Universiteit Leiden
The Netherlands

Opleiding Informatica

Decoding the code-breaking game of Symble

Robin Staal

Supervisors:
Rudy van Vliet & Jan N van Rijn

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

28/08/2024

Abstract

This thesis explores the code-breaking game of Symble, a variation on the popular game of Wordle. Symble brings a new abstract layer to the feedback pattern that makes the game more complex for the codebreaker. In this thesis, we analyse its scaling configurations. We further conduct experiments to measure the efficiency of various standard algorithms, such as the Monte Carlo algorithm and Knuth's algorithm, which was made for the game of Mastermind. We find the impact of word length on the average number of guesses to only have an effect in small configurations, and we find alphabet size to have a $\log_3(k)$ relation with the average number of guesses.

Contents

1 Introduction

Symble is a code-breaking game created by Daniel Baamonde. Specifically, it is a code-breaking game inspired by the popular game of Wordle [Par99], which in turn is based on the popular game show Lingo [1]. Some people may be more familiar with the more old-school code-breaking game of Mastermind [2].

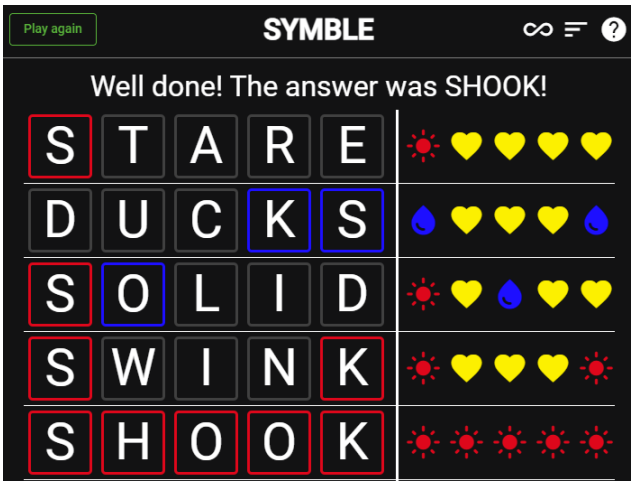

Figure 1: A game of Symble won by the codebreaker, with a red sun, yellow heart and blue raindrop used as *placeholder symbols*.

In the standard game of Symble, the aim is to uncover a random 5-letter English word within at most 8 guesses. The player, or *codebreaker*, can enter any 5-letter guess from the same dictionary used to generate the codeword, and will then receive a two-phase feedback from the *codemaster*. The result of this feedback can be seen in on the right-hand side in Figure 1, where three symbols are used to represent information, but what they represent exactly is unknown at the start of the game.

For this first phase of the feedback, the codemaster determines one of the following for each letter in the codeword:

- This letter is present in the guess and in the same position. (referred to as *green* from here on)
- This letter is present in the guess, but not in the same position. (referred to as *yellow* from here on)
- This letter is not in the word. (referred to as *gray* from here on)

We must note that the feedback of Symble has a slight deviation from that of Wordle, because here the generation of feedback for each letter is evaluated from the codeword respective to the guess instead of the other way around. This mainly affects the *yellow* information type, where with:

$$\text{codeword} = \text{solid}, \text{guess} = \text{thorn}, \text{Wordle marks the third position with yellow}$$

[1] Created by Ralph Andrews. Various international versions produced since 1987.
[2] https://en.wikipedia.org/wiki/Mastermind_(board_game)

codeword = solid, guess = thorn, Mastermind marks a single pin yellow, but does not specify its position

codeword = solid, guess = thorn, Symble marks the second position with yellow

The main spin Symble gives to Wordle is in the second phase of the feedback generation, where the feedback types are mapped to symbols using a bijective mapping that has also been generated by the codemaster. This creates a layer of abstraction where the codebreaker also has to figure out the feedback types these symbols are representing. This feedback pattern with symbols is shown in Figure 1.

It is important to note that there is an exception with the 'gray' information type. While this intuitively means that the letter does not appear in the guess provided by the codebreaker, we can more accurately define the gray information type by saying that, for the letter that was made gray, there is no occurrence in the guess provided by the codebreaker that has not been referenced to yet.

This exception can occur when the codeword contains a double letter, and the letter first iterated over then takes precedence in its feedback type generation. An example of this is:

codeword = hello, guess = lands

where the 'l' in the third position of 'hello' would be marked with yellow but the 'l' in the fourth position would be marked with gray.

After the codebreaker has received feedback on their guess they can then use this information to come up with a new 5-letter English word as a guess, to once again receive feedback in the same manner. The player is allowed to repeat guesses or submit a guess that is not consistent with the feedback given by the codemaster. If the codebreaker is able to work with the information provided in the feedback patterns to deduce the codeword and submit it as a guess before they are out of guesses, they win the game.

1.1 Related work

We mentioned Symble's similarities to Wordle and Mastermind, and while Symble has no history of scientific study, these older and more popular games have been studied extensively. We will now look at a few of these contributions to the field of code-breaking games to understand what is of interest in deciphering these games.

The most well-known study on Mastermind likely is the one by Donald E. Knuth [Knu76], who proposed an algorithm that solves the standard version of Mastermind in at most five guesses. This so-called Knuth's algorithm was further analysed to measure its performance on other configurations of Mastermind and to see how it performs against other algorithms [dG19]. Vivian van Ooijen measured genetic algorithms [Gol89] for Masterminid and found they scale well compared to other algorithms [vO18]. As mentioned, we will be contributing to the performance analysis of several algorithms, but now for instances of Symble.

In terms of computational complexity, Wordle has been proven to be NP-complete, by determining if a Wordle instance admits a winning strategy[Ros22]. Akin to this it has been shown that Mastermind is also NP-complete, specifically the Mastermind Satisfiability Problem, by Jeff Stuckman and Guo-Qiang Zhang [SZ05].

Lastly we see the computation of lower and upper bounds for code-breaking games and with Mastermind specifically. For example it has been proven that the codebreaker can find the codeword with just $O(n \log \log(n))$ guesses with $n^2 \log \log(n)$ colors [DDST13].

1.2 Thesis structure

In this thesis, we first introduce a more formal definition for an instance of Symble in Section 2, which we use to conduct our experiments. In Section 3 we define a set of standard algorithms, to then analyse their performance in Section 4.2 on different dictionaries for Symble. In Section 4.3 we will subsequently estimate a lower bound for varying configurations of Symble and then again conduct experiments in order to see how close a simple heuristic can get to the established bound.

2 Definitions and analysis

As mentioned, it is important we establish a formal definition for an instance of Symble, since this will be of great help when describing the experiments that we will conduct. We define our instance in a similar manner as the original Wordle game definition [Ros22].

A Symble instance is denoted by $\text{SY} = (\Sigma, n, D, S, g)$, where:

- Σ: A finite **alphabet** with $|\Sigma| = k$,
- n: **word length**,
- $D \subseteq \Sigma^n$: **dictionary** of words,
- $S = \{\alpha, \beta, \gamma\}$: placeholder **symbols** (can be any unique symbols as long as $|S| = 3$),
- g: maximum number of guesses

2.1 Feedback example

To show how all of this comes together in an actual game of Symble, consider an example.

At the start of the game the codemaster will choose a codeword $w \in D$, and a bijective mapping μ from feedback types to the set of symbols S. We look at an example of how the feedback generation plays out using the standard configuration of Symble with $n = 5$ and most 5-letter English words as dictionary D. The codemaster may choose the following:

$$\mu(\text{gray}) = \alpha,\ \mu(\text{yellow}) = \beta,\ \mu(\text{green}) = \gamma$$
$$w = \text{żebra}$$

The codebreaker can now enter a guess $x \in D$ to receive feedback from the codemaster $C_{\mu,w}$. Suppose the first guess from the codebreaker is $x =$ 'charm'. The feedback generation would then proceed as follows, where f_i or w_i denotes the ith position in the feedback pattern or codeword respectively:

$$f_1 = \text{gray, since } w_1 = \text{'z', which does not occur in } x$$
$$f_2 = \text{gray, since } w_2 = \text{'e', which does not occur in } x$$
$$f_3 = \text{gray, since } w_3 = \text{'b', which does not occur in } x$$
$$f_4 = \text{green, since } w_4 = \text{'r', which does occur in } x, \text{ and also in the 4th position}$$
$$f_5 = \text{yellow, since } w_5 = \text{'a', which does occur in } x, \text{ but not in the same position}$$

these feedback types are then mapped to symbols using bijective mapping μ:

$$\mu([\text{gray}, \text{gray}, \text{gray}, \text{green}, \text{yellow}]) = \alpha\alpha\alpha\gamma\beta$$

finally resulting in:

$$C_{\mu,w}(\text{charm}) = \alpha\alpha\alpha\gamma\beta$$

We can now look at how the information of this feedback differs from that of Wordle or Lingo due to the uncertain factor of the possible mappings μ. For example, the feedback pattern '$\alpha\alpha\alpha\gamma\beta$' could be interpreted differently due to the uncertainty of μ:

1) If $\mu(\text{gray}) = \alpha$, $\mu(\text{yellow}) = \beta$, $\mu(\text{green}) = \gamma$ and $w =$zebra,
then $C_{\mu,w}(\text{charm}) = \mu([\text{gray, gray, gray, green, yellow}]) = \alpha\alpha\alpha\gamma\beta$.

2) If $\mu(\text{gray}) = \alpha$, $\mu(\text{yellow}) = \gamma$, $\mu(\text{green}) = \beta$, and $w =$gleam,
then $C_{\mu,w}(\text{charm}) = \mu([\text{gray, gray, gray, yellow, green}]) = \alpha\alpha\alpha\gamma\beta$.

3) If $\mu(\text{gray}) = \gamma$, $\mu(\text{yellow}) = \beta$, $\mu(\text{green}) = \alpha$, and $w =$chair,
then $C_{\mu,w}(\text{charm}) = \mu([\text{green, green, green, gray, yellow}]) = \alpha\alpha\alpha\gamma\beta$.

4) If $\mu(\text{gray}) = \beta$, $\mu(\text{yellow}) = \gamma$, $\mu(\text{green}) = \alpha$, and $w =$champ,
then $C_{\mu,w}(\text{charm}) = \mu([\text{green, green, green, yellow, gray}]) = \alpha\alpha\alpha\gamma\beta$.

5) If $\mu(\text{gray}) = \gamma$, $\mu(\text{yellow}) = \alpha$, $\mu(\text{green}) = \beta$, and $w =$harem,
then $C_{\mu,w}(\text{charm}) = \mu([\text{yellow, yellow, yellow, gray, green}]) = \alpha\alpha\alpha\gamma\beta$.

6) If $\mu(\text{gray}) = \beta$, $\mu(\text{yellow}) = \alpha$, $\mu(\text{green}) = \gamma$, and $w =$macro,
then $C_{\mu,w}(\text{charm}) = \mu([\text{yellow, yellow, yellow, green, gray}]) = \alpha\alpha\alpha\gamma\beta$.

Here we see that six different 'feedback type' patterns result in equivalent 'symbol' patterns. Thus, the challenge for the codebreaker is twofold: they must deduce the codeword w while deducing the feedback mapping μ that creates the uncertainty in the structure of the codeword.

2.2 Set of valid codewords

With the feedback from the codemaster, the set of *valid codewords* V can decrease, meaning the codewords $w \in D$ that are still valid answers with regards to the feedback of the codemaster. Before the first guess we can intuitively see that $V = D$, but for future guesses this becomes more complex.

Given that the codemaster has selected a bijective mapping μ and a codeword $w \in D$, we define V as the set of codewords $w' \in D$ that could have returned the same feedback that the codemaster has up until current guess x. Or mathematically, we denote this as:

$$V = \{w' \in D \mid \forall x \in X \quad \exists \mu' : C_{\mu',w'}(x) = C_{\mu,w}(x)\}$$

with X representing the list of guesses submitted by the codebreaker thus far.

2.3 Number of unique feedback patterns

Since the information we get from the feedback of the codemaster is what we use to eventually find w, it is of interest to look at how many unique patterns the codemaster can return, since with more unique patterns the feedback becomes more informative. This number of unique feedback patterns is what we will use to calculate the lower bound in Section 4.3.1.

We know from our definition that a game of Symble contains exactly three unique symbols ($|S| = 3$) for all instances, which will be a relevant factor in calculating the number of unique feedback patterns. While this number of unique patterns f technically evaluates to $|S|^n = 3^n$ with all permutations of the three symbols, there is a caveat to this: some of these patterns are isomorphic.

We have seen an example of this in Section 2.1 above, where there were six different 'feedback type' patterns that resulted in the same 'symbol' pattern when different bijective mappings μ were applied. Because of the variety in μ we end up with equivalence classes of size six for the first guess. Such an equivalence class may look like this:

$$[\alpha\alpha\alpha\gamma\beta, \alpha\alpha\alpha\beta\gamma, \beta\beta\beta\gamma\alpha, \beta\beta\beta\alpha\gamma, \gamma\gamma\gamma\beta\alpha, \gamma\gamma\gamma\alpha\beta]$$

since we could obtain all these feedback patterns in our example in Section 2.1 simply by changing the bijective mapping μ.

With these equivalence classes our actual number of unique feedback patterns decreases to the following:

$$\frac{3^n - 3}{6} + 1 = \frac{3^n}{6} + \frac{1}{2}$$

for the first guess. We divide our original 3^n permutations by a factor of six due to the equivalence classes described above. The -3 and $+1$ terms originate from the unique equivalence class of size 3, with $[\alpha^n, \beta^n, \gamma^n]$.

However, this number of unique patterns f changes from the second guess and onward, since at that point, for every codeword $w \in V$, the set of bijective mappings μ that ensure w stays in the

list of valid codewords Vk has decreased. To further explain this we can look at our example from Section 2.1: we see that in order for $w =$ 'gleam' to return $C_{\mu,w}(\text{charm}) = \alpha\alpha\alpha\gamma\beta$, we had to bind the feedback types in μ as follows:

$$\mu(\text{gray}) = \alpha,\ \mu(\text{yellow}) = \gamma,\ \mu(\text{green}) = \beta$$

meaning that when the codebreakers enters a second guess $x =$ 'stuff':

$$C_{\mu,w}(\text{stuff}) = \mu([\text{gray, gray, gray, gray, gray}])$$

we are only able to create a single symbol feedback pattern with our predetermined bijective mapping μ:

$$\mu([\text{gray, gray, gray, gray, gray}]) = \alpha\alpha\alpha\alpha\alpha$$

, leaving us with equivalence classes of size one, meaning there are no longer any isomorphic patterns.

We now see that the size of the equivalence classes is determined by the number of bijective mappings μ that ensure the codewords w stay in V. This number of valid bijective mappings μ simply depends on the number of unique symbols $s \in S$ that have shown up in the feedback patterns f thus far, since whenever a symbol shows up in f, a codeword w must bind a feedback type in μ to that symbol to stay in the set of valid codewords V. We thus end up with either equivalence classes of size two when a single symbol has been bound in μ for all $w \in V$, or with equivalence classes of size one when all symbols have been bound. This leaves us with the following number of unique patterns:

$$\frac{3^n - 3}{2} + 2 = \frac{3^n}{2} + \frac{1}{2} \text{ for 1 unique } s \in f \text{ after first guess}$$

$$3^n \text{ unique } f \text{ for 2 or 3 unique } s \in f \text{ after first guess}$$

This shows the second guess is likely to give us more information, as we have more unique feedback patterns to distribute the answers over.

3 Algorithms

To conduct the first experiment mentioned in Section 1.2, we will compare the performance of several algorithms. We will first give a description of each of the selected algorithms and explain why they are relevant for comparison in our experiment.

3.1 Knuth's algorithm

The popular Five-Guess Knuth's algorithm was created for the original game of Mastermind(6,4,a), where 6 refers to the number of colors, or $|\Sigma|$ in our definition, 4 refers to word length n and 'a/na' is true/false and refers to allowing/disallowing duplicate colors in the codeword w. Knuth's algorithm is a slightly modified version of Mini-Max and Knuth used this to show Mastermind(6,4,a) is solvable

with five guesses in the worst case. Since this algorithm was effective for solving Mastermind, we speculate that it may perform well for the game of Symble too.

Applying Knuth's algorithm to Symble is quite straight-forward: the algorithm will try to minimize the number of answers left in the worst case. In order to determine what guess x minimizes the number of answer in the worst case, Knuth's algorithm will create a codeword distribution over the feedback patterns as shown in Figures 2 and 3 below. In this distribution we only include the codewords $w \in V$ since we have possibly already eliminated some codewords with earlier guesses. This means that Knuth's algorithm keeps track of this list V to allow for the creation of the distribution. This distribution is then created for all guesses $x \in D$ in the following manner:

- For every codeword $w \in V$, we evaluate $C_{\mu,w}(x)$ to see what feedback pattern we would obtain if w were to be the codeword. The counter of this feedback pattern is then increased by one.
- After the feedback patterns of all codewords $w \in V$ have been determined with respect to said guess x, we know how many times each feedback pattern would occur for guess x, allowing us to create a distribution as the one seen in Figures 2 and 3.

With this distribution Knuth's algorithm now determines the worst case for a guess as the feedback pattern with the most occurrences. In Figure 2 we see the distribution of the guess 'lears', and notice that the feedback pattern with most occurrences is '$\alpha\alpha\alpha\alpha\beta$', totalling 1072 codewords. Then in Figure 3 we see the distribution of the guess 'fuffy', which has a worst case result of 6785 codewords for the feedback pattern '$\alpha\alpha\alpha\alpha\alpha$'. Between these two guesses Knuth's algorithm would thus prefer 'lears' over 'fuffy' due to its more promising worst case scenario.

Knuth's algorithm prefers a guess $x \in V$ over a guess $x' \notin V$ in the case that x and x' have the same worst case. This is done because when $x \in V$, we may just be lucky and immediately end the game, which would not be possible if $x \notin V$, thus giving ourselves slightly more favorable odds.

3.2 Entropy

The theory of entropy [Sha48] uses the principles of information theory to optimize a guessing process such as the one occurring with Symble. Entropy is a measure of expected information gain from a certain variable's distribution of outcomes. The formula for the entropy is as follows:

$$\text{Entropy}(X) = -\sum_{x \in \mathcal{X}} P(x) \log_2(P(x)) \tag{1}$$

where X stands for a variable, which has a set of outcomes $\mathcal{X}$. x then represents a single one of these possible outcomes and is denoted with a value representing the size of the outcome. The entropy of a variable then is measured by the sum of the sizes of the variable's possible outcomes. This entropy value we obtain measures the 'uncertainty' of the outcomes, and in the case of a guessing game, more uncertainty means a more informative result.

Entropy can be applied to a game like Symble since the goal is to gain as much information as possible from a certain guess, where information is defined as the extent to which the list of possible answers is narrowed down. With regards to Symble, in the entropy function, X would refer to a guess that the codebreaker can submit and $\mathcal{X}$ would refer to the distribution of V over the feedback patterns regarding that guess (see Figures 2 and 3). x would then refer to the size of one of these feedback patterns. $P(x)$ would in turn stand for the probability of you receiving that specific feedback pattern when entering the guess, where this probability is calculated by dividing the occurrences of that specific feedback pattern by $|V|$ (all occurrences). The summation over these feedback patterns then results in the entropy of the guess.

While the Entropy algorithm thus uses the same distribution as Knuth's algorithm does in Section 3.1, the entroy algorithm uses the number of occurrences from all feedback patterns instead of just the largest one when determining its guess.

To give an example of what this entropy calculation looks like in a real game of Symble, a comparison of the best first-guess-entropy against the worst first-guess-entropy on the standard configuration is shown in Figures 2 and 3:

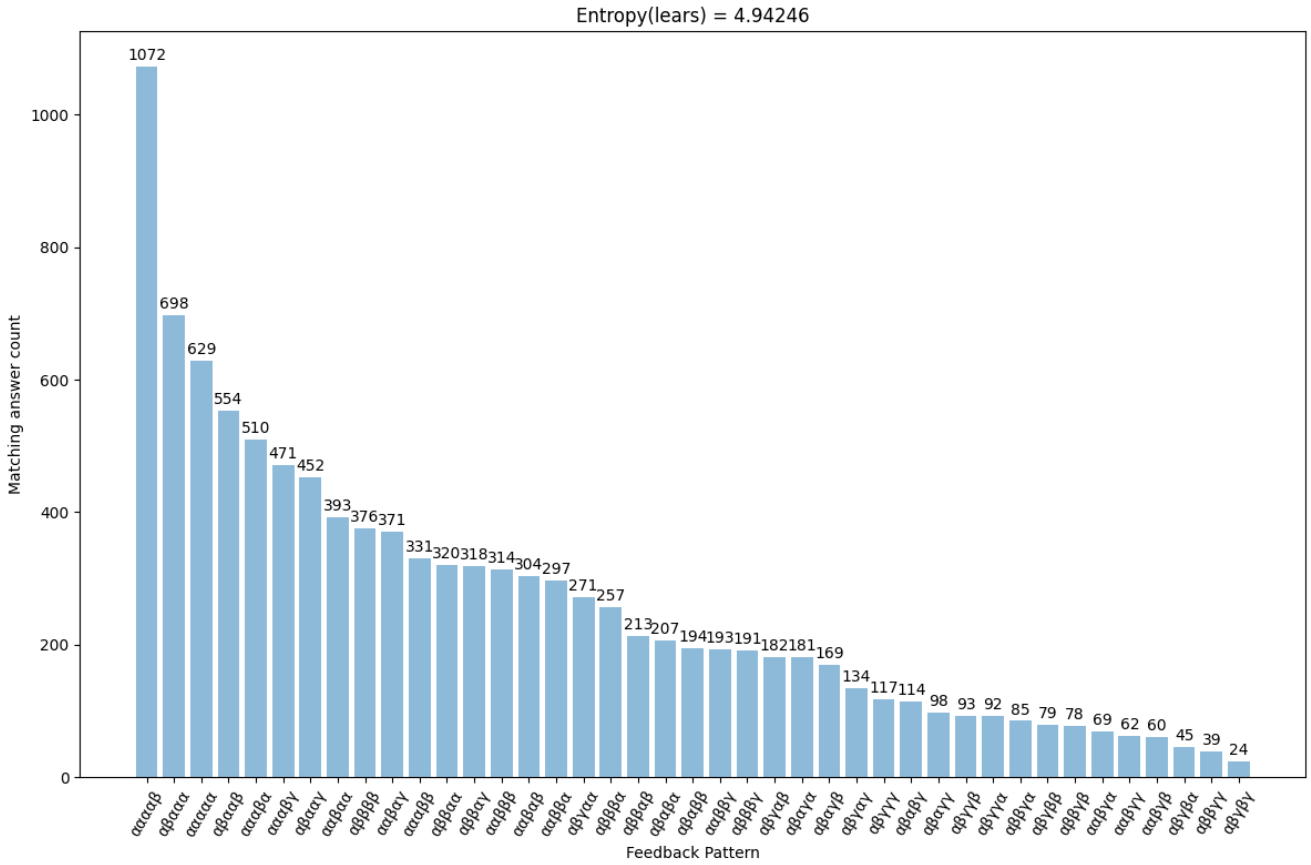

Figure 2: Distribution and entropy of feedback patterns with 'lears' as a first guess on the standard 5-letter English dictionary of Symble. The X-axis shows the possible feedback patterns and the Y-axis shows the number of codewords matching each pattern.

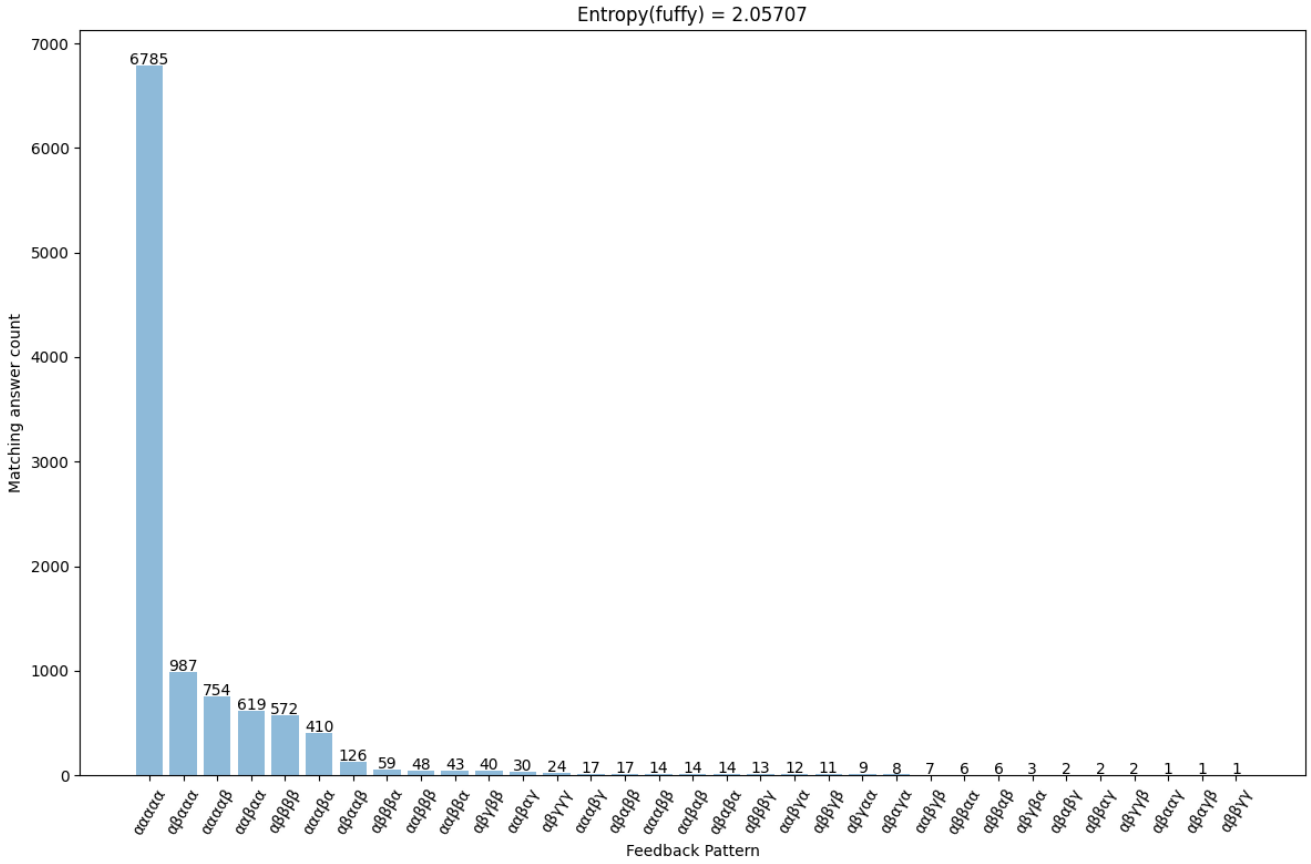

Figure 3: Distribution and entropy of feedback patterns with 'fuffy' as a first guess on the standard 5-letter English dictionary of Symble. The X-axis shows the possible feedback patterns and the Y-axis shows the number of codewords matching each pattern.

We notice that the guess 'lears' has a more uniform distribution and turns out to have an entropy of ≈4.94, whereas 'fuffy' has an entropy of only ≈2.05. This value represents the bits of information gained from the guess, where every bit of information splits the list of possible answers in half. This means the guess 'lears' provides us with more expected information and is therefore considered a better guess than 'fuffy'.

And thus our Entropy algorithm will select the guess that has the highest entropy, which in this case would mean the Entropy algorithm chooses guess 'lears' over 'fuffy'. In case of a tie for highest entropy, the Entropy algorithm does not apply a preference for a guess x being in V, such as Knuth's algorithm has in Section 3.1, but simply chooses randomly.

3.3 Random heuristic

The random heuristic is the simplest form of a heuristic algorithm. This algorithm will pick a random guess every time, but the algorithm chooses only from the list of still valid codewords V. This algorithm is chosen to have a baseline score to compare to.

This heuristic is also quite similar to how a human may play this game or a similar code-breaking game, since discovering a still valid guess at a certain point in the game is generally hard for humans, and figuring out the distributions of all $w \in V$ such as in Figures 2 and 3 is not a feasible task for a human player.

3.4 Monte Carlo

Our fourth algorithm is the popular Monte Carlo algorithm [HH64]. Unlike deterministic algorithms such as Knuth's algorithm and the Entropy algorithm, this algorithm is stochastic and uses a number of iterations for every possible action to determine its move, where these iterations are played out randomly after said move. The Monte Carlo algorithm is often used when exhaustive search is infeasible, such as in our case.

For the game of Symble an 'action' refers to a guess, and thus for Symble, the Monte Carlo algorithm will play a number of games out randomly for every guess $x \in D$. This is done by submitting the currently evaluated guess x, creating a number of copies of the game equal to the chosen amount of iterations, and then randomly playing out those games. The score for a guess x is then determined by the average score from those simulated games. The Monte Carlo algorithm then selects the guess x that has the lowest average score over those games. For our experiment we chose the number of iterations to be 100.

The pseudocode below provides further detail for the application of the Monte Carlo algorithm to game of Symble:

```
function MonteCarloAlgorithm.guess(game):
    wordList = getWordList(game)
    bestAverage = infinity
    bestGuess = null

    for each guess in wordList:
        totalGuesses = 0

        for i = 1 to iters:
            gameCopy = copy of game
            answersLeft = getAnswersLeft(gameCopy)
            set codeword in gameCopy to random element from answersLeft
            submitGuess(gameCopy, guess)

            while game is not over:
                answersLeft = getAnswersLeft(gameCopy)
                randomGuess = random element from answersLeft
                submitGuess(gameCopy, randomGuess)

            totalGuesses = totalGuesses + getScore(gameCopy)
```

```
        average = totalGuesses / iters

        if average is less than bestAverage:
            bestAverage = average
            bestGuess = guess

    return bestGuess
```

3.5 Entropy+Knuth combination

As we mentioned in Section 3.2, the standard Entropy algorithm does not have a preference for guesses $x \in V$, while Knuth's algorithm does prefer to choose a still valid codeword $x \in V$. As explained in Section 3.1, selecting a guess $x \in V$ can give us a slight advantage if we coincidentally end the game immediately by guessing right. This small error may results in a slight decrease in performance for the Entropy algorithm.

With this algorithm we combine the preference from Knuth's algorithm in selecting a guess $x \in V$ with the more sophisticated look at the pattern distribution of the Entropy algorithm to see if this does result in an increase in performance as we expect it to.

4 Experiments

In the following sections we will conduct three experiments to compare the performance of the algorithms from Section 3, determine a lower bound and see how close our random heuristic gets to this lower bound.

4.1 Experiment setup

The experiments must be set up in a way that allows us to properly compare the algorithms and configurations. First it is important to state how performance will be measured. There are three intuitive ways to measure the score over multiple games:

- **Worst-case performance:** Measures the number of guesses needed to find the codeword in the game with the most guesses played.
- **Average number of guesses:** Measures the average number of guesses used to find the codeword across all games played.
- **Percentage of games won:** Measures the proportion of games won within a given number of guesses, g.

The first two options would require $g = \infty$ and the third option would have to select a value g such that not all games are either lost or won. For our experiments the second option (average number of guesses) was chosen, since this gives us the most informative description of the performance without having to find a value for g sufficient for proper comparison.

4.2 Analysis of algorithmic performance

In order to conduct the first experiment stated in Section 1.2 regarding the performance of the algorithms, the aforementioned algorithms will be tested.

We have already chosen to measure the performance of the algorithms by the average number of guesses, and to use $g = \infty$. Now all that is left is to determine the rest of the configurations to evaluate on. We choose to set up fourteen different configurations with the use of a dictionary from a GitHub directory [3], which contains a large number of words from the English language. These words are separated into smaller dictionaries, one for each word length n, where $4 <= n <= 17$. To also see how the performance of the algorithms would scale with increasing n, we selected 1000 random words from each dictionary D so they are all of size $|D| = 1000$. We want the dictionaries to be of the same size so we can assess how a growing word length n affects the performance of all algorithms, and a size of $|D| = 1000$ allowed us to still use a reasonable amount of word lengths. Now our instances are as follows:

- Σ: The English alphabet (26 letters).
- n: Word lengths ranging from 4 to 17.
- D_n: A dictionary of 1000 words for each word length n.
- $S = \{\alpha, \beta, \gamma\}$: Placeholder symbols.
- g: ∞

For each instance $\text{SY}_n = (\Sigma, n, D_n, S, g)$, with n ranging from 4 to 17, we conduct the experiments as follows:

1. **Initialization:** For each n, the dictionary D_n is initialized with a 1000 random words.
2. **Game Simulation:** Each algorithm then plays 1000 games on each configuration, with one game for each codeword $w \in D$.
3. **Performance Comparison:** The average number of guesses needed to win across the 1000 games will be recorded for each algorithm and each word length.

This setup ensures a fair comparison by evaluating each algorithm on the same set of codewords and across a range of word lengths. By examining the average number of guesses, we can determine which algorithm performs best overall and how performance scales with word length.

The results of the experiment are given in Figure 4, where the first thing we can immediately notice is that all algorithms perform better on higher n. To explain this we can look at what we have established in Section 2.3, where we saw that the number of unique feedback patterns is determined by n, with larger n causing exponential growth. Hence, the feedback provides more information with higher n, resulting in fewer guesses needed to find the codeword (since $|D|$ is static).

[3] https://github.com/dwyl/english-words

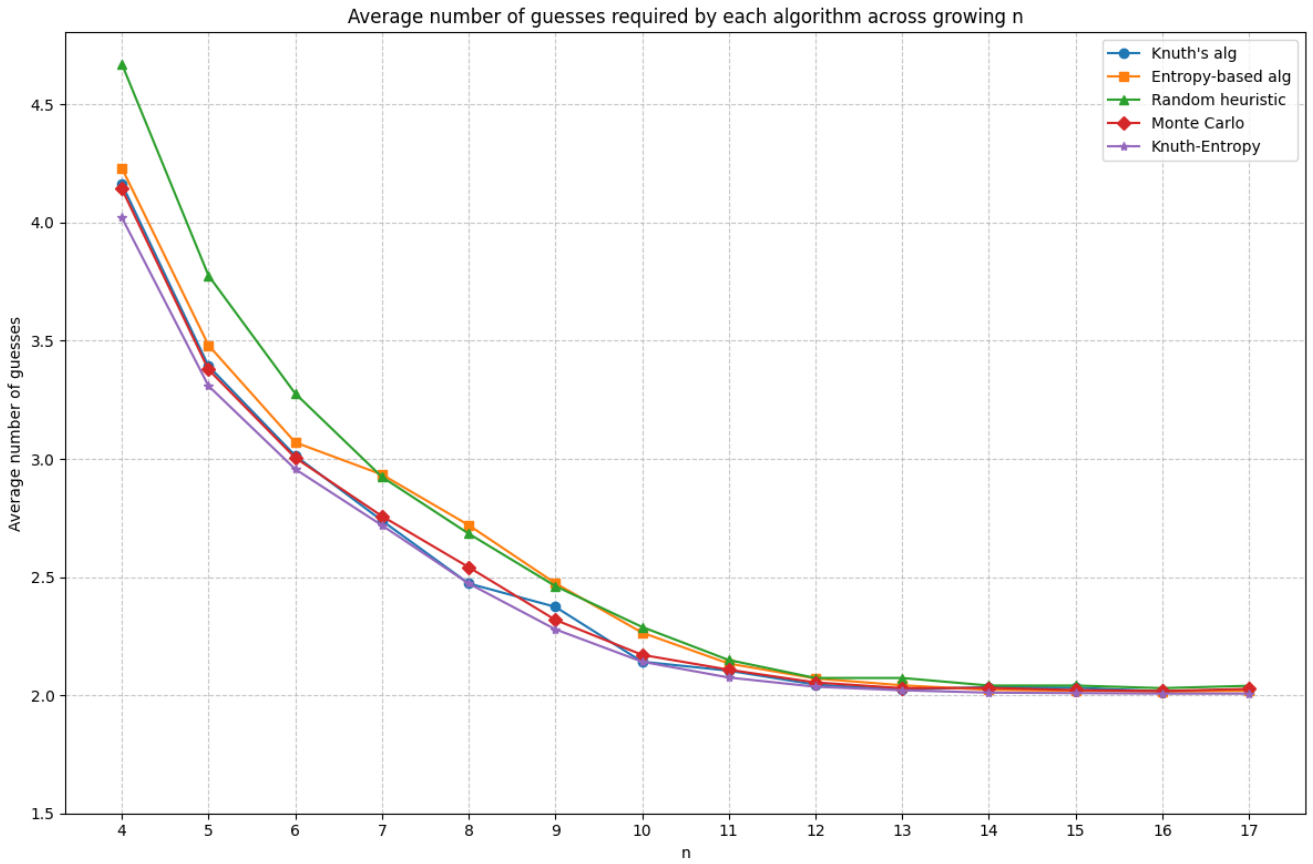

Figure 4: Performance comparison of our algorithms for increasing word length. The X-axis shows word length (4-17 letters) and the Y-axis displays the average number of guesses. A data point indicates the average number of guesses required over 1000 for that word length.

We also note that for higher values of n all algorithms get close to an average score of just two guesses, but never below. This is simply because even if after the first guess the algorithms have already extracted enough information to know the codeword, they still need to submit a second guess to end the game.

When looking at the performance of the individual algorithms we notice that, unsurprisingly, the random heuristic performs worst of all. That being said, the performance is not substantially worse. From the more sophisticated algorithms, we can see Monte Carlo, Knuth's algorithm and the Knuth-Entropy algorithm performing quite similarly, but the Knuth-Entropy algorithm performs best overall. We also note that the Knuth-Entropy algorithm is also not as computationally expensive as the Monte Carlo algorithm, giving us a second reason to conclude that it is the best performing algorithm.

A last thing to note is the difference in performance between Knuth's algorithm and the Entropy algorithm. While they both make use of the distribution of possible outcomes, Knuth's algorithm takes a simpler approach in deciding the best guess from these distributions, and still outperforms the Entropy algorithm. We suspect this is due to the slight imperfection with our Entropy algorithm

we mentioned in Section 3.5, since we can see the combined algorithm of Knuth-Entropy doing better than both Knuth's algorithm and the Entropy algorithm.

4.3 Evaluating performance on variable configurations

We have now examined the difference in performance with the selected algorithms. To prepare for the experiment regarding the configurations mentioned in Section 1.2, we need to set up different configurations for comparison.

While in Section 4.2 we were limited in the variability of configurations by the use of real English words (with 1000 per word length n), for the following experiments we will use a complete dictionary $D = \Sigma^n$, as we know it from Mastermind. With that we will use numbers instead of letters for Σ from here on, since letters will not be making much sense if all permutations are allowed. Using such a complete dictionary allows us to more accurately investigate how different configurations affect the average score, without having to deal with preferences for more common letters.

In these experiments, for consistency's sake, we will use only a single algorithm. The random heuristic was selected for this because of its speed. While the random heuristic performed slightly worse in the previous experiment, this difference was not severe and the algorithm will still be able to show the growth or decline in difficulty with varying configurations. Another advantage that the use of the random heuristic grants is its aforementioned similarity to the intuitive human-player strategy.

With the growing configurations we will no longer be able to conduct a complete test where every $w \in D$ is used as a codeword once. Instead random codewords will now be selected for all games played in the experiments.

4.3.1 Estimating a lower bound with information theory

Before executing the experiments we will construct a lower bound for the number of guesses in growing configurations. This lower bound can show us the minimum number of guesses an algorithm will have to do in the worst case for a certain configuration. The worst case for an algorithm is the maximum number of guesses it needs to solve the game when playing with every possible codeword from the configuration.

For this, we first observe that dictionary size $|D|$ for variables n and alphabet size k equals k^n. We combine this with the number of unique feedback patterns we computed in Section 2.3. In our construction of the lower bound, we assume a uniform distribution of the still valid codewords V over the feedback patterns. We assume a uniform distribution because this is the best any algorithm could ever do in terms of the worst case (meaning most occurring feedback pattern here).

As established in Section 2.3, the first guess has $\left(\frac{3^n}{6} + \frac{1}{2}\right)$ unique feedback patterns, while subsequent guesses will at most have 3^n unique feedback patterns. When we use x to denote the number of guesses needed to figure out the word w, we find:

$$\left(\frac{3^n}{6}+\frac{1}{2}\right)\cdot(3^n)^{x-1}=k^n$$

$$\Longleftrightarrow$$

$$\ln\left(\left(\frac{3^n}{6}+\frac{1}{2}\right)\cdot(3^n)^{x-1}\right)=\ln\left(k^n\right)$$

$$\Longleftrightarrow$$

$$\ln\left(\frac{3^n}{6}+\frac{1}{2}\right)+(x-1)\cdot n\cdot\ln(3)=n\cdot\ln(k)$$

$$\Longleftrightarrow$$

$$x-1=\frac{n\cdot\ln(k)-\ln\left(\frac{3^n}{6}+\frac{1}{2}\right)}{n\cdot\ln(3)}$$

Thus, the lower bound for the number of guesses needed to figure out w in the worst case, is:

$$x=\frac{n\cdot\ln(k)-\ln\left(\frac{3^n}{6}+\frac{1}{2}\right)}{n\cdot\ln(3)}+1 \tag{2}$$

We have to note two more things. First, we have defined x to be the number of guesses needed to figure out w, but in the real game of Symble you would still need to then submit that guess in order to end the game, resulting in an extra term $+1$ for the actual number of guesses.

Second, we must recall that this bound is based on a uniform distribution, which is simply not possible in most, if not all, configurations. To show this we look at an example of how even the guesses with the supposed best distributions according to Knuth's algorithm and our Entropy algorithm, are not uniform:

A uniform distribution for the first guess would result in an entropy of $\log_2(\frac{3^n}{6}+\frac{1}{2})$. As we have already seen in Figure 2 however, for the case of D as five-letter English words, even the guess with the highest entropy of ≈ 4.942 fails to achieve this perfect entropy of $\log_2(\frac{3^5}{6}+\frac{1}{2})\approx 5.358$.

This example was on an incomplete dictionary, and since the experiments we are conducting below are based on complete dictionaries, we will also look at an example of how we fail to reach this uniform distribution with perfect entropy there. We look at the first guess with the highest and lowest entropy for a configuration with $n=4$ and $k=6$. Here the number of unique feedback patterns is $\frac{3^4}{6}+\frac{1}{2}=14$, hence the maximum possible entropy is $\log_2(14)\approx 3.807$. Figure 5 shows the highest first-guess-entropy with guess '0012', and Figure 6 shows the lowest first-guess-entropy with guess '0000'.

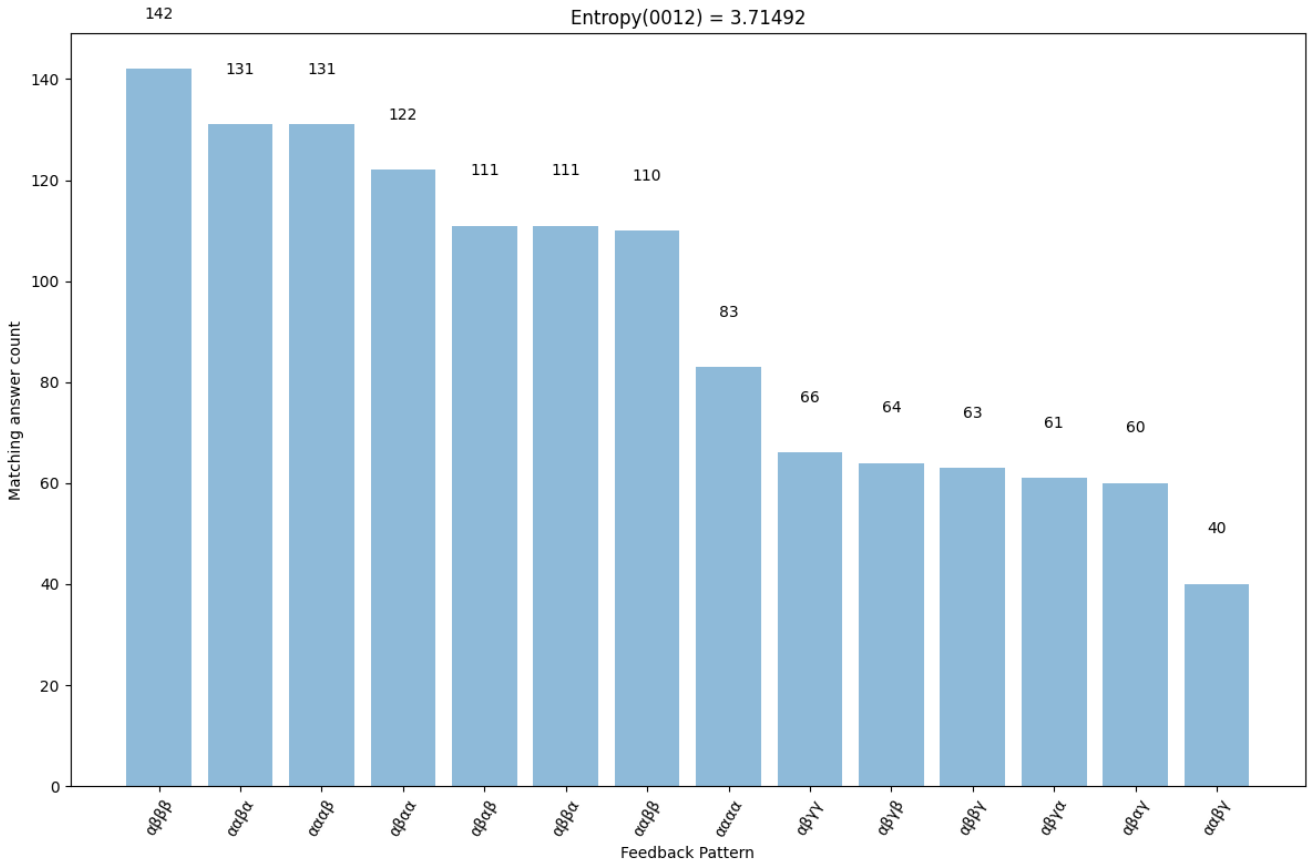

Figure 5: Distribution and entropy of feedback patterns with guess '0012' as a first guess for a configuration with n=4 and k=6. The X-axis shows the possible feedback patterns and the Y-axis shows the number of codewords matching each pattern.

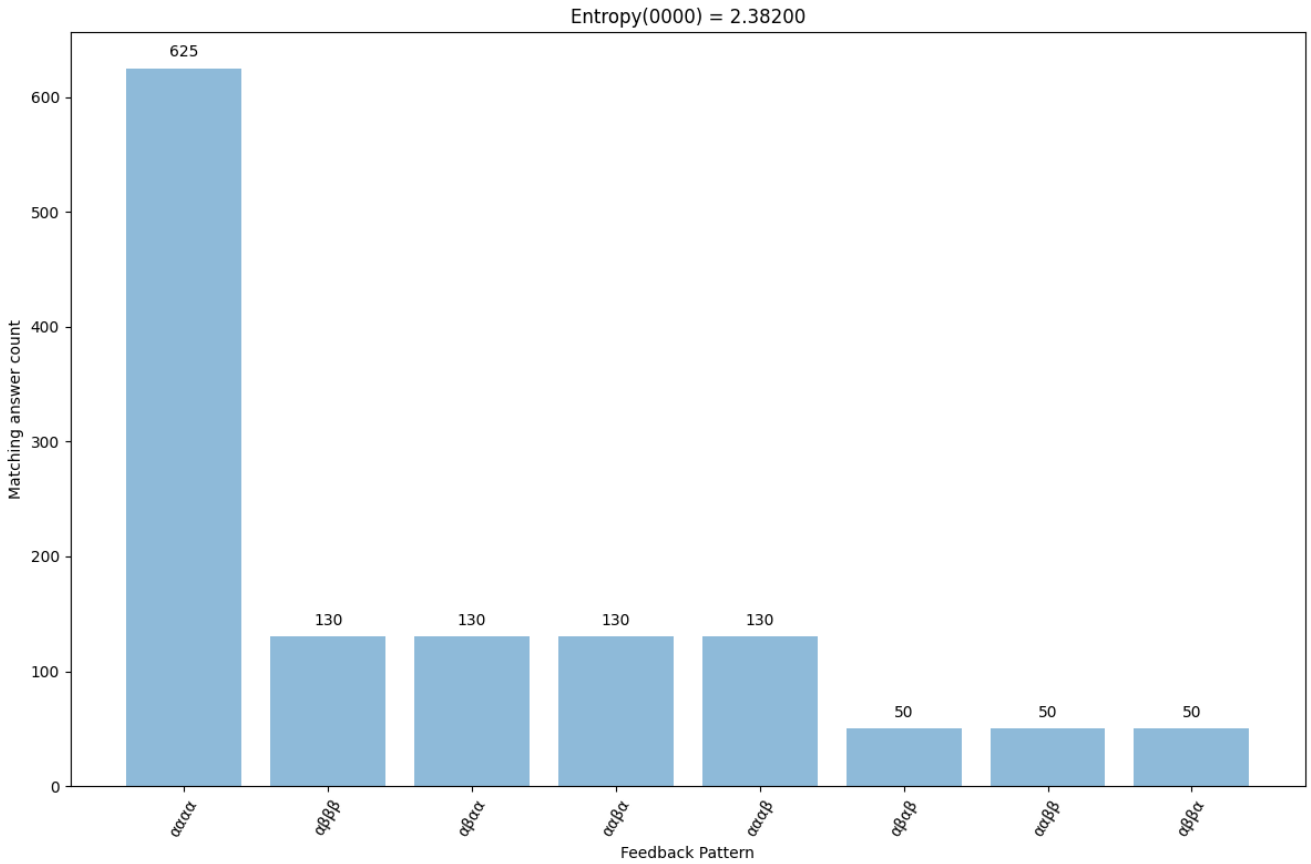

Figure 6: Distribution and entropy of feedback patterns with guess '0000' as a first guess for a configuration with n=4 and k=6. The X-axis shows the possible feedback patterns and the Y-axis shows the number of codewords matching each pattern.

Although the guess '0012' gets relatively close to 3.807 with an entropy of 3.715, it is clear that the uniform distribution that our lower bound assumes is simply not possible due to the nature of the feedback; meaning our lower bound could be made sharper if this inherent imperfection is accounted for, but we will only cover a simple example of this later on.

4.3.2 Growing word size n

Whereas Section 4.2 considered a growing word length n with a static dictionary size, the new setup has the dictionary grow with the word length. With the complete dictionary $D = \Sigma^n$ the aim is to see whether the increasing information gained from the longer pattern can make up for the increasing dictionary size $|D|$.

We have to decide on a fixed alphabet size k for this experiment and we have chosen $k = 6$ for this, since this is also what is used in Mastermind. Our second experiment is thus set up using the following instances of SY:

- $SY_i = (|\Sigma| = 6, i, \Sigma^i, \{\alpha, \beta, \gamma\}, \infty)$ with $2 <= i <= 9$

Since we have chosen a fixed alphabet size k, we can simplify our lower bound to see how we expect the average number of guesses to evolve with growing word length n in this experiment. We do this by substituting $k = 6$ in Equation (2), where we get:

$$x = \frac{n \cdot \ln(6) - \ln\left(\frac{3^n}{6} + \frac{1}{2}\right)}{n \cdot \ln(3)} + 1 = \frac{\ln(6)}{\ln(3)} - \frac{\ln\left(\frac{3^n}{6} + \frac{1}{2}\right)}{n \cdot \ln(3)} + 1$$

While this function is not very intuitive, we can use it to make a prediction of the results from the experiment, since for larger n:

$$\frac{\ln\left(\frac{3^n}{6} + \frac{1}{2}\right)}{n} \approx \frac{\ln\left(\frac{3^n}{6}\right)}{n} \approx \frac{n \cdot \ln(3) - \ln(6)}{n} \approx \ln(3)$$

Thus, for large n, the expression simplifies to:

$$x \approx \frac{\ln(6)}{\ln(3)} - \ln(3) + 1$$

And since $x \approx \frac{\ln(6)}{\ln(3)} - \ln(3) + 1 \approx 1.532$ is a constant, we expect to see the number of guesses remain stable for larger n.

In Table 2, we display the values of the lower bound alongside the results of the experiments to see if they follow the same trend. It is important to keep in mind that the lower bound does not include the +1 term mentioned in Section 4.3.1.

Table 1: Average number of guesses required by the Random heuristic on configurations with growing n and fixed $k = 6$. The lower bound for the worst case for these configurations is also shown for reference.

Word Length	Random heuristic	Lower bound
2	3.938	2.219
3	3.950	2.022
4	3.890	1.924
5	3.862	1.866
6	3.858	1.827
7	3.875	1.799
8	3.882	1.778
9	3.916	1.762

The experiment shows that our assumption was correct, in that the number of guesses remains stable. The results also show that the increase in information gained from longer words balances out the increase in $|D|$.

4.3.3 Growing alphabet k

A third experiment is set up regarding a growing alphabet, using a complete dictionary such as in Section 4.3.2.

While in the previous experiment we tried to see if the difficulty in a growing dictionary would outweigh the ease of more information, in this experiment we do not have such a goal; instead we simply seek to analyse how the number of average guesses grows with alphabet size k.

This time we have to decide on a fixed word length n to measure growing k; and we have chosen $n = 5$ here, just as in the regular game of Symble. We can thus define instances of SY for this experiment:

- $SY_i = (|\Sigma| = i, 5, \Sigma^5, \{\alpha, \beta, \gamma\}, \infty)$ with $2 <= i <= 25$

Since this time we chose a fixed n, we can simplify our lower bound again to see how the minimum number of guesses in the worst case grows with respect to k: We do this by substituting $n = 5$ in Equation (2), where we get:

$$x = \frac{5 \cdot \ln(k) - \ln\left(\frac{3^5}{6} + \frac{1}{2}\right)}{5 \cdot \ln(3)} + 1$$

$$x = \frac{5 \cdot \ln(k) - \ln(41)}{5 \cdot \ln(3)} + 1 = \frac{\ln(k)}{\ln(3)} - \frac{\ln(41)}{5 \cdot \ln(3)} + 1$$

This formula may be more intuitive than the one we saw in Section 4.3.2, since all terms are constants except for $\frac{\ln(k)}{\ln(3)}$, and we thus expect the number of guesses to grow with $\log_3(k)$ for larger alphabet sizes k. The results of our experiment are shown in Table 2.

Table 2: Average number of guesses required by the Random Heuristic on configurations with growing k and fixed $n = 5$

Letters	Random Heuristic	Letters	Random Heuristic
2	2.253	14	6.610
3	2.899	15	6.936
4	3.238	16	7.346
5	3.542	17	7.697
6	3.858	18	8.058
7	4.185	19	8.395
8	4.513	20	8.716
9	4.848	21	9.097
10	5.150	22	9.557
11	5.507	23	10.013
12	5.886	24	10.251
13	6.259	25	10.564

To compare this to our established lower bound we have plotted the results from our experiment alongside our formula for the lower bound in Figure 7. For the lower bound, again the term +1 mentioned in Section 4.3.1 for submitting the solution once it has been determined, is not counted.

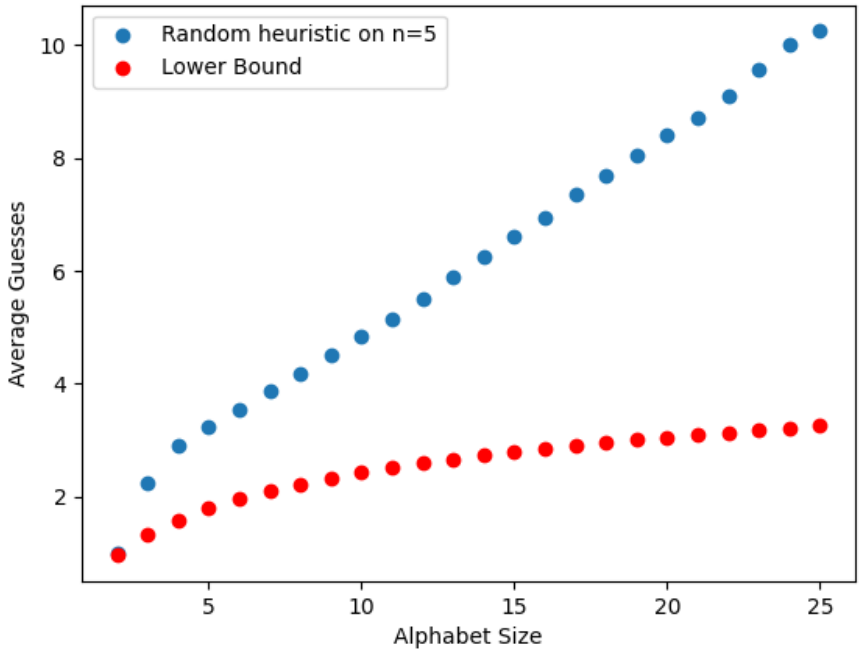

Figure 7: Comparison of average guesses between lower bound and results from our experiment with increasing alphabet size (k) and fixed word length (n=5). The X-axis shows alphabet size and the Y-axis displays the average number of guesses.

In contrast to the previous experiment, we see something unexpected happening: the score deviates in growth from that of the lower bound. The average number of guesses required seems to grow linearly with k instead of logarithmically as the lower bound would indicate.

To explain this deviation, we will revisit the analysis in Section 4.3.1, where we mentioned that our lower bound depends on uniformly distributing the valid codewords V over all feedback patterns. We also showed that such a uniform distribution does not exist in practice. This imperfection in the distribution becomes more apparent for $k > n$. To demonstrate this, let us look at the first guess with the highest entropy for the configuration with $n = 4$ and $k = 30$, which is '0123'. The distribution of V over feedback patterns for the first guess '0123' is shown in Figure 8.

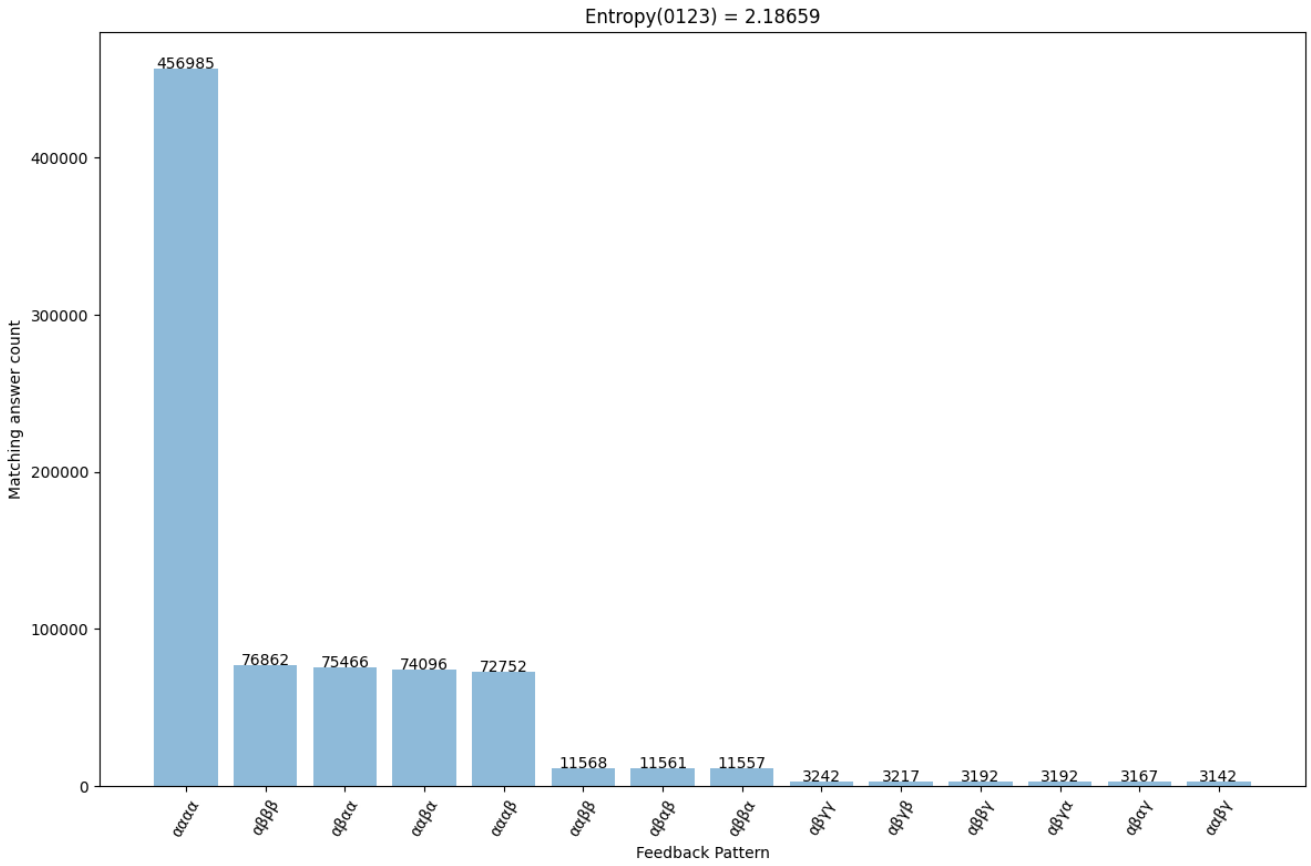

Figure 8: Distribution and entropy of feedback patterns with guess '0123' as a first guess for a configuration with n=4 and k=30. The X-axis shows the possible feedback patterns and the Y-axis shows the number of codewords matching each pattern.

Here we see the highest possible entropy for the first guess is far from the perfect entropy $\log_2(\frac{3^4}{6} + \frac{1}{2}) = \log_2(14) \approx 3.807$. We observe that this is partly because of the number of codewords w with feedback pattern '$\alpha\alpha\alpha\alpha$' becoming rather large.

This abundance of codewords w returning α^n highlights an imperfection in our uniform assumption, because when, for larger alphabet sizes k, a first guess has no repeating letters, $(k-n)^n$ unique codewords w (all w not containing any letters from guess x) fall into the category of only grays. Approximately at $k > 2n$ that category is much larger than the number $k^n/(\frac{3^n}{6} + \frac{1}{2})$ that our uniform distribution assumes.

The feedback pattern 'α^n' not only contains the codewords with all grays, but further contains all codewords w that would return only 'yellow' or 'green'. For a guess without repeating letters the number of codewords w returning only yellows amounts to $\lfloor \frac{n!}{e} + \frac{1}{2} \rfloor$, following the derangement formula [MH12]. The codeword that returns all greens in its feedback pattern is omitted since this would mean the codeword has been found and the game has ended.

We can now calculate the actual number of w returning α^n (for a guess without repeating letters) to create an adjusted entropy approximation for the first guess. This adjusted entropy uses the summation of all grays $(k-n)^n$ in combination with all yellows $\lfloor \frac{n!}{e} + \frac{1}{2} \rfloor$ to calculate the size of α^n, but still assumes a uniform distribution for the rest of the codewords w over the feedback patterns. We now show, for the first guess, the (impossible) perfect entropy, the adjusted entropy that accounts for the large group of α^n and the actual entropy we achieved in our experiment, in Figure 9.

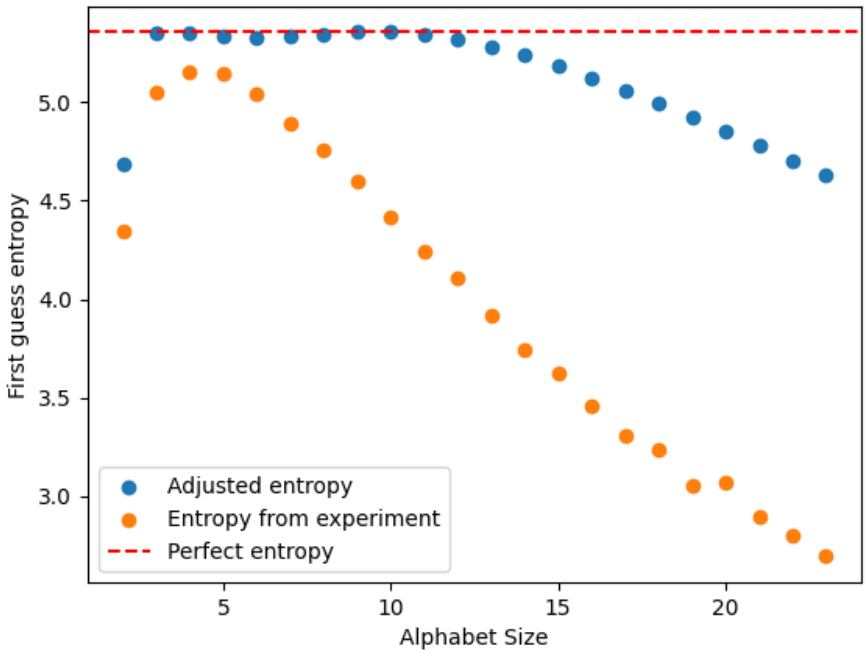

Figure 9: Comparison of different entropy calculations for the first guess for configurations with increasing alphabet size (k) and fixed word length (n=5). X-axis shows alphabet size and the Y-axis displays entropy of the first guess in bits.

Indeed, accounting for the abundance of codewords returning $C_{\mu,w}(x) = \alpha^n$ as feedback, we get a more accurate approximation, especially for sufficiently large k.

Regarding our lower bound we can now also intuitively conclude that the worst case for $k > n$ will have the codemaster return α^n until the correct letters from w have been found. Since the codebreaker is only able to check n letters at once, we can start to see an explanation for the linear relationship, where the first $\lfloor \frac{k}{n} \rfloor$ guesses are lost in this process. Note that this is not an actual lower bound but simply an approximation, which we will include in Figure 10 to see how it holds up.

We conclude that this approximation better follows the growth shown in our experiment. We do, however, observe that the slope of the linear growth measured is roughly twice as high as that of the

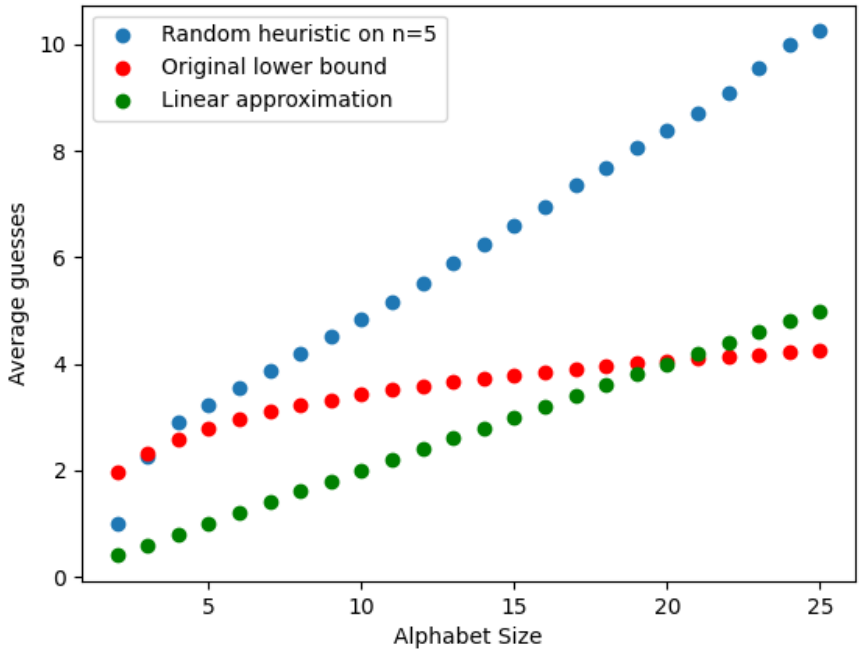

Figure 10: Comparison of average guesses between lower bound, linear approximation and results from our experiment with increasing alphabet size (k) and fixed word length (n=5). The X-axis shows alphabet size and the Y-axis displays the average number of guesses.

linear approximation $\frac{k}{n} = \frac{k}{5}$. We also observe that our original bound is still sharper with relatively low k, since our description of returning α^n until the correct letters from w have been found is only applicable with k sufficiently large. To support this claim, we have repeated the experiment from above for $n = 3$, to be able to explore higher values for k, and we once again compare these results to the adjusted entropy, lower bound and the linear approximation using Figures 11 and 12.

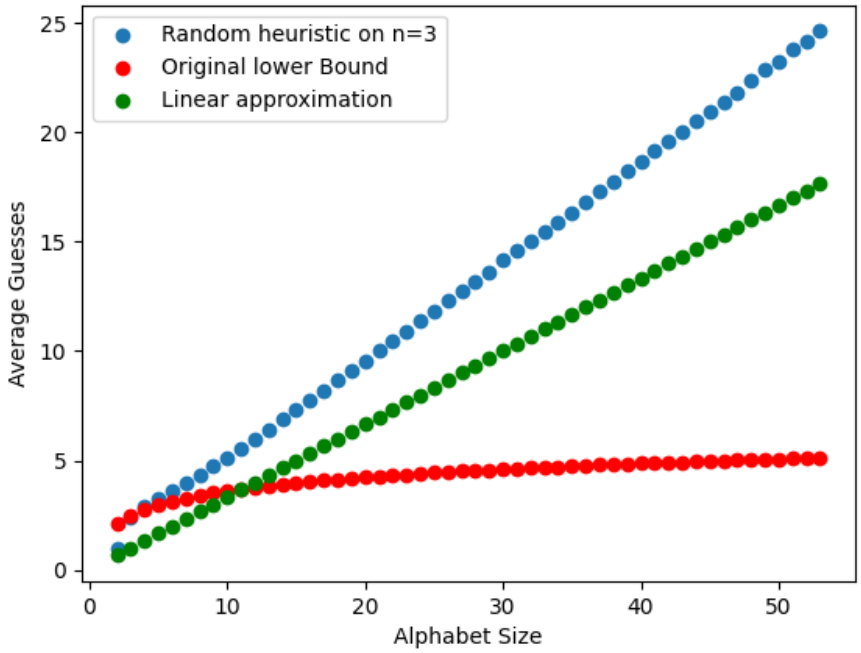

Figure 11: Comparison of average guesses between lower bound, linear approximation and results from our experiment with increasing alphabet size (k) and fixed word length (n=3). The X-axis shows alphabet size and the Y-axis displays the average number of guesses.

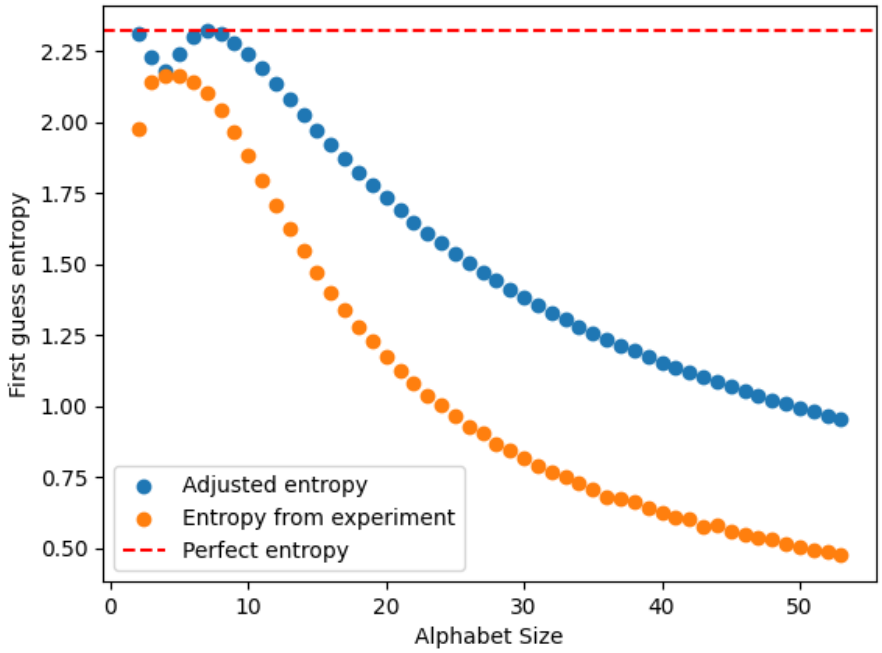

Figure 12: Comparison of different entropy calculations for the first guess for configurations with increasing alphabet size (k) and fixed word length (n=3). X-axis shows alphabet size, Y-axis displays entropy of the first guess in bits.

We can conclude that Figure 11 indeed demonstrates the better accuracy of the approximate $\lfloor \frac{k}{n} \rfloor$ for sufficiently large k.

5 Conclusions and further research

In this thesis, we explored the Wordle-inspired game of Symble, evaluated the performance of various standard algorithms on it, and thoroughly analysed the effect a configuration of Symble has on its difficulty. Through our experiments and analysis with information theory, we gained the following insights:

- Knuth's algorithm, which was created to solve the standard game of Mastermind(6,4,a) also showed to be effective at solving the standard configuration of Symble. A combination of this algorithm with an Entropy-based algorithm performed slightly better.
- A lower bound was established for a configuration of Symble with a complete dictionary like that of Mastermind. With the help of information theory we ended up with the following lower bound for number of guesses to find the codeword:

$$x = \frac{n \cdot \ln(k) - \ln\left(\frac{3^n}{6} + \frac{1}{2}\right)}{n \cdot \ln(3)} + 1$$

- By using a simple random heuristic, we could already get quite close to this lower bound for a fixed alphabet size k, growing word length n and with a complete dictionary D.
- We showed that the lower bound was not sharp for $k > n$ and can be improved upon. This is done by finding a linear approximation $\lfloor \frac{k}{n} \rfloor$ that takes into account the codebreaker 'wasting' guesses on finding the correct letters.

Our results still leave room for further research, where we propose expanding on this research in the same manner as has been done for Wordle and Mastermind, by for example:

- Proving the complexity of Symble, possibly by using a similar method to the one used to prove the NP-completeness of Wordle [Ros22].
- Tightening the lower bound by further analysing the imperfections in pattern distributions.
- Studying the change in complexity when we introduce the concept of 'no double letters' from Mastermind.
- Analysing how an incomplete dictionary, such as the one in the standard game impacts the average number of guesses. Especially by looking at how some letters will appear more frequently than others, or appear more commonly in certain positions and how this affects the number of guesses needed to win in Symble.
- Exploring the performance of additional algorithms, possibly including machine learning, algorithms from the field of natural computing or a non-adaptive such as the one explored in [DDST13]. Here a non-adaptive strategy would also allow for exploration of larger configurations.

References

[DDST13] Benjamin Doerr, Carola Doerr, Reto Spöhel, and Henning Thomas. Playing mastermind with many colors. *Theoretical Computer Science*, 487:1–18, 2013.

[dG19] Sylvester de Graaf. *Cracking the Mastermind Code*. Bachelor thesis, Leiden Institute of Advanced Computer Science, Leiden University, July 2019.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, Reading, MA, 1989.

[HH64] J. M. Hammersley and D. C. Handscomb. Monte Carlo methods. *Methuen & Co Ltd*, 1964.

[Knu76] Donald E. Knuth. The computer as master mind. *Journal of Recreational Mathematics*, 9(1):1–6, 1976.

[MH12] John H. Mathews and Russell W. Howell. *Complex Analysis for Mathematics and Engineering*. Jones & Bartlett Learning, Burlington, MA, 6th edition, 2012. Section on derangements.

[Par99] David Parlett. *The Oxford History of Board Games*. Oxford University Press, Oxford, 1999.

[Ros22] Will Rosenbaum. Finding a winning strategy for Wordle is NP-complete, April 2022. Preprint.

[Sha48] Claude E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27(3):379–423, 1948.

[SZ05] Jeff Stuckman and Guo-Qiang Zhang. Mastermind is NP-complete. *Journal of Recreational Mathematics*, 33(2):116–120, 2005.

[vO18] Vivian van Oijen. *Genetic algorithms playing Mastermind*. Bachelor thesis, Utrecht University, 2018.