Solving Wordle Using Artificial Intelligence - A CSCI4511w Project

Grant Matthews

May 31, 2023

Abstract

The game Wordle took the world by storm in early 2022 and instantly became a hit classic. Not unlike other guessing based games, it is possible to develop an algorithm to solve Wordle and "Wordle-like" puzzles. This project demonstrates one of these algorithms by making use of the Genetic and Minimax algorithms.

This report contains relevant information about how Wordle and "Wordle-like" games work and the history behind solving these games. Furthermore, the Genetic and Minimax Algorithm are explained and the reasoning behind their use in the Wordle solving algorithm is discussed in relation to other problem solving algorithms. The development of a clone game in Python for use by the algorithm is also briefly detailed. Finally, an experimental procedure was developed to test the algorithm and determine how successful it is at solving each puzzle.

1 Problem Description

It feels like everyone is playing developer Josh Wardle's *Wordle* these days. The simple game where you have 6 attempts to guess a unique daily 5 letter word has captured the interests of millions and has spawned hundreds of knock-offs and variations. Wardle originally created the game to help his wife deal with the lows of the pandemic quickly spread after she shared the game with family, who then began to share it with others [14]. As of February 14th 2022, Wardle's version of the game, which was acquired by the New York Times for an undisclosed 7 figure sum, had an estimated 3 million daily players [7]. Furthermore, Wordle has spawned hundreds of copy cats and variations looking to cash in on the global interest. There is Dordle, Quordle and Octordle, versions of the game where the user must play 2,4 and 8 puzzles (respectively) at once. Heardle plays short segments of a song for the user to guess and Worldle shows the user a nation's silhouette and asks them to guess the nation, telling the user how far away (in miles) their guess is from the target nation.

1.1 How does the game work?

The premise of Wordle is rather simple. A user is presented with a game board consisting of a 5x6 grid and a keyboard and asked to guess the secret 5 letter word in the allotted 6 tries. Each guess is recorded on the game board and feedback is provided to the user about their guess. When a user guesses a word, each of the five letters populate a row on the grid. Feedback on the guess is provided to the user by highlighting each letter in the word one of 3 colors. If the letter is highlighted grey, it is not anywhere in the secret word. If the letter is highlighted in yellow, its in the secret word but it's position in the guess isn't it's correct position in the secret word. Finally, if the letter is highlighted in green, it means that the letter is in the secret word in that position. These colors are also applied to the keyboard to make it clear to the user what letters could still be in the secret word. See Figure 1 below for a sample of the game board.

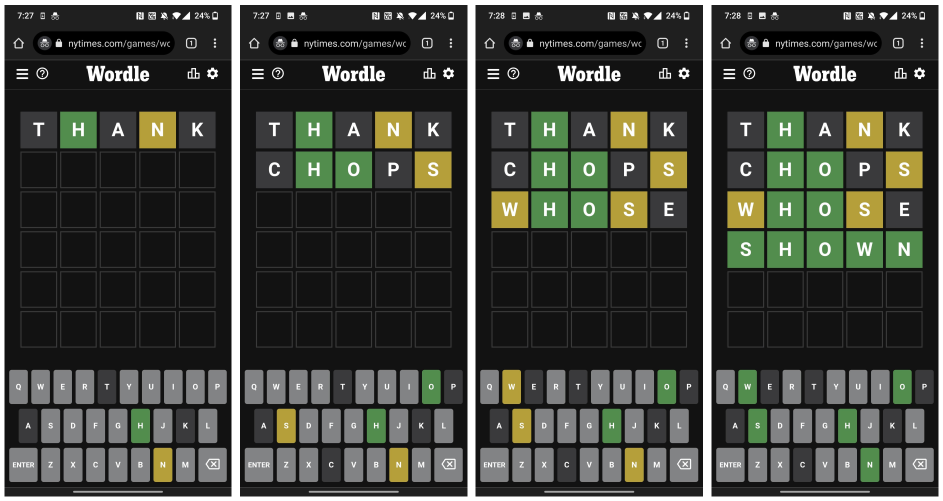

Figure 1: A Game Board from Daily Wordle No.312

Josh Wardle's version of Wordle is a daily puzzle with a new secret word announced every day at midnight. Each consecutive daily puzzle a player solves increases their streak and the results of their guesses can be shared easily with friends, family and coworkers. Competitive players can also enable hard mode, where each guess must contain all of the previously revealed letters. In Figure 1, the user did not play in hard mode as evident by the fact that guesses two and three did not contain the letter N. In hard mode, each guess after THANK would be required to have the letter N in a new position.

1.2 Unique Challenges

All of these variations, along with the original create unique and interesting problems that are possible to solve using the power of computation. These problems have drawn the interest of scientists and users alike, both looking to break down the game and determine the best strategies to guess the daily puzzle in as few tries as possible. An example of these problems can be seen in the choice of a starter word. While the majority of users seem to agree that ADIEU is the best starter for its use of 4 vowels, there is little in the way of consensus with some users preferring alternatives like ABOUT, CANOE, AUDIO, OUIJA or EQUAL [3]l. After the first guess, user strategy becomes even more varied and less well studied.

Another example of a unique challenged posed by Wordle can be seen in the daily puzzle No.265 from March 10th. It's secret word, WATCH seems fairly common place but yet was able to cause no end of headache for users due to its structure. WATCH ends in _ATCH, a very common ending shared with words like PATCH, CATCH, LATCH, HATCH, CATCH, MATCH and BATCH [8]. Given the game only has 6 guess, even if one were to stumble upon an _ATCH word on your first guess, they may still fail the puzzle if they don't happen to guess the word during the subsequent 5 attempts. Strategies to handle double and triple letters are also varied and under studied.

Wordle's strategy for providing feedback on a guess also creates a unique challenge for users to solve. As discussed above, Wordle uses green to indicate a letter is in the same position in the secret word and yellow to indicate that a letter is in the secret word, but not in that position. However, there is no feedback given to indicate to a user the presence of multiple of the same letter in the secret word. Wordle No. 251 from February 26th was VIVID, a word that contains two sets of double letters. A user opening with a guess of UNBID would see the I and D highlighted green on their board, indicating those letters are in the correct position. What they would not be made aware of is that there is actually another I and D in the final word [5]. This led many users to fail the puzzle.

2 Background

To many Wordle is an exciting, brand new concept to enjoy. But to a computer scientist, Wordle and puzzles like it have been familiar challenges for over 50 years. This project builds on the work of previous scientists, and as such it is important to review what makes up the core of this project.

Below one will find an a history of the game Wordle is based on: Mastermind, and an overview and functional outline of the two artificial intelligence algorithms used by this project: Genetic Evolution and the Minimax decision algorithm.

2.1 Mastermind

The aim of this project is to develop an algorithm, trained through a Genetic Algorithm to solve Josh Wardle's game Wordle. However, saying that Josh Wardle invented the concept of Wordle would be a lie. Wordle as a concept dates back to the early 70s with a game called Mastermind, which is actually a development upon a much older game, Bulls and Cows. Mastermind pioneered the idea of a secret code revealed through guessing possible codes by using combinations of colored pegs to represent each code and informing the player of the accuracy of each peg in their guess[4]. Mastermind is a game that has been around almost as long as computers have, and has been demonstrated many times to be NP complete and solve able in a set number of guesses. The first to do so was Donald Knuth who suggested the following algorithm in 1977:

1. Create a Set S of all Possible Codes. Mastermind codes are 4 pegs long and each peg can be 6 possible colors. This means there can be a total of $6^4 = 1296$ possible codes.
2. Start with the initial guess of 1122. Knuth demonstrates that other guesses (like 1123) leave a few puzzles unsolvable in 5 guesses.
3. Play a guess
4. If you get all four pegs the game is won, terminate.
5. Remove any guesses from S that are now impossible due to the results of the previous guess.
6. Apply the minmax algorithm on all remaining guesses in S. The score of each guess is determined by the minimum number of elements in S that will be removed if that guess is used.

Figure 2: Donald Knuth's Agorithm for Solving Mastermind in 5 or fewer Guesses[6]

Figure 1 shows that Mastermind is an NP complete problem, that is to say that it is solvable in a non deterministic amount of time. This idea was expanded in 2005 by Jeff Stuckman and Guo-Qiang Zhang who demonstrated that Mastermind, referred to in the paper as *The Mastermind Satisfiability Problem* or MSP, is NP complete for codes of any length and that for any code length there is a solution to be found[11].

Wordle's game mechanics can clearly be traced back to Mastermind. Wordle is simply a version of Mastermind where each code is 5 pegs (read letters) long and each peg can be one of 26 colors (read letters). Theoretically this would leave us with a potential guess pool of 26^5 possible codes. However, because each guess has to be an English word on Wordle's approved word list, the actual size of the guess pool is far smaller. While the MSP problem has already been solved, the AI developed in this project aims to learn a more efficient way to solve each puzzle given the constraints of Wordle's rules.

2.2 Genetic Evolution

Genetic Algorithm's were first proposed by Alan Turing in 1950, where he suggest that a "learning machine" could be developed that draws parallels with the principles of Evolution [12]. One of the first scientists to work on developing computer simulated evolution, Nils Aall Barricelli, published his first book titled *Symbiogenetic Evolution Processes Realized by Artificial Methods* in 1954 (translated to English in 1957) that details the theoretical creation of an evolutionary algorithm to mimic evolutionary patterns seen in real life[1]. He developed upon these ideas in his 1963 paper, which detailed an algorithm capable of learning to solve a basic game he created called Tac Tix. The game was relatively simple but presented a unique NP complete problem, giving Barricelli the framework needed to develop an algorithm to solve it. His paper outlines the genetic algorithm necessary to learn how to solve the problem[2]. Developments in computational evolution came at an increasingly fast pace throughout the 60s, 70s, and 80s. Notable mentions include the works of Ingo Rechenberg and Hans-Paul Schwefel who developed algorithms to solve multiple complex NP complete engineering problems throughout the late 60s and early 70s. In the late 80s and early 90s General Electric became the first to sell a commercial genetic evolution algorithm in the form of a toolkit designed for industrial processes.

Darrell Whitley's 1994 paper titled *A genetic algorithm tutorial* details multiple modern day genetic algorithms and the algorithm used in this project. The canonical genetic algorithm, detailed in figure 2, represents the basic strategy of this project's algorithm.

1. Create your starting population represented by random binary strings. Each bit represents a trait or part of a trait used to determine an algorithm's behavior in a given function. Each string should be different within starting the population.

2. Using each member of the population's traits, run the algorithm and measure the result in a quantifiable way. How this is done varies depending on the algorithm being used, but it should be possible to score each population member to determine the most and least successful members. It is then possible to average the scores of all members of the population and use the result, S_{ALL}, to calculate how well percentage wise any given member of the population did against the the rest of the population with the formula S_{member}/S_{ALL}. The result of this is hereby referred to as R.

3. R denotes the likelihood any given element is present in the next generation. For example, a population member with an R value of 1.35 has a 100% chance of being present in the next generation once and a 35% of being present in the next generation twice. Likewise, a population member with an R value of .35 has a 35% chance of being in the next population once. Using these rules, we can construct a new population.

4. Take the new population and randomly assign member pairs. Crossover is applied to each pair with according to a probability hereby referred to as P_C. P_C can be altered to adjust the rate of mutation in a genetic algorithm and the value used depends on the algorithm being tested, the length of each population member and potentially many other factors. If a member pair is selected for crossover, a random crossover point is selected and two new children are created. In the example shown below, we take two binary members 1101001100101101 and yxyyxyxxyyyyxyxxy (x=0 and y=0 for clarity) and apply the crossover technique at the 6th bit:

   11010 |01100101101
   yxyyx |yxxyyyyxyxxy
   becomes
   11010yxxyyyyxyxxy and yxyyx01100101101

   For each pair of children created through crossover, we apply a mutation according to a defined mutation probability, P_M. As with P_C, P_M depends on the population and algorithm being used, but typically P_M is less the 1%. If an child is selected for mutation, one of its bits are randomly selected for mutation. In some cases, a mutation means that the bit value is randomly regenerated and therefor has a 50% chance of changing whereas in other cases a mutation means that the bit is flipped. Again either method depends on the algorithm and population members used.

Figure 3: The Canonical Genetic Algorithm[15]

2.3 Minimax Decision Algorithm

The Minimax Algorithm was developed and proven long before the concepts of silicon based computing. In 1924 John von Neumann proved the the minimax theorem which details a process for minimizing the maximum possible loss a player can take when playing against another opponent [13]. While his original paper titled *Zur Theorie der Gesellschaftsspiele* or in English *On the Theory of Board Games* was written in German, the ideas present have been vastly expanded on since then and have been developed into the commonly used algorithm present in this project's solution [9].

Unlike the Genetic Algorithm which relies on traits to determine how the algorithm behaves, the Minimax Decision Algorithm is built to parse through tree structures by analyzing the cost of each node to solve a problem. George Stockman's 1979 paper titled *A Minimax Algorithm Better Than Alpha-Beta?* compares the Minimax algorithm to another well known tree parsing algorithm, Alpha-Beta Pruning and goes over how the Minimax Decision Algorithm works. Figures ADD3 and ADD4 below demonstrate the kind of tree Minimax works on and how the algorithm works [10].

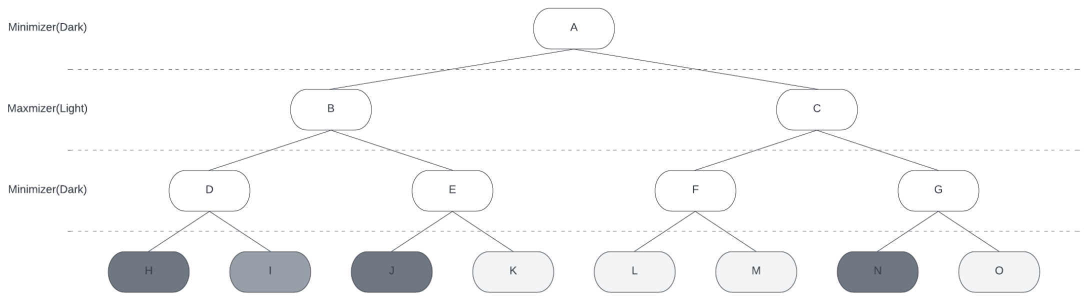

Figure 4: A Sample Decision Tree for the Minimax Algorithm to Parse

Figure 4 shows a binary decision tree with three different outcomes: Dark, Medium and Light. In this tree, our minimizer is trying to find a set of guaranteed decisions that takes it to a dark outcome. To apply the Minimax algorithm, follow the steps in Figure 5 and observe the final state of the tree in Figure 6.

1. Find the states of all possible leaf nodes (this is already done in Figure 4)
2. Move to the first parent Node (D in Figure 4). In our tree, the minimizer controls D and it has two choices, H or I. Because H is a win state, the minimizer will always choose H if it is in position D and therefore can be assigned the dark color.
3. Move to D's parent node, B. The maximizer controls B and has the option of picking D or E. Because the maximizer wants to find a light colored outcome, it won't pick D unless it has to because it knows its a guaranteed loss. Instead it will opt to explore E.
4. E is the minimizer's node, and it will examine its two choices, J or K to determine the best outcome (dark). The minimizer will always choose J here because it is dark, which means E can be colored dark.
5. Move to E's parent node, B, controlled by the maximizer. Here, the maximizer can choose D or E, but because the minimizer has already shown that it will always choose dark if D or E is picked, the maximizer has no choice but to assign dark to B.
6. Move to B's parent node, A, controlled by the minimizer. The minimizer has two choices, it can either chose the path B (dark, guaranteed win) or it can explore node C. Because the minimizer knows it can win by choosing B, it has no need to explore C. This allows us to prune C and all it's children (important later when discussing the algorithm used by this project).

Figure 5: Example of the Minimax Algorithm on Figure 4

3 Approach

The main goal of this project is to create an algorithm capable of solving a Wordle puzzle better than human. The approach taken to create the algorithm is detailed in the following sections. All of the source code was uniquely developed in Python for this project and and is available on Github (linked

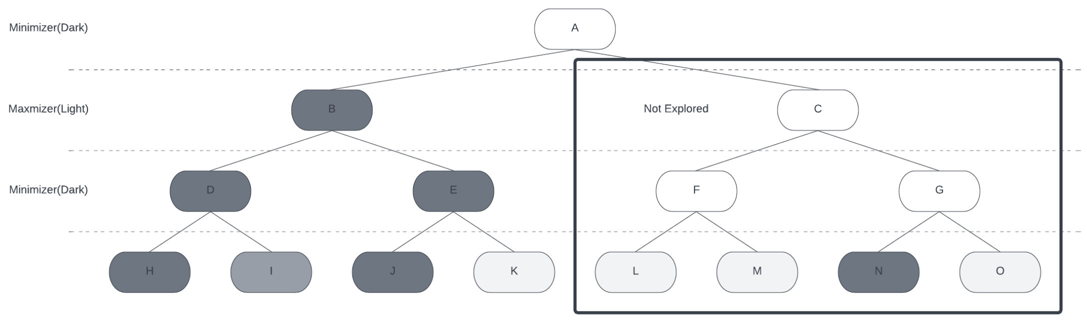

Figure 6: Final State of the Decision Tree from Figure 4

below). The code is made up of 3 major components, the Wordle Game, Agents, and the Agent Coordinator. The algorithms that make up each component are detailed below.

3.1 The Game

The original source code for Josh Wardle's Wordle is readily available online, however it is written in javascript and intended for a web deployment. This meant that it was lacking a lot of the robust reporting features needed for a solving algorithm to understand its position in the game. A decision was made to develop a clone of the original game, written in Python, to enable better access to the game's internal workings for the project's algorithm. The project's game (hereafter referred to as *the game*) was intended to contain an optional Graphical User Interface (GUI) as well as a command line based "verbose" mode. Both modes interface with a Game class object that contains the inner workings of a particular puzzle.

3.1.1 The Game Class

The Game class is the most important component in the project's Game. Each Game object is one instance of a game board solvable by a human player or algorithm. The Game class is made up of the following functions:

3.1.1.1 Constructor

The Game class's constructor takes in up to 6 arguments and initializes the various class variables needed by other game functions. Each of the 6 arguments and what they initialize are listed below:

1. **word (required):** A string object representing the secret word.
2. **num_guesses (required):** An integer object representing the maximum number of guesses a player is allowed. In a classic Wordle puzzle, this would be set to 6.
3. **gui_flag (optional, default: False):** A Boolean for telling the Game object to whether or not to enable to Graphical User Interface (GUI)
4. **keyboard_flag (optional, default: False):** A Boolean for telling the Game object to whether or not to enable to Graphical User Interface's keyboard for GUI interaction.
5. **verbose_flag (optional, default: False):** A Boolean for telling the Game object to whether or not to enable to Verbose Mode (command line) interaction.
6. **debug_flag (optional, default: False):** A Boolean for telling the Game object to whether or not to print the secret word before the game starts and the results of a Game when it has completed.

The Game object also uses it's constructor to create a list of all possible words, to ensure that the user is only guessing approved words. Currently this list of approved words can be found in source code in the /Words/wordle-words.json file. This file was pulled directly from Josh Wardle's source code and contains every possible 5 letter word in the original Wordle. This list is used by multiple classes and functions and will be referenced extensively below.

3.1.1.2 get_max_guesses

This simple public function returns the maximum number of guesses the game will allow. This is an example of why a clone of the original game was developed. While it would be possible to simply code in a hard guess limit of 6 to match the original rules, one of the tertiary goals of this project was to create an expandable code base should future developers wish to implement AI for other Wordle like games.

3.1.1.3 start_game

Calling this public function allows the game to create the necessary tkinter elements for the GUI (if enabled), and print to any command line statements enabled by the debug or verbose mode flags. This command is not strictly necessary to begin a game as the constructor handles all of the vital game construction but this function exists to separate the creation of the GUI or the printing of command line statements from the creation of the game. Therefore, if none of those elements are necessary (as is the case during algorithm training) we can avoid the hardware cycles and save computation time.

3.1.1.4 end_game

This public function serves to inform the user or algorithm playing the game of the final result. It does so in two ways. First, if any of the three display mode flags are set, information will be displayed in the relevant location (GUI or command line) about the result of the game. Secondly, and more importantly for this project's algorithm it will return a string value *win* or *lose* to indicate the result of the game.

3.1.1.5 _set_gui

This private function is called by the start_game function to initialize and display the GUI using the tkinter python library. As it is not strictly necessary for the project, a detailed description of how this function works will not be provided.

3.1.1.6 _event_enter_gui, _update_gui, _update_verbose and _update_debug

These four private functions are used to provide an update to the user regarding the status of the game via the relevant location (GUI or command line). As it is not strictly necessary for the project, a detailed description of how this function works will not be provided.

3.1.1.7 make_guess(guess)

The purpose of this public function is to facilitate an guess on the game board using the passed in *guess* string object and return an *guess* length array containing the color of each letter on the game board. As discussed above, Wordle uses a 3 color system to inform a player about their guess. This function works in the same way, returning an array the same length as the *guess* where each array element is either *g*, *y* or *e* according to the corresponding letter's position in the secret word. For example, if the secret word was SHOWN and our first guess was THANK (like Figure 1), make_guess would return ["e", "g", "e" ,"y", "e"]. This array can be examined by the user or an algorithm to determine the their next guess.

3.1.2 Main Game Script

Originally, a goal of this project was to develop a GUI interface for a user to use to play a game, similar to Josh Wardle's. However, the scope of the project changed and most of the development on a main function for Game was abandoned. Currently, the main function does exist and it is possible to play a Classic game of Wordle using the Graphical User Interface or command line, but many of the options do not work. There is also no Graphical User Interface for creating and testing the project's algorithm on a set of puzzles as was originally planned. Future iterations of this project plan to address these issues.

3.2 The Agent

The Agent Class is the most important Class object in the project. An Agent object represents a single instance of the project's algorithm and can be used to solve Game objects. Below Figure 7, which details the algorithm used by each Agent, is an overview and description of each of an Agent's functions. It may be helpful to review these descriptions before examining Figure 7 to have a better understanding of the algorithm it represents.

3.2.1 The Main Agent Algorithm

1. Using the Class object's master list, create a copy of the words, words_letters and possible_letters lists.

2. Choose a guess from the words list by choosing the word with the highest cost value. The algorithm for determining the cost value of a word described in Section 3.2.2.

3. Send the guess to the Game object.

4. Using the result returned from the Game object modify the two words lists and possible_letters list. The words list is modified so that it only contains words that are possible by now knowing the result of the guess. Using the guess and secret word in Figure 1 as an example, knowing that the letter T from THANK is not in the secret word, the agent will remove any words from the words list that contain the letter T (i.e. TRUCK or CRUST). The words_letters list is modified so that it only contains words that do not include the letters of the guess. Using the guess and secret word in Figure 1 as an example, any words containing T, H, A, N or K are removed from words_letters. The possible_letters list contains all of the letters in the alphabet and is modified with each guess to only contain letters that have not been guessed yet. Again, using the guess and secret word in Figure 1 as an example, the letters T, H, A, N or K are removed from possible_letters.

5. Choose a guess from the words_letters list by choosing the word with then highest cost value. This is done to try and eliminate as many additional words from the board as possible by eliminating potential letters from possible_letters. This is also done to cut down on computation time, as demonstrated below, an Agent's traversal of a decision tree can take time.

6. Repeat Steps 3 and 4.

7. Choose a guess strategy (Step 2 or Step 5) according the results of a decision tree traversal by the agent. This algorithm is explained in greater detail in Section 3.2.2.

8. Repeat Steps 7, 3 and 4 (choose guess, make guess, modify lists according to the result of guess) until either the Agent runs out of guess in the game, or it correctly guesses the word.

9. Return the result of the game (”win” or ”lose”) and the number of guesses it took to reach that state.

Figure 7: The Main Agent Algorithm

3.2.2 An Agent's Algorithm(s) for Determining the Cost of a Word

1. For each word in the word list, separate the word into an array of letters and assign add the letter's position to the string. For example, "thank" becomes ["t0", "h1", "a2", "n3", "k4"].
2. Generate all possible ordered item sets made up of these (five) elements. The first 5 ordered item sets generated from "thank" are ["t0"], ["t0", "h1"], ["t0", "h1", "a2"], ["t0", "h1", "a2", "n3"], ["h1"].
3. Add all of the generated item sets to a master list of item sets and update the support counts accordingly. For example, if the only two words in the list of words are "thank" and "frank" the resulting master list would contain an item set ["a2", "n3", "k4"] with a support count of 2 since both the words in the list of words contain the ending "__ank".
4. For each word in the words list, create an array with len(word)-1 spots (for our 5 letter words, this array would have 4 elements). The first position in this array represents the total support all of the word's single item item sets has, the second represents the total support of all the word's two item item sets etc. Re scan the item sets of each word in the word list. For each item set, get it's support from the master list and add it to the corresponding position in the array. Using "thank" as an example, if ["t0"] has a support of 25, ["h1"] a support of 2, ["a2"] a support of 36, ["n3"] a support of 7 and ["k4"] a support of 12 in the master list, the first value in the created array corresponding to "thank" would be (25+2+36+7+12=) 82.
5. Using an Agent's DNA (which should contain the same number of elements as any given word's total support array), modify each word's total support array according the corresponding value in the Agent's DNA. Because the total support of single item sets is always roughly 10x the support of two item item sets, itself roughly 10x the support of three item item sets etc, apply a 10x multiplier to every value in the words array. For example, if an Agent has the following DNA [0.1, 0.4, 0.2, 0.9] and the a word has the following total support array [3021, 567, 90, 3], the resulting word's array would be

   [(3012/0.1)x1, (567/0.4)x10, (90/0.2)x100, (3/0.9)x1000] which equals
   [30120, 14175, 45000, 3333.3].
6. For each word, add the values together to obtain the cost of the word.

Note: In this project's algorithm, all costs are determined based on the support counts of item sets found in the words the list. The algorithm for determining the cost of in the words_letters list uses the support counts of the items in words. In the above algorithm, when calculating the cost of words in the words_letters, the master list generated in Step 3 and used in Step 4 is replaced with the master list from the words list cost calculation. If any word in words_letters contains an item set not found in the words master list, the support for that item st is assigned to 0.

Figure 8: An Agent's Algorithm(s) for Determining the Cost of a Word

3.2.3 An Agent's Algorithm for Traversing a Decision Tree and Determining a Guess Strategy

An Agent has two possible guessing strategies, it can either make a guess based off the highest cost word in the words list, or it can make a guess based off the highest cost word in the words_letters lists. Furthermore, the Game can return one of $3^5 = 243$ (in the case of classic Wordle) result arrays to indicate the result of the guess. Therefore, for each of the two guess strategies there is potentially 243 outcomes. Each outcome has two guess strategies, of which each has potentially 243 outcomes etc.

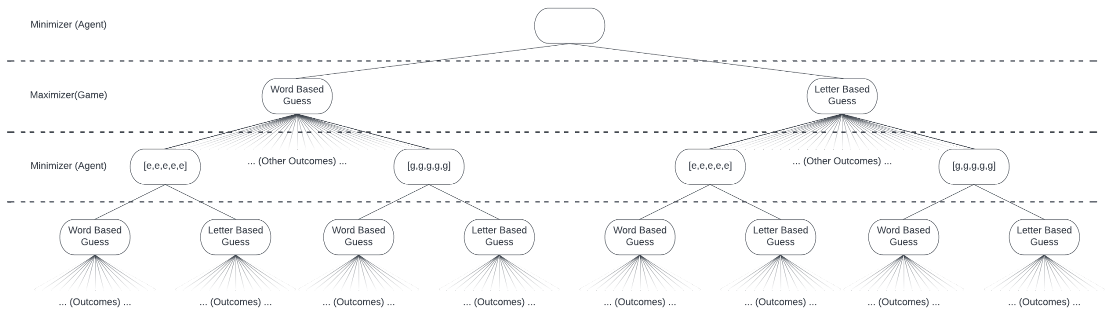

Figure 9: An Agent and Game's Decision Tree

By creating a tree of this relationship (as seen in Figure 9) it becomes clear that the relationship between the Agent and the game can be compared to the Minimax algorithm. Here, the Agent is the minimizer, and is trying to minimize the number of potential words with each guess. The Game can be considered the maximizer, trying to keep the number of potential words as high as possible. The Agent's algorithm uses the Minimax algorithm to map out the current game state and determine if a series of guess strategies will guarantee a win. It uses this information it's strategy decision for the next guess. Figure 10 below outlines the Agent's traversal strategy.

1. Start by examining the Word Based Guess Node. For each possible outcome or result from the Game do the following:
2. Determine if the resulting outcome is possible by removing words from the words list according the the result. If the words list isn't possible, delete the node. Skip to Step 5
3. Determine if the resulting outcome guaranteed a win without any more guesses. This is done by looking at the remaining words in the words list. If the words list has less words than the number of remaining guesses, no matter what the agent is guaranteed to win. Mark the node as a "win". Skip to Step 5
4. If the resulting outcome is possible, but not guaranteed a win, check the current node depth. If the depth of the current node is equal to the maximum number of guesses, return "lose" and ignore all remaining nodes. Otherwise, explore the result by running this algorithm on the resulting word list. Mark the node based on the result of running the algorithm. If the result is "lose" in either case, ignore the remaining outcomes. Because the Agent is fighting the Game (maximizer), it knows that the Game will pick an outcome that guarantees a "lose" condition. As such, if a strategy results in even one "lose" condition, the worst case scenario is the Game picking that strategy.
5. Repeat Steps 2-4 for all possible outcomes of a Word Based Guess.
6. If every possible outcome of the Word Based Guess is "win" or not possible, return "win".
7. If every possible outcome of the Word Based Guess is not "win", repeat steps 1-7 using the Letter Based Guess strategy instead.
8. If neither strategy guarantees a win, return "lose".

Figure 10: An Agent's Algorithm for Traversing Potential Game States

3.2.4 Comments On the Agent's Algorithm

The ability for an agent to map out possible Game states makes it possible to map out every possible game state from a blank game board and determine a number of interesting things. Unfortunately, in the algorithm's current state, this is a very time consuming and computationally expensive process.

Because of this, the decision was made to create the cut down algorithm outlined above. Instead of only examining two possible guess words (words and words_letters guessing strategy), an Agent would have been far more successful examining every possible guess word to determine if any of the words resulted in a guaranteed win state. Unfortunately, building a tree of that size was outside of the scope of this project. For information about why that is and for more information about the future of this project see Section 6.

3.2.5 The Agent Class

The Agent Class is comprised of the following functions that together make up the Agent's solving algorithm as seen in Figure 7.

3.2.5.1 Constructor

The Agent Class's constructor takes in up to 4 arguments and initializes the various class variables required by other Agent Functions. These 4 arguments are listed below:

1. **id (required):** An integer object that represents the Agent's ID. This is used by the Agent Coordinator (see Section 3.3) to keep track of Agents.
2. **words (required or filepath):** A dictionary object with a key for every possible guess. Can be omitted if]**filepath** is provided.
3. **filepath (required or words):** A string object containing a filepath to a .json file containing a dictionary object with a key for every possible guess. Can be omitted if **words** is provided.
4. **dna (optional, default: [0]*word_len):** An array of float objects used to modify the support of a word to calculate the word's cost. For more information see Figure 8.

The Agent constructor also creates a list titled words_letters that it uses in its Letter Based Guess strategy (outlined in Figure 7). This list is created by copying the words list and removing any words that have a duplicates of a letter (i.e SASSY or PUMPS). It then sets the initial costs for each word in both lists (See Figure 8).

3.2.5.2 _evaluate_decision_tree(head, depth, words, words_letters, possible_letters)

This private function takes in a Node object (head) which corresponds to a current game state along with that game state's words, words_letters and possible_letters lists. It then evaluates all possible game states that could spawn from the current passed in game state (head) by using the algorithm shown in Figure 7. (For more information on Node Objects see Section 3.2.6.1).

3.2.5.3 _get_guess_from_words(words)

This private function finds the word in the given list words with the highest cost and returns it.

3.2.5.4 _get_guess(words, words_letters, possible_letters)

The _get_guess private function combines the functions _evaluate_decision_tree and _get_guess_from_words by first determining the which list to pull a guess from (words or words_letters) using _evaluate_decision_tree and the Algorithm in Figure 7. It then uses _get_guess_from_words to retrieve a guess.

3.2.5.5 _rule_out_wordbased(word, possible_letters) and _rule_out_letterbased(guess_arr, result_arr, word)

Both private functions return true if the passed in word should be removed from a list based on the given conditions. Word Based removal means the word is removed if it doesn't fit the after a guess is made. For example, using THANK from Figure 1, the words TRUMP (1st letter T), SHANK (4th letter N in the wrong position) or PACKS (A not in secret word) would all be removed using the Word Based removal algorithm. Letter Based removal means the word is removed if it contains any letters from the guess. Again, using THANK as an example, Letter Based Removal will remove any word that contains T, H, A, N or K anywhere within the word.

3.2.5.6 _update_costs(words, words_letters)

This private function uses the algorithm outlined in Figure 8 to update the costs of the words in words and words_letters.

3.2.5.7 _modify_words_lists(guess, guess_arr, result, words, words_letters, possible_letters)

The private _modify_words_lists function uses _rule_out_wordbased to check every word left in both the words and words_letters lists and remove any words that do not belong. It also updates the possible_letters list by removing all the letters present in the most recent guess. Finally the function uses the _update_costs function to update the costs of the words in both words lists.

3.2.5.8 solve_game(game)

This is the most important function for any Agent. It combines the above functions to solve a puzzle by using the algorithm shown in Figure 7.

3.2.6 Other Agent Algorithms and Files

The Agent Class relies on a few other files and classes. They are listed with a brief overview below.

3.2.6.1 The Node Class

The file decision_tree.py in same directory as the Agent class file (agent.py) contains the definition of a Simple Node Class object. Each node contains a list of children, a cost and a variable to hold the path to a guaranteed agent win (if one exists). It has functions for getting and setting the cost of the node, adding and getting its children, and modifying the best path. These Nodes make up the tree structure used by the agent to map potential game states.

3.2.6.2 Constants

In the same directory as the Agent Class file (agent.py) there exists a file labeled constants.py. This file contains 3 constants used by the Agent and Agent Coordinator classes.

1. **possible_result_array:** An array of array objects, where each object contains every possible guess result.
2. **possible_letters_dict_true and possible_letters_dict_false:** A dictionary object with a key for every letter. Value of each key is either TRUE or FALSE according to constant name.
3. **number_of_game_parameters:** An integer object representing the number of possible arguments needed to create a game.

3.3 Agent Coordinator (Class)

The Agent Coordinator Class is responsible for the coordination of Agent objects through multiple generations of Genetic Evolution. It implements multiprocessing to speed up computation time when multiple agents are running and contains the ability to run multiple generations of agents. It is made up of 3 main functions, outlined below.

3.3.1 Constructor

The Agent Coordinator Class takes in up to 5 arguments which are outlined below.

1. **id (optional):** An integer object representing the Coordinator's ID.
2. **test_cases_filepath (required):** A string object representing a filepath that points to a .json file containing the words for each agent in a generation to solve.
3. **all_words_filepath (required):** A string object representing a filepath that points to a .json file containing all the possible words an agent can guess.
4. **game_parameters (required):** An array of values used to create the games being solved by each agent.

5. **num_threads (optional):** An integer object telling the Coordinator the maximum number of concurrent agents it can have running at any given time.

3.3.2 run_agent(agent_id, agent_dna, pool_results_dict)

This function creates an agent using the agent_id and agent_dna arguments and runs the agent on each of the test cases. When it has finished, it wraps the results of every puzzle into a dictionary and adds it to the pool_results_dict.

3.3.3 run_generation(generation_id, dna_array)

This function is used to run and report the results of an entire generation of agent's results. The optional generation_id represents the generation of DNA the coordinator is running. The second argument, dna_array is very important. It should be passed into the function as an array of array objects, where each array object contains the DNA for an agent (for more information about how an agent uses DNA, see Figure 7). The number of DNA arrays in the dna_array object is the number of agents that will be run.

To run a generation, the Agent Coordinator creates num_threads processes (as detailed above) and assigns each process to run an agent using run_agent. When a process is finished running an agent, the results of the agent's puzzles are stored and the process is given another agent until there are no more agents left to run. When the last agent finishes running, the results of every agent are returned to the calling process via a dictionary object.

4 Experimental Design

The original goal of this project was to develop an algorithm capable of being trained with the Genetic Algorithm to find the optimal way to assign a cost to each possible guess. In order to do this, the algorithm would need to be fed information about each guess, and then through the Genetic Evolution process, find the best way to format that information into a cost. The section below will cover the hardware used for the experimentation, the experimentation itself and the results.

4.1 Hardware

Running the Genetic algorithm can be very time consuming, as multiple generations of agents are needed in order to tune the algorithm. The time for each generation of agent's to run is directly correlated to the amount of computational power being used to run the generation. In order to reduce the time for each generation, a 16 core 32 thread AMD Ryzen CPU was used and each of the 32 threads were assigned an agent. This allowed the generations (each made up of 100 agents) to run in 3 batches instead of individually. This reduced the time for a generation from over 2 days to less than 2 hours. While not implemented due to time frame constraints, the use of Intel Xeon Phi was also considered for this project. Intel's Xeon Phi line of processors was designed with machine learning and artificial intelligence in mind. While each core is relatively weak when compared to that of a traditional CPU (like the AMD Ryzen CPU used in the project), the Xeon Phi line of processors contain huge numbers of cores and threads that can be used in computation where parallelization is important, like artificial intelligence. The Xeon Phi coprocessor that this project planned to implement contained 60 cores, each of which had 4 threads for a total of 240 threads. This would have enabled even quicker computation of a generation and allowed even more generations to be run in order to better tune the algorithm. Unfortunately, this processor was not implemented in this iteration of the project due to time constraints. It also became clear that tuning of the algorithm via Genetic Evolution did not play a significant role in the algorithm's success rate. This is covered in more detail in subsequent sections.

4.2 Experiments

This project conducted two separate experiments on the algorithm to determine whether or not it was a success. However, only one experiment was initially planned. Two experiments were run after the first experiment demonstrated that the Genetic Evolution component of the algorithm was not playing an important role in the algorithm's success. The second experiment was run in an attempt to determine if the Genetic Evolution component of the algorithm would play a bigger role in the algorithm's success if the Minimax algorithm to determine possible future game states was removed. The details of the two experiments can be found in the subsequent sections.

4.2.1 Experiment One

To determine the algorithm's ability to solve Wordle puzzles, previous puzzles from Josh Wardle's Daily Wordle were used. Because the game has been so popular, many resources already exist that document the human population's success at solving the daily puzzles. The majority of resources started collecting data around daily puzzle No. 200, so the daily puzzle (words) from puzzle 1-202 were used as training data and words 202-300 were used as to test the algorithm's success by comparing it the results of the human population. It would have been more appropriate to use a larger test set, however because there have only been (as of April 28th) 313 puzzles and the largest data set that could be found started logging human results at puzzle 202, it would be impossible to compare the algorithm to a larger test set as no test sets exist.

This project planned for Experiment One to train using the full algorithm across multiple generations. Each generation would have consisted of 100 agents that started out with random DNA and were put to solving each puzzle. When each generation finished the Genetic Algorithm was applied to the data set to select the agents with the most success, and use their DNA to create new agents. However, after the first generation, it became clear that no additional generations were needed. Each agent in generation 1 solved the exact same number of puzzles in the exact same number of guesses. This created 100 agent's with random DNA that had the exact same success rate, making it impossible to choose agents for the next generation. The results of Experiment One are discussed in greater detail in the Result's Analysis section of this report. The failure of Experiment Two prompted the creation of Experiment 2, explained below.

4.2.2 Experiment Two

Due to the failure of Experiment One, Experiment Two was created and tested to determine if the Genetic Evolution Algorithm would be more important if the Minimax algorithm was removed. For Experiment Two, the project's algorithm was modified to remove the mapping of possible future game states when picking a guess. The modified algorithm was as follows:

1. Using the Class object's master list, create a copy of the words, words_letters and possible_letters lists.
2. Choose a guess from the words list by choosing the word with the highest cost value. The algorithm for determining the cost value of a word described in Section 3.2.2.
3. Send the guess to the Game object.
4. Using the result returned from the Game object modify the two words lists and possible_letters list. This is explained in more detail in Figure 7 detailing the original algorithm.
5. If the number of words left in the words list (or the number of possible remaining guesses) is less than the total number of remaining guesses, make a word based guess by choosing the word with highest cost from the words list. Else, choose a guess from the words_letters list.
6. Repeat Steps 3, 4 and 5 (choose guess, make guess, modify lists according to the result of guess) until either the Agent runs out of guess in the game, or it correctly guesses the word.
7. Return the result of the game (”win” or ”lose”) and the number of guesses it took to reach that state.

Figure 11: Modified Agent Algorithm for Experiment Two

The modified algorithm in Figure 11 was run on the same test set of words (Wordle daily puzzles 202-300), using the same generation size of 100 agents. Unfortunately, the removal of the Minimax algorithm did not increase the importance of the Genetic Evolution algorithm. Just as in Experiment

One, each of the 100 agents run in the first generation had the exact same success rate, making it again impossible to choose agents for the next generation. The results of both experiments are discussed in the following section.

5 Results

As discussed above, two experiments were run to determine the success of this project's algorithm. Each experiment was evaluated in the same way, this is explained below.

5.1 Criteria for Success

Two metrics were used to evaluate the project's algorithm and determine it's success. These two metrics are:

1. **Average Number of Guesses:** This represents the average number of guesses taken to solve Daily Wordle puzzles 202-300. Only puzzles that are solved successfully impact this number.
2. **Percentage of Puzzles Failed:** This represents the number of puzzles failed as a percentage of the total number of attempted puzzles.

Other metrics are considered and used to compare specific puzzles, but these two metrics enabled the comparison between Human puzzle solvers and the algorithm.

5.2 Human Results

The popularity of Josh Wardlle's daily Wordle puzzles means that a wide variety of resources already exist to guage the ability of human Wordle solvers. After some research, the Twitter bot @WordleStats created by user @gooeybob was chosen as the source of human data to compare against this project's algorithm. This bot takes advantage of the sharing feature built into Wardle's daily Wordle site. After a user solves a daily puzzle, one of the options to share the results with friends and family is a simple array of emoji characters indicating the results of each guess. This can be seen in Figure 12. This representation allows users to share their success (or failure) without also spoiling the answer of the puzzle.

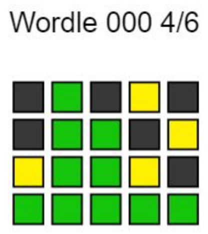

Figure 12: Example: Sharing the Results of the Puzzle in Figure 1

Twitter user @gooeybob discovered that many people were sharing these array of emoji characters on Twitter and created the Twitter bot @WordleStats to monitor all new tweets for any tweets that contain this emoji array. Using these arrays, he was able to figure out how many guesses it took to solve the puzzle for that particular user. While this sampling method has some inherent bias (for example, it only samples Twitter users or it doesn't guarantee that the same population members were used for each daily average), it was the most complete and readily available database of human Wordle results available at the time this project was created. The largest downside of using this database is that records were only kept from daily puzzle 202 onward. This is most likely because this is right around the time Wardle's daily Wordles exploded in popularity.

The daily tweets by @WordleBot containing the percentage of players who solved the puzzle in 1 to 6 (or failed) guesses were recorded into a database for comparison with the results of this project's algorithm. The results of human players on the daily Wordle puzzles 202-300 are shared below:

1. **Average Number of Guesses:** 3.995555556 Guesses
2. **Percentage of Puzzles Failed:** 0.0296969697 or 2.96969697%

If the project's algorithm is able to outperform the human population, it will be considered successful.

5.3 Experiment One Results

The experimental method for Experiment One is described in Section 4.2. It planned to run multiple generations of 100 agents with each agent using the full algorithm described in Figure 7. However, as briefly alluded to in Section 4.2, it immediately became clear that the use of the Genetic Algorithm had no effect on the success of the algorithm. Each one of the 100 agents in Generation One finished with the following:

1. **Average Number of Guesses:** 3.823529412 Guesses
2. **Percentage of Puzzles Failed:** 0.02040816327 or 2.040816327%

These results indicate that the algorithm does indeed outperform the human population and is therefore successful. Despite the Genetic Evolution component of the algorithm not being required, the algorithm still manages to solve daily Wordle puzzles 202-300 in less guesses and at a lower failure rate when compared to our human control group.

5.4 Experiment Two Results

As discussed in Section 4.2.2, Experiment Two was created to determine two things: One, to determine if the Genetic Algorithm would play a bigger role in the algorithm's success if the Minimax component was removed. And two, if mapping of potential future game states and the application of the Minimax algorithm to that map improves the performance of the algorithm as a whole. Unfortunately, as with Experiment One, it was immediately clear after one generation that the planned Genetic Evolution component has no effect on the success of the algorithm, even with the removal of the game mapping. All 100 of the agents in generation one of Experiment Two finished with the exact same average guesses and percentage of puzzles failed making it impossible to apply the genetic evolution algorithm. The results of a single agent (and by extension all agents) in Experiment Two are listed below:

1. **Average Number of Guesses:** 3.8627450980392157 Guesses
2. **Percentage of Puzzles Failed:** 0.02040816327 or 2.040816327%

While both the algorithm in Experiment One (Fig. 7) and the algorithm in Experiment Two (Fig. 11) both failed two of the 98 puzzles, Experiment One finished the 96 puzzles solved successfully in 0.06 less guesses on average. While not a large number, this demonstrates that the mapping of game states and the application of the Minimax algorithm as described in Figure 7 does have an impact on the success rate of the algorithm.

5.5 What Was Learned from the Results

As discussed in previous sections, one of the first things learned about the algorithm developed in this project was that the planned Genetic Evolution component had no effect on the algorithm's success. There are a couple of reasons why this could be. The most obvious is the attributes chosen to represent each word do not provide enough information to the algorithm to determine cost. As discussed in Figure 8, the support of 1, 2 3 and 4 letter item sets in a given word was used to provide the algorithm information about a word. This information alone was what was used to determine the cost, or how important a word is to the algorithm. This information was clearly not enough or important enough for the algorithm to properly distinguish between potential guesses resulting in any word being considered equally as important.

Another arguably more important takeaway from the results of the Experiments can be found in Experiment Two. The fact that the algorithm performed worse when the mapping of game states was removed implies that mapping the game states is important in solving this type of problem.

6 Conclusions and Future Plans

Unfortunately, a single word to describe the success of this project's algorithm would be failure. While the algorithm did manage to outperform our human control group when it came to solving a set of puzzles, it failed to due so in a significant way and furthermore, a large portion of the algorithm itself (the Genetic Evolution component) turned out to be completely unimportant. However, the algorithm was still technically more successful than the human control group at solving the set of puzzles and the project was able to demonstrate that the mapping of potential future game states and the application of the Minimax algorithm did play an important role in the algorithm's success. These findings will help influence future development of this algorithm.

With the game of Wordle being as popular as ever, this project remains extremely relevant. Potential future iterations of this project will use the findings discussed in this report to make changes to the algorithm and improve its performance. After the results of Experiment Two proved that the mapping of potential future game states positively impacted the performance of the algorithm, future iterations of this project's algorithm will place a larger focus on predicting future game states and making guesses based off of these predictions. Instead of paralyzing the execution of multiple agents, the processing power available to this project will be used to create a map of every game state. This means that the result every possible guess will be explored as opposed to just the results of the two guessing strategies. This means that for every one of the 12972 possible guesses in a Wordle game, the 243 possible outcomes will be explored, and for each outcome the result of every possible guess will be explored and so on. It is impossible to calculate the exact number of nodes, as some potential outcomes are impossible, and some words are not possible guesses after a potential outcome, it can be said with confidence that a tree of this size will be many magnitudes higher than the trees explored by the agents in this iteration of the project. After this tree is successfully calculated and stored, the next iteration of this algorithm can use this tree to determine the most optimal guess after each result from the game. Mapping out every game state possible also gives important information such as how many guesses it takes to guarantee a puzzle is solved, and what the probability of successfully solving a puzzle before running out of guesses is. It will also answer definitively what the best starter word is.

7 Notes

Project Github: https://github.com/grantmatthews18/Auto-Wordle

References

[1] Nils Aall Barricelli. Esempi numerici di processi di evoluzione. *Methodos*, 1954.

[2] Nils Aall Barricelli. Numerical testing of evolution theories. *Department of Biology, Vanderbilt University*, 1961.

[3] Amy Eastland. The best starting words for wordle. techradar.com, March 2022.

[4] John Francis. Strategies for playing moo, or "bulls and cows". *JFWAF*, 2010.

[5] Laura Hampson. 'am i just dumb?': Wordle 251 answer leaves players stumped. independent.co.uk, February 2022.

[6] Donald E. Knuth. The computer as master mind. *J. Recreational Mathematics*, 9(1), 1976.

[7] Samantha Lock. Is wordle getting harder? viral game tests players after new york times takeover. theguardian.com, February 2022.

[8] Marc McLaren. Don't freak out over yesterday's wordle — you're just doing it wrong. tomsguide.com, March 2022.

[9] John Von Neumann. On the theory of board games. *Mathematische Annalen*, 100, 1924.

[10] George C. Stockman. A minimax algorithm better than alpha-beta? *Artificial Intelligence*, 12(2):179–196, 1979.

[11] Jeff Stuckman and Guo-Qiang Zhang. Mastermind is np-complete. *Department of Electrical Engineering and Computer Science, Case Western Reserve University*, 2005.

[12] Alan M. Turing. Computing machinery and intelligence. *Mind*, LIX(236), 1950.

[13] Stanford University. Strategies of play: The minimax algorithm. cs.stanford.edu.

[14] Daniel Victor. Wordle is a love story. nytimes.org, January 2022.

[15] Darell Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4, 1994.