Wordle as a Constraint Satisfaction Problem

Michelle Coker

Introduction

For this project, I developed a python program that can solve Wordle problems by representing them as a Constraint Satisfaction Problem. To play Wordle, players have six attempts to guess an unknown, 5-letter word. With each guess, the player gets more clues about the word based on their previous guesses. There are 5 spaces and any alphanumerical character is a valid placement however guesses are only valid if the resulting word is a valid word in the English Language. Users receive clues indicated by color after each guess. A green clue means the letter is correct and is in the correct place. A yellow clue indicates the letter is in the word but is in the wrong place. A grey clue signifies the letter is not present in the word. Constraint Satisfaction Problems are a way of representing a problem using 3 critical components. Variables are the unknown elements of the problem requiring assignment. Each variable has a domain which denotes the possible values that variable can have. Constraints define rules which the assignment of the variables must abide by. Together, this creates a Constraint Satisfaction Problem where the goal is to find a complete and consistent assignment. This means all variables are assigned and no constraints are violated.

In understanding the material behind this problem, I completed some research on Constraint Satisfaction Problems. My main source comes from the International Journal of Computer Science and Mobile Computing in a paper written by Jyoti et. al in which the two give an overview of constraint satisfaction problems, variants of CSPs, ways to solve them, and some theoretical information about CSPs. This project was inspired by the methods and concepts discussed in this paper. The paper heavily discusses different types of CSPs. The paper explores different types of CSPs noting that generic CSPs represents static constrains but variants of CSPs may differ. Dynamic Constraint Satisfaction Problems (DCSPs) can be used to represent problems where the constraints may evolve. This can be depicted as a sequence of generic CSPs where each CSP is an alteration of its predecessor representing the changing

constraints. The author also gives examples of techniques to solve these DCSPs like Local Repair, Oracles, and Constraint Recording. Other types of CSPs the paper discusses include Flexible CSPs where the CSP can violate some constraints although each problem will call for different requirements on how many constraints can be violated or weighs constraints by importance and chooses which constraints to violate.

As for solving CSPs, they are most often solved using Local Search. CSPs are a search problem where the goal is to find the best solution even where one may not exist. This makes them best fit for Local Search. There are many methods involved in creating the best possible search situation including constraint propagation where some search paths can be eliminated due to limited the domain of some variables. Jyoti et al also discuss interesting ways of looking at solving CSPs. Complete Search, Look-Forward, and Look-Back Algorithms all have different advantages over another when considering what algorithm to solve a CSP.

Methods

To complete this approach I first took notes on the code architecture I thought would be necessary to complete this project. I then searched for any packages that could help me complete this program. In my search, I found the python-constraints library. This library is designed to represent CSPs in python and provides users with a variety of tools to accurately reflect a CSP.

The project is divided into four main classes. On the smallest level is a singular WordleLetter which represents a single letter in Wordle and contains a Letter, Position, and Color. This class is used to handle methods dealing with individual letters and adding the appropriate constraints to a problem given the state of a WordleLetter. The WordleProblem class consists of a list of 5 WordleLetters and is used to represent the whole state of a wordle problem. The Solver class is used to represent and solve the WordleProblem as a CSP. The Solver and testSolver classes are where the python-constraints library is most heavily used.

This library allows users to define a CSP by adding variables and appropriate variables and domains to a problem. This library also allows users to add constraints by initializing a `Problem()` and adding constraints using `Problem.addConstraint()` which accepts a constraint and a set of variables to apply the constraint to.

Additionally, users can also specify an algorithm between Backtracking, Optimized Backtracking, Recursive Backtracking, and a Minimum Conflict Solver. Initially, I used the default Solver which does a Complete Search. I attempted the Backtracking Solver as it also includes Forward Checking to improve efficiency but did not observe a notable difference in time complexity. The Solver solves the puzzle by asking the user to input feedback from a guess according to a given format. In a while loop, it adds the appropriate constraints to the problem given the feedback of the letters. This can be considered a Dynamic CSP as with each iteration of the while loop, new constraints are added that narrow the solution space. The Solver uses its respective algorithm to create a list of solutions according to the constraints. After compiling a list of possible words, the program filters out words that do not exist in the English language. The program then orders these words based on a utility score. The utility score consists of two components, the popularity of the total word, and the frequency of all the letters in the word. To find word frequency, I created a script that outputs a .txt file that uses the Google NGram API to reference the popularity of a given word. For letter frequency, I used a dictionary on Github containing letter frequency data, summed the total frequency of all the letters and divided by 100 to balance the weight against the word frequency.

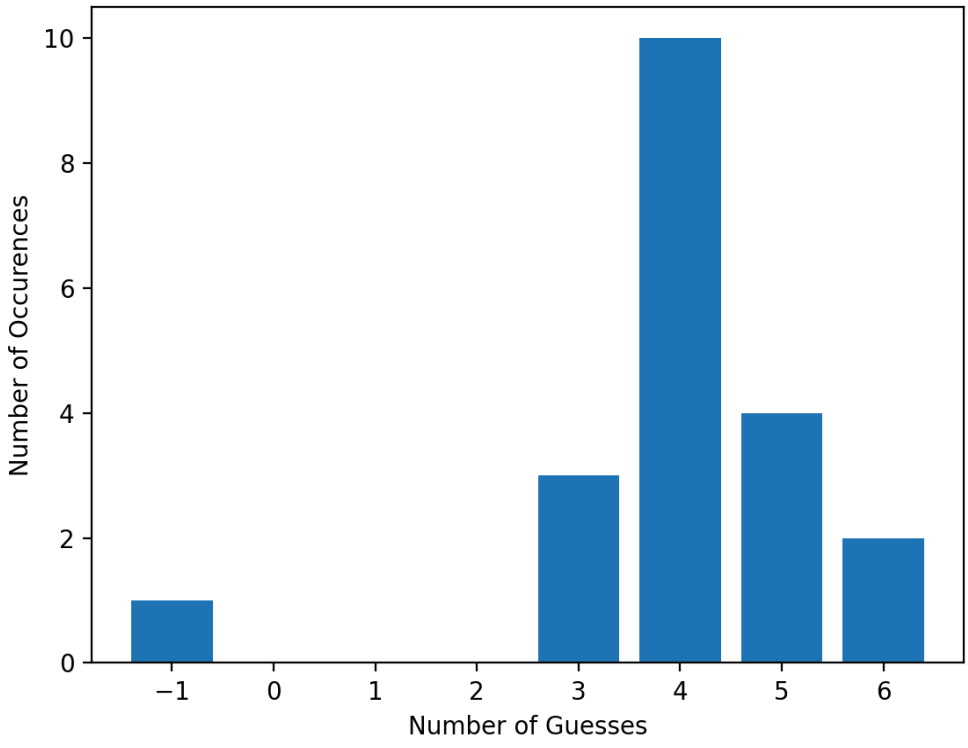
To evaluate my program, I created a testerSolver class that automates the guessing and response process. The Solver class requires human input and is designed for a person to use while trying to solve the wordle. To test the efficacy of the Solver, testSolver alters the solveWordle() method such that the program can solve itself as long as the correct word is known to give hints on the correct word. The automatic solver always starts with “adieu” as its first guess and each subsequent guess is the best guess according to the utility heuristic. I

evaluate the program using the number of guesses it needs to find the correct word. I then plotted two graphs using two different search methods to report results.

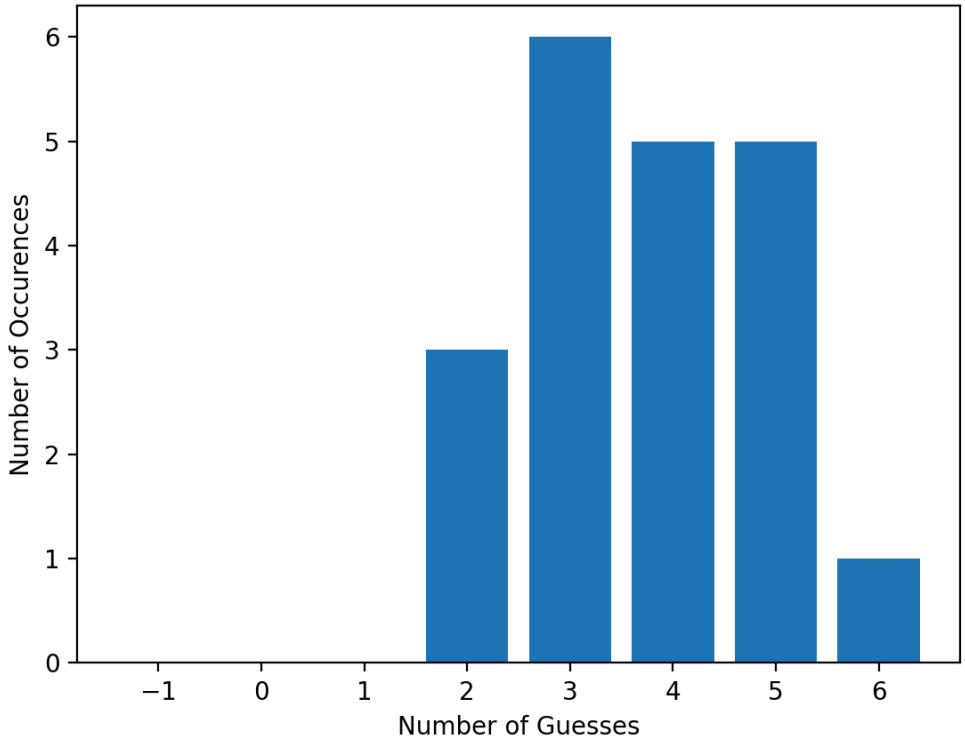
Results

On average, the Python Wordle Solver solved most puzzles in about 4 attempts. This is about average for a human guesser. Compared to other bots tasked with solving Wordle puzzles, this bot performs about average. In my research, the best wordle bot I could find solved puzzles in an average of 3.03 guesses. Given this program solves Wordle puzzles in an average of 4 attempts, this results in about a 25% difference in performance. Two graphs below shows a distribution across a number of 20 games the occurrences of each number of attempts to solve the puzzle. -1 indicates failure to find an answer in 6 attempts or less.

Default Solver (Complete Search)



Backtracking Solver (with Backtracking and ForwardChecking)



Conclusion

Overall, I believe my program worked well. The program manages to guess the word almost every time as “Failures” (words not found in under 7 attempts) are rare. Additionally, the program has similar performance to a human guesser as the average Wordle player can guess the word in around 4 attempts. The main downside of the program is the time complexity. The Solver’s first guess takes quite a bit of time, especially if no information was gained on the first guess (all grey letters). Subsequent guesses take much less time. To improve this in the future there are a few changes I could make. The automatic solver always starts with “adieu” because it has 4 out 5 vowels. However, this limits the starting possibilities for the program and if the correct word does not contain many vowels, it puts the solver at a severe disadvantage timewise. A future implementation could choose from a list of starting words to lessen the risk of having an ill-fitting starting word for the puzzle. Additionally, the heuristic could also be altered to change the utility of words as the number of attempts go up. As the program runs out of

guesses, it makes sense to value word popularity much more than letter popularity. An improved heuristic would account for the changing needs of the program over time. Another improvement could be made on the search algorithm. Additional constraints mandating that the total solution must spell a valid word would eliminate many solutions that cannot be guessed in a real Wordle game and would thus reduce time complexity.

Works Cited

Celles, S., Niemeyer, G., (November 5 2018) python-constraint library (Version 1.4) [Source code]. https://pypi.org/project/python-constraint/

devxleo (2015). letter_frequency_table [Github Repository]. https://gist.github.com/devxleo/9c4f7e917b0f68d7eba3

Google Books Ngram Viewer. (n.d.)Google Books Ngram Viewer. [API]. https://books.google.com/ngrams/

Jyoti et al, CONSTRAINT SATISFACTION PROBLEM: A CASE STUDY, International Journal of Computer Science and Mobile Computing, Vol.4 Issue.5, May- 2015, pg. 33-38, https://ijcsmc.com/docs/papers/May2015/V4I5201520.pdf