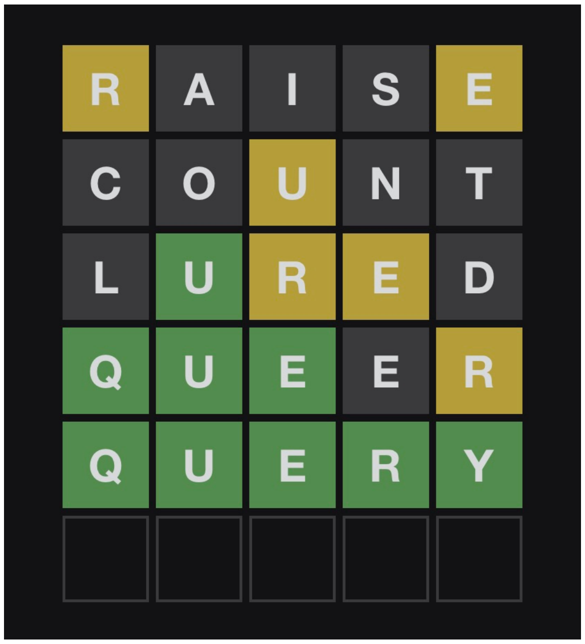
Solving Wordle

Erkin Ötleş

January 10th, 2022

Topics: Wordle, Decision Science, Operations Research, Optimization, Games, Artificial Intelligence, Machine Learning

Let's talk about Wordle. [1] You, like me, might have been drawn into this game recently, courtesy of those yellow and green squares on twitter. The rules are simple, you get 6 attempts to guess the 5 letter word. After every attempt you get feedback in the form of the colored squares around your letters. Grey means this character isn't used at all. Yellow means that the character is used, but in a different position. Finally, green means you nailed the character to (one of) the right position(s). Here's an example of a played game:



A valiant wordle attempt by J.B. Cheadle (January 10th 2022).

It's pretty fun to play, although wracking your brain for 5 letter words can be annoying, especially since you are not allowed to guess words that aren't real words (e.g., you can't use AEIOU). Once I got the hang of the game's mechanics my natural inclination was to not enjoy the once daily word guessing diversion, but was to find a way to "solve wordle".

Now, what does it mean to "solve wordle"? Maybe you would like to start with a really good guess? Maybe you would like to guarantee that you win the game (i.e., guess the right word by your sixth try)? Or perhaps, you'd like to win the game and get the most amount of greens or yellow on the way? "Solving" is a subjective and probably depends on your preferences.

Due to this subjectivity I think there's couple valid ways to tackle wordle. If you have a strong preference for one type of solution you might be able to express that directly and then solve the game in order to get the *optimal* way to play. I'm going to try to avoid the O-word because: 1) I don't know what you'd like to optimize for and 2) these approaches below don't solve for the true optimal solution (they are *heuristics*).

The solution strategies I've explored thus far can be broken down into two major categories. The first set of strategies are trying to find really good first words to start with (**First Word**) and the second set are finding strategies that can be used to pick good words throughout the course of the game in response to responses received from guesses (**Gameplay**).

Let's start with the **First Words** strategies: there are two first word strategies that can be employed based on how you'd like to start your game.

1. **First Word - Common Characters**: ideal if you'd like to start your game using words that have the most common characters with all the solution words. Think of this as trying to maximize the number of yellow characters that you get on the first try.

	Solution Words	Usable Words
1st	later, alter, alert	oater, orate, roate
2nd	sonic, scion	lysin
3rd	Pudgy	chump :)

2. **First Word - Right Character in Right Position**: ideal if you'd like to start the game using words that have the highest likelihood of having the right characters in the right position. This would yield the most number of green characters.

	Solution (& Usable) Words
1st	slate
2nd	crony
3rd	build

Note on solution word vs. usable words. Wordle has two sets of words, solution words and other words. Other words are never the correct answer but can be used as a guess. There's a chance that other words can be used to get a lot of yellows, despite never being the correct answer. So I created a list of usable words that combined the solution words and the other words. Notice that the **First Word - Common Characters** strategy has two lists. That's because there are other words like "oater" that are more likely to produce yellows than the best solution word "later". This isn't the case for the **First Word - Right Character in Right Position**, as it produces the same results for both sets of words.[1]

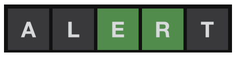
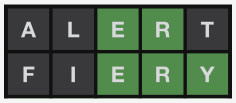
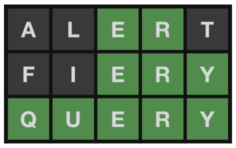
[1] This makes sense as the words in the solution set should produce greens across the board (5 greens) for each character-position for at least one word (themselves). This isn't the case for the words that are usable but not solutions. So they naturally have a disadvantage when it comes to the green metric.

You might also observe that there are several sets of words in terms of 1st, 2nd, and 3rd. If you wanted you could use these strategies over several rounds to build up your knowledge. However, these strategies don't take into account the feedback that you get from the game. So there may be better ways to play the game that take into account what kind of results you get after you put in a guess.

These strategies are the **Gameplay** strategies. I'll present two potential approaches that use knowledge as it is collected.

1. **Gameplay - Refine List + Common Characters**: this one works by sifting through the remaining words that are feasible (e.g., don't use grey characters and have green characters in the right spot) and then uses the **Common Characters** approach to rank the potential word choices that remain.
2. **Gameplay - Reinforcement Learning**: this one works by learning what is the best word to guess given what you have guessed in the past. [2] It does this learning by playing the Wordle many times (e.g., millions) and then collecting a reward based on how it does (+1 point for winning and 0 points for losing). Over repeated plays of the game we can learn what guesses might lead to winning based on the current state of the game.

Here is an example of the **Gameplay - Refine List + Common Characters** strategy in action based on the Wordle from January 10th 2022.

Guess	Green Characters	Grey Characters	Guess	Result
1	*****		alert	
2	**er*	a, l, t	fiery	
3	**ery	a, f, i, l, t	query	

Here you can see that after every guess we get to update the green characters and the grey characters that we know about. For example after round 1, we know that the word must be **er* (where * represent wildcards) and must not contain the characters: a, l (el) or t. I use regular expressions to search through the list of words, the search expression is really simple, it just replaces * in the green character string with tokens for the remaining viable characters (the set of alphabet characters minus the grey characters).

The reinforcement learning based approach would operate in a similar manner for a user. However, the mechanics under the hood are a bit more complicated. If you are interested in how it (or any of the other strategies) work please see the appendix.

As I mentioned above, solving wordle is subjective. You might not like my approaches or might think there are ways for them to be improved. Luckily I'm not the only one thinking about this problem. [3, 4]

Appendix
This contains some technical descriptions of the approaches described above.

First Word - Common Characters
This one is pretty simple. I am essentially trying to find the word that has the most unique characters in common with other words (this is a yellow match).

In order to do this I reduce words down to character strings which are just lists of unique characters that the words are made up of. So for an example, the word “savvy” becomes the string list: a,s,v,y.[2] We then use the chapter strings to count the number of words represented by a character. So using the character string from above the characters a, s, v, and y would all have their counts incremented by 1. These counts represent the number of words covered by a character (word coverage).

We then search through all words and calculate their total word coverage. This is done by summing up the counts for every character in the word. We then select the word with the highest amount of other word coverage. In order to find words to be used in subsequent rounds we can remove the characters already covered by previously selected words and repeats the previous step.

Code can be found in the first_word_common_characters.ipynb notebook.

First Word - Right Character in Right Position
This one is a pretty straightforward extension of the First Word - Common Characters approach that has an added constraint, which is position must be tracked along with the characters.

To do this we count a character-position tuples. For every word we loop through the characters and their positions. We keep track of the number of times a character-position is observed. For example, the world “savvy” would increment the counts for the following character-portion tuples: (s, 1), (a, 2), (v, 3), (v, 4), (y, 5). These counts represent the number of words covered by a character-tuple (word coverage).

We then loop through every word and calculate their total word coverage. This is done by breaking the word into character-position tuples and summing up the counts of the observed character-positions.

Code can be found in the first_word_right_character_in_right_position.ipynb notebook.

Both the First Word strategies can be converted from counts to probabilities. I haven’t done this yet, but maybe I’ll update this post in the future to have that information.

The **Gameplay** strategies are a little more complicated than the **First Word** strategies because they need to be able to incorporate the state of the game into the suggestion for the next move.

[2] Why character strings instead of using all of the characters in the word? We don’t want to double count the words.

Gameplay - Refine List + Common Characters

This approach is reminds me of an AI TA I had. He would always say "AI is just search". Which is true. This approach is pretty much searching over the word list with some filtering and using some distributional knowledge. It was surprised at how easily it came together and how effective it is. As a side note, it was probably the easiest application of regex that I've had in a while.

There are three components to this approach:

1. **Generate Regex**: build the search filter
2. **Get possible solutions**: apply filter to the word list
3. **Rank order solutions**: apply common character counting on the filtered word list

I will briefly detail some of the intricacies of these components.

Generate Regex: the users need to provide 3 things before a guess 1) a string with the green characters positioned correctly and wildcards (*) elsewhere, 2) a list of the yellow characters found thus far, and finally 3) a list of the gray characters. Using this information we build a regular expression that describes the structure of the word we are looking for. For example let's say we had **ery as green letters and every character other than q and u were greyed out then we would have a regex search pattern as follows: [qu][qu]ery.

Get possible solutions: after building the regex search string we can loop through the list of solution words and filter all the words that don't meet the regex search pattern. We can additionally remove any words that do not use characters from the yellow characters list. Finally, we then **Rank Order Solutions** by finding each words coverage using the approach described in **Common Characters** above. This produces a list of words ranked by their likelihood of producing yellow characters on the remaining possible words.

Code can be found in the gameplay_refine_list_common_characters.ipynb notebook. There's also a website with this solver implemented.

Gameplay - Reinforcement Learning

This approach is based on tabular Q-learning. [2, 5] Its a little bit complicated and I'm unsure the training procedure produced ideal results. But I'll provide a brief overview.

Reinforcement learning seeks to learn the right action to take in a given state. [6] You can use it to learn how to play games if you can formulate that game as a series of states (e.g., representing a board position) and actions (potential moves to take). [5] In order to convert tackle the wordle task with RL we need a way to represent the guesses that we've already done (state) and the next guess we should make (action).

The actions are pretty obvious, have one action for each potential solution word we can guess. There's about 2,000 of these.

The states are where things get hairy. If you wanted to encode all the information that the keyboard contains you would need at least 4^26 states. This is because there are 4 states a character can take {black/un-guessed, yellow, green, grey} each character can be in anyone of these states. This is problematic - way too big! Additionally, this doesn't encode the guesses we have tied. What I eventually settled on was a state representation that combined the last guessed word along with the results (the colors) for each character. This is a much more manageable 2,000 x 4^5.

I then coded up the wordle game and used tabular Q-learning to learn the value of state action pairs. This was done through rewarding games that resulted in a win with a 1 and losses getting a 0.

I think this also might be solvable using dynamic programming as we know the winning states. These are terminal and then I think you can work backwards to assign values to the intermediary states. It’s been almost a decade since I took my dynamic programming class, so I need a bit of a refresher before I dive into it.

As you can see, there are a lot of interesting questions that arise from formulating this task as an RL problem. I will probably come back to this and explore it further in the future.

Bibliography

1. *Wordle - A daily word game*. 2022; Available from: https://www.powerlanguage.co.uk/wordle/.
2. Q-Learning - An introduction through a simple table based implementation with learning rate, discount factor and exploration - gotensor. 2019.
3. *Solve Wordle*. 2022; Available from: https://www.solvewordle.com/.
4. Glaiel, T., The mathematically optimal first guess in Wordle. 2022.
5. Friedrich, C., Part 3-Tabular Q Learning, a Tic Tac Toe player that gets better and better. 2018.
6. Sutton, R.S. and A.G. Barto, *Reinforcement learning : an introduction*. Adaptive computation and machine learning. 1998, Cambridge, Mass.: MIT Press. xviii, 322 p.