COSC343: Assignment 1 report

Yashna Shetty (2901410)
August 9, 2022

1 Introduction

This python program implements a version of Wordle. For this assignment, the task involved exploring an agent to solve the game with a means to score better than the pre-implemented "random-agent.py". Though my approach was unable to achieve the best solution, it did achieve a "better than random solution".

This Wordle program involved a variety of different languages, which effectively made implementation of a Wordle solver difficult as optimality of the initial word was difficult to guarantee by using the most number of vowels approach. However, another effective strategy could have been implemented (and will be discussed in *2.3*).

2 Implementation

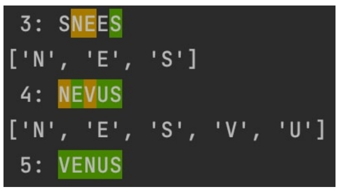
My approach involved eliminating the wrong answers. To accomplish this, my code guesses a random word from the whole dictionary at $guess\_counter = 0$.

This results in a return of information involving the validity of the word guessed, that is, *letter_states*, and *letter_indexes*. From this, I ran a for loop on every word in the dictionary and eliminated words based on the following:

- If states returns 0, delete any word with this letter in it.
- If states returns 1, delete any word without this letter in this position.
- If states returns -1, delete any word with this letter in this position.

There was also the issue of letters returning 0 if the word did not include a double up of that letter. For example, if the final word is VENUS, a guess of SNEES would return a letter_state of $[0, -1, -1, 0, 1]$.

With the above conditions, any words containing an S or E would be eliminated from the set of possible solutions - ultimately eliminating the solution, VENUS. To fix this issue, a *good_letters* variable was implemented to avoid eliminating the correct solution. Essentially, any letters that returned a 1 or a -1 from the solution guessed prior were added to the variable. Iterating through the set of possible solutions after this meant that if a letter in the guess returned a 0 but was a letter in the *good_letters*, then it was treated like a letter returning a state of -1.



```
 3: SNEES
['N', 'E', 'S']
 4: NEVUS
['N', 'E', 'S', 'V', 'U']
 5: VENUS
```

Figure 1: Good Letters List

2.1 Results

Upon completing this implementation of the agent, I found that, unlike the random-agent.py, my agent is able to complete 100 rounds of Wordle in under 6 guesses. Running 20 simulations of 100 games returned an average of 5.65 guesses total for the agent to guess the correct word.

2.2 Evaluation

Admittedly, this implementation was not optimal. In conversation with other classmates, I found some would return an average of 3 guesses. My implementation of this search strategy was quite similar to the Uninformed Search strategies discussed in class and used a depth-first search approach.

In order to evaluate my implemented strategy, I looked at four main components, as discussed in class:

- Completedness
- Time Complexity
- Space Complexity
- Optimality.

The findings concluded that my strategy was not optimal.

2.2.1 Completedness

The Completedness of my strategy was not assured. In some cases, the program was unable to find the correct solution in under 6 moves. Though, if there were not a limit to the number of movesets allowed, the solution would be a guaranteed one in a finite environment.

2.2.2 Time Complexity

I observed Time Complexity to be of exponential growth and wholly dependant on the size of the dictionary used in the program and the length of the word. The agent iterates over the entire dictionary initially, meaning a larger dictionary would return a larger time complexity, that is: $O(b^m)$. Where b is the maximum branching factor and m is the maximum depth of the state space.

2.2.3 Space Complexity

I observe Space Complexity to be of linear growth. A copy of the dictionary is made and requires a full initial search. This means that space complexity is: $O(2bm)$. Where b is the maximum branching factor and m is the maximum depth of the state space.

2.2.4 Optimality

The most cost efficient solution is also not guaranteed as this is similar to a depth-first search so all options are not explored. This solution is therefore not optimal.

2.3 A Different Approach

To improve this strategy, an evaluation function would be necessary to turn this search strategy from an Uninformed to an Informed Search.

In the set of possible solutions, a frequency counter for each letter in the list of possible solutions can be made. The letters are then scored from 1 to 26, 1 being the most frequently occurring letter and 26 being the least. With a score correlating to each letter, the word with the lowest total score is picked from the set as the next guess.

This frequency calculation can also be implemented into the initial guess. Scoring every word in the initial dictionary and picking the highest scoring word means the initial word guessed by the agent is inclusive of the most frequently occurring letters. The letter states returned by this optimal guess is then able to effectively eliminate non-possible words.

A different strategy should also be implemented for the two modes - easy and hard.

Where hard mode would need to be left to chance, a separate solution for easy mode could be implemented. If multiple words result in the same frequency score, the agent should take the letters that differ between these highest frequency scored words and find a word from the original dictionary that contains the most number of these letters as its next guess. Resulting states would affirm the best guess from the possible set of solutions.

3 Conclusion

This assignment proved challenging to my abilities. A vast amount of algorithms for Wordle Solvers online implemented the solver for a single language. This made research for this assignment quite challenging. Words implementing the optimal guess for its first guess was impossible as they involved the use of certain letters (for example, vowels in English), or the history of previous Wordle solutions.

While I understood the assignment in theory and was even able to come up with an optimal solution, I struggled considerably when it came to implementing the optimal approach (as explained in *2.3 A Different Approach*).

I did however enjoy this assignment considerably and will continue to work on it in my own time as I believe implementation of my *Different Approach* is possible and could be optimal. I would also like to learn how to apply my theoretical knowledge to practical situations, so further practice would do well in furthering my programming abilities.

I learned a considerable amount about the Numpy library and its advantages and will aim to utilise this feature more in future assignments. I also found myself a lot more attentive to the optimality of programs in my general programming because of this assignment. Where things like space and time complexity weren't really features of my programming unless specified by the assignment or project, they definitely have become features of my attentiveness.