be described as a series of peaks and troughs. This mixed with the constant need to discover and learn new ways to both solve an almost 80-year-old problem, as well as solve the problems I encountered when making this project. The learning experience provided by this project in both technical and softer skills has been invaluable. The combination of two areas of information I am enthused by really drove my passion to delve deep into each area to ensure that I was maximising my progress and furthering my achievement.

Taking an overall view of the project, its three objectives provided clear milestones to reach, this paired with the project's extensive time and resource planning, ensuring that every step of the project was clearly defined before it was achieved. This allowed for no discrepancy on whether or not the project was a success or not, as once all sprints were completed, the project was complete.

One achievement that perhaps solidifies the success of this project is the nature of breaking Enigma and how its implementation was devised. This best effort method of encrypting the alphabet with every possible rotor combination and comparing the menu generated from the crib and cipher text reduces the need to derive a more complex algorithm and perhaps propagates the benefit of using a functional language like Haskell.

6.1.4 Conclusion

Overall, one could say that this project is successful, as all three of the objectives have been met. Additionally, a deeper analysis of the degree of which each objectives success has been analysed. Although there are some issues surrounding the deciphering of longer phrases, the aim of the project has been met with the ability to both encrypt and decipher Enigma text.

If more time were given, this project could be expanded further. The issue surrounding string length could be resolved, meaning that longer string phrases could be decrypted. Another future development could allow for parallelism to be implemented such that, Enigma cipher text could be decrypted concurrently, which would allow for a greater processing speed and a shorter runtime.

Another future development could be to implement a similar solution in a lower-level language like C or C++, such that memory allocation could be directly be associated with word length. Furthermore, some benchmarking comparison could be made to assess the benefit of using one programming paradigm over the other. Additionally, perhaps attempting to use a dictionary-based attack could provide an interesting comparison between different decrypting methods.

Overall, this project has been successful in meeting its set requirements and does prove that modern computing advances are of great benefit to the world of classical cryptography.

References

Almgren, F., Andersson, G., Granlund, T., Ivansson, L., & Ulfberg, S. (2000). How We Cracked The Code Book Ciphers. *Unpublished manuscript, available at the web site http://answers. codebook. org*.

Alster, K., Urbanowicz, J., & Williams, H. C. (2011). *Public-key Cryptography and Computational Number Theory: Proceedings of the International Conference Organized by the Stefan Banach International Mathematical Center Warsaw, Poland, September 11-15, 2000*. Walter de Gruyter.

Beesly, P. (2000). *Very Special Intelligence. The Story of the Admiralty's Operational Intelligence Centre 1939–45*. Reissued edition (orig. 1977). London: Greenhill Books.

Bloch, G., & Deavours, C. A. (1987). Enigma Before ULTRA Polish Work and the French Contribution. *Cryptologia*, *11*(3), 142–155.

Campbell, M. (2011). Uncrackable Codes: The Second World War's Last Enigma. *New Scientist*, *210*(2813), 44.

Claessen, K., & Hughes, J. (2011). Quickcheck: a lightweight Tool for Random Testing of Haskell Programs. *ACM Sigplan Notices*, *46*(4), 53–64.

The Cryptanalysis of a ThreeRotor Machine Using a Genetic Algorithm., author=Bagnall, Anthony J and McKeown, Geoff P and Rayward-Smith, Victor J, booktitle=ICGA, pages=712–718, year=1997. (n.d.).

Deavours, C. A., & Reeds, J. (1977). The Enigma Part I Historical Perspectives. *Cryptologia*, *1*(4), 381–391.

Final M4 Letter, 1995. (n.d.). `https://enigma.hoerenberg.com/index.php?cat=M4%20Project%202006&page=Publication`. (Accessed: 2019-03-22)

Gaj, K., & Orłowski, A. (2003). Facts and Myths of Enigma: Breaking Stereotypes. In *International conference on the theory and applications of cryptographic techniques* (pp. 106–122).

Good, I. J. (2000). Turing's Anticipation of Empirical Bayes in Connection with the Cryptanalysis of the Naval Enigma. *Journal of Statistical Computation and Simulation*, *66*(2), 101–111.

Hamer, D. H. (1997). Enigma: Actions Involved in the 'Double Stepping' of the Middle Rotor. *Cryptologia*, *21*(1), 47–50.

Hamer, D. H., Sullivan, G., & Weierud, F. (1993). Enigma Variations: An Extended Family of Machines. In *Proceedings-a* (Vol. 140).

Hodges, A. (2012). *Alan Turing: The Enigma*. Random House.

Hudak, P., & Jones, M. P. (1994). Haskell vs. Ada vs. C++ vs. AWK vs.... an Experiment in Software Prototyping Productivity. *Contract*, *14*(92-C), 0153.

Kruh, L., & Deavours, C. (2002). The Commercial Enigma: Beginnings of Machine Cryptography. *Cryptologia*, *26*(1), 1–16.

Mackenzie, A. (1996). Undecidability: The History and Time of the Universal Turing Machine. *Configurations*, *4*(3), 359–379.

Marks, P., & Weierud, F. (2000). Recovering the Wiring of Enigma's Umkehrwalze A. *Cryptologia*, *24*(1), 55–66.

Martin, R. C. (2002). *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall.

Miller, A. R. (1995). The Cryptographic Mathematics of Enigma. *Cryptologia*, *19*(1), 65–80.

Moret, B. M., & Shapiro, H. D. (1992). An Empirical Assessment of Algorithms for Constructing a Minimum Spanning Tree. *Computational Support for Discrete Mathematics*, *15*, 99–117.

Ostwald, O., & Weierud, F. (2017). Modern Breaking of Enigma Ciphertexts. *Cryptologia*, *41*(5), 395–421.

Polish Mathematicians Finding Patterns in Enigma Messages, author=Christensen, Chris, journal=Mathematics Magazine, volume=80, number=4, pages=247–273, year=2007, publisher=Taylor & Francis. (n.d.).

Ratcliff, R. A. (2006). *Delusions of Intelligence: Enigma, Ultra, and the End of Secure Ciphers*. Cambridge University Press.

Rejewski, M. (1980). An Application of the Theory of Permutations in Breaking the Enigma Cipher. *Applicationes mathematicae*, *4*(16), 543–559.

Rejewski, M. (1981). How Polish Mathematicians Broke The Enigma Cipher. *IEEE Annals of the History of Computing*(3), 213–234.

Stengers, J. (1984). Enigma, the French, the Poles and the British, 1931–1940. In *The missing dimension* (pp. 126–137). Springer.

Turing, A. M. (1939). Turing's Treatise on Enigma. *Unpublished Manuscript*.

Ulbricht, H. (2005). *Die Chiffriermaschine Enigma - Trügerische Sicherheit : Ein Beitrag zur Geschichte der Nachrichtendienste* (Doctoral dissertation). Retrieved from `https://publikationsserver.tu-braunschweig.de/receive/dbbs_mods_00001705`

Welchman, G. (1982). *The Hut Six Story: Breaking the Enigma Codes*. McGraw-Hill New York.

Appendices

Appendix A

Other Figures

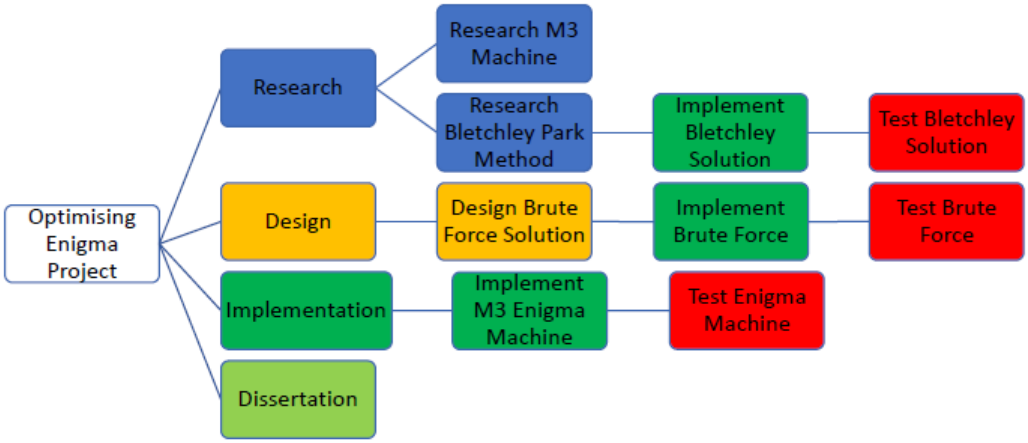

Figure A.1: The initial structure of the project

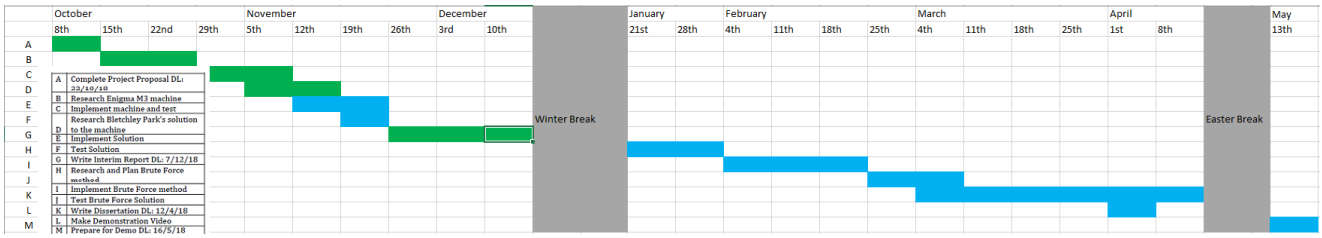

Figure A.2: The initial Gantt chart

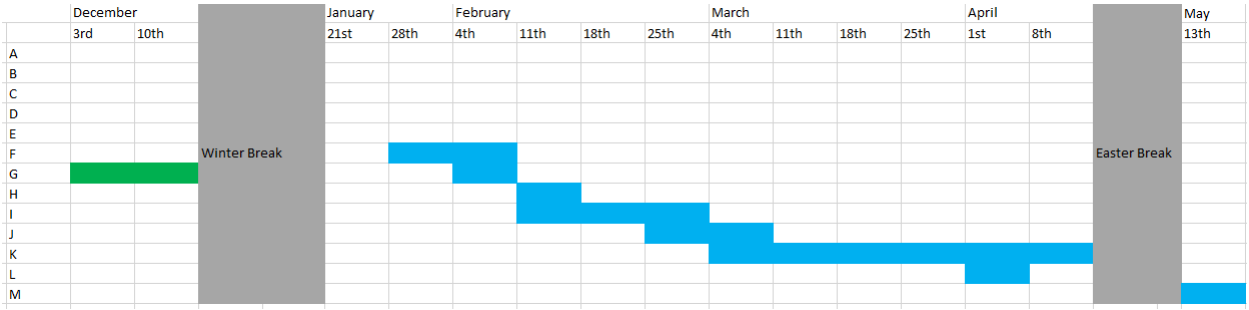

Figure A.3: The updated Gantt chart

Appendix B

Code

Enigma.lhs

```
> module Enigma where

> import Data.Bool
> import Data.Char
> import Data.List
> import Data.Maybe

The reverse of each function has been implemented to account for testing the
    encryption

> alphabet :: String
> alphabet = ['A' .. 'Z']

Two functions are then defined to handle the substitution of letter when
    passed throughout
the various rotors in the machine.
c is the c is the char for which substitution is being handled
s is the substitution of which the permutation of the alphabet is being
    represented

> substitute :: String -> Char -> Char
> substitute s c = fromMaybe c $ lookup c $ zip alphabet s

> unsubstitute :: String -> Char -> Char
> unsubstitute s c = fromMaybe c $ lookup c $ zip s alphabet

A ceasar shift can also be implemented to account for the mapping of the
    alphabet to whatever
order the rotor sets the alphabet to be
The parameter k is used to represent the key of the shift

> shift :: Char -> Char -> Char
> shift k = substitute $ dropWhile (/= k) $ cycle alphabet

> unshift :: Char -> Char -> Char
> unshift k = unsubstitute $ dropWhile (/= k) $ cycle alphabet
```

```
Now we can begin to define the enigma machine:
The Enigma machine has 3 main components, the rotors, the reflectors and the
    plugboard. All of
which are some form of shifting letters from one to another. The Grundstellung
    is the starting
position of the rotors, this was chosen by the operator and was different for
    every message which
is one of the reasons the encryption is so strong. The ringstellung is the
    inital ring setting
relative to the rotor discs, this setting is effectively an initialisation
    vector for the
encryption adding a level of randomness to the machines encryption. The
    plugboard is a variable
wiring system that would be manually reconfigured by the operator using patch
    cables. You could
manually map letters together forming a 'steckered pair' of letters which
    swapped the letter
respectively both before and after the rotor scrambling.

> type SteckeredPair = [(Char,Char)]

> data Enigma = Enigma {
>  rotors :: [(String, String)], reflector :: String,
>  grundstellung :: String, ringstellung :: String,
>  plugboard :: String }
>   | Bombe { rotors :: [Rotor], reflector :: String,
>  grundstellung :: String, ringstellung :: String,
>  plugboard :: String } deriving (Eq, Show)

Rotors:
The M3 Army Enigma Machine was adopted by the German army in 1930 and later
    the Navy in 1934.
Origionally the machine had 3 rotors but a later 2 were added in 1938, giving
    the operator a
choice of 3 out of 5. In 1939 the Navy added two more rotors: these are
    defined as strings below.
A 4 rotor Enigma was implemented in 1942 so rotors have been added to account
    for that.

> type Rotor = (String, String)

> rotorI :: Rotor
> rotorI  = ("EKMFLGDQVZNTOWYHXUSPAIBRCJ", "Q")
> rotorII :: Rotor
> rotorII = ("AJDKSIRUXBLHWTMCQGZNPYFVOE", "E")
> rotorIII :: Rotor
> rotorIII = ("BDFHJLCPRTXVZNYEIWGAKMUSQO", "V")
> rotorIV :: Rotor
> rotorIV = ("ESOVPZJAYQUIRHXLNFTGKDCMWB", "J")
> rotorV :: Rotor
```

```
> rotorV  = ("VZBRGITYUPSDNHLXAWMJQOFECK", "Z")
> rotorVI :: Rotor
> rotorVI = ("JPGVOUMFYQBENHZRDKASXLICTW", "M")
> rotorVII :: Rotor
> rotorVII = ("NZJHGRCXMYSWBOUFAIVLPEKQDT", "Z")
> rotorVIII :: Rotor
> rotorVIII = ("FKQHTLXOCBJSPDZRAMEWNIUYGV", "M")
```

The letters at the end of the rotor definintions are the turnover notches. These were point
if a specific shift in letter occur, it would rotor the next rotor as well as the one being
used.
Reflectors:
A reflector (or reversal rotor) is where the outputs of the final rotor were connected in
pairs and then redirected back through the rotors via a different route. This ensured that no
letter could be mapped back to itself and that encryption was the same as decryption - a flaw
which made cracking the machine possible.

```
> type Reflector = String
> reflectorA :: Reflector
> reflectorA = "EJMZALYXVBWFCRQUONTSPIKHGD"
> reflectorB :: Reflector
> reflectorB = "YRUHQSLDPXNGOKMIEBFZCWVJAT"
> reflectorC :: Reflector
> reflectorC = "FVPJIAOYEDRZXWGCTKUQSBNMHL"
```

Now we can define the Enigma machine we wish to use:

```
> enigmaMachine :: Enigma
> enigmaMachine = Enigma {
>     rotors = [rotorI, rotorII, rotorIII],
>     reflector = reflectorB,
>     grundstellung = "AAA",
>     ringstellung = "AAA",
>     plugboard = alphabet }
```

The rotation function handles the rotation of a rotor. This function performs a case analysis
of the bool to check what the starting value of each rotor then checking if that is an element
in next rotor.
sR refers to the starting value in that rotor. nR refers to the next value in the rotor

```
> rotation :: Enigma -> Enigma
> rotation r = r {
>     grundstellung = [bool sR1 (shift 'B' sR1) $ sR2 `elem` nR2,
```

```
>     bool sR2 (shift 'B' sR2) $ sR2 `elem` nR2 || sR3 `elem` nR3, shift 'B'
    sR3
>    ]}
>     where
>      [sR1, sR2, sR3] = grundstellung r
>      [nR1, nR2, nR3] = snd <$> rotors r
```

Next we apply the shift by conjugating with the alphabet

```
> applyShift :: String -> Char -> String
> applyShift cs key = unshift key . substitute cs . shift key <$> alphabet
```

Then we can apply the rotation to the shifted input

```
> applyRotation :: Enigma -> [String]
> applyRotation cs = zipWith applyShift (fst <$> rotors cs) $ zipWith unshift
    (ringstellung cs) $ grundstellung cs
```

We can then follow the process of the enigma machine e through the shifting, rotation, reflecting and the finally the plugboard mapping.

```
> findMap :: Enigma -> Char -> Char
> findMap e = plugboardOutput . unsubstitute rotatedInput . substitute
    (reflector e)
>    .  substitute rotatedInput . plugboardOutput
>    where
>      rotatedInput = foldr1 (.) (substitute <$> applyRotation e) <$> alphabet
>      plugboardOutput = substitute $ plugboard e
```

Finally we create a function that take a letter, return it encrypted and increment the machine as necessary.

```
> encryptChar :: Enigma -> Char -> (Enigma, Char)
> encryptChar machine c = bool(machine, c)(machine', findMap machine' c) $
    isLetter c
>    where machine' = rotation machine

> encryption :: Traversable t => Enigma -> t Char -> t Char
> encryption machine = snd . mapAccumL encryptChar machine
```

The run machine function is used to traverse a string of chars, apply encryption to each one via the engima machine

```
> runMachine :: Traversable t => t Char -> t Char
> runMachine cs = encryption (enigmaMachine) cs
```

Menu.lhs

```
> module Menu where

> import Crib
> import Data.List
> import Data.Char
> import Data.Function
> import Data.Ord
> import Data.Graph

---------------
Menu Generation
---------------
A menu is a graph showing all the connections between the letters in both the
    crib and the encrypted Word

> tuplesToList :: [(a,a)] -> [[a]]
> tuplesToList = map (\(x,y) -> [x,y])

> listToTuple :: [a] -> (a,a)
> listToTuple [x,y] = (x,y)

> menuToTuple :: [String] -> [(Char, Char)]
> menuToTuple [] = []
> menuToTuple (x:xs) = listToTuple x : menuToTuple xs

groupByVertex groups each pair into a list of each vertex and each letter that
    is linked to that vertex

> groupByVertex :: (Eq a, Ord a) => [(a, b)] -> [(a, [b])]
> groupByVertex = map (\l -> (fst . head $ l, map snd l)) . groupBy ((==) `on`
    fst) . sortBy (comparing fst)

-------------------------
----Menu Generation-----
-------------------------

A menu is a type coined by Alan Turing used to describe a relationship between
    letters in the crib and the cipher text. In this case
these letters are converted to integers to allow for easier mathematical
    computation.

> type Menu = [Int]

findMenu will take a tuple of the crib and the encrypted message and return a
    list of menus

> findMenu :: [(Char, Char)] -> [Menu]
> findMenu crib = growMenu (findLink crib)

No duplicates will check to see if there is a duplicate element in the list
```

```
    and return a boolean
if one exists therefore proving a cycle exists in the menu.

> boolDups :: Eq a => [a] -> Bool
> boolDups [] = True
> boolDups [_] = True
> boolDups (h : t)
>  |(elem h t) = False
>  |otherwise = boolDups t

findLink finds a list of menus based upon a crib and cipher text. It begins by
    splitting the tuple
into segments and creating sublists of based upon the relationship between the
    elements within

> findLink :: [(Char, Char)] -> [Menu]
> findLink crib = [[xs, ys] |(ys, y) <- m,(xs,x) <- c, y ==x]
>  where
>   cycle = [0 .. ((length crib) - 1)]
>   (ms, cs) = unzip crib
>   m = zip cycle ms
>   c = zip cycle cs

 growMenu recursively expands the menu by linking all menus with common
     elements together until no more
 joins can be made. Initlally the null case will return the list of menus if
     no more joins can be done.
 Otherwise the function recurses to do more joins where through list
     comprehension menus are found with
 overlaps. Furthermore all duplicate menus are removed as well as circular
     menus and submenus.

> growMenu :: [Menu] -> [Menu]
> growMenu ms
>  |null joins = ms
>  |otherwise = growMenu noDupsMs
>  where
>   joins = [(joinMenu x y) | x <-ms, y <- ms, x /= y, (findOverlaps x y) > 0]
>   filteredJoins = [m | m <-joins, boolDups m]
>   filteredMenu = removeSubMenus filteredJoins
>   noDupsMs = nub filteredMenu

findOverlaps will take two lists and return any overlapping elements. This
    lists indexed and the
function returns the number of shared elements. The lists are labelled left
    and right such that when
joined later they meet in the correct place.

> findOverlaps :: [Int] -> [Int] -> Int
> findOverlaps xs ys = findOverlaps' xs ys 0
```

```
> findOverlaps' :: [Int] -> [Int] -> Int-> Int
> findOverlaps' xs ys n
>  | n == length xs = 0
>  |left == right = (length left)
>  |otherwise = findOverlaps' xs ys (n+1)
>  where
>   left = (drop n xs)
>   right = (take (length left) ys)

joinMenu effectively concatenates two lists will removing any overlapping
    elements.

> joinMenu :: [Int] -> [Int] -> [Int]
> joinMenu ms ns = (take ((length ms) - n) ms) ++ ns
>  where
>   n = findOverlaps ms ns

> removeSubMenus :: [[Int]] -> [[Int]]
> removeSubMenus ms = [m | m <-ms, null [ns | ns <- ms, (length ns) > (length
    m), m == (take (length m) ns)]]

> menuToChar :: [Menu] -> [String]
> menuToChar menu = [map chr (map (+65) m) |m <- menu]
```

Crib.lhs

```
> module Crib where

> import Enigma
> import Data.List as List
> import Data.Ord
> import Data.Tuple
> import Data.Function

------------------------
Cracking the machine:
------------------------
The first step in attempting to crack the enigma machine is to find a crib. A
    crib is plaintext known to be in encrypted
string. alignCrib is responsible for comparing the crib and the encrypted
    string in order to see if any character is in the
index in both strings

> alignCrib :: String -> String -> String
> alignCrib [] [] = []
> alignCrib crib encryptedOutput = List.map fst . List.filter (uncurry (==)) $
    zip crib encryptedOutput

filterAlignment returns a bool depending on whether or not cribs can be
    aligned without error. If there is an error then the crib
```

```
and the encrypted string need to be realigned such that no clashes occur.

> filterAlignment :: String -> String -> Bool
> filterAlignment crib encryptedOutput = if alignCrib crib encryptedOutput ==
    [] then True else False

shiftCrib adds a space to the beginning of the crib such that it can be tested
    as a new crib for alignment.

> shiftCrib :: String -> String
> shiftCrib crib = " " ++ crib

findNoCrashes is used to return the best shifted crib with the encrypted
    string. A crash is used to refer to positions in which
a letter in the crib matches the letter at the same index in the encrypted
    string. This is because the enigma machine cannot map
letters to itself. Once this returns a valid list of pairs.

For example is we have a string KEINEBESONDERENEREIGNISSE and an encrypted
    string "UAENFVRLBZPWMEPMIHFSRJXFMJKWRAXQEZ" we would get
returned [(' ','U'),(' ','A'),(' ','E'),(' ','N'),('
    ','F'),('K','V'),('E','R'),('I','L'),('N','B'),('E','Z'),('B','P'),
('E','W'),('S','M'),('O','E'),('N','P'),('D','M'),('E','I'),('R','H'),('E','F'),('N','S'),('
('I','F'),('G','M'),('N','J'),('I','K'),('S','W'),('S','R'),('E','A')] as a
    list of pair. Note the offset of 5 letters to account
for the N's and E's matching in the original postions of the crib.

> findNoCrashes :: String -> String -> [(Char, Char)]
> findNoCrashes crib encrypted = if filterAlignment crib encrypted then (zip
    crib encrypted) else findNoCrashes (shiftCrib crib) encrypted

Assuing that we have a valid crib and now have calculated the offset, we can
    then filter out all letters which are encrypted more than
once. For example using the above example:
       Word :
          U|A|E|N|F|V|R|L|B|Z|P|W|M|E|P|M|I|H|F|S|R|J|X|F|M|J|K|W|R|A|X|Q|E|Z|
       Crib : | | | | | K|E|I|N|E|B|E|S|O|N|D|E|R|E|N|E|R|E|I|G|N|I|S|S|E|

----------------
Finding Loops
----------------

A loop is a term used to refer to a closed cycle of letter pairings. Using the
    above example we can see that B is encrypted to P,
N is also encrypted to P and N is encrypted to B this relationship can be used
    to take advantage of the machine.

setUpForLoop removes any pairs containing spaces such that the list of tuples
    has already handled the offset

> setUpForLoop :: [(Char, Char)] -> [(Char, Char)]
```

```
> setUpForLoop alignedText = [c | c <- alignedText, fst c /= ' ']

orderTuples orders the list of tuples in descending order according to the
    function passed into orderTuples, this is typically
fst or snd used for taking the first or second element of a tuple..

> orderTuples :: String -> String -> ((Char, Char) -> Char) -> [(Char, Char)]
> orderTuples crib encrypted f = sortBy (comparing f) (setUpForLoop
    (findNoCrashes crib encrypted))

groupTuples splits the list of ordered tuples into lists depending on the
    inputted function to groupTuples.

> groupTuples :: String -> String -> ((Char, Char) -> Char) -> [[(Char, Char)]]
> groupTuples crib encrypted f = groupBy ((==) `on` f) ((orderTuples crib
    encrypted f))

invert inverts the order of the elements in a tuples

> invert :: [(a, b)] -> [(b, a)]
> invert = List.map swap

Flatten flattens a list such that all elements are on the same depth.

> flatten :: [[a]] -> [a]
> flatten xs = (\z n -> List.foldr (\x y -> List.foldr z y x) n xs) (:) []

joinLists combines the list of all pairs of letters ordered by both first and
    second element in each pair.

> joinLists :: String -> String -> [[(Char, Char)]]
> joinLists crib encrypted = (groupTuples crib encrypted fst) ++ (groupTuples
    crib encrypted snd)


groupTuples' groups both grouped lists from groupTuples and groupTuplesSnd,
    flattens the list such that all elements are on the same level and
then removes any duplicate values in the list

> groupTuples' :: String -> String -> [(Char, Char)]
> groupTuples' crib encrypted = nub (flatten (joinLists crib encrypted))
```

Bombe.lhs

```
> module Bombe where

> import Enigma
> import Menu
> import Data.Maybe
> import Data.List
> import Data.Ord

A list of all the rotors which could be used in the machine set up

> rotorList :: [Rotor]
> rotorList = [rotorI, rotorII, rotorIII, rotorIV, rotorV, rotorVI, rotorVII,
    rotorVIII]

> position :: [Int]
> position = [1 .. 26]

A Bombe is similar to an Enigma machine expect that the rotor list is left
    empty - this is due to the fact the following code
will be working out the rotors based upon the encrypted text.

> bombe :: Enigma
> bombe = Enigma {
>     rotors = [],
>     reflector = reflectorB,
>     grundstellung = [],
>     ringstellung = [],
>     plugboard = alphabet}

The groupInNs takes a list of elements and an integer n and returns all
    possible combinations of the elements in the list
in sublists of size n.

> groupInNs :: Eq a => [a] -> Int -> [[a]]
> groupInNs xs n = filter ((n==) . length . nub) $ mapM (const xs) [1..n]

Fetch rotor combination fetches a set of rotors at a specifies index

> fetchRotorCombination :: Eq a => [a] -> Int -> [a]
> fetchRotorCombination rs n = (groupInNs rs 3) !! n

runBombe will take in a list, in this case the alphabet and return all
    possible encryption variations possible

> runBombe :: Traversable t => t Char -> Int -> t Char
> runBombe cs n = encryption (e) cs
>   where e = Enigma {
>     rotors = (fetchRotorCombination rotorList n),
>     reflector = reflectorB, grundstellung = "AAA",
>     ringstellung = "AAA", plugboard = alphabet }
```

```
The rotor length is used as a limiter in recursive functions to ensure no out
    of bounds errors can occur

> rotorLength :: Int
> rotorLength = length (groupInNs rotorList 3)

Limit bombe is used to handle any missing return value errors

> limitBombe :: Int -> Maybe Int
> limitBombe n = if n < rotorLength then Just n else Nothing

runBombe' uses list comprehension to iterate through the list of rotors such
    that every rotor combination encrypts the alphabet
once

> runBombe' :: Traversable t => t Char -> [t Char]
> runBombe' cs = [runBombe cs n | n <- [0 .. (rotorLength -1)]]

prepBreak makes use of a predicate as a placeholder for the alphabet when the
    function is called in its prime form below.
It takes an integer and a list of characters (i.e. the alphabet) and calls the
    encrypted version found in the encrypted
alphabets made in runBombe. This is then zipped with the alphabet to create a
    list of tuples

> prepBreak :: p -> Int -> [(Char, Char)]
> prepBreak xs n = zip alphabet $ runBombe alphabet n

prepBreak' is a traversable version of the above function which traverses
    through the list of rotors in order to zip each
encrypted alphabet with the standard alphabet in order to create a list of
    tupled letter pairs where one of which in encrypted.

> prepBreak' :: Traversable t => t Char -> [[(Char, Char)]]
> prepBreak' xs = [prepBreak x n| x <- runBombe' xs, n <- [0 .. (rotorLength
    -1)]]

cribUp is used to take the menu and split it into a list of tuples such that
    every letter in the menu can be mapped to any other
letter as long as they are not equal

> cribUp :: Eq a => [a] -> [(a,a)]
> cribUp menu = [(x, y) | x <- menu, y <- menu , x /= y]

findClosestMatch is used to find the closest encrypted alphabet to the
    mappings found in the menu.

> findClosestMatch :: [String] -> Int -> [(Char, Char)]
> findClosestMatch menu n = intersect (cribUp (head menu)) (prepBreak alphabet
    n)
```

```
findClosestMatch' is used to iterate through the list of encrypted alphabets
    and find the best match based upon the menu
inputted

> findClosestMatch' :: [String] -> [[(Char, Char)]]
> findClosestMatch' menu = [findClosestMatch menu n | n <- [0 .. (rotorLength
    -1)]]

sortMatches iis used to sort the matches by length

> sortMatches :: [[a]] -> [[a]]
> sortMatches = sortBy (comparing length)

fetchClosestMatch uses the list of matches to find the largest match between
    the menu and the encrypted alphabet. This means
that the largest majority of letter mappings can be found.

> fetchClosestMatch :: [String] -> [(Char, Char)]
> fetchClosestMatch menu = head(reverse(sortMatches(findClosestMatch' menu)))

findRotorCombination uses list comprehension in order to find if the closest
    matches in each menu can be found in a set
encrypted alphabet. A maybe wrapper is used too handle any null elements.

> findRotorCombination :: [String] -> [Maybe Int]
> findRotorCombination menu = [elemIndex x y | x <- (fetchClosestMatch menu),
    y <- (prepBreak' alphabet)]

filterRotors is used to remove all rotor combinations that return the nothing
    type meaning that only valid letters indexs are
found

> filterRotors :: [String] -> [Maybe Int]
> filterRotors menu = filter (/= Nothing) (findRotorCombination menu)

findRotorCombination is used to find which specific set of rotors is used to
    decrypt the cipher text used to form the menu
back to its original plaintext.

> findRotorCombination' :: [String] -> [Rotor]
> findRotorCombination' menu = fetchRotorCombination rotorList ((fromJust(head
    (filterRotors menu))) - 1)
```

TestEnigma.lhs

```
> module TestEnigma where

> import Enigma
> import Bombe
> import Data.List
> import Data.Char
> import Test.QuickCheck

Testing substitution

> permutation key = nub $ filter isUpper key ++ alphabet

> prop_checkSubsitution key = (unsubstitute ( permutation key) . substitute
    (permutation key) <$> alphabet) == alphabet

Testing reflectors:

> prop_reflector = and $ zipWith (/=) alphabet $ map (substitute reflectorB)
    alphabet
> prop_reflectorCheckSelfInverse = iterate (map (substitute reflectorB))
    alphabet !!2 == alphabet

Testing Shift of mapping

> prop_mapping = findMap enigmaMachine 'A' == 'U'

Testing encryption:

> prop_enigmaEncryption = encryption (enigmaMachine { ringstellung = "BBB" })
    "AAAAA" == "EWTYX"

Testing decryption:

> prop_enigmaDecryption = encryption(enigmaMachine{ringstellung="BBB"})
    "EWTYX" == "AAAAA"

Testing Deciphering

> prop_enigmaDecipher = breakEnigma(crib3) ==
    "NOTHINGTOREPORTWILLREPLYINANHOUR"

Testing encryption and deciphering

breakEnigma is used to break the encryption. This is done by making an Enigma
    machine with the rotors found using the above
function.

> breakEnigma :: [(Char, Char)] -> String
> breakEnigma crib = runMachine (snd(unzip crib))
```

```
> prop_BruteForce = breakEnigma (zip "INCOMINGTRANSMISSIONNOTHINGTOREPORT"
    (runMachine "NOTHINGTOREPORTWILLREPLYINANHOUR")) ==
    "NOTHINGTOREPORTWILLREPLYINANHOUR"

> tests = do
> quickCheck prop_checkSubsitution
> putStrLn "Checked substitution"
> quickCheck prop_reflector
> putStrLn "Checked reflector"
> quickCheck prop_reflectorCheckSelfInverse
> putStrLn "Checked reflector inversion"
> quickCheck prop_mapping
> putStrLn "Checked mapping"
> quickCheck prop_enigmaEncryption
> putStrLn "Checked encryption"
> quickCheck prop_enigmaDecryption
> putStrLn "Checked decryption"
> quickCheck prop_enigmaDecipher
> putStrLn "Checked Deciphering"
> quickCheck prop_BruteForce
> putStrLn "Checked Encryption and Brute Force"
```

Benchmark.lhs

```
> import Bombe
> import Menu
> import Enigma
> import Criterion.Main

> benchmarkBreaking :: [(Char, Char)] -> [Rotor]
> benchmarkBreaking crib = findRotorCombination' (menuToChar(findMenu crib))

> crib1 :: [(Char, Char)]
> crib1 = zip "HELLO" "ILBDA"

> crib2 :: [(Char, Char)]
> crib2 = zip "BATTLEFIELD" "ADHFUNDBWPF"

> crib3 :: [(Char, Char)]
> crib3 = zip "WETTERVORHERSAGEBISKAYA" "RWIVTYRESXBFOGKUHQBAISE"

> crib4 :: [(Char, Char)]
> crib4 = zip "KEINEBESONDERENEREIGNISSE" "VRLBZPWMEPMIHFSRJXFMJKWRA"

> crib5 :: [(Char, Char)]
> crib5 = zip "INCOMINGTRANSMISSIONNOTHINGTOREPORT"
    "YIHKDEZRMXNGPMUTYAMVYESKXGYFREEJ"

> crib6 :: [(Char, Char)]
> crib6 = zip "NORTHNORTHEASTRETURNFIREEXPECTBACKUPSEVENTEENHUNDRED"
```

```
    "JORFRNRVPHFLLFBVSORKMQZUSNRAJJYSWBZQJFFEKKSHCBAAAOMV"

> crib7 :: [(Char, Char)]
> crib7 = zip "NORTHNORTHEASTRETURNFIREEXPECTBACKUPSEVENTEENHUNDREDH"
    "JORFRNRVPHFLLFBVSORKMQZUSNRAJJYSWBZQJFFEKKSHCBAAAOMVK"

> crib8 :: [(Char, Char)]
> crib8 = zip
    "NORTHNORTHEASTRETURNFIREEXPECTBACKUPSEVENTEENHUNDREDHOURSUBOATSAPPROACH"
    "JORFRNRVPHFLLFBVSORKMQZUSNRAJJYSWBZQJFFEKKSHCBAAAOMVYXHCRFTCJOGZHTLBAYM"

> main = defaultMain [
>   bgroup "cribs" [ bench "Length = 5" $ whnf benchmarkBreaking crib1
>              , bench "Length = 11" $ whnf benchmarkBreaking crib2
>              , bench "Length = 23" $ whnf benchmarkBreaking crib3
>              , bench "Length = 25" $ whnf benchmarkBreaking crib4
>              , bench "Length = 35" $ whnf benchmarkBreaking crib5
>              , bench "Length = 52" $ whnf benchmarkBreaking crib6
>              , bench "Length = 53" $ whnf benchmarkBreaking crib7
>              , bench "Length = 71" $ whnf benchmarkBreaking crib8
>              ]
> ]
```