Combinatorial Optimization of Wordle

Intended For The

Department of Mathematics and Statistics

A Thesis

Presented to the

Faculty of

San Diego State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Statistics

---

by

Anthony Cortez

Winter 2025

SAN DIEGO STATE UNIVERSITY

The Undersigned Faculty Committee Approves the

Thesis of Anthony Cortez:

Combinatorial Optimization of Wordle

Intended For The

Department of Mathematics and Statistics

Richard Levine, Chair, Chair
Department of Mathematics and Statistics

Juanjuan Fan, co-Chair
Department of Mathematics and Statistics

Roger Whitney
Department of Computer Science

Approval Date

ABSTRACT OF THE THESIS

Combinatorial Optimization of Wordle
Intended For The
Department of Mathematics and Statistics
by
Anthony Cortez
Master of Science in Statistics
San Diego State University, 2025

This paper investigates the use of combinatorial optimization algorithms to solve the game of Wordle in hard mode. We implemented three solvers: a basic constraint-based solver, a gradient descent solver using a feedback-aligned cost function, and a simulated annealing solver that introduces probabilistic acceptance of higher-cost guesses during beginning stages of the game. Each solver was tested on 2,315 target words with multiple simulations to assess performance. The basic and gradient descent solvers achieved mean guess counts of 5.47 and 5.41, respectively, while the simulated annealing solver performed the worst at 5.70. Early green feedback was a key factor in solver efficiency, and all solvers struggled with words ending in -er or containing double letters. These results suggest that structured feedback interpretation outperforms randomized exploration.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

PAGE

ACKNOWLEDGMENTS

I would like to thank Dr. Levine for allowing me to work on this thesis. I would also like to thank Dr. Fan and Dr. Whitney for being on my committee and attending my thesis defense.

CHAPTER 1
INTRODUCTION

1.1 The Wordle Game

Wordle, a word guessing game made by Josh Wardle in 2021 has fascinated many game enthusiasts around the world. The game is so popular that the New York Times bought it from Wardle on January 31 2022 (12). Today, they offer a daily game, free for players across the world. The player tries to guess an English five-letter word (solution) in six guesses or less. After each guess, the player receives a colored tile for each letter. A green tile if the letter is in the correct place in the solution word, a yellow tile if the letter is in the solution word but in an incorrect place, and a gray tile if the letter is not in the solution word at all. Tables 1.1 and 1.2 illustrate the colors for two example game-plays.

There are two versions of Wordle. The default mode is called *easy mode*. There are no restrictions on the guesses you can make at each turn, other than the guess being a five-letter English word in the Wordle dictionary. This mode allows for more exploration, as you can gain information by using a completely different word each guess.

The second mode is called *hard mode* in which any letter that is yellow or green must be kept for all subsequent guesses. This mode limits exploration of letters for the solution word and makes the game more challenging. Tables 1.1 and 1.2 illustrate game play in both modes.

Table 1.1. Example Wordle game in easy mode: any word may be used at each guess.

Guess #	Guess Word	Feedback	Description
1	CRANE	CRANE	Found correct final letter (E); R is in the word but misplaced
2	SLOPE	SLOPE	Identified S, O, and E in correct spots
3	SCORE	SCORE	All letters correct; word solved

Every day The New York Times Wordle game is an interactive app available on tablets, phones, and personal computers, with a new word for a single game presented

Table 1.2. Example Wordle game in hard mode: green or yellow letters must be used in all subsequent guesses.

Guess #	Guess Word	Feedback	Description
1	SLATE	**S**LATE	Only S is in the solution; it must be reused in the next guess
2	SCRIP	SCRIP	All letters are correct but in the wrong positions
3	CRISP	**CRISP**	All letters correct; word solved with hard mode constraints

every day. Players now have the option to play or replay from an archive of all previous Wordle games dating back to June 19, 2021 (4). The game continues to attract a substantial player base, with tens of million people engaging daily as of early 2022 (3). With the game being very mature, more serious players have developed strategies to win the game in the fewest number of guesses. In addition, the New York Times (NYT) has developed its own algorithm, the Wordle Bot, to solve the game and provide analytics feedback to players.

Some of the basic techniques used have revolved around the starting word. Specifically, find the best starting word, which typically is aimed at knocking out vowels or popular consonants. Another technique involves using a word bank of five letter words and reducing that database based on the colored tiles you receive throughout the game.

There have been more advanced methods for trying to solve the game. One popular algorithmic approach is called the most rapid decrease (MRD) algorithm. This algorithm chooses a guess based on its potential for information gain. The overall goal is to pick a guess that will give us the expected smallest word bank from which to choose our next guess. This algorithm is computationally intensive because, at each guess, it evaluates every potential word by computing the expected remaining word bank size. This means that it must simulate guessing every possible word against every other word remaining, resulting in substantial processing time (11).

This computational complexity motivates the need for an alternative method that retains the effectiveness of MRD while drastically reducing computational overhead. This thesis investigates optimization techniques simulated annealing and gradient descent that do not require exhaustive searches at each guess, but have the potential to achieve optimal or near-optimal performance. By providing a more efficient alternative, we aim to provide a more optimal Wordle strategy. Our goal is not to find the best Wordle solver, nor beat the NYT Wordlebot. This thesis aims to show how

combinatorial optimization strategies may be applied for solving Wordle, and evaluate their relative performance.

1.2 Literature Review

Current solver strategies fall into two categories: heuristic-based methods and optimization methods. The former selects guesses based on letter distribution or vowel frequency, while the latter systematically reduces the size of the candidate word bank. Popular online blogs have shared solver methods. For example, the 10 most commonly used letters are as follows: E, A, R, O, T, L, I, S, N and C. Furthermore, the top five words to choose from as the first word are 'SOARE', 'SAREE', 'SEARE', 'STARE' and 'ROATE' (8). This information is used to improve the quality of the first guess.

Another heuristic attempt can be seen in (6), as their probability strategy identifies which letter-position combinations occur most frequently. It then selects a word that best fits that probability distribution. In addition, he introduces a popularity function, which scores a set of words based on their usage frequency. Another strategy that is introduced involves using words outside the set of possible guesses. For example, let's say we have a list of possible guesses such as *glute*, *glume*, *gloze*, and *glove*. Instead of randomly picking one word, it may be better to make a guess from outside that list to gain more information. His probability strategy solved the game with an average guess count of 4.43 over 200 simulations.

Although successful, these methods represent low hanging fruit, as more sophisticated optimization strategies can incorporate in-game feedback to make better informed guesses. For example, to expand on the MRD algorithm, consider a situation where we have a set of candidate words from which to choose. First, we select a potential guess from this set. Next, we hypothetically assume that each of the remaining words is the actual solution and then determine the feedback (colored tiles) that we would receive. We repeat this process for each candidate solution, grouping words based on the resulting tile patterns. Finally, we calculate the expected size of the word list after each guess and choose the guess that results in the smallest expected size (11).

The most basic algorithmic solver, implemented by (5), uses a naive brute force approach to solve the game. He starts with a good starting word. Based on that feedback he creates all possible letter combinations, then filters them based on the yellow constraints. Finally, he checks the resulting words against a dictionary and picks a word from the subsequent list. This process repeats until he solves the game.

In another blog, (14) creates a basic Wordle solver and incrementally improves it by incorporating heuristics and letter-frequency analysis. Initially, the solver randomly selects words from the available word bank as guesses. After each guess, the word bank

is updated based on the feedback received from the game and this process repeats until the puzzle is solved. Although no comprehensive statistics are provided, (14) notes that this basic approach solved a particular example game, with the target word "SNAKE," in five guesses. The solver is further enhanced by selecting smarter guesses that prioritize words containing the most common letters in the remaining word bank. Finally, the most rapid decrease (MRD) algorithm is implemented once the possible word bank narrows down to fewer than 500 words. With this refinement, the solver successfully completed the same game ("SNAKE") in just three guesses. The last improvement involves maximizing information gain when the remaining possible solutions drop below 50 words. Instead of restricting guesses only to possible solutions, the solver expands its selection to the entire word bank. This optimization significantly improved performance, reducing the required number of guesses from eight to three in a game with the solution word "CATCH."

There is no current literature that applies optimization methods such as gradient descent and simulated annealing to the Wordle game. However, these methods have been applied to a similar game called Mastermind. It is a code-breaking puzzle game in which one player creates a hidden sequence of colored pegs, and the other player attempts to guess this sequence within a limited number of tries. After each guess, the guesser receives feedback in the form of black and white pegs. A black peg indicates a correct color placed in the correct position, while a white peg indicates a correct color placed in an incorrect position. Unlike Wordle, Mastermind guesses are not constrained by a dictionary and instead allow any combination of colors. Table 1.3 presents an example of Mastermind game play.

Table 1.3. Consistent example of a Mastermind game.

Guess #	Guess Sequence	Feedback	Description
1	Green, Red, Yellow, Blue	○ ○ ○○	All four colors are in the code, but none are in the correct positions.
2	Red, Green, Yellow, Blue	● ○ ○○	Red is correct and in the correct position; the other three colors are correct but misplaced.
3	Red, Blue, Yellow, Green	● ● ○○	Red and Blue are in the correct positions; Yellow and Green are correct but still misplaced.
4	Red, Blue, Green, Yellow	● ● ●●	All four pegs are correct in both color and position — the code is solved.

(13) explored the application of machine learning algorithms to the game of Mastermind. She implements two strategies: one that utilizes an exponential scoring function optimized by gradient descent and another based on the Metropolis-Hastings algorithm with a cost function to evaluate the proposed codes. These approaches aim to efficiently navigate the search space for possible codes while reducing computational demands. (13) is relevant to our research as it demonstrates the use of algorithms in solving a game very similar to Wordle. Although the Metropolis-Hastings algorithm is used as a sampling method, it shares a process common to simulated annealing. There is an acceptance probability that decides the outcome of a proposal. Simulated annealing, an optimization algorithm, has a slight tweak when computing the acceptance probability by introducing a temperature parameter that gradually decreases over iterations. This similarity will be key in mapping the algorithmic methods used in Mastermind to Wordle in Section 2.

In another attempt to apply optimization algorithms to Mastermind, (2) used genetic algorithms (GAs) and simulated annealing (SA) to create a solver. Specifically, (2) treated Mastermind as a dynamic constraint optimization task, where every guess has new constraints that reduce the search space of potential solutions. They evaluated three different algorithms: a random search constrained by previous guesses, a genetic algorithm, and a simulated annealing algorithm. Although all three methods performed similarly with respect to the number of guesses required to solve the puzzle, they differed significantly in efficiency. (2) found that simulated annealing was substantially

faster by approximately two orders of magnitude. At the same time, the genetic algorithm required fewer candidate evaluations as the number of potential solutions increased. (2) demonstrates the strengths of simulated annealing in efficiently navigating large search spaces, making it a promising method for similar problems for a Wordle solver.

While our work focuses on solving Wordle through combinatorial optimization methods such as simulated annealing and gradient descent, it is important to recognize that alternative paradigms have also been proposed in the literature. Notably, (15) pursued solutions leveraging deep reinforcement learning, which frame the problem through substantially different lenses.

In Section 2, I will examine the techniques of gradient descent and simulated annealing in greater detail and discuss their application to solving Wordle. I will also introduce a general solver (GS) that utilizes permutation and mutation operations to explore the solution space. This solver will be compared against the previously discussed approaches, including heuristic strategies and the most rapid decrease algorithm. In Section 2.5, I will describe the testing process for each solver, outline the metrics used for comparison, and present the results obtained from running multiple simulations. In Section **??**, I will summarize the general findings and provide recommendations for future research.

All simulations, analyses, and document preparation for this thesis were performed on a 16-inch MacBook Pro (November 2023 model) equipped with Apple's M3 Pro chip and 36 GB of unified memory. This setup allowed for efficient parallel processing during large-scale solver evaluations and provided the necessary performance headroom for repeated Wordle simulations and LaTeX compilation without slowdown.

CHAPTER 2

METHODS

In this section, I will first describe the setup used throughout the simulations followed by a deeper dive into the most rapid decrease (MRD) algorithm. I will then outline the methodology behind the following Wordle-solving algorithms: a general solver, and two optimization approaches, gradient descent and simulated annealing. For each method, I will provide a brief background on its origin and explain how it has been adapted to the structure of Wordle. Lastly, I will perform an analysis to identify the best starting word, expanding on the heuristic approach introduced in a previous blog post by (8). This starting-word analysis provides an additional insight that complements the algorithmic strategies presented.

2.1 Wordle solver optimization problem

Wordle will be played in hard mode and the dictionary contains 12,972 words (10) and the New York Times solution bank contains a subset of 2,315 (9) words.

The baseline MRD algorithm mentioned previously takes a look at every possible guess and picks one that would give us the lowest expected number of possible solutions in the word bank,

$$E(g) = \sum_{p \in P} P(p) \cdot |S_p|. \tag{2.1}$$

Here P is the set of all possible Wordle feedback patterns, $P(p)$ is the probability of receiving a feedback pattern p, S_p is the subset of the remaining words that match the feedback pattern p, and $|S_p|$ is the size of that subset. The best guess g^* is chosen to minimize $E(g)$, namely

$$g^* = \arg\min_{g \in W} E(g) \tag{2.2}$$

where word bank W is the set of all possible five-letter words.

The word bank used for the simulations has 12,972 words. Let us run through one game to see why this method is computationally taxing. The algorithm picks a solution word. Then, during the first guess, it takes the first word in the word bank as a potential guess and compares it against every other word in the bank, simulating the feedback it would receive if each of those words were the true solution. It groups words

by their resulting feedback patterns and calculates the expected size of the remaining word bank if that guess were made. This process is repeated for every possible guess in the word bank. The guess with the lowest expected remaining word bank is selected. As the algorithm progresses, this process repeats at every turn, but with a reduced word bank. Although this approach ensures highly informed guesses, it comes at the cost of evaluating 168,298,212 pairwise comparisons. This algorithm can be seen as the motivation for our subsequent methods.

In order to create the base Wordle solver, I had to implement set operations that track and update the guesses played throughout the game. This idea draws on the methodology presented in (2), where maintaining consistency with previous feedback is critical. For example, let us say that my first guess is SOARE, and the response tiles are Green, Gray, Yellow, Yellow, and Gray. The Yellow tiles are correct letters in the wrong location, requiring a permutation move to potentially find the correct location. The Gray tiles are incorrect letters, requiring a mutation move to potentially find a correct letter if not location.

Based on this feedback, I have up to two possible permutations (from the two yellow tiles) and up to two possible mutations (from the two gray tiles). The locations of these permutations range from index one through five, meaning that some letters may remain in place. For mutations, each gray tile could potentially be replaced by any of the 26 letters in the alphabet, including the possibility of keeping the letter the same. This creates a broad set of potential next guesses, with room for both strong and weak candidates. Once the potential word is created, it is checked against the word bank to ensure that it is a valid word. Once a valid word is made, it is picked as the next guess. After a guess is made and the feedback tiles are received, the word bank is updated to remove any words that are not possible. Then the process repeats itself until the game is solved. Refer to 1 for an example.

Algorithm 1: Permutation and mutation generation based on feedback tiles.

Input: Current guess word, feedback tiles, letter bank

Output: Modified guess, applied operations, mutation map, permutation map

```
Convert current guess to list
Initialize sets: gray_indices, yellow_indices, mutations, permutations, operations
foreach tile index i in feedback do
    if feedback[i] == yellow then
        add i to yellow_indices
    if feedback[i] == gray then
        add i to gray_indices
Set step_size = length(gray_indices + yellow_indices)
if step_size > 2 then
    Random Combination of Mutations and Permutations based on feedback.
    MutationIndexes, PermutationIndexes =
     GenerateRandomMutationPermutationCounts(gray_indices,
     yellow_indices)
    NewMutations = NewLetters(MutationIndexes)
    UpdateGuess(NewMutation, MutationIndexes, PermutationIndexes)
else
    One gray: Up to one Mutation.
    Two grays: Up to two Mutations.
    Two yellows: Up to two Permutations.
    One gray and yellow: Up to One Mutation, Permutation.
    MutationIndexes, PermutationIndexes =
     GenerateRandomMutationPermutationCounts(gray_indices,
     yellow_indices)
    NewMutations = NewLetters(MutationIndexes)
    UpdateGuess(NewMutation, MutationIndexes, PermutationIndexes)

return updated guess, operations, mutations, permutations
```

2.2 Gradient descent solution

The first alteration to the algorithm was to add an additional level of intelligence. This variation involved implementing a scoring system for each guess, where a new guess would be evaluated as better or worse than the previous guess based on its consistency with the feedback received. To design this scoring system, I again drew inspiration from (13) on solving Mastermind. The core idea is to capture the overall similarity or difference between the new guess and the feedback from the previous guess, using this comparison to guide the solver towards better guesses over time. This idea is captured by a cost function that scores a potential guess based on the degree it matches with the feedback of green tiles and yellow tiles.

The process begins right after we submit our first guess and receive our feedback tiles. At this stage, we want to assign a numerical value to how effective the guess was. We do this by comparing the feedback we received to what we would expect from a perfect guess, which would return five green tiles and zero yellow. We focus on the difference in both green and yellow tiles. For example, if the guess returns only one green tile, we subtract that from five, giving us $5 - 1 = 4$, which suggests we are roughly four letters away from a perfect match. The same logic applies to yellow tiles, which indicate correct letters in the wrong position. These differences reflect how far off our guess was and are used to compute what we call the *Previous Cost*, written as C_p. The larger the cost, the further we are from the solution. The formula used to calculate this cost is

$$\text{cost} = |\Delta_{\text{yellow}}| + |\Delta_{\text{green}} + \Delta_{\text{yellow}}|. \tag{2.3}$$

After the previous cost is computed, we move on to generating our next guess by applying a permutation and/or mutation to the first guess until a valid word is produced. We keep track of which permutations and mutations were applied, along with the corresponding index positions. At this stage, we also initialize a list of five green response tiles, referred to as the *guess response*. This list will eventually reflect how our modified guess compares to the original feedback. Once the operations are selected, they are applied to the guess response. A mutation transforms a green tile into gray, while a permutation can result in a yellow tile if we are moving a green tile, or a gray tile if we are simply moving an already gray tile. This updated response list can then be compared directly to the actual feedback to determine whether the modified guess satisfies the constraints given by that feedback. This will also result in a cost called *Current Cost* written as C_c. An example of this process is shown in Table 2.1.

For this example, suppose the solution is `CRANE` and the original guess is `TRACE`.

Table 2.1. Feedback for original guess `TRACE` compared to solution `CRANE`.

Index	Letter in Guess	Feedback
1	T	Gray
2	R	Green
3	A	Green
4	C	Yellow
5	E	Green

Table 2.2. Feedback comparison and previous cost calculation.

	Actual Feedback (Wordle)	Perfect Guess
Green Tiles	3	5
Yellow Tiles	1	0
Δgreen	$5 - 3 = 2$	
Δyellow	$0 - 1 = -1$	
Cost	$\lvert -1\rvert + \lvert 2 + (-1)\rvert = 2$	

Since our feedback tile has one gray and yellow tile the algorithm can do a combination of up to one mutation and one permutation, but for simplicity we will perform exactly one of each. Lucky for us our mutation will change the letter T to N at index 1 (resulting in a gray tile at index 1), which we will permute with index four (resulting in the gray tile at index 4 and yellow tile at index 1). This will result in the word CRANE our solution. It is obvious here that it is a good guess and our Current Cost will reflect that. We update our guess response to also reflect our change. The steps are outlined in Table 2.3.

Table 2.3. Applying mutation and permutation to guess response for the word `TRACE`.

Index	Guess Response (All Green)	Updated Guess Response
1	Green	Yellow (from C, permuted here)
2	Green	Green (unchanged)
3	Green	Green (unchanged)
4	Green	Gray (permuted away)
5	Green	Green (unchanged)

Table 2.4. Feedback comparison and current cost calculation.

	Actual Feedback (Wordle)	**Updated Guess Response**
Green Tiles	3	3
Yellow Tiles	1	1
Δgreen	$3 - 3 = 0$	
Δyellow	$1 - 1 = 0$	
Cost	$\|0\| + \|0 + 0\| = 0$	

Cost function (2.3) penalizes guesses that stray from the feedback pattern, giving favor towards those words that better match the expected response. In addition, we give a larger penalty for differences in permutations. In general, the larger the score, the worse the word and the lower the score, the better. I will refer to this alteration as a gradient descent Wordle solver, since the algorithm minimizes a cost function by iteratively adjusting guesses in a positive direction. While the problem is discrete and non-differentiable, it mirrors the principle of gradient descent. As we can see from our example, we have a score of zero which indicates a very good guess.

The procedure shown in Algorithm 2 outlines the logic of the gradient descent solver. This method starts with an initial guess and iteratively applies random mutation and permutation operations to generate new guesses. Each guess is evaluated using a cost function that penalizes deviation from the observed feedback tiles. If the cost improves or remains the same, the guess is accepted and the process continues. One thing to note for the procedure is that the STATE is the letter representation of the feedback tiles stored as a string. The solution is word the solver is trying to solve for, also stored as a string.

Algorithm 2: Gradient descent algorithm for Wordle solving.

Input: FirstGuess

Output: GuessCount

Set $C_c \leftarrow 0$, $C_p \leftarrow 0$
Make first guess: `Guess(FirstGuess)`
Compute Δ_{yellow}
Compute Δ_{green}
Calculate C_p
while *STATE* $\neq$ *SOLUTION* **do**
    `NextGuess, Perm, Comb ←`
     `PermutationMutationFunction(FirstGuess)`
    Randomly choose number of mutations from feedback tiles
    Randomly choose number of permutations from feedback tiles
    Set $guess\_response \leftarrow [g_1, g_2, g_3, g_4, g_5]$
    Mutate $guess\_response$
    Permute $guess\_response$
    Compute Δ_{yellow} for $guess\_response$
    Compute Δ_{green} for $guess\_response$
    Calculate C_c
    if $C_c \leq C_p$ **then**
        Make another guess: `Guess(SecondGuess)`
        Increment $GuessCount \leftarrow GuessCount + 1$
return $GuessCount$

2.3 Simulated annealing solution

An alteration to the gradient descent algorithm is to update the cost function using a simulated annealing strategy. The name of the algorithm comes from annealing in metallurgy, a process that involves heating and gradually cooling a metal to alter its physical properties. Heating and cooling affect the movement of atoms: at high temperatures, atoms move more freely, allowing them to escape local minima; as the temperature decreases, atomic movement slows and the atoms settle into a stable, low-energy configuration.

This annealing process can be simulated in combinatorial optimization. We define an objective function that includes a temperature parameter. Initially, the temperature is high, allowing the algorithm to explore the solution space freely and accept worse solutions. As the temperature decreases, the probability of accepting worse solutions also decreases, guiding the algorithm toward convergence.

In general, simulated annealing algorithms work as follows. The temperature progressively decreases from an initial positive value to zero. At each time step, the algorithm randomly selects a solution close to the current one, measures its quality, and moves to it according to the temperature-dependent probabilities of selecting better or worse solutions, which during the search respectively remain at 1 (or positive) and decrease toward zero. (1)

Algorithm 3: Simulated annealing algorithm for Wordle solving.

Input: FirstGuess

Output: GuessCount

Set $T \leftarrow 100$, $s \leftarrow 0.99$

Initialize `TotalCost` with C_p

Draw $u \sim \text{Uniform}(0, 1)$

`NextGuess, Perm, Comb` $\leftarrow$ `PermutationMutationFunction(FirstGuess)`

while *STATE* $\neq$ *SOLUTION* **do**

  Calculate C_c

  if $C_c > C_p$ ***and*** $u \leq \exp\left(-\dfrac{TotalCost}{T \cdot s^g}\right)$ **then**

    Make second guess: `Guess(SecondGuess)`

    Increment $GuessCount \leftarrow GuessCount + 1$

  if $C_c \leq C_p$ **then**

    Make second guess: `Guess(SecondGuess)`

    Increment $GuessCount \leftarrow GuessCount + 1$

return $GuessCount$

The simulated annealing algorithm begins with an initial guess and assigns a high starting temperature, which allows for greater exploration of the solution space. At each step, the solver generates a new guess by applying a random mutation or permutation to the previous guess. This new candidate is scored using the cost function, and the solver decides whether to accept it. If the new guess has a lower cost, it is accepted outright. If it is worse, the guess may still be accepted with a probability that depends on how much worse it is and the current temperature. This probability is computed using the expression

$$P = \exp\left(\frac{-\Delta E}{T}\right), \tag{2.4}$$

where ΔE is the increase in cost and T is the current temperature. Over time, the temperature cools according to an exponential decay schedule, reducing the likelihood of accepting worse guesses. This cooling process guides the algorithm toward convergence as the search becomes more focused. The algorithm terminates once the game is solved, which naturally defines a stopping condition for this optimization task.

I will make the following adjustment. Since we are solving Wordle, the algorithm will be fully "cooled" once the game is solved. Furthermore, the probability function will be of the form

$$P = \exp\left(\frac{-\text{TotalCost}}{T \times \text{s}^g}\right), \tag{2.5}$$

where `TotalCost` is the accumulated total cost up to the current guess, T_0 is the initial temperature, s is the cooling scale factor of 0.99, and g is the guess counter (number of guesses made so far).

The temperature is initialized at a high value $T = 100$ and cools down over time according to the scale parameter s and the guess count g. The cooling schedule is exponential, given by $T \cdot s^g$, where s determines how quickly the temperature decays, and g reflects the number of guesses made so far. This setup is used to define the probability of accepting a worse solution.

At early stages (small g), the temperature is high, so the denominator is large, and even large costs yield relatively high probabilities. This encourages exploration by allowing the algorithm to escape local minima. As g increases, $T \cdot s^g$ shrinks exponentially, making the acceptance probability for worse solutions drop quickly. Over time, this transition forces the algorithm to accept only better guesses.

2.4 Best starting word

A common strategy in solving Wordle is using a starting word that returns the most green and or yellow tiles. In other words, a guess that reduces the space of future guess the most. A slight modification was done to the base algorithm and the simulations were done across the whole word bank to find the best starting word. My criteria for the "best" starting word will be the greatest reduction in the word bank after the first guess. The algorithm loops through each letter in the guess and checks it against the solution. If the letter is in the correct spot, it updates the state and removes any word from the word bank that does not have that letter in the same position. If the letter is in the solution but in the wrong spot, it records that letter and its index in a dictionary and removes any word that has that letter in the same wrong position. If the letter is not in the solution at all, it filters out any word that contains it.

2.5 Verifications

To briefly summarize my simulation environment, I will be playing Wordle in hard mode with a full word bank size of 12,972 words (including solutions) and a solution bank size of 2,315 words. Since my algorithms are not deterministic and involve random permutations and mutations, we must run multiple simulations per solution word to adequately capture the true statistics for analysis and comparison. A single run may not reflect typical behavior due to stochastic variation, especially in solvers like simulated annealing or gradient descent.

To ensure statistically reliable estimates of solver performance, we follow the methodology described in (7). Specifically, the number of simulation replications n is determined based on the desired confidence level and margin of error for estimating the population mean of the number of guesses required to solve a Wordle puzzle.

2.5.0.1 Determining the Number of Simulations

Given a confidence level $(1 - \alpha)$, the confidence interval for the mean is expressed as:

$$\bar{X} \pm z_{\alpha/2,n-1} \cdot \frac{s}{\sqrt{n}} \tag{2.6}$$

where $\bar{X}$ is the sample mean, s is the sample standard deviation, n is the number of simulation replications, and $z_{\alpha/2,n-1}$ is the critical value from the Z-distribution.

To achieve a desired margin of error E, we solve for n as

$$n = \left(\frac{z_{\alpha/2,n-1} \cdot s}{E} \right)^2 . \tag{2.7}$$

Table 2.5 shows the required number of simulations per solution word, derived from a pilot study of 1,000 games. The pilot size was selected due to the fast computation speed of the solver, allowing efficient estimation of the standard deviation for use in sample size calculation.

Table 2.5. Sample size calculation based on pilot study of 1,000 games.

Confidence Level	Margin of Error (E)	Estimated SD ($\hat{\sigma}$)	Calculated n	Simulations Used
95% ($z = 1.96$)	0.01	1.3	64,930	1,000
95% ($z = 1.96$)	0.01	1.4	75,300	1,000
95% ($z = 1.96$)	0.01	1.4	75,300	1,000

For the most rapid descent (MRD) algorithm I ran a total of 10 simulations. Tables 2.6 and 2.6 present the summary statistics and the average time it took to run

the simulations. The algorithm performed quite well and solved the game in under six guesses with an average of 5.2 and standard deviation of 2.1 guesses. However, the average time it takes to complete a game is 1.20 hours, with a standard deviation of 0.45 hours. This algorithm is highly inefficient due to one primary reason: the size of the word bank during the first guess.

During this step, the algorithm evaluates every possible word in the word bank as a potential guess, and for each of those guesses, it simulates the feedback against every other word in the word bank as if it were the true solution. This creates a double loop structure, where each guess is compared to every possible solution. If the full word bank contains 12,972 five-letter words, then the number of feedback computations required for the first guess is: 12,972 multiplied by 12,971, which is 168,298,212 comparisons. This makes MRD highly accurate but computationally infeasible for real-time or large-scale use without major refinements.

Table 2.6. Summary statistics for MRD performance (number of guesses).

Min	Q1 (25%)	Median (Q2)	Q3 (75%)	Max	Mean	Std. Dev.
2	4	5	7	8	5.2	2.1

Table 2.7. Summary statistics for MRD performance (execution time in hours).

Min	Q1 (25%)	Median (Q2)	Q3 (75%)	Max	Mean	Std. Dev.
0.42	0.96	1.38	1.51	1.70	1.20	0.45

2.6 Best First Word

Before we go over the results from the other algorithms, I will go over the finding for the best first word. Tables 2.8 and 2.9 show the best top five words based on the number of guesses and the worst five words to use. The best starting word is ‘soare’, with an average bank reduction 92.9%. Something important to note is that the top 4 words each have 3 vowels in them. Furthermore, all of the words start with ‘s’ and four out of the five end with ‘e’.

For the bottom five starting words, two observations can be seen. One is that all words have at least one repeated letter. The second is that they are more obscure

words that are not in common with many words in the words bank. Thus it makes sense that the reduction to the word bank is minimal.

Table 2.8. Top five starting words ranked by average bank reduction.

Rank	Starting Word	Avg. Bank Reduction (%)
1	soare	92.9%
2	saine	92.5%
3	saice	92.2%
4	stoae	91.5%
5	salet	91.5%

Table 2.9. Bottom five starting words ranked by average bank reduction.

Rank	Starting Word	Avg. Bank Reduction (%)
1	xylyl	0.417
2	immix	0.422
3	oxbow	0.452
4	jugum	0.467
5	kibbi	0.474

2.7 Base Wordle Solver

Moving on to the base Wordle solver, we can see the performance in Table 2.10. The average number of guesses to solve the game was 5.5 with a standard deviation of 1.4 guesses. Furthermore, the win rate was 78 percent, meaning we are winning a majority of the time. When looking at the distribution of the guess counts in Figure 2.1, we see that it is right skewed, meaning there are a few high value guess counts that are pulling up the average number of guess.

Table 2.10. Summary statistics for basic solver performance.

Algorithm	Q1 (25%)	Q3 (75%)	Mean	Std. Dev.	Win Rate
Basic Solver	4.0	6.0	5.47	1.42	80%

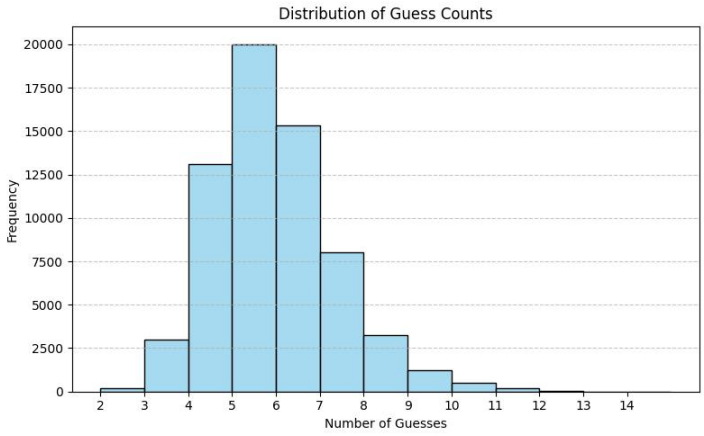

Figure 2.1. Distribution of guess counts for the basic solver.

Let's take a deeper look into the solutions that made the solver have over six guesses. Table 2.11 shows the top 10 hardest words by average guess count (ranging from 8.00 to 8.88 guesses). We observe that the average first green hit occurred at guess 2.95, while the first yellow appeared earlier at exactly 1.00. This shows that the solver typically received useful feedback right away, however it often took six to eight additional guesses to identify the correct word. The delay in green feedback contributed to longer solve times, indicating that early partial information (yellow tiles) was often insufficient to constrain the solution space.

Table 2.11. Top 10 hardest words (grouped across all games).

Word	Guess Count	First Green (Guess #)	First Yellow (Guess #)
later	8.88	2.97	1.0
water	8.62	2.88	1.0
layer	8.56	3.09	1.0
taker	8.38	3.17	1.0
maker	8.29	2.90	1.0
waver	8.12	2.91	1.0
wafer	8.04	2.96	1.0
wager	8.04	2.92	1.0
payer	8.00	2.86	1.0
taper	8.00	2.88	1.0

Most of these difficult words share structural features that extend the duration of the game. These include common suffixes such as -er, -ayer, and -ater, as seen in words like `later`, `taker`, `payer`, and `wager`. Additionally, the frequent overlap of high

frequency letters such as `r`, `a`, and `e` reduces the effectiveness of each guess under hard mode constraints, where green and yellow letters must be reused.

Although these words were difficult, all ten returned at least one yellow tile on the first guess, indicating `soare` as an effective opener. However, this early advantage was not enough to quickly converge on a solution. The solver's performance degraded due to the difficulty of eliminating many possible candidates with similar structures.

Table 2.12. Top 10 easiest words and their first green and yellow guess numbers.

Word	Guess Count	First Green (Guess #)	First Yellow (Guess #)
arose	2.00	1.0	1.0
sorry	2.45	1.0	2.0
soapy	2.58	1.0	2.0
opera	2.60	1.0	1.0
roast	2.83	1.0	1.0
ovate	2.96	1.0	1.0
adore	2.96	1.0	1.0
hoard	3.24	1.0	2.0
roach	3.26	1.0	1.0
stair	3.26	1.0	1.0

Table 2.12 presents the top 10 easiest words. One observation is that the first green tile is received on the first guess for every word. This contrasts with the hardest words, where the average first green appears closer to the third guess. This reinforces how important it is to receive green feedback early in the solve. Furthermore, as expected, many of the easiest words are structurally similar to the starting word `soare`, including `arose`, `soapy`, `opera`, and `roast`. These high-overlap cases allow the solver to converge quickly. Lastly, these words are relatively distinct, meaning there is little risk of entering a loop of guessing structurally similar alternatives.

2.8 Gradient Descent Wordle Solver

Let's take a look at the second algorithm that takes inspiration from gradient descent. Table 2.13 presents the summary statistics and Figure 2.2 presents the distribution of guess counts.

Table 2.13. Summary statistics for gradient descent algorithm performance.

Algorithm	Q1 (25%)	Q3 (75%)	Mean	Std. Dev.	Win Rate
Gradient Descent	4.0	6.0	5.41	1.40	81%

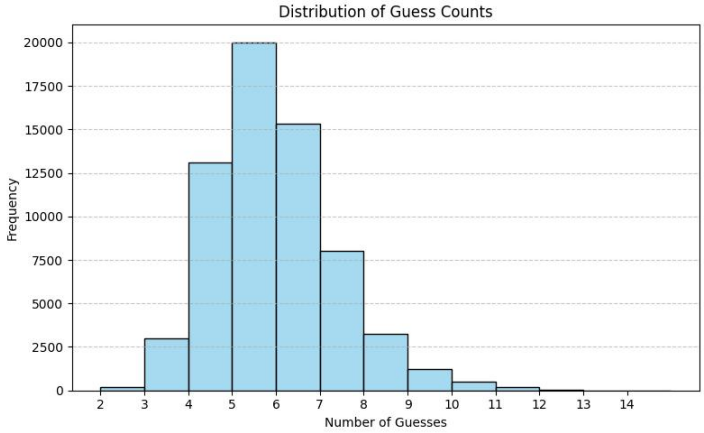

Figure 2.2. Distribution of guess counts for the gradient descent solver.

The gradient descent algorithm performed slightly better than the basic solver in terms of the average and win rate. The average increased from 5.47 to 5.41 and the win rate increased from 80 percent to 81 percent.

To compare the performance of the two Wordle solver algorithms, I used a Welch's t-test. This statistical test evaluates whether two sample means are different while allowing for unequal variances and sample sizes. Each solver was evaluated over many simulations, and the outcome of interest was the number of guesses taken to solve each game.

From Welch's t-test, we can conclude that the adjustment to the base solver and adding a score function made a difference ($p < 0.0001$). However, in context of the game that difference may not be meaningful as any difference of less than one does directly translate to solving the game quicker. We will now analyze the specific words where the gradient descent solver struggled the most, as well as those it solved quickly.

Table 2.14. Top 10 hardest words for the gradient descent solver.

Word	Guess Count	First Green (Guess #)	First Yellow (Guess #)
water	8.55	2.93	1.0
paper	8.46	3.04	1.0
maker	8.46	3.08	1.0
layer	8.40	2.86	1.0
eater	8.32	2.85	1.0
later	8.28	3.03	1.0
wager	8.22	2.89	1.0
baker	8.18	3.04	1.0
lager	8.02	2.91	1.0
waver	7.91	2.69	1.0

Table 2.14 presents the ten words that showed the greatest challenge for the gradient descent algorithm. This solver continues to struggle with the same structural features that affected the basic solver, particularly words with common suffixes and high letter overlap. Although it often finds early green or yellow tiles, the solver takes longer to converge due to the difficulty of eliminating similar candidates. While the gradient descent approach provides a slight improvement in average guess count, it still fails to find the solution when many words share overlapping structures. These results highlight the strong influence of word structure on solver performance, regardless of the underlying algorithm.

Table 2.15. Top 10 easiest words for the gradient descent solver.

Word	Guess Count	First Green (Guess #)	First Yellow (Guess #)
arose	2.00	1.0	1.0
sorry	2.60	1.0	2.0
soapy	2.69	1.0	2.0
opera	2.70	1.0	1.0
roast	2.84	1.0	1.0
ovate	3.00	1.0	1.0
adore	3.07	1.0	1.0
rouse	3.15	1.0	1.0
board	3.15	1.0	2.0
roach	3.21	1.0	1.0

Table 2.15 presents the ten words that the gradient descent algorithm solved most efficiently. The patterns observed are nearly identical to those seen with the basic

solver, with early green tiles and high overlap with the starting word leading to quick convergence. This suggests that while the scoring mechanism contributes to overall solver efficiency, its advantages are less obvious when the target word already aligns well with the initial guess.

2.9 Simulated Annealing Wordle Solver

Let's take a look now at the simulated annealing algorithm that introduces a probabilistic function to allow more exploration in the beginning of the game. Table **??** presents summary statistics and Figure **??** presents the distribution of guess counts.

The simulated annealing solver has a higher mean than the previous two algorithms of 5.88 and the lowest win rate of the three of 73 percent. The poorer performance of the simulated annealing algorithm may be due to the solver's tendency to accept higher-cost guesses during early stages, which leads to inefficient exploration and delayed convergence. While this strategy is intended to escape local minima, it also increases the risk of drifting away from good feedback. In other words, the randomness of bad guesses does not help the solver find more efficient paths. Furthermore, the randomness introduced by simulated annealing sometimes causes the solver to revisit similar guess patterns that do not narrow the word bank effectively.

Table 2.16. Summary statistics for simulated annealing algorithm performance.

Algorithm	Q1 (25%)	Q3 (75%)	Mean	Std. Dev.	Win Rate
Simulated Annealing	5.0	7.0	5.70	1.54	73%

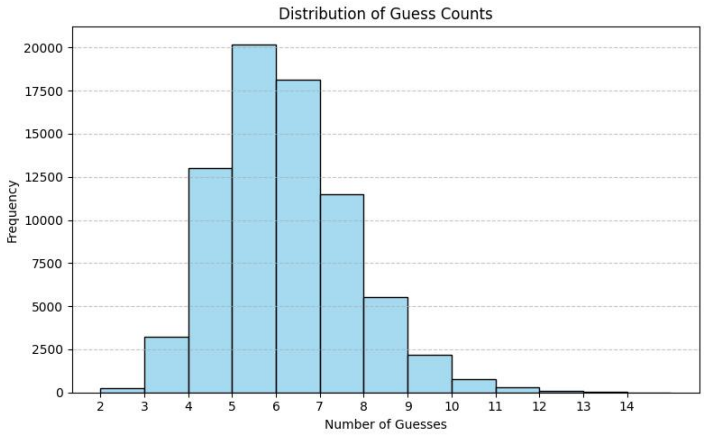

Figure 2.3. Distribution of guess counts for the simulated annealing solver.

To compare the performance of all three solvers, I used a one-way ANOVA to determine if the average number of guesses differed across algorithms. The test compares the means by partitioning the overall variance into between-group and within-group components. We want to consider the within-group, since there may be differences among that group that do not show up in the overall average. ANOVA also allows us to go beyond pairwise comparisons and evaluate solver performance in a single test. From the one-way ANOVA test, we can conclude that the average guess count across the three solvers are different ($p < 0.0001$).

We will now analyze the specific words where the simulated annealing solver struggled the most, as well as those it solved quickly; see Tables 2.17 and 2.18.

Table 2.17. Top 10 hardest words for the simulated annealing solver.

Word	Avg. Guess Count	First Green (Guess #)	First Yellow (Guess #)
holly	8.87	1.00	4.06
joker	8.71	1.00	1.00
boxer	8.61	1.00	1.00
fight	8.57	4.00	3.32
paper	8.53	2.92	1.00
jelly	8.43	3.33	1.00
lager	8.37	2.90	1.00
wager	8.32	2.98	1.00
layer	8.29	2.97	1.00
boozy	8.25	1.00	4.96

The simulated annealing solver struggles with many of the same structural patterns as the other algorithms, particularly words ending in -er. It also has difficulty with words containing repeated letters, such as ‘holly’, ‘jelly’, and ‘boozy’. In several cases, the solver identifies a green tile on the first guess but fails to converge quickly due to its tendency to accept suboptimal guesses during early exploration. For the top 10 easiest words, the patterns are consistent with those observed in the basic and gradient descent solvers. Words that closely resemble the structure of the starting word ‘soare’ continue to yield strong performance.

Table 2.18. Top 10 easiest words for the simulated annealing solver.

Word	Avg. Guess Count	First Green (Guess #)	First Yellow (Guess #)
arose	2.00	1.00	1.00
sorry	2.44	1.00	2.00
soapy	2.70	1.00	2.00
opera	2.77	1.00	1.00
roast	2.90	1.00	1.00
ovate	3.00	1.00	1.00
hoard	3.00	1.00	2.00
board	3.17	1.00	2.22
adore	3.21	1.00	1.00
roach	3.27	1.00	1.00

2.10 Simulated Annealing Usage and Effectiveness

To evaluate the role of simulated annealing (SA) in solver performance, we conducted a total of 75,300 experiments. SA was triggered in approximately 4% of all Wordle turns. Games were divided into two categories based on whether SA was utilized. Their respective performance metrics are summarized below:

Table 2.19. Performance comparison between games with and without simulated annealing.

SA Usage	Total Games	Avg. Guesses	Win Rate
No SA	58,996	5.38	81.1%
SA Used	16,304	6.85	44.3%

The difference in outcomes is statistically meaningful. A 95% confidence interval for the SA win rate is estimated as:

$$\mathrm{CI}_{95\%} = [43.6\%, 45.1\%] \tag{2.8}$$

This narrow confidence band validates the sufficiency of the 16,304 SA-activated games for inference.

Despite high acceptance probabilities, SA was rarely used. The underlying reason lies in the solver's strong performance: in most cases, proposed guesses did not worsen cost, hence bypassing the annealing condition. To support this, we analyzed the cost of guesses per turn, stratified by SA usage.

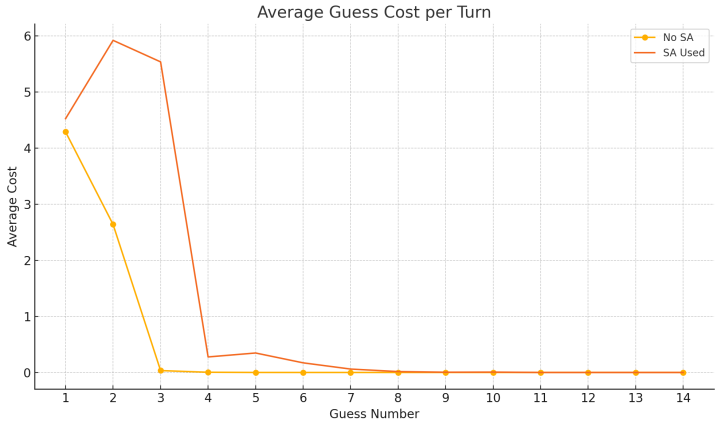

Figure 2.4. Average cost per guess turn across games with and without simulated annealing.

As shown in Figure 2.4, average costs in SA-activated games tend to rise earlier and remain elevated longer compared to games without SA, where the average cost steadily decreases. This behavior supports the hypothesis that simulated annealing is primarily triggered in scenarios where early feedback is less helpful or even misleading.

Additionally, Table 2.20 and Table 2.21 demonstrate that the patterns of word difficulty and ease in SA games are consistent with those observed in the other two solvers.

Table 2.20. Top 10 hardest words for simulated annealing games.

Word	Avg. Guess Count	First Green (Guess #)	First Yellow (Guess #)
holly	8.87	1.00	4.06
joker	8.71	1.00	1.00
boxer	8.61	1.00	1.00
fight	8.57	4.00	3.32
paper	8.53	2.92	1.00
jelly	8.43	3.33	1.00
lager	8.37	2.90	1.00
wager	8.32	2.98	1.00
layer	8.29	2.97	1.00
boozy	8.25	1.00	4.96

Table 2.21. Top 10 easiest words for simulated annealing games.

Word	Avg. Guess Count	First Green (Guess #)	First Yellow (Guess #)
arose	2.00	1.00	1.00
sorry	2.44	1.00	2.00
soapy	2.70	1.00	2.00
opera	2.77	1.00	1.00
roast	2.90	1.00	1.00
ovate	3.00	1.00	1.00
hoard	3.00	1.00	2.00
board	3.17	1.00	2.22
adore	3.21	1.00	1.00
roach	3.27	1.00	1.00

Across all three solvers, we observe consistent performance patterns due to word structure and the solvers' ability to utilize early feedback. The basic solver, while straightforward, is efficient, with a mean guess count of 5.47 and a solid win rate of 80 percent. Its strength lies in consistently applying letter elimination based on hard mode feedback, without over complicating the decision process. The gradient descent solver improves slightly on this by introducing a cost function that guides guess selection, resulting in a slightly lower average guess count and earlier first green hits. It performs particularly well when feedback is clear and the solution space shrinks quickly.

In contrast, the simulated annealing solver shows the weakest overall performance, with the highest average guess count of 5.70 and the lowest win rate of 73 percent. While the algorithm is designed to escape local minima by occasionally accepting higher-cost guesses, this leads to unnecessary exploration and slower convergence. This is especially noticeable in words with repeated letters. Despite these differences, all three solvers tend to excel on words that closely align with the initial guess soare, and struggle with -er endings. Overall, the results highlight a tradeoff between algorithmic complexity and solver efficiency, with the basic and gradient descent solvers offering better performance under hard mode constraints.

CHAPTER 3

Conclusion

This paper explored the use of algorithmic optimization techniques to solve the game of Wordle under hard mode constraints. Three solvers were implemented and evaluated: a basic constraint-based solver, a gradient descent-based solver, and a simulated annealing-based solver. Each solver was tested across over 2,315 solution words with multiple simulations per word to capture performance metrics.

The basic solver achieved good results with a mean guess count of 5.47 and a win rate of 80 percent. The gradient descent solver improved slightly with a mean of 5.41 and win rate of 81 percent by incorporating a cost function that better incorporated in-game feedback. The simulated annealing solver performed the worst, with a mean of 5.70 and win rate of 73 percent. While its probabilistic approach was intended to conduct more exploration early on, it often led to inefficient paths and slower convergence, specifically on words with repeated letters or ambiguous suffixes.

Across all solvers, early green feedback was the strongest predictor of low guess counts. Words that overlapped structurally with the starting guess soare were consistently solved more efficiently. Difficult words often contained common suffixes such as -er or repeated characters, both of which limited the effectiveness of the solvers.

These results highlight the value of combining smart initial guesses with structured feedback. While gradient descent showed promise, further tuning of cost functions or hybridization with heuristics may yield better results. Future work may explore adaptive cooling schedules for simulated annealing or reinforcement learning for solution space traversal and strategy.

BIBLIOGRAPHY

[1] E. H. Aarts and J. Korst, *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*, John Wiley & Sons, Chichester, England, 1989.

[2] J. L. Bernier, C. I. Herráiz, J. J. Merelo, S. Olmeda, and A. Prieto, *Solving master mind using gas and simulated annealing: A case of dynamic constraint optimization*, in Parallel Problem Solving from Nature — PPSN IV, H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, eds., vol. 1141 of Lecture Notes in Computer Science, Springer, 1996, pp. 553–562, https://doi.org/10.1007/3-540-61723-X_1019, https://doi.org/10.1007/3-540-61723-X_1019.

[3] CNN, *Inside the world of wordle at the new york times*, 2024, https://www.cnn.com/2024/03/14/us/wordle-new-york-times-cec/index.html. Accessed: 2025-05-06.

[4] N. Y. T. Games, *You can now play over 1,000 past wordle puzzles*, 2024, https://techcrunch.com/2024/05/07/nyt-games-wordle-archive-access-1000-past-puzzles/. Accessed: 2025-04-28.

[5] A. Holtz, *Wordle: Solve wordle with r*, February 2022, https://www.r-bloggers.com/2022/02/wordle-solve-wordle-with-r/. Accessed: 2025-05-06.

[6] W. Karsten, *Play & analyse wordle games*, April 2022, https://www.r-bloggers.com/2022/04/play-analyse-wordle-games/. Accessed: 2025-05-06.

[7] A. M. Law and W. D. Kelton, *Simulation Modeling and Analysis*, McGraw-Hill, New York, 4 ed., 2006.

[8] M. McLaren, *Best wordle start words to help you keep your streak*, 2022, https://www.tomsguide.com/news/best-wordle-start-words (accessed 2024-03-22).

[9] P. Scholtes, *Wordle word list (la version)*. https://gist.githubusercontent.com/scholtes/94f3c0303ba6a7768b47583aff36654d/raw/

d9cddf5e16140df9e14f19c2de76a0ef36fd2748/wordle-La.txt, 2022. Accessed: 2025-05-18.

[10] P. Scholtes, *Wordle word list (ta version)*. https://gist.githubusercontent.com/scholtes/94f3c0303ba6a7768b47583aff36654d/raw/d9cddf5e16140df9e14f19c2de76a0ef36fd2748/wordle-Ta.txt, 2022. Accessed: 2025-05-18.

[11] M. Short, *Wordle wisely*. https://mshort9.math.gatech.edu/papers/wordlewisely.pdf, 2022. Accessed: 2024-03-22.

[12] J. Smith, *The new york times acquires wordle*, The New York Times, (2022), https://www.nytimes.com/2022/01/31/crosswords/nyt-wordle-purchase.html.

[13] A. B. Snyder, *Mastermind by importance sampling and metropolis-hastings*, master's thesis, University of Chicago, 2004. Available at https://www.webpages.uidaho.edu/~stevel/565/projects.old/mastermind.pdf.

[14] V. Sood, *Building a wordle bot in under 100 lines of python*, 2022, https://medium.com/better-programming/building-a-wordle-bot-in-under-100-lines-of-python-9b980539defb (accessed 2024-03-22).

[15] I. C. White, J. Kenny, A. Shastry, and P. Liang, *Deep wordle: Reward and exploration strategies for solving wordle with deep reinforcement learning*. https://icwhite.github.io/website/papers/deep_woRdLe.pdf, 2022. Accessed: 2024-06-21.

[16] Wikipedia contributors, *Simulated annealing — wikipedia, the free encyclopedia*, 2024, https://en.wikipedia.org/wiki/Simulated_annealing. Accessed: 2025-06-09.