Wordle solving algorithms using Information Theory

Algoritmos de resolución de Wordle usando Teoría de la Información

Authors: Ángel Benítez Gómez and Enrique Cavanillas Puga

Supervisor: Manuel Núñez García

Final Degree Project of the Double Degree in Computer Science Engineering and Mathematics

Faculty of Computer Science

Universidad Complutense de Madrid

Year 2021-2022

I am so clever that sometimes I don't understand a single word of what I am saying.
– Oscar Wilde

Abstract (English)

Wordle is a popular web based game where players must guess five-letter words in six attempts or less. Players are given hints about which letters are incorrect, misplaced or correct, and must use this information to discard and select new candidates for their next guesses. In this project we explore the connection between Information Theory and puzzle solving by using entropy-related concepts and algorithms to solve Wordle. First, we make a practical introduction to Information Theory, entropy and its applications. We define multiple Greedy and Genetic Algorithms and analyze them in order to improve their average scores and reduce their miss rates. We test our algorithms against relaxed versions of the puzzles to estimate a lower bound for the average scores of algorithm-based solutions. Finally, we discuss our results, provide suggestions for future reexaminations and present our candidate for best Wordle starter.

Keywords: Information Theory, Wordle, entropy, uncertainty, information, algorithms, genetic algorithms, puzzle solving.

Resumen (español)

Wordle es un juego de navegador popular en el que los jugadores deben averiguar palabras de cinco letras en seis intentos o menos. Se dan pistas a los jugadores acerca de las letras que son incorrectas, están mal situadas o son correctas, y deben usar esta información para descartar y seleccionar nuevos candidatos para sus siguientes intentos. En este proyecto exploramos la conexión entre la Teoría de la Información y la resolución de puzles mediante conceptos y algoritmos relacionados con la entropía. Primero hacemos una introducción práctica a la Teoría de la Información, la entropía y sus aplicaciones. Definimos múltiples algoritmos voraces y genéticos y los analizamos para mejorar sus puntuaciones medias y reducir sus porcentajes de fallo. Evaluamos nuestros algoritmos frente a versiones relajadas de los puzles para estimar una cota inferior de las puntuaciones medias de soluciones basadas en algoritmos. Finalmente, estudiamos nuestros resultados, sugerimos mejoras para futuras revisiones y presentamos nuestra palabra candidata a mejor apertura del Wordle.

Palabras clave: Teoría de la Información, entropía, incertidumbre, información, algoritmos, algoritmos genéticos, resolución de puzles.

Contents

Chapter 1

Introduction

Information Theory is a field of Mathematics that has made great contributions to the advancement of technology, and still has the potential to achieve major accomplishments in Computer Science, among many other fields. In this project we apply some of the core concepts of Information Theory to an apparently unrelated problem such as puzzle solving. In particular, we study the use of entropy in solving Wordle's puzzles. The goal of this project is to be a self contained exposition of the algorithms we used, with a detailed and practical introduction to Information Theory.

This chapter is an introduction to the key notions of Information Theory that will be used throughout the text and presents our work plan and descriptions of the design methods that we followed during the project. Chapter 2 describes how Wordle works and presents our Information Theory Algorithm. Chapter 3 consists of an introduction to Genetic Algorithms and our approach to solve Wordle using an algorithm of this kind. Chapter 4 describes our individual contributions. Finally, Chapter 5 contains a detailed discussion of the final results of our algorithms and our findings.

1.1 Historical background. What is information?

Information Theory is the scientific field that studies information transmission and storage, founded by Harry Nyquist, Ralph Hartley and Claude Shannon. The works of Nyquist and Hartley regarding information transmission [1, 2] laid the mathematical groundwork whereupon Shannon would later lay the foundation of Information Theory [3]. Shannon developed a way to quantify information through probability, a measure he would name *entropy*, by separating information from meaning.

To understand Information Theory and the metrics we will use to design our algorithms,

one must first fully understand the concept of entropy and information. When studying information storage, Shannon noticed three fundamental features.

- The amount of information in a message cannot be negative. When stripped of meaning, a message can only inform.
- The most unlikely events give the most information when observed. For example, when doctors examine their patients, they look for uncommon characteristics before making any decisions. Unremarkable, very likely features, give little to no information.
- The observation of independent events gives the same information as the observation of each event independently. The height and weight of a person will be somewhat related, so measuring one of these features will give us some information about the other. Learning the persons name, however, tells us nothing of their other features.

The amount of information h given by an event with probability $p \in (0, 1]$ is defined as $h(p) := -\log p$, which trivially satisfies the three features of information we described.

Example 1.1. Suppose we flip two coins. We can represent our sample space with 2 bits; each bit represents a coin, 0 represents *heads* and 1 represents *tails*.

	heads	*tails*
heads	00	01
tails	10	11

Figure 1.1: Sample space, two coins

Suppose we now measure the outcome of the first coin and we observe *heads*. Now we only need 1 bit to represent what remains of our sample space.

heads	*tails*
0	1

Figure 1.2: Sample space, one coin

By learning the outcome of the first coin, we *gained* one bit of information. This coincides exactly with

$$h(\frac{1}{2}) = -\log_2 \frac{1}{2} = 1 \text{ (bit)}$$

Just like $\log_2 n$ tells us the number of bits that are necessary to store $n \in \mathbb{Z}^+$ different values, h tells us how many bits of information we gain by observing an event, by measuring the reduction of the sample space in bits.

These concepts would later have a great impact on not just Communication Theory, but on many other scientific fields. Information Theory has made fundamental contributions to Computer Science, Probability and Statistics and Statistical Physics, among others. In the next section, we will define the entropy of a random variable as the *expected amount of information* given by its support.

1.2 Definition and Properties of Entropy

Entropy is the most important metric in Information Theory. It measures the amount of information –sometimes called uncertainty– in a random variable, and it has proven to be very useful in many parts of Computer Science, most notably in Deep Learning algorithms. In this section, we will introduce its basic definition and properties, along with some other noteworthy concepts. We will then show some properties that these metrics hold.

The concept of entropy was originally introduced by Claude E. Shannon [3] in his studies of the transmission of information. While there are many definitions of entropy, Shannon's proposal is the most commonly used.

Definition 1.1. Let X be a discrete random variable with alphabet $\mathcal{X}$ and probability mass function $p(x) = Pr\{X = x\}$, for $x \in \mathcal{X}$. Then, the entropy of X is defined as:

$$\mathcal{H}(X) = -\sum_{x \in X} p(x) \cdot \log_2\ p(x)$$

Remark 1.1. We usually choose base 2 for the logarithm because most digital systems are binary. By choosing base 2, we are also choosing the bit as the standard unit of information quantity. Entropy then becomes the average number of bits that are needed to describe a random variable.

Remark 1.2. We will use the convention that $0 \cdot \log 0 = 0$, based on the fact that when $x \to 0$ then $x \cdot \log x \to 0$. This follows the idea that if we add elements with null probability to the support of a random variable, the amount of information of the variable does not change.

Remark 1.3. Entropy can be understood as the expected amount of information of a random variable. Let $I(x) := -\log_2 p(x)$ for $x \in \mathcal{X}$ such that $p(x) \neq 0$, this function quantifies information of events as we described in Example 1.1. Then $\mathcal{H}(X) \equiv E[I(X)]$.

Let us show a simple example of how this definition may be applied when we are given the distribution of a random variable.

Example 1.2. Suppose we have a discrete random variable X that can take 16 different values and is uniformly distributed, that is, each element in $\mathcal{X}$ has a probability of $1/16$.

Then the entropy of X is:

$$\mathcal{H}(X) = -\sum_{x \in \mathcal{X}} p(x) \cdot \log_2\ p(x) = -\sum_{i=1}^{16} \frac{1}{16} \cdot \log_2\ \frac{1}{16} = \log_2 16 = 4$$

This result shows that we would need 4 bits to describe the information of the random variable, which is exactly the number of bits that we would normally use to code the 16 different values that the variable can take. This in fact leads to one of the most important applications of entropy in Computer Science, which we will study in more detail in Section 1.3.

The next definitions take inspiration from Probability Theory. So far, we have only used one random variable in our definitions. Can we describe the information given by two random variables? What about the information given by one random variable knowing the value of a different variable? The answer to both of these questions is yes, by using joint probability and conditional probability. By making parallelisms with probability, we can study the uncertainty of anything that can be described through multiple random variables.

Definition 1.2. Let X and Y be a pair of discrete random variables with supports $\mathcal{X}$ and $\mathcal{Y}$, respectively, and joint distribution $p(x, y) = Pr\{X = x, Y = y\}$, for $x \in \mathcal{X}$, $y \in \mathcal{Y}$. Then, the joint entropy of (X, Y) is defined as:

$$\mathcal{H}(X, Y) = -\sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p(x, y) \cdot \log_2\ p(x, y)$$

The conditional entropy of (X, Y) is defined as:

$$\begin{aligned}
\mathcal{H}(X|Y) &= \sum_{y \in \mathcal{Y}} p(y) \cdot \mathcal{H}(X|Y = y) \\
&= -\sum_{y \in \mathcal{Y}} p(y) \sum_{x \in \mathcal{X}} p(x|y) \cdot \log_2\ p(x|y) \\
&= -\sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p(x, y) \cdot \log_2\ p(x|y)
\end{aligned}$$

We will now introduce the concept of mutual information, which measures the quantity of information that a random variable contains about another random variable. In other words, it can be seen as the reduction in the uncertainty of a random variable given knowledge of another one.

Definition 1.3. Let X and Y be a pair of discrete random variables with marginal distributions $p(x) = Pr\{X = x\},\ x \in \mathcal{X}$, and $p(y) = Pr\{Y = y\},\ y \in \mathcal{Y}$, respectively, and joint distribution $p(x, y) = Pr\{X = x, Y = y\}$, for $x \in \mathcal{X}$, $y \in \mathcal{Y}$. Then, their mutual information is defined as:

$$\mathcal{I}(X;Y) = \sum_{x\in\mathcal{X}} \sum_{y\in\mathcal{Y}} p(x,y) \cdot \log_2 \frac{p(x,y)}{p(x) \cdot p(y)}$$

To finish this section, we will show some properties of the previous definitions that are commonly used in studies that involve the use of entropy.

Lemma 1.1. Let X be any discrete random variable. Then $\mathcal{H}(X) \geq 0$.

Lemma 1.2. Let X and Y be a pair of discrete random variables. Then, $\mathcal{I}(X;Y) \geq 0$ and $\mathcal{I}(X;Y) = \mathcal{I}(Y;X)$.

Theorem 1.1. Let X be a discrete random variable. Its entropy, $\mathcal{H}(X)$ is maximized when X is uniformly distributed.

This last result is very important because during many experiments or studies we might not know how our random variables are distributed. In an attempt to solve this problem, investigators follow the Maximum Entropy Formalism, assuming a uniform distribution whenever possible to maximize uncertainty.

Example 1.3. Let X be a random variable that takes the value 1 with probability p and 0 with probability $1 - p$. That is,

$$X = \begin{cases} 1 & \text{with probability } p \\ 0 & \text{with probability } 1-p \end{cases}$$

Then, the entropy of X, expressed in terms of p is

$$\mathcal{H}(X) = -p \log_2 p - (1-p) \log_2 (1-p)$$

Figure 1.3 shows the graph of the value of the entropy for the different values of p. Consonantly with Theorem 1.1, the maximum is reached at $p = 1/2$, which corresponds to a uniform distribution in X. The proof of the following result can be easily obtained by rearranging terms in Definitions 1.2 - 1.3.

Theorem 1.2. Let X and Y be two discrete random variables. Then we have the following results:

$$\mathcal{H}(X,Y) = \mathcal{H}(X) + \mathcal{H}(Y|X)$$
$$\mathcal{I}(X;Y) = \mathcal{H}(X) - \mathcal{H}(X|Y)$$
$$\mathcal{H}(X) - \mathcal{H}(X|Y) = \mathcal{H}(Y) - \mathcal{H}(Y|X)$$
$$\mathcal{I}(X;Y) = \mathcal{H}(X) + \mathcal{H}(Y) - \mathcal{H}(X,Y)$$

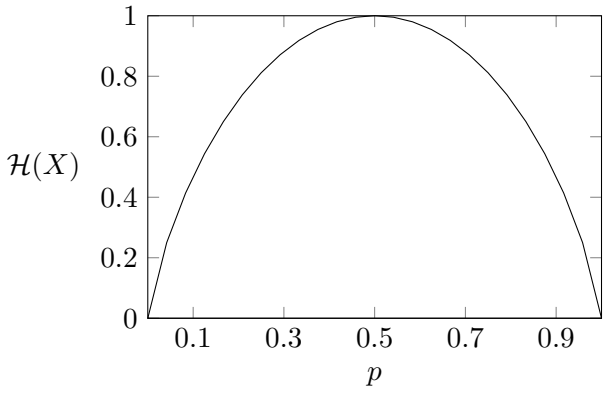


Figure 1.3: Entropy of X for the different values of p

Theorem 1.2 gives us an idea of how the different metrics are related to each other. Figure 1.4 places the different metrics in a Venn Diagram, making the relationships between them easier to appreciate.

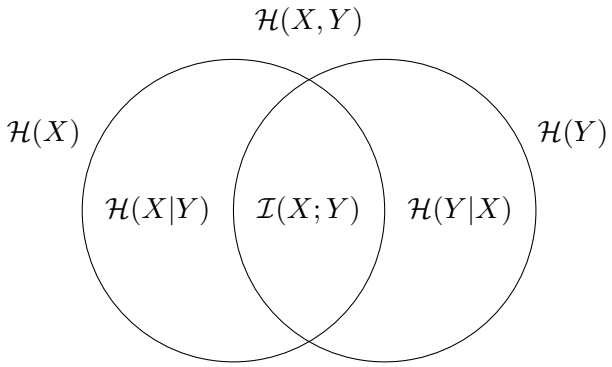


Figure 1.4: Relationship between entropy and mutual information

1.3 Applications of Entropy

In this section, we will present some examples of the uses of entropy. The first example connects entropy and data compression. Suppose two people are communicating digitally using up to eight possible different messages. If we used regular binary coding for these messages, we would need three bits to represent each of them. For instance, we could use the mapping shown in Figure 1.5.

This is not the only way we could code the messages, however. An alternative to regular coding is, for instance, Huffman Coding [4], which assigns binary codes to the messages in

Message	Code	Message	Code
`message1`	000	`message5`	100
`message2`	001	`message6`	101
`message3`	010	`message7`	110
`message4`	011	`message8`	111

Figure 1.5: Regular coding for eight messages

such a way that the most common ones use less bits than the least common ones. A possible Huffman coding for a random assortment of probabilities for the messages can be seen in Figure 1.6.

Message	Probability	Code	Message	Probability	Code
`message1`	0.323	11	`message5`	0.025	10101
`message2`	0.217	01	`message6`	0.179	00
`message3`	0.017	101001	`message7`	0.002	101000
`message4`	0.124	100	`message8`	0.113	1011

Figure 1.6: Huffman coding for eight messages

The weighted average length of the codes is $L = 2.68$ bits, which is less than the three bits that we would need regularly. It has been proven that entropy is a lower bound for the average length of any coding. Let X be the random variable over the possible messages from the previous example. The uncertainty of X , however, is $\mathcal{H}(X) = 2.43$, meaning entropy could be used to estimate data compression minimum sizes. This concept is reflected in *Shannon's First Theorem*, also known as the *Source Coding Theorem*.

Theorem 1.3 (Shannon's First Theorem)**.** Let X be a discrete random variable. Consider an optimal code for the possible values of X that has an expected length of $L = \sum_{x \in \mathcal{X}} p(x) \cdot l(x)$, where $p(x) = Pr\{X = x\}$ and $l(x)$ is the length of the code given to the element x. Then,

$$\mathcal{H}(X) \leq L < \mathcal{H}(X) + 1$$

Some Machine Learning algorithms also take advantage of entropy. Decision trees, for example, are a supervised learning method used to solve classification problems. ID3 is an algorithm invented by Ross Quinlan [5] that generates a decision tree from a data set using entropy. In every iteration, the algorithm calculates the entropy of each attribute and chooses the one with the lowest uncertainty value to make a tree node. To better understand how this algorithm uses entropy, we will give a more specific example.

Day	Outlook	Temperature	Humidity	Windy	Play Tennis
1	sunny	hot	high	false	no
2	sunny	hot	high	true	no
3	overcast	hot	high	false	yes
4	rain	mild	high	false	yes
5	rain	cool	normal	false	yes
6	rain	cool	normal	true	no
7	overcast	cool	normal	true	yes
8	sunny	mild	high	false	no
9	sunny	cool	normal	false	yes
10	rain	mild	normal	false	yes
11	sunny	mild	normal	true	yes
12	overcast	mild	high	true	yes
13	overcast	hot	normal	false	yes
14	rain	mild	high	true	no

Figure 1.7: Weather conditions for the last 14 days

Example 1.4. We have gathered information about the weather conditions for the last 14 days and determined whether tennis could be played or not. This information is shown in Figure 1.7. Let X be the random variable that describes whether tennis can be played. The algorithm checks which attribute – in this case among `Outlook`, `Temperature`, `Humidity` and `Windy`– has the lowest entropy in order to make the first node of the tree. Let O, T, H and W be the random variables over each of the attributes, respectively. The algorithm now computes the conditional entropy between X and these variables to choose the first attribute. Using Definition 1.2 and the conditional probabilities from Figure 1.8, we get:

$$\mathcal{H}(X|O) = 0.693$$
$$\mathcal{H}(X|T) = 0.911$$
$$\mathcal{H}(X|H) = 0.790$$
$$\mathcal{H}(X|W) = 0.891$$

The algorithm now chooses the attribute that leads to the lowest conditional entropy: `Outlook`. In other words, choosing `Outlook` maximizes the mutual information between X and each of the other random variables. The idea behind this is that the first question you make in the decision tree should be the one that gives you the most information, so that you need as few questions as possible to classify a new day. The algorithm would then repeat this process recursively, to generate the rest of the nodes and build the tree. The final result ID3 would produce is shown in Figure 1.9.

Outlook	Play Tennis	Probability	Temperature	Play Tennis	Probability
overcast	Yes	1	hot	Yes	2/4
	No	0		No	2/4
rain	Yes	3/5	mild	Yes	4/6
	No	2/5		No	2/6
sunny	Yes	2/5	cool	Yes	3/4
	No	3/5		No	1/4
Humidity	**Play Tennis**	**Probability**	**Windy**	**Play Tennis**	**Probability**
normal	Yes	6/7	true	Yes	3/6
	No	1/7		No	3/6
high	Yes	3/7	false	Yes	6/8
	No	4/7		No	2/8

Figure 1.8: Conditional probabilities

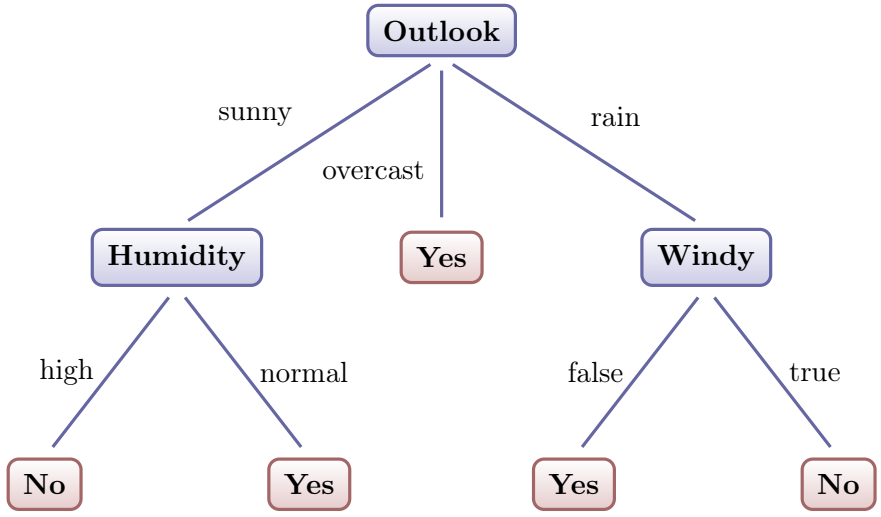


Figure 1.9: Decision tree generated by the ID3 algorithm

There are many other examples of how entropy can be used in different scientific fields. Many current investigations use entropy to define new metrics that could potentially have important consequences on their respective fields. For example, with regard to Software Testing, a metric called *Squeeziness* [6, 7], which is based on the concept of variation of entropy, has been defined to treat and analyze the propagation of errors in a program. Multiple studies regarding Squeeziness have been made already with promising results [8, 9, 10, 11]. Squeeziness, and its relation with *Failed Error Propagation*, is studied in detail in the *Trabajos de Fin de Grado* that we are presenting in the Faculty of Mathematics.

This concludes our introduction to Information Theory, which we have mostly based on T. M. Cover and J. A. Thomas's *Elements of Information Theory* [12]. In the next chapters we use these concepts to construct Information Theory algorithms with the goal of solving Wordle puzzles.

1.4 Project Management and Scheduling

Now we will describe the general organization structure that we followed during the entire project's duration, emphasizing the Project Management and Scheduling tasks.

This project followed a horizontal hierarchy with both members taking the same amount of responsibility and the same authority over work flow decisions and goals. With the help of our supervisor, we discussed the goals for each stage of the project, as well as the resources and dedication that should be directed to each part. An effort was made by both of us to devote the same amount of resources to the elaboration of the project, although individual involvement in each section of the project naturally varies, the extent of which we will discuss in Sections 4.1 and 4.2.

This report was entirely written in LaTeX and our algorithms were coded in Python 3.9. We used Github in order to organize the different code files of the project because we were familiar with the environment and it eased the coordination between us when uploading or modifying the different versions of our algorithms. The repository[1] has three different folders:

- `FirstExamples`: includes some of the first experiments we made.
- `GeneticAlgorithm`: includes the code for our Genetic Algorithms.
- `InformationTheoryAlgorithm`: includes the code for our IT Algorithms.

The decision to work on a Final Degree Project related with Information Theory was made in June 2021, when we first contacted our supervisor. Since we are students from the Double Degree in Computer Science Engineering and Mathematics, we needed to elaborate two Final Degree Projects and we both made our choices with Information Theory in mind. The project was officially assigned to us in September 2021. Initially, our project was to be based on the notion of *Squeeziness* [6] and its adaptation to quantum systems in black-box scenarios. For that reason, during the first semester of this academic year, we started investigating about the core concepts of Information Theory and Quantum Computing. With the start of the second semester, we began writing this report focusing on the Introduction to Information Theory.

[1] https://github.com/enricava/CustomWordleAlgs

While we were describing the fundamentals of Information Theory, we wanted to design original examples with real-world applications. At the time, Wordle was very interesting to us and we made several attempts to study it from an Information Theory perspective. After discussing our endeavour with our supervisor, in March 2022, we decided to move away from Quantum Computing and focus entirely on the Information Theory aspect of the project. From that point onward, we focused on designing our Wordle solving algorithms, maintaining constant communication, with continuous revisions and corrections of our work. The last month was mainly focused on revision to ensure the quality of the project.

With regard to the process of creating and revising the code, we mainly followed an agile iterative and incremental design method. The process of planning, developing, testing and meeting to interpret the results was very helpful in keeping a good workflow. We feel that we made the right choice because we were able to deliver experimental results as soon as new ideas or techniques came to mind, and we felt that we were always making good progress.

Chapter 2

Using entropy to solve Wordle

Wordle is a simple word game created by Josh Wardle in 2013 and published later in October 2021. This year, its puzzles are all the rage and garner more than three hundred thousand players on a daily basis [13, 14, 15]. The rules of the game are simple and the goal is even simpler. In this chapter we will briefly explain how the game works and we will design an Information Theory algorithm that can solve its puzzles in the fewest number of steps.

2.1 Wordle: Rules and Goal

Wordle is not a new game. In fact, Wordle is just a simpler modification of the 1955 head-to-head logic game *Jotto* [16]. What makes Wordle stand out is its simplicity and widespread competition online. Every day, a new secret word is chosen as the daily puzzle's solution. Players must follow the rules in order to find out which word it is.

The daily solution is *randomly* chosen from a list of commonly used five-letter words. Our task is to find this secret word in six or less tries. In each attempt, the player is allowed to input a five-letter word and receives clues pointing towards the answer in the form of colors over each input letter. The goal of the game is not only to find the secret word but also to achieve it using as few attempts as possible.

The clues are given to the player every time an input is entered, with the only restriction that it has to be a real five-letter word in the same language. We will illustrate each of the rules with an example, so that they become clear for the few who have not heard about this game yet. We encourage the reader to try to solve today's puzzle after reading the rules, to internalize the game mechanics. For the sake of simplicity, we will use the word `EULER` as the solution of an example puzzle.

1. Every letter that is not present in the solution will have its background turn **gray**.

E	U	L	E	R
C	H	A	I	N

2. The background of letters that are present in the solution but are misplaced will become **amber**.

E	U	L	E	R
R	O	A	R	S

   Note that only the first `R` has an amber background. This is because the solution itself only has one `R`.

3. Letters that are both present in the solution and placed in the correct position will get a **green** background. This rule overrides the previous rules, and therefore duplicates of a correctly placed letter in the input will not receive an amber background unless the letter is also repeated in the solution.

E	U	L	E	R
R	U	L	E	R

The next example showcases each of the rules in a possible sequence of inputs. We encourage the reader to follow closely if they are unfamiliar with the game mechanics.

Example 2.1. The following is a real attempt to solve a daily puzzle, without following any particular algorithm. Note that the player can use words that are not commonly used, such as `PALEA`, although daily solutions are guaranteed to be somewhat common words.

C	R	A	N	E
P	A	L	E	A
G	R	O	A	N
S	I	N	U	S
S	Q	U	A	D

The goal of hardcore Wordle players is to guess the solution in the fewest tries – three attempts or less are considered an excellent score– and therefore there is a huge debate regarding the best opening inputs. In the next section we will propose an algorithm based solely on Information Theory concepts in an attempt to produce a game winning algorithm. As a side effect of this goal, we will also present our candidate for best opener word when we discuss the results in Section 5.1.

2.2 Information Theory Algorithm

The algorithm we created is based solely on Information Theory concepts and uses entropy to rank words according to how much information they provide. The intuition behind this choice is that entropy is a way to measure how a sample space is partitioned after knowing a piece of information. When a player submits a five letter word as an attempt, the pattern that results after comparing the submitted word and the solution produces a reduction in the sample space – words that are incompatible with the pattern should be eliminated from it, or should not make it into further attempts to reach the solution. An argument can be made that testing words incompatible with the pattern is a good way to make out a couple of letters that can be present in the answer, which is why this algorithm is not presented as an optimal solution.

Suppose the player submitted the word `BLACK` and obtained the following answer:

B	L	A	C	K

Then, the remaining 17 compatible words are:

chack	crack	drack	frack	kiack	knack	kyack	quack	shack
smack	snack	stack	swack	thack	track	whack	wrack	

This reduction of the sample space brings conditional probability to mind: this is what gave us the idea to use Information Theory on Wordle. For any given word, we can assign to each pattern its probability of occurring by counting how many words would be left in the sample space, and assuming a uniform distribution over the set of allowed words. For this particular pattern, its probability is 17 divided by the number of allowed words.

We will identify each of the 3^5 possible patterns by its ternary representation, where 0 identifies gray, 1 identifies amber and 2 identifies green. The leftmost 'trit' represents the leftmost position in the word.

Figure 2.1 presents the relative frequency of each pattern for the word `BLACK`. The bar plot of Figure 2.1 is very intuitive: the pattern represented by 0 contains all the words that do not share any letter with `BLACK`, and it makes sense that it is also the most frequent pattern. The frequencies seem to be –almost– decreasing, and this is a consequence of the ternary representation we chose, where patterns represented by smaller numbers are produced by the words with the least similarities to `BLACK`. The appearance of spikes also makes sense given that words are not simply random sequences and, therefore, many words share similarities.

Figure 2.1 is also an interesting example to motivate why we chose entropy as a measure of how good a guess can be. The most improbable patterns are also the ones that give us the

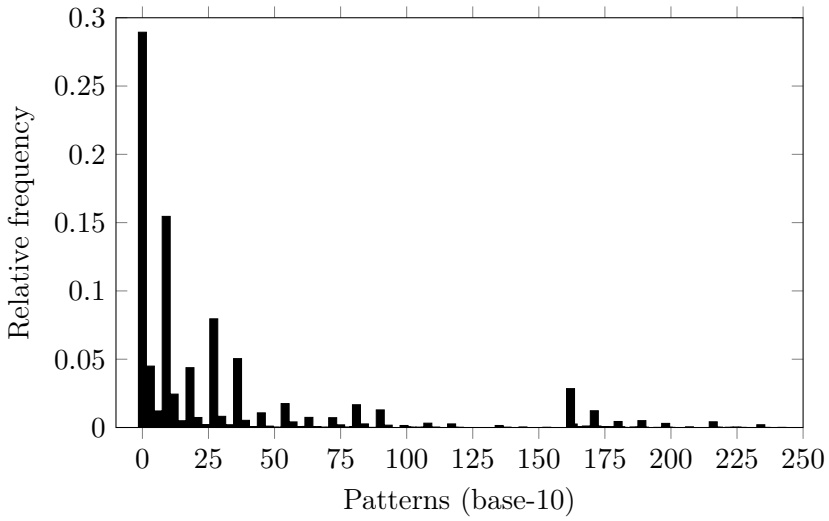


Figure 2.1: Relative pattern frequencies for `BLACK`

most information if they are chosen. For example, if the leftmost pattern –all grays– were to appear after using the word `BLACK`, we would not learn much about the solution since it is the most probable pattern. On the other hand, if we obtained the rightmost pattern –all greens and least probable– we would obtain all knowledge of the solution.

We need to know every pattern for every allowed word. The list of words that can be used has been made publicly available: it contains 12972 words and only a subset of those can be answers. This subset of possible solutions can be obtained by inspecting the game's code as well, but we will only use this knowledge to simulate the games and not as a cheat to reach the solutions in fewer guesses. While we explore this idea further in Section 2.2.5, we believe that our algorithm's goal should be to play following the same rules as a human player, which entails not knowing which words can be solutions.

Before we continue, we must set a boundary for the tools our algorithms can use: we may use the list of allowed guesses in our algorithms because inputting words that do not exist has no impact on the games –failed attempts to produce an allowed guess are not penalized by Wordle's rules, and the game simply ignores the guesses until a valid one is entered–. By limiting our guesses to the list of allowed words, we are effectively removing the process of 'producing valid guesses' without breaking any rules or using knowledge that should be kept secret from the player. On the other hand, our algorithms may not use the knowledge of which words can ultimately be solutions because the player is not meant to be able to tell the difference. We will, however, design our simulation bench to test our algorithms against the list of solutions to be able to accurately benchmark the results.

2.2.1 Obtaining the patterns

To find the pattern that we would obtain after submitting `word2` when the solution is `word1`, we first do a *green pass* over both strings and mark the letters that are correctly positioned. Then, we do an *amber pass* with the remaining letters to find the ones that are misplaced. The rest of the letters are gray – corresponding to 0 in our ternary representation – and therefore need no further consideration.

```
def pattern(word1, word2):
    used1 = [False for _ in range(5)]
    used2 = [False for _ in range(5)]
    return ( greenPass(word1, word2, used1, used2)
           + amberPass(word1, word2, used1, used2) )
```

Figure 2.2: Pattern function 1

The green pass simply finds out which letters are the same when checking the same position in each word and marks them as used so that they are not checked again in the amber pass. The amber pass then checks for misplaced letters that have not already been previously used, that is, each letter in `word1` can only account for one color change according to the rules of the game.

```
def greenPass(word1, word2, used1, used2):
    result = 0
    for i in range(5):
        if word1[i] == word2[i]:
            result += 2 * 3**(4-i)
            used1[i] = used2[i] = True
    return result
```

```
def amberPass(word1, word2, used1, used2):
    result = 0
    for (i,c1) in enumerate(word1):
        for (j,c2) in enumerate(word2):
            if c1 == c2 and not (used1[i] or used2[j]):
                result += 3**(4-j)
                used1[i] = used2[j] = True
                break
    return result
```

Figure 2.3: Green and amber passes 1

The first iterations of our algorithms used the code in Figures 2.2 and 2.3. The computation of the patterns for every pair of words took around 15 minutes. In order to shorten execution time, we replaced loops by vectorized operations taking inspiration from Grant Sanderson's implementation [17]. By using vectorized operations inside Python's Numpy module, we can make use of SIMD instructions and decrease execution times substantially.

After refactoring our code, we now compute every pattern simultaneously. Let `list1` and `list2` be lists of words with lengths `l1` and `l2`. We will compute the patterns that result after submitting each word in `list1` when the solutions are the words in `list2`.

First, we will create an equality matrix with the results of comparing each pair of words in a tensor-like manner by using Numpy's `ufunc.outer()` function. The code snippet in Figure 2.4 produces a `l1` $\times$ `l2` $\times$ 5 $\times$ 5 matrix where:

`equalities[a,b,i,j] = (list1[a][i] == list2[b][j])`.

```
import numpy as np
from itertools import product
...
equalities = np.zeros((l1, l2, 5, 5), dtype=bool)
for i, j in product(range(5), range(5)):
    equalities[:,:,i,j] = np.equal.outer(list1[:,i], list2[:,j])
```

Figure 2.4: Equality matrix

Next, we follow the same steps as in Figures 2.2 and 2.3 and make green and amber passes. We will store the results in an `l1` $\times$ `l2` matrix. By exploiting matrix indexing, we are again taking advantage of modern processors' architectures and speeding up the code.

```
matrix = np.zeros((l1,l2), dtype=np.uint8)

#Green pass
for i in range(5):
    # matches[a, b] : words1[a][i] == words2[b][i]
    matches = equalities[:, :, i, i].flatten()
    matrix.flat[matches] += 2 * 3**(4-i)
    for k in range(5):
        # Avoid amber pass
        equalities[:, :, k, i].flat[matches] = False
        equalities[:, :, i, k].flat[matches] = False
```

Figure 2.5: Green pass 2

```
# Amber pass
for i, j in product(range(5), range(5)):
    # matches[a, b] : words1[a][i] == words2[b][j]
    matches = equalities[:, :, i, j].flatten()
    matrix.flat[matches] += 3**(4-i)
    #Avoid next passes
    for k in range(5):
        equalities[:, :, k, j].flat[matches] = False
        equalities[:, :, i, k].flat[matches] = False
```

Figure 2.6: Amber pass 2

After incorporating the instructions in Figures 2.5 and 2.6, the code for the computation of the patterns saw a significant speed up.

2.2.2 Precomputation and best starters

Like we mentioned before, we are using a ternary representation for the patterns. Therefore, instead of storing lists of colours, we store the base-10 number that represents each of the possible 3^5 patterns. To further optimize storage space, we use Numpy's `uint8` variables for the patterns and we use positions in the list of allowed words to index our matrix. For example, the section of our pattern matrix where the words `PIVOT` and `PIXEL` are submitted against the solutions `JOLLY`, `JOLTS` and `JOLTY` is:

...	jolly	jolls	jolty
pivot	[0,0,0,1,0]	[0,0,0,1,1]	[0,0,0,1,1]
pixel	[0,0,0,0,1]	[0,0,0,0,1]	[0,0,0,0,1]

$\rightarrow$

...	5585	5586	5587
8294	3	4	4
8295	1	1	1

Let $f_{w,i}$ be the relative frequency of the i-th pattern after submitting the word w. We compute the entropy of w as described in Definition 1.1, using a random variable X_w over the set of patterns with probability distribution $p_{X_w}(i) = f_{w,i}$ for $0 \leq i \leq 3^5 - 1$.

$$\mathcal{H}(X_w) = -\sum_{i=0}^{3^5-1} p_{X_w}(i) \log_2 p_{X_w}(i)$$

For example, the entropy of the word `BLACK` that we used as an example in Figure 2.1 is equal to:

$$\mathcal{H}(X_{\texttt{BLACK}}) = 4.140 \text{ bits}$$

We are ready to rank each word according to its entropy. Figure 2.7 contains the top 10 words sorted by decreasing entropy.

w	$\mathcal{H}(X_w)$
TARES	6.1940525443754435
LARES	6.149918742453124
RALES	6.114343099454227
RATES	6.096242642514605
TERAS	6.076619177276175
NARES	6.066830765753897
SOARE	6.061395399096263
TALES	6.054987761401191
REAIS	6.049777632888325
TEARS	6.032338670239807

Figure 2.7: Naïve best starters

We call this first list the *naïve best starters*. According to the list, TARES should be the most informing first word on average.

2.2.3 Naïve Algorithm

Our first attempt naively assumes that entropy alone is the best metric for ranking the words. The algorithm starts with a user input opener and obtains a pattern. Then, it removes words from the sample space that are incompatible with the pattern and computes a new ranking for the remaining compatible words. The algorithm picks the best ranked word and repeats these steps until either finding the solution or reaching 9 attempts.

```
def play(solution, allowed=words, starter='tares', mode='naive'):
    remaining_words = np.array(allowed, dtype = 'str')
    next = starter
    for attempt in range(9):
        pattern = get_pattern(next, solution)
        if pattern == 242:
            break   # found the solution!
        remaining_patterns = get_pattern_matrix(next, remaining_words)
        remaining_words = remaining_words[remaining_patterns.flatten() ==
    pattern]
        i, next = make_guess(remaining_words, mode)
        remaining_words = np.delete(remaining_words, i)
    return attempt
```

Figure 2.8: Structure of the simulations

As we mentioned before, the list of all possible solutions is available. So, in order to

test how well the algorithm performs, we simulated each of the 2309 possible games. To verify that the *naïve best starters* are not necessarily the best starters –or well sorted–, the simulations were run 10 additional times, starting the games with each of the top ten *naïve best starters*.

After the simulations were finished, it became clear that `TARES` was not the best opener. By following our naïve strategy, `TALES` produced a lower score on average. We discuss the results of the algorithm in more detail in Section 5.1. The main flaw of this approach is that, while the solutions of the puzzles are guaranteed to be somewhat common words, the algorithm will often favor uncommon words that are more informative over words that could be the solution, resulting in a worse score.

2.2.4 Greedy algorithm

The next step is to somehow distinguish common words from uncommon words. A *cheat* that produces excellent results is to only use words from the list of possible solutions. However, we would like to design an algorithm that can solve Wordle puzzles without knowing which words can be solutions. We use Python's module `wordfreq` to obtain the relative frequencies of each of the allowed guesses in the English language. We do not only want to use word frequencies to break ties. In addition, we would like our algorithm to sometimes prioritize guesses that are more common, even if they have a lower entropy.

Let G be the set of allowed words and r_w be the relative frequency of the word $w \in G$. The greedy algorithm will always choose the word that reaches the following maximum:

$$\max_{w \in G} \{\, g(r_w) \cdot \mathcal{H}(X_w) \,\}$$

where $g : \mathbb{R} \to (0, 1]$ is a continuous non decreasing function that we must define. In fact, most of the time invested in this adaptation of the naïve algorithm was spent deciding the behaviour of g. For example, when g is the identity function, words such as `TARES`, with a very high entropy, are never chosen because of their low relative frequency. In this case, $r_{\texttt{TARES}} = 7.94 \cdot 10^{-8}$.

After much consideration, it became clear that the values returned by g should be very close to binary values –that is, close to 0 and 1– so that the least common words have a very low prospect of being chosen and common words have a higher prospect of being chosen. We explicitly exclude 0 to allow cases where the expected amount of information to be gained balances a low relative frequency. Thus, with this behaviour we mimic the act of 'sacrificing' a guess to learn more information and achieve an overall lower score.

Defining g as a step function with values 0 and 1 would be the same as restricting the set of allowed words to a subset, and, as we mentioned before, there are cases where we want

to be able to use words with very low relative frequency. Instead of arbitrarily determining which words make it into this new subset of words, we use a shifted and adjusted sigmoid function such as:

$$g(r_w) = \frac{1}{1 + e^{-10 \cdot (r_w - 0.5)}}$$

so that the transition between 0 and 1 is smoother and no words are immediately discarded.

The algorithm that chooses the next guess can be seen in Figure 2.9. We use the `@cache` decorator from the `functools` module to take advantage of memoization and speed up the simulations, and compute the entropy of the words via a Numpy dot product.

```
@cache
def make_guess(wordlist, mode):
    patterns = get_pattern_matrix(wordlist, wordlist)
    l = len(wordlist)
    maxe = best = -1
    for i,w in enumerate(patterns):
        # List of distinct patterns and their counts
        ps, counts = np.unique(w, return_counts=True)
        e = - np.dot(counts/l, np.log2(counts/l))
        if mode == 'greedy1':
            e *= get_word_priority(wordlist[i])
        if e > maxe:
            maxe, best = e, i
    return best, wordlist[best]
```

Figure 2.9: Next guess choice function

Before advancing to Section 3.1, where we attempt to solve Wordle by using Genetic Algorithms, we will test our algorithms with different 'relaxed' versions of the original problem.

2.2.5 Relaxations of the problem

Although we have implicitly described the goal of our algorithms, we would like to precisely describe the problem that we are solving. We call the original problem P_0.

P_0
Find the solution to any Wordle puzzle in as few attempts as possible using the words from the list of allowed words.

In order to test whether the Greedy Algorithm could be improved, we decided to modify the Naïve Algorithm so that its goal is to solve P_1, a relaxation of P_0.

P_1
Find the solution to any Wordle puzzle in as few attempts as possible using *only words from the list of solutions.*

By removing many words from the set of allowed words, we are also removing uncertainty from the problem. In order to show this, let X and Y be uniformly distributed random variables over the set of allowed words G and the set of Wordle solutions S, respectively. Then:

$$\mathcal{H}(X) - \mathcal{H}(Y) = \log_2 |G| - \log_2 |S| = \log_2 12972 - \log_2 2309 = 2.49 \text{ (bits)}$$

It makes sense that, by reducing the set of allowed words to a fourth of its size, the solutions can be reached in less attempts, especially when there are words in S with similar entropy to those in G. In Figure 2.10 we enumerate the 80 best naïve starters, with entropies between 6.2 and 5.8 bits. Note how there are many solutions –in blue– among the most informing words.

TARES	ARLES	LANES	SARED	LEATS	TAROS	RAITS	SNARE
LARES	TORES	LAERS	ROLES	NEARS	RILES	RACES	SIREN
RALES	SALET	PARES	TEALS	TOEAS	ARETS	RITES	PALES
RATES	AEROS	CARES	AURES	STRAE	HARES	LARIS	TONES
TERAS	DARES	TIRES	EARLS	RONES	TORAS	RAPES	RINES
NARES	SANER	SAINE	TAELS	NATES	SLATE	LEANS	LASER
SOARE	REALS	SERAL	RAISE	EARNS	DEARS	AEONS	DATES
TALES	LEARS	MARES	TRIES	TASER	SOLER	DORES	TILES
REAIS	LORES	REANS	ROTES	TOLES	GARES	RAILS	LAIRS
TEARS	SERAI	ALOES	SOREL	DALES	BARES	STARE	READS

Figure 2.10: Solutions among best naïve starters

After testing our Naïve Algorithm for the problem P_1, we obtained a significantly better score, even better, in fact, than with the Greedy Algorithm and P_0. We will discuss this further in Section 5.1, but the reason behind this is that the set S of solutions does not have strictly more common words than any of $G \setminus S$, because S has been handpicked without following any particular method.

We also decided to test our algorithms for uninterrupted, competitive Wordle. Competitive Wordle players keep an average of their accumulated scores in order to compete with each other. Instead of playing singular Wordle puzzles, we decided to test the algorithms for the continued solution of Wordle's daily puzzles.

P_2

Find the solution to *all of Wordle's daily puzzles* in as few attempts as possible using the words from the list of allowed words.

The introduction of P_2 allows us to make use of a hidden rule of Wordle in our simulations: *any daily puzzle can only happen once.* Therefore, past solutions will no longer be considered. This relaxation has a peculiarity, which is that the average score depends on the order of the sequence of tested solutions.

Finally, just for the sake of curiosity, we also tested our algorithms for a final relaxation of P_0, which is a combination of the previous problems P_1 and P_2:

P_3

Find the solution to *all of Wordle's daily puzzles* in as few attempts as possible using *only words from the list of solutions.*

The simulations of P_3 gave us the best scores for both of the algorithms and we consider these a lower bound for the scores that we can achieve without cheating.

The results of our simulations for the problems P_0, P_1, P_2 and P_3 will be discussed in more detail in Section 5.1.

Chapter 3

Using Genetic Algorithms to solve Wordle

In this chapter, we will study other ways in which we may find solutions for Wordle. In particular, we will study the use of Genetic Algorithms. The chapter starts with an introduction to Genetic Algorithms, where we will present their general structure and show specific examples. Next, we show and discuss the techniques we used in our algorithm.

3.1 Introduction to Genetic Algorithms

Genetic Algorithms [18, 19] are an attempt to mimic the processes that take place in natural evolution, invented by John Holland [20] in the 1960s. They aim to emulate a population's evolution over time, taking inspiration from natural genetic variation, which involves the combination of chromosomes or their mutation, and natural selection or 'survival of the fittest'. Evolutionary computation techniques try to convert these properties of natural evolution into algorithms that can find optimal solutions for a class of problems. The task of a Genetic Algorithm is to find an optimal solution among the many possible solutions that may exist for a given problem. This would be done by generating populations with increasingly superior individuals. The advantages that Genetic Algorithms have over other search algorithms are their simple operators and that they need very few parameters in comparison to them. In fact, the 'only' thing a Genetic Algorithm needs is a function that is able to describe how good or 'fit' a solution is.

The terminology used in Genetic Algorithms comes from the analogy made with genetics. For that reason, we shall start this introduction by defining some fundamental concepts:

- Chromosome: It is a candidate solution to the problem, often encoded as a string of bits.
- Genes: They are the parts in which a chromosome is divided into. If a chromosome is a bit string, then a gene could be one of those bits or a group of them if they all represent an aspect of the solution.
- Alleles: They are the possible values that a gene can take. In the case of bit strings, alleles would be either 0 or 1.
- Fitness function: It is a function that assigns chromosomes a value in terms of how adequate they are. This function helps classify chromosomes heuristically; we say a chromosome is better than others if it has a higher fitness value. Fitness functions help us artificially select candidates for evolution, which is how we emulate natural selection in an attempt to find the optimal solution.

Note that in the previous definitions we mention the codification of a possible solution. That is because the first step in a Genetic Algorithm is to encode solutions in a way that makes it easy to work with the operators used. The most common procedure is to use a string of bits to represent a possible solution, but other approaches can be made. For instance, you could encode a solution using a string of numbers from 0 to 9. Later in this section we will provide an example of a Genetic Algorithm that illustrates these ideas.

Now that we have defined how a solution might be represented, or encoded, we need to do some operations with our population of chromosomes to produce new generations of individuals. This is the part of Genetic Algorithms that ties with the interactions that take place during natural evolution. The operators used are the following:

- Selection/Reproduction: It is the process in which individuals of the population are selected to produce offspring. Individuals with higher fitness values have a higher probability of producing offspring for the next generation. This is the operator that matches the idea of natural selection. One of the most typical and simple ways to implement this operator is to create a biased roulette wheel where each individual has a wheel slot sized proportionally to its fitness value. Let us show an example of this. Suppose our population is composed of the individuals presented in Figure 3.1. The weight of each chromosome is computed by the following expression:

$$p_i = \frac{fitness(x_i)}{\sum_{i=1}^{n} fitness(x_i)}, \ i = 1, 2, ...n,$$

where $\{x_1, x_2, ..., x_n\}$ are the individuals of the population. Therefore, for the previous example we would get the weights shown in Figure 3.2.

	Chromosome	Fitness Value
x_1	1 0 1 0 0 1 0	151
x_2	0 1 0 1 1 1 1	245
x_3	1 0 1 0 1 0 1	491
x_4	1 0 0 0 1 0 1	94

Figure 3.1: Example of a population in a Genetic Algorithm

	Fitness Value	Weight (p_i)	Cumulative Probability (q_i)
x_1	151	$151/981 = 0.154$	0.154
x_2	245	$245/981 = 0.25$	0.404
x_3	491	$491/981 = 0.501$	0.905
x_4	94	$94/981 = 0.095$	1

Figure 3.2: Information used to create a biased roulette wheel for a population

The corresponding roulette wheel that would be used for this example population is shown in Figure 3.3. A simple method to select individuals from this structure is to generate a random number a between 0 and 1. Then, the individual x_i will be selected if $q_{i-1} < a \leq q_i$, for $1 \leq i \leq n$ and $q_0 = 0$, where q_i are the cumulative probabilities given in Figure 3.2.

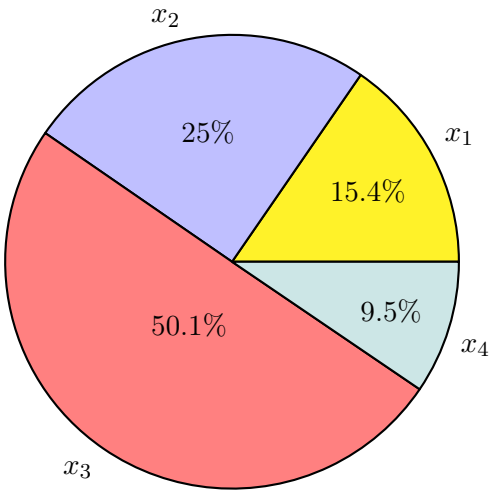


Figure 3.3: Example of a roulette wheel used for the selection operator

- Crossover: It is the process in which the selected individuals combine with each other. That is, two individuals combine their genes to produce a new chromosome. The most commonly used method to implement crossover is by dividing the strings that

represent individuals into two parts at a random point $k \in [0, l-2]$, where l is the length of the chromosomes, and swapping all the genes between positions $k+1$ and $l-1$, both included. For instance, consider the chromosomes $x_1 = 1\ 0\ 1\ 0\ 0\ 0\ 1$ and $x_2 = 0\ 1\ 0\ 1\ 1\ 0\ 0$. We would need to generate a random integer between 0 and 5. Suppose we get $k = 3$, then the offspring generated in this crossover would result from swapping the substrings from positions 4 to 6 between the chromosomes. The resulting chromosomes would be $x_1' = 1\ 0\ 1\ 0\ 1\ 0\ 0$ and $x_2' = 0\ 1\ 0\ 1\ 0\ 0\ 1$. This method for implementing crossover is called 1-point crossover. It is important to note that depending on how the representation of the possible solutions is designed, a crossover might produce a new individual that does not meet the requirements of a solution. In these situations, we would need to define the crossover operator more carefully. Consider the following example: you are tasked to maximize a function f that takes 10 arguments in the form of numbers from 0 to 9, with the restriction that you can only use each number once. Suppose you have the following chromosomes in your population, and they both reach high values for f.

$$\text{Chromosome } x_1\text{: } 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 0 \rightarrow f(1,2,3,4,5,6,7,8,9,0)$$
$$\text{Chromosome } x_2\text{: } 6\ 5\ 8\ 3\ 9\ 0\ 1\ 7\ 2\ 4 \rightarrow f(6,5,8,3,9,0,1,7,2,4)$$

It would be reasonable to think that by combining x_1 and x_2 we could maximize f further. If we implement a 1-point crossover like the one we have just described, we could get the following offspring (suppose $k = 3$):

$$x_1' = 1\ 2\ 3\ 4\ 9\ 0\ 1\ 7\ 2\ 4$$
$$x_2' = 6\ 5\ 8\ 3\ 5\ 6\ 7\ 8\ 9\ 0$$

Clearly, none of the offspring produced fit the problem requirements. It is the task of the crossover operator to handle these situations: either by defining an operation that always generates possible solutions or by rejecting offspring which do not meet the requirements of the problem.

- Mutation: It is an occasional random alteration of a gene in a chromosome. Although it could seem counter-intuitive, mutation can be useful to reach possibilities that might otherwise not have been examined with just the use of crossover. Genetic Algorithms usually set a low probability for mutation. When using the bit string representation, mutation often negates a random bit in the string. Similarly to the crossover operator, situations in which the new individual is not a possible solution must be taken care of.

The only thing that remains to be seen is how the algorithms are initialized and when they stop. Initialization often consists of randomly generating a number of individuals which will be part of the first population. The algorithm will stop after the optimal solution is

Data: $N > 0$, $maxNG \geq 0$, $crossoverProb \in [0,1]$, $mutationProb \in [0,1]$
Result: Best solution found for the problem
$population \leftarrow createFirstGeneration(N)$
$numberGens \leftarrow 0$
while not $solutionFound$ **and** $numberGens \neq maxNG$ **do**
  $getFitness(population)$
  $nextGeneration \leftarrow empty$
  for $i = 0$ **to** $N/2$ **do**
    $parent1, parent2 \leftarrow select(population)$
    $r \leftarrow randomValue(0,1)$
    if $r < crossoverProb$ **then**
      $child1, child2 \leftarrow crossover(parent1, parent2)$
    else
      $child1, child2 \leftarrow parent1, parent2$
    end
    $mutation(child1, mutationProb)$
    $mutation(child2, mutationProb)$
    $add(nextGeneration, child1, child2)$
  end
  $population \leftarrow nextGeneration$
  $numberGens++$
end
return $bestIndividual(population)$

Algorithm 1: Pseudocode for a Genetic Algorithm

found or after a previously set number of generations have been evaluated. In the latter case, the algorithm will end with a final population from which we may choose an individual as the best solution found. The general structure of Genetic Algorithms is given in Algorithm 1.

To finish our introduction, we will explore the solution to a problem that can be solved through Genetic Algorithms, emphasizing the importance of the decisions that have to be made to design a Genetic Algorithm.

Example 3.1. Suppose we have a device with five switches. For every combination of the switches, the device gives us an output value which we want to maximize; for simplicity we call this function f. Our task here is to find the combination of switches that maximizes the output of f. Note that we do not have any information about f –we can only test f by getting output values for specific combinations of the switches. In order to create a working Genetic Algorithm, the following steps should be taken:

- Defining the representation of a solution.
  We are working with a device that has five switches and each of them has two different positions, *on* and *off*. We choose the usual binary representation where 1 represents *on* and 0 represents *off*. As there are five switches, we will need a string of five bits to represent each combination of the switches. In other words, a possible solution will be a bit string of length 5. Therefore, using the terminology we have defined, we have:
  - Chromosome: A bit string of length 5.
  - Gene: Each bit of the string, which represents a switch.
  - Allele: 0 or 1 for each gene, which represents a switch being on or off.
- Defining the fitness function.
  Our choice of fitness function should allow us to measure how good a given possible chromosome is. In this example, we can simply use f as our fitness function.
- Implementing a selection operator.
  We will implement this operator with a biased roulette wheel, as we defined it previously in this section.
- Implementing a crossover operator.
  Given two possible solutions, note that a 1-point crossover between them produces chromosomes that still meet the problem's requirements, that is, they will still be strings composed of only 0's and 1's. We choose a probability of 0.7 for a crossover occurring.
- Implementing a mutation operator.
  The choice of binary representation allows us to define mutation as the bitwise negation of a random bit in the string. Another possible mutation operator could be defined as swapping two genes in a chromosome. We will give mutation a probability of 0.1.

Now that all of the algorithm's parameters have been set, we will show the first iteration of the execution of our Genetic Algorithm, following the structure of Algorithm 1.

1. Create the first generation.
   We will randomly generate `n = 4` (small for Genetic Algorithms standards) bit strings of length 5. The first population is made up of the individuals shown in the next table:

Chromosome	Fitness Value	Chromosome	Fitness Value
0 0 0 0 1	5	1 1 1 0 1	136
1 0 1 0 1	24	0 1 1 0 0	37

2. Selection.
   First, the results of implementing the roulette wheel are shown below:

Fitness Value	p_i	q_i
0 0 0 0 1	$5/202 = 0.154$	0.025
1 0 1 0 1	$24/202 = 0.25$	0.144
1 1 1 0 1	$136/202 = 0.501$	0.817
0 1 1 0 0	$37/202 = 0.095$	1

Now, we generate two random numbers between 0 and 1: `r1 = 0.8887` and `r2 = 0.1801`. The wheel's design means `parent1` = 0 1 1 0 0 and `parent2` = 1 1 1 0 1 are chosen as parents for a crossover.

3. Crossover.
   We generate a random number to check whether we should apply the operator or not. We get `r = 0.2307 < 0.7`, which means a 1-point crossover will happen, and `k = 1` is randomly selected as the position for the crossover. The offspring are `child1` = 0 1 1 0 1 and `child2` = 1 1 1 0 0.

4. Mutation.
   We generate two random numbers that will determine whether a mutation takes place in each of the offspring. This can be seen below:

   `r1 = 0.0415 < 0.1` → `k=3`. Negate `child1`[3] → `child1` mutates to 0 1 1 1 1
   `r2 = 0.1471 > 0.1` → No mutation → `child2` stays the same

5. The chromosomes generated by the crossover and mutation operators are then added to the next generation and the process and steps 2-5 are repeated until the maximum number of generations is reached.

Once the maximum number of generations have been evaluated and a final population is reached, the algorithm chooses the most fit individual from the final population as the solution to the problem.

3.2 Solution to Wordle with a Genetic Algorithm

From what we saw in the previous section, we could say that the idea behind Genetic Algorithms is to have an initial collection of possible solutions that, after combining with each other, lead to better and better solutions until the best one is found. This idea seemed very similar to how Wordle works in the sense that one starts with a word and tries to 'improve' guesses for the next attempts until the solution is found. This is what gave us the motivation to try to solve Wordle with a Genetic Algorithm. In this section, we will explain how we approached the problem to design a Genetic Algorithm for the game.

3.2.1 First Genetic Algorithm

Our first approach was to design our algorithm by following the exact structure of Genetic Algorithms, while using our knowledge of Information Theory. However, we encountered several problems in our endeavour to adapt some of the ideas that characterize these algorithms.

The first obstacle we encountered was the choice and behaviour of the fitness function. In order to determine the fitness value of a guess, we have to be able to measure how close it is to the solution of the puzzle. The fitness function is usually a 'fixed' function in the sense that it behaves the same way for every generation. In this case, depending on what we know about the solution, the next guesses will have a varying degree of compatibility. For example, if we know that the solution has a letter `B` in the first position, the words `BLACK` and `BLOCK` could have very similar fitness values. If we then learn that the solution also contains a letter `A`, then `BLACK` should have a higher fitness value than `BLOCK`. The simplest fitness function would be one that directly compares every individual with the solution, but we decided to dismiss this idea because our algorithm must abide by the rules of the game, and therefore should only compare guesses with the solution 6 times. Since the objective is to make as few guesses as possible, we decided the algorithm should only attempt to guess the word once for every generation. Our fitness function would have to take into account the letter matching of a word with the information that we have obtained about the solution. Entropy would then be used to 'break ties' between words that share similarities with the solution. The formula that we reached for the function is the following:

$$fitness(w) = \mathcal{H}(X_w) + 5 \cdot G(w) + 2 \cdot A(w) - 2 \cdot N(w),$$

where $G(w)$ is the number of correctly placed letters in w, $A(w)$ is the number of letters that we know are present in both w and the solution but are misplaced in w and $N(w)$ is the number of letters that w has that are not contained in the solution. Note that we do not directly learn this information by using the solution, but with the data that we have acquired in the previous attempts at guessing the solution.

Suppose the information we have about the solution is that its first letter is a `P`, that it contains an `R`, but not in the third position, and that it does not contain letters `A` or `K`. The algorithm will store this information as:

P	-	R	-	-

$gray = \{\texttt{A}, \texttt{K}\}$

The fitness values for `PREYS` and `SPACE` is:

$$fitness(\texttt{PREYS}) = \mathcal{H}(X_{\texttt{PREYS}}) + 5 \cdot 1 + 2 \cdot 1 = 5.10480 + 7 = 12.10480$$

$$fitness(\texttt{SPACE}) = \mathcal{H}(X_{\texttt{SPACE}}) - 2 \cdot 1 = 5.29281 - 2 = 3.29281$$

The code in Figure 3.4 shows the implementation of the fitness function. The `pattern` variable stores the ternary representation of the colors obtained for each of the letters of the previous attempt, while `guessed` stores the actual letters that have been guessed as green or amber. For example, if `pattern[i] = 2` and `guessed[i] = 'b'`, then the solution has a letter `B` in the i-th position. The letters that have been marked as incorrect are stored in `gray`. After this, the function computes the fitness value of a `w` word as we explained before. If the algorithm has no information about the solution, then its fitness will just be the entropy of the word.

```
def fitness(w, guessed, pattern, gray):
    f = get_entropy(get_word_position(w))
    if guessed != '-----': # No information
        for i in range(5):
            if pattern[i] == 2 and guessed[i] == w[i]:
                f += 5
            elif pattern[i] == 1 and guessed[i] != w[i] and guessed[i] in w:
                f += 2
            if w[i] in gray:
                f -= 2
    return f
```

Figure 3.4: Fitness function for the initial Genetic Algorithm

We mentioned that we would only attempt to guess the solution once every generation. Let us discuss how we implemented this step. The usual procedure in a Genetic Algorithm is to take two individuals from the population (selection operator) and combine them (crossover operator) to produce new individuals. This is the way the algorithm improves the population, by selecting solutions to generate offspring that could potentially be better fit than their parents. As we mentioned before, the way Wordle players gain information is by submitting a word and receiving feedback for the comparison of said word with the solution. At first glance, crossover seems difficult to implement because a vast majority of the times, combining the letters of two words will not result in an actual word –remember that Wordle only accepts real words. Our idea for initial tests was to define a selection operator in a way that it returned the 'best' individual of the population, which would be used as an attempt to guess the solution. The code for this operator is shown in Figure 3.5.

Using the word obtained in the selection operator, we get a color pattern that can be used in the crossover operator to generate new words that match the overall information that we have about the solution. The code of the function that is in charge of doing this can

```
def selection(population, guessed, pattern, gray):
    bestfit = fitness(population[0],guessed, pattern, gray)
    bestword = population[0]
    for i in range(1,len(population)):
        fit = fitness(population[i],guessed, pattern, gray)
        if fit >= bestfit:
            bestfit = fit
            bestword = population[i]
    return bestword
```

Figure 3.5: Selection operator for the initial Genetic Algorithm

be seen in Figure 3.6. Instead of giving crossover a probability of occurring, we decided that it should always take place. Figure 3.7 shows the implementation of the crossover operator.

```
def attempt(best, solution, guessed, colors, gray):
    newpattern = get_pattern(get_word_position(best),
   get_word_position(solution))
    for i in range(5):
        if newpattern[i] > colors[i]:
            colors[i] = newpattern[i]
            guessed[i] = best[i]
        if newpattern[i] == 0:
            gray.add(best[i])
    return guessed, colors, gray
```

Figure 3.6: Function that simulates an attempt of guessing the solution

Mutation remains mostly unchanged in how it works in regular Genetic Algorithms, as it can be implemented by simply changing a letter in a word, making sure that the result is an allowed word. We will have to handle the cases in which a word cannot mutate. For example, the word `IVORY` will not produce a new word with the mutation operator because there are no other words that match the patterns `-VORY`, `I-ORY`, `IV-RY`, `IVO-Y` or `IVOR-`. Figure 3.8 shows the code used for the mutation operator.

To show how the crossover and mutation operators work, Figure 3.9 is an example of offspring produced by this algorithm and examples of mutations of the word `BLACK`. Note that, for this specific example, crossover guarantees that the chromosomes have a `B` in the first position and an `A` in a position that is not the third one –since it was shown as amber. If any of these words include letters that we know to be gray, their fitness will be lower.

In our first experiments, the algorithm took 5 attempts in average to reach the solution.

```
def crossover(guessed, colors, attempts):
    found = False
    while not found:
        k = rnd.randint(0,len(words)-1)
        word = words[k]
        if not word in attempts:
            found = True
            for i in range(5):
                if colors[i] == 2:
                    found = found and word[i] == guessed[i]
                elif colors[i] == 1:
                    found = found and word[i] != guessed[i]
                    found = found and guessed[i] in word
    return word
```

Figure 3.7: Crossover operator for the initial Genetic Algorithm

```
def mutation(child, attempts, maxTries = 100):
    found = False
    tries = 0
    while not found and tries < maxTries:
        w = list(child)
        i = rnd.randint(0,4)
        w[i] = chr(rnd.randrange(ord('a'), ord('z')))
        new_word = ''.join(map(str,w))
        k = get_word_position(new_word)
        # Check if new word exists
        found = k > 0 and k < len(words) and words[k] == new_word
        found = found and new_word != child
        found = found and not new_word in attempts
        tries += 1
    if found:
        return words[k]
    else:
        return child
```

Figure 3.8: Crossover operator for the initial Genetic Algorithm

Words with high entropy would be reached quicker, within 2 or 3 tries, but some words took more than 15 attempts. This is because many of the operations that take place in Genetic Algorithms depend on random decisions and the algorithm could be unlucky in obtaining new better words. We observed that the number of attempts used in the algorithm highly

B	-	A	-	-
B	U	L	L	A
B	A	N	J	O
B	A	R	D	S
B	E	L	A	R
B	A	C	K	S
B	A	L	K	Y
B	I	V	I	A

B	L	A	C	K
B	L	O	C	K
F	L	A	C	K
S	L	A	C	K
B	L	A	N	K
C	L	A	C	K

Figure 3.9: Example of results given by the crossover and mutation operators

depends on the initial population and the first choices. If the initial words are not similar to the solution, the algorithm might have to use 2 or 3 attempts to finally learn something about the solution. On the contrary, if the algorithm starts with a guess that is very similar to the solution, it sometimes reaches the solution very quickly. For example, while attempting to reach the solution `MERRY`, we obtained the following sequence of words:

M	E	R	R	Y
O	R	L	E	S
M	A	I	R	E
M	E	R	R	Y

By learning that the letters `E` and `R` were misplaced in `ORLES`, the word `MAIRE` was randomly generated. Immediately after, `MERRY` was guessed. This example shows how dependent the algorithm is on chance, as simulations for the word `MERRY` gave an average number of attempts greater than 5. An example of a very bad guess from this algorithm was found for the word `FIZZY`, which took 16 tries. In particular, the algorithm got the sequence `CASTE`, `BARON`, `KLIEG`, `PICUL`, `MIKED`, `NIFTY`, `JIFFY`, `WITHY`, `WINEY`, `SIXTY`, `DIKEY`, `TIZZY`, `DIZZY`, `MIZZY`, `BIZZY` and, finally, `FIZZY`. Not only can we see that the algorithm tried many words ending in `IZZY` but it also took many tries until it finally learnt any information about the solution.

Although Genetic Algorithms usually take a little bit of time to find solutions, we decided to look for an improvement in the efficiency of this first approach as we felt that some simulations took more time that they should. This is because of the way some of the operators were defined. Take the crossover operator, for example. We take a random word from the list of allowed words and check if it respects the information that we have already gathered. If we are lucky then the algorithm might take a word that does fulfill all the current requirements for the solution but the it might have to repeat this process of checking the requirements for a lot of words until it finally finds a valid one. This complicated simulations

and making comparisons to the previous algorithms because sometimes a single simulation for a word took around 1 minute and it took more than an hour to make a simulation for all the possible solutions. Results obtained with this initial algorithm are shown in Chapter 5.

The huge gap in execution time that this algorithm presented compared to the other ones that we had studied made us reconsider the decisions that we had taken for this algorithm and so for a better iteration of a Genetic Algorithm we decided to take a different approach. Instead of trying to build a Genetic Algorithm from scratch, we would use the information and concepts used in the Information Theory Algorithm presented in Section 2.2, which we already observed provides good results, to adapt them into the structure of a Genetic Algorithm and see if it could be improved with that. The next section shows the new algorithm that we designed.

3.2.2 Genetic Algorithm with Information Theory

As we explained in the previous section, our goal for this final algorithm was to use all the information that we has gathered in the Information Theory Algorithm described in Section 2.2 and adapt it into the structure of a Genetic Algorithm to see if its performance improves with it. In this section we will provide the details of how the algorithm was defined, explained each of the operation used in the final Genetic Algorithm.

- Defining the representation of a solution.
  For this algorithm, we decided to treat the solutions as integers, which represent a word from the allowed list. That is, the 'solution' `i` represents the i-th word from the allowed list. We decided to do this because for many operations, we do not need to check the letters of the actual word like we did in our first Genetic Algorithm. Every time we need information relative to a word we can easily access it in the precomputed data using the integer that represents the word.

- Defining the fitness function.
  As we observed that entropy gives positive results for choosing which words to use for attempts in Wordle, we decided to define the fitness value of a word as the product of its entropy and its frequency, which are both precomputed. The use of the frequency of words is due to the improvement that it showed for the Information Theory Algorithm.

- Implementing the selection operator.
  This time, we decided to implement the selection operator with a biased roulette wheel. We thought this should help in situations where the algorithm has to choose among very similar words. If we recall the example with the word `MERRY`, the initial algorithms would choose uncommon words like `DERRY` or `HERRY` before guessing the right word. Using the biased roulette wheel, as `MERRY` is a more usual word, it would have a higher

chance of being chosen as an attempt to guess the solution in the game. The code that we made for this implementation can be seen in Figure 3.10.

```
def selection(population):
    totalFitness = 0
    individuals = []
    for w in population:
        f = fitness(w)
        totalFitness += f
        individuals.append([w,f])
    probs = np.empty(len(individuals))
    probs[0] = individuals[0][1]/totalFitness
    for i in range(1,len(individuals)):
        probs[i] = probs[i-1] + individuals[i][1]/totalFitness
    r = rnd.uniform(0,1)
    p = np.searchsorted(probs,r)
    return individuals[p][0]
```

Figure 3.10: Selection operator for the Genetic Information Theory Algorithm

- Implementing the crossover operator.
  This operator will stay similar to how it was in our first Genetic Algorithm. The change that we added is that once we get the pattern for the word obtained in selection and before executing crossover, we reduce the space of possible solutions to keep the words that match the pattern that we received, just like the Information Theory algorithm does. At that point, the crossover operator gets a random word from the list of words that remain after the reduction.

- Implementing the mutation operator.
  Mutation functions like it did in our first Genetic Algorithm. Given a word, it randomly changes one of its letters, making sure the result is a word accepted in Wordle.

The final algorithm that we implemented follows the structure show in Algorithm 2. Next we show some examples of good executions of the algorithm to guess the word `MERRY`.

T	A	P	A	S
L	O	U	N	D
M	I	C	H	E
M	E	R	R	Y

F	L	E	X	O
M	E	S	I	C
M	E	R	R	Y

C	H	A	R	K
S	W	O	R	N
M	I	T	R	E
M	E	R	R	Y

We can appreciate, like in our first Genetic Algorithm, that although some similarities can be observed in the obtained solutions, the algorithm could be very lucky and guess the solution in very few steps and without having a lot of information about the solution.

Data: $N > 0$, $mutationProb \in [0, 1]$
Result: List of attempts made until the solutions is found
$population \leftarrow createFirstGeneration(N)$
$remainingWords \leftarrow allowedWords$
$attempts \leftarrow empty$
while not $solutionFound$ **do**
  $best \leftarrow selection(population)$
  $pattern \leftarrow$ color pattern with the solution
  $add(attempts, best)$
  $solutionFound \leftarrow best = solution$
  update remainingWords
  $nextGeneration \leftarrow empty$
  for $i = 0$ **to** N **do**
    $child \leftarrow crossover(remainingWords)$
    $r \leftarrow randomValue(0, 1)$
    if $r < mutationProb$ **then**
      $child \leftarrow mutation(child)$
    end
    $add(nextGeneration, child)$
  end
  $population \leftarrow nextGeneration$
end
return $attempts$

Algorithm 2: Pseudocode for our Genetic Information Theory Algorithm

Unfortunately, it can also be very unlucky and encounter situations in which it tries, for example, the words `DERRY`, `HERRY`, `KERRY` or `TERRY` before it finally guesses the correct word.

In conclusion, we think that this algorithm represents an improvement with respect to the previous Genetic Algorithm. In the first experiments, this algorithm seemed to use less attempts to guess the solution and it was more uncommon to see situations in which it needs more than 10 tries to guess the word. In terms of time, it also has much shorter execution times and closer to the Information Theory Algorithm's execution times –simulations for all the possible solutions took around a minute. At this point, we can say that with this algorithm we have gained more evidence that entropy is a good metric in dealing with puzzles like Wordle. The results observed for the algorithm will be studied in more detail in Chapter 5, where we will present how this algorithm works depending on the values of its parameters and give some ideas on how this algorithm could be improved in future revisions.

Chapter 4

Individual contributions

This chapter presents a desription of the individual contributions made to the project by each of the members.

4.1 Contribution of Ángel Benítez Gómez

The first contact made with our supervisor goes back to June of 2021, after the list of available Final Degree Projects was published. An important factor that led us to choosing our project was the fact that you could combine Computer Science Engineering's and Mathematics's Final Degree Projects. In other words, the same topic could be developed, following different directions, for both faculties and, for that reason, there are similarities between this project and the one for the Degree in Mathematics. These similar features especially relate to the general introduction to Information Theory that both projects have. It is important to note that during the entire elaboration of this project, there has been constant communication between the two members and it would be difficult to say that a particular part of the project was made by only one of them.

After the first meeting with our supervisor, we decided to start doing some research about notions of Information Theory and Quantum Computing, which were the initial topics that the project revolved around. Aside from some material given to us during the years of the degree, the main source that I used to study and research Information Theory was the book *Elements of Information Theory* [12], by T. M. Cover and J. A. Thomas. As for Quantum Computing, I made a general research of the key concepts and initially used IBM's Quantum Lab to get familiar with quantum circuits, which would have been needed for the project.

With the start of the second semester, the topic of the project for Computer Science Engineering changed slightly, leaving out the part related to Quantum Computing and

entirely focusing on Information Theory. This is the moment when Enrique and I started to organize and plan the steps that we would follow and settled on the idea that we would try to communicate with each other as much as possible, so that everything done in this project had the insight from both members. The first thing we wanted to do was to determine how the introduction to Information Theory would be made in this project, from which I gathered more information about the applications that entropy has over other scientific fields, including Deep Learning and Data Compression. The meeting with our supervisor that led to the change of the topic of the project helped us come up with the idea of using some sort of Genetic Algorithm combined with concepts and ideas from Information Theory to solve Wordle. For that reason, I started making a more thorough research of Genetic Algorithms. We had seen an introduction to them in the subject of Artificial Intelligence that is taught in the degree. Using the material from this subject and more specific books gave me a better understanding of Genetic Algorithms and I came up with several ideas of how they could be adapted to solve Wordle. The main books that I used to gather information were *An introduction to genetic algorithms* [18], by M. Mitchell, and *Genetic Algorithms in Search, Optimisation and Machine Learning* [19], by D. E. Goldberg, which also helped me elaborate the examples related to Genetic Algorithms that are presented in this project.

At this point, we started to code our first versions of the algorithms. We continuously talked about the ideas and possible implementations of the algorithms and, as I had made a slightly bigger effort in researching Genetic Algorithms, we decided that I should be in charge of programming the Genetic Algorithm version of Wordle's solver. Once the first versions of the algorithms were finished, we gave each other insight about possible improvements and future directions for them. Because the Information Theory Algorithm gave better results than the Genetic Algorithm, we decided to make an algorithm that would adapt all the ideas from the Information Theory Algorithm into a genetic structure. Once again, we decided I should be in charge of that and so I had more impact over the Genetic Information Theory Algorithm. Once the algorithms were completed and after the various meetings with our supervisor, we decided that our next step would be to evaluate how the algorithms worked under certain conditions or relaxations. Following the general structure that we had used to divide our tasks, my workmate focused on studying the final versions and results for the Information Theory Algorithm and I centered my attention on the corresponding work for the Genetic Algorithms.

The final goal that we had with our project was to compare it to other algorithms that people had made to solve Wordle. Although we had already observed how some of the already existing algorithms worked, we both made some more research and shared our findings in a joint session, which served to write the final results section together.

In regard to writing this report, the organization that we followed ties with how we worked in the rest of the stages of the project. We did not 'assign' particular sections to any of the members, but instead we freely decided to write about the advances that we had made as they were happening. After the final topic of the project was changed, we made some brainstorming about how each of us pictured the project in their minds and the potential investigations that we considered interesting. This discussion led to a 'final' structure of the project that we used for this report. The fact that each of us programmed different algorithms made it so that each would have more control over the sections that involved them. That means that Sections 3.2 and 5.2 were initially developed entirely by me and were later revised by Enrique. As I mentioned, the rest of the sections were made collaboratively, so that I revised what Enrique wrote and he did the same with the parts that I wrote, making it so that the final text has been revised and accepted by both members.

To conclude, I would like to give special thanks to my workmate Enrique as I consider his work has had great impact on this project and his involvement was excellent. I would also like to thank our supervisor for his help and support during the entire development of this project. Some of the ideas that we thought about were unfortunately unable to be done in the time that we had to make this project and, hopefully, they can be worked upon in the future.

4.2 Contribution of Enrique Cavanillas Puga

In June 2021, one of the Final Degree Project titles that we were offered at UCM piqued my interest: *Squeeziness in quantum computers*. This project seemed to connect two very separate areas of science, both of which I would have loved to study in more detail throughout the course of the Double Degree. After researching more about the topics mentioned in the project's description and an exchange of emails with our supervisor, it was clear to me that this assignment had a lot of potential.

Ángel and I have a very fruitful record of working together on assignments throughout the degree, so the decision to work on this jointly was natural when he also showed interest in the project. On my part, after meeting with the project's supervisor and setting an initial course for the project, I started studying Information Theory and Quantum Computing through multiple sources, and often discussed my findings and doubts with Ángel and our supervisor.

Information Theory was not exactly new to me, entropy was a concept that we learnt in Artificial Intelligence I and II, but it was clear that we had barely scratched the surface of what it had to offer. To properly prepare for the project, I studied T.M. Cover and J.A. Thomas's introduction to Information Theory [12], and subsequently carefully examined

previous research that related Squeeziness with Software Testing [6, 9, 11]. At the same time, I familiarized myself with Quantum Computing; I studied the basics mainly through other Final Degree Projects such as L. A. Galindo and J. Pellejero's *Computación cuántica: pruebas de mutación* [21], and tinkered generously with IBM's Quantum Lab, Q# and Qiskit.

It was after reading an extensive collection of papers on Information Theory that I realized that very little effort had been made by the authors to carefully explain the concept of *information*, most likely because it came naturally to experienced researchers and the extent of their papers is usually limited. After reading Shannon's original article on entropy, *A Mathematical Theory of Communication* [3], I decided to thoroughly explain his method for designing the formula for entropy, and determined to use many examples to explain the concept before moving on to Squeeziness and Quantum Computing.

While designing an example of the applications of Information Theory, I tried to connect entropy with Wordle, and started to code what would eventually become the first iteration of the Information Theory Algorithm. I met with Ángel and Dr. Núñez and commented that this example of the practicality of entropy could be a worthwhile experiment, and eventually we decided to pursue the goal of solving Wordle through Information Theory. Ángel and Dr. Núñez had the idea of designing a Genetic Algorithm that used Information Theory to solve some of the shortcomings of the original algorithm. At the same time, there were some obstacles that had to be overcome before our algorithms could be extended to simulate puzzles, mainly the execution times of the entropy calculations and restructuring of the sample space after every guess made by the player.

At this point, Ángel started to work on the structure of the Genetic Algorithm and I undertook the task of fixing the code's slow execution. After restructuring some parts of the code, some `for` loops where still giving us slow execution speeds, so I investigated other attempts to solve Wordle. Grant Sanderson [17] mentioned the use of vectorization in the calculation of the patterns, so I did some research on SIMD instructions on Python, mainly by reading the documentation on the Numpy module [22]. Once Ángel and I had set the foundation of both algorithms, we started working together again by giving each other pointers on how to improve the code, the scores, and the execution times. By working together and experiencing the differences in the answers given by both of our algorithms, we gained insight into how to improve all of our work. Since the Information Theory Algorithm was ahead in terms of average scores, it allowed me to tinker with its parameters and make many modifications in search of even better scores, like introducing word frequencies when looking for the next guesses.

Our open communication and the freedom we had when working separately allowed us to produce better results after every meeting, and eventually we set to find a practical lower

bound for the average scores of our algorithms. I came up with some relaxations of the original problem that could perhaps point us in the right direction and modified our test bench accordingly.

When it came to writing the report, we both took the same amount of responsibility and distributed the sections among ourselves with no particular structure. Since I was more involved with the Information Theory Algorithm, I took on the task of writing the draft for Sections 2.2 and 5.1, and took the responsibility of revising and editing Ángel's work as well. The rest of the report was actively written and edited by both members equally.

Ángel and our supervisor's involvement in the project was invaluable; Ángel had excellent ideas and the resolve to always see them through. He also did not complain at all any time I pointed out our algorithms had to be completely redesigned to abide by the rules or to shave a couple of seconds of execution time. Working with Manuel Núñez allowed us to see the inside of a researcher's mind and he always had relevant suggestions and pointers for each of my qualms.

Chapter 5

Results and Conclusions

To conclude this project, this final chapter describes the results that we obtained for our different algorithms in full detail. We also compare these results with other algorithms that have also been designed to solve Wordle. Finally, we present some thoughts and reflections on our findings.

5.1 Information Theory Algorithm

5.1.1 Naïve Algorithm

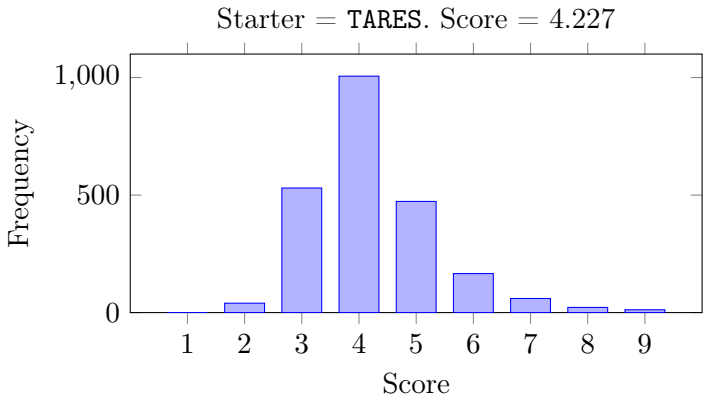


Figure 5.1: Simulation results of the Naïve Algorithm

With the initial naïve strategy, we achieved an average score of 4.227 attempts after simulating games for each of the 2309 possible solutions using `TARES` as starter. As shown in Figure 5.1, we had that 4.07% of the solutions could not be reached within 6 tries and

required 7 or more attempts. These results are not outstanding. In fact, the average score achieved by this first algorithm is on par with the average score for human players. As we mentioned in Section 2.2.3, one of the most likely reasons for this is that we are assuming a uniform distribution over the set of allowed words, which is a mistake. Since we know that solutions will always be common words, the algorithm should also use as many common words as possible when attempting to reach the solution. For example, when the solution is `BLESS`, the following sequence of attempts is made by the algorithm:

B	L	E	S	S
T	A	R	E	S
D	E	I	L	S
B	L	E	Y	S
B	L	E	B	S
B	L	E	S	S

The algorithm does not discern between common and uncommon words, which seems to be very useful until a lot of information has been gained. Once the set of compatible words is very small, choices like `BLEBS` are made over `BLESS`, leading to unnecessary attempts.

The most notable example of this problem is the solution `MERRY`, which takes 9 attempts to be found and points to another additional problem of this naïve approach. The following sequence is obtained: `TARES, RORIE, BERRY, DERRY, FERRY, HERRY, JERRY, KERRY, MERRY`. Not only is the Naïve Algorithm favoring extremely uncommon words like `DERRY` and `HERRY`, but the sample space is so restricted that it just tests every word ending in `-ERRY` in aplhabetical order.

5.1.2 Greedy Algorithm

In order to solve this problem, we decided to implement the Greedy Algorithm that considers the relative frequency of the words, so that even if we are very restricted to a handful of words with similar entropy, we always check the most common ones first. After incorporating word frequencies, the average scores saw a significant improvement, and the best starters also changed. Now, the highest ranked starter is `RATES`, which is almost identical in structure to `TARES`, but is a much more common word. The average score for `RATES` using this algorithm is 3.843, and less than 1% of the solutions required more than 6 attempts.

During the testing stage, we also found that some starters with lower entropy and lower g-priority score can achieve a better score than higher ranked words. The best score was obtained by starting with `TALES`, with 3.773 attempts and even a lower percentage of failed attempts than `RATES` despite having less entropy and a sixth of its relative frequency. Thus, we present `TALES` as our candidate for best opener.

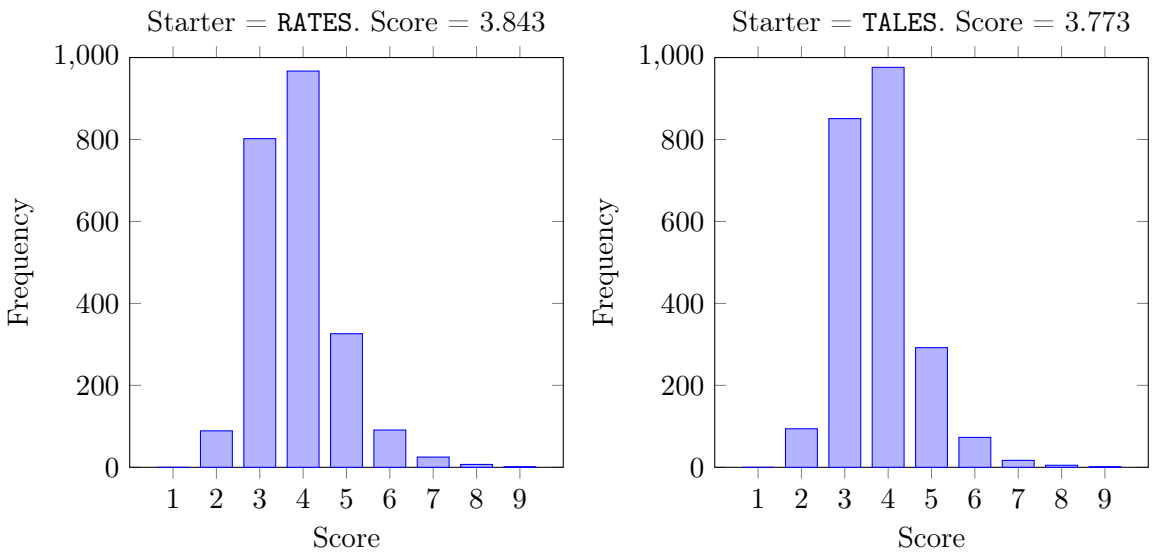


Figure 5.2: RATES vs TALES

5.1.3 Relaxations of the problem

As we mentioned in Section 2.2.5, we designed several relaxations of the original problem P_0. These allowed us to estimate a lower bound for the scores of our algorithms, and gave us insight into how they could be improved in future investigations.

P_1	Find the solution to any Wordle puzzle in as few attempts as possible using *only words from the list of solutions.*
P_2	Find the solution to *all of Wordle's daily puzzles* in as few attempts as possible using the words from the list of allowed words.
P_3	Find the solution to *all of Wordle's daily puzzles* in as few attempts as possible using *only words from the list of solutions.*

Problem P_1 obtained a score as low as 3.527 after opening with SLATE and using the Naïve Algorithm, only missing 7 solutions. Remember that SLATE is the 56th best naïve opener when every word is considered. When restricting the set of allowed words to just the solutions, however, SLATE is the second ranked opener, slightly behind RAISE. P_1 made us realize that, while using the frequency of words improved the scores significantly, our function g could never be adjusted to reach such good scores, because the solution set had been handpicked. For example, SANER is 100 times less frequent than TALES –our candidate for best starter– and yet TALES is not in the solution set, and SANER is. This human factor is a limiting factor that should be considered in further investigations.

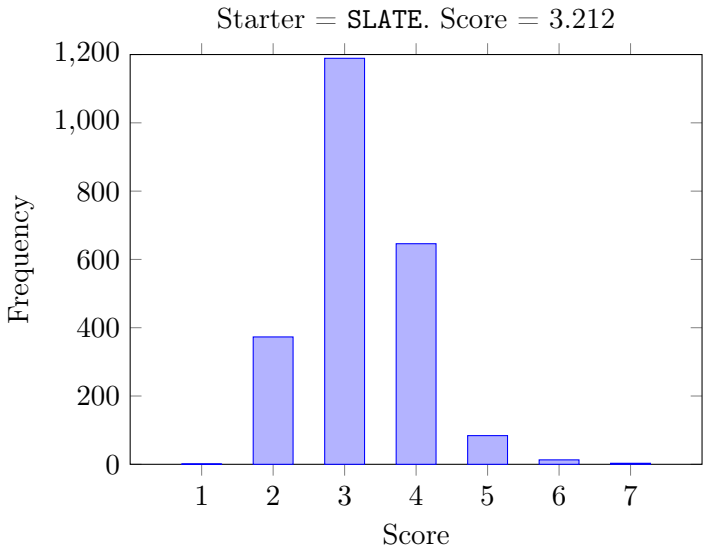


Figure 5.3: Simulation result of the Naïve Algorithm for P_3

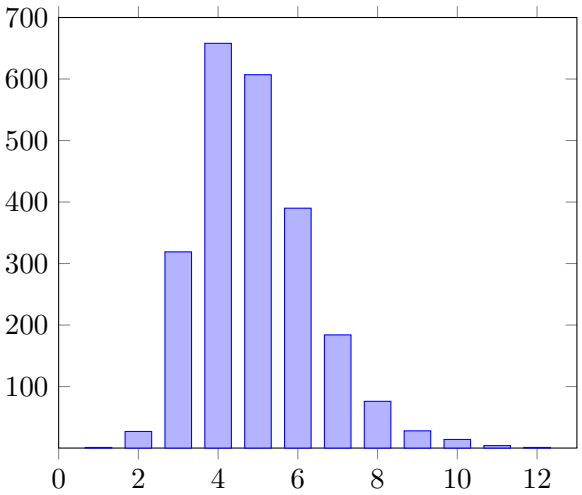
Problem P_2 obtained an average score of 3.535 for multiple random orderings of the solutions when opening with `TALES` using the Greedy Algorithm, and missed only a couple of solutions on average. If we decided to make a 'daily' Wordle bot, we would certainly implement it this way, since it never reuses solutions and is not cheating by knowing exactly which words could be solutions. The only caveat is that the score obtained is slightly dependent on the ordering of the solutions, but in our testing the differences were consistently less than 1% of the score.

Finally, the last relaxation of the original problem, P_3, obtained the lowest average score of 3.225, and usually reached most of the solutions using `SLATE` as starter and the Greedy Algorithm. Our greedy algorithm would be the perfect Wordle cheat for competitors who are looking to boast their scores without having to expense the effort of solving the puzzles. Figure 5.3 is of the bar plot of one of the simulations of P_3.

We consider this score a lower bound of what can be achieved through 'fair' competition for P_0, the original problem. This lower bound seems to be consistent with other investigations by Grant Sanderson [17], Jonathan Olson [23] and Alex Selby [24]. Sanderson goes as far as to compute the best starter and best follow up through a brute force search and concludes that 1.16 bits of uncertainty remain in the puzzle and, therefore, $2^{1.16} = 1.12$ more attempts need to be made, which leaves us with a lower bound of 3.12 for the average score.

5.2 Genetic Information Theory Algorithm

As we stated in Section 3.2.1, testing our first Genetic Algorithm did not seem very interesting because we could see that it had a worse performance, both in execution time and average tries, than the Information Theory Algorithm. For that reason, we made few simulations with it and focused more on the Genetic Information Theory Algorithm. Figure 5.4 shows an example of results obtained from simulations made for our first Genetic Algorithm. An important highlight from this algorithm is how volatile it is. That is, although several simulations gave a similar average number of tries, simulating words individually several times gave very different results. In Section 3.2.1 we talked about how having the word `FIZZY` as the solution led to the algorithm guessing it in 16 tries, but other executions had the algorithm guess it in just 3 tries. It is also important to note that the simulations that we made for this algorithm always showed that more than 12% of the words could not be guessed with 6 or less tries.



Population size	50
Mutation Probability	0.1
Average number of tries	4.921
Maximum number of tries	12
Failed words	13.29%

Figure 5.4: Simulations results for an execution of our first Genetic Algorithm

For the Genetic Information Theory Algorithm, we first decided to study the results that we would obtain using different values for its parameters, the population size and the probability of mutation. Starting with the size of the population, we decided to run several simulations to determine which sizes lead to better results. We measured the average number of tries that the algorithm takes to guess every word, the maximum number of tries that it needed to guess a word during the simulation and the percentage of words that the algorithm could not guess with 6 or less tries. Note that, being a Genetic Algorithm, different executions would lead to different results because of the random choices made within the algorithm. However, the results were consistent and Figure 5.5 shows what we obtained for

some of these simulations, where the probability of mutation was always set to 0.1.

Population size	10	25	50	100	250	500
Average number of tries	4.737	4.764	4.718	4.707	4.699	4.722
Maximum number of tries	11	12	12	12	11	11
Failed words	8.7%	8.7%	9.2%	7.5%	8.5%	8.8%

Figure 5.5: Results of experiments with different population sizes

Although the results are very similar for the different values of the size of the population, we can see a slightly better performance of the algorithm in the range between 100 and 250. Figure 5.6 shows graphs for different executions using 100 as the population size. All these simulations with different population sizes showed us that the algorithm usually takes between 4 and 5 tires to guess a word, using up to 12 in the worst cases. More than 90% of the times, the algorithm guesses the word in 6 or less attempts.

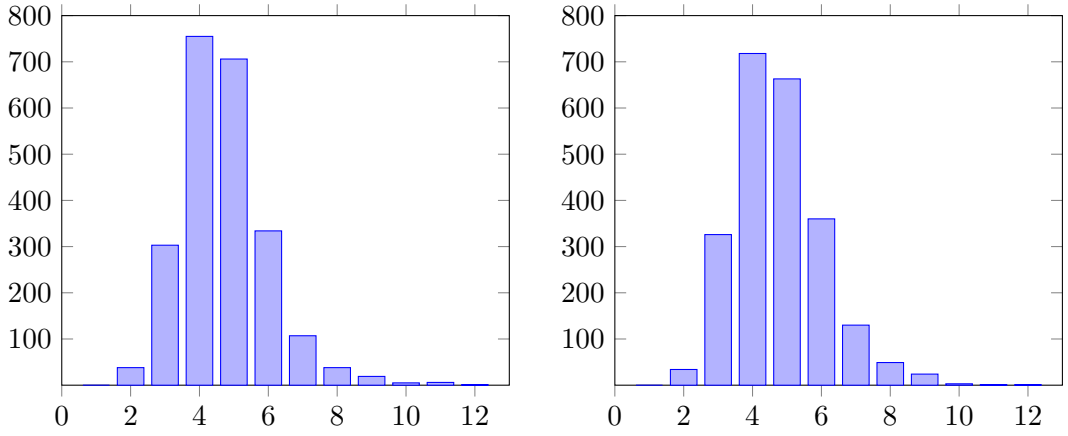
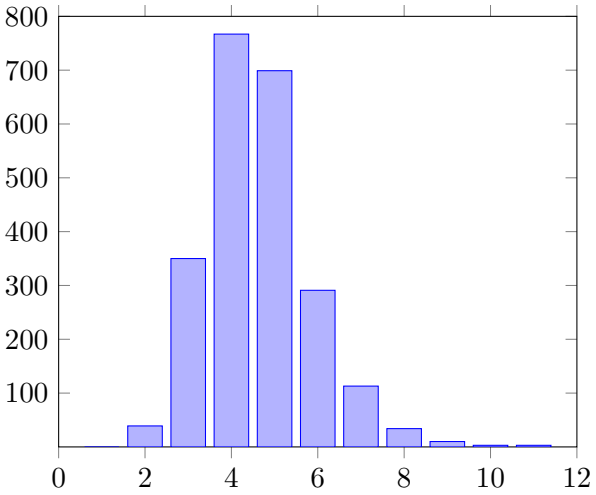


Figure 5.6: Frequencies of the different scores obtained using a population size of `N = 100`

Our next step was to analyze the impact of mutation on the algorithm. For that, we set the population size to 100 and tried executing the algorithm with different values for the probability of mutation. As we made the probability higher, we could tell a tendency of the algorithm getting worse results. This made us think, that mutation might be counterproductive for this algorithm as very high values for the probability led to executions in which more than 15 tries were done to guess a word. Figure 5.7 shows the results we obtained for this analysis. We studied how the algorithm works if we did not consider mutation and the results were better than any that we had gathered. With no mutation involved in the algorithm, the average number of tries dropped to around 4.6, which, although not by much, is lower than what previous simulations showed. Figure 5.8 shows the best execution of the algorithm that we obtained after giving mutation a null probability.

Mutation Probability	0.1	0.2	0.3	0.4
Average number of tries	4.760	4.852	4.951	5.086
Maximum number of tries	12	12	14	15
Failed words	8.3%	10.82%	13.16%	15.5%

Figure 5.7: Results of experiments with different mutation probabilities

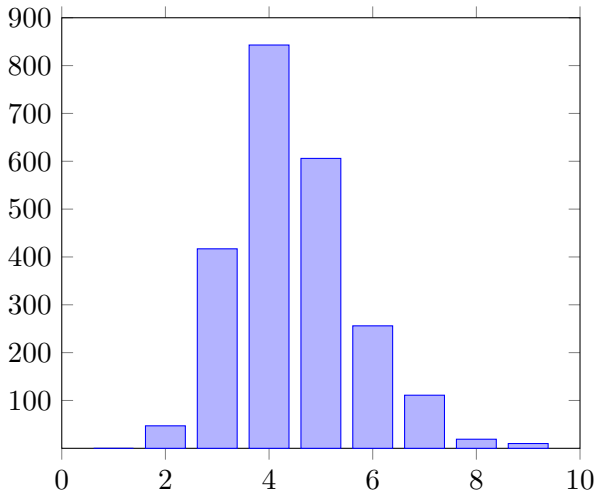


Population size	100
Average number of tries	4.614
Maximum number of tries	11
Failed words	7.01%

Figure 5.8: Best execution obtained using null mutation probability

To summarize the conclusions that we obtained from these simulations, the best population size that we found was `N = 100` and using a mutation probability equal to zero led to the best results. Following these simulations, we tried to incorporate ideas from the Information Theory Algorithm to try to improve the performance of the genetic version. We though about how important it is to choose a good starting word when you play Wordle. However, the Genetic Algorithm starts with a random population, which means that you might get very good words from the beginning or that the words randomly generated might be very bad. For that reason, we thought that initializing the population with the best words could be of great help for the algorithm. That is, instead of randomly generating `N` words, the initial population would be composed of the `N` words with the most entropy. Using that, combined with the previous parameters that showed better results, we consistently obtained better executions of the algorithm. Figure 5.9 shows the best execution that we obtained during our process of testing the algorithm.

The results that this algorithm obtained were encouraging, especially with how we were able to find configurations and tiny improvements that provided better results. When it comes to execution time, we were able close the gap between the Genetic and Information



Population size	100
Average number of tries	4.463
Maximum number of tries	9
Failed words	6.06%

Figure 5.9: Best execution obtained for the Genetic Information Theory Algorithm

Theory algorithms, as well as in precision –the percentage of words that the algorithms guess with 6 or less tries– were more similar too. The main goal now would be to reduce the average number of tries, which is what makes up for the biggest difference between both algorithms and makes this one a worse option. As we could not focus a lot on that part of this algorithm, future work could revolve around upgrading the fitness function. That is, defining different fitness functions that provide better results for the selection operator.

Word	Entropy	Priority	Fitness Value
`BERRY`	4.143	0.841	3.484
`DERRY`	4.319	0.779	3.367
`FERRY`	3.970	0.843	3.349
`JERRY`	3.683	0.855	3.149
`KERRY`	4.098	0.834	3.421
`MERRY`	4.233	0.833	3.526
`PERRY`	4.248	0.853	3.624
`TERRY`	4.480	0.853	3.825

Figure 5.10: Fitness values for the words that end in `-ERRY`.

In some situations, the fitness values of several words were not 'different enough' to make common words stand out from other less common ones. The values from Figure 5.10 show the entropy and word priority of several words ending in `-ERRY`. Since we know that the only possible solutions are `BERRY`, `FERRY` and `MERRY` – these are as well the most common words in the set – we want the fitness function to give them a considerably better fitness

value. However our fitness function gives them all very similar fitness scores, which would mean that the selection operator is going to choose among them without almost the same probability. Future investigations could also revolve around the initial population, which is something that we could tell made an improvement in this algorithm.

5.3 Comparison to other algorithms and Conclusions

The competition to make the best Wordle solver is fierce, although the goals vary among researchers. Jonathan Olson [23], for example, designed decision trees coupled with heuristic metrics in the search of an algorithm that can find the solution to all words in less than 4 attempts. Currently, his research supports the idea that it is computationally verifiable that this is impossible to accomplish and his algorithms are short by 18 words, or 0.78% of the solutions.

Many researchers use brute force methods to explore the entire decision tree, and use intelligent pruning methods to come up with good strategies. Tyler Glaiel's [25] algorithm uses metrics that reminded us of our first Genetic Algorithm (see Section 3.2.1) because it scores possible guesses based on the colours of their letters and it originally achieved similar average scores to ours.

The most comprehensive and formal research was done by Alex Selby [24]. His work includes his list of the best 100 starter words, after using strategies similar to the famous *adversarial game* with the only drawback that his code takes very long to compute, in his own words, when approaching his best scores. Selby uses a very smart method of producing upper bounds for the scores of each word in a very short amount of time, after which he re-evaluates his candidates ensuring exact answers.

Grant Sanderson's algorithm is very similar to ours in the sense that his team also uses Information Theory metrics in their research [17]. Sanderson's work includes a 2-step calculation where words are ranked according to the conditional entropy of the best follow-up guess as well as their own entropy score. Sanderson's investigation is very similar to ours and we were delighted to find that his best naïve starters matched ours in Figure 2.7. After reviewing Sanderson's code, we found he had designed a very astute way to compute the patterns, and we decided to overhaul our pattern constructing algorithm to integrate SIMD instructions to speed up our code.

In general, most research seems to halt when the average scores reach 3.43. As we mentioned in Section 5.1, we believe there is a practical lower bound when closing in on a 3.2 average score, and lower scores necessarily must include some form of *cheating*. The problem with trying to break the 3.42 barrier is that execution times reach a tipping point of diminishing returns, with algorithms taking two days in Alex Selby's case.

Researcher	Best straters	Average Best Score
Grant Sanderson	SALET, CRATE, TRACE	3.43
Alex Selby	SALET, CRATE, REAST	3.42
Jonathan Olson	SALET, RANCE, REAST	3.42
Tyler Glaiel	SOARE, ROATE, RAISE	3.43

Figure 5.11: Caption

In our research, the words in Figure 5.11 were excellent starters and their structures seem similar to our initial and final candidates, TARES and TALES. We were pleased to find that our candidates are close to the top in the lists of best starters of the other researchers, even if they were not the top candidates.

In the last stages of our project we had already begun to experiment with other metrics and obtained positive results. We feel encouraged to continue revising and improving the project in the future and already have some ideas to potentially enhance our algorithms. In particular, we believe that implementing multiple-step entropy calculations could improve our algorithms significantly, at the cost of some execution time. With regard to our Genetic Algorithms, we feel that an effort can be made to improve their performance through the definition of more elaborate fitness functions.

Overall, we had a very positive experience researching and developing the strategies we used in our investigation, and look forward to other opportunities to use our mathematical background to solve stimulating puzzles that at first glance have no relationship with computer science or mathematics. We are highly grateful to our supervisor, Manuel Núñez, for his support in the development of the project and for the positive atmosphere we were all able to participate in.

Bibliography

[1] H. Nyquist. "Certain Factors Affecting Telegraph Speed". In: *The Bell System Technical Journal* 3.2 (1924), pp. 324–346.

[2] R. Hartley. "Transmission of information". In: *The Bell System Technical Journal* 7.3 (1927), 535–563.

[3] C. E. Shannon. "A Mathematical Theory of Communication". In: *The Bell System Technical Journal* 27 (1948), pp. 379–423, 623–656.

[4] D. A. Huffman. "A Method for the Construction of Minimum Redundancy Codes". In: *Proceedings of the Institute of Radio Engineers* 40.9 (1952), pp. 1098–1101.

[5] J. R. Quinlan. "Induction of decision trees". In: *Machine Learning* 1.1 (1986), pp. 81–106.

[6] D. Clark and R. M. Hierons. "Squeeziness: An information theoretic measure for avoiding fault masking". In: *Information Processing Letters* 112.8-9 (2012), pp. 335–340.

[7] D. Clark, R. M. Hierons, and K. Patel. "Normalised Squeeziness and Failed Error Propagation". In: *Information Processing Letters* 149 (2019), pp. 6–9.

[8] K. Androutsopoulos et al. "An analysis of the relationship between conditional entropy and failed error propagation in software testing". In: *36th Int. Conf. on Software Engineering, ICSE'14*. ACM Press, 2014, pp. 573–583.

[9] M. Núñez A. Ibias and R. M. Hierons. "Using Squeeziness to test component-based systems defined as Finite State Machines". In: *Information & Software Technology* 112 (2019), pp. 132–147.

[10] A. Ibias and M. Núñez. "Estimating Fault Masking using Squeeziness based on Rényi's Entropy". In: *35th ACM Symposium on Applied Computing, SAC'20*. ACM Press, 2020, pp. 1936–1943.

[11] K. Patel, R. M. Hierons, and D. Clark. "An information theoretic notion of software testability". In: *Information & Software Technology* 143 (2022).

[12] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. 2nd. Wiley, 2006.

[13] S. Machkovech. *Wordle creator describes game's rise, says NYT sale was "a way to walk away"*. 2022. URL: https://arstechnica.com/gaming/2022/03/wordle-creator-describes-games-rise-says-nyt-sale-was-a-way-to-walk-away/.

[14] D. Victor. *Wordle is a love story*. 2022. URL: https://www.nytimes.com/2022/01/03/technology/wordle-word-game-creator.html.

[15] R. Hall. *Wordle creator overwhelmed by global success of hit puzzle*. 2022. URL: https://www.theguardian.com/games/2022/jan/11/wordle-creator-overwhelmed-by-global-success-of-hit-puzzle.

[16] S. O'Sullivan. *Jotto Game Review*. 2002. URL: http://www.panix.com/~sos/bc/jotto.html.

[17] G. Sanderson. *Solving Wordle using Information Theory*. 2022. URL: 3blue1brown.com.

[18] M. Mitchell. *An introduction to genetic algorithms*. MIT Press, 1998.

[19] D. E. Goldberg. *Genetic Algorithms in Search, Optimisation and Machine Learning*. Addison-Wesley, 1989.

[20] J. H. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, 1992.

[21] L. Aguirre Galindo and J. Pellejero Ortega. *Computación cuántica: pruebas de mutación*. Trabajo de Fin de Grado. Facultad de Informática. Universidad Complutense de Madrid. 2020.

[22] C. R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (2020), pp. 357–362.

[23] J. Olson. *Optimal Wordle Solutions*. 2022. URL: https://jonathanolson.net/experiments/optimal-wordle-solutions.

[24] A. Selby. *The best strategies for Wordle*. 2022. URL: https://sonorouschocolate.com/notes/index.php?title=The_best_strategies_for_Wordle.

[25] T. Glaiel. *The mathematically optimal first guess in Wordle*. 2022. URL: https://medium.com/@tglaiel/the-mathematically-optimal-first-guess-in-wordle-cbcb03c19b0a.