Using Information Theory to Play Wordle as Optimally as Possible

Neelesh Mishra

Received December 18, 2024
Accepted April 01, 2024
Electronic access April 30, 2024

Wordle is a game where the player must guess a secret word within 6 chances. With each guess, the player obtains information in the form of a pattern, which shows how close the guess is to the secret word. The goal of the game is to guess the secret word with as few guesses as possible, and the most efficient way to solve this game is through an algorithm. Given that we are dealing with observations and a limited space of possibilities, we can use the idea of entropy, which is the average expected information gain from each guess. Along with entropy, we can use other ideas to add to the algorithm, which not only improve the success rate of the algorithm but also minimizes the average number of guesses it needs to guess the secret word. This paper introduces the mathematical background using information theory, formally defines the problem and explains the rules of the game, and proposes an algorithmic solution which was developed using Google Colab. It also discusses how the algorithm is implemented in python as well as in-depth explanations for each section of the code, as well as how one could use these learnings to improve human strategy for playing the game as optimally as possible. As for testing the algorithm and its performance, thorough testing was conducted on the multiple versions of the algorithm, using all of the possible answer words which are used for the game. The performance of the algorithm has also been discussed in detail, providing key insights as well as what could be done to improve it further.

Introduction

Motivation: My motivation to write this research paper and implement an algorithmic solution of my own was not only to explain the fundamentals of information science to a broad audience in a concise and straightforward manner, but also to further explore the applications of information theory with a formal definition of a puzzle such as wordle, by developing an algorithm to achieve better performance compared to existing solutions.

Research Questions: The specific research question I aim to explore is how can we use concepts in information theory such as entropy to solve wordle as optimally as possible? From my research, I have found that using entropy is one of the best and simplest ways to tackle this problem in a systematic manner. Therefore, I would like to explore solutions based on this research which could be used to improve human strategy. I intend to justify this by comparing my results with those of other strategies. I plan to observe what can be done to improve this strategy further.

Defining the Problem: Wordle is a game where the player must guess a mystery 5-letter word, and they have 6 chances to do so. After each guess, the player is provided with information about the guess in the form of a pattern of colored squares, which shows how close that guess was to the final answer. For example, if the secret word was, "TRAIN", and the guess was "CRANE", the corresponding response would be "BGGYB", where 'B' represents a black square, 'G' represents a green square, and 'Y' represents a yellow square. A black square means that letter is absent from the word, a green square means that letter is present in the word and is in the correct position, and a yellow square means that letter is present in the word but in the wrong position. Sometimes, tricky scenarios can arise where the guess may have 2 instances of the same letter, but the secret word has only one instance of that letter. If one of those instances is in the correct position, the other instance of that same letter will simply be marked as a black square. However, if neither of the two instances are in the correct position, the first instance will be marked as a yellow square and the second will be marked as a black square. For example, if the secret word was "ARISE" and the guess was "RADAR", the according response would be "YYBBB", where only the first instances of the letters R and A are marked yellow, and the second instances of those letters are marked as black, because the secret word only contains one instance of each of those letters. The goal of the game is to try and guess the answer in the least possible number of guesses.

Wordle is a good example of how we can use information theory, and more specifically entropy to design a strategy to play a game as optimally as possible. In simple terms, entropy is the measure of how informative an observation is with respect to achieving the final goal, but the topic of entropy will be discussed in more detail later.

Literature Review

In this section, I will be discussing some of the existing solutions to the wordle problem including determining the best

starting words using character statistics, and combining rank one approximation with latent semantic indexing.

A. Selecting Starting Words Based on Character Statistics

The objective of Ninansa de Silva's work [2] is to determine the best set of the first 3 guesses covering 15 unique characters in Wordle by using character statistics. The methodology is as follows:

- Obtain the official list of words which are valid guesses that are accepted by wordle.
- Initialize and populate a character frequency map, which maps each character of the alphabet to how frequently it occurs in the list of words.
- Derive another subset of words from the main list of words that contain no repeated letters.
- Define a word value map, which maps each word to a value based on the character frequencies of the letters in that word.
- Define a function for word overlap which outputs a Boolean value depending on whether there is at least one letter common between two candidate words.
- Define a function that returns a list of the best words, meaning the words from the word value map which have the highest values.
- Define another function which simplifies the list of best words that was just created. It does this by removing words from the list which have any letter overlap with the first element in the list (best guess), and keeping the rest.
- Define the filtered word value map, which is filtered based on a given word such that the word value map contains only the words which don't have letter overlap with the given word.
- Define a candidate processing function which returns sets of 3 words that have the highest word values and have no overlap (15 unique letters).

The results obtained from this methodology are as follows:
Best set of first three guesses: 'AESIR','DONUT','LYMPH'
Best set of first three guesses with more common words:
'RAISE','CLOUT','NYMPH'

This work was successful in that these sets of starting words guarantee a very high chance of success for any player that uses them, although it may not be the most optimal way to solve the game.

Table 1 Calculated Character Frequencies

Character	Frequency	Character	Frequency
A	0.1124	N	0.0546
B	0.0268	O	0.0653
C	0.0330	P	0.0263
D	0.0355	Q	0.0015
E	0.0994	R	0.0648
F	0.0147	S	0.0754
G	0.0236	T	0.0491
H	0.0290	U	0.0399
J	0.0661	V	0.0114
K	0.0057	W	0.0135
L	0.0224	X	0.0042
I	0.0556	Y	0.0314
M	0.0318	Z	0.0069

B. Rank one Approximation

The work done by Michael Bonthron[3] proposes a solution to solve wordle optimally by converting a word to a column vector and combining a rank one approximation and latent semantic indexing to a matrix representing the list of possible solutions. The methodology is as follows:

1. • Each word can be represented using a vector that has 26 rows and 5 columns, with each row representing each letter in the alphabet, and each column representing the position where that letter occurs in the word. This structure can then be converted to a 130 * 1 column vector by stacking each of the columns on top of each other.
2. A 130 * n matrix can then be used to represent n words which are all 130 * 1 column vectors.
3. Rank one approximation is performed on this matrix, and the result is a column vector that best represents this matrix. Latent semantic indexing can then be used to find the column from the original matrix which is closest to the result we got from rank one approximation.
4. Latent semantic indexing works by taking a query vector and sorting a set of vectors based on their similarity to this query vector.

Applying this to Wordle:

- A list of possible answers is created based on the information obtained from a guess. This is done as follows. For each answer in the current list of possible answers, check if the pattern obtained from that answer and the guess is the same. If the two patterns match, then that word is still a possible answer, otherwise that word can safely be eliminated from the list of possible answers.

- each of the words in the list to a column vector, and combine all of the column vectors to create a matrix.
- Perform rank one approximation on the matrix, and use the result of the approximation as the query vector for latent semantic indexing.
- As a result, we obtain the word which is most representative of the list, and we use this as our guess.
- Repeat this process until the secret word is guessed.

The results for this methodology are are presented in Table: 2.

Observations:

- Letter location is simply the order in which the letters are placed in the word.
- Considering the letter location results in a lower average number of guesses- this may be because certain letters may occur in particular positions in the word more often than other positions.
- "SLATE" seems to be the best starting word for both methods.

This work was able to achieve a systematic solution to the Wordle problem with a relatively low average number of guesses at 4.04, but it is at a disadvantage because it cannot distinguish between words that are simply considered as valid guesses and words that are actually possible answers.

Contribution

The objective of this paper is to explain the fundamentals of information theory to a wide audience in a simple and concise manner, as well as testing the boundaries of algorithmic solutions to wordle that involve entropy. I have written and published my own code for this algorithm as open-source, with clear comments that explain what each part of the code does, so that it can also be used as a tool for learning along with this paper. The paper is organized as follows, First it provides a formal definition of wordle as a math problem, then it describes the implementation of my algorithm including explanations for each function, moreover it describes the tests that were conducted as well as the code used for those tests, Section V reports the results from the tests, and finally it provides discussion for those results.

Problem Formulation

This section aims to formally define the wordle problem as a math problem. It is as follows:

- Obtain a word list *G*, which contains all 12972 words (the official list) which are accepted as valid guesses by wordle.
- Obtain a second word list A, which contains all 2315 words that are the possible secret words (a subset of *G*).
- The goal is to maximize the information gain(entropy) from each guess, the formula for which is shown in Equation 1 below:

$$H(X) = \sum_{x \in X} p(x) \times \log_2\left(\frac{1}{p(x)}\right)$$

Assumptions:

- The algorithm will operate based on the knowledge that only 2315 out of the total 12972 words are actual answers, meaning that it knows which words are not possible answers, and it can hence exclude those from the initial space of possibilities.
- The algorithm considers each of the 2315 words in A equally likely to be an answer, even though some words may seem more common than others.
- After an answer has been guessed, the algorithm doesn't exclude that answer from the space of possibilities for the next round of wordle; it starts with the same initial list of words for each round.

Algorithm and Implementation

This section describes the details of implementation of the algorithm, which was written in Python on Google Colab.

First, we need to import a few libraries which will be used later, and define our answer list and guess list, which we can get from GitHub:

Pattern function

In order to "play" wordle, we need a function which will return a color pattern when given a secret word and a guess word. Figure II below shows the code for that function.

The following describes how the function works:

- The function takes two parameters, secret and guess, and returns a pattern.
- Both the secret and guess are converted from strings to lists using the list() function, which makes it easier to compare the individual letters in them.

Table 2 Results From Each Method

Control Method				
	Avg. Guesses	**Win %**		
RANDOM	4.59	88.2%		
Rank One Approximation with Latent Semantic Indexing				
	Not Considering Letter Location		**Considering Letter Location**	
Starting Word	**Avg. Guesses**	**Win %**	**Avg. Guesses**	**Win %**
SOARE	4.22	93.4%	4.13	97.8%
ALERT	4.21	96.2%	4.10	98.1%
SORES	4.40	93.1%	4.26	97.5%
BARES	4.27	95.9%	4.14	98.3%
SLATE	4.15	96.3%	4.04	98.7%

```
import pandas as pd
import math
import requests

guess_words_url = 'https://gist.githubusercontent.com/dracos/dd0668f281e685bad51479e5acaadb93/raw/6bfa15d263d6d5b63840a8e5b64e04b382fdb079/valid-wordle-words.txt'
possible_guess = requests.get(guess_words_url).text.split('\n')
possible_guess = [w.upper() for w in possible_guess if len(w)==5]

answer_words_url = 'https://gist.github.com/cfreshman/a7b776506c73284511034e63af1017ee/raw/845966807347a7b857d53294525263408be967ce/wordle-nyt-answers-alphabetical.txt'
possible_answers = requests.get(answer_words_url).text.split('\n')
possible_answers = [w.upper() for w in possible_answers if len(w)==5]
```

Fig. 1 Defining answer and guess list

```
def wordle(secret, guess):
  secret2 = list(secret)
  guess2 = list(guess)
  pattern = ["","","","",""]
  #Checking green
  for a in range(5):
    if guess2[a] == secret2[a]:
      pattern[a] = "G"
      secret2[a] = "2"
      guess2[a] = "1"
  #Checking yellow
  for b in range(5):
    for c in range(5):
      if guess2[b] == secret2[c]:
        pattern[b] = "Y"
        guess2[b] = "1"
        secret2[c] = "2"
  #Checking black
  for d in range(5):
    if pattern[d] == "":
      pattern[d] = "B"
  newstring = ""
  for ele in pattern:
    newstring += ele
  pattern = newstring
  return pattern
```

Fig. 2 Pattern function

- The function contains 3 for loops. The first for loop checks which letters should be marked green. It simply checks each position of the guess word and sees if the corresponding letter at the same position in the secret word is the same. If they are the same, the list called pattern will contain a 'G' in that same position. It also changes those compared letters to numbers, to ensure that that position is not considered for comparison again.
- The second loop checks for yellow letters, and it actually uses a for loop nested within another for loop. For each letter in the guess word, it compares that letter with each letter in the secret word. If they match, that means that the letter does indeed exist in the secret word, but it isn't in the correct position, so it is marked with a 'Y' in the corresponding position.
- Finally, the third loop checks for black letters, and it simply does this by marking those letters which have not been marked yet as a 'B', because if they are not green or yellow then they must not be in the word at all.
- A variable called pattern is returned as a string.

As stated earlier, the goal is to maximize the information gain from each guess, so for now let's assume that means choosing the guess which cuts down our space of possibilities by the most.

In other words, we choose the guess that results in the smallest list of possible answers left on average.

Performance metric 1: Average size

We need a function that returns the size of the answer list on average when that word is used as a guess. We will use a python dictionary which maps each word to its average size. The code for this function is shown below:

```
46 def AvgSize(possible_guess,possible_answers):
47   SizeDict = {}
48   for guess in possible_guess:
49     avg_size = 0
50     for answer in possible_answers:
51       pattern = wordle(answer,guess)
52       newlist = [word for word in possible_answers if pattern == wordle(word,guess)]
53       avg_size += len(newlist)
54     avg_size = avg_size / len(possible_answers)
55     SizeDict[guess] = avg_size
56     print(SizeDict)
```

Fig. 3 Average size function

The following describes how the function works:

- The function takes two parameters, the list of all possible guesses(possible_guess), and the list of all possible answers(possible_answers).
- We create an empty dictionary called SizeDict, which will map each guess word to its average size.
- For each guess, we iterate through the list of possible answers and compute the pattern that results from that particular guess and answer, on line 51.
- Line 52 is very important and it is executed many times. We iterate through each of the answers and compare its pattern with the pattern from Line 51. If the two patterns match, then we consider that word as a possible answer. Let's take the following example where "WATCH" is the secret word:

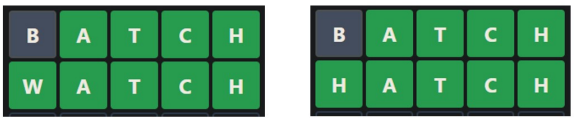

Fig. 4 Examples of wordle patterns

As you can see, the guess "BATCH" results in the same pattern when the secret word is either "WATCH" or "HATCH". This means that "HATCH" would be left remaining as a possible answer given the information we would have gotten if we guessed "BATCH" and got the above pattern. This is because "HATCH" is one of the words that fit this pattern, and can therefore be considered as a possible answer. Any words which do not fit this pattern cannot be considered, and we exclude them from the updated list of answers.

We can then compute the average size for that guess word, add it to the dictionary, and select the word with the lowest average size as our guess.

Although the average size is a good performance metric to determine the information gained from a guess, there is a better one called entropy, which is commonly used in information theory. The reason why entropy is preferred over average size is explained later.

What is Entropy?

Entropy, in basic terms, is the measure of how informative an observation is, and this is measured in a unit called the 'bit'. Let's say that we have an observation that cuts our space of possibilities in half. In that case, it can be said that this observation has 1 bit of information. Given that this observation has 1 bit of information associated with it, the corresponding probability of the observation, p, is equal to ½. This is represented below in Figure 5.

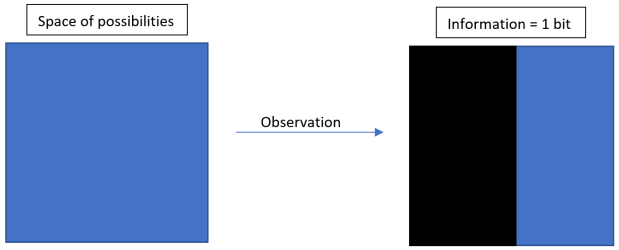

Fig. 5 Entropy of an observation

Similarly, an observation that has 2 bits of information will cut your space of possibilities down into a quarter, and p = 1/4. This is represented in Figure 6.

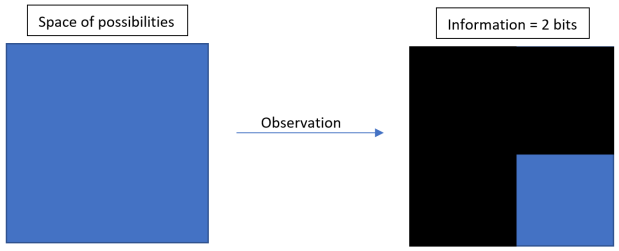

Fig. 6 Entropy of an observation

Using this, we can derive a simple equation for entropy using the following set of equations.

Bits are convenient to use as opposed to probability, firstly because it is much easier to express the amount of information

$$\left(\frac{1}{2}\right)^I = p \longrightarrow 2^I = \frac{1}{p} \longrightarrow I = log_2\left(\frac{1}{p}\right)$$

in terms of bits. For example, saying that an observation has 20 bits of information is easier than saying that the probability of an observation occurring is 0.00000095. Also, similar to how probabilities like to multiply, information likes to add. For example, if the first guess has 3 bits of information, and then the next guess has 2 bits of information, both guesses together have 5 bits of total information gain.

In general, observations which are less likely to occur tend to provide more information. For example, if it was revealed that the secret word contained a 'W', that would cut down the space of possibilities by a lot because there aren't that many 5-letter words that contain a 'W'. On the other hand, the probability that the secret word contains a 'W' is also very low.

Explaining the Formula for Entropy

As you may have seen before, the formula for entropy is given in Figure 7 below:

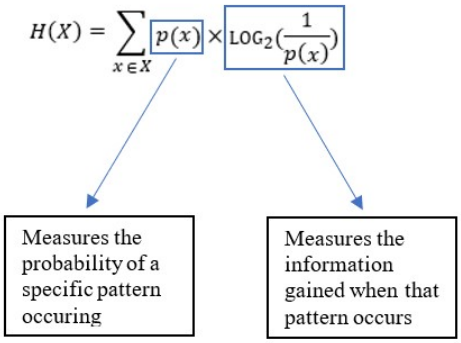

Fig. 7 Formula of entropy

The entropy of a guess is calculated by summing over the product of the probability of a specific pattern occurring and the log of 1/probability. We are essentially taking the weighted average of the information gain for that guess, because we look at the information gain for every possible pattern, and we also look at how likely that pattern is to occur (which is the weightage), and we repeat this process for every pattern until we get the average information gain for that guess, or entropy.

Why entropy is preferred over average size

There are two main reasons why we should use entropy instead of average size:

1. Entropy is a more accurate representation of information gain than average size. This is because average size does not take into account the probability of a pattern occurring, only the information gained when that pattern occurs, meaning that each pattern is weighted equally. This would cause the value to be less accurate because patterns that occur very rarely will still have as much influence on the final value as a pattern which is much more likely to occur, which may lead to the final value being misrepresentative of the actual information gain for a guess. On the other hand, entropy takes into account both the information gain from a pattern as well as the probability that will occur, hence giving a more accurate value of average information gain.
2. Entropy is also a simpler way to represent information gain compared to average size, as discussed earlier.

Performance metric 2: Entropy

We now need a function that will compute the entropy for each guess word given the list of remaining possible answers. The code for this function is shown below:

```
66 def Entropy(possible_guess,possible_answers,score):
67   if score < 2:
68     guess_pool = possible_guess
69   else:
70     guess_pool = possible_answers
71   EntropyDict = {}
72   for guess in guess_pool:
73     PatternDict = {}
74     AvgEntropy = 0
75     for answer in possible_answers:
76       pattern = wordle(answer,guess)
77       if PatternDict.get(pattern) == None:
78         PatternDict[pattern] = 1
79       else:
80         PatternDict[pattern] += 1
81
82     for pattern in PatternDict:
83       occur = PatternDict.get(pattern)
84       prob = occur/len(possible_answers)
85       entropy = prob * math.log((1/prob),2)
86       AvgEntropy += entropy
87
88     EntropyDict[guess] = AvgEntropy
89
90   max_entropy = max(EntropyDict.values())
91   best_guess = GetKey(max_entropy,EntropyDict)
92   return best_guess, EntropyDict
```

Fig. 8 Entropy Function

The following describes how the function works:

- The function takes three parameters: the complete list of valid guesses, the remaining answers, and the number of

guesses which have been played. The reason we need the number of guesses is explained later.

- Depending on the current "score", our guess pool is set to either the complete list of guesses or the list of remaining answers, as shown by the if statement on line 67.
- We then create an empty dictionary called EntropyDict, which maps each guess to its entropy.
- For each guess in the guess_pool, we create an empty dictionary called PatternDict, which stores every unique pattern and the number of times each pattern occurs. We then iterate through the list of remaining answers, and for each answer we compute the pattern for the guess and that answer. If that pattern already exists in the dictionary, we simply increment its value by 1, but if it does not already exist, we add that pattern to the dictionary and set its value to 1.
- Moving on to the second for loop, we iterate through the pattern dictionary, looking at each unique pattern in it. We calculate the probability of that pattern occurring by dividing the total number of occurrences by the total number of answers. We can then calculate the entropy for that pattern by simply using the formula for entropy, and summing all of these entropies up with the variable AvgEntropy. After that, add each guess and its entropy to the entropy dictionary.
- Once the process has been repeated for every guess from our guess pool, we can select the word which has the highest entropy from the entropy dictionary using a function called GetKey(). This function simply returns a key from a dictionary when that key's corresponding value is entered.
- The guess with the highest entropy is our "best guess".

We now have two different performance metrics that we can use, but for testing purposes we will only be using entropy.

Testing

For testing, we need a function that can actually play wordle given a secret word and the list which contains the 2315 possible wordle answers. The code for this function is shown below:

The following describes how the function works:

- The function takes two parameters, a secret word and the complete list of possible wordle answers.
- We play the same opening guess for every round of wordle, that is "TARSE". From using the entropy function earlier, we can determine that "TARSE" is the guess with the highest entropy, and it will always have the same expected information gain on the first guess, which is why we don't need to run the entropy function for the first guess and can instead just use "TARSE" for every round.
- We then update the list of possible answers and set the score to 1.
- Using a Boolean variable called won, which is initially set to false, we can run a while loop until won is set to true, which means the game is over and we have guessed the word.
- We store the outputs of the function Entropy() in an array, and we can then take the guess (which is the best guess) from that array and also get its corresponding entropy from the dictionary.
- The function also checks if the max entropy is 0, because if it is that means no guess can provide any more information, and hence there is only one answer remaining, which is the secret word. Hence, that word is chosen as the next guess.
- If the max entropy is not 0, which it most likely won't be, the guess we got from the entropy function earlier is chosen as the next guess, the answer list is updated and the score is incremented by 1.
- This process is repeated until the guess word matches the secret word, after which the score is returned.

```
123 def PlayWordleMain(secret,possible_answers):
124   pattern = wordle(secret,"TARSE")
125   answers = [word for word in possible_answers if pattern == wordle(word,"TARSE")]
126   score = 1
127   won = False
128   while won == False:
129     EntropyArray = Entropy(possible_guess,answers,score)
130     guess = EntropyArray[0]
131     EntropyDict = EntropyArray[1]
132     max_entropy = EntropyDict[guess]
133     if max_entropy == 0:
134       guess = answers[0]
135       score += 1
136     else:
137       pattern = wordle(secret,guess)
138       answers = [word for word in answers if pattern == wordle(word,guess)]
139       score += 1
140     if guess == secret:
141         won = True
142   return score
```

Fig. 9 Function to Play Wordle

After creating the function, we need some more code to actually run the function as well as picking the secret word and keeping track of average score. The code for this is shown below:

The following describes what the code does:

- Firstly, we need to import the random library so that we can select a random word from the 2315 answers.

```
import random
TotalScore = 0
for i in range(2315):
  x = random.randint(0,2314)
  RandWord = possible_answers[x]
  print(RandWord)
  Score = PlayWordleMain(RandWord,possible_answers)
  TotalScore += Score
AvgScore = TotalScore/2315
print(AvgScore)
```

Fig. 10 Testing the algorithm

- We create a random integer x by using the random.randint() to select a random number between 0 and 2314.
- We use this random integer to index the list which contains all of the answers, thus selecting a random word.
- We then store the result of the PlayWordleMain() function which was shown earlier, in a variable called Score.
- Use the variable called TotalScore to keep a running total of the score, and then divide TotalScore by the number of answers tested to get the average score.

There are actually two versions of the algorithm:

- In the entropy function from the first version, all guesses are made using the complete list of valid guesses.
- In the entropy function from the second version, the guess pool is determined based on the current score in the game, choosing a guess from either the complete list of guesses or just from the possible answers. Although this may sacrifice some amount of entropy, this is done to increase the probability of guessing the secret word, while still narrowing down the list of possible answers.

Tests were conducted for both of these versions, and the results are shown in the next section.

Results

Version 1:
Average number of guesses: 3.594
Success rate: 100%
Total number of secret words tested: 100
Version 2:
Average number of guesses: 3.451
Success rate: 99.7%
Total number of secret words tested: 2315
Along with these two tests, an additional test was done where the algorithm cannot distinguish between the set of words that

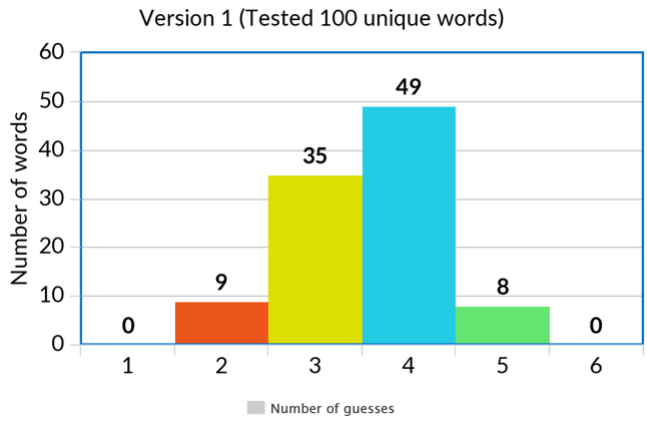

Fig. 11 Histogram for version 1

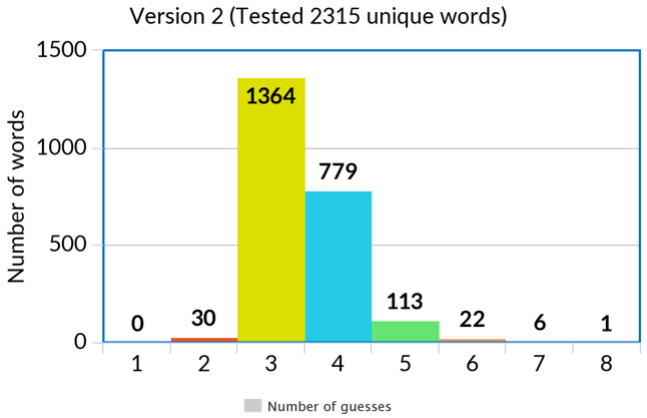

Fig. 12 Histogram for version 2

are possible answers and the total set of valid guess words, and therefore considers each valid guess as a possible answer as well. This makes the starting entropy of the 12972-word list 13.66 bits whereas for the previous tests the starting entropy was 11.17 bits. The average number of guesses for this test was 4.14 guesses with a success rate of 100%, tested for 100 unique words.

Discussion

As we can see from the results, Version 2 performs better than Version 1 by about 0.15 guesses on average, with the only difference between the two versions being that Version starts selecting guesses from only the remaining answers after the first two guesses are done. Essentially, it maximizes information gain in the first two guesses, and then tries to go for gold with the third guess while sacrificing some information gain. On the

other hand, version 1 maximizes information gain throughout, and secures the secret word within 6 guesses 100% of the time, compared to Version 2 which fails 7 out of the 2315 times.

One disadvantage of Version 2 is that it struggles to guess certain words which have other words that are very similar to it. One example of such a word is "JOLLY". There are 4 other words which are very similar to it which are "DOLLY", "FOLLY", "GOLLY", and "HOLLY". When the algorithm comes across a word like this, what ends up happening is that it will guess all of these words in sequence until it finally reaches "JOLLY", which increases the number of guesses on average. Version 1, however, would use a word from the valid guess list which may eliminate a lot of these possibilities and it can hence guess the word in 2, maybe 3 guesses. A few suggestions for improvements that could make the algorithm even better:

- Search for expected information gain two steps forward, rather than just one. Essentially, find the combination of two words which gives the most expected information gain.
- Instead of switching the guess pool from valid guesses to valid answers at the same point during each round, evaluate a score for each word with weightages assigned to entropy as well as the likelihood that that word is the answer. Automate the process of preferring either information gain or probability of success.

Conclusion

In conclusion, using entropy does seem to be the best approach to solving Wordle optimally in terms of the average score, beating out other strategies like rank one approximation or reinforcement learning significantly, and the original algorithm that I wrote does succeed in solving the game quite optimally, having around the same performance as 3Blue1Brown's algorithm[4], who managed to get 3.42 guesses on average. I hope that this paper can be useful for anyone who is willing to learn about information theory, or wants to make their own wordle algorithm. While this is a small step in applying information theory to solving word games such as Wordle, I intend to explore better solutions based on the insights from this study. We also hope others will be encouraged to study the potential applications of information theory to other problems.

Acknowledgments

I would like to thank Joshua Whitman, University of Illinois at Urbana-Champaign, for his guidance on this research work.

References

1 https://www.nytimes.com/games/wordle/index.html

1. De Silva, Nisansa. "Selecting optimum seed words for Wordle using character statistics." 2022 Moratuwa Engineering Research Conference (MERCon), 2022, https://doi.org/10.1109/mercon55799.2022.9906176.
2. Bonthron, Michael. Rank One Approximation as a Strategy for Wordle, 11 Apr. 2022.
3. Solving Wordle Using Information Theory, YouTube, 6 Feb. 2022, Wordle.
4. Liu, Chao-Lin. "Using wordle for learning to design and compare strategies." 2022 IEEE Conference on Games (CoG), 2022, https://doi.org/10.1109/cog51982.2022.9893585.
5. Meyer, Jesse, and Benton J Anderson. Finding the Optimal Human Strategy for Wordle Using Maximum Correct Letter Probabilities and Reinforcement Learning, 1 Feb. 2022.
6. Bhambri, Siddhant, et al. Reinforcement Learning Methods for Wordle: A POMDP/Adaptive Control Approach, 29 Nov. 2022.
7. https://colab.research.google.com/drive/1QIVpRAfYnalI2xnSEWgwTAdNPK0xrcVX