PROJECT REPORT
WORDLE GAME

Course: Introduction to Artificial Intelligence - Code: IT3160E

Instructor: Prof. Nguyen Nhat Quang

Fall 2022

Participants:

Nguyen Ba Thiem 20214931

Pham Quang Tung 20210919

Doan The Vinh 20210940

I. Practical problem description

THE WORDLE GAME

Overview

Wordle is a word game (owned by The New York Times Company), where players are given six attempts to guess a five-letter word. For each guess, the game returns a feedback in the form of colored tiles. Specifically,
A green tile suggests a correct letter (a letter in the answer) and in the right position.
A yellow tile suggests a correct letter in the wrong position.
A gray tile suggests a wrong letter (a letter not in the answer).

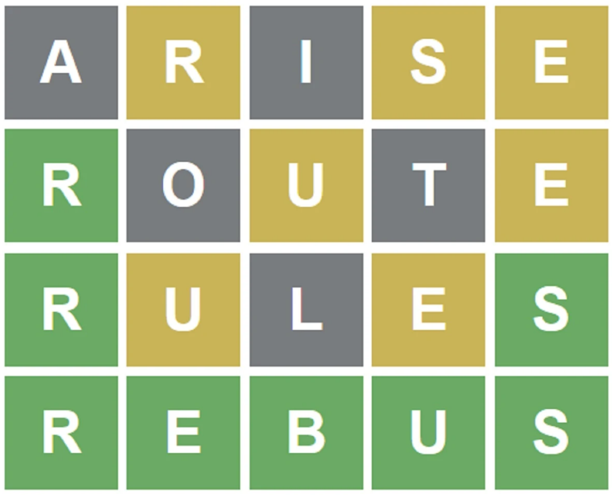

Figure 1. Wordle gameplay example

Requirement

The player of Wordle wins if one takes no more than six guesses to reach the secret answer. In addition, the player cannot just put anything as a guess—from the source code, we learn there is a list of approximately 13,000 meaningful words that the game accepts.
However, the majority of these 13,000 words are unheard-of and rarely used in real life. As a result, Wordle keeps a smaller human-curated list of possible answers (approximately 2,300 words), which are at least somewhat common words.
Wordle is played in two modes: Normal mode and Hard mode.
In Normal mode, players can use any of the 13,000 words to guess, regardless of previous feedback.
In Hard mode, any revealed hints must be used in subsequent guesses. This means players can only use a word with green letters in the correct positions and identified yellow letters.

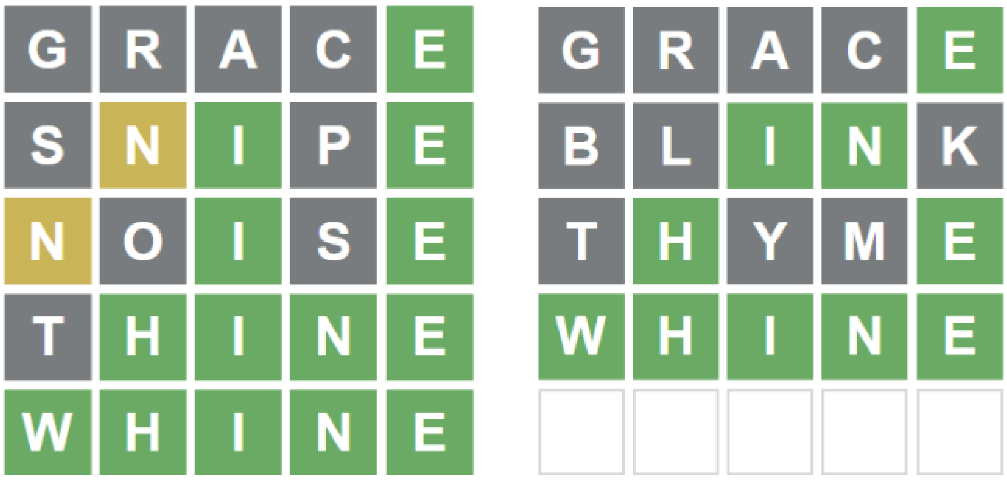

Figure 2. Hard mode (left) and Normal mode (right)

THE WORDLEBOT

Target

We wish to devise an algorithm that assists people in playing Wordle. Not only would we want to maximize the win rate, or the chance of needing no more than six guesses, when this win rate is satisfactory (near perfect or 100%), we would also want to minimize the number of guesses needed.

Of course, in the official game by the New York Times, the list of approximately 2,300 possible answers are made publicly available, and theoretically, we can improve performance by limiting our algorithms to only work with this list. However, we believe that our algorithm's goal should be to play following the same rules as a human player, which means not knowing which words can be a solution.

Thus, the algorithms below will work with the entire list of approximately 13,000 words to recommend guesses. We can use the list of permitted guesses in our algorithms because entering words that do not exist has no consequences for the game. Failed attempts to make a permitted guess are not punished by Wordle's rules; instead, the game just ignores the guesses until a valid one is entered. However, we will use the 2,300-word list for performance testing.

Input - Output

Below, we will provide two interfaces for utilizing the Wordle algorithms.

The first is an interface where the player gets recommendations for each guess. Then, in the actual game or an available simulated game (i.e. on the official New York Times site), the

player puts in a guess and receives a feedback. The player then feeds the guess and the subsequent feedback to the algorithm, which will then recommend words for the next guess.
The second is an interface where the player gives the algorithm a word to guess. Then, the algorithm simulates playing a game of Wordle from scratch, displaying the words it has chosen for each guess.

Expected outcome

According to surveys, for most human players, a four-guess gameplay success is standard, while a three-guess success is often thought of as brilliant and lucky. For developers, preliminary algorithms often require a bit more than four guesses, and further refinements may further reduce this number. This is the outcome we wish to obtain. We will also see why this number may never reach as low as three.

Data set

We use the list of 13,000 allowed words and 2,300 possible guesses for the Wordle game on the official New York Times site.
We also obtain data on word frequency from the Google Books English n-gram public dataset.
References to these are found in the list of bibliographic references.

Resources used

We use the *colorama* module in Python in order to print out color in the terminal.
We also use common modules: *sys*, *os*, and *json* in Python to help deal with the data sets and develop a functional interface
We use *np*, *math* and *random* in Python to assist in calculations, as well as *time* and *matplotlib* to help visualize testing results and calculate time needed for performance testing.
Since the game is new, existing resources (i.e. libraries, software packages, codes) are few and often the work of individual amateur developers. Thus, the code for the below algorithms, as well as the interactive and simulation interfaces mentioned above, are our own work.

II. Algorithm details

THE NAIVE APPROACH

Description

For the average human player, Wordle essentially boils down to using the feedbacks to eliminate words and shrink down the list of words that might still be the answer. From here on out, this list shall be called a "guess space"—the original guess space will contain approximately 13,000 words. This is also the intuition behind the naive approach.

At a given moment, the algorithm picks a random word from a guess space and receives a feedback. It then uses this feedback to shrink down the guess space. The process repeats until the algorithm reaches a guess space with only one word.

By picking only in the current guess space, this algorithm automatically plays on Hard mode.

Pseudocode

```
def naive_approach(words_list,answer):
    i = 0
    valid_set = words_list
    win = False
    while not win:
        if i == 0:
            guess = random(words_list)
        else:
            feedback = get_feedback(guess,answer)
            valid_set = reduce_word_list(valid_set,feedback)
            guess = random(valid_set)
            if check_win(feedback) == True:
                break
```

GREEDY ALGORITHM WITH LETTER FREQUENCY-BASED HEURISTIC

Greedy algorithm

An informed search method known as a greedy algorithm takes the best decision at each stage with the intention of eventually producing a globally optimum solution. In order to define this "best decision", a greedy algorithm utilizes a heuristic in order to rank possible choices. As the greedy algorithm decides without regard for consequences, final results may be suboptimal.

Description

For a guess, the naive approach selects a random word from the current guess space. However, human intuition suggests that, for example, "pzazz", though indeed an acceptable word, is a worse guess than "grace", because the former contains repeated and uncommon letters.

We try to capture this idea by devising a score to quantify the suitability of a word used as a guess. From the original guess space, we deduce the frequency of each letter as it appears among the 13,000 words. Then, using this, each of the 13,000 words get assigned a score. As a result, before each guess, from the current guess space, the algorithm recommends the words with the highest scores.

Specifically, the score is computed by summing up the frequencies of the distinct letters in a word. For example, Score('grace') = freq('g') + freq('r') + freq('a') + freq('c') + freq('e'), while Score('eerie') = freq('e') + freq('r') + freq('i'). By this, words with common letters and without repetition are preferred. For words

By picking only in the current guess space, this algorithm automatically plays on Hard mode.

Pseudocode

```
def letter_frequency(word_list,answer):
        i = 0
        valid_set = word_list
        import score_based_on_letter_frequency_list
        win = False
        while not win:
                if i == 1:
                        guess = the_word_having_highest_score_in_word_list
                else:
                        feedback = get_feedback(guess,answer)
                        valid_set = reduce_word_list(valid_set, feedback)
                        guess = the_word_having_highest_score_in_valid_set
                        if check_win(feedback) == True:
                                break
```

GREEDY ALGORITHM WITH ENTROPY-BASED HEURISTIC

Description

We will see that the heuristic that is based on letter frequency helps improve the algorithm's performance compared with the naive approach (see Section III). However, this criteria may be suboptimal. In this specific case, we cannot rate the difference between words with the same letters but in different order, such as "nails" and "snail". The same can be said about some other different criteria, such as the number of vowels.
However, these approaches share one target—intuitively, they try to prefer words that have the potential to provide the most information. For instance, the above algorithm replicates the human feeling of wanting a colored feedback, since some people think that these tend to be very informative in terms of shrinking a guess space.
Thus, we try to mathematically quantify the expected amount of information given by a guess, given a particular guess space at some point during gameplay.

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.

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

The standard unit of information is bit. If we have an observation (in our case, a pattern) that cuts the space of possibilities in half, we say that it has one bit of information. If instead a new fact chops down that space of possibilities by a factor of four, we say that it has two bits of information. So the formula of the number of bits in term of the probability of an occurrence is

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

From that, we obtain a definition: Let X be a discrete random variable and probability mass function $p(x) = Pr(X = x)$.Then the entropy of X is defined as

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

Entropy can be understood as the expected amount of information of a random variable.

Application of entropy to problem

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 is returned shrinks down the guess space i.e. words that are incompatible with the pattern should be eliminated. 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, assuming a uniform distribution over the set of allowed words. The sample space initially is the allowed_words and then it is partitioned by the pattern we receive at each turn of play.

Thus, at each turn, we compute entropy for all words in the guess space (i.e. words that can still be the answer given the previous feedbacks). Afterwards, we rank them in decreasing order and the greedy algorithm picks the word with the highest entropy without considering future consequences. This will be discussed in detail in Section III.

Hard mode

In Hard mode, the player can only select a guess from the guess space at a particular point during gameplay. (This is not actually true, since Hard mode still allows the reusing of known gray letters, and a word with known gray letters cannot be in the current guess space. However, reusing gray letters offers no new information, so even if we compute the entropy for these words outside the guess space, it will be lower than that of the words in the guess space).
As a result, to recommend words, the algorithm **finds the maximum entropy among words still in the guess space**.

Normal mode

In Normal mode, the player can select any of the 13,000 accepted words, regardless of previous feedback. To recommend words, the algorithm then **finds the maximum entropy among the original guess space of 13,000 words**.
This is because it is very likely that at a point during gameplay, there exists a word outside the guess space that offers more information, i.e. has higher entropy, in terms of reducing the current guess space.
To illustrate, suppose the player knows that the answer is "_atch". The gray letter can be 'p', 'r', 'n', or 'w'. In Hard mode, the player must use at most four guesses - "patch", "ratch", "natch", and "watch" - to find the missing letter (note how having to repeat the last four

letter gives no information), while in Normal mode, the player can use a single guess "prawn" to know which of the four letters is correct.
It is easily seen that for the first guess, since the algorithm computes entropy for each of the 13,000 words, based on the original guess space, the algorithm will always start with "tares", regardless of game mode.

Pseudocode

```
def entropy_based(word_list,answer):
    i=0
    valid_set=word_list[:]
    win=False
    while not win:
        if normalMode:
            entropy_list = entropy(word_list) #entire 13,000-word list
        if hardMode:
            entropy_list = entropy(valid_set) #only the current guess space
        if i==0:
            guess = max(entropy_list)
        else:
            feedback = get_feedback(guess,answer)
            valid_set = reduce_word_list(valid_set,feedback)
            guess = the_word_having_highest_entropy_in_valid_set
            if check_win(feedback) == True:
                break
```

GREEDY ALGORITHM WITH ENTROPY-BASED AND WORD FREQUENCY-BASED HEURISTIC

Description

In the previous algorithm, we assume that every word is equally likely and then use that assumption to compute entropy of each word. We want to incorporate the word frequencies into the calculation of entropy to try to create an updated version giving better performance. The question is: how do we know the probability of occurrence of a particular pattern, respective to a word in sample space?

We will assign each word a probability that a word can be the answer of the game. We want to determine words that can be the answer or not. The problem is how likely a word can be the final answer shouldn't be proportional to the relative frequency. For example, 'which' has frequency 0.002 and 'braid' has frequency 0.000002 but these two words are common enough that the chance of them being the answer should be relatively the same. Hence, we want a binary cutoff. So we propose the idea to place the sorted list of frequencies into the x-axis over a designated length and pass them through a modified Sigmoid function.

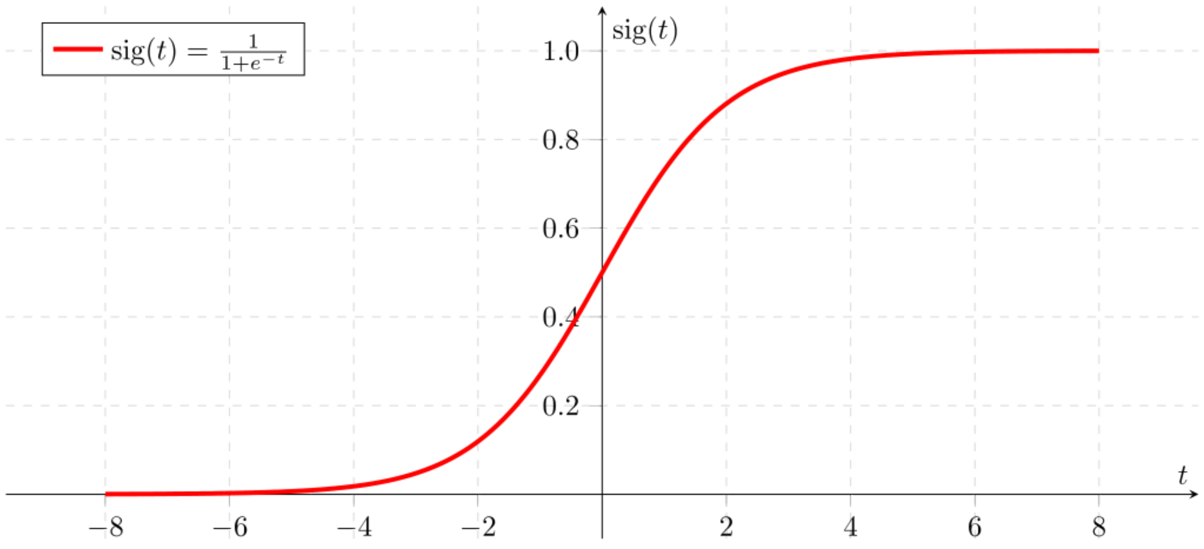

Figure 3. Graph of Sigmoid function

We will let the space under Sigmoid be 10. We make the arbitrary decision that about 3000 out of 12972 words should be common enough to be the answer. So we arrange that list of frequencies on the x-axis such that 3000 words have the sigmoid value being greater than 0.5 by determine a value:

$$c = 10(-0.5 + \frac{3000}{12972})$$

And we will divide the interval $[c-5; c+5]$ into 12972 small equal intervals and use that to pass through to the Sigmoid function which returns the probability of how likely the word will be the answer.

We know that sample space initially is the allowed_words and is partitioned after each turn of play. Now, each word in sample space is assigned with a relative probability of being the final answer. We will assign each pattern the probability of occurring by taking the sum of probabilities of the words left in the guess space to divide by the sum of probabilities by dividing each element the sum of all probabilities of all words. Then we have a refined probability of a pattern occurring.

Now, we can calculate the entropy incorporating the frequencies and at each turn, we choose the word having highest entropy incorporating the frequencies and redo this process.

Pseudocode

```
def entropy_based(word_list,answer):
    i=0
    valid_set=word_list[:]
    win=False
    while not win:
        import word_frequency
        if normalMode:
        #entire 13,000-word list, applying Sigmoid, refining probability of pattern
            entropy_list = entropy_based_on_freq(word_list)
        if hardMode:
        #only current guess space, applying Sigmoid, refining probability of pattern
            entropy_list = entropy_based_on_freq(valid_set)
        if i==0:
            guess = max(entropy_list)
        else:
            feedback = get_feedback(guess,answer)
            valid_set = reduce_word_list(valid_set,feedback)
            guess = the_word_having_highest_entropy_in_valid_set
            if check_win(feedback) == True:
                break
```

A* ALGORITHM WITH ENTROPY-BASED AND WORD FREQUENCY-BASED HEURISTIC

A* algorithm

An informed search method known as an A* algorithm also takes the best decision at each stage with the intention of eventually producing a globally optimum solution. In order to define this "best decision", a greedy algorithm utilizes a value in order to rank possible choices. Specifically, this value **f** is a parameter equal to the sum of two other parameters **g** and **h**. **g** represents the cost the algorithm has accumulated to get to a particular state, while **h** is the estimated cost to reach the destination. This is often referred to as a heuristic. Finding and refining this h function is crucial. Generally, A* algorithm works better (intuitively, h represents looking out for future consequences, something that a greedy algorithm does not do), but may still be suboptimal.

Description

We want to calculate the average number of guesses needed when we input a particular guess. To approach this, we simulate approximately 2,300 Wordle games with the previous algorithm (entropy-based and word frequency-based) and collect data of these games by storing the number of bits left of the current guess space and the number of guesses to reach the answer from that point. For example, we only need one more guess if the number of bits left in the guess space is 0, but we may need one or two more guesses if the number of bits left is 1 bit.

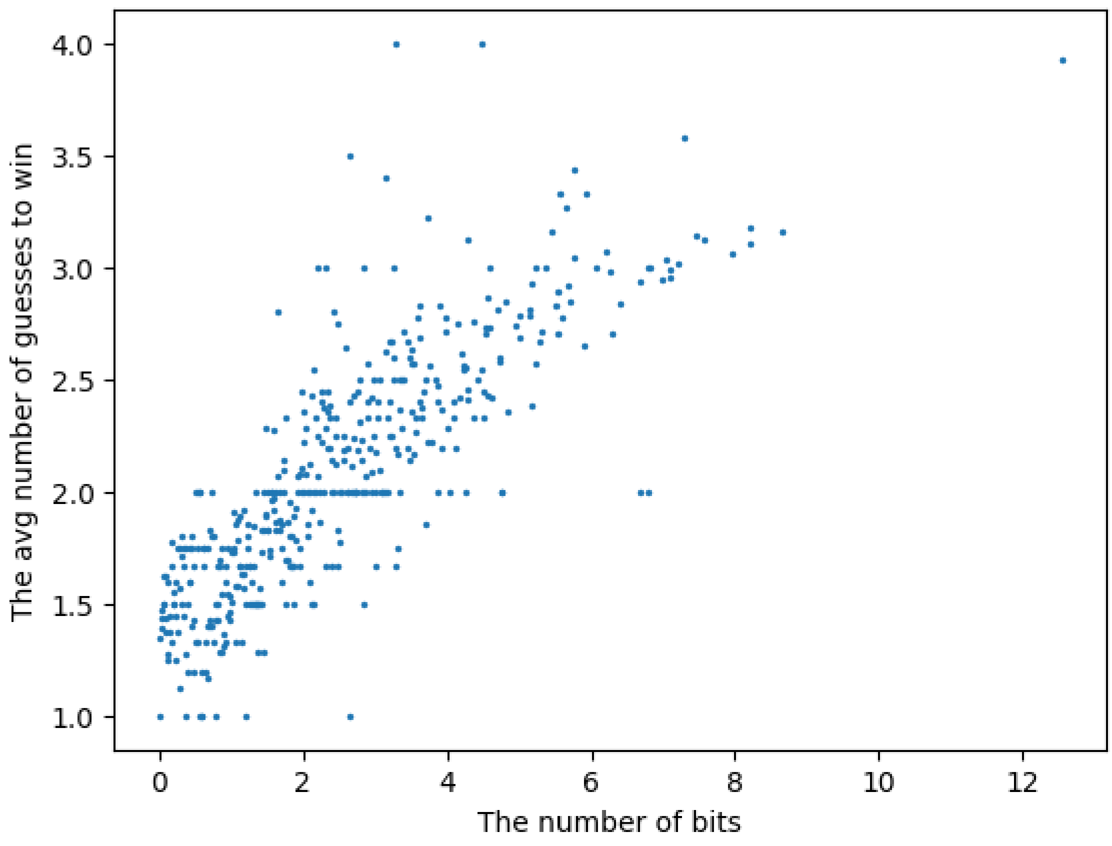

Figure 4. Graph of data collected from the simulation of 2,300 games, using the greedy algorithm with entropy-based and word frequency-based heuristic

The bits left in a word list where each word have a particular probability of being the answer is:

$$I(space) = \sum p.log_2(p) \text{ for } p(word) = \frac{frequency(word)}{\sum_{space} frequency}$$

We will use Gradient Descent to determine a function **f** predicting the necessary number of guesses from the bits of sample space. Hence, the function **f** is the heuristic function in our A* algorithm that estimates the number of guesses from a sample space to the answer.

Then, we obtain the formula of average value of score of a guess:

$$E[Score] = p(word).x + (1 - p(word)).(x + f(H0 - H1))$$

,where $p(word)$ is the probability of being the answer of that guess, **x** is the number of guesses, **H0** is the entropy of the previous guess space, and **H1** is the actual bits of information obtained from the previous guess.

The first guess is still “tares”, according to the previous algorithm, since we cannot use the above formula to find the best word for the first guess. From the second guess, we will calculate the score of words in the reduced guess space and choose the word having the lowest score.

III. Implementation details

TEST PERFORMANCE EXPLANATION AND ANALYSIS

The naive approach

We test the performance of the naive approach by randomly selecting a word from the official list of approximately 2,300 possible answers and recording the number of guesses needed. Since the algorithm is not deterministic–given a current guess space, the algorithm picks a random word, we repeat this 100,000 times.

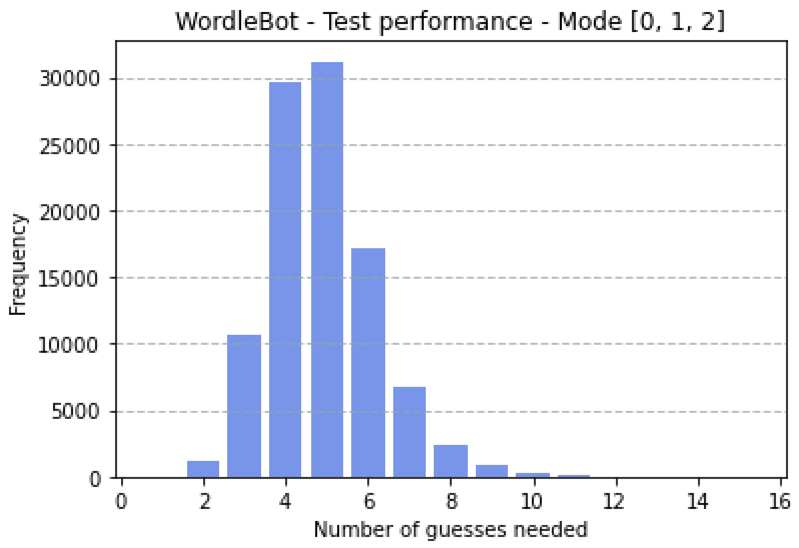

Figure 5. Guess distribution for naive approach - ***<u>Average: 4.89</u>*** *-* ***<u>Win rate: 89.47%</u>***

We conclude that this average performance is worse than that of an average human player. Thus, we try to inform the search by selecting a heuristic, as explained in Section II.

Greedy algorithm with letter-frequency based heuristic

We test the performance of the greedy algorithm with letter-frequency based criteria by testing it on the official list of approximately 2,300 possible answers. For testing, each word in this list only needs to be used as the secret word once, since the algorithm is deterministic–given a current guess space, the algorithm will only pick the word with the highest score. Computation of this score is described in Section II.

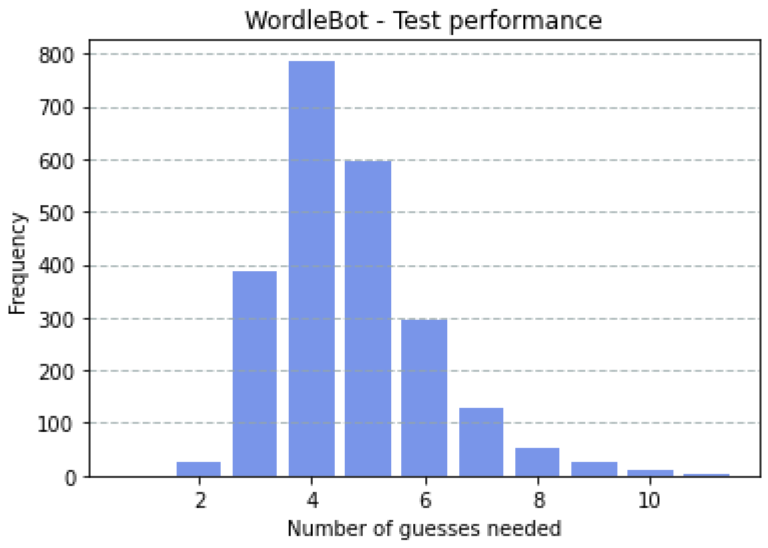

Figure 6. Guess distribution for greedy algorithm with letter frequency-based heuristic - ***Average: 4.67*** *-* ***Win rate: 91.13%***

We can conclude that by informing our search with a heuristic, the algorithm's performance has improved. However, the performance is still not up to that of the average human (around 4); thus, we aim to select a "better" heuristic, one that encapsulates the idea of a guess having the potential to provide a quantifiable amount of information. Details are in Section II.

Greedy algorithm with entropy-based heuristic

We test the performance of the greedy algorithm with entropy-based heuristic by testing it on the official list of approximately 2,300 possible answers. As with the previous algorithm, this is deterministic. Computation of entropy is described in Section II.
As opposed to the previous two approaches, using entropy means that we do not necessarily have to play in Hard mode, i.e. a guess does not necessarily have to come from the current guess space. As such, we present the performance of this algorithm when we play in both Hard mode and Normal mode.

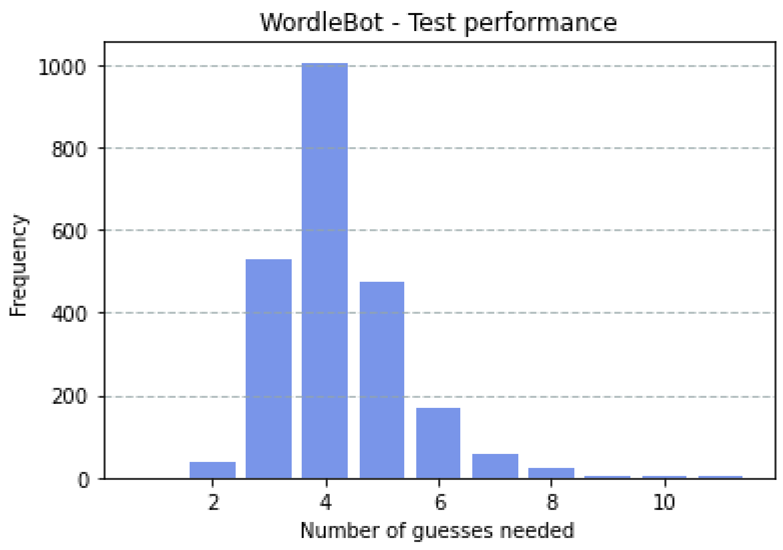

Figure 7. Guess distribution for greedy algorithm with entropy-based heuristic, played in Hard mode - ***Average: 4.23*** *-* ***Win rate: 96.25%***

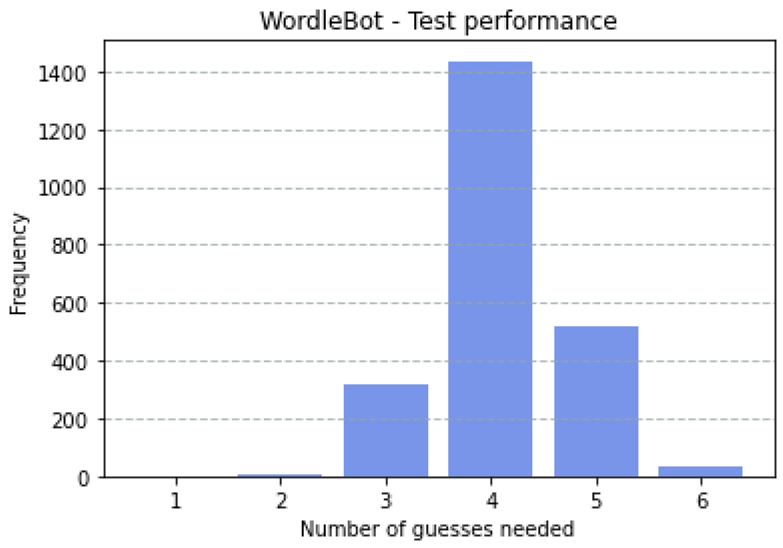

Figure 8. Guess distribution for greedy algorithm with entropy-based heuristic, played in Normal mode - ***Average: 4.11*** *-* ***Win rate: 100%***

True to the theory on entropy, from the results, we can conclude that by informing our search with an entropy-based heuristic, the algorithm's performance has improved significantly, from an average of 4.67 to 4.23, when we're strictly considering Hard mode. When the algorithm plays in Normal mode, the win rate reaches 100%, suggesting that this perfect rate is plausible. We then aim to further reduce the average number of guesses needed.

These results also confirm that performance in Hard mode tends to be worse than in Normal mode. Specifically, the former distribution has more variance—for example, there are more words guessed with just two turns in Hard mode than in Normal mode (38 vs. 4), but on the contrary, there is also a larger number of words needing six turns than in Normal mode (170 vs. 34).

In addition, according to "Gómez, Á. B., & Puga, E. C. (2022). *Wordle solving algorithms using Information Theory* (dissertation)", this same approach produces an even better average performance when the starting word is not “tares” (highest entropy) but rather “tales” (top 8 highest). This goes to show how the greedy algorithm, as it selects the best choice at each stage, may end up being suboptimal altogether.

Greedy algorithm with entropy-based and word frequency-based heuristic

We test the performance of the greedy algorithm with entropy-based and word frequency-based heuristic by testing it on the official list of approximately 2,300 possible answers. As with the previous algorithm, this is deterministic. Changes in computation, compared with the previous algorithm, are described in Section II.
Again, using entropy means that we do not necessarily have to play in Hard mode. As such, we present the performance of this algorithm when we play in both Hard mode and Normal mode.

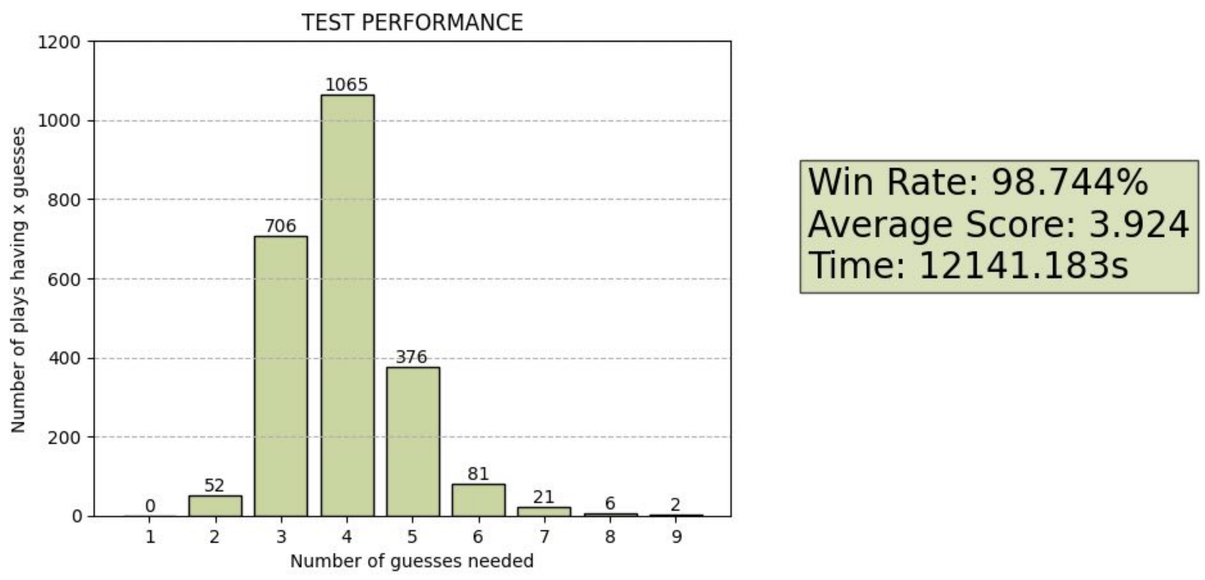

Figure 9. Guess distribution for greedy algorithm with entropy-based and word frequency-based heuristic, played in Hard *mode -* ***Average: 3.92*** *-* ***Win rate: 98.74%***

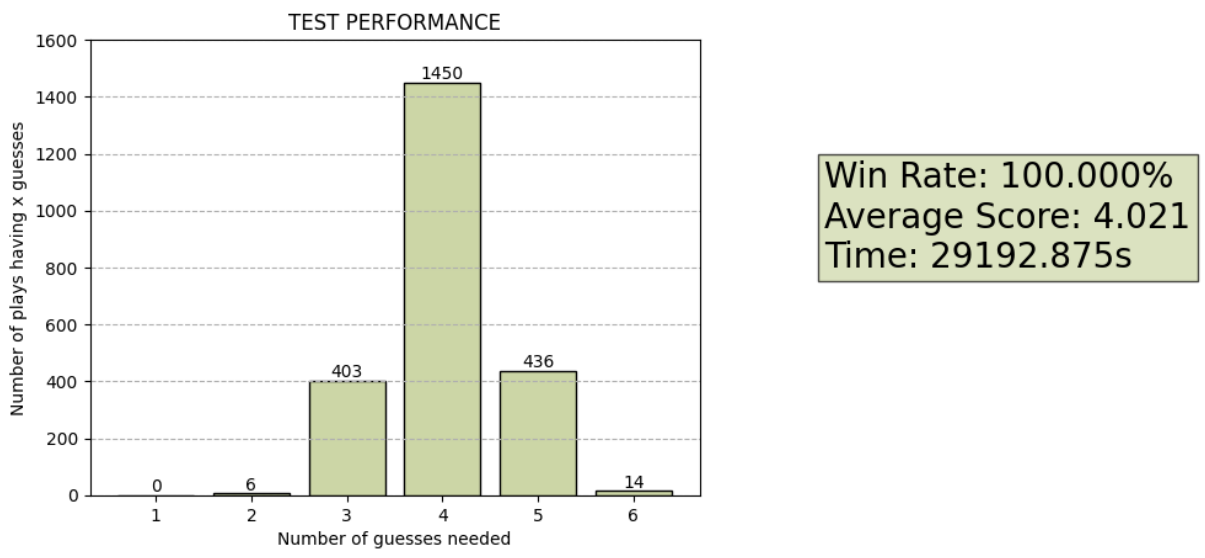

Figure 10. Guess distribution for greedy algorithm with entropy-based and word frequency-based heuristic, played in <u>*Normal*</u> *mode -* ***Average: 4.02*** *-* ***Win rate: 100%***

Compared with the previous heuristic, incorporating word frequency, which would change the computation of entropy, seems to further help improve performance. When strictly considering Hard mode, we see that the win rate has come up from 96.25% to 98.74% and the average number of guesses needed has fallen to 3.92; this also suggests that the algorithm has a less varied performance across the test set. For the Normal mode, the win rate is kept at 100%, and the average number of guesses needed has dropped from 4.11 to 4.02.

We also note that the average performance of Hard mode is a bit better than of Normal mode (3.92 vs. 4.02). This seems to suggest with a refined heuristic, the trade off between a higher entropy (at one point, a word outside the guess space may have higher entropy) and the chance of being an answer (at one point, only a word inside the guess space may be the answer) over-favors the former. Thus, we wish to further incorporate word frequency, from just using it to modify entropy calculation, to using it as a direct parameter in our heuristic.

A* algorithm with entropy-based and word frequency-based heuristic

We test the performance of the A* algorithm with entropy-based and word frequency-based heuristic by testing it on the official list of approximately 2,300 possible answers. As with the previous algorithm, this is deterministic. Computation and use of the heuristic function is described in Section II.

Again, using entropy means that we do not necessarily have to play in Hard mode. However, since the previous algorithms can already achieve the win rate 100% in Normal mode, we only work with hard mode in this version to avoid wasting time.

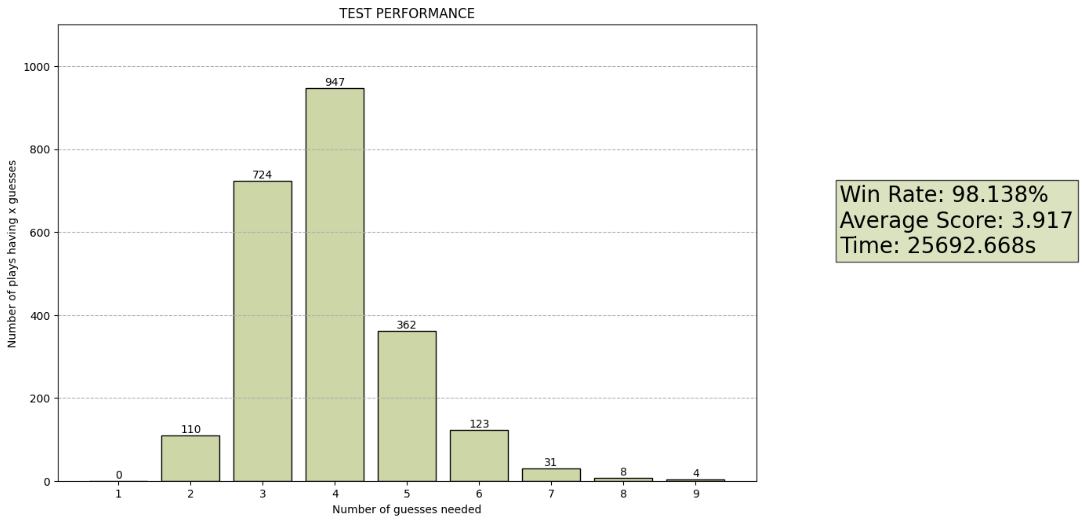

Figure 11. Guess distribution for A algorithm, played in Hard mode - **Average: 3.91** - **Win rate: 98.1%***

From the statistics, we see that this algorithm has a slightly better average score (3.924 down to 3.917) but it has a lower win rate than the previous algorithm. From here, we have decided that in the future, for the A* algorithm, we would like to find a better heuristic function to describe the number of guesses.

DIFFICULTIES

Time-consuming performance testing

While completing the project, one of the most time-consuming parts was testing the algorithm's performance, especially when we resort to entropy-based and word frequency-based heuristic. This is because to compute entropy for one guess, even without word frequency, the algorithm has to parse through the entire list (denoted L) of 13,000 words, getting the feedback for each of L used as an answer. This means that calculating entropy for an entire guess space (denoted G) requires obtaining a feedback **||G|| * ||L||** times, and across multiple turns in a game, this number stacks up. Fetching word frequency complicates things even further.
We also observe that the most time-consuming subtask is to compute the entropy for 13,000 words, based on 13,000 words, for the first guess. Since this computation should be the same for every gameplay, we pre-compute this, store it in a *json* or *csv* file, and fetch it when needed.

Few existing resources, source codes, software packages for interfaces

As this is a fairly new game, most of the existing coding resources are works by amateur individuals. But as stated in the bibliography, there are indeed videos, blog posts, and papers that focus on the theoretical side of this problem. Therefore, we have opted to build the source codes from scratch, including the algorithm and the interface. As inexperienced Python learners, the source codes can be disorganized, but throughout the project, we have also revised the basics of programming: functions, moduling, packaging. Also, because this is a collaborative project, we have also learned the use of git and GitHub for version control.

New knowledge and theories

The majority of this project relies on the concepts in Information Theory, specially the idea of entropy. Though we are able to obtain a fundamental understanding from videos and scientific articles, delving deep into the theory and applying the idea into this specific problem can be difficult. To deal with this problem, we try to find even more materials to research on, as well as finding details that relate directly to the Intro to AI course. This includes concepts on informed search and fuzzy sets.

PROPOSALS AND POSSIBLE EXTENSIONS

Problem relaxation

If we have more time, we would also like to let the algorithms work with the list of 2,300 words only. Though at the beginning, we have set a limitation saying that the algorithm should only know about the 13,000-word list, and the list of possible answers should only be used for performance testing, we believe that only working with 2,300 words can help us learn about the experimental lower bound for all possible algorithms. According to 3Blue1Brown's Grant Sanderson's work, as well as Gómez, Á. B., & Puga, E. C. (2022). *Wordle solving algorithms using Information Theory* (dissertation), from experiment, 3.42 seems to be the lowest possible average performance.

Arbitrary choices

In addition, throughout the project, we have made arbitrary choices in a number of places in the algorithms. Before utilizing entropy, using letter frequency is a choice based on our intuition. For example, letter frequency, relative to position in a word, is also a heuristic that we may have chosen. We wish to see how these heuristics perform, in comparison with one another and with entropy.

Another arbitrary choice is the choosing of a sigmoid function. To explain, when using word frequency to calculate the probability of a word being an answer, we determine a length on the x-axis to place the words and determine a cutoff point to isolate words that might be more likely (probability > 0.5) to be the answer. At present, length is fixed at 10, and 3000 words with the highest frequency are deemed "more likely". This is related to the concept of a fuzzy set with a membership function, and we want to know if we can utilize operations on fuzzy sets to further refine our heuristic.

Another arbitrary choice is the sigmoid function itself. This function is used to replicate the idea of a binary cutoff between “this word is in the 2,300-word possible list” and “this word is not”. There are other functions which take in a domain and return values that are either close to 0 or close to 1. We wish to experiment with different functions to refine our heuristic.

Other algorithms and implementation

The dissertation by Gómez, Á. B., & Puga, E. C. (2022) also utilizes the concept of genetic algorithm, something we have not had time to consider and implement. In addition, some individual work, notably by Games Computers Play (YouTube) also aims to build a tree to optimize time complexity. We would like to visit these ideas in the future

IV. List of tasks

Task	Subtasks	Participants - Percentage of contribution		
		Nguyễn Bá Thiêm	Phạm Quang Tùng	Đoàn Thế Vinh
Discuss & Research	Finding available materials (articles, papers, videos, etc.)	33%	33%	33%
Source code	For naive approach	50%	-	50%
	For greedy algorithm with letter frequency-based heuristic	-	-	100%
	For greedy algorithm with entropy-based heuristic	50%	-	50%
	For greedy algorithm with entropy-based and word frequency-based heuristic	50%	50%	-
	For A* algorithm with entropy-based and word frequency-based heuristic	-	100%	-
Interface	Interactive mode	50%	-	50%
	Simulation mode	50%	-	50%
Compiling files	Working with repositories and packaging files	100%	-	-
Analysis of results	Discussion on test performance implementations and results	33%	33%	33%
Report	Section I. Practical problem description	-	-	100%
	Section II. Algorithm details	-	50%	50%
	Section III. Implementation details	-	50%	50%
Presentati on slides	Preparing slides, demo video, images, scripts	33%	33%	33%

V. List of bibliographic references

Sites to play Wordle can be found in these links.

The New York Times. (n.d.). *Wordle - a daily word game*. The New York Times. Retrieved December 13, 2023, from https://www.nytimes.com/games/wordle/index.html

Wordle game: Guess the hidden word. Wordle Game - Play Unlimited. (n.d.). Retrieved January 2, 2023, from https://wordlegame.org/?seed=20230101

As stated in Section I, we use the following data sets for our algorithms.

3b1b. (2022, February 20). *VIDEOS/ALLOWED_WORDS.TXT at master · 3B1B/videos*. GitHub. Retrieved December 13, 2023, from https://github.com/3b1b/videos/blob/master/_2022/wordle/data/allowed_words.txt

3b1b. (n.d.). *VIDEOS/POSSIBLE_WORDS.TXT at master · 3B1B/videos*. GitHub. Retrieved December 13, 2023, from https://github.com/3b1b/videos/blob/master/_2022/wordle/data/possible_words.txt

WordFrequencyData-Wolfram language documentation. (n.d.). Retrieved December 13, 2023, from https://reference.wolfram.com/language/ref/WordFrequencyData.html

Throughout the project, we have referred to these materials for inspiration.

Gómez, Á. B., & Puga, E. C. (2022). *Wordle solving algorithms using Information Theory* (dissertation).

Russell, S. J., & Norvig, P. (2022). *Artificial Intelligence: A modern approach*. Pearson.

Sanderson, G. (n.d.). *Solving Wordle using information theory. YouTube*. Retrieved from https://www.youtube.com/watch?v=v68zYyaEmEA&t=1201s.

Perfect Wordle algorithm (and how I found it). (n.d.). *YouTube*. Retrieved from https://www.youtube.com/watch?v=sVCe779YC6A&t=418s.

Aravind, S. *Information gain and entropy explained: Data Science*. Humaneer. (n.d.). Retrieved December 13, 2023, from https://www.humaneer.org/blog/data-science-information-gain-and-entropy-explained/